

专业人士笔记 Java® 专业人士笔记

Java® Notes for Professionals



900+ 页
专业提示和技巧

目录

关于	1
第1章：Java语言入门	2
第1.1节：创建你的第一个Java程序	2
第2章：类型转换	8
第2.1节：数值原始类型转换	8
第2.2节：基本数值提升	8
第2.3节：非数值原始类型转换	8
第2.4节：对象转换	9
第2.5节：使用instanceof测试对象是否可以被转换	9
第3章：访问器和设置器	10
第3.1节：使用设置器或访问器实现约束	10
第3.2节：为什么使用访问器和设置器？	10
第3.3节：添加访问器和设置器	11
第4章：引用数据类型	13
第4.1节：解引用	13
第4.2节：实例化引用类型	13
第5章：Java编译器 - 'javac'	14
第5.1节：“javac”命令——入门	14
第5.2节：为不同版本的Java编译	16
第6章：Java代码文档编写	18
第6.1节：从命令行生成Javadoc	18
第6.2节：类文档	18
第6.3节：方法文档	19
第6.4节：包文档	20
第6.5节：链接	20
第6.6节：文档中的代码片段	21
第6.7节：现场文档	22
第6.8节：内联代码文档	22
第7章：命令行参数处理	24
第7.1节：使用GWT ToolBase进行参数处理	24
第7.2节：手动处理参数	24
第8章：Java命令——‘java’和‘javaw’	27
第8.1节：入口点类	27
第8.2节：‘java’命令故障排除	27
第8.3节：带库依赖的Java应用程序运行	29
第8.4节：Java选项	30
第8.5节：参数中的空格和其他特殊字符	31
第8.6节：运行可执行JAR文件	33
第8.7节：通过“main”类运行Java应用程序	33
第9章：字面量	35
第9.1节：使用下划线提高可读性	35
第9.2节：十六进制、八进制和二进制字面量	35
第9.3节：布尔字面量	36
第9.4节：字符串字面量	36
第9.5节：空字面量	37
第9.6节：字面量中的转义序列	37

Contents

About	1
Chapter 1: Getting started with Java Language	2
Section 1.1: Creating Your First Java Program	2
Chapter 2: Type Conversion	8
Section 2.1: Numeric primitive casting	8
Section 2.2: Basic Numeric Promotion	8
Section 2.3: Non-numeric primitive casting	8
Section 2.4: Object casting	9
Section 2.5: Testing if an object can be cast using instanceof	9
Chapter 3: Getters and Setters	10
Section 3.1: Using a setter or getter to implement a constraint	10
Section 3.2: Why Use Getters and Setters?	10
Section 3.3: Adding Getters and Setters	11
Chapter 4: Reference Data Types	13
Section 4.1: Dereferencing	13
Section 4.2: Instantiating a reference type	13
Chapter 5: Java Compiler - 'javac'	14
Section 5.1: The ‘javac’ command - getting started	14
Section 5.2: Compiling for a different version of Java	16
Chapter 6: Documenting Java Code	18
Section 6.1: Building Javadocs From the Command Line	18
Section 6.2: Class Documentation	18
Section 6.3: Method Documentation	19
Section 6.4: Package Documentation	20
Section 6.5: Links	20
Section 6.6: Code snippets inside documentation	21
Section 6.7: Field Documentation	22
Section 6.8: Inline Code Documentation	22
Chapter 7: Command line Argument Processing	24
Section 7.1: Argument processing using GWT ToolBase	24
Section 7.2: Processing arguments by hand	24
Chapter 8: The Java Command - 'java' and 'javaw'	27
Section 8.1: Entry point classes	27
Section 8.2: Troubleshooting the ‘java’ command	27
Section 8.3: Running a Java application with library dependencies	29
Section 8.4: Java Options	30
Section 8.5: Spaces and other special characters in arguments	31
Section 8.6: Running an executable JAR file	33
Section 8.7: Running a Java applications via a “main” class	33
Chapter 9: Literals	35
Section 9.1: Using underscore to improve readability	35
Section 9.2: Hexadecimal, Octal and Binary literals	35
Section 9.3: Boolean literals	36
Section 9.4: String literals	36
Section 9.5: The Null literal	37
Section 9.6: Escape sequences in literals	37

第9.7节：字符字面量	38
第9.8节：十进制整数字面量	38
第9.9节：浮点字面量	39
第10章：原始数据类型	42
第10.1节：char原始类型	42
第10.2节：原始类型速查表	42
第10.3节：浮点数原始类型	43
第10.4节：整数原始类型	44
第10.5节：原始类型转换	45
第10.6节：原始类型与装箱类型的内存消耗	45
第10.7节：双精度原始类型	46
第10.8节：长整型原始类型	47
第10.9节：布尔原始类型	48
第10.10节：字节原始类型	48
第10.11节：负值表示	49
第10.12节：短原始类型	50
第11章：字符串	51
第11.1节：字符串比较	51
第11.2节：改变字符串中字符的大小写	53
第11.3节：在一个字符串中查找另一个字符串	55
第11.4节：字符串池和堆存储	56
第11.5节：拆分字符串	57
第11.6节：使用分隔符连接字符串	59
第11.7节：字符串连接和StringBuilder	60
第11.8节：子字符串	61
第11.9节：平台无关的新行分隔符	62
第11.10节：字符串反转	62
第11.11节：为自定义对象添加toString()方法	63
第11.12节：移除字符串开头和结尾的空白字符	64
第11.13节：不区分大小写的开关	64
第11.14节：替换字符串的部分内容	65
第11.15节：获取字符串的长度	66
第11.16节：获取字符串中的第n个字符	66
第11.17节：统计字符串中子串或字符的出现次数	66
第12章：字符串缓冲区	68
第12.1节：StringBuffer类	68
第13章：StringBuilder	69
第13.1节：比较StringBuffer、StringBuilder、Formatter和StringJoiner	69
第13.2节：重复字符串n次	70
第14章：字符串分词器	71
第14.1节：使用空格分割的 StringTokenizer	71
第14.2节：使用逗号','分割的 StringTokenizer	71
第15章：将字符串拆分为固定长度的部分	72
第15.1节：将字符串拆分为已知长度的子字符串	72
第15.2节：将字符串拆分为可变长度的子字符串	72
第16章：日期类	73
第16.1节：将java.util.Date转换为java.sql.Date	73
第16.2节：基本日期输出	73
第16.3节：Java 8的LocalDate和LocalDateTime对象	74

Section 9.7: Character literals	38
Section 9.8: Decimal Integer literals	38
Section 9.9: Floating-point literals	39
Chapter 10: Primitive Data Types	42
Section 10.1: The char primitive	42
Section 10.2: Primitive Types Cheatsheet	42
Section 10.3: The float primitive	43
Section 10.4: The int primitive	44
Section 10.5: Converting Primitives	45
Section 10.6: Memory consumption of primitives vs. boxed primitives	45
Section 10.7: The double primitive	46
Section 10.8: The long primitive	47
Section 10.9: The boolean primitive	48
Section 10.10: The byte primitive	48
Section 10.11: Negative value representation	49
Section 10.12: The short primitive	50
Chapter 11: Strings	51
Section 11.1: Comparing Strings	51
Section 11.2: Changing the case of characters within a String	53
Section 11.3: Finding a String Within Another String	55
Section 11.4: String pool and heap storage	56
Section 11.5: Splitting Strings	57
Section 11.6: Joining Strings with a delimiter	59
Section 11.7: String concatenation and StringBuilder	60
Section 11.8: Substrings	61
Section 11.9: Platform independent new line separator	62
Section 11.10: Reversing Strings	62
Section 11.11: Adding toString() method for custom objects	63
Section 11.12: Remove Whitespace from the Beginning and End of a String	64
Section 11.13: Case insensitive switch	64
Section 11.14: Replacing parts of Strings	65
Section 11.15: Getting the length of a String	66
Section 11.16: Getting the nth character in a String	66
Section 11.17: Counting occurrences of a substring or character in a string	66
Chapter 12: StringBuffer	68
Section 12.1: String Buffer class	68
Chapter 13: StringBuilder	69
Section 13.1: Comparing StringBuffer, StringBuilder, Formatter and StringJoiner	69
Section 13.2: Repeat a String n times	70
Chapter 14: String Tokenizer	71
Section 14.1: StringTokenizer Split by space	71
Section 14.2: StringTokenizer Split by comma ','	71
Chapter 15: Splitting a string into fixed length parts	72
Section 15.1: Break a string up into substrings all of a known length	72
Section 15.2: Break a string up into substrings all of variable length	72
Chapter 16: Date Class	73
Section 16.1: Convert java.util.Date to java.sql.Date	73
Section 16.2: A basic date output	73
Section 16.3: Java 8 LocalDate and LocalDateTime objects	74

第16.4节：创建特定日期	75
第16.5节：将日期转换为特定字符串格式	75
第16.6节：本地时间	76
第16.7节：将格式化的日期字符串转换为日期对象	76
第16.8节：创建日期对象	77
第16.9节：比较日期对象	77
第16.10节：将字符串转换为日期	80
第16.11节：时区与 java.util.Date	80
第17章：日期和时间 (java.time.*)	82
第17.1节：计算两个LocalDate之间的差异	82
第17.2节：日期和时间	82
第17.3节：日期和时间的操作	82
第17.4节：瞬时点 (Instant)	82
第17.5节：日期时间API中各类的使用	83
第17.6节：日期时间格式化	85
第17.7节：简单日期操作	85
第18章：本地时间 (LocalTime)	87
第18.1节：两个本地时间之间的时间量	87
第18.2节：简介	88
第18.3节：时间修改	88
第18.4节：时区及其时间差	88
第19章：BigDecimal	90
第19.1节：比较BigDecimal	90
第19.2节：使用BigDecimal代替float	90
第19.3节：BigDecimal.valueOf()	91
第19.4节：BigDecimal的数学运算	91
第19.5节：使用值为零、一或十初始化BigDecimal	94
第19.6节：BigDecimal对象是不可变的	94
第20章：BigInteger	96
第20.1节：初始化	96
第20.2节：BigInteger数学运算示例	97
第20.3节：比较大整数	99
第20.4节：大整数的二进制逻辑运算	100
第20.5节：生成随机大整数	101
第21章：数字格式化	103
第21.1节：数字格式化	103
第22章：位操作	104
第22.1节：检查、设置、清除和切换单个位。使用long作为位掩码	104
第22.2节：java.util.BitSet类	104
第22.3节：检查一个数字是否是2的幂	105
第22.4节：有符号与无符号移位	107
第22.5节：表示2的幂	107
第22.6节：作为位片段的值的打包/解包	108
第23章：数组	109
第23.1节：创建和初始化数组	109
第23.2节：从数组创建列表	115
第23.3节：从集合创建数组	117
第23.4节：多维数组和锯齿状数组	117
第23.5节：数组索引越界异常 (ArrayIndexOutOfBoundsException)	119

Section 16.4: Creating a Specific Date	75
Section 16.5: Converting Date to a certain String format	75
Section 16.6: LocalTime	76
Section 16.7: Convert formatted string representation of date to Date object	76
Section 16.8: Creating Date objects	77
Section 16.9: Comparing Date objects	77
Section 16.10: Converting String into Date	80
Section 16.11: Time Zones and java.util.Date	80
Chapter 17: Dates and Time (java.time.*)	82
Section 17.1: Calculate Difference between 2 LocalDates	82
Section 17.2: Date and time	82
Section 17.3: Operations on dates and times	82
Section 17.4: Instant	82
Section 17.5: Usage of various classes of Date Time API	83
Section 17.6: Date Time Formatting	85
Section 17.7: Simple Date Manipulations	85
Chapter 18: LocalTime	87
Section 18.1: Amount of time between two LocalTime	87
Section 18.2: Intro	88
Section 18.3: Time Modification	88
Section 18.4: Time Zones and their time difference	88
Chapter 19: BigDecimal	90
Section 19.1: Comparing BigDecimals	90
Section 19.2: Using BigDecimal instead of float	90
Section 19.3: BigDecimal.valueOf()	91
Section 19.4: Mathematical operations with BigDecimal	91
Section 19.5: Initialization of BigDecimals with value zero, one or ten	94
Section 19.6: BigDecimal objects are immutable	94
Chapter 20: BigInteger	96
Section 20.1: Initialization	96
Section 20.2: BigInteger Mathematical Operations Examples	97
Section 20.3: Comparing BigIntegers	99
Section 20.4: Binary Logic Operations on BigInteger	100
Section 20.5: Generating random BigIntegers	101
Chapter 21: NumberFormat	103
Section 21.1: NumberFormat	103
Chapter 22: Bit Manipulation	104
Section 22.1: Checking, setting, clearing, and toggling individual bits. Using long as bit mask	104
Section 22.2: java.util.BitSet class	104
Section 22.3: Checking if a number is a power of 2	105
Section 22.4: Signed vs unsigned shift	107
Section 22.5: Expressing the power of 2	107
Section 22.6: Packing / unpacking values as bit fragments	108
Chapter 23: Arrays	109
Section 23.1: Creating and Initializing Arrays	109
Section 23.2: Creating a List from an Array	115
Section 23.3: Creating an Array from a Collection	117
Section 23.4: Multidimensional and Jagged Arrays	117
Section 23.5: ArrayIndexOutOfBoundsException	119

第23章：数组	120
第23.1节：数组协变	120
第23.2节：数组转流	121
第23.3节：数组迭代	121
第23.4节：数组转字符串	123
第23.5节：数组排序	124
第23.6节：获取数组长度	126
第23.7节：在数组中查找元素	126
第23.8节：如何更改数组的大小？	127
第23.9节：在原始类型和包装类型之间转换数组	128
第23.10节：从数组中移除元素	129
第23.11节：比较数组是否相等	130
第23.12节：复制数组	130
第23.13节：数组类型转换	131
第24章：集合	133
第24.1节：在循环中从列表中移除项目	133
第24.2节：从现有数据构建集合	135
第24.3节：声明ArrayList并添加对象	137
第24.4节：遍历集合	137
第24.5节：不可变的空集合	139
第24.6节：子集合	139
第24.7节：不可修改的集合	140
第24.8节：陷阱：并发修改异常	141
第24.9节：使用迭代器从列表中移除匹配项	141
第24.10节：合并列表	142
第24.11节：创建自己的可迭代结构以供迭代器或增强型for循环使用	142
第24.12节：集合与原始值	144
第25章：列表	146
第25.1节：排序通用列表	146
第25.2节：将整数列表转换为字符串列表	147
第25.3节：实现列表的类——优缺点	147
第25.4节：查找两个列表的公共元素	150
第25.5节：列表元素的原地替换	150
第25.6节：使列表不可修改	151
第25.7节：在列表中移动对象	151
第25.8节：创建、添加和移除ArrayList中的元素	152
第25.9节：创建列表	152
第25.10节：位置访问操作	153
第25.11节：遍历列表中的元素	155
第25.12节：从列表B中移除存在于列表A的元素	155
第26章：集合	157
第26.1节：初始化	157
第26.2节：集合基础	157
第26.3节：集合的类型和用法	158
第26.4节：从现有集合创建列表	159
第26.5节：使用集合去除重复项	159
第26.6节：声明带有值的哈希集合	160
第27章：列表与集合	161
第27.1节：列表与集合	161
第28章：映射	162
第28.1节：高效迭代映射条目	162

Section 23.6: Array Covariance	120
Section 23.7: Arrays to Stream	121
Section 23.8: Iterating over arrays	121
Section 23.9: Arrays to a String	123
Section 23.10: Sorting arrays	124
Section 23.11: Getting the Length of an Array	126
Section 23.12: Finding an element in an array	126
Section 23.13: How do you change the size of an array?	127
Section 23.14: Converting arrays between primitives and boxed types	128
Section 23.15: Remove an element from an array	129
Section 23.16: Comparing arrays for equality	130
Section 23.17: Copying arrays	130
Section 23.18: Casting Arrays	131
Chapter 24: Collections	133
Section 24.1: Removing items from a List within a loop	133
Section 24.2: Constructing collections from existing data	135
Section 24.3: Declaring an ArrayList and adding objects	137
Section 24.4: Iterating over Collections	137
Section 24.5: Immutable Empty Collections	139
Section 24.6: Sub Collections	139
Section 24.7: Unmodifiable Collection	140
Section 24.8: Pitfall: concurrent modification exceptions	141
Section 24.9: Removing matching items from Lists using Iterator	141
Section 24.10: Join lists	142
Section 24.11: Creating your own Iterable structure for use with Iterator or for-each loop	142
Section 24.12: Collections and Primitive Values	144
Chapter 25: Lists	146
Section 25.1: Sorting a generic list	146
Section 25.2: Convert a list of integers to a list of strings	147
Section 25.3: Classes implementing List - Pros and Cons	147
Section 25.4: Finding common elements between 2 lists	150
Section 25.5: In-place replacement of a List element	150
Section 25.6: Making a list unmodifiable	151
Section 25.7: Moving objects around in the list	151
Section 25.8: Creating, Adding and Removing element from an ArrayList	152
Section 25.9: Creating a List	152
Section 25.10: Positional Access Operations	153
Section 25.11: Iterating over elements in a list	155
Section 25.12: Removing elements from list B that are present in the list A	155
Chapter 26: Sets	157
Section 26.1: Initialization	157
Section 26.2: Basics of Set	157
Section 26.3: Types and Usage of Sets	158
Section 26.4: Create a list from an existing Set	159
Section 26.5: Eliminating duplicates using Set	159
Section 26.6: Declaring a HashSet with values	160
Chapter 27: List vs Set	161
Section 27.1: List vs Set	161
Chapter 28: Maps	162
Section 28.1: Iterating Map Entries Efficiently	162

第28.2节：HashMap的使用	164
第28.3节：使用Java 8中Map的默认方法	165
第28.4节：遍历Map的内容	167
第28.5节：合并、组合和构造Map	168
第28.6节：添加多个项目	169
第28.7节：创建和初始化映射	171
第28.8节：检查键是否存在	172
第28.9节：添加元素	172
第28.10节：清空映射	173
第28.11节：使用自定义对象作为键	173
第29章：LinkedHashMap	175
第29.1节：Java LinkedHashMap类	175
第30章：WeakHashMap	176
第30.1节：WeakHashMap的概念	176
第31章：SortedMap	177
第31.1节：SortedMap简介	177
第32章：TreeMap和TreeSet	178
第32.1节：简单Java类型的TreeMap	178
第32.2节：简单Java类型的TreeSet	178
第32.3节：自定义Java类型的TreeMap/TreeSet	179
第32.4节：TreeMap和TreeSet的线程安全性	180
第33章：队列和双端队列	182
第33.1节：PriorityQueue的使用	182
第33.2节：双端队列（Deque）	182
第33.3节：栈	183
第33.4节：阻塞队列	184
第33.5节：作为FIFO队列的链表	185
第33.6节：队列接口	186
第34章：双端队列接口	187
第34.1节：向双端队列添加元素	187
第34.2节：从双端队列移除元素	187
第34.3节：检索元素而不移除	187
第34.4节：遍历双端队列	187
第35章：枚举	189
第35.1节：声明和使用基本枚举	189
第35.2节：带构造函数的枚举	192
第35.3节：带抽象方法的枚举	193
第35.4节：实现接口	194
第35.5节：使用单元素枚举实现单例模式	195
第35.6节：使用方法和静态代码块	196
第35.7节：零实例枚举	196
第35.8节：作为有界类型参数的枚举	197
第35.9节：枚举的文档编制	197
第35.10节：枚举常量特定主体	198
第35.11节：获取枚举的值	199
第35.12节：枚举多态模式	200
第35.13节：枚举值的比较与包含	201
第35.14节：通过名称获取枚举常量	201
第35.15节：带属性（字段）的枚举	202

Section 28.2: Usage of HashMap	164
Section 28.3: Using Default Methods of Map from Java 8	165
Section 28.4: Iterating through the contents of a Map	167
Section 28.5: Merging, combine and composing Maps	168
Section 28.6: Add multiple items	169
Section 28.7: Creating and Initializing Maps	171
Section 28.8: Check if key exists	172
Section 28.9: Add an element	172
Section 28.10: Clear the map	173
Section 28.11: Use custom object as key	173
Chapter 29: LinkedHashMap	175
Section 29.1: Java LinkedHashMap class	175
Chapter 30: WeakHashMap	176
Section 30.1: Concepts of WeakHashMap	176
Chapter 31: SortedMap	177
Section 31.1: Introduction to sorted Map	177
Chapter 32: TreeMap and TreeSet	178
Section 32.1: TreeMap of a simple Java type	178
Section 32.2: TreeSet of a simple Java Type	178
Section 32.3: TreeMap/TreeSet of a custom Java type	179
Section 32.4: TreeMap and TreeSet Thread Safety	180
Chapter 33: Queues and Deques	182
Section 33.1: The usage of the PriorityQueue	182
Section 33.2: Deque	182
Section 33.3: Stacks	183
Section 33.4: BlockingQueue	184
Section 33.5: LinkedList as a FIFO Queue	185
Section 33.6: Queue Interface	186
Chapter 34: Dequeue Interface	187
Section 34.1: Adding Elements to Deque	187
Section 34.2: Removing Elements from Deque	187
Section 34.3: Retrieving Element without Removing	187
Section 34.4: Iterating through Deque	187
Chapter 35: Enums	189
Section 35.1: Declaring and using a basic enum	189
Section 35.2: Enums with constructors	192
Section 35.3: Enums with Abstract Methods	193
Section 35.4: Implements Interface	194
Section 35.5: Implement Singleton pattern with a single-element enum	195
Section 35.6: Using methods and static blocks	196
Section 35.7: Zero instance enum	196
Section 35.8: Enum as a bounded type parameter	197
Section 35.9: Documenting enums	197
Section 35.10: Enum constant specific body	198
Section 35.11: Getting the values of an enum	199
Section 35.12: Enum Polymorphism Pattern	200
Section 35.13: Compare and Contains for Enum values	201
Section 35.14: Get enum constant by name	201
Section 35.15: Enum with properties (fields)	202

第35.16节：将枚举转换为字符串	203
第35.17节：带有静态字段的枚举	203
第36章：枚举映射	205
第36.1节：枚举映射图书示例	205
第37章：EnumSet类	206
第37.1节：枚举集合示例	206
第38章：以数字开头的枚举	207
第38.1节：以名称开头的枚举	207
第39章：哈希表	208
第39.1节：哈希表	208
第40章：运算符	209
第40.1节：自增/自减运算符 (++/--)	209
第40.2节：条件运算符 (? :)	209
第40.3节：按位与逻辑运算符 (~, &, , ^)	211
第40.4节：字符串连接运算符 (+)	212
第40.5节：算术运算符 (+, -, *, /, %)	214
第40.6节：移位运算符 (<<, >> 和 >>>)	216
第40.7节：instanceof 运算符	217
第40.8节：赋值运算符 (=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, = 和 ^=)	218
第40.9节：条件与运算符和条件或运算符 (&& 和)	220
第40.10节：关系运算符 (<, <=, >, >=)	221
第40.11节：相等运算符 (==, !=)	222
第40.12节：Lambda运算符 (->)	224
第41章：构造函数	225
第41.1节：默认构造函数	225
第41.2节：调用父类构造函数	226
第41.3节：带参数的构造函数	227
第42章：对象类方法与构造函数	229
第42.1节：hashCode()方法	229
第42.2节：toString() 方法	231
第42.3节：equals() 方法	232
第42.4节：wait() 和 notify() 方法	234
第42.5节：getClass() 方法	236
第42.6节：clone() 方法	237
第42.7节：对象构造函数	238
第42.8节：finalize() 方法	239
第43章：注解	241
第43.1节：注解背后的理念	241
第43.2节：定义注解类型	241
第43.3节：通过反射进行运行时注解检查	243
第43.4节：内置注解	243
第43.5节：使用注解处理器的编译时处理	246
第43.6节：重复注解	250
第43.7节：继承注解	251
第43.8节：运行时获取注解值	252
第43.9节：“this”和接收者参数的注解	253
第43.10节：添加多个注释值	254
第44章：不可变类	255
第44.1节：无可变引用的示例	255

Section 35.16: Convert enum to String	203
Section 35.17: Enums with static fields	203
Chapter 36: Enum Map	205
Section 36.1: Enum Map Book Example	205
Chapter 37: EnumSet class	206
Section 37.1: Enum Set Example	206
Chapter 38: Enum starting with number	207
Section 38.1: Enum with name at beginning	207
Chapter 39: Hashtable	208
Section 39.1: Hashtable	208
Chapter 40: Operators	209
Section 40.1: The Increment/Decrement Operators (++/--)	209
Section 40.2: The Conditional Operator (? :)	209
Section 40.3: The Bitwise and Logical Operators (~, &, , ^)	211
Section 40.4: The String Concatenation Operator (+)	212
Section 40.5: The Arithmetic Operators (+, -, *, /, %)	214
Section 40.6: The Shift Operators (<<, >> and >>>)	216
Section 40.7: The Instanceof Operator	217
Section 40.8: The Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, = and ^=)	218
Section 40.9: The conditional-and and conditional-or Operators (&& and)	220
Section 40.10: The Relational Operators (<, <=, >, >=)	221
Section 40.11: The Equality Operators (==, !=)	222
Section 40.12: The Lambda operator (->)	224
Chapter 41: Constructors	225
Section 41.1: Default Constructor	225
Section 41.2: Call parent constructor	226
Section 41.3: Constructor with Arguments	227
Chapter 42: Object Class Methods and Constructor	229
Section 42.1: hashCode() method	229
Section 42.2: toString() method	231
Section 42.3: equals() method	232
Section 42.4: wait() and notify() methods	234
Section 42.5: getClass() method	236
Section 42.6: clone() method	237
Section 42.7: Object constructor	238
Section 42.8: finalize() method	239
Chapter 43: Annotations	241
Section 43.1: The idea behind Annotations	241
Section 43.2: Defining annotation types	241
Section 43.3: Runtime annotation checks via reflection	243
Section 43.4: Built-in annotations	243
Section 43.5: Compile time processing using annotation processor	246
Section 43.6: Repeating Annotations	250
Section 43.7: Inherited Annotations	251
Section 43.8: Getting Annotation values at run-time	252
Section 43.9: Annotations for 'this' and receiver parameters	253
Section 43.10: Add multiple annotation values	254
Chapter 44: Immutable Class	255
Section 44.1: Example without mutable refs	255

第44.2节：不可变性的优势是什么？	255	Section 44.2: What is the advantage of immutability?	255
第44.3节：定义不可变类的规则	255	Section 44.3: Rules to define immutable classes	255
第44.4节：有可变引用的示例	256	Section 44.4: Example with mutable refs	256
第45章：不可变对象	257	Chapter 45: Immutable Objects	257
第45.1节：使用防御性复制创建类型的不可变版本	257	Section 45.1: Creating an immutable version of a type using defensive copying	257
第45.2节：不可变类的设计方案	257	Section 45.2: The recipe for an immutable class	257
第45.3节：阻止类成为不可变的典型设计缺陷	258	Section 45.3: Typical design flaws which prevent a class from being immutable	258
第46章：可见性（控制类成员的访问）	262	Chapter 46: Visibility (controlling access to members of a class)	262
第46.1节：私有可见性	262	Section 46.1: Private Visibility	262
第46.2节：公共可见性	262	Section 46.2: Public Visibility	262
第46.3节：包可见性	263	Section 46.3: Package Visibility	263
第46.4节：受保护可见性	263	Section 46.4: Protected Visibility	263
第46.5节：类成员访问修饰符总结	264	Section 46.5: Summary of Class Member Access Modifiers	264
第46.6节：接口成员	264	Section 46.6: Interface members	264
第47章：泛型	265	Chapter 47: Generics	265
第47.1节：创建泛型类	265	Section 47.1: Creating a Generic Class	265
第47.2节：在 `T`、`? super T` 和 `? extends T` 之间的选择	267	Section 47.2: Deciding between `T`, `? super T`, and `? extends T`	267
第47.3节：菱形语法	269	Section 47.3: The Diamond	269
第47.4节：声明泛型方法	269	Section 47.4: Declaring a Generic Method	269
第47.5节：要求多个上界（“extends A & B”）	270	Section 47.5: Requiring multiple upper bounds ("extends A & B")	270
第47.6节：在运行时获取满足泛型参数的类	270	Section 47.6: Obtain class that satisfies generic parameter at runtime	270
第47.7节：泛型类和接口的优点	271	Section 47.7: Benefits of Generic class and interface	271
第47.8节：实例化泛型类型	272	Section 47.8: Instantiating a generic type	272
第47.9节：创建有界泛型类	272	Section 47.9: Creating a Bounded Generic Class	272
第47.10节：在自身声明中引用声明的泛型类型	274	Section 47.10: Referring to the declared generic type within its own declaration	274
第47.11节：将泛型参数绑定到多个类型	275	Section 47.11: Binding generic parameter to more than 1 type	275
第47.12节：使用泛型自动类型转换	276	Section 47.12: Using Generics to auto-cast	276
第47.13节：泛型中 instanceof 的使用	276	Section 47.13: Use of instanceof with Generics	276
第47.14节：实现泛型接口（或继承泛型类）的不同方式	278	Section 47.14: Different ways for implementing a Generic Interface (or extending a Generic Class)	278
第48章：类与对象	280	Chapter 48: Classes and Objects	280
第48.1节：方法重载	280	Section 48.1: Overloading Methods	280
第48.2节：解释什么是方法重载和重写	281	Section 48.2: Explaining what is method overloading and overriding	281
第48.3节：构造函数	283	Section 48.3: Constructors	283
第48.4节：使用静态初始化器初始化静态常量字段	284	Section 48.4: Initializing static final fields using a static initializer	284
第48.5节：基本对象的构造与使用	285	Section 48.5: Basic Object Construction and Use	285
第48.6节：最简单的类	287	Section 48.6: Simplest Possible Class	287
第48.7节：对象成员与静态成员	287	Section 48.7: Object Member vs Static Member	287
第49章：局部内部类	289	Chapter 49: Local Inner Class	289
第49.1节：局部内部类	289	Section 49.1: Local Inner Class	289
第50章：嵌套类与内部类	290	Chapter 50: Nested and Inner Classes	290
第50.1节：使用嵌套类的简单栈	290	Section 50.1: A Simple Stack Using a Nested Class	290
第50.2节：静态嵌套类与非静态嵌套类	290	Section 50.2: Static vs Non Static Nested Classes	290
第50.3节：内部类的访问修饰符	292	Section 50.3: Access Modifiers for Inner Classes	292
第50.4节：匿名内部类	293	Section 50.4: Anonymous Inner Classes	293
第50.5节：从外部创建非静态内部类的实例	294	Section 50.5: Create instance of non-static inner class from outside	294
第50.6节：方法局部内部类	295	Section 50.6: Method Local Inner Classes	295
第50.7节：从非静态内部类访问外部类	295	Section 50.7: Accessing the outer class from a non-static inner class	295
第51章：java.util.Objects类	297	Chapter 51: The java.util.Objects Class	297
第51.1节：对象空值检查的基本用法	297	Section 51.1: Basic use for object null check	297

第51.2节：在流API中使用Objects.nonNull()方法引用	297
第52章：默认方法	298
第52.1节：默认方法的基本用法	298
第52.2节：从实现类访问被重写的默认方法	298
第52.3节：为什么使用默认方法？	299
第52.4节：在默认方法中访问其他接口方法	299
第52.5节：默认方法多重继承冲突	300
第52.6节：类、抽象类和接口方法的优先级	301
第53章：包	303
第53.1节：使用包创建同名类	303
第53.2节：使用包保护范围	303
第54章：继承	305
第54.1节：继承	305
第54.2节：抽象类	306
第54.3节：使用“final”限制继承和重写	308
第54.4节：里氏替换原则	309
第54.5节：抽象类和接口的使用：“是一个”关系与“拥有”能力	310
第54.6节：静态继承	313
第54.7节：面向接口编程	314
第54.8节：继承中的重写	316
第54.9节：变量遮蔽	317
第54.10节：对象引用的收窄与扩展	317
第54.11节：继承与静态方法	318
第55章：引用类型	320
第55.1节：不同的引用类型	320
第56章：控制台输入输出	322
第56.1节：从控制台读取用户输入	322
第56.2节：在控制台对齐字符串	323
第56.3节：实现基本的命令行行为	324
第57章：流	326
第57.1节：使用流	326
第57.2节：消费流	328
第57.3节：创建频率映射	330
第57.4节：无限流	330
第57.5节：将流的元素收集到集合中	331
第57.6节：使用流实现数学函数	334
第57.7节：使用flatMap()扁平化流	334
第57.8节：并行流	335
第57.9节：创建流	336
第57.10节：查找数值流的统计信息	337
第57.11节：将迭代器转换为流	337
第57.12节：使用IntStream遍历索引	337
第57.13节：合并流	338
第57.14节：使用流进行归约	338
第57.15节：使用Map.Entry流在映射后保留初始值	341
第57.16节：IntStream转字符串	341
第57.17节：查找第一个匹配谓词的元素	341
第57.18节：使用流和方法引用编写自文档化流程	342
第57.19节：将Optional流转换为值流	343
第57.20节：获取流的一个切片	343

Section 51.2: Objects.nonNull() method reference use in stream api	297
Chapter 52: Default Methods	298
Section 52.1: Basic usage of default methods	298
Section 52.2: Accessing overridden default methods from implementing class	298
Section 52.3: Why use Default Methods?	299
Section 52.4: Accessing other interface methods within default method	299
Section 52.5: Default method multiple inheritance collision	300
Section 52.6: Class, Abstract class and Interface method precedence	301
Chapter 53: Packages	303
Section 53.1: Using Packages to create classes with the same name	303
Section 53.2: Using Package Protected Scope	303
Chapter 54: Inheritance	305
Section 54.1: Inheritance	305
Section 54.2: Abstract Classes	306
Section 54.3: Using 'final' to restrict inheritance and overriding	308
Section 54.4: The Liskov Substitution Principle	309
Section 54.5: Abstract class and Interface usage: "Is-a" relation vs "Has-a" capability	310
Section 54.6: Static Inheritance	313
Section 54.7: Programming to an interface	314
Section 54.8: Overriding in Inheritance	316
Section 54.9: Variable shadowing	317
Section 54.10: Narrowing and Widening of object references	317
Section 54.11: Inheritance and Static Methods	318
Chapter 55: Reference Types	320
Section 55.1: Different Reference Types	320
Chapter 56: Console I/O	322
Section 56.1: Reading user input from the console	322
Section 56.2: Aligning strings in console	323
Section 56.3: Implementing Basic Command-Line Behavior	324
Chapter 57: Streams	326
Section 57.1: Using Streams	326
Section 57.2: Consuming Streams	328
Section 57.3: Creating a Frequency Map	330
Section 57.4: Infinite Streams	330
Section 57.5: Collect Elements of a Stream into a Collection	331
Section 57.6: Using Streams to Implement Mathematical Functions	334
Section 57.7: Flatten Streams with flatMap()	334
Section 57.8: Parallel Stream	335
Section 57.9: Creating a Stream	336
Section 57.10: Finding Statistics about Numerical Streams	337
Section 57.11: Converting an iterator to a stream	337
Section 57.12: Using IntStream to iterate over indexes	337
Section 57.13: Concatenate Streams	338
Section 57.14: Reduction with Streams	338
Section 57.15: Using Streams of Map Entry to Preserve Initial Values after Mapping	341
Section 57.16: IntStream to String	341
Section 57.17: Finding the First Element that Matches a Predicate	341
Section 57.18: Using Streams and Method References to Write Self-Documenting Processes	342
Section 57.19: Converting a Stream of Optional to a Stream of Values	343
Section 57.20: Get a Slice of a Stream	343

第57.21节：基于流创建映射	343
第57.22节：将流连接成单个字符串	344
第57.23节：使用流进行排序	345
第57.24节：基本类型流	346
第57.25节：流操作类别	346
第57.26节：将流的结果收集到数组中	347
第57.27节：使用流生成随机字符串	347
第58章：输入流和输出流	349
第58.1节：关闭流	349
第58.2节：将输入流读取为字符串	349
第58.3节：包装输入/输出流	350
第58.4节：DataInputStream示例	351
第58.5节：向输出流写入字节	351
第58.6节：将输入流复制到输出流	351
第59章：读取器和写入器	353
第59.1节：缓冲读取器（BufferedReader）	353
第59.2节：StringWriter示例	354
第60章：首选项	355
第60.1节：使用首选项	355
第60.2节：添加事件监听器	355
第60.3节：获取偏好的子节点	356
第60.4节：协调多个应用实例之间的偏好访问	357
第60.5节：导出偏好设置	357
第60.6节：导入偏好设置	358
第60.7节：移除事件监听器	359
第60.8节：获取偏好值	360
第60.9节：设置偏好值	360
第61章：集合工厂方法	361
第61.1节：List<E>工厂方法示例	361
第61.2节：Set<E>工厂方法示例	361
第61.3节：Map<K, V>工厂方法示例	361
第62章：替代集合	362
第62.1节：Guava、Apache和Eclipse集合中的多重映射（Multimap）	362
第62.2节：Apache HashBag、Guava HashMultiset 和 Eclipse HashBag	364
第62.3节：与集合的比较操作 - 创建集合	366
第63章：并发集合	371
第63.1节：线程安全的集合	371
第63.2节：向ConcurrentHashMap插入数据	371
第63.3节：并发集合	372
第64章：选择集合	374
第64.1节：Java集合流程图	374
第65章：super关键字	375
第65.1节：super关键字的使用及示例	375
第66章：序列化	378
第66.1节：Java中的基本序列化	378
第66.2节：自定义序列化	379
第66.3节：版本控制和serialVersionUID	382
第66.4节：使用Gson进行序列化	383
第66.5节：使用Jackson进行自定义JSON反序列化	384

Section 57.21: Create a Map based on a Stream	343
Section 57.22: Joining a stream to a single String	344
Section 57.23: Sort Using Stream	345
Section 57.24: Streams of Primitives	346
Section 57.25: Stream operations categories	346
Section 57.26: Collect Results of a Stream into an Array	347
Section 57.27: Generating random Strings using Streams	347
Chapter 58: InputStreams and OutputStreams	349
Section 58.1: Closing Streams	349
Section 58.2: Reading InputStream into a String	349
Section 58.3: Wrapping Input/Output Streams	350
Section 58.4: DataInputStream Example	351
Section 58.5: Writing bytes to an OutputStream	351
Section 58.6: Copying Input Stream to Output Stream	351
Chapter 59: Readers and Writers	353
Section 59.1: BufferedReader	353
Section 59.2: StringWriter Example	354
Chapter 60: Preferences	355
Section 60.1: Using preferences	355
Section 60.2: Adding event listeners	355
Section 60.3: Getting sub-nodes of Preferences	356
Section 60.4: Coordinating preferences access across multiple application instances	357
Section 60.5: Exporting preferences	357
Section 60.6: Importing preferences	358
Section 60.7: Removing event listeners	359
Section 60.8: Getting preferences values	360
Section 60.9: Setting preferences values	360
Chapter 61: Collection Factory Methods	361
Section 61.1: List<E> Factory Method Examples	361
Section 61.2: Set<E> Factory Method Examples	361
Section 61.3: Map<K, V> Factory Method Examples	361
Chapter 62: Alternative Collections	362
Section 62.1: Multimap in Guava, Apache and Eclipse Collections	362
Section 62.2: Apache HashBag, Guava HashMultiset and Eclipse HashBag	364
Section 62.3: Compare operation with collections - Create collections	366
Chapter 63: Concurrent Collections	371
Section 63.1: Thread-safe Collections	371
Section 63.2: Insertion into ConcurrentHashMap	371
Section 63.3: Concurrent Collections	372
Chapter 64: Choosing Collections	374
Section 64.1: Java Collections Flowchart	374
Chapter 65: super keyword	375
Section 65.1: Super keyword use with examples	375
Chapter 66: Serialization	378
Section 66.1: Basic Serialization in Java	378
Section 66.2: Custom Serialization	379
Section 66.3: Versioning and serialVersionUID	382
Section 66.4: Serialization with Gson	383
Section 66.5: Custom JSON Deserialization with Jackson	384

第67章：Optional（可选值）	387
第67.1节：Map（映射）	387
第67.2节：如果Optional为空则返回默认值	388
第67.3节：如果没有值则抛出异常	388
第67.4节：使用Supplier延迟提供默认值	388
第67.5节：过滤	389
第67.6节：对原始数字类型使用Optional容器	389
第67.7节：仅在存在值时运行代码	390
第67.8节：FlatMap	390
第68章：对象引用	391
第68.1节：作为方法参数的对象引用	391
第69章：异常及异常处理	394
第69.1节：使用try-catch捕获异常	394
第69.2节：try-with-resources语句	395
第69.3节：自定义异常	398
第69.4节：处理InterruptedException	400
第69.5节：try_catch块中的返回语句	401
第69.6节：介绍	402
第69.7节：Java异常层次结构——未检查异常和已检查异常	403
第69.8节：创建和读取堆栈跟踪	406
第69.9节：抛出异常	409
第69.10节：异常的高级特性	411
第69.11节：try-finally 和 try-catch-finally 语句	412
第69.12节：方法声明中的 'throws' 子句	414
第70章：日历及其子类	416
第70.1节：创建日历对象	416
第70.2节：增加/减少日历字段	416
第70.3节：日历相减	416
第70.4节：查找上午/下午	416
第71章：使用static关键字	418
第71.1节：从静态上下文引用非静态成员	418
第71.2节：使用static声明常量	418
第72章：Properties类	420
第72.1节：加载属性	420
第72.2节：将属性保存为XML	420
第72.3节：属性文件注意事项：尾随空白	421
第73章：Lambda表达式	424
第73.1节：Java Lambda简介	424
第73.2节：使用Lambda表达式对集合进行排序	427
第73.3节：方法引用	428
第73.4节：实现多个接口	430
第73.5节：Lambda - 监听器示例	430
第73.6节：使用lambda表达式的Java闭包	431
第73.7节：Lambda与内存利用	432
第73.8节：使用lambda表达式与自定义函数式接口	433
第73.9节：传统风格到Lambda风格	433
第73.10节：`return` 仅从lambda表达式返回，而不是从外部方法返回	434
第73.11节：Lambda表达式与执行环绕模式	436
第73.12节：使用Lambda表达式和谓词从列表中获取特定值	436

Chapter 67: Optional	387
Section 67.1: Map	387
Section 67.2: Return default value if Optional is empty	388
Section 67.3: Throw an exception, if there is no value	388
Section 67.4: Lazily provide a default value using a Supplier	388
Section 67.5: Filter	389
Section 67.6: Using Optional containers for primitive number types	389
Section 67.7: Run code only if there is a value present	390
Section 67.8: FlatMap	390
Chapter 68: Object References	391
Section 68.1: Object References as method parameters	391
Chapter 69: Exceptions and exception handling	394
Section 69.1: Catching an exception with try-catch	394
Section 69.2: The try-with-resources statement	395
Section 69.3: Custom Exceptions	398
Section 69.4: Handling InterruptedException	400
Section 69.5: Return statements in try catch block	401
Section 69.6: Introduction	402
Section 69.7: The Java Exception Hierarchy - Unchecked and Checked Exceptions	403
Section 69.8: Creating and reading stacktraces	406
Section 69.9: Throwing an exception	409
Section 69.10: Advanced features of Exceptions	411
Section 69.11: The try-finally and try-catch-finally statements	412
Section 69.12: The 'throws' clause in a method declaration	414
Chapter 70: Calendar and its Subclasses	416
Section 70.1: Creating Calendar objects	416
Section 70.2: Increasing / Decreasing calendar fields	416
Section 70.3: Subtracting calendars	416
Section 70.4: Finding AM/PM	416
Chapter 71: Using the static keyword	418
Section 71.1: Reference to non-static member from static context	418
Section 71.2: Using static to declare constants	418
Chapter 72: Properties Class	420
Section 72.1: Loading properties	420
Section 72.2: Saving Properties as XML	420
Section 72.3: Property files caveat: trailing whitespace	421
Chapter 73: Lambda Expressions	424
Section 73.1: Introduction to Java lambdas	424
Section 73.2: Using Lambda Expressions to Sort a Collection	427
Section 73.3: Method References	428
Section 73.4: Implementing multiple interfaces	430
Section 73.5: Lambda - Listener Example	430
Section 73.6: Java Closures with lambda expressions	431
Section 73.7: Lambdas and memory utilization	432
Section 73.8: Using lambda expression with your own functional interface	433
Section 73.9: Traditional style to Lambda style	433
Section 73.10: `return` only returns from the lambda, not the outer method	434
Section 73.11: Lambdas and Execute-around Pattern	436
Section 73.12: Using lambda expressions & predicates to get a certain value(s) from a list	436

第74章：基本控制结构	438	Chapter 74: Basic Control Structures	438
第74.1节：Switch语句	438	Section 74.1: Switch statement	438
第74.2节：do...while循环	439	Section 74.2: do...while Loop	439
第74.3节：For 循环	440	Section 74.3: For Each	440
第74.4节：Java中的Continue语句	441	Section 74.4: Continue Statement in Java	441
第74.5节：If / Else If / Else 控制结构	441	Section 74.5: If / Else If / Else Control	441
第74.6节：For 循环	441	Section 74.6: For Loops	441
第74.7节：三元运算符	442	Section 74.7: Ternary Operator	442
第74.8节：Try ... Catch ... Finally	443	Section 74.8: Try ... Catch ... Finally	443
第74.9节：跳出	443	Section 74.9: Break	443
第74.10节：While循环	444	Section 74.10: While Loops	444
第74.11节：If / Else	444	Section 74.11: If / Else	444
第74.12节：嵌套的break / continue	444	Section 74.12: Nested break / continue	444
第75章：BufferedWriter（缓冲写入器）	446	Chapter 75: BufferedWriter	446
第75.1节：向文件写入一行文本	446	Section 75.1: Write a line of text to File	446
第76章：新文件输入输出	447	Chapter 76: New File I/O	447
第76.1节：创建路径	447	Section 76.1: Creating paths	447
第76.2节：操作路径	447	Section 76.2: Manipulating paths	447
第76.3节：获取路径信息	447	Section 76.3: Retrieving information about a path	447
第76.4节：使用文件系统获取信息	448	Section 76.4: Retrieving information using the filesystem	448
第76.5节：读取文件	449	Section 76.5: Reading files	449
第76.6节：写入文件	449	Section 76.6: Writing files	449
第77章：文件输入输出	450	Chapter 77: File I/O	450
第77.1节：从java.io.File迁移到Java 7 NIO (java.nio.file.Path)	450	Section 77.1: Migrating from java.io.File to Java 7 NIO (java.nio.file.Path)	450
第77.2节：从文件读取图像	452	Section 77.2: Reading an image from a file	452
第77.3节：使用InputStream/FileOutputStream进行文件读写	452	Section 77.3: File Read/Write Using FileInputStream/FileOutputStream	452
第77.4节：将所有字节读取到byte[]中	453	Section 77.4: Reading all bytes to a byte[]	453
第77.5节：使用Channel复制文件	454	Section 77.5: Copying a file using Channel	454
第77.6节：将byte[]写入文件	454	Section 77.6: Writing a byte[] to a file	454
第77.7节：流与Writer/Reader API的比较	455	Section 77.7: Stream vs Writer/Reader API	455
第77.8节：使用Scanner读取文件	456	Section 77.8: Reading a file with a Scanner	456
第77.9节：使用InputStream和OutputStream复制文件	457	Section 77.9: Copying a file using InputStream and OutputStream	457
第77.10节：从二进制文件读取	457	Section 77.10: Reading from a binary file	457
第77.11节：使用通道和缓冲区读取文件	457	Section 77.11: Reading a file using Channel and Buffer	457
第77.12节：添加目录	458	Section 77.12: Adding Directories	458
第77.13节：阻止或重定向标准输出/错误	459	Section 77.13: Blocking or redirecting standard output / error	459
第77.14节：一次性读取整个文件	460	Section 77.14: Reading a whole file at once	460
第77.15节：锁定	460	Section 77.15: Locking	460
第77.16节：使用BufferedInputStream读取文件	460	Section 77.16: Reading a file using BufferedInputStream	460
第77.17节：遍历目录并打印其中的子目录	461	Section 77.17: Iterate over a directory printing subdirectories in it	461
第77.18节：使用通道和缓冲区写入文件	461	Section 77.18: Writing a file using Channel and Buffer	461
第77.19节：使用PrintStream写入文件	462	Section 77.19: Writing a file using PrintStream	462
第77.20节：遍历目录并按文件扩展名过滤	462	Section 77.20: Iterating over a directory and filter by file extension	462
第77.21节：访问ZIP文件内容	463	Section 77.21: Accessing the contents of a ZIP file	463
第78章：扫描器	464	Chapter 78: Scanner	464
第78.1节：完成最常见任务的一般模式	464	Section 78.1: General Pattern that does most commonly asked about tasks	464
第78.2节：使用自定义分隔符	466	Section 78.2: Using custom delimiters	466
第78.3节：使用扫描器读取系统输入	466	Section 78.3: Reading system input using Scanner	466
第78.4节：使用Scanner读取文件输入	466	Section 78.4: Reading file input using Scanner	466
第78.5节：使用Scanner将整个输入读取为字符串	467	Section 78.5: Read the entire input as a String using Scanner	467

第78章：仔细关闭Scanner	467	Section 78.6: Carefully Closing a Scanner	467
第78.7节：从命令行读取整数	468	Section 78.7: Read an int from the command line	468
第79章：接口	469	Chapter 79: Interfaces	469
第79.1节：实现多个接口	469	Section 79.1: Implementing multiple interfaces	469
第79.2节：声明和实现接口	470	Section 79.2: Declaring and Implementing an Interface	470
第79.3节：扩展接口	470	Section 79.3: Extending an interface	470
第79.4节：接口的实用性	471	Section 79.4: Usefulness of interfaces	471
第79.5节：默认方法	473	Section 79.5: Default methods	473
第79.6节：接口中的修饰符	475	Section 79.6: Modifiers in Interfaces	475
第79.7节：在泛型中使用接口	475	Section 79.7: Using Interfaces with Generics	475
第79.8节：加强有界类型参数	478	Section 79.8: Strengthen bounded type parameters	478
第79.9节：在抽象类中实现接口	478	Section 79.9: Implementing interfaces in an abstract class	478
第80章：正则表达式	480	Chapter 80: Regular Expressions	480
第80.1节：使用捕获组	480	Section 80.1: Using capture groups	480
第80.2节：通过带标志的Pattern编译使用自定义行为的正则表达式	481	Section 80.2: Using regex with custom behaviour by compiling the Pattern with flags	481
第80.3节：转义字符	481	Section 80.3: Escape Characters	481
第80.4节：不匹配给定字符串	482	Section 80.4: Not matching a given string	482
第80.5节：使用正则表达式字面量匹配	482	Section 80.5: Matching with a regex literal	482
第80.6节：匹配反斜杠	482	Section 80.6: Matching a backslash	482
第81章：Comparable和Comparator	484	Chapter 81: Comparable and Comparator	484
第81.1节：使用Comparable<T>或Comparator<T>对列表进行排序	484	Section 81.1: Sorting a List using Comparable<T> or a Comparator<T>	484
第81.2节：compareTo和compare方法	487	Section 81.2: The compareTo and compare Methods	487
第81.3节：自然排序（Comparable）与显式排序（Comparator）	488	Section 81.3: Natural (comparable) vs explicit (comparator) sorting	488
第81.4节：使用comparing方法创建Comparator	489	Section 81.4: Creating a Comparator using comparing method	489
第81.5节：排序映射条目	489	Section 81.5: Sorting Map entries	489
第82章：Java浮点运算	491	Chapter 82: Java Floating Point Operations	491
第82.1节：比较浮点数值	491	Section 82.1: Comparing floating point values	491
第82.2节：溢出和下溢	493	Section 82.2: OverFlow and UnderFlow	493
第82.3节：浮点数值的格式化	494	Section 82.3: Formatting the floating point values	494
第82.4节：严格遵守IEEE规范	494	Section 82.4: Strict Adherence to the IEEE Specification	494
第83章：货币与资金	496	Chapter 83: Currency and Money	496
第83.1节：添加自定义货币	496	Section 83.1: Add custom currency	496
第84章：对象克隆	497	Chapter 84: Object Cloning	497
第84.1节：执行深拷贝的克隆	497	Section 84.1: Cloning performing a deep copy	497
第84.2节：使用复制工厂的克隆	498	Section 84.2: Cloning using a copy factory	498
第84.3节：使用复制构造函数的克隆	498	Section 84.3: Cloning using a copy constructor	498
第84.4节：通过实现Clonable接口进行克隆	498	Section 84.4: Cloning by implementing Clonable interface	498
第84.5节：执行浅拷贝的克隆	499	Section 84.5: Cloning performing a shallow copy	499
第85章：递归	501	Chapter 85: Recursion	501
第85.1节：递归的基本思想	501	Section 85.1: The basic idea of recursion	501
第85.2节：Java中深度递归的问题	501	Section 85.2: Deep recursion is problematic in Java	501
第85.3节：递归的类型	503	Section 85.3: Types of Recursion	503
第85.4节：计算第N个斐波那契数	503	Section 85.4: Computing the Nth Fibonacci Number	503
第85.5节：StackOverflowError与递归转循环	504	Section 85.5: StackOverflowError & recursion to loop	504
第85.6节：计算一个数的第N次方	506	Section 85.6: Computing the Nth power of a number	506
第85.7节：使用递归遍历树形数据结构	506	Section 85.7: Traversing a Tree data structure with recursion	506
第85.8节：使用递归反转字符串	507	Section 85.8: Reverse a string using Recursion	507
第85.9节：计算从1到N的整数和	507	Section 85.9: Computing the sum of integers from 1 to N	507
第86章：字符串的转换	508	Chapter 86: Converting to and from Strings	508

第86章：字符串转换为其他数据类型	508
第86.1节：字符串转换为其他数据类型	508
第86.2节：与字节的转换	509
第86.3节：Base64编码/解码	509
第86.4节：将其他数据类型转换为字符串	510
第86.5节：从`InputStream`获取`String`	511
第87章：随机数生成	512
第87.1节：伪随机数	512
第87.2节：特定范围内的伪随机数	512
第87.3节：生成密码学安全的伪随机数	513
第87.4节：使用指定种子生成随机数	513
第87.5节：选择无重复的随机数	514
第87.6节：使用apache-common lang3生成随机数	515
第88章：单例模式	516
第88.1节：枚举单例	516
第88.2节：不使用枚举的单例（饿汉式初始化）	516
第88.3节：使用持有者类的线程安全懒加载 Bill Pugh单例实现	517
第88.4节：使用双重检查锁定的线程安全单例模式	517
第88.5节：扩展单例（单例继承）	518
第89章：自动装箱	521
第89.1节：int和Integer的互换使用	521
第89.2节：自动拆箱可能导致空指针异常	522
第89.3节：在if语句中使用Boolean	522
第89.4节：Integer和int可以互换使用的不同情况	522
第89.5节：自动装箱的内存和计算开销	524
第90章：Java中的二维图形	525
第90.1节：示例1：使用Java绘制和填充矩形	525
第90.2节：绘制和填充椭圆	527
第91章：JAXB	528
第91.1节：读取XML文件（解组）	528
第91.2节：写入XML文件（对象编组）	528
第91.3节：手动字段/属性XML映射配置	529
第91.4节：将XML命名空间绑定到可序列化的Java类	530
第91.5节：使用XmlAdapter生成所需的XML格式	530
第91.6节：使用XmlAdapter修剪字符串	532
第91.7节：自动字段/属性XML映射配置（@XmlAccessorType）	532
第91.8节：指定XmlAdapter实例以（重新）使用现有数据	534
第92章：类 - Java反射	537
第92.1节：Object类的getClass()方法	537
第93章：网络	538
第93.1节：使用Socket的基本客户端和服务器通信	538
第93.2节：使用UDP（数据报）的基本客户端/服务器通信	540
第93.3节：从输入流加载信任库和密钥库	541
第93.4节：套接字示例——使用简单套接字读取网页	542
第93.5节：临时禁用SSL验证（用于测试目的）	543
第93.6节：使用通道下载文件	543
第93.7节：多播	544
第94章：NIO - 网络编程	547
第94.1节：使用Selector等待事件（以OP_CONNECT为例）	547
第95章：HttpURLConnection	549

Section 86.1: Converting String to other datatypes	508
Section 86.2: Conversion to / from bytes	509
Section 86.3: Base64 Encoding / Decoding	509
Section 86.4: Converting other datatypes to String	510
Section 86.5: Getting a `String` from an `InputStream`	511
Chapter 87: Random Number Generation	512
Section 87.1: Pseudo Random Numbers	512
Section 87.2: Pseudo Random Numbers in Specific Range	512
Section 87.3: Generating cryptographically secure pseudorandom numbers	513
Section 87.4: Generating Random Numbers with a Specified Seed	513
Section 87.5: Select random numbers without duplicates	514
Section 87.6: Generating Random number using apache-common lang3	515
Chapter 88: Singletons	516
Section 88.1: Enum Singleton	516
Section 88.2: Singleton without use of Enum (eager initialization)	516
Section 88.3: Thread-safe lazy initialization using holder class Bill Pugh Singleton implementation	517
Section 88.4: Thread safe Singleton with double checked locking	517
Section 88.5: Extending singleton (singleton inheritance)	518
Chapter 89: Autoboxing	521
Section 89.1: Using int and Integer interchangeably	521
Section 89.2: Auto-unboxing may lead to NullPointerException	522
Section 89.3: Using Boolean in if statement	522
Section 89.4: Different Cases When Integer and int can be used interchangeably	522
Section 89.5: Memory and Computational Overhead of Autoboxing	524
Chapter 90: 2D Graphics in Java	525
Section 90.1: Example 1: Draw and Fill a Rectangle Using Java	525
Section 90.2: Example 2: Drawing and Filling Oval	527
Chapter 91: JAXB	528
Section 91.1: Reading an XML file (unmarshalling)	528
Section 91.2: Writing an XML file (marshalling an object)	528
Section 91.3: Manual field/property XML mapping configuration	529
Section 91.4: Binding an XML namespace to a serializable Java class	530
Section 91.5: Using XmlAdapter to generate desired xml format	530
Section 91.6: Using XmlAdapter to trim string	532
Section 91.7: Automatic field/property XML mapping configuration (@XmlAccessorType)	532
Section 91.8: Specifying a XmlAdapter instance to (re)use existing data	534
Chapter 92: Class - Java Reflection	537
Section 92.1: getClass() method of Object class	537
Chapter 93: Networking	538
Section 93.1: Basic Client and Server Communication using a Socket	538
Section 93.2: Basic Client/Server Communication using UDP (Datagram)	540
Section 93.3: Loading TrustStore and KeyStore from InputStream	541
Section 93.4: Socket example - reading a web page using a simple socket	542
Section 93.5: Temporarily disable SSL verification (for testing purposes)	543
Section 93.6: Downloading a file using Channel	543
Section 93.7: Multicasting	544
Chapter 94: NIO - Networking	547
Section 94.1: Using Selector to wait for events (example with OP_CONNECT)	547
Chapter 95: HttpURLConnection	549

第95.1节：从URL获取响应体字符串	549
第95.2节：POST数据	550
第95.3节：删除资源	550
第95.4节：检查资源是否存在	551
第96章：JAX-WS	553
第96.1节：基本认证	553
第97章：Nashorn JavaScript引擎	554
第97.1节：执行JavaScript文件	554
第97.2节：拦截脚本输出	554
第97.3节：你好，Nashorn	555
第97.4节：计算算术字符串	555
第97.5节：设置全局变量	555
第97.6节：设置和获取全局变量	556
第97.7节：Nashorn中JavaScript中Java对象的使用	556
第97.8节：从脚本实现接口	557
第98章：Java本地接口	558
第98.1节：从Java调用C++方法	558
第98.2节：从C++调用Java方法（回调）	559
第98.3节：加载本地库	561
第99章：函数式接口	563
第99.1节：按签名列出的标准Java运行时库函数式接口列表	563
第100章：流畅接口	565
第100.1节：流畅编程风格	565
第100.2节：Truth - 流畅测试框架	566
第101章：远程方法调用（RMI）	567
第101.1节：回调：在“客户端”上调用方法	567
第101.2节：带有客户端和服务器实现的简单RMI示例	571
第101.3节：客户端-服务器：在一个JVM中调用另一个JVM的方法	573
第102章：迭代器和可迭代	576
第102.1节：使用迭代器移除元素	576
第102.2节：创建你自己的可迭代对象	576
第102.3节：在for循环中使用可迭代对象	577
第102.4节：使用原始迭代器	578
第103章：反射API	579
第103.1节：动态代理	579
第103.2节：简介	580
第103.3节：使用反射的恶意Java技巧	581
第103.4节：滥用反射API修改私有和最终变量	583
第103.5节：获取和设置字段	584
第103.6节：调用构造函数	585
第103.7节：调用嵌套类的构造函数	586
第103.8节：调用方法	586
第103.9节：根据（完全限定的）名称获取类	587
第103.10节：获取枚举的常量	587
第103.11节：使用反射调用重载构造函数	588
第104章：ByteBuffer	590
第104.1节：基本用法 - 使用DirectByteBuffer	590
第104.2节：基本用法 - 创建ByteBuffer	590
第104.3节：基本用法 - 向缓冲区写入数据	591

Section 95.1: Get response body from a URL as a String	549
Section 95.2: POST data	550
Section 95.3: Delete resource	550
Section 95.4: Check if resource exists	551
Chapter 96: JAX-WS	553
Section 96.1: Basic Authentication	553
Chapter 97: Nashorn JavaScript engine	554
Section 97.1: Execute JavaScript file	554
Section 97.2: Intercept script output	554
Section 97.3: Hello Nashorn	555
Section 97.4: Evaluate Arithmetic Strings	555
Section 97.5: Set global variables	555
Section 97.6: Set and get global variables	556
Section 97.7: Usage of Java objects in JavaScript in Nashorn	556
Section 97.8: Implementing an interface from script	557
Chapter 98: Java Native Interface	558
Section 98.1: Calling C++ methods from Java	558
Section 98.2: Calling Java methods from C++ (callback)	559
Section 98.3: Loading native libraries	561
Chapter 99: Functional Interfaces	563
Section 99.1: List of standard Java Runtime Library functional interfaces by signature	563
Chapter 100: Fluent Interface	565
Section 100.1: Fluent programming style	565
Section 100.2: Truth - Fluent Testing Framework	566
Chapter 101: Remote Method Invocation (RMI)	567
Section 101.1: Callback: invoking methods on a "client"	567
Section 101.2: Simple RMI example with Client and Server implementation	571
Section 101.3: Client-Server: invoking methods in one JVM from another	573
Chapter 102: Iterator and Iterable	576
Section 102.1: Removing elements using an iterator	576
Section 102.2: Creating your own Iterable	576
Section 102.3: Using Iterable in for loop	577
Section 102.4: Using the raw iterator	578
Chapter 103: Reflection API	579
Section 103.1: Dynamic Proxies	579
Section 103.2: Introduction	580
Section 103.3: Evil Java hacks with Reflection	581
Section 103.4: Misuse of Reflection API to change private and final variables	583
Section 103.5: Getting and Setting fields	584
Section 103.6: Call constructor	585
Section 103.7: Call constructor of nested class	586
Section 103.8: Invoking a method	586
Section 103.9: Get Class given its (fully qualified) name	587
Section 103.10: Getting the Constants of an Enumeration	587
Section 103.11: Call overloaded constructors using reflection	588
Chapter 104: ByteBuffer	590
Section 104.1: Basic Usage - Using DirectByteBuffer	590
Section 104.2: Basic Usage - Creating a ByteBuffer	590
Section 104.3: Basic Usage - Write Data to the Buffer	591

第105章：小程序	592	Chapter 105: Applets	592
第105.1节：最小小程序	592	Section 105.1: Minimal Applet	592
第105.2节：创建图形用户界面	593	Section 105.2: Creating a GUI	593
第105.3节：从小程序内打开链接	593	Section 105.3: Open links from within the applet	593
第105.4节：加载图像、音频及其他资源	594	Section 105.4: Loading images, audio and other resources	594
第106章：表达式	596	Chapter 106: Expressions	596
第106.1节：运算符优先级	596	Section 106.1: Operator Precedence	596
第106.2节：表达式基础	597	Section 106.2: Expression Basics	597
第106.3节：表达式求值顺序	598	Section 106.3: Expression evaluation order	598
第106.4节：常量表达式	599	Section 106.4: Constant Expressions	599
第107章：Java中的JSON	601	Chapter 107: JSON in Java	601
第107.1节：使用Jackson对象映射器	601	Section 107.1: Using Jackson Object Mapper	601
第107.2节：JSON转对象（Gson库）	602	Section 107.2: JSON To Object (Gson Library)	602
第107.3节：JSONObject.NULL	602	Section 107.3: JSONObject.NULL	602
第107.4节：JSON构建器 - 链式方法	603	Section 107.4: JSON Builder - chaining methods	603
第107.5节：对象转JSON（Gson库）	603	Section 107.5: Object To JSON (Gson Library)	603
第107.6节：JSON迭代	603	Section 107.6: JSON Iteration	603
第107.7节：optXXX方法与getXXX方法的比较	604	Section 107.7: optXXX vs getXXX methods	604
第107.8节：从JSON中提取单个元素	604	Section 107.8: Extract single element from JSON	604
第107.9节：JSONArray 转换为 Java 列表（Gson 库）	604	Section 107.9: JSONArray to Java List (Gson Library)	604
第107.10节：将数据编码为JSON	605	Section 107.10: Encoding data as JSON	605
第107.11节：解码JSON数据	605	Section 107.11: Decoding JSON data	605
第108章：使用JAXP API解析XML	607	Chapter 108: XML Parsing using the JAXP APIs	607
第108.1节：使用StAX API解析文档	607	Section 108.1: Parsing a document using the StAX API	607
第108.2节：使用DOM API解析和导航文档	608	Section 108.2: Parsing and navigating a document using the DOM API	608
第109章：XML XPath评估	610	Chapter 109: XML XPath Evaluation	610
第109.1节：在单个XML中解析多个XPath表达式	610	Section 109.1: Parsing multiple XPath Expressions in a single XML	610
第109.2节：在XML中多次解析单个XPath表达式	610	Section 109.2: Parsing single XPath Expression multiple times in an XML	610
第109.3节：评估XML文档中的NodeList	611	Section 109.3: Evaluating a NodeList in an XML document	611
第110章：XOM - XML对象模型	612	Chapter 110: XOM - XML Object Model	612
第110.1节：读取XML文件	612	Section 110.1: Reading a XML file	612
第110.2节：写入XML文件	614	Section 110.2: Writing to a XML File	614
第111章：多态性	617	Chapter 111: Polymorphism	617
第111.1节：方法重写	617	Section 111.1: Method Overriding	617
第111.2节：方法重载	618	Section 111.2: Method Overloading	618
第111.3节：多态性及不同类型的重写	619	Section 111.3: Polymorphism and different types of overriding	619
第111.4节：虚函数	622	Section 111.4: Virtual functions	622
第111.5节：通过添加类而不修改现有代码来增加行为	623	Section 111.5: Adding behaviour by adding classes without touching existing code	623
第112章：封装	625	Chapter 112: Encapsulation	625
第112.1节：封装以保持不变式	625	Section 112.1: Encapsulation to maintain invariants	625
第112.2节：封装以减少耦合	626	Section 112.2: Encapsulation to reduce coupling	626
第113章：Java代理	627	Chapter 113: Java Agents	627
第113.1节：使用代理修改类	627	Section 113.1: Modifying classes with agents	627
第113.2节：在运行时添加代理	627	Section 113.2: Adding an agent at runtime	627
第113.3节：设置基本代理	628	Section 113.3: Setting up a basic agent	628
第114章：可变参数（Variable Argument）	629	Chapter 114: Varargs (Variable Argument)	629
第114.1节：使用可变参数（Varargs）参数	629	Section 114.1: Working with Varargs parameters	629
第114.2节：指定可变参数（varargs）参数	629	Section 114.2: Specifying a varargs parameter	629
第115章：日志记录（java.util.logging）	630	Chapter 115: Logging (java.util.logging)	630

第115.1节：高效记录复杂消息	630
第115.2节：使用默认日志记录器	631
第115.3节：日志级别	632
第116章：log4j / log4j2	634
第116.1节：使用属性文件记录到数据库	634
第116.2节：如何获取Log4j	634
第116.3节：设置属性文件	635
第116.4节：基本的log4j2.xml配置文件	636
第116.5节：如何在Java代码中使用Log4j	636
第116.6节：从log4j 1.x迁移到2.x	637
第116.7节：按级别过滤日志输出（log4j 1.x）	638
第117章：Oracle官方代码标准	639
第117.1节：命名规范	639
第117.2节：类结构	640
第117.3节：注解	641
第117.4节：导入语句	641
第117.5节：大括号	642
第117.6节：多余的括号	643
第117.7节：修饰符	643
第117.8节：缩进	644
第117.9节：字面量	644
第117.10节：包声明	644
第117.11节：Lambda表达式	644
第117.12节：Java源文件	645
第117.13节：换行语句	645
第117.14节：换行方法声明	646
第117.15节：换行表达式	646
第117.16节：空白字符	647
第117.17节：特殊字符	647
第117.18节：变量声明	648
第118章：字符编码	649
第118.1节：从UTF-8编码的文件读取文本	649
第118.2节：以UTF-8编码写入文本到文件	649
第118.3节：获取字符串的UTF-8字节表示	650
第119章：Apache Commons Lang	651
第119.1节：实现equals()方法	651
第119.2节：实现hashCode()方法	651
第119.3节：实现toString()方法	652
第120章：本地化与国际化	654
第120.1节：区域设置	654
第120.2节：使用“locale”自动格式化日期	655
第120.3节：字符串比较	655
第121章：使用Fork/Join框架的并行编程	656
第121.1节：Java中的Fork/Join任务	656
第122章：非访问修饰符	658
第122.1节：final	658
第122.2节：static	659
第122.3节：abstract	660
第122.4节：strictfp	661

Section 115.1: Logging complex messages (efficiently)	630
Section 115.2: Using the default logger	631
Section 115.3: Logging levels	632
Chapter 116: log4j / log4j2	634
Section 116.1: Properties-File to log to DB	634
Section 116.2: How to get Log4j	634
Section 116.3: Setting up property file	635
Section 116.4: Basic log4j2.xml configuration file	636
Section 116.5: How to use Log4j in Java code	636
Section 116.6: Migrating from log4j 1.x to 2.x	637
Section 116.7: Filter Logoutput by level (log4j 1.x)	638
Chapter 117: Oracle Official Code Standard	639
Section 117.1: Naming Conventions	639
Section 117.2: Class Structure	640
Section 117.3: Annotations	641
Section 117.4: Import statements	641
Section 117.5: Braces	642
Section 117.6: Redundant Parentheses	643
Section 117.7: Modifiers	643
Section 117.8: Indentation	644
Section 117.9: Literals	644
Section 117.10: Package declaration	644
Section 117.11: Lambda Expressions	644
Section 117.12: Java Source Files	645
Section 117.13: Wrapping statements	645
Section 117.14: Wrapping Method Declarations	646
Section 117.15: Wrapping Expressions	646
Section 117.16: Whitespace	647
Section 117.17: Special Characters	647
Section 117.18: Variable Declarations	648
Chapter 118: Character encoding	649
Section 118.1: Reading text from a file encoded in UTF-8	649
Section 118.2: Writing text to a file in UTF-8	649
Section 118.3: Getting byte representation of a string in UTF-8	650
Chapter 119: Apache Commons Lang	651
Section 119.1: Implement equals() method	651
Section 119.2: Implement hashCode() method	651
Section 119.3: Implement toString() method	652
Chapter 120: Localization and Internationalization	654
Section 120.1: Locale	654
Section 120.2: Automatically formatted Dates using “locale”	655
Section 120.3: String Comparison	655
Chapter 121: Parallel programming with Fork/Join framework	656
Section 121.1: Fork/Join Tasks in Java	656
Chapter 122: Non-Access Modifiers	658
Section 122.1: final	658
Section 122.2: static	659
Section 122.3: abstract	660
Section 122.4: strictfp	661

第122.5节：易失性	661	Section 122.5: volatile	661
第122.6节：同步	662	Section 122.6: synchronized	662
第122.7节：瞬态	663	Section 122.7: transient	663
第123章：进程	664	Chapter 123: Process	664
第123.1节：陷阱：Runtime.exec、Process和ProcessBuilder不理解shell语法	664	Section 123.1: Pitfall: Runtime.exec, Process and ProcessBuilder don't understand shell syntax	664
第123.2节：简单示例（Java版本低于1.5）	666	Section 123.2: Simple example (Java version < 1.5)	666
第124章：Java本地访问	667	Chapter 124: Java Native Access	667
第124.1节：JNA简介	667	Section 124.1: Introduction to JNA	667
第125章：模块	668	Chapter 125: Modules	668
第125.1节：定义基本模块	668	Section 125.1: Defining a basic module	668
第126章：并发编程（线程）	669	Chapter 126: Concurrent Programming (Threads)	669
第126.1节：Callable和Future	669	Section 126.1: Callable and Future	669
第126.2节：CountDownLatch	670	Section 126.2: CountDownLatch	670
第126.3节：基本多线程	672	Section 126.3: Basic Multithreading	672
第126.4节：作为同步辅助的锁	673	Section 126.4: Locks as Synchronisation aids	673
第126.5节：信号量	674	Section 126.5: Semaphore	674
第126.6节：同步	675	Section 126.6: Synchronization	675
第126.7节：可运行对象	676	Section 126.7: Runnable Object	676
第126.8节：创建基本死锁系统	677	Section 126.8: Creating basic deadlocked system	677
第126.9节：创建java.lang.Thread实例	679	Section 126.9: Creating a java.lang.Thread instance	679
第126.10节：原子操作	680	Section 126.10: Atomic operations	680
第126.11节：独占写入 / 并发读取访问	681	Section 126.11: Exclusive write / Concurrent read access	681
第126.12节：生产者-消费者	682	Section 126.12: Producer-Consumer	682
第126.13节：使用同步/易失性时可视化读/写屏障	684	Section 126.13: Visualizing read/write barriers while using synchronized / volatile	684
第126.14节：获取程序启动的所有线程状态（不包括系统线程）	685	Section 126.14: Get status of all threads started by your program excluding system threads	685
第126.15节：使用ThreadLocal	686	Section 126.15: Using ThreadLocal	686
第126.16节：使用共享全局队列的多生产者/消费者示例	687	Section 126.16: Multiple producer/consumer example with shared global queue	687
第126.17节：使用线程池对两个int数组进行加法运算	688	Section 126.17: Add two `int` arrays using a Threadpool	688
第126.18节：暂停执行	689	Section 126.18: Pausing Execution	689
第126.19节：线程中断 / 停止线程	690	Section 126.19: Thread Interruption / Stopping Threads	690
第127章：执行器、ExecutorService 和线程池	693	Chapter 127: Executor, ExecutorService and Thread pools	693
第127.1节：ThreadPoolExecutor	693	Section 127.1: ThreadPoolExecutor	693
第127.2节：从计算中获取值 - Callable	694	Section 127.2: Retrieving value from computation - Callable	694
第127.3节：submit() 与 execute() 异常处理差异	695	Section 127.3: submit() vs execute() exception handling differences	695
第127.4节：处理被拒绝的执行	697	Section 127.4: Handle Rejected Execution	697
第127.5节：火并忘记——可运行任务	697	Section 127.5: Fire and Forget - Runnable Tasks	697
第127.6节：不同类型并发构造的使用案例	698	Section 127.6: Use cases for different types of concurrency constructs	698
第127.7节：等待ExecutorService中所有任务完成	699	Section 127.7: Wait for completion of all tasks in ExecutorService	699
第127.8节：不同类型ExecutorService的使用案例	701	Section 127.8: Use cases for different types of ExecutorService	701
第127.9节：调度任务在固定时间、延迟后或重复执行	703	Section 127.9: Scheduling tasks to run at a fixed time, after a delay or repeatedly	703
第127.10节：使用线程池	704	Section 127.10: Using Thread Pools	704
第128章：ThreadLocal	705	Chapter 128: ThreadLocal	705
第128.1节：ThreadLocal的基本用法	705	Section 128.1: Basic ThreadLocal usage	705
第128.2节：ThreadLocal的Java 8函数式初始化	706	Section 128.2: ThreadLocal Java 8 functional initialization	706
第128.3节：多个线程共享一个对象	707	Section 128.3: Multiple threads with one shared object	707
第129章：在多线程应用中使用ThreadPoolExecutor。	709	Chapter 129: Using ThreadPoolExecutor in MultiThreaded applications.	709
第129.1节：使用Runnable类执行无需返回值的异步任务	709	Section 129.1: Performing Asynchronous Tasks Where No Return Value Is Needed Using a Runnable Class	709
 实例	709	 Instance	709
第129.2节：使用Callable类执行需要返回值的异步任务	710	Section 129.2: Performing Asynchronous Tasks Where a Return Value Is Needed Using a Callable Class	710
 实例	710	 Instance	710

第129.3节：使用Lambda内联定义异步任务	713	Section 129.3: Defining Asynchronous Tasks Inline using Lambdas	713
第130章：常见的Java陷阱	715	Chapter 130: Common Java Pitfalls	715
第130.1节：陷阱：使用==比较原始包装对象，如Integer	715	Section 130.1: Pitfall: using == to compare primitive wrappers objects such as Integer	715
第130.2节：陷阱：使用==比较字符串	715	Section 130.2: Pitfall: using == to compare strings	715
第130.3节：陷阱：忘记释放资源	717	Section 130.3: Pitfall: forgetting to free resources	717
第130.4节：陷阱：在尝试打开文件之前测试文件	718	Section 130.4: Pitfall: testing a file before attempting to open it	718
第130.5节：陷阱：将变量视为对象	719	Section 130.5: Pitfall: thinking of variables as objects	719
第130.6节：陷阱：内存泄漏	722	Section 130.6: Pitfall: memory leaks	722
第130.7节：陷阱：不了解String是不可变类	723	Section 130.7: Pitfall: Not understanding that String is an immutable class	723
第130.8节：陷阱：将赋值和副作用结合使用	724	Section 130.8: Pitfall: combining assignment and side-effects	724
第131章：Java陷阱——异常使用	725	Chapter 131: Java Pitfalls - Exception usage	725
第131.1节：陷阱——捕获Throwable、Exception、Error或RuntimeException	725	Section 131.1: Pitfall - Catching Throwable, Exception, Error or RuntimeException	725
第131.2节：陷阱——忽略或压制异常	726	Section 131.2: Pitfall - Ignoring or squashing exceptions	726
第131.3节：陷阱 - 抛出Throwable、Exception、Error或RuntimeException	727	Section 131.3: Pitfall - Throwing Throwable, Exception, Error or RuntimeException	727
第131.4节：陷阱 - 使用异常进行正常流程控制	728	Section 131.4: Pitfall - Using exceptions for normal flowcontrol	728
第131.5节：陷阱 - 直接继承Throwable`	729	Section 131.5: Pitfall - Directly subclassing 'Throwable'	729
第131.6节：陷阱 - 捕获InterruptedException	729	Section 131.6: Pitfall - Catching InterruptedException	729
第131.7节：陷阱 - 过多或不恰当的堆栈跟踪	731	Section 131.7: Pitfall - Excessive or inappropriate stacktraces	731
第132章：Java陷阱 - 语句语法	732	Chapter 132: Java Pitfalls - Language syntax	732
第132.1节：陷阱——在'switch'语句的case中遗漏'break'	732	Section 132.1: Pitfall - Missing a 'break' in a 'switch' case	732
第132.2节：陷阱——声明与标准类同名的类	732	Section 132.2: Pitfall - Declaring classes with the same names as standard classes	732
第132.3节：陷阱——遗漏大括号：“悬挂的if”和“悬挂的else”问题	733	Section 132.3: Pitfall - Leaving out braces: the "dangling if" and "dangling else" problems	733
第132.4节：陷阱——八进制字面量	735	Section 132.4: Pitfall - Octal literals	735
第132.5节：陷阱——使用'=='测试布尔值	735	Section 132.5: Pitfall - Using '==' to test a boolean	735
第132.6节：陷阱——忽视方法的可见性	736	Section 132.6: Pitfall - Ignoring method visibility	736
第132.7节：陷阱——使用'assert'进行参数或用户输入验证	736	Section 132.7: Pitfall: Using 'assert' for argument or user input validation	736
第132.8节：陷阱——通配符导入可能使你的代码变得脆弱	737	Section 132.8: Pitfall - Wildcard imports can make your code fragile	737
第132.9节：陷阱——分号位置错误和缺失大括号	738	Section 132.9: Pitfall - Misplaced semicolons and missing braces	738
第132.10节：陷阱——重载而非重写	739	Section 132.10: Pitfall - Overloading instead of overriding	739
第132.11节：自动拆箱空对象为基本类型的陷阱	740	Section 132.11: Pitfall of Auto-Unboxing Null Objects into Primitives	740
第133章：Java陷阱——线程与并发	741	Chapter 133: Java Pitfalls - Threads and Concurrency	741
第133.1节：陷阱——继承“java.lang.Thread”	741	Section 133.1: Pitfall - Extending 'java.lang.Thread'	741
第133.2节：陷阱——线程过多会使应用程序变慢	742	Section 133.2: Pitfall - Too many threads makes an application slower	742
第133.3节：陷阱：wait() / notify() 的错误使用	743	Section 133.3: Pitfall: incorrect use of wait() / notify()	743
第133.4节：陷阱：共享变量需要适当的同步	743	Section 133.4: Pitfall: Shared variables require proper synchronization	743
第133.5节：陷阱——线程创建相对昂贵	746	Section 133.5: Pitfall - Thread creation is relatively expensive	746
第134章：Java陷阱——空值和空指针异常	749	Chapter 134: Java Pitfalls - Nulls and NullPointerException	749
第134.1节：陷阱——“修复”意外的空值	749	Section 134.1: Pitfall - "Making good" unexpected nulls	749
第134.2节：陷阱——使用null表示空数组或集合	750	Section 134.2: Pitfall - Using null to represent an empty array or collection	750
第134.3节：陷阱——关闭时未检查I/O流是否已初始化	751	Section 134.3: Pitfall - Not checking if an I/O stream isn't even initialized when closing it	751
第134.4节：陷阱——返回null而不是抛出异常	751	Section 134.4: Pitfall - Returning null instead of throwing an exception	751
第134.5节：陷阱——不必要的使用基本类型包装类可能导致NullPointerException	752	Section 134.5: Pitfall - Unnecessary use of Primitive Wrappers can lead to NullPointerExceptions	752
第134.6节：陷阱——使用“尤达式写法”以避免NullPointerException	753	Section 134.6: Pitfall - Using "Yoda notation" to avoid NullPointerException	753
第135章：Java陷阱——性能问题	754	Chapter 135: Java Pitfalls - Performance Issues	754
第135.1节：陷阱——循环中字符串连接的扩展性差	754	Section 135.1: Pitfall - String concatenation in a loop does not scale	754
第135.2节：陷阱——使用size()判断集合是否为空效率低下	755	Section 135.2: Pitfall - Using size() to test if a collection is empty is inefficient	755
第135.3节：陷阱——为了使用==而对字符串进行常量池化是个坏主意	755	Section 135.3: Pitfall - Interning strings so that you can use == is a bad idea	755
第135.4节：陷阱——使用'new'创建基本类型包装类实例效率低下	757	Section 135.4: Pitfall - Using 'new' to create primitive wrapper instances is inefficient	757
第135.5节：陷阱——正则表达式的效率问题	757	Section 135.5: Pitfall - Efficiency concerns with regular expressions	757
第135.6节：陷阱——对无缓冲流进行小量读写效率低下	760	Section 135.6: Pitfall - Small reads / writes on unbuffered streams are inefficient	760

第135.7节：陷阱 - 过度使用原始包装类型效率低下	762
第135.8节：陷阱 - 创建日志消息的开销	763
第135.9节：陷阱 - 遍历Map的键可能效率低下	764
第135.10节：陷阱 - 调用System.gc()效率低下	764
第135.11节：陷阱 - 调用'new String(String)'效率低下	765
第136章：ServiceLoader	766
第136.1节：简单的ServiceLoader示例	766
第136.2节：日志服务	767
第137章：类加载器	769
第137.1节：实现自定义类加载器	769
第137.2节：加载外部.class文件	769
第137.3节：实例化和使用类加载器	770
第138章：通过程序创建图像	772
第138.1节：通过程序创建简单图像并显示	772
第138.2节：将图像保存到磁盘	773
第138.3节：在缓冲图像（BufferedImage）中设置单个像素的颜色	773
第138.4节：指定图像渲染质量	774
第138.5节：使用缓冲图像（BufferedImage）类创建图像	776
第138.6节：使用BufferedImage编辑和重用图像	777
第138.7节：如何缩放缓冲图像	778
第139章：原子类型	779
第139.1节：创建原子类型	779
第139.2节：原子类型的动机	779
第140章：RSA加密	783
第140.1节：使用由OAEP和GCM组成的混合密码系统的示例	783
第141章：安全对象	788
第141.1节：SealedObject（javax.crypto.SealedObject）	788
第141.2节：SignedObject（java.security.SignedObject）	788
第142章：安全与密码学	790
第142.1节：计算密码哈希	790
第142.2节：使用公钥/私钥加密和解密数据	790
第142.3节：生成密码学随机数据	791
第142.4节：生成公钥/私钥对	791
第142.5节：计算和验证数字签名	792
第143章：安全与密码学	793
第143.1节：Java加密扩展（JCE）	793
第143.2节：密钥及密钥管理	793
第143.3节：常见的Java漏洞	793
第143.4节：网络相关问题	793
第143.5节：随机性及其应用	793
第143.6节：哈希与验证	793
第144章：SecurityManager	795
第144.1节：对由ClassLoader加载的类进行沙箱保护	795
第144.2节：启用SecurityManager	796
第144.3节：实现策略拒绝规则	796
第145章：JNDI	804
第145.1节：通过JNDI进行RMI	804
第146章：sun.misc.Unsafe	808
第146.1节：通过反射实例化sun.misc.Unsafe	808

Section 135.7: Pitfall - Over-use of primitive wrapper types is inefficient	762
Section 135.8: Pitfall - The overheads of creating log messages	763
Section 135.9: Pitfall - Iterating a Map's keys can be inefficient	764
Section 135.10: Pitfall - Calling System.gc() is inefficient	764
Section 135.11: Pitfall - Calling 'new String(String)' is inefficient	765
Chapter 136: ServiceLoader	766
Section 136.1: Simple ServiceLoader Example	766
Section 136.2: Logger Service	767
Chapter 137: Classloaders	769
Section 137.1: Implementing a custom classLoader	769
Section 137.2: Loading an external .class file	769
Section 137.3: Instantiating and using a classloader	770
Chapter 138: Creating Images Programmatically	772
Section 138.1: Creating a simple image programmatically and displaying it	772
Section 138.2: Save an Image to disk	773
Section 138.3: Setting individual pixel's color in BufferedImage	773
Section 138.4: Specifying image rendering quality	774
Section 138.5: Creating an image with BufferedImage class	776
Section 138.6: Editing and re-using image with BufferedImage	777
Section 138.7: How to scale a BufferedImage	778
Chapter 139: Atomic Types	779
Section 139.1: Creating Atomic Types	779
Section 139.2: Motivation for Atomic Types	779
Chapter 140: RSA Encryption	783
Section 140.1: An example using a hybrid cryptosystem consisting of OAEP and GCM	783
Chapter 141: Secure objects	788
Section 141.1: SealedObject（javax.crypto.SealedObject）	788
Section 141.2: SignedObject（java.security.SignedObject）	788
Chapter 142: Security & Cryptography	790
Section 142.1: Compute Cryptographic Hashes	790
Section 142.2: Encrypt and Decrypt Data with Public / Private Keys	790
Section 142.3: Generate Cryptographically Random Data	791
Section 142.4: Generate Public / Private Key Pairs	791
Section 142.5: Compute and Verify Digital Signatures	792
Chapter 143: Security & Cryptography	793
Section 143.1: The JCE	793
Section 143.2: Keys and Key Management	793
Section 143.3: Common Java vulnerabilities	793
Section 143.4: Networking Concerns	793
Section 143.5: Randomness and You	793
Section 143.6: Hashing and Validation	793
Chapter 144: SecurityManager	795
Section 144.1: Sandboxing classes loaded by a ClassLoader	795
Section 144.2: Enabling the SecurityManager	796
Section 144.3: Implementing policy deny rules	796
Chapter 145: JNDI	804
Section 145.1: RMI through JNDI	804
Chapter 146: sun.misc.Unsafe	808
Section 146.1: Instantiating sun.misc.Unsafe via reflection	808

第146.2节：通过启动类路径实例化sun.misc.Unsafe	808	Section 146.2: Instantiating sun.misc.Unsafe via bootclasspath	808
第146.3节：获取Unsafe实例	808	Section 146.3: Getting Instance of Unsafe	808
第146.4节：Unsafe的用途	809	Section 146.4: Uses of Unsafe	809
第147章：Java内存模型	810	Chapter 147: Java Memory Model	810
第147.1节：内存模型的动机	810	Section 147.1: Motivation for the Memory Model	810
第147.2节：先行发生关系	812	Section 147.2: Happens-before relationships	812
第147.3节：如何避免需要理解内存模型	813	Section 147.3: How to avoid needing to understand the Memory Model	813
第147.4节：应用先行发生（Happens-before）推理的几个例子	814	Section 147.4: Happens-before reasoning applied to some examples	814
第148章：Java部署	817	Chapter 148: Java deployment	817
第148.1节：从命令行制作可执行JAR	817	Section 148.1: Making an executable JAR from the command line	817
第148.2节：为应用及其依赖创建UberJAR	818	Section 148.2: Creating an UberJAR for an application and its dependencies	818
第148.3节：创建JAR、WAR和EAR文件	819	Section 148.3: Creating JAR, WAR and EAR files	819
第148.4节：Java Web Start简介	820	Section 148.4: Introduction to Java Web Start	820
第149章：Java插件系统实现	823	Chapter 149: Java plugin system implementations	823
第149.1节：使用URLClassLoader	823	Section 149.1: Using URLClassLoader	823
第150章：JavaBean	827	Chapter 150: JavaBean	827
第150.1节：基础Java Bean	827	Section 150.1: Basic Java Bean	827
第151章：Java SE 7特性	828	Chapter 151: Java SE 7 Features	828
第151.1节：新的Java SE 7编程语言特性	828	Section 151.1: New Java SE 7 programming language features	828
第151.2节：二进制字面量	828	Section 151.2: Binary Literals	828
第151.3节：带资源的try语句	828	Section 151.3: The try-with-resources statement	828
第151.4节：数字字面量中的下划线	829	Section 151.4: Underscores in Numeric Literals	829
第151.5节：泛型实例创建的类型推断	829	Section 151.5: Type Inference for Generic Instance Creation	829
第151.6节：switch语句中的字符串	829	Section 151.6: Strings in switch Statements	829
第152章：Java SE 8特性	831	Chapter 152: Java SE 8 Features	831
第152.1节：新的Java SE 8编程语言特性	831	Section 152.1: New Java SE 8 programming language features	831
第153章：动态方法调度	832	Chapter 153: Dynamic Method Dispatch	832
第153.1节：动态方法调度 - 示例代码	832	Section 153.1: Dynamic Method Dispatch - Example Code	832
第154章：生成Java代码	835	Chapter 154: Generating Java Code	835
第154.1节：从JSON生成POJO	835	Section 154.1: Generate POJO From JSON	835
第155章：JShell	836	Chapter 155: JShell	836
第155.1节：编辑代码片段	836	Section 155.1: Editing Snippets	836
第155.2节：进入和退出JShell	837	Section 155.2: Entering and Exiting JShell	837
第155.3节：表达式	837	Section 155.3: Expressions	837
第155.4节：方法和类	838	Section 155.4: Methods and Classes	838
第155.5节：变量	838	Section 155.5: Variables	838
第156章：堆栈遍历API	839	Chapter 156: Stack-Walking API	839
第156.1节：打印当前线程的所有堆栈帧	839	Section 156.1: Print all stack frames of the current thread	839
第156.2节：打印当前调用者类	840	Section 156.2: Print current caller class	840
第156.3节：显示反射及其他隐藏帧	840	Section 156.3: Showing reflection and other hidden frames	840
第157章：套接字	842	Chapter 157: Sockets	842
第157.1节：从套接字读取	842	Section 157.1: Read from socket	842
第158章：Java套接字	843	Chapter 158: Java Sockets	843
第158.1节：一个简单的TCP回显服务器	843	Section 158.1: A simple TCP echo back server	843
第159章：FTP（文件传输协议）	846	Chapter 159: FTP (File Transfer Protocol)	846
第159.1节：连接并登录FTP服务器	846	Section 159.1: Connecting and Logging Into a FTP Server	846
第160章：在Java中使用其他脚本语言	851	Chapter 160: Using Other Scripting Languages in Java	851
第160.1节：在nashorn的脚本模式下评估JavaScript文件	851	Section 160.1: Evaluating A JavaScript file in -scripting mode of nashorn	851

第161章：C++比较	854
第161.1节：静态类成员	854
第161.2节：在其他结构中定义的类	854
第161.3节：值传递与引用传递	856
第161.4节：继承与组合	857
第161.5节：异常向下转型	857
第161.6节：抽象方法与类	857
第162章：音频	859
第162.1节：播放MIDI文件	859
第162.2节：循环播放音频文件	860
第162.3节：基本音频输出	860
第162.4节：裸机声音	861
第163章：Java打印服务	863
第163.1节：构建将被打印的文档	863
第163.2节：发现可用的打印服务	863
第163.3节：定义打印请求属性	864
第163.4节：监听打印作业请求状态变化	864
第163.5节：发现默认打印服务	866
第163.6节：从打印服务创建打印作业	866
第164章：CompletableFuture	868
第164.1节：CompletableFuture的简单示例	868
第165章：运行时命令	869
第165.1节：添加关闭钩子	869
第166章：单元测试	870
第166.1节：什么是单元测试？	870
第167章：断言	873
第167.1节：使用assert检查算术运算	873
第168章：多版本JAR文件	874
第168.1节：多版本Jar文件内容示例	874
第168.2节：使用jar工具创建多版本Jar	874
第168.3节：多版本Jar中已加载类的URL	875
第169章：即时编译器（JIT）	877
第169.1节：概述	877
第170章：字节码修改	879
第170.1节：什么是字节码？	879
第170.2节：如何使用ASM编辑jar文件	880
第170.3节：如何将ClassNode加载为Class	882
第170.4节：如何重命名jar文件中的类	883
第170.5节：Javassist基础	883
第171章：反汇编与反编译	885
第171.1节：使用javap查看字节码	885
第172章：JMX	892
第172.1节：使用平台MBean服务器的简单示例	892
第173章：Java虚拟机（JVM）	896
第173.1节：基础知识	896
第174章：XJC	897
第174.1节：从简单XSD文件生成Java代码	897
第175章：JVM标志	900

Chapter 161: C++ Comparison	854
Section 161.1: Static Class Members	854
Section 161.2: Classes Defined within Other Constructs	854
Section 161.3: Pass-by-value & Pass-by-reference	856
Section 161.4: Inheritance vs Composition	857
Section 161.5: Outcast Downcasting	857
Section 161.6: Abstract Methods & Classes	857
Chapter 162: Audio	859
Section 162.1: Play a MIDI file	859
Section 162.2: Play an Audio file Looped	860
Section 162.3: Basic audio output	860
Section 162.4: Bare metal sound	861
Chapter 163: Java Print Service	863
Section 163.1: Building the Doc that will be printed	863
Section 163.2: Discovering the available print services	863
Section 163.3: Defining print request attributes	864
Section 163.4: Listening print job request status change	864
Section 163.5: Discovering the default print service	866
Section 163.6: Creating a print job from a print service	866
Chapter 164: CompletableFuture	868
Section 164.1: Simple Example of CompletableFuture	868
Chapter 165: Runtime Commands	869
Section 165.1: Adding shutdown hooks	869
Chapter 166: Unit Testing	870
Section 166.1: What is Unit Testing?	870
Chapter 167: Asserting	873
Section 167.1: Checking arithmetic with assert	873
Chapter 168: Multi-Release JAR Files	874
Section 168.1: Example of a multi-release Jar file's contents	874
Section 168.2: Creating a multi-release Jar using the jar tool	874
Section 168.3: URL of a loaded class inside a multi-release Jar	875
Chapter 169: Just in Time (JIT) compiler	877
Section 169.1: Overview	877
Chapter 170: Bytecode Modification	879
Section 170.1: What is Bytecode?	879
Section 170.2: How to edit jar files with ASM	880
Section 170.3: How to load a ClassNode as a Class	882
Section 170.4: How to rename classes in a jar file	883
Section 170.5: Javassist Basic	883
Chapter 171: Disassembling and Decompiling	885
Section 171.1: Viewing bytecode with javap	885
Chapter 172: JMX	892
Section 172.1: Simple example with Platform MBean Server	892
Chapter 173: Java Virtual Machine (JVM)	896
Section 173.1: These are the basics	896
Chapter 174: XJC	897
Section 174.1: Generating Java code from simple XSD file	897
Chapter 175: JVM Flags	900

第175.1节 : -XXaggressive	900
第175.2节 : -XXallocClearChunks	900
第175.3节 : -XXallocClearChunkSize	900
第175.4节 : -XXcallProfiling	900
第175.5节 : -XXdisableFatSpin	901
第175.6节 : -XXdisableGCHeistics	901
第175.7节 : -XXdumpSize	901
第175.8节 : -XXexitOnOutOfMemory	902
第176章 : JVM工具接口	903
第176.1节 : 遍历从对象可达的对象 (堆1.0)	903
第176.2节 : 获取JVMTI环境	905
第176.3节 : Agent_OnLoad方法内的初始化示例	905
第177章 : Java内存管理	907
第177.1节 : 设置堆、永久代和栈大小	907
第177.2节 : 垃圾回收	908
第177.3节 : Java中的内存泄漏	910
第177.4节 : 终结处理	911
第177.5节 : 手动触发垃圾回收	912
第178章 : Java性能调优	913
第178.1节 : 基于证据的Java性能调优方法	913
第178.2节 : 减少字符串数量	914
第178.3节 : 一般方法	914
第179章 : 基准测试	916
第179.1节 : 简单的JMH示例	916
第180章 : 上传文件到AWS	919
第180.1节 : 上传文件到s3存储桶	919
第181章 : AppDynamics和TIBCO BusinessWorks的轻松集成监控	921
第181.1节 : Appdynamics中所有BW应用程序一次性仪表化示例	921
附录A : 安装Java (标准版)	922
A.1节 : 在Windows上安装后设置%PATH%和%JAVA_HOME%	922
A.2节 : 在Linux上安装Java JDK	923
A.3节 : 在macOS上安装Java JDK	925
A.4节 : 在Windows上安装Java JDK或JRE	926
附录 A.5 : 使用 alternatives 配置和切换 Linux 上的 Java 版本	927
附录 A.6 : Java 开发所需内容	928
附录 A.7 : 选择合适的 Java SE 版本	928
附录 A.8 : Java 版本和发布命名	929
附录 A.9 : 使用最新 tar 文件在 Linux 上安装 Oracle Java	929
附录 A.10 : Linux 上安装后的检查和配置	930
附录B : Java版本、版本号、发布和发行版	933
B.1节 : Java SE JRE与Java SE JDK发行版的区别	933
B.2节 : Java SE版本	934
B.3节 : Java EE、Java SE、Java ME和JavaFX的区别	935
C附录 : 类路径	937
C.1节 : 指定类路径的不同方式	937
C.2节 : 将目录中的所有JAR添加到类路径	937
C.3节 : 从类路径加载资源	938
C.4节 : 类路径语法	938

Section 175.1: -XXaggressive	900
Section 175.2: -XXallocClearChunks	900
Section 175.3: -XXallocClearChunkSize	900
Section 175.4: -XXcallProfiling	900
Section 175.5: -XXdisableFatSpin	901
Section 175.6: -XXdisableGCHeistics	901
Section 175.7: -XXdumpSize	901
Section 175.8: -XXexitOnOutOfMemory	902
Chapter 176: JVM Tool Interface	903
Section 176.1: Iterate over objects reachable from object (Heap 1.0)	903
Section 176.2: Get JVMTI environment	905
Section 176.3: Example of initialization inside of Agent_OnLoad method	905
Chapter 177: Java Memory Management	907
Section 177.1: Setting the Heap, PermGen and Stack sizes	907
Section 177.2: Garbage collection	908
Section 177.3: Memory leaks in Java	910
Section 177.4: Finalization	911
Section 177.5: Manually triggering GC	912
Chapter 178: Java Performance Tuning	913
Section 178.1: An evidence-based approach to Java performance tuning	913
Section 178.2: Reducing amount of Strings	914
Section 178.3: General approach	914
Chapter 179: Benchmarks	916
Section 179.1: Simple JMH example	916
Chapter 180: FileUpload to AWS	919
Section 180.1: Upload file to s3 bucket	919
Chapter 181: AppDynamics and TIBCO BusinessWorks Instrumentation for Easy Integration	921
Section 181.1: Example of Instrumentation of all BW Applications in a Single Step for Appdynamics	921
Appendix A: Installing Java (Standard Edition)	922
Section A.1: Setting %PATH% and %JAVA_HOME% after installing on Windows	922
Section A.2: Installing a Java JDK on Linux	923
Section A.3: Installing a Java JDK on macOS	925
Section A.4: Installing a Java JDK or JRE on Windows	926
Section A.5: Configuring and switching Java versions on Linux using alternatives	927
Section A.6: What do I need for Java Development	928
Section A.7: Selecting an appropriate Java SE release	928
Section A.8: Java release and version naming	929
Section A.9: Installing Oracle Java on Linux with latest tar file	929
Section A.10: Post-installation checking and configuration on Linux	930
Appendix B: Java Editions, Versions, Releases and Distributions	933
Section B.1: Differences between Java SE JRE or Java SE JDK distributions	933
Section B.2: Java SE Versions	934
Section B.3: Differences between Java EE, Java SE, Java ME and JavaFX	935
Appendix C: The Classpath	937
Section C.1: Different ways to specify the classpath	937
Section C.2: Adding all JARs in a directory to the classpath	937
Section C.3: Load a resource from the classpath	938
Section C.4: Classpath path syntax	938

C.5节：动态类路径	939
C.6节：将类名映射到路径名	939
C.7节：引导类路径	939
C.8节：类路径的含义：搜索机制	940
D附录：资源（在类路径上）	941
D.1节：加载默认配置	941
D.2节：从资源加载图像	941
D.3节：使用类加载器查找和读取资源	941
D.4节：从多个JAR加载同名资源	943
鸣谢	944
你可能还喜欢	958

Section C.5: Dynamic Classpath	939
Section C.6: Mapping classnames to pathnames	939
Section C.7: The bootstrap classpath	939
Section C.8: What the classpath means: how searches work	940
Appendix D: Resources (on classpath)	941
Section D.1: Loading default configuration	941
Section D.2: Loading an image from a resource	941
Section D.3: Finding and reading resources using a classloader	941
Section D.4: Loading same-name resource from multiple JARs	943
Credits	944
You may also like	958

请随意免费分享此PDF，
本书的最新版本可从以下网址下载：

<https://goalkicker.com/JavaBook>

本Java® 专业人士笔记一书汇编自[Stack Overflow Documentation](#)，内容由Stack Overflow的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

这是一本非官方的免费书籍，旨在教育用途，与官方Java®组织或公司及Stack Overflow无关。所有商标和注册商标均为其各自公司所有者所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:

<https://goalkicker.com/JavaBook>

This Java® Notes for Professionals book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow.
Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Java® group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：Java语言入门

Java SE 版本	代号	终止支持（免费1）	发布日期
Java SE 10 (早期访问)	无	未来	2018-03-20
Java SE 9	无	未来	2017-07-27
Java SE 8	蜘蛛	未来	2014-03-18
Java SE 7	海豚	2015-04-14	2011-07-28
Java SE 6	Mustang	2013-04-16	2006-12-23
Java SE 5	Tiger	2009-11-04	2004-10-04
Java SE 1.4	Merlin	2009-11-04之前	2002-02-06
Java SE 1.3	Kestrel	2000-05-08 之前	2009-11-04
Java SE 1.2	2009-11-04 之前的游乐场	1998-12-08	
Java SE 1.1	无	2009-11-04 之前	1997-02-19
Java SE 1.0	橡树 (Oak)	2009-11-04 之前	1996-01-21

第1.1节：创建你的第一个Java程序

在你的[文本编辑器](#)或[集成开发环境 \(IDE\)](#) 中创建一个名为HelloWorld.java的新文件。然后将以下代码块粘贴到文件中并保存：

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

[在 Ideone 上实时运行](#)

注意：为了让Java识别这是一个public class（并且不抛出编译时错误），文件名必须与类名相同（本例中为HelloWorld），并且带有.java扩展名。类名前还应有一个public访问修饰符。

命名规范建议Java类以大写字母开头，并采用驼峰命名法格式（即每个单词的首字母大写）。规范建议避免使用下划线（_）和美元符号（\$）。

要编译，打开终端窗口并导航到HelloWorld.java所在的目录：

```
cd /path/to/containing/folder/
```

注意：`cd`是终端中用于切换目录的命令。

输入javac，后跟文件名和扩展名，如下所示：

```
$ javac HelloWorld.java
```

即使你已经安装了JDK并且能够从IDE（例如eclipse等）运行程序，出现错误'javac'不是内部或外部命令，也不是可运行的程序或批处理文件的情况也很常见。

因为默认情况下路径没有添加到环境变量中。

Chapter 1: Getting started with Java Language

Java SE Version	Code Name	End-of-life (free1)	Release Date
Java SE 10 (Early Access)	None	future	2018-03-20
Java SE 9	None	future	2017-07-27
Java SE 8	Spider	future	2014-03-18
Java SE 7	Dolphin	2015-04-14	2011-07-28
Java SE 6	Mustang	2013-04-16	2006-12-23
Java SE 5	Tiger	2009-11-04	2004-10-04
Java SE 1.4	Merlin	prior to 2009-11-04	2002-02-06
Java SE 1.3	Kestrel	prior to 2009-11-04	2000-05-08
Java SE 1.2	Playground	prior to 2009-11-04	1998-12-08
Java SE 1.1	None	prior to 2009-11-04	1997-02-19
Java SE 1.0	Oak	prior to 2009-11-04	1996-01-21

Section 1.1: Creating Your First Java Program

Create a new file in your [text editor](#) or [IDE](#) named `HelloWorld.java`. Then paste this code block into the file and save:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

[Run live on Ideone](#)

Note: For Java to recognize this as a `public class` (and not throw a [compile time error](#)), the filename must be the same as the class name (`HelloWorld` in this example) with a `.java` extension. There should also be a `public` access modifier before it.

Naming conventions recommend that Java classes begin with an uppercase character, and be in [camel case](#) format (in which the first letter of each word is capitalized). The conventions recommend against underscores (_) and dollar signs (\$).

To compile, open a terminal window and navigate to the directory of `HelloWorld.java`:

```
cd /path/to/containing/folder/
```

Note: `cd` is the terminal command to change directory.

Enter `javac` followed by the file name and extension as follows:

```
$ javac HelloWorld.java
```

It's fairly common to get the error '`javac`' is not recognized as an internal or external command, operable program or batch file. even when you have installed the JDK and are able to run the program from IDE ex. eclipse etc. Since the path is not added to the environment by default.

如果你在 Windows 上遇到这个问题，解决方法是，首先尝试浏览到你的javac.exe路径，它很可能位于你的 C:\Program Files\Java\jdk(版本号)\bin。然后尝试用下面的命令运行。

```
$ C:\Program Files\Java\jdk(版本号)\bin\javac HelloWorld.java
```

之前我们调用javac时，命令和上面的一样。只是那时你的操作系统知道javac的位置。所以现在我们告诉它，这样你就不必每次都输入完整路径。我们需要把这个路径添加到PATH中。

在 Windows XP/Vista/7/8/10 中编辑PATH环境变量的方法：

- 控制面板 ⇒ 系统 ⇒ 高级系统设置
- 切换到“高级”选项卡 ⇒ 环境变量
- 在“系统变量”中，向下滚动选择“PATH” ⇒ 编辑

此操作不可撤销请小心操作。首先将现有路径复制到记事本。然后为了获取准确的javac路径，手动浏览到javac所在的文件夹，点击地址栏并复制。路径应类似于 c:\Program Files\Java\jdk1.8.0_xx\bin

在“变量值”字段中，将此路径粘贴到所有现有目录的前面，后面加分号 (;)。不要删除任何现有条目。

```
变量名 : PATH  
变量值 : c:\Program Files\Java\jdk1.8.0_xx\bin;[现有条目...]
```

现在这个问题应该解决了。

对于基于Linux的系统请尝试[这里](#)。

注意：javac命令调用Java编译器。

编译器随后将生成一个名为HelloWorld.class的字节码文件，该文件可以在Java虚拟机（JVM）中执行。Java编程语言的编译器javac读取用Java编程语言编写的源文件，并将其编译成字节码类文件。编译器还可以选择使用可插拔注解处理API处理源文件和类文件中的注解。编译器是一个命令行工具，但也可以通过Java编译器API调用。

要运行你的程序，输入java，后跟包含main方法的类名（在我们的例子中是HelloWorld）。注意省略了.class：

```
$ java HelloWorld
```

注意：java命令运行Java应用程序。

这将在控制台输出：

你好，世界！

你已经成功编写并构建了你的第一个Java程序！

注意：为了使Java命令（java、javac等）被识别，您需要确保：

- 已安装 JDK（例如Oracle、OpenJDK及其他来源）

In case you get this on windows, to resolve, first try browsing to your javac.exe path, it's most probably in your C:\Program Files\Java\jdk(version number)\bin. Then try running it with below.

```
$ C:\Program Files\Java\jdk(version number)\bin\javac HelloWorld.java
```

Previously when we were calling javac it was same as above command. Only in that case your OS knew where javac resided. So let's tell it now, this way you don't have to type the whole path every-time. We would need to add this to our PATH

To edit the PATH environment variable in Windows XP/Vista/7/8/10:

- Control Panel ⇒ System ⇒ Advanced system settings
- Switch to "Advanced" tab ⇒ Environment Variables
- In "System Variables", scroll down to select "PATH" ⇒ Edit

You cannot undo this so be careful. First copy your existing path to notepad. Then to get the exact PATH to your javac browse manually to the folder where javac resides and click on the address bar and then copy it. It should look something like c:\Program Files\Java\jdk1.8.0_xx\bin

In "Variable value" field, paste this IN FRONT of all the existing directories, followed by a semi-colon (;). DO NOT DELETE any existing entries.

```
Variable name : PATH  
Variable value : c:\Program Files\Java\jdk1.8.0_xx\bin;[Existing Entries...]
```

Now this should resolve.

For Linux Based systems [try here](#).

Note: The javac command invokes the Java compiler.

The compiler will then generate a [bytecode](#) file called [HelloWorld.class](#) which can be executed in the [Java Virtual Machine \(JVM\)](#). The Java programming language compiler, javac, reads source files written in the Java programming language and compiles them into bytecode class files. Optionally, the compiler can also process annotations found in source and class files using the Pluggable Annotation Processing API. The compiler is a command line tool but can also be invoked using the Java Compiler API.

To run your program, enter java followed by the name of the class which contains the main method (HelloWorld in our example). Note how the .class is omitted:

```
$ java HelloWorld
```

Note: The java command runs a Java application.

This will output to your console:

Hello, World!

You have successfully coded and built your very first Java program!

Note: In order for Java commands (java, javac, etc) to be recognized, you will need to make sure:

- A JDK is installed (e.g. [Oracle](#), [OpenJDK](#) and other sources)

- 您的环境变量已正确设置

您需要使用由您的 JVM 提供的编译器 (javac) 和执行器 (java)。要查看您安装了哪些版本, 请在命令行输入`java -version`和`javac -version`。程序的版本号将显示在终端中 (例如1.8.0_73)。

深入了解 Hello World 程序

"Hello World"程序包含一个文件, 该文件由一个HelloWorld类定义、一个main方法以及main方法内的一条语句组成。

```
public class HelloWorld {
```

`class`关键字开始定义名为HelloWorld的类。每个 Java 应用程序至少包含一个类定义 (有关类的更多信息)。

```
    public static void main(String[] args) {
```

这是一个入口点方法 (由其名称和签名`public static void main(String[])`定义), JVM 可以从这里运行您的程序。每个 Java 程序都应有一个。它是:

- **public**: 表示该方法可以从任何地方调用, 包括程序外部。有关详细信息, 请参见 Visibility。
- **static**: 表示该方法存在且可以独立运行 (在类级别, 无需创建对象)。
- **void**: 表示该方法不返回任何值。注意: 这与C和C++不同, 后者期望返回一个如`int`的返回码 (Java的方式是`System.exit()`)。

该main方法接受:

- 一个数组 (通常称为args) 的`String`类型参数传递给main函数 (例如来自命令行参数)。

几乎所有这些都是Java入口点方法所必需的。

非必需部分:

- 名称args是一个变量名, 可以随意命名, 尽管通常称为args。
- 其参数类型是数组 (`String[] args`) 还是可变参数 (`String... args`) 无关紧要, 因为数组可以传递给可变参数。

注意: 单个应用程序可能包含多个类, 每个类都包含入口点 (main) 方法。应用程序的入口点由传递给java命令的类名决定。

在主方法内部, 我们看到以下语句:

```
System.out.println("Hello, World!");
```

让我们逐个元素分解这条语句:

元素	用途
<code>System</code>	这表示后续表达式将调用来自 <code>java.lang</code> 包中的 <code>System</code> 类。

- Your environment variables are properly [set up](#)

You will need to use a compiler (javac) and an executor (java) provided by your JVM. To find out which versions you have installed, enter `java -version` and `javac -version` on the command line. The version number of your program will be printed in the terminal (e.g. 1.8.0_73).

A closer look at the Hello World program

The "Hello World" program contains a single file, which consists of a HelloWorld class definition, a `main` method, and a statement inside the `main` method.

```
public class HelloWorld {
```

The `class` keyword begins the class definition for a class named `HelloWorld`. Every Java application contains at least one class definition (Further information about classes).

```
    public static void main(String[] args) {
```

This is an entry point method (defined by its name and signature of `public static void main(String[])`) from which the JVM can run your program. Every Java program should have one. It is:

- **public**: meaning that the method can be called from anywhere mean from outside the program as well. See Visibility for more information on this.
- **static**: meaning it exists and can be run by itself (at the class level without creating an object).
- **void**: meaning it returns no value. **Note:** This is unlike C and C++ where a return code such as `int` is expected (Java's way is `System.exit()`).

This main method accepts:

- An array (typically called args) of `String`s passed as arguments to main function (e.g. from command line arguments).

Almost all of this is required for a Java entry point method.

Non-required parts:

- The name args is a variable name, so it can be called anything you want, although it is typically called args.
- Whether its parameter type is an array (`String[] args`) or Varargs (`String... args`) does not matter because arrays can be passed into varargs.

Note: A single application may have multiple classes containing an entry point (`main`) method. The entry point of the application is determined by the class name passed as an argument to the java command.

Inside the main method, we see the following statement:

```
System.out.println("Hello, World!");
```

Let's break down this statement element-by-element:

Element	Purpose
<code>System</code>	this denotes that the subsequent expression will call upon the <code>System</code> class, from the <code>java.lang</code> package.

这是一个“点操作符”。点操作符使你能够访问类的成员1；即它的字段（变量）和方法。在本例中，这个点操作符允许你引用System类中的out静态字段。

out

这是System类中一个类型为PrintStream的静态字段名称，包含标准输出功能。

这是另一个点操作符。该点操作符用于访问变量中的println方法。

println

这是PrintStream类中的一个方法名。该方法特别用于将参数内容打印到控制台，并在后面插入换行符。

(

这个括号表示正在访问一个方法（而不是字段），并开始传递给println方法的参数。

"Hello,
World!"

这是作为参数传递给println方法的字符串字面量。两端的双引号将文本限定为字符串。

)

这个括号表示传递给println方法的参数结束。

;

这个分号标志着语句的结束。

注意：Java 中的每条语句必须以分号（;）结尾。

方法体和类体随后被关闭。

```
} // 主函数作用域结束  
} // HelloWorld 类作用域结束
```

这里有另一个演示面向对象范式的例子。让我们用一个（是的，只有一个！）成员来建模一个足球队。成员可以更多，但我们在讲到数组时讨论。

首先，定义我们的Team类：

```
public class Team {  
    Member member;  
    public Team(Member member) { // 这个队伍里有谁?  
        this.member = member; // 这个团队中有一个“成员”!  
    }  
}
```

现在，让我们定义我们的Member类：

```
class Member {  
    private String name;  
    private String type;  
    private int level; // 注意这里的数据类型  
    private int rank; // 同样注意这里的数据类型  
  
    public Member(String name, String type, int level, int rank) {  
        this.name = name;  
        this.type = type;  
        this.level = level;  
        this.rank = rank;  
    }  
}
```

我们为什么在这里使用private？嗯，如果有人想知道你的名字，他们应该直接问你，而不是伸手进你的口袋拿出你的社会保障卡。这个private就起到了类似的作用：它防止外部实体访问你的变量。你只能通过getter函数（如下所示）返回private成员。

this is a "dot operator". Dot operators provide you access to a classes members1; i.e. its fields (variables) and its methods. In this case, this dot operator allows you to reference the out static field within the System class.

out

this is the name of the static field of PrintStream type within the System class containing the standard output functionality.

这是另一个点操作符。该点操作符提供了对out变量中println方法的访问。

println

this is the name of a method within the PrintStream class. This method in particular prints the contents of the parameters into the console and inserts a newline after.

(

this parenthesis indicates that a method is being accessed (and not a field) and begins the parameters being passed into the println method.

"Hello,
World!"

this is the String literal that is passed as a parameter, into the println method. The double quotation marks on each end delimit the text as a String.

)

this parenthesis signifies the closure of the parameters being passed into the println method.

;

this semicolon marks the end of the statement.

Note: Each statement in Java must end with a semicolon (;).

The method body and class body are then closed.

```
} // end of main function scope  
} // end of class HelloWorld scope
```

Here's another example demonstrating the OO paradigm. Let's model a football team with one (yes, one!) member. There can be more, but we'll discuss that when we get to arrays.

First, let's define our Team class:

```
public class Team {  
    Member member;  
    public Team(Member member) { // who is in this Team?  
        this.member = member; // one 'member' is in this Team!  
    }  
}
```

Now, let's define our Member class:

```
class Member {  
    private String name;  
    private String type;  
    private int level; // note the data type here  
    private int rank; // note the data type here as well  
  
    public Member(String name, String type, int level, int rank) {  
        this.name = name;  
        this.type = type;  
        this.level = level;  
        this.rank = rank;  
    }  
}
```

Why do we use **private** here? Well, if someone wanted to know your name, they should ask you directly, instead of reaching into your pocket and pulling out your Social Security card. This **private** does something like that: it prevents outside entities from accessing your variables. You can only return **private** members through getter functions (shown below).

将所有内容整合在一起，并添加之前讨论的getter方法和主方法后，我们得到：

```
public class Team {  
    Member member;  
    public Team(Member member) {  
        this.member = member;  
    }  
  
    // 这是我们的主方法  
    public static void main(String[] args) {  
        Member myMember = new Member("Aurieel", "light", 10, 1);  
        Team myTeam = new Team(myMember);  
        System.out.println(myTeam.member.getName());  
        System.out.println(myTeam.member.getType());  
        System.out.println(myTeam.member.getLevel());  
        System.out.println(myTeam.member.getRank());  
    }  
}  
  
class Member {  
    private String name;  
    private String type;  
    private int level;  
    private int rank;  
  
    public Member(String name, String type, int level, int rank) {  
        this.name = name;  
        this.类型 = 类型;  
        this.等级 = 等级;  
        this.排名 = 排名;  
    }  
  
    /* 让我们在这里定义getter函数 */  
    public String getName() { // 你叫什么名字?  
        return this.name; // 我的名字是...  
    }  
  
    public String getType() { // 你是什么类型?  
        return this.type; // 我的类型是 ...  
    }  
  
    public int getLevel() { // 你的等级是多少?  
        return this.level; // 我的等级是 ...  
    }  
  
    public int getRank() { // 你的排名是多少?  
        return this.rank; // 我的排名是  
    }  
}
```

输出：

```
Aurieel  
light  
10  
1
```

[在 ideone 上运行](#)

After putting it all together, and adding the getters and main method as discussed before, we have:

```
public class Team {  
    Member member;  
    public Team(Member member) {  
        this.member = member;  
    }  
  
    // here's our main method  
    public static void main(String[] args) {  
        Member myMember = new Member("Aurieel", "light", 10, 1);  
        Team myTeam = new Team(myMember);  
        System.out.println(myTeam.member.getName());  
        System.out.println(myTeam.member.getType());  
        System.out.println(myTeam.member.getLevel());  
        System.out.println(myTeam.member.getRank());  
    }  
}  
  
class Member {  
    private String name;  
    private String type;  
    private int level;  
    private int rank;  
  
    public Member(String name, String type, int level, int rank) {  
        this.name = name;  
        this.type = type;  
        this.level = level;  
        this.rank = rank;  
    }  
  
    /* let's define our getter functions here */  
    public String getName() { // what is your name?  
        return this.name; // my name is ...  
    }  
  
    public String getType() { // what is your type?  
        return this.type; // my type is ...  
    }  
  
    public int getLevel() { // what is your level?  
        return this.level; // my level is ...  
    }  
  
    public int getRank() { // what is your rank?  
        return this.rank; // my rank is  
    }  
}
```

Output:

```
Aurieel  
light  
10  
1
```

[Run on ideone](#)

再次说明，`main` 方法位于 `Test` 类中，是我们程序的入口点。没有 `main` 方法，我们无法告诉 Java 虚拟机（JVM）从哪里开始执行程序。

1 - 因为`HelloWorld`类与`System`类关系不大，所以它只能访问`public`数据。

Once again, the `main` method inside the `Test` class is the entry point to our program. Without the `main` method, we cannot tell the Java Virtual Machine (JVM) from where to begin execution of the program.

1 - Because the `HelloWorld` class has little relation to the `System` class, it can only access `public` data.

第2章：类型转换

第2.1节：数值原始类型转换

数值原始类型可以通过两种方式进行转换。隐式转换发生在源类型的范围小于目标类型时。

```
//隐式转换
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intVar;
float floatVar = longVar;
double doubleVar = floatVar;
```

当源类型的范围大于目标类型时，必须进行显式转换。

```
//显式转换
double doubleVar = 42.0d;
float floatVar = (float) doubleVar;
long longVar = (long) floatVar;
int intVar = (int) longVar;
short shortVar = (short) intVar;
byte byteVar = (byte) shortVar;
```

当将浮点原始类型 (float, double) 转换为整数原始类型时，数字会被向下取整。

第2.2节：基本数值提升

```
static void testNumericPromotion() {

    char char1 = 1, char2 = 2;
    short short1 = 1, short2 = 2;
    int int1 = 1, int2 = 2;
    float float1 = 1.0f, float2 = 2.0f;

    // char1 = char1 + char2;      // 错误：无法从 int 转换为 char ;
    // short1 = short1 + short2;  // 错误：无法从 int 转换为 short ;
    int1 = char1 + char2;        // char 被提升为 int。
    int1 = short1 + short2;      // short 被提升为 int。
    int1 = char1 + short2;       // char 和 short 都被提升为 int。
    float1 = short1 + float2;    // short 被提升为 float。
    int1 = int1 + int2;          // int 保持不变。
}
```

第2.3节：非数值原始类型转换

boolean 类型不能转换为/自任何其他原始类型。

通过使用 Unicode 指定的码点映射，char 可以转换为任何数值类型，反之亦然。一个 char 在内存中表示为无符号的 16 位整数值（2 字节），因此转换为 byte (1 字节) 时会丢弃其中的 8 位（对于 ASCII 字符来说这是安全的）。Character 类的实用方法使用 int (4 字节) 来传输码点值，但使用 short (2 字节) 存储 Unicode 码点也足够。

```
int badInt = (int) true; // 编译错误：类型不兼容
```

Chapter 2: Type Conversion

Section 2.1: Numeric primitive casting

Numeric primitives can be cast in two ways. *Implicit* casting happens when the source type has smaller range than the target type.

```
//Implicit casting
byte byteVar = 42;
short shortVar = byteVar;
int intVar = shortVar;
long longVar = intVar;
float floatVar = longVar;
double doubleVar = floatVar;
```

Explicit casting has to be done when the source type has larger range than the target type.

```
//Explicit casting
double doubleVar = 42.0d;
float floatVar = (float) doubleVar;
long longVar = (long) floatVar;
int intVar = (int) longVar;
short shortVar = (short) intVar;
byte byteVar = (byte) shortVar;
```

When casting floating point primitives (float, double) to whole number primitives, the number is **rounded down**.

Section 2.2: Basic Numeric Promotion

```
static void testNumericPromotion() {

    char char1 = 1, char2 = 2;
    short short1 = 1, short2 = 2;
    int int1 = 1, int2 = 2;
    float float1 = 1.0f, float2 = 2.0f;

    // char1 = char1 + char2;      // Error: Cannot convert from int to char ;
    // short1 = short1 + short2;  // Error: Cannot convert from int to short ;
    int1 = char1 + char2;        // char is promoted to int.
    int1 = short1 + short2;      // short is promoted to int.
    int1 = char1 + short2;       // both char and short promoted to int.
    float1 = short1 + float2;    // short is promoted to float.
    int1 = int1 + int2;          // int is unchanged.
}
```

Section 2.3: Non-numeric primitive casting

The boolean type cannot be cast to/from any other primitive type.

A char can be cast to/from any numeric type by using the code-point mappings specified by Unicode. A char is represented in memory as an unsigned 16-bit integer value (2 bytes), so casting to byte (1 byte) will drop 8 of those bits (this is safe for ASCII characters). The utility methods of the Character class use int (4 bytes) to transfer to/from code-point values, but a short (2 bytes) would also suffice for storing a Unicode code-point.

```
int badInt = (int) true; // Compiler error: incompatible types
```

```

char char1 = (char) 65; // A
byte byte1 = (byte) 'A'; // 65
short short1 = (short) 'A'; // 65
int int1 = (int) 'A'; // 65

char char2 = (char) 8253; // ?
byte byte2 = (byte) '?'; // 61 (截断的代码点进入ASCII范围)
short short2 = (short) '?'; // 8253
int int2 = (int) '?'; // 8253

```

第2.4节：对象类型转换

与基本类型一样，对象也可以显式和隐式地进行类型转换。

隐式类型转换发生在源类型扩展或实现目标类型时（转换为超类或接口）。

当目标类型扩展或实现源类型时，必须进行显式类型转换（转换为子类型）。当被转换的对象不是目标类型（或目标的子类型）时，这可能会产生运行时异常（ClassCastException）。

```

Float floatVar = new Float(42.0f);
Number n = floatVar;           //隐式转换 (Float 实现了 Number)
Float floatVar2 = (Float) n;    //显式转换
Double doubleVar = (Double) n;  //抛出异常 (对象不是 Double 类型)

```

第2.5节：使用 instanceof 测试对象是否可以转换

Java 提供了 instanceof 操作符，用于测试对象是否为某个类型或该类型的子类。程序随后可以选择是否对该对象进行类型转换。

```

Object obj = Calendar.getInstance();
long time = 0;

if(obj instanceof Calendar)
{
    time = ((Calendar)obj).getTime();
}
if(obj instanceof Date)
{
    time = ((Date)obj).getTime(); // 这行代码永远不会被执行，obj 不是 Date 类型。
}

```

```

char char1 = (char) 65; // A
byte byte1 = (byte) 'A'; // 65
short short1 = (short) 'A'; // 65
int int1 = (int) 'A'; // 65

char char2 = (char) 8253; // ?
byte byte2 = (byte) '?'; // 61 (truncated code-point into the ASCII range)
short short2 = (short) '?'; // 8253
int int2 = (int) '?'; // 8253

```

Section 2.4: Object casting

As with primitives, objects can be cast both explicitly and implicitly.

Implicit casting happens when the source type extends or implements the target type (casting to a superclass or interface).

Explicit casting has to be done when the source type is extended or implemented by the target type (casting to a subtype). This can produce a runtime exception ([ClassCastException](#)) when the object being cast is not of the target type (or the target's subtype).

```

Float floatVar = new Float(42.0f);
Number n = floatVar;           //Implicit (Float implements Number)
Float floatVar2 = (Float) n;    //Explicit
Double doubleVar = (Double) n;  //Throws exception (the object is not Double)

```

Section 2.5: Testing if an object can be cast using instanceof

Java provides the **instanceof** operator to test if an object is of a certain type, or a subclass of that type. The program can then choose to cast or not cast that object accordingly.

```

Object obj = Calendar.getInstance();
long time = 0;

if(obj instanceof Calendar)
{
    time = ((Calendar)obj).getTime();
}
if(obj instanceof Date)
{
    time = ((Date)obj).getTime(); // This line will never be reached, obj is not a Date type.
}

```

第三章：Getter 和 Setter

本文讨论了 getter 和 setter；这是在 Java 类中提供数据访问的标准方式。

第3.1节：使用 setter 或 getter 实现约束

Setter 和 Getter 允许对象包含私有变量，这些变量可以在有限制的情况下被访问和修改。例如，

```
public class Person {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        if(name!=null && name.length()>2)  
            this.name = name;  
    }  
}
```

在这个Person类中，有一个变量：name。这个变量可以通过getName()方法访问，也可以通过setName(String)方法修改，但设置名字时要求新名字长度大于2个字符且不为null。使用setter方法而不是将变量name设为public，允许其他人以一定限制来设置name的值。相同的限制也可以应用到getter方法：

```
public String getName(){  
    if(name.length()>16)  
        return "名字太长了！";  
    else  
        return name;  
}
```

在上面修改后的getName()方法中，只有当name的长度小于或等于16时才返回它。否则，返回"Name is too large"。这样允许程序员创建可访问且可修改的变量，同时防止客户端类不必要的编辑这些变量。

第3.2节：为什么使用Getter和Setter？

考虑一个包含带有getter和setter的对象的基本Java类：

```
public class CountHolder {  
    private int count = 0;  
  
    public int getCount() { return count; }  
    public void setCount(int c) { count = c; }  
}
```

我们无法访问count变量，因为它是私有的。但我们可以访问getCount()和setCount(int)方法，因为它们是公共的。对此，有人可能会提出疑问：为什么要引入中间人？为什么不直接将count设为公共？

```
public class CountHolder {
```

Chapter 3: Getters and Setters

This article discusses getters and setters; the standard way to provide access to data in Java classes.

Section 3.1: Using a setter or getter to implement a constraint

Setters and Getters allow for an object to contain private variables which can be accessed and changed with restrictions. For example,

```
public class Person {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        if(name!=null && name.length()>2)  
            this.name = name;  
    }  
}
```

In this Person class, there is a single variable: name. This variable can be accessed using the getName() method and changed using the setName(String) method, however, setting a name requires the new name to have a length greater than 2 characters and to not be null. Using a setter method rather than making the variable name public allows others to set the value of name with certain restrictions. The same can be applied to the getter method:

```
public String getName(){  
    if(name.length()>16)  
        return "Name is too large!";  
    else  
        return name;  
}
```

In the modified getName() method above, the name is returned only if its length is less than or equal to 16. Otherwise, "Name is too large" is returned. This allows the programmer to create variables that are reachable and modifiable however they wish, preventing client classes from editing the variables unwantedly.

Section 3.2: Why Use Getters and Setters?

Consider a basic class containing an object with getters and setters in Java:

```
public class CountHolder {  
    private int count = 0;  
  
    public int getCount() { return count; }  
    public void setCount(int c) { count = c; }  
}
```

We can't access the count variable because it's private. But we can access the getCount() and the setCount(int) methods because they are public. To some, this might raise the question; why introduce the middleman? Why not just simply make they count public?

```
public class CountHolder {
```

```
public int count = 0;  
}
```

在所有实际用途上，这两者在功能上完全相同。它们之间的区别在于可扩展性。考虑一下每个类所表达的内容：

- 第一：“我有一个方法，它会给你一个int值，还有一个方法会将该值设置为另一个int。”
- 第二：“我有一个int，你可以随意设置和获取它。”

这两者听起来可能相似，但第一个实际上在本质上更加受保护；它只允许你按照int的方式与其内部状态交互。这把主动权交给了它自己；它可以选择内部交互的方式。

第二个则将其内部实现暴露在外部，现在不仅容易受到外部用户的影响，而且在API的情况下，committed于维护该实现（否则将发布非向后兼容的API）。

让我们考虑如果想要同步访问以修改和访问计数。在第一个例子中，这很简单：

```
public class CountHolder {  
    private int count = 0;  
  
    public synchronized int getCount() { return count; }  
    public synchronized void setCount(int c) { count = c; }  
}
```

但在第二个例子中，除非逐一修改引用count变量的每个地方，否则几乎不可能实现这一点。更糟的是，如果这是你在库中提供给他人使用的项目，你没有办法进行这种修改，只能做出上述艰难的选择。

所以问题来了；公共变量是否有时是好事（或者至少不是坏事）？

我不确定。一方面，你可以看到一些经受住时间考验的公共变量示例（例如：out变量，在System.out中引用）。另一方面，提供公共变量除了极小的开销和可能减少冗长之外，没有其他好处。我的指导原则是，如果你打算将变量设为公共，应该以极端的偏见来评判它是否符合以下标准：

1. 该变量在实现上不应有任何可以想象的理由ever发生变化。这是一个
这非常容易出错（即使你做对了，需求也可能会改变），这就是为什么getter/setter是常用方法的原因。如果
你要有一个公共变量，这真的需要仔细考虑，尤其是在库/框架/API中发布时。
2. 该变量需要被频繁引用，以至于减少冗长带来的微小收益是值得的。我甚至认为使用方法与直接引用之间的
开销不应被考虑。在我保守估计的99.9%的应用中，这种开销微不足道。

可能还有我一时想不到的其他情况。如果你有疑问，始终使用getter/setter。

第3.3节：添加Getter和Setter

封装是面向对象编程的基本概念。它是将数据和代码封装为一个整体。在这种情况下，最好将变量声明为private，然后通过Getter和Setter来访问它们，以查看和/或修改它们。

```
public class Sample {
```

```
    public int count = 0;  
}
```

For all intents and purposes, these two are exactly the same, functionality-wise. The difference between them is the extensibility. Consider what each class says:

- **First:** "I have a method that will give you an **int** value, and a method that will set that value to another **int**".
- **Second:** "I have an **int** that you can set and get as you please."

These might sound similar, but the first is actually much more guarded in its nature; it only lets you interact with its internal nature as **it** dictates. This leaves the ball in its court; it gets to choose how the internal interactions occur. The second has exposed its internal implementation externally, and is now not only prone to external users, but, in the case of an API, **committed** to maintaining that implementation (or otherwise releasing a non-backward-compatible API).

Lets consider if we want to synchronize access to modifying and accessing the count. In the first, this is simple:

```
public class CountHolder {  
    private int count = 0;  
  
    public synchronized int getCount() { return count; }  
    public synchronized void setCount(int c) { count = c; }  
}
```

but in the second example, this is now nearly impossible without going through and modifying each place where the count variable is referenced. Worse still, if this is an item that you're providing in a library to be consumed by others, you do **not** have a way of performing that modification, and are forced to make the hard choice mentioned above.

So it begs the question; are public variables ever a good thing (or, at least, not evil)?

I'm unsure. On one hand, you can see examples of public variables that have stood the test of time (IE: the out variable referenced in [System.out](#)). On the other, providing a public variable gives no benefit outside of extremely minimal overhead and potential reduction in wordiness. My guideline here would be that, if you're planning on making a variable public, you should judge it against these criteria with **extreme** prejudice:

1. The variable should have no conceivable reason to **ever** change in its implementation. This is something that's extremely easy to screw up (and, even if you do get it right, requirements can change), which is why getters/setters are the common approach. If you're going to have a public variable, this really needs to be thought through, especially if released in a library/framework/API.
2. The variable needs to be referenced frequently enough that the minimal gains from reducing verbosity warrants it. I don't even think the overhead for using a method versus directly referencing should be considered here. It's far too negligible for what I'd conservatively estimate to be 99.9% of applications.

There's probably more than I haven't considered off the top of my head. If you're ever in doubt, always use getters/setters.

Section 3.3: Adding Getters and Setters

Encapsulation is a basic concept in OOP. It is about wrapping data and code as a single unit. In this case, it is a good practice to declare the variables as **private** and then access them through Getters and Setters to view and/or modify them.

```
public class Sample {
```

```

private String name;
private int age;

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

```

这些私有变量不能被类外部直接访问。因此它们受到未经授权访问的保护。但如果你想查看或修改它们，可以使
用Getter和Setter方法。

`getXXX()` 方法将返回变量xxx的当前值，而你可以使用`setXXX()`来设置变量xxx的值。

方法的命名规范是（以变量名为variableName为例）：

- 所有非boolean变量

```

getVariableName() //Getter, 变量名应以大写字母开头
setVariableName(..) //Setter, 变量名应以大写字母开头

```

- boolean变量

```

isVariableName() //Getter, 变量名应以大写字母开头
setVariableName(..) //Setter, 变量名应以大写字母开头

```

公共Getter和Setter是Java Bean的Property定义的一部分。

```

private String name;
private int age;

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

```

These private variables cannot be accessed directly from outside the class. Hence they are protected from unauthorized access. But if you want to view or modify them, you can use Getters and Setters.

`getXXX()` method will return the current value of the variable xxx, while you can set the value of the variable xxx using `setXXX()`.

The naming convention of the methods are (in example variable is called `variableName`):

- All non `boolean` variables

```

getVariableName() //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase

```

- `boolean` variables

```

isVariableName() //Getter, The variable name should start with uppercase
setVariableName(..) //Setter, The variable name should start with uppercase

```

Public Getters and Setters are part of the [Property](#) definition of a Java Bean.

第4章：引用数据类型

第4.1节：解引用

解引用通过.操作符进行：

```
Object obj = new Object();
String text = obj.toString(); // 对 'obj' 进行了解引用。
```

解引用遵循存储在引用中的内存地址，定位到实际对象所在的内存位置。
当找到对象后，调用请求的方法（本例中为toString）。

当引用的值为null时，解引用会导致NullPointerException异常：

```
Object obj = null;
obj.toString(); // 执行此语句时抛出NullPointerException异常。
```

null表示值的缺失，即遵循内存地址无效。因此没有对象可供调用请求的方法。

第4.2节：实例化引用类型

```
Object obj = new Object(); // 注意 'new' 关键字
```

其中：

- Object 是一种引用类型。
- obj 是用于存储新引用的变量。
- Object() 是对 Object 构造函数的调用。

发生了什么：

- 为对象分配了内存空间。
- 调用构造函数 Object() 来初始化该内存空间。
- 内存地址存储在 obj 中，因此它 引用 新创建的对象。

这与基本类型不同：

```
int i = 10;
```

实际值10存储在 i中。

Chapter 4: Reference Data Types

Section 4.1: Dereferencing

Dereferencing happens with the . operator:

```
Object obj = new Object();
String text = obj.toString(); // 'obj' is dereferenced.
```

Dereferencing follows the memory address stored in a reference, to the place in memory where the actual object resides. When an object has been found, the requested method is called (toString in this case).

When a reference has the value null, dereferencing results in a NullPointerException:

```
Object obj = null;
obj.toString(); // Throws a NullpointerException when this statement is executed.
```

null indicates the absence of a value, i.e. following the memory address leads nowhere. So there is no object on which the requested method can be called.

Section 4.2: Instantiating a reference type

```
Object obj = new Object(); // Note the 'new' keyword
```

Where:

- Object is a reference type.
- obj is the variable in which to store the new reference.
- Object() is the call to a constructor of Object.

What happens:

- Space in memory is allocated for the object.
- The constructor Object() is called to initialize that memory space.
- The memory address is stored in obj, so that it references the newly created object.

This is different from primitives:

```
int i = 10;
```

Where the actual value 10 is stored in i.

第5章：Java编译器 - 'javac'

第5.1节：'javac'命令 - 入门

简单示例

假设“HelloWorld.java”包含以下Java源代码：

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

(有关上述代码的解释，请参阅Java语言入门。)

我们可以使用以下命令编译上述文件：

```
$ javac HelloWorld.java
```

这将生成一个名为“HelloWorld.class”的文件，然后我们可以按如下方式运行：

```
$ java HelloWorld  
Hello world!
```

该示例的关键点如下：

1. 源文件名 "HelloWorld.java" 必须与源文件中的类名匹配 ... 即HelloWorld。如果它们不匹配时，会出现编译错误。
2. 字节码文件名 "HelloWorld.class" 与类名对应。如果你重命名了 "HelloWorld.class"，运行时会出现错误。
3. 运行 Java 应用程序时使用java命令，传入的是类名，而不是字节码文件名。

带包的示例

大多数实际的 Java 代码使用包来组织类的命名空间，减少类名冲突的风险。

如果我们想在名为com.example的包中声明HelloWorld类，"HelloWorld.java"文件将包含以下 Java 源代码：

```
package com.example;  
  
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

该源代码文件需要存放在与包命名对应的目录结构中。

```
. # 当前目录 (本例中)  
|  
----com  
|
```

Chapter 5: Java Compiler - 'javac'

Section 5.1: The 'javac' command - getting started

Simple example

Assuming that the "HelloWorld.java" contains the following Java source:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

(For an explanation of the above code, please refer to Getting started with Java Language.)

We can compile the above file using this command:

```
$ javac HelloWorld.java
```

This produces a file called "HelloWorld.class", which we can then run as follows:

```
$ java HelloWorld  
Hello world!
```

The key points to note from this example are:

1. The source filename "HelloWorld.java" must match the class name in the source file ... which is HelloWorld. If they don't match, you will get a compilation error.
2. The bytecode filename "HelloWorld.class" corresponds to the classname. If you were to rename the "HelloWorld.class", you would get an error when you tried to run it.
3. When running a Java application using java, you supply the classname NOT the bytecode filename.

Example with packages

Most practical Java code uses packages to organize the namespace for classes and reduce the risk of accidental class name collision.

If we wanted to declare the HelloWorld class in a package call com.example, the "HelloWorld.java" would contain the following Java source:

```
package com.example;  
  
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

This source code file needs to stored in a directory tree whose structure corresponds to the package naming.

```
. # the current directory (for this example)  
|  
----com  
|
```

```
----example  
|  
---HelloWorld.java
```

我们可以使用以下命令编译上述文件：

```
$ javac com/example/HelloWorld.java
```

这将生成一个名为 "com/example/HelloWorld.class" 的文件；即编译后，文件结构应如下所示：
：

```
. # 当前目录 (本例中)  
|  
----com  
|  
----example  
|  
---HelloWorld.java  
---HelloWorld.class
```

然后我们可以按如下方式运行该应用程序：

```
$ java com.example.HelloWorld  
Hello world!
```

从这个例子中还需注意的几点是：

1. 目录结构必须与包名结构相匹配。
2. 运行类时，必须提供完整的类名；即“com.example.HelloWorld”，而不是“HelloWorld”。
3. 你不必在当前目录之外编译和运行Java代码。这里这样做只是为了说明。

使用 'javac' 一次编译多个文件。

如果你的应用程序由多个源代码文件组成（大多数都是！），你可以一次编译一个文件。
或者，你可以通过列出路径名同时编译多个文件：

```
$ javac Foo.java Bar.java
```

或者使用命令行的文件名通配符功能.....

```
$ javac *.java  
$ javac com/example/*.java  
$ javac */**/*.java #仅在 Zsh 或启用 shell 的 globstar 时有效
```

这将分别编译当前目录、“com/example”目录及其子目录中的所有 Java 源文件。第三种方法是将源文件名列表（及编译器选项）写入文件。例如：

```
$ javac @sourcefiles
```

其中 sourcefiles 文件包含：

```
Foo.java  
Bar.java
```

```
----example  
|  
---HelloWorld.java
```

We can compile the above file using this command:

```
$ javac com/example/HelloWorld.java
```

This produces a file called "com/example/HelloWorld.class"; i.e. after compilation, the file structure should look like this:

```
. # the current directory (for this example)  
|  
----com  
|  
----example  
|  
---HelloWorld.java  
---HelloWorld.class
```

We can then run the application as follows:

```
$ java com.example.HelloWorld  
Hello world!
```

Additional points to note from this example are:

1. The directory structure must match the package name structure.
2. When you run the class, the full class name must be supplied; i.e. "com.example.HelloWorld" not "HelloWorld".
3. You don't have to compile and run Java code out of the current directory. We are just doing it here for illustration.

Compiling multiple files at once with 'javac'.

If your application consists of multiple source code files (and most do!) you can compile them one at a time.
Alternatively, you can compile multiple files at the same time by listing the pathnames:

```
$ javac Foo.java Bar.java
```

or using your command shell's filename wildcard functionality

```
$ javac *.java  
$ javac com/example/*.java  
$ javac */**/*.java #Only works on Zsh or with globstar enabled on your shell
```

This will compile all Java source files in the current directory, in the "com/example" directory, and recursively in child directories respectively. A third alternative is to supply a list of source filenames (and compiler options) as a file. For example:

```
$ javac @sourcefiles
```

where the sourcefiles file contains:

```
Foo.java  
Bar.java
```

注意：这种方式适合小型单人项目和一次性程序。对于更复杂的项目，建议选择并使用 Java 构建工具。或者，大多数程序员使用集成了编译器和增量构建“项目”功能的 Java IDE（例如 NetBeans、eclipse、IntelliJ IDEA）。

常用的 'javac' 选项

以下是一些可能对你有用的 javac 命令选项

- -d 选项设置写入 ".class" 文件的目标目录。
- "-sourcepath" 选项设置源代码搜索路径。
- -cp 或者 -classpath 选项设置查找外部和先前编译类的搜索路径。有关类路径及其指定方法的更多信息，请参阅类路径主题。
- -version 选项打印编译器的版本信息。

更完整的编译器选项列表将在另一个示例中描述。

参考文献

javac 命令的权威参考是 Oracle 的 javac 手册页。

第5.2节：为不同版本的Java编译

自Java编程语言（及其运行时）首次公开发布以来，经历了众多变化。这些变化包括：

- Java编程语言的语法和语义变化
- Java标准类库提供的API变化。
- Java（字节码）指令集和类文件格式的变化。

除极少数例外（例如 enum 关键字、某些“内部”类的更改等），这些变化都是向后兼容的。

- 使用较旧版本的Java工具链编译的Java程序可以在较新版本的Java平台上运行，无需重新编译。
- 用较旧版本的Java编写的Java程序可以用新的Java编译器成功编译。

用较新编译器编译旧版Java

如果你需要在较新的Java平台上（重新）编译旧版Java代码以在新平台上运行，通常不需要提供任何特殊的编译选项。在少数情况下（例如，如果你曾使用 enum 作为标识符），你可以使用-source 选项来禁用新语法。例如，给定以下类：

```
public class OldSyntax {
    private static int enum; // 在Java 5或更高版本中无效
}
```

使用Java 5编译器（或更高版本）编译该类时需要如下操作：

```
$ javac -source 1.4 OldSyntax.java
```

为较旧的执行平台编译

如果你需要编译Java以在较旧的Java平台上运行，最简单的方法是安装最旧版本的JDK

Note: compiling code like this is appropriate for small one-person projects, and for once-off programs. Beyond that, it is advisable to select and use a Java build tool. Alternatively, most programmers use a Java IDE (e.g. [NetBeans](#), [eclipse](#), [IntelliJ IDEA](#)) which offers an embedded compiler and incremental building of "projects".

Commonly used 'javac' options

Here are a few options for the javac command that are likely to be useful to you

- The -d option sets a destination directory for writing the ".class" files.
- The -sourcepath option sets a source code search path.
- The -cp or -classpath option sets the search path for finding external and previously compiled classes. For more information on the classpath and how to specify it, refer to the [The Classpath Topic](#).
- The -version option prints the compiler's version information.

A more complete list of compiler options will be described in a separate example.

References

The definitive reference for the javac command is the [Oracle manual page for javac](#).

Section 5.2: Compiling for a different version of Java

The Java programming language (and its runtime) has undergone numerous changes since its release since its initial public release. These changes include:

- Changes in the Java programming language syntax and semantics
- Changes in the APIs provided by the Java standard class libraries.
- Changes in the Java (bytecode) instruction set and classfile format.

With very few exceptions (for example the enum keyword, changes to some "internal" classes, etc), these changes are backwards compatible.

- A Java program that was compiled using an older version of the Java toolchain will run on a newer version Java platform without recompilation.
- A Java program that was written in an older version of Java will compile successfully with a new Java compiler.

Compiling old Java with a newer compiler

If you need to (re-)compile older Java code on a newer Java platform to run on the newer platform, you generally don't need to give any special compilation flags. In a few cases (e.g. if you had used enum as an identifier) you could use the -source option to disable the new syntax. For example, given the following class:

```
public class OldSyntax {
    private static int enum; // invalid in Java 5 or later
}
```

the following is required to compile the class using a Java 5 compiler (or later):

```
$ javac -source 1.4 OldSyntax.java
```

Compiling for an older execution platform

If you need to compile Java to run on an older Java platforms, the simplest approach is to install a JDK for the oldest

您需要支持的版本，并在构建中使用该 JDK 的编译器。

您也可以使用更新的Java编译器进行编译，但过程较为复杂。首先，有一些必须满足的重要前提条件：

- 您正在编译的代码不得使用目标Java版本中不可用的Java语言结构。
- 代码不得依赖于旧平台中不可用的标准Java类、字段、方法等。
- 代码依赖的第三方库也必须为较旧的平台构建，并且在编译时和运行时可用。

在满足前提条件的情况下，您可以使用-target选项重新编译适用于较旧平台的代码。例如，

```
$ javac -target 1.4 SomeClass.java
```

将编译上述类以生成与 Java 1.4 或更高版本 JVM 兼容的字节码。（实际上，-source 选项意味着一个兼容的 -target，因此 `javac -source 1.4 ...` 会有相同的效果。`-source` 和 `-target` 之间的关系在 Oracle 文档中有描述。

话虽如此，如果你只是使用-target或-source，仍然会针对编译器的JDK提供的标准类库进行编译。如果不小心，你可能会得到字节码版本正确，但依赖于不可用API的类。解决方案是使用-bootclasspath选项。例如：

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

将针对另一套运行时库进行编译。如果被编译的类对较新的库有（意外的）依赖，将会导致编译错误。

version you need to support, and use that JDK's compiler in your builds.

You can also compile with a newer Java compiler, but there are complications. First of all, there are some important preconditions that must be satisfied:

- The code you are compiling must not use Java language constructs that were not available in the version of Java that you are targeting.
- The code must not depend on standard Java classes, fields, methods and so on that were not available in the older platforms.
- Third party libraries that the code depends on must also be built for the older platform and available at compile-time and run-time.

Given the preconditions are met, you can recompile code for an older platform using the `-target` option. For example,

```
$ javac -target 1.4 SomeClass.java
```

will compile the above class to produce bytecodes that are compatible with Java 1.4 or later JVM. (In fact, the `-source` option implies a compatible `-target`, so `javac -source 1.4 ...` would have the same effect. The relationship between `-source` and `-target` is described in the Oracle documentation.)

Having said that, if you simply use `-target` or `-source`, you will still be compiling against the standard class libraries provided by the compiler's JDK. If you are not careful, you can end up with classes with the correct bytecode version, but with dependencies on APIs that are not available. The solution is to use the `-bootclasspath` option. For example:

```
$ javac -target 1.4 --bootclasspath path/to/java1.4/rt.jar SomeClass.java
```

will compile against an alternative set of runtime libraries. If the class being compiled has (accidental) dependencies on newer libraries, this will give you compilation errors.

第6章：Java代码文档编写

Java代码的文档通常使用javadoc生成。Javadoc由Sun Microsystems创建，目的是从Java源代码生成HTML格式的API文档。使用HTML格式的好处是可以方便地将相关文档通过超链接连接起来。

第6.1节：从命令行构建Javadoc

许多IDE支持自动从Javadoc生成HTML；一些构建工具（例如Maven和Gradle）也有可以处理HTML生成的插件。

然而，生成Javadoc HTML并不依赖这些工具；这可以通过命令行javadoc工具完成。

该工具最基本的用法是：

```
javadoc JavaFile.java
```

这将从JavaFile.java中的Javadoc注释生成HTML。

命令行工具的一个更实用的用法是，递归读取[源代码目录]中的所有Java文件，为[包名]及其所有子包创建文档，并将生成的HTML放置在[docs-目录]中，命令如下：

```
javadoc -d [docs-目录] -subpackages -sourcepath [源代码目录] [包名]
```

第6.2节：类文档

所有Javadoc注释以块注释开始，紧跟一个星号（/**），并在块注释结束时（*/）终止。每行开头可以有任意空白字符和一个星号；这些在生成文档文件时会被忽略。

```
/**  
 * 本类的简要摘要，以句号结尾。  
 *  
 * 通常在摘要和详细说明之间留一个空行。  
 * 摘要（第一个句号之前的所有内容）用于类或包的  
 * 概览部分。  
 *  
 * 可以使用以下内联标签（非详尽列表）：  
 * {@link some.other.class.Documentation} 用于链接到其他文档或符号 * {@link some.other.  
 class.Documentation Some Display Name} 链接的显示形式可以通过在文档或符号定位器后添加显示名称进行  
自定义  
 * {@code code goes here} 用于格式化为代码  
 * {@literal <>[]()foo} 用于解释为字面文本而不转换为HTML标记  
 * 或其他标签。  
 *  
 * 可选地，以下标签可用于类文档的末尾  
 * （非详尽列表）：  
 *  
 * @author 约翰·多伊  
 * @version 1.0  
 * @since 2015年5月10日  
 * @see some.other.class.Documentation  
 * @deprecated 此类已被 some.other.package.BetterFileReader 替代  
 */
```

Chapter 6: Documenting Java Code

Documentation for java code is often generated using [javadoc](#). Javadoc was created by Sun Microsystems for the purpose of [generating API documentation](#) in HTML format from java source code. Using the HTML format gives the convenience of being able to hyperlink related documents together.

Section 6.1: Building Javadocs From the Command Line

Many IDEs provide support for generating HTML from Javadocs automatically; some build tools ([Maven](#) and [Gradle](#), for example) also have plugins that can handle the HTML creation.

However, these tools are not required to generate the Javadoc HTML; this can be done using the command line javadoc tool.

The most basic usage of the tool is:

```
javadoc JavaFile.java
```

Which will generate HTML from the Javadoc comments in JavaFile.java.

A more practical use of the command line tool, which will recursively read all java files in [source-directory], create documentation for [package.name] and all sub-packages, and place the generated HTML in the [docs-directory] is:

```
javadoc -d [docs-directory] -subpackages -sourcepath [source-directory] [package.name]
```

Section 6.2: Class Documentation

All Javadoc comments begin with a block comment followed by an asterisk（/**）and end when the block comment does（*/）。 Optionally, each line can begin with arbitrary whitespace and a single asterisk; these are ignored when the documentation files are generated.

```
/**  
 * Brief summary of this class, ending with a period.  
 *  
 * It is common to leave a blank line between the summary and further details.  
 * The summary (everything before the first period) is used in the class or package  
 * overview section.  
 *  
 * The following inline tags can be used (not an exhaustive list):  
 * {@link some.other.class.Documentation} for linking to other docs or symbols  
 * {@link some.other.class.Documentation Some Display Name} the link's appearance can be  
 * customized by adding a display name after the doc or symbol locator  
 * {@code code goes here} for formatting as code  
 * {@literal <>[]()foo} for interpreting literal text without converting to HTML markup  
 * or other tags.  
 *  
 * Optionally, the following tags may be used at the end of class documentation  
 * (not an exhaustive list):  
 *  
 * @author John Doe  
 * @version 1.0  
 * @since 5/10/15  
 * @see some.other.class.Documentation  
 * @deprecated This class has been replaced by some.other.package.BetterFileReader  
 */
```

```

* 你也可以使用自定义标签来显示附加信息。
* 使用 @custom.<NAME> 标签和 -tag custom.<NAME>:htmltag:"context"
* 命令行选项，可以创建自定义标签。
*
* 示例自定义标签和生成：
* @custom.updated 2.0
* Javadoc 标记：-tag custom.updated:a:"版本更新于："
* 上述标记将在“版本更新于：”下显示 @custom.updated 的值
*
*/
public class FileReader {
}

```

用于类的相同标签和格式也可以用于枚举和接口。

第6.3节：方法文档

所有Javadoc注释以块注释开始，紧跟一个星号（/**），并在块注释结束时（*/）终止。每行开头可以有任意空白字符和一个星号；这些在生成文档文件时会被忽略。

```

/**
* 方法简要说明，以句号结尾。
*
* 方法的进一步描述及其功能，包括尽可能多的详细信息。
* 适当时候可使用内联标签，如
* {@code 代码示例}、{@link some.other.Docs} 和 {@literal 文本示例}。
*
* 如果方法重写了超类方法，可以使用 {@inheritDoc} 来复制
* 文档
* 来自超类方法
*
* @param stream 描述此参数。包含适当的详细信息
*   参数文档通常像这里一样对齐，但这不是强制的。
*   与其他文档一样，第一句句号之前的内容
*   用作摘要。
*
* @return 描述返回值。包含适当的详细信息
*   返回类型文档通常像这里一样对齐，但这不是强制的。
*   与其他文档一样，第一句句号之前的内容用作
*   摘要。
*
* @throws IOException 描述何时以及为何会抛出此异常。
*   异常文档通常像这里一样对齐，但这不是
*   强制的。
*   与其他文档一样，第一句句号之前的内容
*   用作摘要。
*   除了 @throws，也可以使用 @exception。
*
* @since 2.1.0
* @see some.other.class.Documentation
* @deprecated 描述此方法为何已过时。也可以指定替代方法。
*/
public String[] read(InputStream stream) throws IOException {
    return null;
}

```

```

* You can also have custom tags for displaying additional information.
* Using the @custom.<NAME> tag and the -tag custom.<NAME>:htmltag:"context"
* command line option, you can create a custom tag.
*
* Example custom tag and generation:
* @custom.updated 2.0
* Javadoc flag: -tag custom.updated:a:"Updated in version:"
* The above flag will display the value of @custom.updated under "Updated in version:"
*
*/
public class FileReader {
}

```

The same tags and format used for Classes can be used for Enums and Interfaces as well.

Section 6.3: Method Documentation

All Javadoc comments begin with a block comment followed by an asterisk (/*) and end when the block comment does (*). Optionally, each line can begin with arbitrary whitespace and a single asterisk; these are ignored when the documentation files are generated.

```

/**
* Brief summary of method, ending with a period.
*
* Further description of method and what it does, including as much detail as is
* appropriate. Inline tags such as
* {@code code here}, {@link some.other.Docs}, and {@literal text here} can be used.
*
* If a method overrides a superclass method, {@inheritDoc} can be used to copy the
* documentation
* from the superclass method
*
* @param stream Describe this parameter. Include as much detail as is appropriate
*   Parameter docs are commonly aligned as here, but this is optional.
*   As with other docs, the documentation before the first period is
*   used as a summary.
*
* @return Describe the return values. Include as much detail as is appropriate
*   Return type docs are commonly aligned as here, but this is optional.
*   As with other docs, the documentation before the first period is used as a
*   summary.
*
* @throws IOException Describe when and why this exception can be thrown.
*   Exception docs are commonly aligned as here, but this is
*   optional.
*   As with other docs, the documentation before the first period
*   is used as a summary.
*   Instead of @throws, @exception can also be used.
*
* @since 2.1.0
* @see some.other.class.Documentation
* @deprecated Describe why this method is outdated. A replacement can also be specified.
*/
public String[] read(InputStream stream) throws IOException {
    return null;
}

```

第6.4节：包文档

版本 ≥ Java SE 5

可以使用名为 `package-info.java` 的文件在Javadocs中创建包级文档。该文件必须按以下格式编写。前导空白和星号是可选的，通常每行都会有，用于格式化原因

```
/**  
 * 包文档写在这里；第一个句号之前的任何文档将  
 * 用作摘要。  
 *  
 * 通常在摘要和其余文档之间留一个空行；使用此空白 * 以尽可能详细地描述该包。  
 *  
 * 可以此文档中使用内联标签，如 {@code 代码示例}、{@link reference.to.other.Documentation}，  
 * 和 {@literal 这里的文本}。  
 */  
package com.example.foo;  
  
// 文件的其余部分必须为空。
```

在上述情况下，必须将此文件 `package-info.java` 放在Java包 `com.example.foo` 的文件夹内。

第6.5节：链接

使用@link标签可以链接到其他Javadoc：

```
/**  
 * 你可以使用{@link ClassName}链接到已导入类的Javadoc。  
 *  
 * 如果类尚未导入，也可以使用全限定名：  
 * {@link some.other.ClassName}  
 *  
 * 你可以这样链接到类的成员（字段或方法）：  
 * {@link ClassName#someMethod()}  
 * {@link ClassName#someMethodWithParameters(int, String)}  
 * {@link ClassName#someField}  
 * {@link #someMethodInThisClass()} - 用于链接当前类中的成员  
 *  
 * 你可以这样为链接的Javadoc添加标签：  
 * {@link ClassName#someMethod() link text}  
 */
```

You can link to the javadoc of an already imported class using [ClassName](#).

You can also use the fully-qualified name, if the class is not already imported: [some.other.ClassName](#)

You can link to members (fields or methods) of a class like so:

[ClassName.someMethod\(\)](#)

[ClassName.someMethodWithParameters\(int, String\)](#)

[ClassName.someField](#)

[someMethodInThisClass\(\)](#) - used to link to members in the current class

You can add a label to a linked javadoc like so: [link text](#)

Section 6.4: Package Documentation

Version ≥ Java SE 5

It is possible to create package-level documentation in Javadocs using a file called `package-info.java`. This file must be formatted as below. Leading whitespace and asterisks optional, typically present in each line for formatting reason

```
/**  
 * Package documentation goes here; any documentation before the first period will  
 * be used as a summary.  
 *  
 * It is common practice to leave a blank line between the summary and the rest  
 * of the documentation; use this space to describe the package in as much detail  
 * as is appropriate.  
 *  
 * Inline tags such as {@code code here}, {@link reference.to.other.Documentation}，  
 * and {@literal text here} can be used in this documentation.  
 */  
package com.example.foo;  
  
// The rest of the file must be empty.
```

In the above case, you must put this file `package-info.java` inside the folder of the Java package `com.example.foo`.

Section 6.5: Links

Linking to other Javadocs is done with the @link tag:

```
/**  
 * You can link to the javadoc of an already imported class using {@link ClassName}.  
 *  
 * You can also use the fully-qualified name, if the class is not already imported:  
 * {@link some.other.ClassName}  
 *  
 * You can link to members (fields or methods) of a class like so:  
 * {@link ClassName#someMethod()}  
 * {@link ClassName#someMethodWithParameters(int, String)}  
 * {@link ClassName#someField}  
 * {@link #someMethodInThisClass()} - used to link to members in the current class  
 *  
 * You can add a label to a linked javadoc like so:  
 * {@link ClassName#someMethod() link text}  
 */
```

You can link to the javadoc of an already imported class using [ClassName](#).

You can also use the fully-qualified name, if the class is not already imported: [some.other.ClassName](#)

You can link to members (fields or methods) of a class like so:

[ClassName.someMethod\(\)](#)

[ClassName.someMethodWithParameters\(int, String\)](#)

[ClassName.someField](#)

[someMethodInThisClass\(\)](#) - used to link to members in the current class

You can add a label to a linked javadoc like so: [link text](#)

使用@see标签可以向“参见”部分添加元素。与@param或@return类似，它们出现的位置并不重要。规范建议应写在@return之后。

```
/**  
 * 该方法有一个很好的解释，但你可能会在底部找到更多信息。  
 *  
 * @see ClassName#someMethod()  
 */
```

This method has a nice explanation but you might found further information at the bottom.

See Also:

[ClassName.someMethod\(\)](#)

如果你想添加外部资源链接，你可以直接使用HTML的标签。你可以在任何地方内联使用它，或者在@link和@see标签内使用。

```
/**  
 * 想知道这是如何工作的吗？你可能想  
 * 查看这个很棒的服务。  
 *  
 * @see <a href="http://stackoverflow.com/">Stack Overflow</a>  
 */
```

Wondering how this works? You might want to check this [great service](#).

See Also:

[Stack Overflow](#)

第6.6节：文档中的代码片段

在文档中编写代码的规范方式是使用{@code }结构。如果你有多行代码，请用

```
</pre>
```

包裹。

```
/**  
 * 类 TestUtils.  
 * <p>  
 * 这是一个{@code inline("代码示例")}。  
 * <p>  
 * 编写多行代码时，应将其包裹在pre标签中。  
 * <pre>{@code  
 *   示例 example1 = new FirstLineExample();  
 *   example1.butYouCanHaveMoreThanOneLine();  
 * }</pre>  
 * <p>  
 * 感谢阅读。  
 */  
class TestUtils {
```

有时你可能需要在 javadoc 注释中放入一些复杂代码。符号 @ 特别成问题。使用旧的 `标签配合 {@literal } 结构可以解决这个问题。`

```
/**  
 * 用法：  
 * <pre><code>  
 * class SomethingTest {  
 * {@literal @}Rule
```

With the @see tag you can add elements to the See also section. Like @param or @return the place where they appear is not relevant. The spec says you should write it after @return.

```
/**  
 * This method has a nice explanation but you might found further  
 * information at the bottom.  
 *  
 * @see ClassName#someMethod()  
 */
```

This method has a nice explanation but you might found further information at the bottom.

See Also:

[ClassName.someMethod\(\)](#)

If you want to add **links to external resources** you can just use the HTML tag. You can use it inline anywhere or inside both @link and @see tags.

```
/**  
 * Wondering how this works? You might want  
 * to check this <a href="http://stackoverflow.com/">great service</a>.  
 *  
 * @see <a href="http://stackoverflow.com/">Stack Overflow</a>  
 */
```

Wondering how this works? You might want to check this [great service](#).

See Also:

[Stack Overflow](#)

Section 6.6: Code snippets inside documentation

The canonical way of writing code inside documentation is with the {@code } construct. If you have multiline code wrap inside

```
</pre>
```

```
/**  
 * The Class TestUtils.  
 * <p>  
 * This is an {@code inline("code example")}.  
 * <p>  
 * You should wrap it in pre tags when writing multiline code.  
 * <pre>{@code  
 *   Example example1 = new FirstLineExample();  
 *   example1.butYouCanHaveMoreThanOneLine();  
 * }</pre>  
 * <p>  
 * Thanks for reading.  
 */  
class TestUtils {
```

Sometimes you may need to put some complex code inside the javadoc comment. The @ sign is specially problematic. The use of the old `标签 alongside the {@literal } construct solves the problem.`

```
/**  
 * Usage:  
 * <pre><code>  
 * class SomethingTest {  
 * {@literal @}Rule
```

```

* public SingleTestRule singleTestRule = new SingleTestRule("test1");
*
* {@literal @}Test
* public void test1() {
*     // 只有这个测试会被执行
* }
*
* ...
*
* </code></pre>
*/
class SingleTestRule implements TestRule { }

```

第6.7节：字段文档

所有Javadoc注释以块注释开始，紧跟一个星号（/**），并在块注释结束时（*/）终止。每行开头可以有任意空白字符和一个星号；这些在生成文档文件时会被忽略。

```

/**
* 字段也可以进行文档说明。
*
* 与其他javadoc一样，第一句结束前的内容用作 * 摘要，通常与其余文档之间用空行分隔。
*
*
* 字段文档可以使用内联标签，例如：
* {@code 这里是代码}
* {@literal 这里是文本}
* {@link other.docs.Here}
*
* 字段文档还可以使用以下标签：
*
* @since 2.1.0
* @see some.other.class.Documentation
* @deprecated 说明该字段为何已过时
*/
public static final String CONSTANT_STRING = "foo";

```

第6.8节：内联代码文档

除了Javadoc文档外，代码还可以内联注释。

单行注释以//开始，可以位于同一行语句之后，但不能位于之前。

```

public void method() {
    //单行注释
    someMethodCall(); //语句后的单行注释
}

```

多行注释定义在/*和*/之间。它们可以跨多行，甚至可以位于语句之间。

```

public void method(Object object) {
    /*

```

```

* public SingleTestRule singleTestRule = new SingleTestRule("test1");
*
* {@literal @}Test
* public void test1() {
*     // only this test will be executed
* }
*
* ...
*
* </code></pre>
*/
class SingleTestRule implements TestRule { }

```

Section 6.7: Field Documentation

All Javadoc comments begin with a block comment followed by an asterisk (/**) and end when the block comment does (*). Optionally, each line can begin with arbitrary whitespace and a single asterisk; these are ignored when the documentation files are generated.

```

/**
* Fields can be documented as well.
*
* As with other javadocs, the documentation before the first period is used as a
* summary, and is usually separated from the rest of the documentation by a blank
* line.
*
* Documentation for fields can use inline tags, such as:
* {@code code here}
* {@literal text here}
* {@link other.docs.Here}
*
* Field documentation can also make use of the following tags:
*
* @since 2.1.0
* @see some.other.class.Documentation
* @deprecated Describe why this field is outdated
*/
public static final String CONSTANT_STRING = "foo";

```

Section 6.8: Inline Code Documentation

Apart from the Javadoc documentation code can be documented inline.

Single Line comments are started by // and may be positioned after a statement on the same line, but not before.

```

public void method() {
    //single line comment
    someMethodCall(); //single line comment after statement
}

```

Multi-Line comments are defined between /* and */. They can span multiple lines and may even be positioned between statements.

```

public void method(Object object) {
    /*

```

多行注释

```
/*
对象/*行内注释*.方法();
}
```

JavaDocs 是一种特殊形式的多行注释，以`/**`开头。

由于过多的行内注释可能降低代码的可读性，除非代码不够自解释或设计决策不明显，否则应尽量少用。

单行注释的另一个用例是使用 TAG，即简短的、基于约定的关键字。

一些开发环境会识别此类单行注释的特定约定。常见示例有

- `//TODO`
- `//FIXME`

或者问题引用，例如 Jira 的

- `//PRJ-1234`

```
multi
line
comment
*/
object/*inner-line-comment*.method();
}
```

JavaDocs are a special form of multi-line comments, starting with `/**`.

As too many inline comments may decrease readability of code, they should be used sparsely in case the code isn't self-explanatory enough or the design decision isn't obvious.

An additional use case for single-line comments is the use of TAGs, which are short, convention driven keywords. Some development environments recognize certain conventions for such single-comments. Common examples are

- `//TODO`
- `//FIXME`

Or issue references, i.e. for Jira

- `//PRJ-1234`

第7章：命令行参数处理

参数

详情

命令行参数。假设main方法由Java启动器调用，args将不为null，且不会包含任何**null**元素。

第7.1节：使用GWT ToolBase进行参数处理

如果您想解析更复杂的命令行参数，例如带有可选参数，最好的方法是使用谷歌的GWT方案。所有类均公开可用，地址为：

<https://gwt.googlesource.com/gwt/+/2.8.0-beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java>

处理命令行myprogram -dir "~/Documents" -port 8888的示例是：

```
public class MyProgramHandler extends ToolBase {
    protected File dir;
    protected int port;
    // dir和port的getter方法
    ...

    public MyProgramHandler() {
        this.registerHandler(new ArgHandlerDir() {
            @Override
            public void setDir(File dir) {
                this.dir = dir;
            }

            this.registerHandler(new ArgHandlerInt() {
                @Override
                public String[] getTagArgs() {
                    return new String[]{"port"};
                }
                @Override
                public void setInt(int value) {
                    this.port = value;
                }
            }
        });
        public static void main(String[] args) {
            MyProgramHandler myShell = new MyProgramHandler();
            if (myShell.processArgs(args)) {
                // main program operation
                System.out.println(String.format("port: %d; dir: %s",
                    myShell.getPort(), myShell.getDir()));
            }
            System.exit(1);
        }
    }
}
```

ArgHandler还有一个方法 `isRequired()`，可以被重写以表明命令行参数是必需的（默认返回值为 `false`，因此该参数是可选的）。

第7.2节：手动处理参数

当应用程序的命令行语法很简单时，完全用自定义代码进行命令参数

Chapter 7: Command line Argument Processing

Parameter

Details

args The command line arguments. Assuming that the `main` method is invoked by the Java launcher, args will be non-null, and will have no `null` elements.

Section 7.1: Argument processing using GWT ToolBase

If you want to parse more complex command-line arguments, e.g. with optional parameters, than the best is to use google's GWT approach. All classes are public available at:

<https://gwt.googlesource.com/gwt/+/2.8.0-beta1/dev/core/src/com/google/gwt/util/tools/ToolBase.java>

An example for handling the command-line `myprogram -dir "~/Documents" -port 8888` is:

```
public class MyProgramHandler extends ToolBase {
    protected File dir;
    protected int port;
    // getters for dir and port
    ...

    public MyProgramHandler() {
        this.registerHandler(new ArgHandlerDir() {
            @Override
            public void setDir(File dir) {
                this.dir = dir;
            }

            this.registerHandler(new ArgHandlerInt() {
                @Override
                public String[] getTagArgs() {
                    return new String[]{"port"};
                }
                @Override
                public void setInt(int value) {
                    this.port = value;
                }
            });
        });
        public static void main(String[] args) {
            MyProgramHandler myShell = new MyProgramHandler();
            if (myShell.processArgs(args)) {
                // main program operation
                System.out.println(String.format("port: %d; dir: %s",
                    myShell.getPort(), myShell.getDir()));
            }
            System.exit(1);
        }
    }
}
```

ArgHandler also has a method `isRequired()` which can be overwritten to say that the command-line argument is required (default return is `false` so that the argument is optional).

Section 7.2: Processing arguments by hand

When the command-line syntax for an application is simple, it is reasonable to do the command argument

处理是合理的。

在本例中，我们将展示一系列简单的案例研究。在每种情况下，如果参数不可接受，代码将产生错误信息然后调用 `System.exit(1)` 告诉 shell 命令执行失败。（我们假设每种情况下 Java 代码都是通过名为“myapp”的包装器调用的。）

无参数命令

在本案例中，命令不需要参数。代码演示了 `args.length` 告诉我们命令行参数的数量。

```
public class Main {  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            System.err.println("usage: myapp");  
            System.exit(1);  
        }  
        // 运行应用程序  
        System.out.println("它成功了");  
    }  
}
```

带有两个参数的命令

在本案例研究中，该命令精确要求两个参数。

```
public class Main {  
    public static void main(String[] args) {  
        if (args.length != 2) {  
            System.err.println("用法: myapp <arg1> <arg2>");  
            System.exit(1);  
        }  
        // 运行应用程序  
        System.out.println("它成功了: " + args[0] + ", " + args[1]);  
    }  
}
```

注意，如果我们忽略检查`args.length`，当用户传入的命令行参数过少时，命令将会崩溃。

带有“标志”选项且至少有一个参数的命令

在本案例研究中，该命令有几个（可选的）标志选项，并且在选项之后至少需要一个参数。

```
package tommy;  
public class Main {  
    public static void main(String[] args) {  
        boolean feelMe = false;  
        boolean seeMe = false;  
        int index;  
loop: for (index = 0; index < args.length; index++) {  
        String opt = args[index];  
        switch (opt) {  
        case "-c":  
            seeMe = true;  
            break;  
        case "-f":  
            feelMe = true;  
            break;  
        }  
    }  
    }  
}
```

processing entirely in custom code.

In this example, we will present a series of simple case studies. In each case, the code will produce error messages if the arguments are unacceptable, and then call `System.exit(1)` to tell the shell that the command has failed. (We will assume in each case that the Java code is invoked using a wrapper whose name is "myapp".)

A command with no arguments

In this case-study, the command requires no arguments. The code illustrates that `args.length` gives us the number of command line arguments.

```
public class Main {  
    public static void main(String[] args) {  
        if (args.length > 0) {  
            System.err.println("usage: myapp");  
            System.exit(1);  
        }  
        // Run the application  
        System.out.println("It worked");  
    }  
}
```

A command with two arguments

In this case-study, the command requires at precisely two arguments.

```
public class Main {  
    public static void main(String[] args) {  
        if (args.length != 2) {  
            System.err.println("usage: myapp <arg1> <arg2>");  
            System.exit(1);  
        }  
        // Run the application  
        System.out.println("It worked: " + args[0] + ", " + args[1]);  
    }  
}
```

Note that if we neglected to check `args.length`, the command would crash if the user ran it with too few command-line arguments.

A command with "flag" options and at least one argument

In this case-study, the command has a couple of (optional) flag options, and requires at least one argument after the options.

```
package tommy;  
public class Main {  
    public static void main(String[] args) {  
        boolean feelMe = false;  
        boolean seeMe = false;  
        int index;  
loop: for (index = 0; index < args.length; index++) {  
        String opt = args[index];  
        switch (opt) {  
        case "-c":  
            seeMe = true;  
            break;  
        case "-f":  
            feelMe = true;  
            break;  
        }  
    }  
    }  
}
```

```

feelMe = true;
    break;
default:
    if (!opts.isEmpty() && opts.charAt(0) == '-') {
        error("未知选项: " + opt + "");
    }
    break loop;
}
if (index >= args.length) {
    error("缺少参数");
}

// 运行应用程序
// ...
}

private static void error(String message) {
    if (message != null) {
        System.err.println(message);
    }
    System.err.println("用法: myapp [-f] [-c] [ <arg> ...]");
    System.exit(1);
}

```

如您所见，如果命令语法复杂，处理参数和选项会变得相当繁琐。建议使用“命令行解析”库；请参见其他示例。

```

feelMe = true;
    break;
default:
    if (!opts.isEmpty() && opts.charAt(0) == '-') {
        error("Unknown option: '" + opt + "'");
    }
    break loop;
}
if (index >= args.length) {
    error("Missing argument(s)");
}

// Run the application
// ...
}

private static void error(String message) {
    if (message != null) {
        System.err.println(message);
    }
    System.err.println("usage: myapp [-f] [-c] [ <arg> ...]");
    System.exit(1);
}

```

As you can see, processing the arguments and options gets rather cumbersome if the command syntax is complicated. It is advisable to use a "command line parsing" library; see the other examples.

第8章：Java命令 - 'java' 和 'javaw'

第8.1节：入口点类

Java入口点类具有以下签名和修饰符的main方法：

```
public static void main(String[] args)
```

附注：由于数组的工作方式，也可以写成 (String args[])

当 java 命令启动虚拟机时，它会加载指定的入口类并尝试找到 main 方法。如果成功，命令行传入的参数将被转换为 Java String 对象并组装成数组。如果以这种方式调用 main，数组将 not 是 null，且不会包含任何 null 条目。

一个有效的入口类方法必须满足以下条件：

- 方法名为 main (区分大小写)
- 必须是 **public** 和 **static**
- 返回类型为 **void**
- 有且仅有一个参数，类型为数组 **String[]**。参数必须存在，且不允许有多个参数。
- 不能是泛型方法：不允许有类型参数。
- 拥有一个非泛型的顶层（非嵌套或内部）封闭类

通常将类声明为**public**，但这并非严格必要。从Java 5开始，main 方法的参数类型可以是**String**可变参数，而不是字符串数组。main方法可以选择抛出异常，其参数名称可以是任意的，但惯例上通常为args。

JavaFX入口点

从Java 8开始，java命令也可以直接启动JavaFX应用程序。JavaFX的文档见JavaFX标签，但JavaFX入口点必须满足以下条件：

- 继承**javafx.application.Application**
- 必须是**public**且非**abstract**
- 不能是泛型或嵌套类
- 必须有显式或隐式的**public**无参构造函数

第8.2节：'java'命令故障排除

本示例涵盖使用'java'命令时的常见错误。

“命令未找到”

如果你收到如下错误信息：

```
java: 命令未找到
```

尝试运行java命令时，表示在你的shell命令搜索路径中没有java命令。原因可能是：

Chapter 8: The Java Command - 'java' and 'javaw'

Section 8.1: Entry point classes

A Java entry-point class has a **main** method with the following signature and modifiers:

```
public static void main(String[] args)
```

Sidenote: because of how arrays work, it can also be (String args[])

When the **java** command starts the virtual machine, it loads the specified entry-point classes and tries to find **main**. If successful, the arguments from command line are converted to Java **String** objects and assembled into an array. If **main** is invoked like this, the array will not be **null** and won't contain any **null** entries.

A valid entry-point class method must do the following:

- Be named **main** (case-sensitive)
- Be **public** and **static**
- Have a **void** return type
- Have a single argument with an array **String[]**. The argument must be present and no more than one argument is allowed.
- Be generic: type parameters are not allowed.
- Have a non-generic, top-level (not nested or inner) enclosing class

It is conventional to declare the class as **public** but this not strictly necessary. From Java 5 onward, the **main** method's argument type may be a **String** varargs instead of a string array. **main** can optionally throw exceptions, and its parameter can be named anything, but conventionally it is **args**.

JavaFX entry-points

From Java 8 onwards the **java** command can also directly launch a JavaFX application. JavaFX is documented in the JavaFX tag, but a JavaFX entry-point must do the following:

- Extend **javafx.application.Application**
- Be **public** and not **abstract**
- Not be generic or nested
- Have an explicit or implicit **public** no-args constructor

Section 8.2: Troubleshooting the 'java' command

This example covers common errors with using the 'java' command.

"Command not found"

If you get an error message like:

```
java: command not found
```

when trying to run the **java** command, this means that there is no **java** command on your shell's command search path. The cause could be:

- 你根本没有安装Java JRE或JDK,
- 你没有在shell初始化文件中（正确地）更新PATH环境变量，或者你没有在当前shell中“source”相关的初始化文件。

请参阅“安装Java”了解你需要采取的步骤。

“找不到或无法加载主类”

如果java命令无法找到/加载你指定的入口点类，就会输出此错误信息。一般来说，出现此问题有三大原因：

- 你指定的入口点类不存在。
- 类存在，但你指定错误。
- 该类存在且您已正确指定，但由于类路径不正确，Java无法找到它。

以下是诊断和解决问题的步骤：

1. 找出入口类的全名。

- 如果您有一个类的源代码，那么完整名称由包名和简单类名组成。如果“Main”类的实例声明在包“com.example.myapp”中，那么它的完整名称就是“com.example.myapp.Main”。
- 如果你有一个已编译的类文件，可以通过运行javap命令来查找类名。
- 如果类文件位于某个目录中，您可以从目录名称推断出完整的类名。
- 如果类文件位于 JAR 或 ZIP 文件中，则可以从 JAR 或 ZIP 文件中的文件路径推断出完整的类名。

2. 查看java命令的错误信息。该信息应以完整的类名结尾 java 正在尝试使用。

- 检查它是否与入口类的完整类名完全匹配。
- 它不应以“.java”或“.class”结尾。
- 它不应包含斜杠或任何其他在Java标识符中不合法的字符。
- 名称的大小写应与完整类名完全匹配。

3. 如果您使用的是正确的类名，请确保该类实际上在类路径中：

- 计算类名映射到的路径名；参见“将类名映射到路径名”计算类路径是什么；参见此示例：指定类路径的不同方式查看类路径上的每个JAR和ZIP文件，看看它们是否包含具有所需路径名的类。
- 查看每个目录，看看路径名是否解析为目录中的文件。

如果手动检查类路径未发现问题，您可以添加-Xdiag和-XshowSettings选项。前者列出所有加载的类，后者打印包括JVM有效类路径在内的设置。

最后，还有一些晦涩的原因导致此问题：

- 具有Main-Class属性的可执行JAR文件，该属性指定了不存在的类。
- 具有错误Class-Path属性的可执行JAR文件。
- 如果你在类名之前搞乱了选项，java 命令可能会尝试解释其中一个选项

- you don't have a Java JRE or JDK installed at all,
- you have not updated the PATH environment variable (correctly) in your shell initialization file, or
- you have not "sourced" the relevant initialization file in the current shell.

Refer to “Installing Java” for the steps that you need to take.

“Could not find or load main class”

This error message is output by the java command if it has been unable to find / load the entry-point class that you have specified. In general terms, there are three broad reasons that this can happen:

- You have specified an entry point class that does not exist.
- The class exists, but you have specified it incorrectly.
- The class exists and you have specified it correctly, but Java cannot find it because the classpath is incorrect.

Here is a procedure to diagnose and solve the problem:

1. Find out the full name of the entry-point class.

- If you have source code for a class, then the full name consists of the package name and the simple class name. The instance the “Main” class is declared in the package “com.example.myapp” then its full name is “com.example.myapp.Main”.
- If you have a compiled class file, you can find the class name by running javap on it.
- If the class file is in a directory, you can infer the full class name from the directory names.
- If the class file is in a JAR or ZIP file, you can infer the full class name from the file path in the JAR or ZIP file.

2. Look at the error message from the java command. The message should end with the full class name that java is trying to use.

- Check that it exactly matches the full classname for the entry-point class.
- It should not end with “.java” or “.class”.
- It should not contain slashes or any other character that is not legal in a Java identifier.
- The casing of the name should exactly match the full class name.

3. If you are using the correct classname, make sure that the class is actually on the classpath:

- Work out the pathname that the classname maps to; see Mappingclassnames to pathnames
- Work out what the classpath is; see this example: Different ways to specify the classpath
- Look at each of the JAR and ZIP files on the classpath to see if they contain a class with the required pathname.
- Look at each directory to see if the pathname resolves to a file within the directory.

If checking the classpath by hand did not find the issue, you could add the -Xdiag and -XshowSettings options. The former lists all classes that are loaded, and the latter prints out settings that include the effective classpath for the JVM.

Finally, there are some *obscure* causes for this problem:

- An executable JAR file with a Main-Class attribute that specifies a class that does not exist.
- An executable JAR file with an incorrect Class-Path attribute.
- If you mess up the options before the classname, the java command may attempt to interpret one of them

作为类名。

- 如果有人忽略了Java的风格规则，使用了仅在字母大小写上不同的包或类标识符，并且你运行的平台对文件名的字母大小写不敏感。
- 代码中或命令行中类名的同形异义字问题。

“在类 <name> 中未找到主方法”

当 java 命令能够找到并加载你指定的类，但随后无法找到入口方法时，会出现此问题。

有三种可能的解释：

- 如果你尝试运行可执行的JAR文件，则该JAR的清单文件中“Main-Class”属性错误，指定了一个不是有效入口点类的类。
- 你告诉java命令的类不是入口点类。
- 入口点类不正确；更多信息请参见入口点类。

其他资源

- [“无法找到或加载主类”是什么意思？](#)
- <http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html>

1 - 从Java 8及以后版本开始，java命令会自动将文件名分隔符（“/”或“\”）映射为句点（“.”）。但是，这种行为并未在手册页中记录。

2 - 一个非常隐晦的情况是，如果你从格式化文档中复制粘贴命令，而文本编辑器使用了“长破折号”而非普通连字符。

第8.3节：运行带有库依赖的Java应用程序

这是“主类”和“可执行JAR”示例的延续。

典型的Java应用程序由特定于应用的代码和各种可重用的库代码组成，这些库代码可能是你自己实现的，也可能是第三方实现的。后者通常称为库依赖，通常打包为JAR文件。

Java是一种动态绑定语言。当你运行带有库依赖的Java应用程序时，JVM需要知道依赖的位置，以便按需加载类。大致来说，有两种处理方式：

- 应用程序及其依赖可以重新打包成一个包含所有所需类和资源的单一JAR文件。
- JVM 可以通过运行时类路径指定依赖的 JAR 文件所在位置。

对于可执行的 JAR 文件，运行时类路径由清单属性中的 “Class-Path” 指定。（*编辑注释*：

这应在关于 jar 命令的单独主题中描述。）否则，运行时类路径需要通过 -cp 选项或使用 CLASSPATH 环境变量提供。

例如，假设我们有一个 Java 应用程序，位于名为 "myApp.jar" 的文件中，其入口点类是 com.example.MyApp。假设该应用程序依赖于库 JAR 文件 "lib/library1.jar" 和 "lib/library2.jar"。我们可以在命令行中使用 java 命令如下启动该应用程序：

```
$ # 备选方案 1 (首选)
```

as the classname.

- If someone has ignored Java style rules and used package or class identifiers that differ only in letter case, and you are running on a platform that treats letter case in filenames as non-significant.
- Problems with homoglyphs in class names in the code or on the command line.

“Main method not found in class <name>”

This problem happens when the java command is able to find and load the class that you nominated, but is then unable to find an entry-point method.

There are three possible explanations:

- If you are trying to run an executable JAR file, then the JAR's manifest has an incorrect "Main-Class" attribute that specifies a class that is not a valid entry point class.
- You have told the java command a class that is not an entry point class.
- The entry point class is incorrect; see Entry point classes for more information.

Other Resources

- [What does "Could not find or load main class" mean?](#)
- <http://docs.oracle.com/javase/tutorial/getStarted/problems/index.html>

1 - From Java 8 and later, the java command will helpfully map a filename separator ("/" or "") to a period ("."). However, this behavior is not documented in the manual pages.

2 - A really obscure case is if you copy-and-paste a command from a formatted document where the text editor has used a "long hyphen" instead of a regular hyphen.

Section 8.3: Running a Java application with library dependencies

This is a continuation of the "main class" and "executable JAR" examples.

Typical Java applications consist of an application-specific code, and various reusable library code that you have implemented or that has been implemented by third parties. The latter are commonly referred to as library dependencies, and are typically packaged as JAR files.

Java is a dynamically bound language. When you run a Java application with library dependencies, the JVM needs to know where the dependencies are so that it can load classes as required. Broadly speaking, there are two ways to deal with this:

- The application and its dependencies can be repackaged into a single JAR file that contains all of the required classes and resources.
- The JVM can be told where to find the dependent JAR files via the runtime classpath.

For an executable JAR file, the runtime classpath is specified by the "Class-Path" manifest attribute. (*Editorial Note: This should be described in a separate Topic on the jar command.*) Otherwise, the runtime classpath needs to be supplied using the -cp option or using the CLASSPATH environment variable.

For example, suppose that we have a Java application in the "myApp.jar" file whose entry point class is com.example.MyApp. Suppose also that the application depends on library JAR files "lib/library1.jar" and "lib/library2.jar". We could launch the application using the java command as follows in a command line:

```
$ # Alternative 1 (preferred)
```

```
$ java -cp myApp.jar:lib/library1.jar:lib/library2.jar com.example.MyApp
```

```
$ # 备选方案 2  
$ export CLASSPATH=myApp.jar:lib/library1.jar:lib/library2.jar  
$ java com.example.MyApp
```

(在 Windows 上，类路径分隔符应使用 ; 替代 :，且设置（本地）CLASSPATH 变量时应使用 set 而非 export。)

虽然 Java 开发者对此很熟悉，但这对用户来说并不“友好”。因此，通常的做法是编写一个简单的 shell 脚本（或 Windows 批处理文件）来隐藏用户不需要了解的细节。例如，如果你将以下 shell 脚本保存为名为 "myApp" 的文件，使其可执行，并将其放入命令搜索路径中的某个目录：

```
#!/bin/bash  
# "myApp" 包装脚本  
  
export DIR=/usr/libexec/myApp  
export CLASSPATH=$DIR/myApp.jar:$DIR/lib/library1.jar:$DIR/lib/library2.jar  
java com.example.MyApp
```

然后你可以按如下方式运行它：

```
$ myApp arg1 arg2 ...
```

命令行上的任何参数都会通过"\$@"扩展传递给Java应用程序。（你也可以用Windows批处理文件做类似的事情，虽然语法不同。）

第8.4节：Java选项

java命令支持多种选项：

- 所有选项都以单个连字符 (-) 开头：不支持GNU/Linux中使用--表示“长”选项的约定。
- 选项必须出现在<classname>或-jar <jarfile>参数之前才能被识别。它们之后的任何参数都将被视为传递给正在运行的Java应用程序的参数。
- 不以-X或-XX开头的选项是标准选项。您可以依赖所有Java实现1来支持任何标准选项。
- 以-X开头的选项是非标准选项，可能会在某个Java版本后被取消。
- 以-XX开头的选项是高级选项，也可能被取消。

使用-D设置系统属性

-D<property>=<value>选项用于在系统Properties对象中设置属性。该参数可以重复使用以设置不同的属性。

内存、栈和垃圾回收器选项

控制堆和栈大小的主要选项记录在“设置堆、永久代和栈大小”中。（编辑注：垃圾回收器选项应在同一主题中描述。）

```
$ java -cp myApp.jar:lib/library1.jar:lib/library2.jar com.example.MyApp
```

```
$ # Alternative 2  
$ export CLASSPATH=myApp.jar:lib/library1.jar:lib/library2.jar  
$ java com.example.MyApp
```

(On Windows, you would use ; instead of : as the classpath separator, and you would set the (local) CLASSPATH variable using set rather than export.)

While a Java developer would be comfortable with that, it is not "user friendly". So it is common practice to write a simple shell script (or Windows batch file) to hide the details that the user doesn't need to know about. For example, if you put the following shell script into a file called "myApp", make it executable, and put it into a directory on the command search path:

```
#!/bin/bash  
# The 'myApp' wrapper script  
  
export DIR=/usr/libexec/myApp  
export CLASSPATH=$DIR/myApp.jar:$DIR/lib/library1.jar:$DIR/lib/library2.jar  
java com.example.MyApp
```

then you could run it as follows:

```
$ myApp arg1 arg2 ...
```

Any arguments on the command line will be passed to the Java application via the "\$@" expansion. (You can do something similar with a Windows batch file, though the syntax is different.)

Section 8.4: Java Options

The java command supports a wide range of options:

- All options start with a single hyphen or minus-sign (-): the GNU/Linux convention of using -- for "long" options is not supported.
- Options must appear before the <classname> or the -jar <jarfile> argument to be recognized. Any arguments after them will be treated as arguments to be passed to Java app that is being run.
- Options that do not start with -X or -XX are standard options. You can rely on all Java implementations¹ to support any standard option.
- Options that start with -X are non-standard options, and may be withdrawn from one Java version to the next.
- Options that start with -XX are advanced options, and may also be withdrawn.

Setting system properties with -D

The -D<property>=<value> option is used to set a property in the system [Properties](#) object. This parameter can be repeated to set different properties.

Memory, Stack and Garbage Collector options

The main options for controlling the heap and stack sizes are documented in Setting the Heap, PermGen and Stack sizes. (Editorial note: Garbage Collector options should be described in the same topic.)

启用和禁用断言

-ea和-da选项分别用于启用和禁用Javaassert检查：

- 默认情况下，所有断言检查均被禁用。
- ea选项启用所有断言的检查
- ea:<packagename>...启用对某个包及其所有子包中的断言检查。
- ea:<classname>... 选项启用对某个类中断言的检查。
- da 选项禁用所有断言的检查
- da:<packagename>... 禁用对某个包及其所有子包中断言的检查。
- da:<classname>... 禁用对某个类中断言的检查。
- esa 选项启用对所有系统类的断言检查。
- dsa 选项禁用对所有系统类的断言检查。

这些选项可以组合使用。例如：

```
$ # 启用非系统类中所有断言的检查  
$ java -ea -dsa MyApp
```

```
$ # 启用某个包中所有类的断言，但排除其中一个类。  
$ java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyApp
```

注意，启用断言检查可能会改变Java程序的行为。

- 这通常会使应用程序变慢。
- 它可能导致特定方法运行时间变长，从而改变多线程应用中线程的时序。
- 它可能引入偶然的happens-before关系，导致内存异常消失。
- 错误实现的assert语句可能产生不良副作用。

选择虚拟机类型

选项-client和-server允许你在两种不同形式的HotSpot虚拟机之间选择：

- “client”形式针对用户应用进行了优化，启动更快。
- “server”形式针对长时间运行的应用进行了优化。在JVM“预热”期间收集统计信息时间更长，这使得JIT编译器能够更好地优化本地代码。

默认情况下，JVM会根据平台能力尽可能以64位模式运行。选项-d32和-d64允许你显式选择模式。

1 - 查阅java命令的官方手册。有时某个standard选项会被描述为“可能变更”。

第8.5节：参数中的空格和其他特殊字符

首先，处理参数中空格的问题实际上不是Java的问题。相反，这是一个需要由你运行Java程序时所使用的命令行解释器来处理的问题。

举个例子，假设我们有以下简单程序，用于打印文件的大小：

```
import java.io.File;  
  
public class PrintFileSizes {
```

Enabling and disabling assertions

The -ea and -da options respectively enable and disable Java **assert** checking:

- All assertion checking is disabled by default.
- The -ea option enables checking of all assertions
- The -ea:<packagename>... enables checking of assertions in a package *and all subpackages*.
- The -ea:<classname>... enables checking of assertions in a class.
- The -da option disables checking of all assertions
- The -da:<packagename>... disables checking of assertions in a package *and all subpackages*.
- The -da:<classname>... disables checking of assertions in a class.
- The -esa option enables checking for all system classes.
- The -dsa option disables checking for all system classes.

The options can be combined. For example.

```
$ # Enable all assertion checking in non-system classes  
$ java -ea -dsa MyApp  
  
$ # Enable assertions for all classes in a package except for one.  
$ java -ea:com.wombat.fruitbat... -da:com.wombat.fruitbat.Brickbat MyApp
```

Note that enabling to assertion checking is liable to alter the behavior of a Java programming.

- It is liable make the application slower in general.
- It can cause specific methods to take longer to run, which could change timing of threads in a multi-threaded application.
- It can introduce serendipitous *happens-before* relations which can cause memory anomalies to disappear.
- An incorrectly implemented **assert** statement could have unwanted side-effects.

Selecting the VM type

The -client and -server options allow you to select between two different forms of the HotSpot VM:

- The “client” form is tuned for user applications and offers faster startup.
- The “server” form is tuned for long running applications. It takes longer capturing statistic during JVM “warm up” which allows the JIT compiler to do a better of job of optimizing the native code.

By default, the JVM will run in 64bit mode if possible, depending on the capabilities of the platform. The -d32 and -d64 options allow you to select the mode explicitly.

1 - Check the official manual for the java command. Sometimes a *standard* option is described as “subject to change”.

Section 8.5: Spaces and other special characters in arguments

First of all, the problem of handling spaces in arguments is NOT actually a Java problem. Rather it is a problem that needs to be handled by the command shell that you are using when you run a Java program.

As an example, let us suppose that we have the following simple program that prints the size of a file:

```
import java.io.File;  
  
public class PrintFileSizes {
```

```

public static void main(String[] args) {
    for (String name: args) {
        File file = new File(name);
        System.out.println("Size of '" + file + "' is " + file.size());
    }
}

```

现在假设我们想打印一个路径名中带有空格的文件的大小；例如 /home/steve/Test File.txt。如果我们这样运行命令：

```
$ java PrintFileSizes /home/steve/Test File.txt
```

命令行解释器不会知道 /home/steve/Test File.txt 实际上是一个路径名。相反，它会将两个不同的参数传递给 Java 应用程序，Java 程序会尝试查找它们各自的文件大小，但会失败，因为这些路径的文件（很可能）不存在。

使用 POSIX shell 的解决方案

POSIX shell 包括 sh 以及其派生版本如 bash 和 ksh。如果你正在使用这些 shell 之一，那么你可以通过 引用 参数来解决问题。

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

路径名周围的双引号告诉 shell 应该将其作为单个参数传递。传递时引号会被去除。还有其他几种方法可以做到这一点：

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

单引号（直引号）的处理方式类似于双引号，但它们还会抑制参数内的各种扩展。

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

反斜杠转义后面的空格，使其不被解释为参数分隔符。

有关更全面的文档，包括如何处理参数中的其他特殊字符，请参阅 Bash 文档中的引用主题。

Windows 解决方案

Windows 的根本问题在于操作系统层面，参数作为单个字符串（source）传递给子进程。这意味着解析（或重新解析）命令行的最终责任落在程序或其运行时库上。存在很多不一致的情况。

在 Java 的情况下，长话短说：

- 你可以在 java 命令的参数周围加上双引号，这样就可以传递带有空格的参数。
- 显然，java 命令本身会解析命令字符串，并且解析得差不多正确但是，当你尝试将其与批处理文件中的 SET 和变量替换结合使用时，是否会移除双引号就变得非常复杂了。

```

public static void main(String[] args) {
    for (String name: args) {
        File file = new File(name);
        System.out.println("Size of '" + file + "' is " + file.size());
    }
}

```

Now suppose that we want print the size of a file whose pathname has spaces in it; e.g. /home/steve/Test File.txt. If we run the command like this:

```
$ java PrintFileSizes /home/steve/Test File.txt
```

the shell won't know that /home/steve/Test File.txt is actually one pathname. Instead, it will pass 2 distinct arguments to the Java application, which will attempt to find their respective file sizes, and fail because files with those paths (probably) do not exist.

Solutions using a POSIX shell

POSIX shells include sh as well derivatives such as bash and ksh. If you are using one of these shells, then you can solve the problem by *quoting* the argument.

```
$ java PrintFileSizes "/home/steve/Test File.txt"
```

The double-quotes around the pathname tell the shell that it should be passed as a single argument. The quotes will be removed when this happens. There are a couple of other ways to do this:

```
$ java PrintFileSizes '/home/steve/Test File.txt'
```

Single (straight) quotes are treated like double-quotes except that they also suppress various expansions within the argument.

```
$ java PrintFileSizes /home/steve/Test\ File.txt
```

A backslash escapes the following space, and causes it not to be interpreted as an argument separator.

For more comprehensive documentation, including descriptions of how to deal with other special characters in arguments, please refer to the quoting topic in the Bash documentation.

Solution for Windows

The fundamental problem for Windows is that at the OS level, the arguments are passed to a child process as a single string ([source](#)). This means that the ultimate responsibility of parsing (or re-parsing) the command line falls on either program or its runtime libraries. There is lots of inconsistency.

In the Java case, to cut a long story short:

- You can put double-quotes around an argument in a java command, and that will allow you to pass arguments with spaces in them.
- Apparently, the java command itself is parsing the command string, and it gets it more or less right
- However, when you try to combine this with the use of SET and variable substitution in a batch file, it gets really complicated as to whether double-quotes get removed.

- cmd.exe shell显然有其他的转义机制；例如，双倍双引号和使用^转义。

更多细节请参阅批处理文件文档。

第8.6节：运行可执行的JAR文件

可执行的JAR文件是将Java代码组装成单个可执行文件的最简单方式。*(编辑注：JAR文件的创建应由单独的主题覆盖。)*

假设你有一个路径为<jar-path>的可执行JAR文件，你应该可以按如下方式运行它：

```
java -jar <jar-path>
```

如果命令需要命令行参数，请在<jar-path>后添加它们。例如：

```
java -jar <jar-path> arg1 arg2 arg3
```

如果需要在java命令行上提供额外的JVM选项，这些选项需要放在-jar选项之前。
请注意，如果使用-jar选项，-cp / -classpath选项将被忽略。应用程序的类路径由JAR文件清单决定。

第8.7节：通过“main”类运行Java应用程序

当应用程序未打包为可执行JAR时，需要在java命令行上提供入口点类的名称。

运行HelloWorld类

“HelloWorld”示例在创建新Java程序中有描述。它由一个名为HelloWorld的单个类组成，满足入口点的要求。

假设（已编译的）“HelloWorld.class”文件在当前目录中，可以按如下方式启动：

```
java HelloWorld
```

需要注意的一些重要事项是：

- 我们必须提供类的名称：而不是“.class”文件或“.java”文件的路径名。
- 如果类声明在一个包中（大多数Java类都是如此），那么我们提供给java命令的类名必须是完整的类名。例如，如果SomeClass声明在com.example包中，那么完整的类名将是com.example.SomeClass。

指定类路径

除非我们使用java-jar命令语法，否则java命令通过搜索类路径来查找要加载的类；参见类路径。上述命令依赖于默认类路径是（或包含）当前目录。我们可以通过使用

-cp选项更明确地指定要使用的类路径。

```
java -cp . HelloWorld
```

这表示将当前目录（即“.”所指的目录）作为类路径上的唯一条目。

- The cmd.exe shell apparently has other escaping mechanisms; e.g. doubling double-quotes, and using ^ escapes.

For more detail, please refer to the Batch-File documentation.

Section 8.6: Running an executable JAR file

Executable JAR files are the simplest way to assemble Java code into a single file that can be executed. *(Editorial Note: Creation of JAR files should be covered by a separate Topic.) *

Assuming that you have an executable JAR file with pathname <jar-path>, you should be able to run it as follows:

```
java -jar <jar-path>
```

If the command requires command-line arguments, add them after the <jar-path>. For example:

```
java -jar <jar-path> arg1 arg2 arg3
```

If you need to provide additional JVM options on the java command line, they need to go *before* the -jar option.
Note that a -cp / -classpath option will be ignored if you use -jar. The application's classpath is determined by the JAR file manifest.

Section 8.7: Running a Java applications via a "main" class

When an application has not been packaged as an executable JAR, you need to provide the name of an entry-point class on the java command line.

Running the HelloWorld class

The “HelloWorld” example is described in Creating a new Java program . It consists of a single class called HelloWorld which satisfies the requirements for an entry-point.

Assuming that the (compiled) “HelloWorld.class” file is in the current directory, it can be launched as follows:

```
java HelloWorld
```

Some important things to note are:

- We must provide the name of the class: not the pathname for the ".class" file or the ".java" file.
- If the class is declared in a package (as most Java classes are), then the class name we supply to the java command must be the full classname. For instance if SomeClass is declared in the com.example package, then the full classname will be com.example.SomeClass.

Specifying a classpath

Unless we are using in the java -jar command syntax, the java command looks for the class to be loaded by searching the classpath; see The Classpath. The above command is relying on the default classpath being (or including) the current directory. We can be more explicit about this by specifying the classpath to be used using the -cp option.

```
java -cp . HelloWorld
```

This says to make the current directory (which is what “.” refers to) the sole entry on the classpath.

-cp是由java命令处理的一个选项。所有针对java命令的选项应放在类名之前。类名之后的任何内容都将被视为Java应用程序的命令行参数，并作为传递给main方法的String[]传入应用程序。

(如果未提供-cp选项，java将使用CLASSPATH环境变量指定的类路径。如果该变量未设置或为空，java将使用"."作为默认类路径。)

The -cp is an option that is processed by the java command. All options that are intended for the java command should be before the classname. Anything after the class will be treated as a command line argument for the Java application, and will be passed to application in the `String[]` that is passed to the `main` method.

(If no -cp option is provided, the java will use the classpath that is given by the CLASSPATH environment variable. If that variable is unset or empty, java uses "." as the default classpath.)

第9章：字面量

Java 字面量是一个语法元素（即你在 Java 程序的源代码中找到的东西），它表示一个值。示例有 1、0.333F、false、"Hello world"。

第9.1节：使用下划线提高可读性

自 Java 7 起，可以使用一个或多个下划线（_）来分隔原始数字字面量中的数字组，以提高其可读性。

例如，以下两个声明是等价的：

```
版本 ≥ Java SE 7
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

这可以应用于所有原始数字字面量，如下所示：

```
版本 ≥ Java SE 7
byte color = 1_2_3;
short yearsAnnoDomini= 2_016;
int socialSecurityNumber = 999_99_9999;
long creditCardNumber = 1234_5678_9012_3456L;
float piFourDecimals = 3.14_15F;
double piTenDecimals = 3.14_15_92_65_35;
```

这也适用于二进制、八进制和十六进制的前缀：

```
版本 ≥ Java SE 7
short binary= 0b0_1_0_1;
int octal = 07_7_7_7_7_7_7_0;
long hexBytes = 0xFF_EC_DE_5E;
```

关于下划线有一些规则，禁止它们出现在以下位置：

- 数字的开头或结尾（例如 _123 或 123_ 都无效）
- 浮点字面量的小数点相邻位置（例如 1._23 或 1_.23 都无效）
- F 或 L 后缀之前（例如 1.23_F 或 9999999_L 都无效）
- 在期望数字串的位置（例如 0xFFFF 是无效）

第9.2节：十六进制、八进制和二进制字面量

十六进制数字是基数为16的值。有16个数字，0-9 和字母 A-F（大小写无关）。A-F 代表 10-15。

八进制数字是基数为8的值，使用数字 0-7。

二进制数字是基数为2的数值，使用数字0和1。

所有这些数字的结果都是相同的值，110：

```
int dec = 110;          // 无前缀 --> 十进制字面量
int bin = 0b1101110;    // '0b' 前缀 --> 二进制字面量
int oct = 0156;         // '0' 前缀 --> 八进制字面量
```

Chapter 9: Literals

A Java literal is a syntactic element (i.e. something you find in the *source code* of a Java program) that represents a value. Examples are 1, 0.333F, false, "Hello world\n".

Section 9.1: Using underscore to improve readability

Since Java 7 it has been possible to use one or more underscores (_) for separating groups of digits in a primitive number literal to improve their readability.

For instance, these two declarations are equivalent:

```
Version ≥ Java SE 7
int i1 = 123456;
int i2 = 123_456;
System.out.println(i1 == i2); // true
```

This can be applied to all primitive number literals as shown below:

```
Version ≥ Java SE 7
byte color = 1_2_3;
short yearsAnnoDomini= 2_016;
int socialSecurityNumber = 999_99_9999;
long creditCardNumber = 1234_5678_9012_3456L;
float piFourDecimals = 3.14_15F;
double piTenDecimals = 3.14_15_92_65_35;
```

This also works using prefixes for binary, octal and hexadecimal bases:

```
Version ≥ Java SE 7
short binary= 0b0_1_0_1;
int octal = 07_7_7_7_7_7_0;
long hexBytes = 0xFF_EC_DE_5E;
```

There are a few rules about underscores which **forbid** their placement in the following places:

- At the beginning or end of a number (e.g. _123 or 123_ are *not* valid)
- Adjacent to a decimal point in a floating point literal (e.g. 1._23 or 1_.23 are *not* valid)
- Prior to an F or L suffix (e.g. 1.23_F or 9999999_L are *not* valid)
- In positions where a string of digits is expected (e.g. 0_xFFFF is *not* valid)

Section 9.2: Hexadecimal, Octal and Binary literals

A hexadecimal number is a value in base-16. There are 16 digits, 0-9 and the letters A-F (case does not matter). A-F represent 10-15.

An octal number is a value in base-8, and uses the digits 0-7.

A binary number is a value in base-2, and uses the digits 0 and 1.

All of these numbers result in the same value, 110:

```
int dec = 110;          // no prefix --> decimal literal
int bin = 0b1101110;    // '0b' prefix --> binary literal
int oct = 0156;         // '0' prefix --> octal literal
```

```
int hex = 0x6E; // '0x' 前缀 -> 十六进制字面量
```

注意，二进制字面量语法是在 Java 7 中引入的。

八进制字面量很容易成为语义错误的陷阱。如果你在十进制字面量前加上前导 '`0`'，你将得到错误的值：

```
int a = 0100; // 不是 100, 而是 a == 64
```

第9.3节：布尔字面量

布尔字面量是 Java 编程语言中最简单的字面量。两个可能的 `boolean` 值分别由字面量 `true` 和 `false` 表示。这些是区分大小写的。例如：

```
boolean flag = true; // 使用 'true' 字面量  
flag = false; // 使用 'false' 字面量
```

第9.4节：字符串字面量

字符串字面量提供了在 Java 源代码中表示字符串值的最便捷方式。一个字符串字面量

由以下部分组成：

- 一个开头的双引号 ("") 字符。
- 零个或多个既不是双引号也不是换行符的其他字符。（反斜杠 (\) 字符会改变后续字符的含义；参见字面量中的转义序列。）
- 一个结尾的双引号字符。

例如：

```
"Hello world" // 表示一个11个字符的字符串字面量"" // 表示一个空字符串(长度为零)的字面量|"" // 表示一个由一个双引号字符组成的字符串字面量  
"123" // 另一个包含转义序列的字面量
```

注意，单个字符串字面量不能跨越多行源代码。在字面量的结束双引号之前出现换行符（或源文件结尾）将导致编译错误。例如：

```
"Jello world // 编译错误(在行尾!)
```

长字符串

如果您需要一个过长而无法放在一行的字符串，传统的表达方式是将其拆分成多个字面量，并使用连接运算符 (+) 将各部分连接起来。例如

```
字符串 typingPractice = "快速的棕色狐狸" +  
    "跳过了" +  
    "懒狗"
```

像上面这样的由字符串字面量和 + 组成的表达式满足成为常量表达式的要求。这意味着该表达式将由编译器计算，并在运行时由单个 `String` 对象表示。

字符串字面量的驻留

```
int hex = 0x6E; // '0x' prefix -> hexadecimal literal
```

Note that binary literal syntax was introduced in Java 7.

The octal literal can easily be a trap for semantic errors. If you define a leading '`0`' to your decimal literals you will get the wrong value:

```
int a = 0100; // Instead of 100, a == 64
```

Section 9.3: Boolean literals

Boolean literals are the simplest of the literals in the Java programming language. The two possible `boolean` values are represented by the literals `true` and `false`. These are case-sensitive. For example:

```
boolean flag = true; // using the 'true' literal  
flag = false; // using the 'false' literal
```

Section 9.4: String literals

String literals provide the most convenient way to represent string values in Java source code. A String literal consists of:

- An opening double-quote ("") character.
- Zero or more other characters that are neither a double-quote or a line-break character. (A backslash (\) character alters the meaning of subsequent characters; see Escape sequences in literals.)
- A closing double-quote character.

For example:

```
"Hello world" // A literal denoting an 11 character String  
"" // A literal denoting an empty (zero length) String  
"\\" // A literal denoting a String consisting of one  
// double quote character  
"1\t2\t3\n" // Another literal with escape sequences
```

Note that a single string literal may not span multiple source code lines. It is a compilation error for a line-break (or the end of the source file) to occur before a literal's closing double-quote. For example:

```
"Jello world // Compilation error (at the end of the line!)
```

Long strings

If you need a string that is too long to fit on a line, the conventional way to express it is to split it into multiple literals and use the concatenation operator (+) to join the pieces. For example

```
String typingPractice = "The quick brown fox" +  
    "jumped over" +  
    "the lazy dog"
```

An expression like the above consisting of string literals and + satisfies the requirements to be a Constant Expression. That means that the expression will be evaluated by the compiler and represented at runtime by a single `String` object.

Interning of string literals

当包含字符串字面量的类文件被 JVM 加载时，相应的 String 对象会被运行时系统驻留。这意味着在多个类中使用的字符串字面量所占用的空间不会超过在一个类中使用时的空间。

有关驻留和字符串池的更多信息，请参阅字符串主题中的字符串池和堆存储示例。

第9.5节：空字面量

空字面量（写作 null）表示空类型的唯一值。以下是一些示例

```
MyClass 对象 = null;
MyClass[] 对象 = new MyClass[]{new MyClass(), null, new MyClass()};

myMethod(null);

if (objects != null) {
    // 执行某些操作
}
```

null 类型相当特殊。它没有名称，因此你无法在 Java 源代码中表达它。（而且它在运行时也没有表示。）

null 类型的唯一目的是作为 null 的类型。它与所有引用类型赋值兼容，并且可以被强制转换为任何引用类型。（在后者情况下，强制转换不涉及运行时类型检查。）

最后，null 具有这样的属性：null instanceof <某个引用类型> 总是计算为 false，无论该类型是什么。

When class file containing string literals is loaded by the JVM, the corresponding String objects are *interned* by the runtime system. This means that a string literal used in multiple classes occupies no more space than if it was used in one class.

For more information on interning and the string pool, refer to the String pool and heap storage example in the Strings topic.

Section 9.5: The Null literal

The Null literal (written as null) represents the one and only value of the null type. Here are some examples

```
MyClass object = null;
MyClass[] objects = new MyClass[]{new MyClass(), null, new MyClass()};

myMethod(null);

if (objects != null) {
    // Do something
}
```

The null type is rather unusual. It has no name, so you cannot express it in Java source code. (And it has no runtime representation either.)

The sole purpose of the null type is to be the type of null. It is assignment compatible with all reference types, and can be type cast to any reference type. (In the latter case, the cast does not entail a runtime type check.)

Finally, null has the property that null instanceof <SomeReferenceType> will evaluate to false, no matter what the type is.

第9.6节：字面量中的转义序列

字符串和字符字面量提供了一种转义机制，允许表达字面量中本不允许出现的字符代码。转义序列由反斜杠字符 (\) 后跟一个或多个其他字符组成。相同的序列在字符和字符串字面量中均有效。

完整的转义序列集合如下：

转义序列	含义
\\	表示反斜杠 (\) 字符
'	表示单引号 (') 字符
\"	表示双引号 (") 字符
\n	表示换行符 (LF) 字符
\r	表示回车符 (CR) 字符
\f	表示水平制表符 (HT) 字符
\t	表示换页符 (FF) 字符
\b	表示退格符 (BS) 字符
\<octal>	表示范围在0到255之间的字符代码。

上述的<octal>由一位、两位或三位八进制数字 ('0'到'7') 组成，表示一个介于0到255（十进制）之间的数字。

注意，反斜杠后跟随的任何其他字符都是无效的转义序列。无效的转义序列被JLS视为编译错误。

Escape sequence	Meaning
\\	Denotes an backslash (\) character
'	Denotes a single-quote (') character
\"	Denotes a double-quote (") character
\n	Denotes a line feed (LF) character
\r	Denotes a carriage return (CR) character
\t	Denotes a horizontal tab (HT) character
\f	Denotes a form feed (FF) character
\b	Denotes a backspace (BS) character
\<octal>	Denotes a character code in the range 0 to 255.

The <octal> in the above consists of one, two or three octal digits ('0' through '7') which represent a number between 0 and 255 (decimal).

Note that a backslash followed by any other character is an invalid escape sequence. Invalid escape sequences are treated as compilation errors by the JLS.

- [JLS 3.10.6. 字符和字符串字面量的转义序列](#)

Unicode 转义

除了上述描述的字符串和字符转义序列外，Java还有一种更通用的Unicode转义机制，如JLS 3.3. Unicode Escapes中定义的。Unicode转义的语法如下：

```
'\u' <十六进制数字> <十六进制数字> <十六进制数字> <十六进制数字>
```

其中<十六进制数字>是以下之一：'0'、'1'、'2'、'3'、'4'、'5'、'6'、'7'、'8'、'9'、'a'、'b'、'c'、'd'、'e'、'f'、'D'、'E'、'F'。

Java编译器将Unicode转义映射为一个字符（严格来说是一个16位的Unicode代码单元），并且可以在源代码中任何映射字符有效的位置使用。它通常用于字符和字符串字面量中，当你需要在字面量中表示非ASCII字符时使用。

正则表达式中的转义

待定

第9.7节：字符字面量

字符字面量提供了在Java源代码中表达char值的最便捷方式。一个字符字面量由以下部分组成：

- 一个开头的单引号（'）字符。
- 一个字符的表示。该表示不能是单引号或换行符，但可以是由反斜杠（\）引入的转义序列；详见字面量中的转义序列。
- 一个闭合的单引号（'）字符。

例如：

```
char a = 'a';
char doubleQuote = "";
char singleQuote = '\'';
```

字符字面量中的换行符是编译错误：

```
char newline =
// 上一行编译错误char newLine = ""; // 正确
```

第9.8节：十进制整数字面量

整数字面量提供可以用于byte、short、int、long或char实例的值。（本例侧重于简单的十进制形式。其他示例解释了如何使用八进制、十六进制和二进制的字面量，以及使用下划线提高可读性。）

普通整数字面量

最简单且最常见的整数字面量形式是十进制整数字面量。例如：

```
0 // 十进制数字零 (类型 'int')
```

Reference:

- [JLS 3.10.6. Escape Sequences for Character and String Literals](#)

Unicode escapes

In addition to the string and character escape sequences described above, Java has a more general Unicode escaping mechanism, as defined in [JLS 3.3. Unicode Escapes](#). A Unicode escape has the following syntax:

```
'\u' <hex-digit> <hex-digit> <hex-digit> <hex-digit>
```

where <hex-digit> is one of '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C', 'D', 'E', 'F'.

A Unicode escape is mapped by the Java compiler to a character (strictly speaking a 16-bit Unicode *code unit*), and can be used anywhere in the source code where the mapped character is valid. It is commonly used in character and string literals when you need to represent a non-ASCII character in a literal.

Escaping in regexes

TBD

Section 9.7: Character literals

Character literals provide the most convenient way to express `char` values in Java source code. A character literal consists of:

- An opening single-quote (') character.
- A representation of a character. This representation cannot be a single-quote or a line-break character, but it can be an escape sequence introduced by a backslash (\) character; see Escape sequences in literals.
- A closing single-quote (') character.

For example:

```
char a = 'a';
char doubleQuote = "";
char singleQuote = '\'';
```

A line-break in a character literal is a compilation error:

```
char newline =
// Compilation error in previous line
char newLine = '\n'; // Correct
```

Section 9.8: Decimal Integer literals

Integer literals provide values that can be used where you need a `byte`, `short`, `int`, `long` or `char` instance. (This example focuses on the simple decimal forms. Other examples explain how to literals in octal, hexadecimal and binary, and the use of underscores to improve readability.)

Ordinary integer literals

The simplest and most common form of integer literal is a decimal integer literal. For example:

```
0 // The decimal number zero (type 'int')
```

```
1 // 十进制数字一      (类型 'int')
42 // 十进制数字四十二 (类型 'int')
```

你需要注意前导零。前导零会导致整数字面量被解释为八进制而非十进制。

```
077 // 这个字面量实际上表示  $7 \times 8 + 7 \dots$  即十进制的63!
```

整数字面量是无符号的。如果你看到类似-10或+10的表达式，它们实际上是使用一元-和一元+运算符的表达式。

这种形式的整数字面量的内在类型是int，且必须在零到 2^{31} 或21亿4748万3648之间。

注意 2^{31} 比Integer.MAX_VALUE大1。字面量从0到2147483647可以在任何地方使用，但使用2147483648而没有前导一元-运算符会导致编译错误。（换句话说，它被保留用于表示Integer.MIN_VALUE的值。）

```
int max = 2147483647; // 正确
int min = -2147483648; // 正确
int tooBig = 2147483648; // 错误
```

长整数字面量

类型为long的字面量通过添加L后缀来表示。例如：

```
0L // 十进制数字零      (类型 'long')
1L // 十进制数字一      (类型 'long')
2147483648L // Integer.MAX_VALUE + 1 的值

long big = 2147483648; // 错误
long big2 = 2147483648L; // 正确
```

请注意，int 和 long 字面量之间的区别在其他地方也很重要。例如

```
int i = 2147483647;
long l = i + 1; // 产生负值, 因为操作是
                // 使用32位算术执行的, 且
                // 加法溢出
long l2 = i + 1L; // 产生(直观上)正确的值。
```

参考：[JLS 3.10.1 - 整数字面量](#)

第9.9节：浮点字面量

浮点字面量提供可用于需要float或double实例的值。有三种浮点字面量类型。

- 简单十进制形式
- 缩放十进制形式
- 十六进制形式

(JLS语法规则将两种小数形式合并为一种形式。为了便于说明，我们将它们分开处理。)

```
1 // The decimal number one      (type 'int')
42 // The decimal number forty two (type 'int')
```

You need to be careful with leading zeros. A leading zero causes an integer literal to be interpreted as octal not decimal.

```
077 // This literal actually means  $7 \times 8 + 7 \dots$  or 63 decimal!
```

Integer literals are unsigned. If you see something like -10 or +10, these are actually expressions using the unary - and unary + operators.

The range of integer literals of this form have an intrinsic type of int, and must fall in the range zero to 2^{31} or 2,147,483,648.

Note that 2^{31} is 1 greater than Integer.MAX_VALUE. Literals from 0 through to 2147483647 can be used anywhere, but it is a compilation error to use 2147483648 without a preceding unary - operator. (In other words, it is reserved for expressing the value of Integer.MIN_VALUE.)

```
int max = 2147483647; // OK
int min = -2147483648; // OK
int tooBig = 2147483648; // ERROR
```

Long integer literals

Literals of type long are expressed by adding an L suffix. For example:

```
0L // The decimal number zero      (type 'long')
1L // The decimal number one      (type 'long')
2147483648L // The value of Integer.MAX_VALUE + 1

long big = 2147483648; // ERROR
long big2 = 2147483648L; // OK
```

Note that the distinction between int and long literals is significant in other places. For example

```
int i = 2147483647;
long l = i + 1; // Produces a negative value because the operation is
                // performed using 32 bit arithmetic, and the
                // addition overflows
long l2 = i + 1L; // Produces the (intuitively) correct value.
```

Reference：[JLS 3.10.1 - Integer Literals](#)

Section 9.9: Floating-point literals

Floating point literals provide values that can be used where you need a float or double instance. There are three kinds of floating point literal.

- Simple decimal forms
- Scaled decimal forms
- Hexadecimal forms

(The JLS syntax rules combine the two decimal forms into a single form. We treat them separately for ease of explanation.)

对于float和double字面量，有不同的字面类型，通过后缀表示。各种形式使用字母来表达不同的含义。这些字母不区分大小写。

简单的十进制形式

最简单的浮点字面量形式由一个或多个十进制数字和一个小数点(.)组成，后面可跟一个可选后缀(f、F、d或D)。可选后缀允许你指定字面量是float(f或F)还是double(d或D)值。默认情况下(未指定后缀时)为double。

例如

```
0.0    // 这表示零  
.0    // 这也表示零  
0.    // 这也表示零  
3.14159 // 这表示圆周率，精确到(大约!)小数点后5位。  
1.0F   // 一个`float`字面量  
1.0D   // 一个`double`字面量。(如果没有后缀，默认是`double`)
```

实际上，带有后缀的十进制数字也是浮点字面量。

```
1F    // 与1.0F含义相同
```

十进制字面量的含义是IEEE浮点数，它是与十进制浮点形式所表示的无限精度数学实数最接近的值。该概念值通过“舍入到最近值”转换为IEEE二进制浮点表示。(十进制转换的精确定义见Double.valueOf(String)和Float.valueOf(String)的javadoc，需注意两者的数字语法存在差异。)

带标度的十进制形式

带标度的十进制形式由简单的十进制数和由E或e引入的指数部分组成，指数部分后跟带符号的整数。指数部分是将十进制数乘以十的幂的简写，如下面的示例所示。还有一个可选的后缀用来区分float和double字面量。以下是一些示例：

```
1.0E1    // 表示1.0 x 10^1 ... 即10.0 (double)  
1E-1D    // 表示1.0 x 10^{(-1)} ... 即0.1 (double)  
1.0e10f  // 表示1.0 x 10^{(10)} ... 即1000000000.0 (float)
```

字面量的大小受表示范围限制(float或double)。如果标度因子导致值过大或过小，则为编译错误。

十六进制形式

从Java 6开始，可以用十六进制表示浮点字面量。十六进制形式的语法与简单和缩放的十进制形式类似，但有以下区别：

1. 每个十六进制浮点字面量都以零(0)开头，随后是一个x或X。
2. 数字部分(但不包括指数部分！)还包括十六进制数字a到f以及它们的大写等价字符。
3. 指数是必需的，由字母p(或P)引入，而不是e或E。指数表示的是以2为底的幂的缩放因子，而不是以10为底的幂。

以下是一些示例：

There are distinct literal types for **float** and **double** literals, expressed using suffixes. The various forms use letters to express different things. These letters are case insensitive.

Simple decimal forms

The simplest form of floating point literal consists of one or more decimal digits and a decimal point(.) and an optional suffix(f、F、d或D)。The optional suffix allows you to specify that the literal is a **float**(f or F) or **double**(d or D) value. The default (when no suffix is specified) is **double**.

For example

```
0.0    // this denotes zero  
.0    // this also denotes zero  
0.    // this also denotes zero  
3.14159 // this denotes Pi, accurate to (approximately!) 5 decimal places.  
1.0F   // a `float` literal  
1.0D   // a `double` literal. (`double` is the default if no suffix is given)
```

In fact, decimal digits followed by a suffix is also a floating point literal.

```
1F    // means the same thing as 1.0F
```

The meaning of a decimal literal is the IEEE floating point number that is closest to the infinite precision mathematical Real number denoted by the decimal floating point form. This conceptual value is converted to IEEE binary floating point representation using *round to nearest*. (The precise semantics of decimal conversion are specified in the javadocs for [Double.valueOf\(String\)](#) and [Float.valueOf\(String\)](#), bearing in mind that there are differences in the number syntaxes.)

Scaled decimal forms

Scaled decimal forms consist of simple decimal with an exponent part introduced by an E or e, and followed by a signed integer. The exponent part is a short hand for multiplying the decimal form by a power of ten, as shown in the examples below. There is also an optional suffix to distinguish **float** and **double** literals. Here are some examples:

```
1.0E1    // this means 1.0 x 10^1 ... or 10.0 (double)  
1E-1D    // this means 1.0 x 10^{(-1)} ... or 0.1 (double)  
1.0e10f  // this means 1.0 x 10^{(10)} ... or 1000000000.0 (float)
```

The size of a literal is limited by the representation(**float**或**double**)。It is a compilation error if the scale factor results in a value that is too large or too small.

Hexadecimal forms

Starting with Java 6, it is possible to express floating point literals in hexadecimal. The hexadecimal form have an analogous syntax to the simple and scaled decimal forms with the following differences:

1. Every hexadecimal floating point literal starts with a zero(0) and then an x or X.
2. The digits of the number(但not the exponent part!) also include the hexadecimal digits a through f and their uppercase equivalents.
3. The exponent is *mandatory*, and is introduced by the letter p(或P) instead of an e or E. The exponent represents a scaling factor that is a power of 2 instead of a power of 10.

Here are some examples:

```
0x0.0p0f // 这是以十六进制形式表示的零 (`float')
0xff.0p19 // 这是 255.0 x 2^19 (`double')
```

建议：由于大多数Java程序员对十六进制浮点形式不熟悉，建议尽量少用它们。

下划线

从Java 7开始，允许在三种浮点字面量的数字字符串中使用下划线。这也适用于“指数”部分。参见使用下划线提高可读性。

特殊情况

如果浮点字面量表示的数字过大或过小，无法用所选表示法表示，即数字会溢出为+INF或-INF，或下溢为0.0，则编译时会出错。然而，字面量表示非零非规格化数是合法的。

浮点字面量语法不提供IEEE 754特殊值（如INF和NaN）的字面量表示。如果需要在源代码中表达它们，推荐的方式是使用java.lang.Float和java.lang.Double中定义的常量；例如Float.NaN、Float.NEGATIVE_INFINITY和Float.POSITIVE_INFINITY。

```
0x0.0p0f // this is zero expressed in hexadecimal form (`float')
0xff.0p19 // this is 255.0 x 2^19 (`double')
```

Advice: since hexadecimal floating-point forms are unfamiliar to most Java programmers, it is advisable to use them sparingly.

Underscores

Starting with Java 7, underscores are permitted within the digit strings in all three forms of floating point literal. This applies to the "exponent" parts as well. See Using underscores to improve readability.

Special cases

It is a compilation error if a floating point literal denotes a number that is too large or too small to represent in the selected representation; i.e. if the number would overflow to +INF or -INF, or underflow to 0.0. However, it is legal for a literal to represent a non-zero denormalized number.

The floating point literal syntax does not provide literal representations for IEEE 754 special values such as the INF and NaN values. If you need to express them in source code, the recommended way is to use the constants defined by the `java.lang.Float` and `java.lang.Double`; e.g. `Float.NaN`, `Float.NEGATIVE_INFINITY` and `Float.POSITIVE_INFINITY`.

第10章：原始数据类型

8种原始数据类型byte、short、int、long、char、boolean、float和double是Java程序中存储大多数原始数值数据的类型。

第10.1节：char原始类型

char可以存储单个16位Unicode字符。字符字面量用单引号括起来

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

它的最小值为\u0000（十进制表示中的0，也称为空字符），最大值为\uffff（65,535）。

char 的默认值是 \u0000。

```
char defaultChar; // defaultChar == \u0000
```

为了定义值为 ' 的 char，必须使用转义序列（以反斜杠开头的字符）：

```
char singleQuote = '\'';
```

还有其他转义序列：

```
char tab = '\t';
char backspace = '\b';
char newline = '\n';
char carriageReturn = '\r';
char formfeed = '\f';

char singleQuote = '\'';
char doubleQuote = '\"'; // 这里转义是多余的；"" 也可以；但仍然允许
char backslash = '\\';
char unicodeChar = '\uXXXX' // XXXX 代表你想显示的字符的 Unicode 值
```

你可以声明任何 Unicode 字符的 char。

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // 打印一行包含 "♥" 的内容。
```

也可以对char进行加法操作。例如，要遍历所有小写字母，可以这样做：

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

第10.2节：原始类型速查表

显示所有原始类型大小和取值范围的表格：

数据类型	数值表示	取值范围	默认值
------	------	------	-----

Chapter 10: Primitive Data Types

The 8 primitive data types **byte**, **short**, **int**, **long**, **char**, **boolean**, **float**, and **double** are the types that store most raw numerical data in Java programs.

Section 10.1: The char primitive

A **char** can store a single 16-bit Unicode character. A character literal is enclosed in single quotes

```
char myChar = 'u';
char myChar2 = '5';
char myChar3 = 65; // myChar3 == 'A'
```

It has a minimum value of \u0000 (0 in the decimal representation, also called the *null character*) and a maximum value of \uffff (65,535).

The default value of a **char** is \u0000.

```
char defaultChar; // defaultChar == \u0000
```

In order to define a char of ' value an escape sequence (character preceded by a backslash) has to be used:

```
char singleQuote = '\'';
```

There are also other escape sequences:

```
char tab = '\t';
char backspace = '\b';
char newline = '\n';
char carriageReturn = '\r';
char formfeed = '\f';
char singleQuote = '\'';
char doubleQuote = '\"'; // escaping redundant here; "" would be the same; however still allowed
char backslash = '\\';
char unicodeChar = '\uXXXX' // XXXX represents the Unicode-value of the character you want to display
```

You can declare a **char** of any Unicode character.

```
char heart = '\u2764';
System.out.println(Character.toString(heart)); // Prints a line containing "♥".
```

It is also possible to add to a **char**. e.g. to iterate through every lower-case letter, you could do the following:

```
for (int i = 0; i <= 26; i++) {
    char letter = (char) ('a' + i);
    System.out.println(letter);
}
```

Section 10.2: Primitive Types Cheatsheet

Table showing size and values range of all primitive types:

data type numeric representation	range of values	default value
----------------------------------	-----------------	---------------

布尔型	n/a	false 和 true	false
字节	8位有符号	-27 到 27 - 1 -128 到 +127	0
短整型	16位有符号	-215 到 215 - 1 -32,768 到 +32,767	0
整型	32位有符号	-231 到 231 - 1 -2,147,483,648 到 +2,147,483,647	0
长整型	64位有符号	-2的63次方 到 2的63次方减1 -9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	0L
浮点数	32位浮点数	1.401298464e-45 到 3.402823466e+38 (正负均可)	0.0F
双精度浮点数	64位浮点数	4.94065645841246544e-324d 到 1.79769313486231570e+308d (正负均可)	0.0D
字符	16位无符号	0 到 216 - 1 0 到 65,535	0

备注：

- Java语言规范规定，有符号整型（byte 到 long）使用二进制补码表示法
补码表示，浮点类型使用标准的IEEE 754二进制浮点表示。
- Java 8及更高版本提供了对int和long执行无符号算术运算的方法。虽然这些
这些方法允许程序将相应类型的值视为无符号，但这些类型仍然是有符号类型。
- 上面显示的最小浮点数是次正规数；即它们的精度低于正规值。
最小的正规数是1.175494351e-38和2.2250738585072014e-308
- A **字符**通常表示一个Unicode / UTF-16代码单元。
- 虽然boolean只包含一位信息，但其在内存中的大小取决于Java
虚拟机实现（参见boolean类型）。

第10.3节：float原始类型

float是单精度32位IEEE 754浮点数。默认情况下，小数被解释为double类型。
要创建float，只需在小数后添加一个f。

```
double doubleExample = 0.5;      // 数字后无'f' = double
float floatExample = 0.5f;        // 数字后有'f' = float

float myFloat = 92.7f;           // 这是一个float...
float positiveFloat = 89.3f;     // 它可以是正数,
float negativeFloat = -89.3f;    // 也可以是负数
float integerFloat = 43.0f;      // 它可以是整数 (不是int)
float underZeroFloat = 0.0549f;  // 它可以是小于0的分数值
```

float支持五种常见的算术运算：加法、减法、乘法、除法和取模。

注意：以下结果可能因浮点误差而略有不同。部分结果已为清晰和易读起见进行了四舍五入（例如，加法示例的打印结果实际上是34.600002）。

```
// 加法
float result = 37.2f + -2.6f; // 结果: 34.6

// 减法
float result = 45.1f - 10.3f; // 结果: 34.8
```

boolean	n/a	false and true	false
byte	8-bit signed	-27 to 27 - 1 -128 to +127	0
short	16-bit signed	-215 to 215 - 1 -32,768 to +32,767	0
int	32-bit signed	-231 to 231 - 1 -2,147,483,648 to +2,147,483,647	0
long	64-bit signed	-263 to 263 - 1 -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
float	32-bit floating point	1.401298464e-45 to 3.402823466e+38 (positive or negative)	0.0F
double	64-bit floating point	4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)	0.0D
char	16-bit unsigned	0 to 216 - 1 0 to 65,535	0

Notes:

- The Java Language Specification mandates that signed integral types (**byte** through **long**) use binary two-complement representation, and the floating point types use standard IEEE 754 binary floating point representations.
- Java 8 and later provide methods to perform unsigned arithmetic operations on **int** and **long**. While these methods allow a program to treat values of the respective types as unsigned, the types remain signed types.
- The smallest floating point shown above are *subnormal*; i.e. they have less precision than a *normal* value. The smallest normal numbers are 1.175494351e-38 and 2.2250738585072014e-308
- A **char** conventionally represents a Unicode / UTF-16 code unit.
- Although a **boolean** contains just one bit of information, its size in memory varies depending on the Java Virtual Machine implementation (see [boolean type](#)).

Section 10.3: The float primitive

A **float** is a single-precision 32-bit IEEE 754 floating point number. By default, decimals are interpreted as doubles.
To create a **float**, simply append an f to the decimal literal.

```
double doubleExample = 0.5;      // without 'f' after digits = double
float floatExample = 0.5f;        // with 'f' after digits = float

float myFloat = 92.7f;           // this is a float...
float positiveFloat = 89.3f;     // it can be positive,
float negativeFloat = -89.3f;    // or negative
float integerFloat = 43.0f;      // it can be a whole number (not an int)
float underZeroFloat = 0.0549f;  // it can be a fractional value less than 0
```

FLOATS handle the five common arithmetical operations: addition, subtraction, multiplication, division, and modulus.

Note: The following may vary slightly as a result of floating point errors. Some results have been rounded for clarity and readability purposes (i.e. the printed result of the addition example was actually 34.600002).

```
// addition
float result = 37.2f + -2.6f; // result: 34.6

// subtraction
float result = 45.1f - 10.3f; // result: 34.8
```

```
// 乘法
float result = 26.3f * 1.7f; // 结果: 44.71

// 除法
float result = 37.1f / 4.8f; // 结果: 7.729166

// 取模
float result = 37.1f % 4.8f; // 结果: 3.4999971
```

由于浮点数的存储方式（即二进制形式），许多数字没有精确的表示。

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

虽然使用float对于大多数应用来说是可以的，但float和double都不应该用于存储十进制数字（如货币金额）或需要更高精度的数字的精确表示。
相反，应该使用BigDecimal类。

float的默认值是0.0f。

```
float defaultFloat; // defaultFloat == 0.0f
```

float的精度大约是百万分之一的误差。

注意：Float.POSITIVE_INFINITY、Float.NEGATIVE_INFINITY、Float.NaN都是float类型的值。NaN表示无法确定的运算结果，例如两个无限值相除。此外，0f和-0f是不同的，但==比较结果为true：

```
float f1 = 0f;
float f2 = -0f;
System.out.println(f1 == f2); // true
System.out.println(1f / f1); // Infinity
System.out.println(1f / f2); // -Infinity
System.out.println(Float.POSITIVE_INFINITY / Float.POSITIVE_INFINITY); // NaN
```

第10.4节：int原始类型

原始数据类型如int直接将值存储在使用它的变量中，而使用Integer声明的变量则保存对该值的引用。

根据[java API](#)：“Integer类将原始类型int的值封装在一个对象中。类型为Integer的对象包含一个类型为int的单一字段。”

默认情况下，int是一个32位有符号整数。它可以存储的最小值为 -2^{31} ，最大值为 $2^{31} - 1$ 。

```
int 示例 = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

如果需要存储超出此范围的数字，应使用long。超过int的值范围会导致整数溢出，使超出范围的值被加到范围的另一端（正数变为负数，反之亦然）。该值为 $((value - MIN_VALUE) \% RANGE) + MIN_VALUE$ ，或 $((value$

```
// multiplication
float result = 26.3f * 1.7f; // result: 44.71

// division
float result = 37.1f / 4.8f; // result: 7.729166

// modulus
float result = 37.1f % 4.8f; // result: 3.4999971
```

Because of the way floating point numbers are stored (i.e. in binary form), many numbers don't have an exact representation.

```
float notExact = 3.1415926f;
System.out.println(notExact); // 3.1415925
```

While using **float** is fine for most applications, neither **float** nor **double** should be used to store exact representations of decimal numbers (like monetary amounts), or numbers where higher precision is required. Instead, the **BigDecimal** class should be used.

The default value of a **float** is 0.0f.

```
float defaultFloat; // defaultFloat == 0.0f
```

A **float** is precise to roughly an error of 1 in 10 million.

Note: `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY`, `Float.NaN` are **float** values. NaN stands for results of operations that cannot be determined, such as dividing 2 infinite values. Furthermore 0f and -0f are different, but == yields true:

```
float f1 = 0f;
float f2 = -0f;
System.out.println(f1 == f2); // true
System.out.println(1f / f1); // Infinity
System.out.println(1f / f2); // -Infinity
System.out.println(Float.POSITIVE_INFINITY / Float.POSITIVE_INFINITY); // NaN
```

Section 10.4: The int primitive

A primitive data type such as **int** holds values directly into the variable that is using it, meanwhile a variable that was declared using **Integer** holds a reference to the value.

According to [java API](#): "The Integer class wraps a value of the primitive type int in an object. An object of type Integer contains a single field whose type is int."

By default, **int** is a 32-bit signed integer. It can store a minimum value of -231, and a maximum value of 231 - 1.

```
int example = -42;
int myInt = 284;
int anotherInt = 73;

int addedInts = myInt + anotherInt; // 284 + 73 = 357
int subtractedInts = myInt - anotherInt; // 284 - 73 = 211
```

If you need to store a number outside of this range, **long** should be used instead. Exceeding the value range of **int** leads to an integer overflow, causing the value exceeding the range to be added to the opposite site of the range (positive becomes negative and vice versa). The value is $((value - MIN_VALUE) \% RANGE) + MIN_VALUE$, or $((value$

```
+ 2147483648) % 4294967296) - 2147483648
```

```
int demo = 2147483647; //最大正整数  
System.out.println(demo); //打印2147483647  
demo = demo + 1; //导致整数溢出  
System.out.println(demo); //打印-2147483648
```

int的最大值和最小值可以在以下位置找到：

```
int high = Integer.MAX_VALUE; // high == 2147483647  
int low = Integer.MIN_VALUE; // low == -2147483648
```

一个int的默认值是0

```
int defaultInt; // defaultInt == 0
```

第10.5节：转换原语

在 Java 中，我们可以在整数值和浮点值之间进行转换。此外，由于每个字符在 Unicode 编码中对应一个数字，char 类型可以与整数和浮点类型相互转换。

boolean 是唯一一种不能转换为其他任何原始数据类型，也不能从其他原始数据类型转换而来的原始数据类型。

有两种类型的转换：扩展转换和缩小转换。

拓宽转换是指将一种数据类型的值转换为占用比前者更多位数的另一种数据类型的值。在这种情况下不存在数据丢失的问题。

相应地，缩小转换是指将一种数据类型的值转换为另一种占用更少位数的数据类型的值。在这种情况下可能会发生数据丢失。

Java 会自动执行拓宽转换。但如果你想执行缩窄转换（如果你确定不会发生数据丢失），那么你可以使用一种称为强制类型转换的语言结构来强制 Java 执行该转换。

拓宽转换：

```
int a = 1;  
double d = a; // 有效的转换为 double, 无需强制类型转换 (拓宽转换)
```

缩窄转换：

```
double d = 18.96  
int b = d; // 无效的转换为 int, 会导致编译时错误  
int b = (int) d; // 有效的转换为 int, 但结果被截断 (向下取整)  
// 这就是类型强制转换  
// 现在, b = 18
```

第10.6节：原始类型与装箱原始类型的内存消耗

原始类型 装箱类型 原始类型/装箱类型的内存大小

原始类型	装箱类型	原始类型/装箱类型的内存大小
boolean	Boolean	1 字节 / 16 字节
byte	字节	1 字节 / 16 字节
短整型	短整型	2 字节 / 16 字节

```
+ 2147483648) % 4294967296) - 2147483648
```

```
int demo = 2147483647; //maximum positive integer  
System.out.println(demo); //prints 2147483647  
demo = demo + 1; //leads to an integer overflow  
System.out.println(demo); // prints -2147483648
```

The maximum and minimum values of **int** can be found at:

```
int high = Integer.MAX_VALUE; // high == 2147483647  
int low = Integer.MIN_VALUE; // low == -2147483648
```

The default value of an **int** is 0

```
int defaultInt; // defaultInt == 0
```

Section 10.5: Converting Primitives

In Java, we can convert between integer values and floating-point values. Also, since every character corresponds to a number in the Unicode encoding, **char** types can be converted to and from the integer and floating-point types. **boolean** is the only primitive datatype that cannot be converted to or from any other primitive datatype.

There are two types of conversions: *widening conversion* and *narrowing conversion*.

A *widening conversion* is when a value of one datatype is converted to a value of another datatype that occupies more bits than the former. There is no issue of data loss in this case.

Correspondingly, A *narrowing conversion* is when a value of one datatype is converted to a value of another datatype that occupies fewer bits than the former. Data loss can occur in this case.

Java performs *widening conversions* automatically. But if you want to perform a *narrowing conversion* (if you are sure that no data loss will occur), then you can force Java to perform the conversion using a language construct known as a *cast*.

Widening Conversion:

```
int a = 1;  
double d = a; // valid conversion to double, no cast needed (widening)
```

Narrowing Conversion:

```
double d = 18.96  
int b = d; // invalid conversion to int, will throw a compile-time error  
int b = (int) d; // valid conversion to int, but result is truncated (gets rounded down)  
// This is type-casting  
// Now, b = 18
```

Section 10.6: Memory consumption of primitives vs. boxed primitives

Primitive Boxed Type Memory Size of primitive / boxed

Primitive	Boxed Type	Memory Size of primitive / boxed
boolean	Boolean	1 byte / 16 bytes
byte	Byte	1 byte / 16 bytes
short	Short	2 bytes / 16 bytes

字符	字符	2 字节 / 16 字节
整型	整型	4 字节 / 16 字节
长整型	长整型	8 字节 / 16 字节
浮点数	浮点数	4字节 / 16字节
double	双精度浮点数	8字节 / 16字节

装箱对象总是需要8字节用于类型和内存管理，并且由于对象的大小总是8的倍数，装箱类型总共需要16字节。此外，每次使用装箱对象都需要存储一个引用，根据JVM及其选项不同，这个引用占用4或8字节。

在数据密集型操作中，内存消耗对性能有重大影响。使用数组时内存消耗会更大：一个float[5]数组只需32字节；而一个存储5个不同非空值的Float[5]数组总共需要112字节（在64位无压缩指针环境下，这个数字增加到152字节）。

装箱值缓存

装箱类型的空间开销可以通过装箱值缓存在一定程度上缓解。一些装箱类型实现了实例缓存。例如，默认情况下，Integer类会缓存表示范围在-128到+127之间的数字的实例。但这并不能减少由额外内存间接访问带来的额外开销。

如果你通过自动装箱或调用静态方法valueOf(primitive)创建装箱类型实例，运行时系统会尝试使用缓存值。如果你的应用大量使用缓存范围内的值，这可以显著减少使用装箱类型的内存开销。当然，如果你是“手动”创建装箱值实例，最好使用valueOf而不是new。（new操作总是创建一个新实例。）但如果大多数值不在缓存范围内，调用new并跳过缓存查找可能更快。

第10.7节：双精度原始类型

A double 是一种双精度64位IEEE 754浮点数。

```
double example = -7162.37;
double myDouble = 974.21;
double anotherDouble = 658.7;

double addedDoubles = myDouble + anotherDouble; // 315.51
double subtractedDoubles = myDouble - anotherDouble; // 1632.91

double scientificNotationDouble = 1.2e-3; // 0.0012
```

由于浮点数的存储方式，许多数字没有精确的表示。

```
double notExact = 1.32 - 0.42; // 结果应为0.9
System.out.println(notExact); // 0.9000000000000001
```

虽然使用double适用于大多数应用，但无论是float还是double都不应被用来存储精确数字如货币。应使用BigDecimal类代替

double的默认值是0.0d

```
public double defaultDouble; // defaultDouble == 0.0
```

char	Char	2 bytes / 16 bytes
int	Integer	4 bytes / 16 bytes
long	Long	8 bytes / 16 bytes
float	Float	4 bytes / 16 bytes
double	Double	8 bytes / 16 bytes

Boxed objects always require 8 bytes for type and memory management, and because the size of objects is always a multiple of 8, boxed types *all require 16 bytes total*. In addition, each usage of a boxed object entails storing a reference which accounts for another 4 or 8 bytes, depending on the JVM and JVM options.

In data-intensive operations, memory consumption can have a major impact on performance. Memory consumption grows even more when using arrays: a `float[5]` array will require only 32 bytes; whereas a `Float[5]` storing 5 distinct non-null values will require 112 bytes total (on 64 bit without compressed pointers, this increases to 152 bytes).

Boxed value caches

The space overheads of the boxed types can be mitigated to a degree by the boxed value caches. Some of the boxed types implement a cache of instances. For example, by default, the `Integer` class will cache instances to represent numbers in the range -128 to +127. This does not, however, reduce the additional cost arising from the additional memory indirection.

If you create an instance of a boxed type either by autoboxing or by calling the static `valueOf(primitive)` method, the runtime system will attempt to use a cached value. If your application uses a lot of values in the range that is cached, then this can substantially reduce the memory penalty of using boxed types. Certainly, if you are creating boxed value instances "by hand", it is better to use `valueOf` rather than `new`. (The `new` operation always creates a new instance.) If, however, the majority of your values are *not* in the cached range, it can be faster to call `new` and save the cache lookup.

Section 10.7: The double primitive

A double is a double-precision 64-bit IEEE 754 floating point number.

```
double example = -7162.37;
double myDouble = 974.21;
double anotherDouble = 658.7;

double addedDoubles = myDouble + anotherDouble; // 315.51
double subtractedDoubles = myDouble - anotherDouble; // 1632.91

double scientificNotationDouble = 1.2e-3; // 0.0012
```

Because of the way floating point numbers are stored, many numbers don't have an exact representation.

```
double notExact = 1.32 - 0.42; // result should be 0.9
System.out.println(notExact); // 0.9000000000000001
```

While using double is fine for most applications, neither float nor double should be used to store precise numbers such as currency. Instead, the BigDecimal class should be used

The default value of a double is 0.0d

```
public double defaultDouble; // defaultDouble == 0.0
```

注意：Double.POSITIVE_INFINITY、Double.NEGATIVE_INFINITY、Double.NaN 是double类型的值。NaN表示无法确定的运算结果，例如两个无限值相除。此外，0d和-0d是不同的，但==的结果为真：

```
double d1 = 0d;
double d2 = -0d;
System.out.println(d1 == d2); // true
System.out.println(1d / d1); // Infinity
System.out.println(1d / d2); // -Infinity
System.out.println(Double.POSITIVE_INFINITY / Double.POSITIVE_INFINITY); // NaN
```

第10.8节：long原始类型

默认情况下，long 是64位有符号整数（在Java 8中，它可以是有符号或无符号）。作为有符号类型，它可以存储的最小值为 -2^{63} ，最大值为 $2^{63} - 1$ ；作为无符号类型，它可以存储的最小值为0，最大值为 $2^{64} - 1$

```
long example = -42;
long myLong = 284;
long anotherLong = 73;

//必须在数字末尾添加"L"，因为默认情况下
//数字被假定为int类型。添加"L"使其成为long类型
//由于549755813888 ( $2^{39}$ ) 大于int的最大值 ( $2^{31} - 1$ ) ，
//必须添加"L"
long bigNumber = 549755813888L;

long addedLongs = myLong + anotherLong; // 284 + 73 = 357
long subtractedLongs = myLong - anotherLong; // 284 - 73 = 211
```

long 的最大值和最小值可以在以下位置找到：

```
long high = Long.MAX_VALUE; // high == 9223372036854775807L
long low = Long.MIN_VALUE; // low == -9223372036854775808L
```

long 的默认值是 0L

```
long defaultLong; // defaultLong == 0L
```

注意：在 long 字面量末尾添加的字母 "L" 大小写不敏感，但建议使用大写，因为这样更容易与数字1区分：

```
2L == 2l; // true
```

警告：Java 缓存范围在 -128 到 127 之间的 Integer 对象实例。原因解释见此处：

https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

可以得到以下结果：

```
Long val1 = 127L;
Long val2 = 127L;

System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;
```

Note: Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY, Double.NaN are double values. NaN stands for results of operations that cannot be determined, such as dividing 2 infinite values. Furthermore 0d and -0d are different, but == yields true:

```
double d1 = 0d;
double d2 = -0d;
System.out.println(d1 == d2); // true
System.out.println(1d / d1); // Infinity
System.out.println(1d / d2); // -Infinity
System.out.println(Double.POSITIVE_INFINITY / Double.POSITIVE_INFINITY); // NaN
```

Section 10.8: The long primitive

By default, long is a 64-bit signed integer (in Java 8, it can be either signed or unsigned). Signed, it can store a minimum value of -2^{63} , and a maximum value of $2^{63} - 1$, and unsigned it can store a minimum value of 0 and a maximum value of $2^{64} - 1$

```
long example = -42;
long myLong = 284;
long anotherLong = 73;

//an "L" must be appended to the end of the number, because by default,
//numbers are assumed to be the int type. Appending an "L" makes it a long
//as 549755813888 ( $2^{39}$ ) is larger than the maximum value of an int ( $2^{31} - 1$ ),
//"L" must be appended
long bigNumber = 549755813888L;

long addedLongs = myLong + anotherLong; // 284 + 73 = 357
long subtractedLongs = myLong - anotherLong; // 284 - 73 = 211
```

The maximum and minimum values of long can be found at:

```
long high = Long.MAX_VALUE; // high == 9223372036854775807L
long low = Long.MIN_VALUE; // low == -9223372036854775808L
```

The default value of a long is 0L

```
long defaultLong; // defaultLong == 0L
```

Note: letter "L" appended at the end of long literal is case insensitive, however it is good practice to use capital as it is easier to distinct from digit one:

```
2L == 2l; // true
```

Warning: Java caches Integer objects instances from the range -128 to 127. The reasoning is explained here:
https://blogs.oracle.com/darcy/entry/boxing_and_caches_integer_valueof

The following results can be found:

```
Long val1 = 127L;
Long val2 = 127L;

System.out.println(val1 == val2); // true

Long val3 = 128L;
Long val4 = 128L;
```

```
System.out.println(val3 == val4); // false
```

要正确比较两个 Object 类型的 Long 值, 请使用以下代码 (Java 1.7 及以后版本) :

```
Long val3 = 128L;  
Long val4 = 128L;
```

```
System.out.println(Objects.equal(val3, val4)); // true
```

将基本类型 long 与 Object 类型的 long 比较, 不会像用 == 比较两个对象那样出现假阴性。

第 10.9 节 : boolean 基本类型

boolean 可以存储两个值之一, true 或 false

```
boolean foo = true;  
System.out.println("foo = " + foo); // foo = true  
  
boolean bar = false;  
System.out.println("bar = " + bar); // bar = false  
  
boolean notFoo = !foo;  
System.out.println("notFoo = " + notFoo); // notFoo = false  
  
boolean fooAndBar = foo && bar;  
System.out.println("fooAndBar = " + fooAndBar); // fooAndBar = false  
  
boolean fooOrBar = foo || bar;  
System.out.println("fooOrBar = " + fooOrBar); // fooOrBar = true  
  
boolean fooXorBar = foo ^ bar;  
System.out.println("fooXorBar = " + fooXorBar); // fooXorBar = true
```

boolean 的默认值是 false

```
boolean defaultBoolean; // defaultBoolean == false
```

第10.10节 : byte原始类型

一个字节是一个8位有符号整数。它可以存储的最小值是-2的7次方 (-128), 最大值是2的7次方减1 (127)

```
byte example = -36;  
byte myByte = 96;  
byte anotherByte = 7;  
  
byte addedBytes = (byte) (myByte + anotherByte); // 103  
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

byte 的最大值和最小值可以在以下位置找到:

```
byte high = Byte.MAX_VALUE; // high == 127  
byte low = Byte.MIN_VALUE; // low == -128
```

byte 的默认值是 0

```
System.out.println(val3 == val4); // false
```

To properly compare 2 Object Long values, use the following code(From Java 1.7 onward):

```
Long val3 = 128L;  
Long val4 = 128L;  
  
System.out.println(Objects.equal(val3, val4)); // true
```

Comparing a primitive long to an Object long will not result in a false negative like comparing 2 objects with == does.

Section 10.9: The boolean primitive

A boolean can store one of two values, either true or false

```
boolean foo = true;  
System.out.println("foo = " + foo); // foo = true  
  
boolean bar = false;  
System.out.println("bar = " + bar); // bar = false  
  
boolean notFoo = !foo;  
System.out.println("notFoo = " + notFoo); // notFoo = false  
  
boolean fooAndBar = foo && bar;  
System.out.println("fooAndBar = " + fooAndBar); // fooAndBar = false  
  
boolean fooOrBar = foo || bar;  
System.out.println("fooOrBar = " + fooOrBar); // fooOrBar = true  
  
boolean fooXorBar = foo ^ bar;  
System.out.println("fooXorBar = " + fooXorBar); // fooXorBar = true
```

The default value of a boolean is false

```
boolean defaultBoolean; // defaultBoolean == false
```

Section 10.10: The byte primitive

A byte is a 8-bit signed integer. It can store a minimum value of -27 (-128), and a maximum value of 27 - 1 (127)

```
byte example = -36;  
byte myByte = 96;  
byte anotherByte = 7;  
  
byte addedBytes = (byte) (myByte + anotherByte); // 103  
byte subtractedBytes = (byte) (myBytes - anotherByte); // 89
```

The maximum and minimum values of byte can be found at:

```
byte high = Byte.MAX_VALUE; // high == 127  
byte low = Byte.MIN_VALUE; // low == -128
```

The default value of a byte is 0

```
byte defaultByte; // defaultByte == 0
```

第10.11节：负值表示

Java和大多数其他语言使用一种称为2的补码表示法来存储负整数。

对于使用 n 位的唯一二进制数据类型表示，值的编码方式如下：

最低有效的 $n - 1$ 位以整数表示存储一个正整数 x 。最高有效位存储一个值为 s 的位。这些位表示的值为

$$x - s * 2^{n-1}$$

即如果最高有效位为1，则从其他位所能表示的数字中减去一个比它大1的值 ($2^n - 2 + 2^{n-1} - 3 + \dots + 2^1 + 2^0 = 2^n - 1 - 1$)，从而实现每个值从 -2^{n-1} ($s = 1; x = 0$) 到 $2^{n-1} - 1$ ($s = 0; x = 2^{n-1} - 1$) 的唯一二进制表示。

这还有一个好处，就是你可以像对待正二进制数一样对二进制表示进行加法：

		加法结果	
s1	s2	x1 + x2	溢出
0 0 否		$x_1 + x_2 = v_1 + v_2$	
0 0 是		数据类型无法表示，数值过大（溢出）	
0 1 否		$x_1 + x_2 - 2^{n-1} = x_1 + x_2 - s_2 * 2^{n-1}$ $= v_1 + v_2$	
0 1 是		$(x_1 + x_2) \bmod 2^{n-1} = x_1 + x_2 - 2^{n-1}$ $= v_1 + v_2$	
1 0 *		见上文（交换加数）	
1 1 否		数据类型无法表示，数值过小 ($x_1 + x_2 - 2^n < -2^{n-1}$ ；下溢）	
1 1 是		$(x_1 + x_2) \bmod 2^{n-1} - 2^{n-1} = (x_1 + x_2 - 2^{n-1}) - 2^{n-1}$ $= (x_1 - s_1 * 2^{n-1}) + (x_2 - s_2 * 2^{n-1})$ $= v_1 + v_2$	

注意，这一事实使得找到加法逆元（二进制负值）变得简单：

观察将按位取反加到数字上会使所有位变为1。现在加1使值溢出，得到中性元素0（所有位为0）。

因此，一个数字 i 的负值可以通过以下方式计算（此处忽略可能的提升为 int）：

$$(\sim i) + 1$$

示例：取0的负值（byte）：

对 0 取负的结果是 11111111。加1后得到 00000000 (9位)。由于 byte 只能存储8位，最左边的位被截断，结果是 00000000

原始值	过程	结果
0 (00000000)	取负	-0 (11111111)
11111111	二进制00000000加1	
100000000	截断为8位 00000000 (-0等于0)	

```
byte defaultByte; // defaultByte == 0
```

Section 10.11: Negative value representation

Java and most other languages store negative integral numbers in a representation called 2's complement notation.

For a unique binary representation of a data type using n bits, values are encoded like this:

The least significant $n - 1$ bits store a positive integral number x in integral representation. Most significant value stores a bit with value s . The value represented by those bits is

$$x - s * 2^{n-1}$$

i.e. if the most significant bit is 1, then a value that is just by 1 larger than the number you could represent with the other bits ($2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0 = 2^n - 1 - 1$) is subtracted allowing a unique binary representation for each value from -2^{n-1} ($s = 1; x = 0$) to $2^{n-1} - 1$ ($s = 0; x = 2^{n-1} - 1$)

This also has the nice side effect, that you can add the binary representations as if they were positive binary numbers:

		v1 = x1 - s1 * 2 ⁿ⁻¹	v2 = x2 - s2 * 2 ⁿ⁻¹	addition result
s1	s2	x1 + x2 overflow		
0 0 No		$x_1 + x_2 = v_1 + v_2$		
0 0 Yes		too large to be represented with data type (overflow)		
0 1 No		$x_1 + x_2 - 2^{n-1} = x_1 + x_2 - s_2 * 2^{n-1}$ $= v_1 + v_2$		
0 1 Yes		$(x_1 + x_2) \bmod 2^{n-1} = x_1 + x_2 - 2^{n-1}$ $= v_1 + v_2$		
1 0 *		see above (swap summands)		
1 1 No		too small to be represented with data type ($x_1 + x_2 - 2^n < -2^{n-1}$; underflow)		
1 1 Yes		$(x_1 + x_2) \bmod 2^{n-1} - 2^{n-1} = (x_1 + x_2 - 2^{n-1}) - 2^{n-1}$ $= (x_1 - s_1 * 2^{n-1}) + (x_2 - s_2 * 2^{n-1})$ $= v_1 + v_2$		

Note that this fact makes finding binary representation of the additive inverse (i.e. the negative value) easy:

Observe that adding the bitwise complement to the number results in all bits being 1. Now add 1 to make value overflow and you get the neutral element 0 (all bits 0).

So the negative value of a number i can be calculated using (ignoring possible promotion to int here)

$$(\sim i) + 1$$

Example: taking the negative value of 0 (byte):

The result of negating 0, is 11111111. Adding 1 gives a value of 100000000 (9 bits). Because a byte can only store 8 bits, the leftmost value is truncated, and the result is 00000000

Original	Process	Result
0 (00000000)	Negate	-0 (11111111)
11111111	Add 1 to binary	100000000
100000000	Truncate to 8 bits	00000000 (-0 equals 0)

第10.12节：短整型（short）原始类型

short是一个16位有符号整数。它的最小值是-2的15次方 (-32,768) , 最大值是2的15次方-1 (32,767)

```
short示例= -48;  
short myShort = 987;  
short anotherShort = 17;  
  
short addedShorts = (short) (myShort + anotherShort); // 1,004  
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

可以找到short的最大值和最小值：

```
short high = Short.MAX_VALUE; // high == 32767  
short low = Short.MIN_VALUE; // low == -32768
```

short的默认值是0

```
short defaultShort; // defaultShort == 0
```

Section 10.12: The short primitive

A **short** is a 16-bit signed integer. It has a minimum value of -2¹⁵ (-32,768), and a maximum value of 2¹⁵-1 (32,767)

```
short example = -48;  
short myShort = 987;  
short anotherShort = 17;  
  
short addedShorts = (short) (myShort + anotherShort); // 1,004  
short subtractedShorts = (short) (myShort - anotherShort); // 970
```

The maximum and minimum values of **short** can be found at:

```
short high = Short.MAX_VALUE; // high == 32767  
short low = Short.MIN_VALUE; // low == -32768
```

The default value of a **short** is 0

```
short defaultShort; // defaultShort == 0
```

第11章：字符串

字符串（`java.lang.String`）是存储在程序中的文本片段。字符串在Java中不是原始数据类型，但它们在Java程序中非常常见。

在Java中，字符串是不可变的，意味着它们不能被更改。（[点击这里获取关于不可变性的更详细解释。](#)）

第11.1节：比较字符串

为了比较字符串是否相等，应使用String对象的`equals`或`equalsIgnoreCase`方法。

例如，以下代码片段将判断两个String实例的所有字符是否相等：

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
    // 两个字符串内容相同。
}
```

[实时演示](#)

此示例将比较它们，忽略大小写差异：

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // 两个字符串相等，忽略各字符的大小写。
}
```

[实时演示](#)

注意 `equalsIgnoreCase` 不允许你指定 `Locale`。例如，如果你比较两个单词 "Taki" 和 "TAKI"，在英语中它们是相等的；然而，在土耳其语中它们是不同的（在土耳其语中，小写的 I 是 İ）。对于这种情况，将两个字符串都转换为带有 `Locale` 的小写（或大写），然后用 `equals` 比较是解决方案。

```
String firstString = "Taki";
String secondString = "TAKI";

System.out.println(firstString.equalsIgnoreCase(secondString)); // 打印 true

Locale locale = Locale.forLanguageTag("tr-TR");

System.out.println(firstString.toLowerCase(locale).equals(
    secondString.toLowerCase(locale))); // 打印 false
```

[实时演示](#)

不要使用 `==` 运算符比较字符串

除非你能保证所有字符串都已被常量池化（见下文），否则你不应该使用`==`或`!=`

Chapter 11: Strings

Strings (`java.lang.String`) are pieces of text stored in your program. Strings are **not** a primitive data type in Java, however, they are very common in Java programs.

In Java, Strings are immutable, meaning that they cannot be changed. (Click [here](#) for a more thorough explanation of immutability.)

Section 11.1: Comparing Strings

In order to compare Strings for equality, you should use the String object's `equals` or `equalsIgnoreCase` methods.

For example, the following snippet will determine if the two instances of `String` are equal on all characters:

```
String firstString = "Test123";
String secondString = "Test" + 123;

if (firstString.equals(secondString)) {
    // Both Strings have the same content.
}
```

[Live demo](#)

This example will compare them, independent of their case:

```
String firstString = "Test123";
String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
}
```

[Live demo](#)

Note that `equalsIgnoreCase` does not let you specify a `Locale`. For instance, if you compare the two words "Taki" and "TAKI" in English they are equal; however, in Turkish they are different (in Turkish, the lowercase I is İ). For cases like this, converting both strings to lowercase (or uppercase) with `Locale` and then comparing with `equals` is the solution.

```
String firstString = "Taki";
String secondString = "TAKI";

System.out.println(firstString.equalsIgnoreCase(secondString)); // prints true

Locale locale = Locale.forLanguageTag("tr-TR");

System.out.println(firstString.toLowerCase(locale).equals(
    secondString.toLowerCase(locale))); // prints false
```

[Live demo](#)

Do not use the `==` operator to compare Strings

Unless you can guarantee that all strings have been interned (see below), you **should not** use the `==` or `!=`

运算符来比较字符串。这些运算符实际上比较的是引用，而由于多个String对象可能表示相同的字符串，这很可能导致错误的结果。

相反，应使用String.equals(Object)方法，该方法会基于字符串的值进行比较。详细说明请参见陷阱：使用 == 比较字符串。

在 switch 语句中比较字符串

版本 ≥ Java SE 7

从 Java 1.7 开始，可以在switch语句中将字符串变量与字面量进行比较。确保字符串不为 null，否则总会抛出NullPointerException。值的比较使用String.equals，即区分大小写。

```
String stringToSwitch = "A";

switch (stringToSwitch) {
    case "a":
        System.out.println("a");
        break;
    case "A":
        System.out.println("A"); //代码写在这里
        break;
    case "B":
        System.out.println("B");
        break;
    default:
        break;
}
```

[实时演示](#)

与常量值比较字符串

当比较一个String与常量值时，可以将常量值放在equals的左侧，以确保如果另一个字符串为null时，不会抛出NullPointerException

```
"baz".equals(foo)
```

虽然foo.equals("baz")在foo为null时会抛出NullPointerException "baz".equals(foo)则会返回false。

版本 ≥ Java SE 7

一个更易读的替代方法是使用Objects.equals()，它会对两个参数进行null检查：
Objects.equals(foo, "baz")。

(注意：是否应该普遍避免NullPointerExceptions，还是让它们发生后再修复根本原因，这一点存在争议；参见[这里](#)和[这里](#)。当然，将避免策略称为“最佳实践”是不合理的。) 字符串排序

String类实现了Comparable<String>接口，使用String.compareTo方法（如本示例开头所述）。这使得String对象的自然排序为区分大小写的顺序。String类提供了一个名为CASE_INSENSITIVE_ORDER的Comparator<String>常量，适用于不区分大小写的排序。

operators to compare Strings. These operators actually test references, and since multiple String objects can represent the same String, this is liable to give the wrong answer.

Instead, use the [String.equals\(Object\)](#) method, which will compare the String objects based on their values. For a detailed explanation, please refer to Pitfall: using == to compare strings.

Comparing Strings in a switch statement

Version ≥ Java SE 7

As of Java 1.7, it is possible to compare a String variable to literals in a switch statement. Make sure that the String is not null, otherwise it will always throw a NullPointerException. Values are compared using [String.equals](#), i.e. case sensitive.

```
String stringToSwitch = "A";

switch (stringToSwitch) {
    case "a":
        System.out.println("a");
        break;
    case "A":
        System.out.println("A"); //the code goes here
        break;
    case "B":
        System.out.println("B");
        break;
    default:
        break;
}
```

[Live demo](#)

Comparing Strings with constant values

When comparing a String to a constant value, you can put the constant value on the left side of equals to ensure that you won't get a NullPointerException if the other String is null.

```
"baz".equals(foo)
```

While foo.equals("baz") will throw a NullPointerException if foo is null, "baz".equals(foo) will evaluate to false.

Version ≥ Java SE 7

A more readable alternative is to use [Objects.equals\(\)](#), which does a null check on both parameters:
Objects.equals(foo, "baz").

(Note: It is debatable as to whether it is better to avoid NullPointerExceptions in general, or let them happen and then fix the root cause; see [here](#) and [here](#). Certainly, calling the avoidance strategy "best practice" is not justifiable.)

String orderings

The [String](#) class implements Comparable<String> with the [String.compareTo](#) method (as described at the start of this example). This makes the natural ordering of String objects case-sensitive order. The [String](#) class provides a Comparator<String> constant called CASE_INSENSITIVE_ORDER suitable for case-insensitive sorting.

与驻留字符串的比较

Java语言规范 (JLS 3.10.6) 指出：

“此外，字符串字面量总是引用同一个String类的实例。这是因为字符串字面量——或者更广义地说，作为常量表达式值的字符串——是通过interned来共享唯一实例的，使用的方法是String.intern。”

这意味着使用==比较两个字符串字面量的引用是安全的。此外，使用String.intern()方法生成的String对象的引用也同样适用。

例如：

```
String strObj = new String("Hello!");
String str = "Hello!";

// 这两个字符串引用指向的字符串是相等的
if (strObj.equals(str)) {
    System.out.println("字符串相等");
}

// 这两个字符串引用并不指向同一个对象
if (strObj != str) {
    System.out.println("字符串不是同一个对象");
}

// 如果我们对一个等于给定字面量的字符串进行驻留，结果是
// 一个与该字面量具有相同引用的字符串。
String internedStr = strObj.intern();

if (internedStr == str) {
    System.out.println("驻留字符串和字面量是同一个对象");
}
```

在幕后，驻留机制维护了一个包含所有仍然可达的驻留字符串的哈希表。当你在一个String上调用intern()方法时，该方法会在哈希表中查找该对象：

- 如果找到该字符串，则返回该值作为驻留字符串。
- 否则，将该字符串的副本添加到哈希表中，并返回该字符串作为驻留字符串。

可以使用驻留机制使字符串能够通过==进行比较。然而，这样做存在重大问题；详情请参见“陷阱 - 为了使用 == 而驻留字符串是个坏主意”。大多数情况下不推荐这样做。

第11.2节：改变字符串中字符的大小写

String类型提供了两种方法用于在大写和小写之间转换字符串：

- [toUpperCase](#) 将所有字符转换为大写
- [toLowerCase](#) 将所有字符转换为小写

这两种方法都返回转换后的字符串作为新的String实例：原始的String对象不会被修改，因为String在Java中是不可变的。有关不可变性的更多信息，请参见：Java中字符串的不可变性

```
String string = "这是一个随机字符串";
```

Comparing with interned Strings

The Java Language Specification ([JLS 3.10.6](#)) states the following:

"Moreover, a string literal always refers to the same instance of class `String`. This is because string literals - or, more generally, strings that are the values of constant expressions - are *interned* so as to share unique instances, using the method `String.intern()`."

This means it is safe to compare references to two string *literals* using ==. Moreover, the same is true for references to `String` objects that have been produced using the `String.intern()` method.

For example:

```
String strObj = new String("Hello!");
String str = "Hello!";

// The two string references point two strings that are equal
if (strObj.equals(str)) {
    System.out.println("The strings are equal");
}

// The two string references do not point to the same object
if (strObj != str) {
    System.out.println("The strings are not the same object");
}

// If we intern a string that is equal to a given literal, the result is
// a string that has the same reference as the literal.
String internedStr = strObj.intern();

if (internedStr == str) {
    System.out.println("The interned string and the literal are the same object");
}
```

Behind the scenes, the interning mechanism maintains a hash table that contains all interned strings that are still *reachable*. When you call `intern()` on a `String`, the method looks up the object in the hash table:

- If the string is found, then that value is returned as the interned string.
- Otherwise, a copy of the string is added to the hash table and that string is returned as the interned string.

It is possible to use interning to allow strings to be compared using ==. However, there are significant problems with doing this; see Pitfall - Interning strings so that you can use == is a bad idea for details. It is not recommended in most cases.

Section 11.2: Changing the case of characters within a String

The `String` type provides two methods for converting strings between upper case and lower case:

- [toUpperCase](#) to convert all characters to upper case
- [toLowerCase](#) to convert all characters to lower case

These methods both return the converted strings as new `String` instances: the original `String` objects are not modified because `String` is immutable in Java. See this for more on immutability : [Immutability of Strings in Java](#)

```
String string = "This is a Random String";
```

```

String upper = string.toUpperCase();
String lower = string.toLowerCase();

System.out.println(string); // 输出 "这是一个随机字符串"
System.out.println(lower); // 输出 "这是一个随机字符串"
System.out.println(upper); // 输出 "这是一个随机字符串"

```

非字母字符，如数字和标点符号，不受这些方法的影响。注意，这些方法在某些条件下可能会错误处理某些Unicode字符。

注意：这些方法是locale-sensitive的，如果用于需要独立于区域设置解释的字符串，可能会产生意外结果。例如编程语言标识符、协议键和HTML标签。

例如，在土耳其语区域设置中，“TITLE”.toLowerCase()返回“title”，其中ı(\u0131)是无点小写字母i字符。要获得区域设置无关字符串的正确结果，请将Locale.ROOT作为参数传递给相应的大小写转换方法（例如toLowerCase(Locale.ROOT)或toUpperCase(Locale.ROOT)）。

虽然在大多数情况下使用Locale.ENGLISH也是正确的，但语言不变的方式是Locale.ROOT。

可以在Unicode联盟网站上找到需要特殊大小写处理的Unicode字符的详细列表。

更改ASCII字符串中特定字符的大小写：

要更改ASCII字符串中特定字符的大小写，可以使用以下算法：

步骤：

1. 声明一个字符串。
2. 输入字符串。
3. 将字符串转换为字符数组。
4. 输入要搜索的字符。
5. 在字符数组中搜索该字符。
6. 如果找到，检查该字符是小写还是大写。
 - 如果是大写字母，则将字符的ASCII码加32。
 - 如果是小写字母，则将字符的ASCII码减32。
7. 从字符数组中更改原始字符。
8. 将字符数组转换回字符串。

完成，字符的大小写已更改。

该算法的代码示例如下：

```

Scanner scanner = new Scanner(System.in);
System.out.println("Enter the String");
String s = scanner.next();
char[] a = s.toCharArray();
System.out.println("Enter the character you are looking for");
System.out.println(s);
String c = scanner.next();
char d = c.charAt(0);

for (int i = 0; i <= s.length(); i++) {
    if (a[i] == d) {
        if (d >= 'a' && d <= 'z') {
            d -= 32;
        }
    }
}

```

```

String upper = string.toUpperCase();
String lower = string.toLowerCase();

```

```

System.out.println(string); // prints "This is a Random String"
System.out.println(lower); // prints "this is a random string"
System.out.println(upper); // prints "THIS IS A RANDOM STRING"

```

Non-alphabetic characters, such as digits and punctuation marks, are unaffected by these methods. Note that these methods may also incorrectly deal with certain Unicode characters under certain conditions.

Note: These methods are *locale-sensitive*, and may produce unexpected results if used on strings that are intended to be interpreted independent of the locale. Examples are programming language identifiers, protocol keys, and [HTML tags](#).

For instance, “TITLE”.toLowerCase() in a Turkish locale returns “title”, where ı (\u0131) is the [LATIN SMALL LETTER DOTLESS I](#) character. To obtain correct results for locale insensitive strings, pass [Locale.ROOT](#) as a parameter to the corresponding case converting method (e.g. [toLowerCase\(Locale.ROOT\)](#) or [toUpperCase\(Locale.ROOT\)](#)).

Although using [Locale.ENGLISH](#) is also correct for most cases, the **language invariant** way is [Locale.ROOT](#).

A detailed list of Unicode characters that require special casing can be found [on the Unicode Consortium website](#).

Changing case of a specific character within an ASCII string:

To change the case of a specific character of an ASCII string following algorithm can be used:

Steps:

1. Declare a string.
2. Input the string.
3. Convert the string into a character array.
4. Input the character that is to be searched.
5. Search for the character into the character array.
6. If found, check if the character is lowercase or uppercase.
 - If Uppercase, add 32 to the ASCII code of the character.
 - If Lowercase, subtract 32 from the ASCII code of the character.
7. Change the original character from the Character array.
8. Convert the character array back into the string.

Voila, the Case of the character is changed.

An example of the code for the algorithm is:

```

Scanner scanner = new Scanner(System.in);
System.out.println("Enter the String");
String s = scanner.next();
char[] a = s.toCharArray();
System.out.println("Enter the character you are looking for");
System.out.println(s);
String c = scanner.next();
char d = c.charAt(0);

for (int i = 0; i <= s.length(); i++) {
    if (a[i] == d) {
        if (d >= 'a' && d <= 'z') {
            d -= 32;
        }
    }
}

```

```

} 否则如果 (d >= 'A' && d <= 'Z') {
    d += 32;
}
a[i] = d;
break;
}
s = String.valueOf(a);
System.out.println(s);

```

第11.3节：在字符串中查找另一个字符串

要检查特定字符串 `a` 是否包含在字符串 `b` 中，我们可以使用方法 `String.contains()`，语法如下：

```
b.contains(a); // 如果 b 中包含 a 则返回 true, 否则返回 false
```

`String.contains()` 方法可用于验证是否能在字符串中找到一个 `CharSequence`。该方法以区分大小写的方式查找字符串 `a` 是否存在于字符串 `b` 中。

```

String str1 = "Hello World";
String str2 = "Hello";
String str3 = "helLO";

System.out.println(str1.contains(str2)); // 打印 true
System.out.println(str1.contains(str3)); // 打印 false

```

[Ideone 在线演示](#)

要查找一个字符串在另一个字符串中开始的确切位置，使用 `String.indexOf()` 方法：

```

String s = "this is a long sentence";
int i = s.indexOf('i');      // 字符串中第一个 'i' 的索引是 2
int j = s.indexOf("long"); // "long" 在 s 中首次出现的索引是 10
int k = s.indexOf('z');      // k 是 -1, 因为在字符串 s 中未找到 'z'
int h = s.indexOf("LoNg"); // h 是 -1, 因为在字符串 s 中未找到 "LoNg"

```

[Ideone 在线演示](#)

`String.indexOf()` 方法返回一个字符 (char) 或字符串 (String) 在另一个字符串中的第一个索引位置。如果未找到，则返回 -1。

注意：`String.indexOf()` 方法区分大小写。

忽略大小写搜索的示例：

```

String str1 = "Hello World";
String str2 = "wOr";
str1.indexOf(str2);           // -1
str1.toLowerCase().contains(str2.toLowerCase()); // true
str1.toLowerCase().indexOf(str2.toLowerCase()); // 6

```

[Ideone 在线演示](#)

```

} else if (d >= 'A' && d <= 'Z') {
    d += 32;
}
a[i] = d;
break;
}
s = String.valueOf(a);
System.out.println(s);

```

Section 11.3: Finding a String Within Another String

To check whether a particular String `a` is being contained in a String `b` or not, we can use the method `String.contains()` with the following syntax:

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

The `String.contains()` method can be used to verify if a `CharSequence` can be found in the String. The method looks for the String `a` in the String `b` in a case-sensitive way.

```

String str1 = "Hello World";
String str2 = "Hello";
String str3 = "helLO";

System.out.println(str1.contains(str2)); // prints true
System.out.println(str1.contains(str3)); // prints false

```

[Live Demo on Ideone](#)

To find the exact position where a String starts within another String, use `String.indexOf()`:

```

String s = "this is a long sentence";
int i = s.indexOf('i');      // the first 'i' in String is at index 2
int j = s.indexOf("long"); // the index of the first occurrence of "long" in s is 10
int k = s.indexOf('z');      // k is -1 because 'z' was not found in String s
int h = s.indexOf("LoNg"); // h is -1 because "LoNg" was not found in String s

```

[Live Demo on Ideone](#)

The `String.indexOf()` method returns the first index of a `char` or `String` in another `String`. The method returns -1 if it is not found.

Note: The `String.indexOf()` method is case sensitive.

Example of search ignoring the case:

```

String str1 = "Hello World";
String str2 = "wOr";
str1.indexOf(str2);           // -1
str1.toLowerCase().contains(str2.toLowerCase()); // true
str1.toLowerCase().indexOf(str2.toLowerCase()); // 6

```

[Live Demo on Ideone](#)

第11.4节：字符串池和堆存储

像许多Java对象一样，所有String实例都在堆上创建，即使是字面量。当JVM发现一个String字面量在堆中没有对应的引用时，JVM会在堆上创建一个对应的String实例，并且它还会在字符串池中存储对新创建的String实例的引用。对同一个String字面量的任何其他引用都会被替换为堆中先前创建的String实例。

让我们看下面的例子：

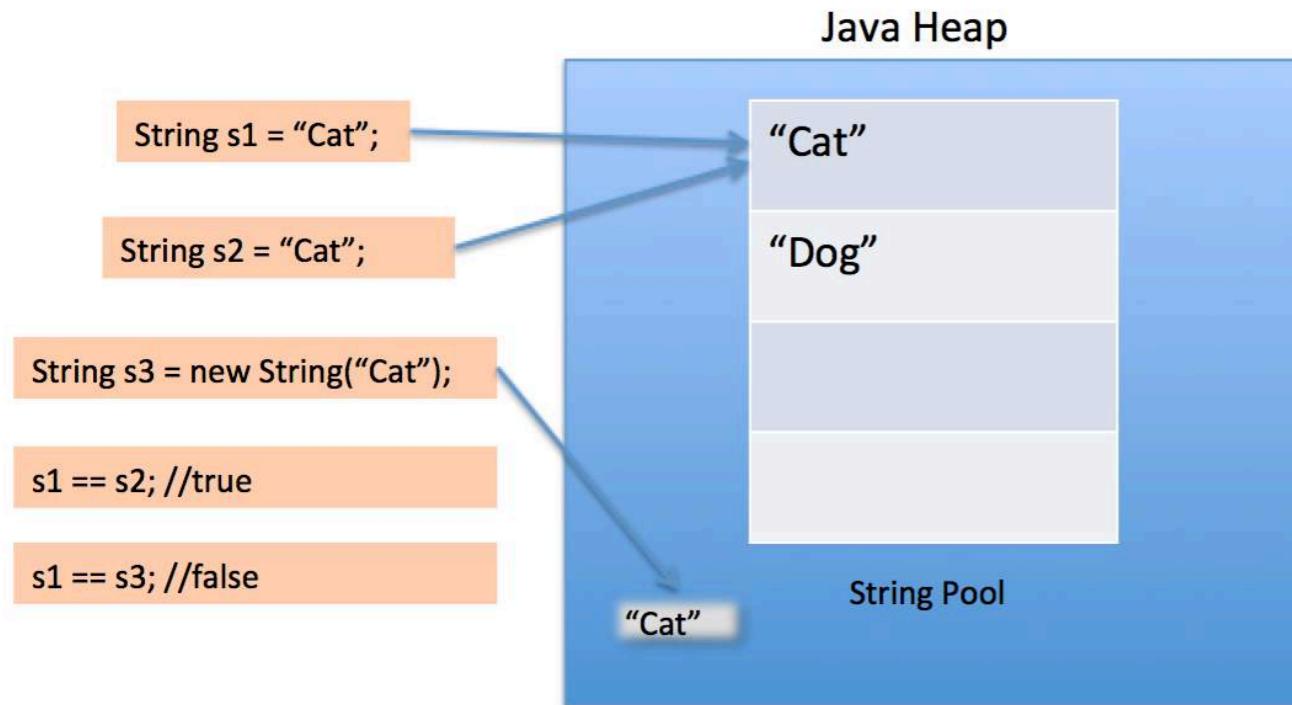
```
class Strings
{
    public static void main (String[] args)
    {
        String a = "alpha";
        String b = "alpha";
        String c = new String("alpha");

        //所有三个字符串都是等价的
        System.out.println(a.equals(b) && b.equals(c));

        //尽管只有a和b引用了同一个堆对象
        System.out.println(a == b);
        System.out.println(a != c);
        System.out.println(b != c);
    }
}
```

上述代码的输出是：

```
true
true
true
true
```



Section 11.4: String pool and heap storage

Like many Java objects, **all** `String` instances are created on the heap, even literals. When the JVM finds a `String` literal that has no equivalent reference in the heap, the JVM creates a corresponding `String` instance on the heap **and** it also stores a reference to the newly created `String` instance in the String pool. Any other references to the same `String` literal are replaced with the previously created `String` instance in the heap.

Let's look at the following example:

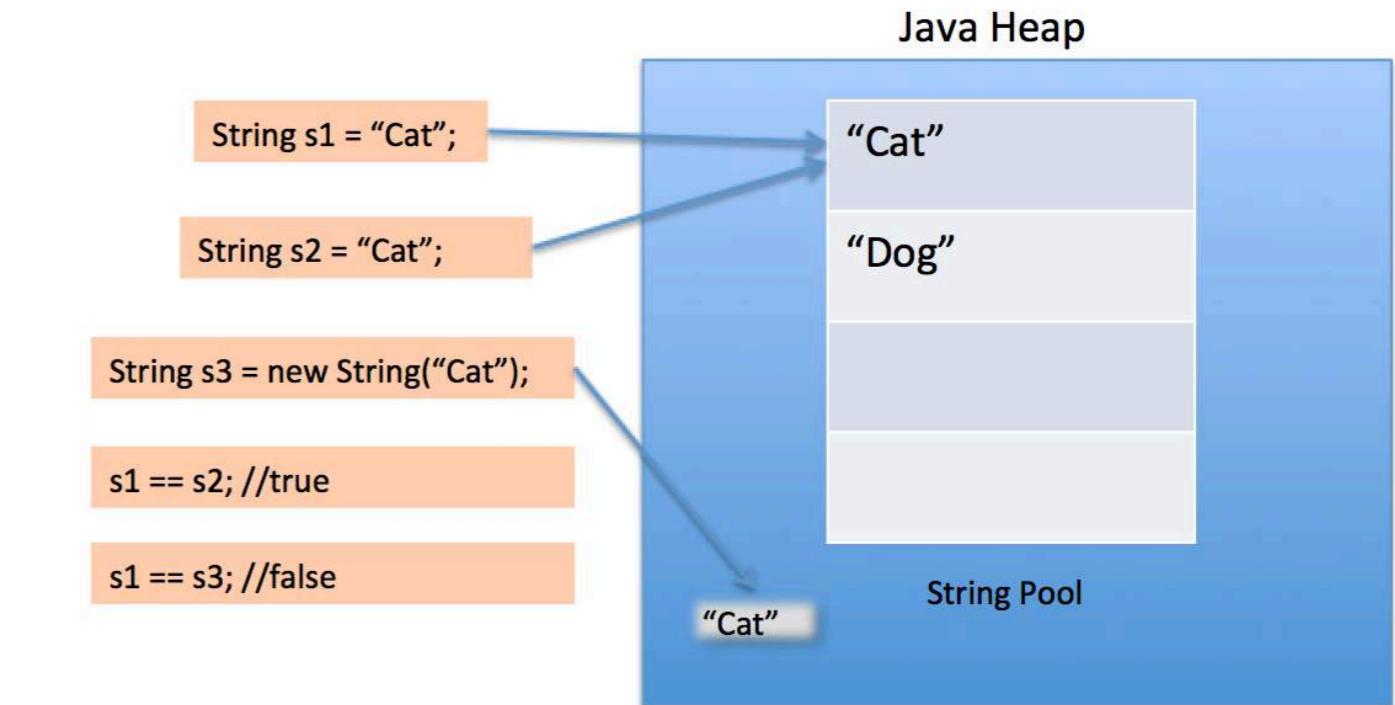
```
class Strings
{
    public static void main (String[] args)
    {
        String a = "alpha";
        String b = "alpha";
        String c = new String("alpha");

        //All three strings are equivalent
        System.out.println(a.equals(b) && b.equals(c));

        //Although only a and b reference the same heap object
        System.out.println(a == b);
        System.out.println(a != c);
        System.out.println(b != c);
    }
}
```

The output of the above is:

```
true
true
true
true
```



当我们使用双引号创建字符串时，它首先会在字符串池中查找具有相同值的字符串，如果找到则直接返回该引用，否则会在池中创建一个新的字符串，然后返回该引用。

然而，使用new操作符时，我们强制字符串类在堆空间中创建一个新的字符串对象。我们可以使用intern()方法将其放入池中，或者引用字符串池中具有相同值的其他字符串对象。

字符串池本身也创建在堆上。

版本 < Java SE 7

在Java 7之前，字符串字面量存储在方法区的运行时常量池中，该区域属于PermGen，且大小固定。

字符串池也位于PermGen中。

版本 ≥ Java SE 7

[RFC: 6962931](#)

在 JDK 7 中，驻留字符串不再分配在 Java 堆的永久代中，而是分配在 Java 堆的主要部分（称为年轻代和老年代），与应用程序创建的其他对象一起。此更改将导致更多数据驻留在主 Java 堆中，永久代中的数据减少，因此可能需要调整堆大小。大多数应用程序由于此更改在堆使用上只会看到相对较小的差异，但加载许多类或大量使用String.intern()方法的较大应用程序将看到更显著的差异。

When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.

However using new operator, we force String class to create a new String object in heap space. We can use intern() method to put it into the pool or refer to other String object from string pool having same value.

The String pool itself is also created on the heap.

Version < Java SE 7

Before Java 7, **String literals** were stored in the runtime constant pool in the method area of PermGen, that had a fixed size.

The String pool also resided in PermGen.

Version ≥ Java SE 7

[RFC: 6962931](#)

In JDK 7, interned strings are no longer allocated in the permanent generation of the Java heap, but are instead allocated in the main part of the Java heap (known as the young and old generations), along with the other objects created by the application. This change will result in more data residing in the main Java heap, and less data in the permanent generation, and thus may require heap sizes to be adjusted. Most applications will see only relatively small differences in heap usage due to this change, but larger applications that load many classes or make heavy use of the **String.intern()** method will see more significant differences.

第11.5节：拆分字符串

你可以根据特定的分隔字符或正则表达式拆分String，使用String的split()方法，其签名如下：

```
public String[] split(String regex)
```

注意，分隔字符或正则表达式会从结果的字符串数组中移除。

使用分隔字符的示例：

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// 结果是 dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

使用正则表达式的示例：

```
String lineFromInput = "What    do you need    from me?";
String[] words = lineFromInput.split("\\s+"); // 一个或多个空格字符
// 结果是 words = {"What", "do", "you", "need", "from", "me?"};
```

你甚至可以直接拆分一个String字面量：

```
String[] firstNames = "米奇, 弗兰克, 艾丽西亚, 汤姆".split(", ");
// 结果是 firstNames = {"米奇", "弗兰克", "艾丽西亚", "汤姆"};
```

Section 11.5: Splitting Strings

You can split a **String** on a particular delimiting character or a Regular Expression, you can use the **String.split()** method that has the following signature:

```
public String[] split(String regex)
```

Note that delimiting character or regular expression gets removed from the resulting String Array.

Example using delimiting character:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Example using regular expression:

```
String lineFromInput = "What    do you need    from me?";
String[] words = lineFromInput.split("\\s+"); // one or more space chars
// Result is words = {"What", "do", "you", "need", "from", "me?"};
```

You can even directly split a **String** literal:

```
String[] firstNames = "Mickey, Frank, Alicia, Tom".split(", ");
// Result is firstNames = {"Mickey", "Frank", "Alicia", "Tom"};
```

警告：别忘了参数总是被当作正则表达式处理。

```
"aaa.bbb".split("."); // 这会返回一个空数组
```

在前面的例子中，`.` 被当作匹配任意字符的正则表达式通配符，因为每个字符都是分隔符，结果是一个空数组。

基于作为正则表达式元字符的分隔符进行拆分

以下字符在正则表达式中被视为特殊字符（也称元字符）

```
< > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

要基于上述分隔符之一拆分字符串，你需要对它们进行转义，使用`\`，或者使用模式。引用()：

- 使用Pattern.quote()：

```
String s = "a|b|c";
String regex = Pattern.quote("|");
String[] arr = s.split(regex);
```

- 转义特殊字符：

```
String s = "a|b|c";
String[] arr = s.split("\\|");
```

split 会移除空值

split(delimiter) 默认会从结果数组中移除尾部的空字符串。要关闭此机制，我们需要使用重载版本的 split(delimiter, limit)，并将 limit 设置为负值，例如

```
String[] split = data.split("\\|", -1);
```

split(regex) 内部返回 split(regex, 0) 的结果。

limit 参数控制模式应用的次数，因此影响结果数组的长度。

如果 limit n 大于零，则模式最多应用 $n - 1$ 次，数组长度不会超过 n ，且数组的最后一项将包含最后一个匹配分隔符之后的所有输入内容。

如果 n 是负数，则模式将尽可能多次应用，数组可以具有任意长度。

如果 n 是零，则模式将尽可能多次应用，数组可以具有任意长度，且尾部的空字符串将被丢弃。

使用 StringTokenizer 进行拆分

除了 split() 方法，字符串也可以使用 StringTokenizer 进行拆分。

StringTokenizer 比 String.split() 更加严格，也稍微难用一些。它本质上是为提取由固定字符集（以 String 形式给出）分隔的标记而设计的。每个字符都将作为分隔符。由于这一限制，它的速度大约是 String.split() 的两倍。

默认的字符集为空格 (`\t\n\f`)。以下示例将分别打印出每个单词。

Warning: Do not forget that the parameter is always treated as a regular expression.

```
"aaa.bbb".split("."); // This returns an empty array
```

In the previous example `.` is treated as the regular expression wildcard that matches any character, and since every character is a delimiter, the result is an empty array.

Splitting based on a delimiter which is a regex meta-character

The following characters are considered special (aka meta-characters) in regex

```
< > - = ! ( ) [ ] { } \ ^ $ | ? * + .
```

To split a string based on one of the above delimiters, you need to either escape them using `\` or use Pattern.quote()：

- Using Pattern.quote()：

```
String s = "a|b|c";
String regex = Pattern.quote("|");
String[] arr = s.split(regex);
```

- Escaping the special characters：

```
String s = "a|b|c";
String[] arr = s.split("\\|");
```

Split removes empty values

split(delimiter) by default removes trailing empty strings from result array. To turn this mechanism off we need to use overloaded version of split(delimiter, limit) with limit set to negative value like

```
String[] split = data.split("\\|", -1);
```

split(regex) internally returns result of split(regex, 0).

The limit parameter controls the number of times the pattern is applied and therefore affects the length of the resulting array.

If the limit n is greater than zero then the pattern will be applied at most $n - 1$ times, the array's length will be no greater than n , and the array's last entry will contain all input beyond the last matched delimiter.

If n is negative, then the pattern will be applied as many times as possible and the array can have any length.

If n is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded.

Splitting with a StringTokenizer

Besides the split() method Strings can also be split using a StringTokenizer.

StringTokenizer is even more restrictive than String.split(), and also a bit harder to use. It is essentially designed for pulling out tokens delimited by a fixed set of characters (given as a String). Each character will act as a separator. Because of this restriction, it's about twice as fast as String.split().

Default set of characters are empty spaces (`\t\n\f`). The following example will print out each word separately.

```
String str = "the lazy fox jumped over the brown fence";
 StringTokenizer tokenizer = new StringTokenizer(str);
 while (tokenizer.hasMoreTokens()) {
     System.out.println(tokenizer.nextToken());
 }
```

这将打印出：

```
the
lazy
fox
jumped
over
the
brown
fence
```

您可以使用不同的字符集进行分隔。

```
String str = "jumped over";
// 在这种情况下，字符 `u` 和 `e` 将被用作分隔符
StringTokenizer tokenizer = new StringTokenizer(str, "ue");
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

这将打印出：

```
j
mp
d ov
r
```

第11.6节：使用分隔符连接字符串

版本 ≥ Java SE 8

可以使用静态方法 `String.join()` 来连接字符串数组：

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // 输出 "foo + bar + foobar"
```

同样，`String.join()` 方法对 `Iterable` 也有重载版本。

要对连接操作进行细粒度控制，可以使用`StringJoiner`类：

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
// 最后两个参数是可选的,
// 它们定义了结果字符串的前缀和后缀

sj.add("foo");
sj.add("bar");
sj.add("foobar");

System.out.println(sj); // 输出 "[foo, bar, foobar]"
```

```
String str = "the lazy fox jumped over the brown fence";
StringTokenizer tokenizer = new StringTokenizer(str);
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

This will print out:

```
the
lazy
fox
jumped
over
the
brown
fence
```

You can use different character sets for separation.

```
String str = "jumped over";
// In this case character `u` and `e` will be used as delimiters
StringTokenizer tokenizer = new StringTokenizer(str, "ue");
while (tokenizer.hasMoreTokens()) {
    System.out.println(tokenizer.nextToken());
}
```

This will print out:

```
j
mp
d ov
r
```

Section 11.6: Joining Strings with a delimiter

Version ≥ Java SE 8

An array of strings can be joined using the static method [String.join\(\)](#):

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // Prints "foo + bar + foobar"
```

Similarly, there's an overloaded [String.join\(\)](#) method for `Iterables`.

To have a fine-grained control over joining, you may use [StringJoiner](#) class:

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
// The last two arguments are optional,
// they define prefix and suffix for the result string

sj.add("foo");
sj.add("bar");
sj.add("foobar");

System.out.println(sj); // Prints "[foo, bar, foobar]"
```

要连接字符串流，可以使用joining collector : _____

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // 输出 "foo, bar, foobar"
```

这里也可以定义前缀和后缀选项 :

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining("", "", "{}"));
System.out.println(joined); // 输出 "{foo, bar, foobar}"
```

第11.7节：字符串连接和StringBuilder

字符串连接可以使用+运算符进行。例如：

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

通常编译器实现会在底层使用涉及

StringBuilder的方法来执行上述连接。编译后，代码看起来类似于以下内容：

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

StringBuilder有多个重载方法用于追加不同类型，例如追加一个int

而不是一个String。例如，可以将以下实现转换为：

```
String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2
```

转换为以下内容：

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

上述示例说明了一个简单的连接操作，该操作实际上在代码中的单一位置完成。

连接操作涉及单个StringBuilder实例。在某些情况下，连接是以累积的方式进行的，例如在循环中：

```
String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;
```

在这种情况下，通常不会应用编译器优化，每次迭代都会创建一个新的StringBuilder对象。可以通过显式地将代码转换为使用单个StringBuilder来优化：

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
}
```

To join a stream of strings, you may use the [joining collector](#):

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // Prints "foo, bar, foobar"
```

There's an option to define [prefix and suffix](#) here as well:

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining("", "", "{}"));
System.out.println(joined); // Prints "{foo, bar, foobar}"
```

Section 11.7: String concatenation and StringBuilder

String concatenation can be performed using the + operator. For example:

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

Normally a compiler implementation will perform the above concatenation using methods involving a [StringBuilder](#) under the hood. When compiled, the code would look similar to the below:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

StringBuilder has several overloaded methods for appending different types, for example, to append an int instead of a String. For example, an implementation can convert:

```
String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2
```

to the following:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

The above examples illustrate a simple concatenation operation that is effectively done in a single place in the code. The concatenation involves a single instance of the StringBuilder. In some cases, a concatenation is carried out in a cumulative way such as in a loop:

```
String result = "";
for(int i = 0; i < array.length; i++) {
    result += extractElement(array[i]);
}
return result;
```

In such cases, the compiler optimization is usually not applied, and each iteration will create a new StringBuilder object. This can be optimized by explicitly transforming the code to use a single StringBuilder:

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
}
```

```
}
```

```
return result.toString();
```

一个StringBuilder将初始化为仅包含16个字符的空空间。如果你事先知道将构建更大的字符串，预先以足够的大小初始化它是有益的，这样内部缓冲区就不需要调整大小：

```
StringBuilder buf = new StringBuilder(30); // 默认是16个字符
buf.append("0123456789");
buf.append("0123456789"); // 否则会导致内部缓冲区重新分配
String result = buf.toString(); // 生成字符串的20字符副本
```

如果你要生成许多字符串，建议重用StringBuilder：

```
StringBuilder buf = new StringBuilder(100);
for (int i = 0; i < 100; i++) {
    buf.setLength(0); // 清空缓冲区
    buf.append("这是第 ").append(i).append(")");
    outputfile.write(buf.toString());
}
```

如果（且仅当）多个线程写入同一个缓冲区时，使用StringBuffer，它是StringBuilder的同步版本。但因为通常只有单个线程写入缓冲区，通常使用StringBuilder且不加同步会更快。

使用 concat() 方法：

```
String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2); // "Hello world"
```

这会返回一个新字符串，即在 string1 末尾添加了 string2。你也可以用 concat() 方法连接字符串字面量，例如：

```
"My name is ".concat("Buyya");
```

第11.8节：子字符串

```
字符串 s = "this is an example";
字符串 a = s.substring(11); // a 将保存从第11个字符开始直到结尾的字符串
("example")
字符串 b = s.substring(5, 10); // b 将保存从第5个字符开始直到第10个字符之前的字符串
("is an")
字符串 b = s.substring(5, b.length()-3); // b 将保存从第5个字符开始直到 b 的长度减去3之前的字符串
("is an exam")
```

子字符串也可以用于切片并在原字符串中添加或替换字符。例如，你遇到一个包含中文字符的中文日期，但你想将其存储为格式良好的日期字符串。

```
字符串 datestring = "2015年11月17日"
datestring = datestring.substring(0, 4) + "—" + datestring.substring(5, 7) + "—" +
datestring.substring(8, 10);
//结果将是 2015-11-17
```

substring 方法用于提取字符串的一部分。当只提供一个参数时，该参数表示起始位置，

```
}
```

```
return result.toString();
```

A StringBuilder will be initialized with an empty space of only 16 characters. If you know in advance that you will be building larger strings, it can be beneficial to initialize it with sufficient size in advance, so that the internal buffer does not need to be resized:

```
StringBuilder buf = new StringBuilder(30); // Default is 16 characters
buf.append("0123456789");
buf.append("0123456789"); // Would cause a reallocation of the internal buffer otherwise
String result = buf.toString(); // Produces a 20-chars copy of the string
```

If you are producing many strings, it is advisable to reuse StringBuilders:

```
StringBuilder buf = new StringBuilder(100);
for (int i = 0; i < 100; i++) {
    buf.setLength(0); // Empty buffer
    buf.append("This is line ").append(i).append('\n');
    outputfile.write(buf.toString());
}
```

If (and only if) multiple threads are writing to the same buffer, use [StringBuffer](#), which is a **synchronized** version of StringBuilder. But because usually only a single thread writes to a buffer, it is usually faster to use StringBuilder without synchronization.

Using concat() method:

```
String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2); // "Hello world"
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in:

```
"My name is ".concat("Buyya");
```

Section 11.8: Substrings

```
String s = "this is an example";
String a = s.substring(11); // a will hold the string starting at character 11 until the end
("example")
String b = s.substring(5, 10); // b will hold the string starting at character 5 and ending right
before character 10 ("is an")
String b = s.substring(5, b.length()-3); // b will hold the string starting at character 5 ending
right before b's length is out of 3 ("is an exam")
```

Substrings may also be applied to slice and add/replace character into its original String. For instance, you faced a Chinese date containing Chinese characters but you want to store it as a well format Date String.

```
String datestring = "2015年11月17日"
datestring = datestring.substring(0, 4) + "—" + datestring.substring(5, 7) + "—" +
datestring.substring(8, 10);
//Result will be 2015-11-17
```

The [substring](#) method extracts a piece of a [String](#). When provided one parameter, the parameter is the start and

该部分一直延伸到String的末尾。当给出两个参数时，第一个参数是起始字符，第二个参数是结束字符后面的索引（该索引处的字符不包括在内）。一种简单的检查方法是用第二个参数减去第一个参数，结果应为字符串的预期长度。

版本 < Java SE 7

在 JDK 7u6 之前的版本中，substring 方法会实例化一个 String，该字符串共享与原始 String 相同的底层 char[]，并且内部的 offset 和 count 字段被设置为结果的起始位置和长度。这样的共享可能导致内存泄漏，可以通过调用 new String(s.substring(...)) 来强制创建一个副本，从而避免内存泄漏，之后底层的 char[] 可以被垃圾回收。

版本 ≥ Java SE 7

从 JDK 7u6 开始，substring 方法总是复制整个底层的 char[] 数组，使得复杂度相比之前的常数变为线性，但同时保证了不会发生内存泄漏。

第 11.9 节：平台无关的新行分隔符

由于新行分隔符因平台而异（例如 Unix 类系统上为
，Windows 上为 \r

`System.getProperty("line.separator")`

版本 ≥ Java SE 7

由于新行分隔符非常常用，从 Java 7 开始提供了一个快捷方法，返回的结果与上述代码完全相同：

`System.lineSeparator()`

注意：由于程序执行期间新行分隔符极少变化，建议将其存储在静态常量变量中，而不是每次需要时都从系统属性中获取。

使用 String.format 时，使用 %n 而非 \n 或 '\r' 来输出平台无关的新行分隔符。

`System.out.println(String.format("line 1: %s.%nline 2: %s%n", lines[0], lines[1]));`

第11.10节：字符串反转

有几种方法可以将字符串反转，使其变成倒序。

1. StringBuilder/StringBuffer :

```
String code = "code";
System.out.println(code);

StringBuilder sb = new StringBuilder(code);
code = sb.reverse().toString();

System.out.println(code);
```

2. 字符数组：

the piece extends until the end of the String. When given two parameters, the first parameter is the starting character and the second parameter is the index of the character right after the end (the character at the index is not included). An easy way to check is the subtraction of the first parameter from the second should yield the expected length of the string.

Version < Java SE 7

In JDK <7u6 versions the substring method instantiates a String that shares the same backing char[] as the original String and has the internal offset and count fields set to the result start and length. Such sharing may cause memory leaks, that can be prevented by calling new String(s.substring(...)) to force creation of a copy, after which the char[] can be garbage collected.

Version ≥ Java SE 7

From JDK 7u6 the substring method always copies the entire underlying char[] array, making the complexity linear compared to the previous constant one but guaranteeing the absence of memory leaks at the same time.

Section 11.9: Platform independent new line separator

Since the new line separator varies from platform to platform (e.g. \n on Unix-like systems or \r\n on Windows) it is often necessary to have a platform-independent way of accessing it. In Java it can be retrieved from a system property:

`System.getProperty("line.separator")`

Version ≥ Java SE 7

Because the new line separator is so commonly needed, from Java 7 on a shortcut method returning exactly the same result as the code above is available:

`System.lineSeparator()`

Note: Since it is very unlikely that the new line separator changes during the program's execution, it is a good idea to store it in a static final variable instead of retrieving it from the system property every time it is needed.

When using String.format, use %n rather than \n or '\r\n' to output a platform independent new line separator.

`System.out.println(String.format('line 1: %s.%nline 2: %s%n', lines[0], lines[1]));`

Section 11.10: Reversing Strings

There are a couple ways you can reverse a string to make it backwards.

1. StringBuilder/StringBuffer:

```
String code = "code";
System.out.println(code);

StringBuilder sb = new StringBuilder(code);
code = sb.reverse().toString();

System.out.println(code);
```

2. Char array:

```

String code = "code";
System.out.println(code);

char[] array = code.toCharArray();
for (int index = 0, mirroredIndex = array.length - 1; index < mirroredIndex; index++, mirroredIndex--) {
    char temp = array[index];
    array[index] = array[mirroredIndex];
    array[mirroredIndex] = temp;
}

// 打印反转后的内容
System.out.println(new String(array));

```

```

String code = "code";
System.out.println(code);

char[] array = code.toCharArray();
for (int index = 0, mirroredIndex = array.length - 1; index < mirroredIndex; index++, mirroredIndex--) {
    char temp = array[index];
    array[index] = array[mirroredIndex];
    array[mirroredIndex] = temp;
}

// print reversed
System.out.println(new String(array));

```

第11.11节：为自定义对象添加toString()方法

假设你定义了以下Person类：

```

public class Person {

    String name;
    int age;

    public Person (int age, String name) {
        this.age = age;
        this.name = name;
    }
}

```

如果你实例化一个新的Person对象：

```
Person person = new Person(25, "John");
```

然后在你的代码中使用以下语句来打印对象：

```
System.out.println(person.toString());
```

[Ideone 在线演示](#)

你将得到类似以下的输出：

```
Person@7ab89d
```

这是在Object类中定义的toString()方法的实现结果，Object是Person的超类。关于[Object.toString\(\)](#)的文档说明：

Object类的toString方法返回一个字符串，该字符串由对象所属类的名称、符号`@`以及对象哈希码的无符号十六进制表示组成。换句话说，该方法返回的字符串等于以下表达式的值：

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

因此，为了获得有意义的输出，你需要重写toString()方法：

```
@Override
```

Section 11.11: Adding toString() method for custom objects

Suppose you have defined the following Person class:

```

public class Person {

    String name;
    int age;

    public Person (int age, String name) {
        this.age = age;
        this.name = name;
    }
}

```

If you instantiate a new Person object:

```
Person person = new Person(25, "John");
```

and later in your code you use the following statement in order to print the object:

```
System.out.println(person.toString());
```

[Live Demo on Ideone](#)

you'll get an output similar to the following:

```
Person@7ab89d
```

This is the result of the implementation of the `toString()` method defined in the `Object` class, a superclass of Person. The documentation of `Object.toString()` states:

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@`, and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

So, for meaningful output, you'll have to **override** the `toString()` method:

```
@Override
```

```
public String toString() {
    return "我的名字是 " + this.name + ", 我的年龄是 " + this.age;
}
```

现在输出将是：

我叫约翰，年龄是25岁

你也可以写

```
System.out.println(person);
```

[Ideone 在线演示](#)

事实上，`println()`会隐式调用对象的`toString`方法。

第11.12节：移除字符串开头和结尾的空白字符

`trim()`方法返回一个新的字符串，去除了开头和结尾的空白字符。

```
String s = new String("Hello World!! ");
String t = s.trim(); // t = "Hello World!!"
```

如果你对一个没有空白字符需要移除的字符串使用`trim`，返回的将是同一个字符串实例。

注意`trim()`方法有它自己对空白字符的定义，这与
字符.是空白字符() 方法：

- 所有ASCII控制字符，代码范围为U+0000到U+0020，都被视为空白字符，并会被`trim()`方法移除。这包括U+0020的“空格 (SPACE)”、U+0009的“制表符 (CHARACTER TABULATION)”、U+000A的“换行符 (LINE FEED)”和U+000D的“回车符 (CARRIAGE RETURN)”，但也包括像U+0007的“响铃 (BELL)”等字符。
- Unicode空白字符如U+00A0的“不间断空格 (NO-BREAK SPACE)”或U+2003的“EM空格 (EM SPACE)”不被`trim()`方法识别。

第11.13节：不区分大小写的switch

版本 ≥ Java SE 7

`switch`本身不能参数化为不区分大小写，但如果绝对需要，可以通过使用`toLowerCase()`或`toUpperCase()`来对输入字符串实现不区分大小写的行为：

```
switch(myString.toLowerCase()) {
    case "case1" :
        ...
        break;
    case "case2" :
        ...
        break;
}
```

注意

- 区域设置 (Locale) 可能会影响大小写转换的结果！

```
public String toString() {
    return "My name is " + this.name + " and my age is " + this.age;
}
```

Now the output will be:

My name is John and my age is 25

You can also write

```
System.out.println(person);
```

[Live Demo on Ideone](#)

In fact, `println()` implicitly invokes the `toString` method on the object.

Section 11.12: Remove Whitespace from the Beginning and End of a String

The `trim()` method returns a new String with the leading and trailing whitespace removed.

```
String s = new String("Hello World!! ");
String t = s.trim(); // t = "Hello World!!"
```

If you `trim` a String that doesn't have any whitespace to remove, you will be returned the same String instance.

Note that the `trim()` method has its own notion of whitespace, which differs from the notion used by the `Character.isWhitespace()` method:

- All ASCII control characters with codes U+0000 to U+0020 are considered whitespace and are removed by `trim()`. This includes U+0020 'SPACE', U+0009 'CHARACTER TABULATION', U+000A 'LINE FEED' and U+000D 'CARRIAGE RETURN' characters, but also the characters like U+0007 'BELL'.
- Unicode whitespace like U+00A0 'NO-BREAK SPACE' or U+2003 'EM SPACE' are not recognized by `trim()`.

Section 11.13: Case insensitive switch

Version ≥ Java SE 7

`switch` itself can not be parameterised to be case insensitive, but if absolutely required, can behave insensitive to the input string by using `toLowerCase()` or `toUpperCase`:

```
switch (myString.toLowerCase()) {
    case "case1" :
        ...
        break;
    case "case2" :
        ...
        break;
}
```

Beware

- `Locale` might affect how changing cases happen!

- 必须注意标签中不能有大写字母——这些标签永远不会被执行！

- Care must be taken not to have any uppercase characters in the labels - those will never get executed!

第11.14节：字符串部分替换

替换的两种方式：通过正则表达式或精确匹配。

注意：原始的String对象不会改变，返回值包含修改后的字符串。

精确匹配

用另一个单字符替换单个字符：

```
String replace(char oldChar, char newChar)
```

返回一个新字符串，该字符串是将此字符串中所有出现的oldChar替换为newChar后的结果。

```
String s = "popcorn";
System.out.println(s.replace('p', 'W'));
```

结果：

WoWcorn

用另一段字符替换字符序列：

```
String replace(CharSequence target, CharSequence replacement)
```

替换此字符串中每个与字面目标序列匹配的子字符串为指定的字面替换序列。

```
String s = "metal petal et al.";
System.out.println(s.replace("etal", "etallica"));
```

结果：

metallica petallica et al.

正则表达式

注意：分组使用\$字符来引用分组，如\$1。

替换所有匹配项：

```
String replaceAll(String regex, String replacement)
```

替换此字符串中每个匹配给定正则表达式的子字符串为给定的替换内容。

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)", "$1lica"));
```

结果：

Section 11.14: Replacing parts of Strings

Two ways to replace: by regex or by exact match.

Note: the original String object will be unchanged, the return value holds the changed String.

Exact match

Replace single character with another single character:

```
String replace(char oldChar, char newChar)
```

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

```
String s = "popcorn";
System.out.println(s.replace('p', 'W'));
```

Result:

WoWcorn

Replace sequence of characters with another sequence of characters:

```
String replace(CharSequence target, CharSequence replacement)
```

Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.

```
String s = "metal petal et al.";
System.out.println(s.replace("etal", "etallica"));
```

Result:

metallica petallica et al.

Regex

Note: the grouping uses the \$ character to reference the groups, like \$1.

Replace all matches:

```
String replaceAll(String regex, String replacement)
```

Replaces each substring of this string that matches the given regular expression with the given replacement.

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)", "$1lica"));
```

Result:

spiral metallica petalica 等人。

仅替换第一个匹配项：

```
String replaceFirst(String regex, String replacement)
```

将此字符串中第一个匹配给定正则表达式的子字符串替换为给定的替换内容

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)", "$1lica"));
```

结果：

```
spiral metalica petal 等人。
```

第11.15节：获取字符串的长度

为了获取一个String对象的长度，调用其length()方法。长度等于字符串中UTF-16代码单元（char）的数量。

```
String str = "Hello, World!";
System.out.println(str.length()); // 输出13
```

[Ideone 在线演示](#)

字符串中的char是UTF-16值。Unicode码点值 $\geq 0x1000$ （例如，大多数表情符号）使用两个char位置。要计算字符串中Unicode码点的数量，无论每个码点是否适合UTF-16char值，都可以使用codePointCount方法：

```
int length = str.codePointCount(0, str.length());
```

你也可以使用代码点的流（Stream），从Java 8开始支持：

```
int length = str.codePoints().count();
```

第11.16节：获取字符串中的第n个字符

```
String str = "My String";
System.out.println(str.charAt(0)); // "M"
System.out.println(str.charAt(1)); // "y"
System.out.println(str.charAt(2)); // " "
System.out.println(str.charAt(str.length-1)); // 最后一个字符 "g"
```

要获取字符串中的第n个字符，只需在字符串（String）上调用charAt(n)，其中 n 是你想要获取的字符的索引

注意：索引 n 从0开始，所以第一个元素的索引是n=0。

第11.17节：统计字符串中子串或

spiral metallica petalica et al.

Replace first match only:

```
String replaceFirst(String regex, String replacement)
```

Replaces the first substring of this string that matches the given regular expression with the given replacement

```
String s = "spiral metal petal et al.";
System.out.println(s.replaceAll("(\\w*etal)", "$1lica"));
```

Result:

```
spiral metalica petal et al.
```

Section 11.15: Getting the length of a String

In order to get the length of a String object, call the length() method on it. The length is equal to the number of UTF-16 code units (chars) in the string.

```
String str = "Hello, World!";
System.out.println(str.length()); // Prints out 13
```

[Live Demo on Ideone](#)

A char in a String is UTF-16 value. Unicode codepoints whose values are $\geq 0x1000$ (for example, most emojis) use two char positions. To count the number of Unicode codepoints in a String, regardless of whether each codepoint fits in a UTF-16 char value, you can use the codePointCount method:

```
int length = str.codePointCount(0, str.length());
```

You can also use a Stream of codepoints, as of Java 8:

```
int length = str.codePoints().count();
```

Section 11.16: Getting the nth character in a String

```
String str = "My String";
System.out.println(str.charAt(0)); // "M"
System.out.println(str.charAt(1)); // "y"
System.out.println(str.charAt(2)); // " "
System.out.println(str.charAt(str.length-1)); // Last character "g"
```

To get the nth character in a string, simply call charAt(n) on a String, where n is the index of the character you would like to retrieve

NOTE: index n is starting at 0, so the first element is at n=0.

Section 11.17: Counting occurrences of a substring or

字符的出现次数

countMatches 方法来自 `org.apache.commons.lang3.StringUtils`, 通常用于计算字符串中子串或字符的出现次数：

```
import org.apache.commons.lang3.StringUtils;

String text = "一条鱼, 两条鱼, 红鱼, 蓝鱼";

// 计算子字符串出现的次数
String stringTarget = "fish";
int stringOccurrences = StringUtils.countMatches(text, stringTarget); // 4

// 计算字符出现的次数
char charTarget = ',';
int charOccurrences = StringUtils.countMatches(text, charTarget); // 3
```

否则, 使用标准Java API也可以实现相同功能, 你可以使用正则表达式：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

String text = "一条鱼, 两条鱼, 红鱼, 蓝鱼";
System.out.println(countStringInString("fish", text)); // 输出 4
System.out.println(countStringInString(",", text)); // 输出 3

public static int countStringInString(String search, String text) {
    Pattern pattern = Pattern.compile(search);
    Matcher matcher = pattern.matcher(text);

    int stringOccurrences = 0;
    while (matcher.find()) {
        stringOccurrences++;
    }
    return stringOccurrences;
}
```

character in a string

countMatches method from `org.apache.commons.lang3.StringUtils` is typically used to count occurrences of a substring or character in a `String`:

```
import org.apache.commons.lang3.StringUtils;

String text = "One fish, two fish, red fish, blue fish";

// count occurrences of a substring
String stringTarget = "fish";
int stringOccurrences = StringUtils.countMatches(text, stringTarget); // 4

// count occurrences of a char
char charTarget = ',';
int charOccurrences = StringUtils.countMatches(text, charTarget); // 3
```

Otherwise for does the same with standard Java API's you could use Regular Expressions:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

String text = "One fish, two fish, red fish, blue fish";
System.out.println(countStringInString("fish", text)); // prints 4
System.out.println(countStringInString(",", text)); // prints 3

public static int countStringInString(String search, String text) {
    Pattern pattern = Pattern.compile(search);
    Matcher matcher = pattern.matcher(text);

    int stringOccurrences = 0;
    while (matcher.find()) {
        stringOccurrences++;
    }
    return stringOccurrences;
}
```

第12章：StringBuffer

Java StringBuffer 类简介。

第12.1节：StringBuffer 类

要点：

- 用于创建可变（可修改）的字符串。
- 可变的：可以被更改的。
- 是线程安全的，即多个线程不能同时访问它。

方法：

- public synchronized StringBuffer append(String s)
- public synchronized StringBuffer insert(int offset, String s)
- public synchronized StringBuffer replace(int startIndex, int endIndex, String str)
- public synchronized StringBuffer delete(int startIndex, int endIndex)
- public synchronized StringBuffer reverse()
- public int capacity()
- public void ensureCapacity(int minimumCapacity)
- public char charAt(int index)
- public int length()
- public String substring(int beginIndex)
- public String substring(int beginIndex, int endIndex)

示例显示 String 和 StringBuffer 实现之间的区别：

```
class Test {  
    public static void main(String args[])  
    {  
        String str = "study";  
        str.concat("tonight");  
        System.out.println(str);      // 输出: study  
  
        StringBuffer strB = new StringBuffer("study");  
        strB.append("tonight");  
        System.out.println(strB);    // 输出: studytonight  
    }  
}
```

Chapter 12: StringBuffer

Introduction to Java StringBuffer class.

Section 12.1: String Buffer class

Key Points:

- used to created mutable (modifiable) string.
- **Mutable**: Which can be changed.
- is thread-safe i.e. multiple threads cannot access it simultaneously.

Methods:

- public synchronized StringBuffer append(String s)
- public synchronized StringBuffer insert(int offset, String s)
- public synchronized StringBuffer replace(int startIndex, int endIndex, String str)
- public synchronized StringBuffer delete(int startIndex, int endIndex)
- public synchronized StringBuffer reverse()
- public int capacity()
- public void ensureCapacity(int minimumCapacity)
- public char charAt(int index)
- public int length()
- public String substring(int beginIndex)
- public String substring(int beginIndex, int endIndex)

Example Showing difference between String and String Buffer implementation:

```
class Test {  
    public static void main(String args[])  
    {  
        String str = "study";  
        str.concat("tonight");  
        System.out.println(str);      // Output: study  
  
        StringBuffer strB = new StringBuffer("study");  
        strB.append("tonight");  
        System.out.println(strB);    // Output: studytonight  
    }  
}
```

第13章：StringBuilder

Java StringBuilder类用于创建可变（可修改）的字符串。Java StringBuilder类与StringBuffer类相同，只是它是非同步的。自JDK 1.5起提供。

第13.1节：比较StringBuffer、StringBuilder、Formatter和StringJoiner

StringBuffer、StringBuilder、Formatter和StringJoiner类是Java SE的实用类，主要用于从其他信息组装字符串：

- StringBuffer类自Java 1.0起存在，提供了多种方法来构建和修改包含字符序列的“缓冲区”。
- StringBuilder类在Java 5中添加，用于解决原始StringBuffer类的性能问题。这两个类的API基本相同。StringBuffer和StringBuilder的主要区别是前者是线程安全且同步的，后者则不是。

此示例展示了如何使用StringBuilder：

```
int one = 1;
String color = "red";
StringBuilder sb = new StringBuilder();sb.append("One=").append(one).append(", Color=").append(color).append(");System.out.print(sb);
// 打印 "One=1, Colour=red" 后跟一个 ASCII 换行符。
```

(StringBuffer 类的用法相同：只需将上面的 StringBuilder 改为 StringBuffer)

StringBuffer 和 StringBuilder 类适合用于组装和修改字符串；即它们提供了替换和删除字符以及以各种方式添加字符的方法。剩下的两个类专门用于组装字符串的任务。

- Formatter 类在 Java 5 中添加，松散地模仿了 C 标准库中的 sprintf 函数。它接受一个带有嵌入式 format specifiers 的 format 字符串和一系列其他参数，通过将参数转换为文本并替换格式说明符来生成字符串。格式说明符的细节说明了参数如何被转换为文本。
- StringJoiner 类在 Java 8 中添加。它是一个专用的格式化器，用于简洁地格式化带有分隔符的字符串序列。它设计了一个 fluent API，并且可以与 Java 8 流一起使用。

以下是一些 Formatter 使用的典型示例：

```
// 这与上面的 StringBuilder 示例做了同样的事情
int one = 1;
String color = "red";
Formatter f = new Formatter();
System.out.print(f.format("One=%d, colour=%s%n", one, color));
// 打印 "One=1, Colour=red", 后跟平台的换行符

// 使用 `String.format` 便捷方法实现相同功能
System.out.print(String.format("One=%d, color=%s%n", one, color));
```

Chapter 13: StringBuilder

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Section 13.1: Comparing StringBuffer, StringBuilder, Formatter and StringJoiner

The [StringBuffer](#), [StringBuilder](#), [Formatter](#) and [StringJoiner](#) classes are Java SE utility classes that are primarily used for assembling strings from other information:

- The [StringBuffer](#) class has been present since Java 1.0, and provides a variety of methods for building and modifying a "buffer" containing a sequence of characters.
- The [StringBuilder](#) class was added in Java 5 to address performance issues with the original [StringBuffer](#) class. The APIs for the two classes are essentially the same. The main difference between [StringBuffer](#) and [StringBuilder](#) is that the former is thread-safe and synchronized and the latter is not.

This example shows how [StringBuilder](#) is can be used:

```
int one = 1;
String color = "red";
StringBuilder sb = new StringBuilder();
sb.append("One=").append(one).append(", Color=").append(color).append('\n');
System.out.print(sb);
// Prints "One=1, Colour=red" followed by an ASCII newline.
```

(The [StringBuffer](#) class is used the same way: just change [StringBuilder](#) to [StringBuffer](#) in the above)

The [StringBuffer](#) and [StringBuilder](#) classes are suitable for both assembling and modifying strings; i.e they provide methods for replacing and removing characters as well as adding them in various. The remaining two classes are specific to the task of assembling strings.

- The [Formatter](#) class was added in Java 5, and is loosely modeled on the [sprintf](#) function in the C standard library. It takes a *format* string with embedded *format specifiers* and a sequences of other arguments, and generates a string by converting the arguments into text and substituting them in place of the *format specifiers*. The details of the *format specifiers* say how the arguments are converted into text.
- The [StringJoiner](#) class was added in Java 8. It is a special purpose formatter that succinctly formats a sequence of strings with separators between them. It is designed with a *fluent API*, and can be used with Java 8 streams.

Here are some typical examples of [Formatter](#) usage:

```
// This does the same thing as the StringBuilder example above
int one = 1;
String color = "red";
Formatter f = new Formatter();
System.out.print(f.format("One=%d, colour=%s%n", one, color));
// Prints "One=1, Colour=red" followed by the platform's line separator

// The same thing using the `String.format` convenience method
System.out.print(String.format("One=%d, color=%s%n", one, color));
```

StringJoiner 类不适合上述任务，下面是格式化字符串数组的示例。

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// 打印 "[A, B, C]"
```

这四个类的使用场景可以总结如下：

- StringBuilder 适用于任何字符串组装或字符串修改任务。
- StringBuffer 仅在需要线程安全版本的StringBuilder时使用。
- Formatter 提供了更丰富的字符串格式化功能，但效率不如StringBuilder。这是因为每次调用Formatter.format(..)都会涉及：
 - 解析格式字符串，
 - 创建并填充一个varargs数组，
 - 并对任何基本类型参数进行自动装箱。
- StringJoiner提供了带分隔符的字符串序列的简洁高效格式化，但不适用于其他格式化任务。

第13.2节：重复字符串n次

问题：创建一个包含String s重复 n次的String。

最简单的方法是反复连接String

```
final int n = ...
final String s = ...
String result = "";

for (int i = 0; i < n; i++) {
    result += s;
}
```

这会创建 n个新的字符串实例，包含1到 n次的 s重复，导致运行时间为 $O(s.length() * n^2) = O(s.length() * (1+2+...+(n-1)+n))$ 。

为避免此问题，应使用StringBuilder，它允许在 $O(s.length() * n)$ 时间内创建String，示例如下：

```
final int n = ...
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

The StringJoiner class is not ideal for the above task, so here is an example of a formatting an array of strings.

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
for (String s : new String[]{"A", "B", "C"}) {
    sj.add(s);
}
System.out.println(sj);
// Prints "[A, B, C]"
```

The use-cases for the 4 classes can be summarized:

- StringBuilder suitable for any string assembly OR string modification task.
- StringBuffer use (only) when you require a thread-safe version of StringBuilder.
- Formatter provides much richer string formatting functionality, but is not as efficient as StringBuilder. This is because each call to Formatter.format(...) entails:
 - parsing the format string,
 - creating and populate a varargs array, and
 - autoboxing any primitive type arguments.
- StringJoiner provides succinct and efficient formatting of a sequence of strings with separators, but is not suitable for other formatting tasks.

Section 13.2: Repeat a String n times

Problem: Create a String containing n repetitions of a String s.

The trivial approach would be repeatedly concatenating the String

```
final int n = ...
final String s = ...
String result = "";

for (int i = 0; i < n; i++) {
    result += s;
}
```

This creates n new string instances containing 1 to n repetitions of s resulting in a runtime of $O(s.length() * n^2) = O(s.length() * (1+2+...+(n-1)+n))$.

To avoid this StringBuilder should be used, which allows creating the String in $O(s.length() * n)$ instead:

```
final int n = ...
final String s = ...

StringBuilder builder = new StringBuilder();

for (int i = 0; i < n; i++) {
    builder.append(s);
}

String result = builder.toString();
```

第14章：字符串分词器

java.util.StringTokenizer 类允许你将字符串拆分成标记。这是一种简单的字符串拆分方式。

分隔符集合（用于分隔标记的字符）可以在创建时指定，也可以在每个标记的基础上指定。

第14.1节： StringTokenizer 按空格拆分

```
import java.util.StringTokenizer;
public class Simple{
    public static void main(String args[]){
        StringTokenizer st = new StringTokenizer("apple ball cat dog", " ");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

输出：

苹果

球

猫

狗

第14.2节：使用逗号","分割的 StringTokenizer

```
public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
}
```

输出：

苹果

球 猫\

狗

Chapter 14: String Tokenizer

The java.util.StringTokenizer class allows you to break a string into tokens. It is simple way to break string.

The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.

Section 14.1: StringTokenizer Split by space

```
import java.util.StringTokenizer;
public class Simple{
    public static void main(String args[]){
        StringTokenizer st = new StringTokenizer("apple ball cat dog", " ");
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

Output:

apple

ball

cat

dog

Section 14.2: StringTokenizer Split by comma ','

```
public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer("apple,ball cat,dog", ",");
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
}
```

Output:

apple

ball cat

dog

第15章：将字符串拆分为固定长度的部分

第15.1节：将字符串拆分为已知长度的子字符串

关键是使用带有正则表达式\G的回顾断言，它表示“上一次匹配的结束位置”：

```
String[] parts = str.split("(?=<=\G.{8})");
```

该正则表达式匹配上一次匹配结束后紧接的8个字符。由于此处匹配是零宽度的，我们可以更简单地理解为“上一次匹配之后的8个字符”。

方便的是，\G 被初始化为输入的起始位置，因此它也适用于输入的第一部分。

第15.2节：将字符串拆分为多个可变长度的子字符串

与已知长度的示例相同，但将长度插入正则表达式中：

```
int length = 5;
String[] parts = str.split("(?=<=\G.{ " + length + " })");
```

Chapter 15: Splitting a string into fixed length parts

Section 15.1: Break a string up into substrings all of a known length

The trick is to use a look-behind with the regex \G, which means "end of previous match":

```
String[] parts = str.split("(?=<=\G.{8})");
```

The regex matches 8 characters after the end of the last match. Since in this case the match is zero-width, we could more simply say "8 characters after the last match".

Conveniently, \G is initialized to start of input, so it works for the first part of the input too.

Section 15.2: Break a string up into substrings all of variable length

Same as the known length example, but insert the length into regex:

```
int length = 5;
String[] parts = str.split("(?=<=\G.{ " + length + " })");
```

第16章：日期类

参数

无参数 使用分配时间（精确到毫秒）创建一个新的Date对象

long date 使用自“纪元”（1970年1月1日00:00:00 GMT）以来的毫秒数创建一个新的Date对象，时间被设置为该毫秒数

说明

java.util.Date 到 java.sql.Date 的转换通常是在需要将 Date 对象写入数据库时进行的。

java.sql.Date 是对毫秒值的封装，JDBC 用它来标识 SQL DATE 类型在下面的示例中，我们使用了 java.util.Date() 构造函数，该构造函数创建一个 Date 对象并初始化为表示最接近毫秒的时间。此日期用于 convert(java.util.Date utilDate) 方法中，返回一个 java.sql.Date 对象

示例

```
public class UtilToSqlConversion {  
  
    public static void main(String args[]) {  
        java.util.Date utilDate = new java.util.Date();  
        System.out.println("java.util.Date 是：" + utilDate);  
        java.sql.Date sqlDate = convert(utilDate);  
        System.out.println("java.sql.Date 是：" + sqlDate);  
        DateFormat df = new SimpleDateFormat("dd/MM/YYYY - hh:mm:ss");  
        System.out.println("格式化后的日期是：" + df.format(utilDate));  
    }  
  
    private static java.sql.Date convert(java.util.Date uDate) {  
        java.sql.Date sDate = new java.sql.Date(uDate.getTime());  
        return sDate;  
    }  
}
```

输出

```
java.util.Date 是 : Fri Jul 22 14:40:35 IST 2016  
java.sql.Date 是 : 2016-07-22  
格式化后的日期是 : 22/07/2016 - 02:40:35
```

java.util.Date 包含日期和时间信息，而 java.sql.Date 仅包含日期信息

第16.2节：基本日期输出

使用以下代码和格式字符串yyyy/MM/dd hh:mm:ss，将得到以下输出

```
2016/04/19 11:45:36
```

Chapter 16: Date Class

Parameter

No parameter Creates a new Date object using the allocation time (to the nearest millisecond)

long date Creates a new Date object with the time set to the number of milliseconds since "the epoch" (January 1, 1970, 00:00:00 GMT)

Explanation

java.util.Date to java.sql.Date conversion is usually necessary when a Date object needs to be written in a database.

java.sql.Date is a wrapper around millisecond value and is used by JDBC to identify an SQL DATE type

In the below example, we use the java.util.Date() constructor, that creates a Date object and initializes it to represent time to the nearest millisecond. This date is used in the convert(java.util.Date utilDate) method to return a java.sql.Date object

Example

```
public class UtilToSqlConversion {  
  
    public static void main(String args[]) {  
        java.util.Date utilDate = new java.util.Date();  
        System.out.println("java.util.Date is :" + utilDate);  
        java.sql.Date sqlDate = convert(utilDate);  
        System.out.println("java.sql.Date is :" + sqlDate);  
        DateFormat df = new SimpleDateFormat("dd/MM/YYYY - hh:mm:ss");  
        System.out.println("dateFormated date is :" + df.format(utilDate));  
    }  
  
    private static java.sql.Date convert(java.util.Date uDate) {  
        java.sql.Date sDate = new java.sql.Date(uDate.getTime());  
        return sDate;  
    }  
}
```

Output

```
java.util.Date is : Fri Jul 22 14:40:35 IST 2016  
java.sql.Date is : 2016-07-22  
dateFormated date is : 22/07/2016 - 02:40:35
```

java.util.Date has both date and time information, whereas java.sql.Date only has date information

Section 16.2: A basic date output

Using the following code with the format string yyyy/MM/dd hh:mm:ss, we will receive the following output

```
2016/04/19 11:45:36
```

```

// 定义要使用的格式
String formatString = "yyyy/MM/dd hh:mm:ss";

// 获取当前日期对象
Date date = Calendar.getInstance().getTime();

// 创建格式化器
SimpleDateFormat simpleDateFormat = new SimpleDateFormat(formatString);

// 格式化日期
String formattedDate = simpleDateFormat.format(date);

// 打印它
System.out.println(formattedDate);

// 上述所有代码的单行版本
System.out.println(new SimpleDateFormat("yyyy/MM/dd
hh:mm:ss").format(Calendar.getInstance().getTime()));

```

第16.3节：Java 8 LocalDate 和 LocalDateTime 对象

Date 和 LocalDate 对象不能被精确地相互转换，因为 Date 对象表示特定的日期和时间，而 LocalDate 对象不包含时间或时区信息。然而，如果你只关心实际的日期信息而非时间信息，转换两者之间可能会很有用。

创建一个 LocalDate

```

// 创建一个默认日期
LocalDate lDate = LocalDate.now();

// 根据数值创建日期
lDate = LocalDate.of(2017, 12, 15);

// 从字符串创建日期
lDate = LocalDate.parse("2017-12-15");

// 从时区创建日期
LocalDate.now(ZoneId.systemDefault());

```

创建一个LocalDateTime

```

// 创建一个默认的日期时间
LocalDateTime lDateTime = LocalDateTime.now();

// 从数值创建日期时间
lDateTime = LocalDateTime.of(2017, 12, 15, 11, 30);

// 从字符串创建日期时间
lDateTime = LocalDateTime.parse("2017-12-05T11:30:30");

// 从时区创建日期时间
LocalDateTime.now(ZoneId.systemDefault());

```

LocalDate 与 Date 互转

```

Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();

```

```

// define the format to use
String formatString = "yyyy/MM/dd hh:mm:ss";

// get a current date object
Date date = Calendar.getInstance().getTime();

// create the formatter
SimpleDateFormat simpleDateFormat = new SimpleDateFormat(formatString);

// format the date
String formattedDate = simpleDateFormat.format(date);

// print it
System.out.println(formattedDate);

// single-line version of all above code
System.out.println(new SimpleDateFormat("yyyy/MM/dd
hh:mm:ss").format(Calendar.getInstance().getTime()));

```

Section 16.3: Java 8 LocalDate and LocalDateTime objects

Date and LocalDate objects **cannot** be *exactly* converted between each other since a Date object represents both a specific day and time, while a LocalDate object does not contain time or timezone information. However, it can be useful to convert between the two if you only care about the actual date information and not the time information.

Creates a LocalDate

```

// Create a default date
LocalDate lDate = LocalDate.now();

// Creates a date from values
lDate = LocalDate.of(2017, 12, 15);

// create a date from string
lDate = LocalDate.parse("2017-12-15");

// creates a date from zone
LocalDate.now(ZoneId.systemDefault());

```

Creates a LocalDateTime

```

// Create a default date time
LocalDateTime lDateTime = LocalDateTime.now();

// Creates a date time from values
lDateTime = LocalDateTime.of(2017, 12, 15, 11, 30);

// create a date time from string
lDateTime = LocalDateTime.parse("2017-12-05T11:30:30");

// create a date time from zone
LocalDateTime.now(ZoneId.systemDefault());

```

LocalDate to Date and vice-versa

```

Date date = Date.from(Instant.now());
ZoneId defaultZoneId = ZoneId.systemDefault();

```

```
// Date 转 LocalDate  
LocalDate localDate = date.toInstant().atZone(defaultZoneId).toLocalDate();
```

```
// LocalDate 转 Date  
Date.from(localDate.atStartOfDay(defaultZoneId).toInstant());
```

LocalDateTime 与 Date 互转

```
Date date = Date.from(Instant.now());  
ZoneId defaultZoneId = ZoneId.systemDefault();
```

```
// Date 转 LocalDateTime  
LocalDateTime localDateTime = date.toInstant().atZone(defaultZoneId).toLocalDateTime();
```

```
// LocalDateTime 转 Date  
Date out = Date.from(localDateTime.atZone(defaultZoneId).toInstant());
```

第16.4节：创建特定日期

虽然Java的Date类有多个构造函数，但你会注意到大多数都已被弃用。直接创建Date实例的唯一可接受方式是使用无参数构造函数，或者传入一个long类型的值（自标准基准时间以来的毫秒数）。除非你想获取当前日期或已经有另一个Date实例，否则这两种方式都不太方便。

要创建一个新的日期，你需要一个Calendar实例。然后你可以将Calendar实例设置为所需的日期。

```
Calendar c = Calendar.getInstance();
```

这会返回一个设置为当前时间的新Calendar实例。Calendar有许多方法可以修改其日期和时间，或者直接设置。在本例中，我们将其设置为一个特定日期。

```
c.set(1974, 6, 2, 8, 0, 0);  
Date d = c.getTime();
```

getTime方法返回我们需要的Date实例。请记住，Calendar的set方法只设置一个或多个字段，并不会全部设置。也就是说，如果你设置了年份，其他字段保持不变。

陷阱

在许多情况下，这段代码片段能够实现其目的，但请注意日期/时间的两个重要部分未被定义。

- 参数(1974, 6, 2, 8, 0, 0)是在默认时区内解释的，该时区在其他地方定义。
- 毫秒数未被设置为零，而是在创建Calendar实例时从系统时钟中获取填充。

第16.5节：将日期转换为特定字符串格式

SimpleDateFormat类中的format()方法通过使用提供的pattern string帮助将Date对象转换为特定格式的String对象。

```
Date today = new Date();
```

```
// Date to LocalDate  
LocalDate localDate = date.toInstant().atZone(defaultZoneId).toLocalDate();
```

```
// LocalDate to Date  
Date.from(localDate.atStartOfDay(defaultZoneId).toInstant());
```

LocalDateTime to Date and vice-versa

```
Date date = Date.from(Instant.now());  
ZoneId defaultZoneId = ZoneId.systemDefault();
```

```
// Date to LocalDateTime  
LocalDateTime localDateTime = date.toInstant().atZone(defaultZoneId).toLocalDateTime();
```

```
// LocalDateTime to Date  
Date out = Date.from(localDateTime.atZone(defaultZoneId).toInstant());
```

Section 16.4: Creating a Specific Date

While the Java Date class has several constructors, you'll notice that most are deprecated. The only acceptable way of creating a Date instance directly is either by using the empty constructor or passing in a long (number of milliseconds since standard base time). Neither are handy unless you're looking for the current date or have another Date instance already in hand.

To create a new date, you will need a Calendar instance. From there you can set the Calendar instance to the date that you need.

```
Calendar c = Calendar.getInstance();
```

This returns a new Calendar instance set to the current time. Calendar has many methods for mutating its date and time or setting it outright. In this case, we'll set it to a specific date.

```
c.set(1974, 6, 2, 8, 0, 0);  
Date d = c.getTime();
```

The getTime method returns the Date instance that we need. Keep in mind that the Calendar set methods only set one or more fields, they do not set them all. That is, if you set the year, the other fields remain unchanged.

PITFALL

In many cases, this code snippet fulfills its purpose, but keep in mind that two important parts of the date/time are not defined.

- the (1974, 6, 2, 8, 0, 0) parameters are interpreted within the default timezone, defined somewhere else,
- the milliseconds are not set to zero, but filled from the system clock at the time the Calendar instance is created.

Section 16.5: Converting Date to a certain String format

format() from SimpleDateFormat class helps to convert a Date object into certain format String object by using the supplied pattern string.

```
Date today = new Date();
```

```
SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //这里指定了模式  
System.out.println(dateFormat.format(today)); //25-Feb-16
```

可以通过使用applyPattern()再次应用模式

```
dateFormat.applyPattern("dd-MM-yyyy");  
System.out.println(dateFormat.format(today)); //25-02-2016  
  
dateFormat.applyPattern("dd-MM-yyyy HH:mm:ss E");  
System.out.println(dateFormat.format(today)); //25-02-2016 06:14:33 Thu
```

注意：这里的mm（小写字母m）表示分钟，MM（大写字母M）表示月份。格式化年份时请特别注意：大写的“Y”（Y）表示“年中的周数”，而小写的“y”（y）表示年份。

第16.6节：LocalTime

要仅使用日期的时间部分，请使用LocalTime。你可以通过几种方式实例化一个LocalTime对象

```
1. LocalTime time = LocalTime.now();  
2. time = LocalTime.MIDNIGHT;  
3. time = LocalTime.NOON;  
4. time = LocalTime.of(12, 12, 45);
```

LocalTime 也有一个内置的toString方法，可以非常漂亮地显示格式。

```
System.out.println(time);
```

你还可以从LocalTime对象中获取、添加和减去小时、分钟、秒和纳秒，例如

```
time.plusMinutes(1);  
time.getMinutes();  
time.minusMinutes(1);
```

你可以使用以下代码将其转换为日期对象：

```
LocalTime lTime = LocalTime.now();  
Instant instant = lTime.atDate(LocalDate.of(某年, 某月, 某日)).  
    atZone(ZoneId.systemDefault()).toInstant();  
Date time = Date.from(instant);
```

这个类在定时器类中工作得非常好，可以模拟闹钟。

第16.7节：将格式化的日期字符串转换为日期对象

此方法可用于将格式化的日期字符串转换为Date对象。

```
/**  
 * 使用给定的格式解析日期。  
 *  
 * @param formattedDate 格式化的日期字符串  
 *          * @param dateFormat 用于创建该字符串的日期格式。  
 * @return 日期  
 */  
public static Date parseDate(String formattedDate, String dateFormat) {  
    Date date = null;
```

```
SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MMM-yy"); //pattern is specified here  
System.out.println(dateFormat.format(today)); //25-Feb-16
```

Patterns can be applied again by using applyPattern()

```
dateFormat.applyPattern("dd-MM-yyyy");  
System.out.println(dateFormat.format(today)); //25-02-2016
```

```
dateFormat.applyPattern("dd-MM-yyyy HH:mm:ss E");  
System.out.println(dateFormat.format(today)); //25-02-2016 06:14:33 Thu
```

Note: Here mm (small letter m) denotes minutes and MM (capital M) denotes month. Pay careful attention when formatting years: capital "Y" (Y) indicates the "week in the year" while lower-case "y" (y) indicates the year.

Section 16.6: LocalTime

To use just the time part of a Date use LocalTime. You can instantiate a LocalTime object in a couple ways

```
1. LocalTime time = LocalTime.now();  
2. time = LocalTime.MIDNIGHT;  
3. time = LocalTime.NOON;  
4. time = LocalTime.of(12, 12, 45);
```

LocalTime also has a built in toString method that displays the format very nicely.

```
System.out.println(time);
```

you can also get, add and subtract hours, minutes, seconds, and nanoseconds from the LocalTime object i.e.

```
time.plusMinutes(1);  
time.getMinutes();  
time.minusMinutes(1);
```

You can turn it into a Date object with the following code:

```
LocalTime lTime = LocalTime.now();  
Instant instant = lTime.atDate(LocalDate.of(A_YEAR, A_MONTH, A_DAY)).  
    atZone(ZoneId.systemDefault()).toInstant();  
Date time = Date.from(instant);
```

this class works very nicely within a timer class to simulate an alarm clock.

Section 16.7: Convert formatted string representation of date to Date object

This method can be used to convert a formatted string representation of a date into a Date object.

```
/**  
 * Parses the date using the given format.  
 *  
 * @param formattedDate the formatted date string  
 *          * @param dateFormat the date format which was used to create the string.  
 * @return the date  
 */  
public static Date parseDate(String formattedDate, String dateFormat) {  
    Date date = null;
```

```

SimpleDateFormat objDf = new SimpleDateFormat(dateFormat);
try {
date = objDf.parse(formattedDate);
} catch (ParseException e) {
// 对异常进行相应处理。
}
return date;
}

```

第16.8节：创建日期对象

```

Date date = new Date();
System.out.println(date); // 2016年2月25日星期四 05:03:59 IST

```

这里的Date对象包含了创建该对象时的当前日期和时间。

```

Calendar calendar = Calendar.getInstance();
calendar.set(90, Calendar.DECEMBER, 11);
Date myBirthDate = calendar.getTime();
System.out.println(myBirthDate); // Mon Dec 31 00:00:00 IST 1990

```

日期对象最好通过Calendar实例创建，因为使用数据构造器已被弃用且不推荐。为此，我们需要通过工厂方法获取Calendar类的实例。然后可以使用数字设置年、月和日，或者使用Calendar类提供的月份常量，以提高可读性并减少错误。

```

calendar.set(90, Calendar.DECEMBER, 11, 8, 32, 35);
Date myBirthDate = calendar.getTime();
System.out.println(myBirthDate); // 1990年12月31日 星期一 08:32:35 IST

```

除了日期，我们还可以按小时、分钟和秒的顺序传递时间。

第16.9节：比较日期对象

Calendar、Date 和 LocalDate

版本 < Java SE 8

before、after、compareTo 和 equals 方法

```

// 使用 Calendar 和 Date 对象
final Date today = new Date();
final Calendar calendar = Calendar.getInstance();
calendar.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date birthdate = calendar.getTime();

final Calendar calendar2 = Calendar.getInstance();
calendar2.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date samebirthdate = calendar2.getTime();

//示例之前
System.out.printf("Is %1$tF before %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.before(birthdate)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", today, today,
Boolean.valueOf(today.before(today)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.before(today)));

//示例之后
System.out.printf("Is %1$tF after %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(birthdate)));

```

```

SimpleDateFormat objDf = new SimpleDateFormat(dateFormat);
try {
date = objDf.parse(formattedDate);
} catch (ParseException e) {
// Do what ever needs to be done with exception.
}
return date;
}

```

Section 16.8: Creating Date objects

```

Date date = new Date();
System.out.println(date); // Thu Feb 25 05:03:59 IST 2016

```

Here this Date object contains the current date and time when this object was created.

```

Calendar calendar = Calendar.getInstance();
calendar.set(90, Calendar.DECEMBER, 11);
Date myBirthDate = calendar.getTime();
System.out.println(myBirthDate); // Mon Dec 31 00:00:00 IST 1990

```

Date objects are best created through a Calendar instance since the use of the data constructors is deprecated and discouraged. To do so we need to get an instance of the Calendar class from the factory method. Then we can set year, month and day of month by using numbers or in case of months constants provided by the Calendar class to improve readability and reduce errors.

```

calendar.set(90, Calendar.DECEMBER, 11, 8, 32, 35);
Date myBirthDate = calendar.getTime();
System.out.println(myBirthDate); // Mon Dec 31 08:32:35 IST 1990

```

Along with date, we can also pass time in the order of hour, minutes and seconds.

Section 16.9: Comparing Date objects

Calendar, Date, and LocalDate

Version < Java SE 8

before, after, compareTo and equals methods

```

//Use of Calendar and Date objects
final Date today = new Date();
final Calendar calendar = Calendar.getInstance();
calendar.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date birthdate = calendar.getTime();

final Calendar calendar2 = Calendar.getInstance();
calendar2.set(1990, Calendar.NOVEMBER, 1, 0, 0, 0);
Date samebirthdate = calendar2.getTime();

//Before example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.before(birthdate)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", today, today,
Boolean.valueOf(today.before(today)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.before(today)));

//After example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(birthdate)));

```

```

Boolean.valueOf(today.after(birthdate)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(today)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.after(today)));

//比较示例
System.out.printf("比较 %1$tF 和 %2$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(birthdate)));
System.out.printf("比较 %1$tF 和 %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(today)));
System.out.printf("比较 %2$tF 和 %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(birthdate.compareTo(today)));

//相等示例
System.out.printf("%1$tF 是否等于 %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.equals(birthdate)));
System.out.printf("%1$tF 是否等于 %2$tF? %3$b%n", birthdate, samebirthdate,
Boolean.valueOf(birthdate.equals(samebirthdate)));
System.out.printf(
    "因为 birthdate.getTime() -> %1$d 与 samebirthdate.getTime() -> %2$d 不同,
存在毫秒差 !%n",
    Long.valueOf(birthdate.getTime()), Long.valueOf(samebirthdate.getTime()));

//清除日历中的毫秒
calendar.clear(Calendar.MILLISECOND);
calendar2.clear(Calendar.MILLISECOND);
birthdate = calendar.getTime();
samebirthdate = calendar2.getTime();

System.out.printf("清除毫秒后 %1$tF 是否等于 %2$tF? %3$b%n", birthdate, samebirthdate,
Boolean.valueOf(birthdate.equals(samebirthdate)));

```

版本 ≥ Java SE 8

isBefore、isAfter、compareTo 和 equals 方法

```

// 使用 LocalDate
final LocalDate now = LocalDate.now();
final LocalDate birthdate2 = LocalDate.of(2012, 6, 30);
final LocalDate birthdate3 = LocalDate.of(2012, 6, 30);

// 小时、分钟、秒和纳秒也可以通过另一个类 LocalDateTime 配置
// LocalDateTime.of(year, month, dayOfMonth, hour, minute, second, nanoOfSecond);

// isBefore 示例
System.out.printf("%1$tF 是否早于 %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(birthdate2)));
System.out.printf("%1$tF 是否早于 %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(now)));
System.out.printf("现在是 %2$tF 之前 %1$tF 吗? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isBefore(now)));

//isAfter 示例
System.out.printf("现在是否在 %2$tF 之后? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(birthdate2)));
System.out.printf("现在是否在 %1$tF 之后? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(now)));
System.out.printf("%2$tF 是否在 %1$tF 之后? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isAfter(now)));

//compareTo 示例
System.out.printf("比较 %1$tF 和 %2$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(birthdate2)));

```

```

Boolean.valueOf(today.after(birthdate)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.after(today)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", today, birthdate,
Boolean.valueOf(birthdate.after(today)));

//Compare example
System.out.printf("Compare %1$tF to %2$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(birthdate)));
System.out.printf("Compare %1$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(today.compareTo(today)));
System.out.printf("Compare %2$tF to %1$tF: %3$d%n", today, birthdate,
Integer.valueOf(birthdate.compareTo(today)));

//Equal example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", today, birthdate,
Boolean.valueOf(today.equals(birthdate)));
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", birthdate, samebirthdate,
Boolean.valueOf(birthdate.equals(samebirthdate)));
System.out.printf(
    "Because birthdate.getTime() -> %1$d is different from samebirthdate.getTime() -> %2$d,
there are millisecondes!%n",
    Long.valueOf(birthdate.getTime()), Long.valueOf(samebirthdate.getTime()));

//Clear ms from calendars
calendar.clear(Calendar.MILLISECOND);
calendar2.clear(Calendar.MILLISECOND);
birthdate = calendar.getTime();
samebirthdate = calendar2.getTime();

System.out.printf("Is %1$tF equal to %2$tF after clearing ms? %3$b%n", birthdate, samebirthdate,
Boolean.valueOf(birthdate.equals(samebirthdate)));

```

Version ≥ Java SE 8

isBefore, isAfter, compareTo and equals methods

```

//Use of LocalDate
final LocalDate now = LocalDate.now();
final LocalDate birthdate2 = LocalDate.of(2012, 6, 30);
final LocalDate birthdate3 = LocalDate.of(2012, 6, 30);

//Hours, minutes, second and nanoOfSecond can also be configured with an other class LocalDateTime
//LocalDateTime.of(year, month, dayOfMonth, hour, minute, second, nanoOfSecond);

//isBefore example
System.out.printf("Is %1$tF before %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(birthdate2)));
System.out.printf("Is %1$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isBefore(now)));
System.out.printf("Is %2$tF before %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isBefore(now)));

//isAfter example
System.out.printf("Is %1$tF after %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(birthdate2)));
System.out.printf("Is %1$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isAfter(now)));
System.out.printf("Is %2$tF after %1$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(birthdate2.isAfter(now)));

//compareTo example
System.out.printf("Compare %1$tF to %2$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(birthdate2)));

```

GoalKicker.com – Java® 专业人士笔记

78

GoalKicker.com – Java® Notes for Professionals

78

```

System.out.printf("比较 %1$tF 和 %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(now)));
System.out.printf("比较 %2$tF 和 %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(birthdate2.compareTo(now)));

//equals 示例
System.out.printf("现在的日期 %1$tF 是否等于 %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.equals(birthdate2)));
System.out.printf("%1$tF 是否等于 %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.equals(birthdate3)));

```

```

//isEqual 示例
System.out.printf("现在的日期 %1$tF 是否等于 %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isEqual(birthdate2)));
System.out.printf("%1$tF 是否等于 %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.isEqual(birthdate3)));

```

Java 8 之前的日期比较

在 Java 8 之前，可以使用[java.util.Calendar](#)和[java.util.Date](#)类来比较日期。Date 类提供了 4 个方法来比较日期：

- [after\(Date when\)](#)
- [before\(Date when\)](#)
- [compareTo\(Date anotherDate\)](#)
- [equals\(Object obj\)](#)

after、before、compareTo 和 equals 方法比较的是每个日期调用getTime()方法返回的值。

compareTo方法返回正整数。

- 值大于0：当日期晚于日期参数时
- 值大于0：当日期早于日期参数时
- 值等于0：当日期等于日期参数时

equals 结果可能会令人惊讶，如示例所示，因为像毫秒这样的值如果没有显式赋值，初始化时不会是相同的值。

自 Java 8 起

在 Java 8 中，提供了一个用于处理日期的新对象[java.time.LocalDate](#)。 LocalDate实现了[ChronoLocalDate](#)，这是日期的抽象表示，其中的历法或日历系统是可插拔的。

要获得日期时间的精度，必须使用对象[java.time.LocalDateTime](#)。 LocalDate和LocalDateTime 使用相同的方法名进行比较。

使用LocalDate比较日期与使用ChronoLocalDate不同，因为前者不考虑历法或日历系统。

由于大多数应用应使用LocalDate，示例中未包含ChronoLocalDate。更多内容请参见[这里](#)。

大多数应用应将方法签名、字段和变量声明为LocalDate，而非此[ChronoLocalDate]接口。

LocalDate有5个用于比较日期的方法：

```

System.out.printf("Compare %1$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(now.compareTo(now)));
System.out.printf("Compare %2$tF to %1$tF %3$d%n", now, birthdate2,
Integer.valueOf(birthdate2.compareTo(now)));

//equals example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.equals(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.equals(birthdate3)));

```

```

//isEqual example
System.out.printf("Is %1$tF equal to %2$tF? %3$b%n", now, birthdate2,
Boolean.valueOf(now.isEqual(birthdate2)));
System.out.printf("Is %1$tF to %2$tF? %3$b%n", birthdate2, birthdate3,
Boolean.valueOf(birthdate2.isEqual(birthdate3)));

```

Date comparison before Java 8

Before Java 8, dates could be compared using [java.util.Calendar](#) and [java.util.Date](#) classes. Date class offers 4 methods to compare dates :

- [after\(Date when\)](#)
- [before\(Date when\)](#)
- [compareTo\(Date anotherDate\)](#)
- [equals\(Object obj\)](#)

after, before, compareTo and equals methods compare the values returned by [getTime\(\)](#) method for each date.

compareTo method returns positive integer.

- Value greater than 0 : when the Date is after the Date argument
- Value greater than 0 : when the Date is before the Date argument
- Value equals to 0 : when the Date is equal to the Date argument

equals results can be surprising as shown in the example because values, like milliseconds, are not initialize with the same value if not explicitly given.

Since Java 8

With Java 8 a new Object to work with Date is available [java.time.LocalDate](#). LocalDate implements [ChronoLocalDate](#), the abstract representation of a date where the Chronology, or calendar system, is pluggable.

To have the date time precision the Object [java.time.LocalDateTime](#) has to be used. LocalDate and LocalDateTime use the same methods name for comparing.

Comparing dates using a LocalDate is different from using ChronoLocalDate because the chronology, or calendar system are not taken in account the first one.

Because most application should use LocalDate, ChronoLocalDate is not included in examples. Further reading [here](#).

Most applications should declare method signatures, fields and variables as LocalDate, not this[ChronoLocalDate] interface.

LocalDate has 5 methods to compare dates :

- [isAfter\(ChronoLocalDate other\)](#)
- [isBefore\(ChronoLocalDate other\)](#)
- [isEqual\(ChronoLocalDate other\)](#)
- [compareTo\(ChronoLocalDate other\)](#)
- [equals\(Object obj\)](#)

在LocalDate参数的情况下，isAfter、isBefore、isEqual、equals和compareTo现在使用此方法：

```
int compareTo0(LocalDate otherDate) {
    int cmp = (year - otherDate.year);
    if (cmp == 0) {
        cmp = (month - otherDate.month);
        if (cmp == 0) {
            cmp = (day - otherDate.day);
        }
    }
    return cmp;
}
```

equals 方法首先检查参数引用是否等于日期，而 isEqual 则直接调用 compareTo0。

如果是 ChronoLocalDate 的其他类实例，则使用 Epoch Day 比较日期。Epoch Day 计数是一个简单的递增天数计数，其中第0天是1970-01-01 (ISO) 。

第16.10节：将字符串转换为日期

来自 SimpleDateFormat 类的 parse() 方法有助于将 String 模式转换为 Date 对象。

```
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.SHORT, Locale.US);
String dateStr = "02/25/2016"; // 输入字符串
Date date = dateFormat.parse(dateStr);
System.out.println(date.getYear()); // 116
```

文本格式有4种不同的样式，SHORT、MEDIUM（默认）、LONG和FULL，所有样式均依赖于区域设置。如果未指定区域设置，则使用系统默认区域设置。

样式	Locale.US	Locale.France
SHORT	6/30/09	30/06/09
MEDIUM	2009年6月30日	2009年6月30日
LONG	2009年6月30日	2009年6月30日
完整	2009年6月30日星期二	2009年6月30日星期二

第16.11节：时区与java.util.Date

一个java.util.Date对象没有时区的概念。

- 无法为Date设置时区
- 无法更改Date对象的时区使用new Date()默认构造函数创建
- 建的Date对象将以系统默认时区的当前时间初始化

但是，可以使用例如java.text.SimpleDateFormat来显示Date对象所表示的时间点在不同时间区的日期：

```
Date date = new Date();
```

- [isAfter\(ChronoLocalDate other\)](#)
- [isBefore\(ChronoLocalDate other\)](#)
- [isEqual\(ChronoLocalDate other\)](#)
- [compareTo\(ChronoLocalDate other\)](#)
- [equals\(Object obj\)](#)

In case of LocalDate parameter, isAfter, isBefore, isEqual, equals and compareTo now use this method:

```
int compareTo0(LocalDate otherDate) {
    int cmp = (year - otherDate.year);
    if (cmp == 0) {
        cmp = (month - otherDate.month);
        if (cmp == 0) {
            cmp = (day - otherDate.day);
        }
    }
    return cmp;
}
```

equals method check if the parameter reference equals the date first whereas isEqual directly calls compareTo0.

In case of an other class instance of ChronoLocalDate the dates are compared using the Epoch Day. The Epoch Day count is a simple incrementing count of days where day 0 is 1970-01-01 (ISO).

Section 16.10: Converting String into Date

parse() from [SimpleDateFormat](#) class helps to convert a [String](#) pattern into a [Date](#) object.

```
DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.SHORT, Locale.US);
String dateStr = "02/25/2016"; // input String
Date date = dateFormat.parse(dateStr);
System.out.println(date.getYear()); // 116
```

There are 4 different styles for the text format, SHORT, MEDIUM (this is the default), LONG and FULL, all of which depend on the locale. If no locale is specified, the system default locale is used.

Style	Locale.US	Locale.France
SHORT	6/30/09	30/06/09
MEDIUM	Jun 30, 2009	30 juin 2009
LONG	June 30, 2009	30 juin 2009
FULL	Tuesday, June 30, 2009	mardi 30 juin 2009

Section 16.11: Time Zones and java.util.Date

A [java.util.Date](#) object does not have a concept of time zone.

- There is no way to **set** a timezone for a Date
- There is no way to **change** the timezone of a Date object
- A Date object created with the **new Date()** default constructor will be initialised with the current time in the system default timezone

However, it is possible to display the date represented by the point in time described by the Date object in a different time zone using e.g. [java.text.SimpleDateFormat](#):

```
Date date = new Date();
```

```
//打印默认时区
System.out.println(TimeZone.getDefault().getDisplayName());
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); //注意：格式中不包含时区！

//打印原时区的日期
System.out.println(sdf.format(date));
//伦敦当前时间
sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));
System.out.println(sdf.format(date));
```

输出：

```
中欧时间
2016-07-21 22:50:56
2016-07-21 21:50:56
```

```
//print default time zone
System.out.println(TimeZone.getDefault().getDisplayName());
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); //note: time zone not in
format!
//print date in the original time zone
System.out.println(sdf.format(date));
//current time in London
sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));
System.out.println(sdf.format(date));
```

Output:

```
Central European Time
2016-07-21 22:50:56
2016-07-21 21:50:56
```

第17章：日期和时间 (java.time.*)

第17.1节：计算两个LocalDate之间的差异

使用LocalDate和ChronoUnit：

```
LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);
```

现在，由于ChronoUnit枚举的between方法接受两个Temporal作为参数，因此你可以毫无问题地传递LocalDate实例

```
long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );
```

第17.2节：日期和时间

无时区信息的日期和时间

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

带时区信息的日期和时间

```
ZoneId zoneId = ZoneId.of("UTC+2");
ZonedDateTime dateTime = ZonedDateTime.of(2016, Month.JULY, 27, 7, 0, 0, 235, zoneId);
ZonedDateTime composition = ZonedDateTime.of(localDate, localTime, zoneId);
ZonedDateTime now = ZonedDateTime.now(); // 默认时区
ZonedDateTime parsed = ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]");
```

带偏移信息的日期和时间（即不考虑夏令时变化）

```
ZoneOffset zoneOffset = ZoneOffset.ofHours(2);
OffsetDateTime dateTime = OffsetDateTime.of(2016, 7, 27, 7, 0, 0, 235, zoneOffset);
OffsetDateTime composition = OffsetDateTime.of(localDate, localTime, zoneOffset);
OffsetDateTime now = OffsetDateTime.now(); // 偏移量取自默认的 ZoneId
OffsetDateTime parsed = OffsetDateTime.parse("2016-07-27T07:00:00+02:00");
```

第17.3节：日期和时间的操作

```
LocalDate 明天 = LocalDate.now().plusDays(1);
LocalDateTime 一小时后 = LocalDateTime.now().plusHours(1);
Long 天数间隔 = java.time.temporal.ChronoUnit.DAYS.between(LocalDate.now(),
LocalDate.now().plusDays(3)); // 3
Duration 持续时间 = Duration.between(Instant.now(),
ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]"))
```

第17.4节：Instant（瞬时点）

表示时间的一个瞬时点。可以被视为Unix时间戳的一个封装。

```
Instant 现在 = Instant.now();
Instant 纪元1 = Instant.ofEpochMilli(0);
```

Chapter 17: Dates and Time (java.time.*)

Section 17.1: Calculate Difference between 2 LocalDates

Use LocalDate and ChronoUnit:

```
LocalDate d1 = LocalDate.of(2017, 5, 1);
LocalDate d2 = LocalDate.of(2017, 5, 18);
```

now, since the method between of the ChronoUnit enumerator takes 2 Temporals as parameters so you can pass without a problem the LocalDate instances

```
long days = ChronoUnit.DAYS.between(d1, d2);
System.out.println( days );
```

Section 17.2: Date and time

Date and time without time zone information

```
LocalDateTime dateTime = LocalDateTime.of(2016, Month.JULY, 27, 8, 0);
LocalDateTime now = LocalDateTime.now();
LocalDateTime parsed = LocalDateTime.parse("2016-07-27T07:00:00");
```

Date and time with time zone information

```
ZoneId zoneId = ZoneId.of("UTC+2");
ZonedDateTime dateTime = ZonedDateTime.of(2016, Month.JULY, 27, 7, 0, 0, 235, zoneId);
ZonedDateTime composition = ZonedDateTime.of(localDate, localTime, zoneId);
ZonedDateTime now = ZonedDateTime.now(); // Default time zone
ZonedDateTime parsed = ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]");
```

Date and time with offset information (i.e. no DST changes taken into account)

```
ZoneOffset zoneOffset = ZoneOffset.ofHours(2);
OffsetDateTime dateTime = OffsetDateTime.of(2016, 7, 27, 7, 0, 0, 235, zoneOffset);
OffsetDateTime composition = OffsetDateTime.of(localDate, localTime, zoneOffset);
OffsetDateTime now = OffsetDateTime.now(); // Offset taken from the default ZoneId
OffsetDateTime parsed = OffsetDateTime.parse("2016-07-27T07:00:00+02:00");
```

Section 17.3: Operations on dates and times

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
LocalDateTime anHourFromNow = LocalDateTime.now().plusHours(1);
Long daysBetween = java.time.temporal.ChronoUnit.DAYS.between(LocalDate.now(),
LocalDate.now().plusDays(3)); // 3
Duration duration = Duration.between(Instant.now(),
ZonedDateTime.parse("2016-07-27T07:00:00+01:00[Europe/Stockholm]"))
```

Section 17.4: Instant

Represents an instant in time. Can be thought of as a wrapper around a Unix timestamp.

```
Instant now = Instant.now();
Instant epoch1 = Instant.ofEpochMilli(0);
```

```
Instant 纪元2 = Instant.parse("1970-01-01T00:00:00Z");
java.time.temporal.ChronoUnit.MICROS.between(纪元1, 纪元2); // 0
```

第17.5节：日期时间API中各类的使用

以下示例中也包含了理解示例所需的说明。

```
import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.TimeZone;
public class SomeMethodsExamples {

    /**
     * 该类 {@link LocalDateTime} 的方法

    public static void checkLocalDateTime() {
        LocalDateTime localDateTime = LocalDateTime.now();
        System.out.println("使用静态 now() 方法获取的本地日期时间 :::: >>> "
            +localDateTime);

        LocalDateTime ldt1 = LocalDateTime.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("AET")));
        System.out
            .println("使用 now(ZoneId zoneId) 方法获取的本地时间 :::: >>>"
            +ldt1);

        LocalDateTime ldt2 = LocalDateTime.now(Clock.system(ZoneId
            .of(ZoneId.SHORT_IDS.get("PST"))));
        System.out
            .println("使用 now(Clock.system(ZoneId.of())) 方法获取的本地时间 :::: >>>> "
            +ldt2);

        System.out
            .println("以下是ZoneId类中的一个静态映射，映射了简短的时区名称到它们的实际时区名称");
        System.out.println(ZoneId.SHORT_IDS);
    }

    /**
     * 这是类 {@link LocalDate} 的方法
     */
    public static void checkLocalDate() {
        LocalDate localDate = LocalDate.now();
        System.out.println("使用 now() 方法获取不带时间的日期。 >> "
            +localDate);
        LocalDate localDate2 = LocalDate.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("ECT")));
        System.out
            .println("now() 方法被重载以接受 ZoneID 作为参数，使用它我们可以获得
不同时区下的相同日期。 >> "
            +localDate2);
    }
}
```

```
Instant epoch2 = Instant.parse("1970-01-01T00:00:00Z");
java.time.temporal.ChronoUnit.MICROS.between(epoch1, epoch2); // 0
```

Section 17.5: Usage of various classes of Date Time API

Following example also have explanation required for understanding example within it.

```
import java.time.Clock;
import java.time.Duration;
import java.time.Instant;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.TimeZone;
public class SomeMethodsExamples {

    /**
     * Has the methods of the class {@link LocalDateTime}
     */
    public static void checkLocalDateTime() {
        LocalDateTime localDateTime = LocalDateTime.now();
        System.out.println("Local Date time using static now() method :::: >>> "
            + localDateTime);

        LocalDateTime ldt1 = LocalDateTime.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("AET")));
        System.out
            .println("LOCAL TIME USING now(ZoneId zoneId) method :::: >>>"
            + ldt1);

        LocalDateTime ldt2 = LocalDateTime.now(Clock.system(ZoneId
            .of(ZoneId.SHORT_IDS.get("PST"))));
        System.out
            .println("Local TIME USING now(Clock.system(ZoneId.of())) :::: >>>> "
            + ldt2);

        System.out
            .println("Following is a static map in ZoneId class which has mapping of short timezone
names to their Actual timezone names");
        System.out.println(ZoneId.SHORT_IDS);
    }

    /**
     * This has the methods of the class {@link LocalDate}
     */
    public static void checkLocalDate() {
        LocalDate localDate = LocalDate.now();
        System.out.println("Gives date without Time using now() method. >> "
            + localDate);
        LocalDate localDate2 = LocalDate.now(ZoneId.of(ZoneId.SHORT_IDS
            .get("ECT")));
        System.out
            .println("now() is overridden to take ZoneID as parameter using this we can get the
same date under different timezones. >> "
            + localDate2);
    }
}
```

* 该类包含抽象类{@link Clock}的方法。Clock可用于带有{@link TimeZone}的时间。

```
/*
public static void checkClock() {
    Clock clock = Clock.systemUTC();
    // 表示符合ISO 8601的时间
    System.out.println("使用Clock类的时间：" + clock.instant());
}

/**
* 该类包含{@link Instant}类的方法。
*/
public static void checkInstant() {
    Instant instant = Instant.now();

    System.out.println("使用now()方法的Instant :: " + instant);

    Instant ins1 = Instant.now(Clock.systemUTC());

    System.out.println("使用now(Clock clock)的Instant :: " + ins1);
}

/**
* 该类检查{@link Duration}类的方法。
*/
public static void checkDuration() {
    // toString() 根据 ISO 8601 标准将持续时间转换为 PTnHnMnS 格式。如果某个字段为零
    // ，则忽略该字段。

    // P 是持续时间标识符（历史上称为“period”），放置在持续时间表示的开头。
    // Y 是年份标识符，跟在年份数值之后。
    // M 是月份标识符，跟在月份数值之后。
    // W 是周数标识符，跟在周数数值之后。
    // D 是天数标识符，跟在天数数值之后。
    // T 是时间标识符，位于时间部分的前面。
    // H 是小时标识符，跟在小时数值之后。
    // M 是分钟标识符，跟在分钟数值之后。
    // S 是秒数标识符，跟在秒数数值之后。

    System.out.println(Duration.ofDays(2));
}

/**
* 显示不带日期的本地时间。它不存储或表示日期和时间，而是像墙上的时钟一样表示时间。
*/
public static void checkLocalTime() {
    LocalTime localTime = LocalTime.now();
    System.out.println("LocalTime :: " + localTime);
}
```

* This has the methods of abstract class {@link Clock}. Clock can be used
* for time which has time with {@link TimeZone}.

```
/*
public static void checkClock() {
    Clock clock = Clock.systemUTC();
    // Represents time according to ISO 8601
    System.out.println("Time using Clock class : " + clock.instant());
}

/**
* This has the {@link Instant} class methods.
*/
public static void checkInstant() {
    Instant instant = Instant.now();

    System.out.println("Instant using now() method :: " + instant);

    Instant ins1 = Instant.now(Clock.systemUTC());

    System.out.println("Instants using now(Clock clock) :: " + ins1);
}

/**
* This class checks the methods of the {@link Duration} class.
*/
public static void checkDuration() {
    // toString() converts the duration to PTnHnMnS format according to ISO
    // 8601 standard. If a field is zero its ignored.

    // P is the duration designator (historically called "period") placed at
    // the start of the duration representation.
    // Y is the year designator that follows the value for the number of
    // years.
    // M is the month designator that follows the value for the number of
    // months.
    // W is the week designator that follows the value for the number of
    // weeks.
    // D is the day designator that follows the value for the number of
    // days.
    // T is the time designator that precedes the time components of the
    // representation.
    // H is the hour designator that follows the value for the number of
    // hours.
    // M is the minute designator that follows the value for the number of
    // minutes.
    // S is the second designator that follows the value for the number of
    // seconds.

    System.out.println(Duration.ofDays(2));
}

/**
* Shows Local time without date. It doesn't store or represent a date and
* time. Instead its a representation of Time like clock on the wall.
*/
public static void checkLocalTime() {
    LocalTime localTime = LocalTime.now();
    System.out.println("LocalTime :: " + localTime);
}
```

* 一个带有时区信息的日期时间，符合 ISO-8601 标准。

```
/*
public static void checkZonedDateTime() {
    ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId
        .of(ZoneId.SHORT_IDS.get("CST")));
    System.out.println(zonedDateTime);
}
```

第 17 章：日期时间格式化

text 中有 DateFormat 和 SimpleDateFormat 类，这些遗留代码将继续使用一段时间。

但是，Java 8 提供了处理格式化和解析的现代方法。

在格式化和解析中，首先将一个 String 对象传递给 DateTimeFormatter，然后使用它进行格式化或解析。

```
import java.time.*;
import java.time.format.*;

class DateTimeFormat
{
    public static void main(String[] args) {

        //解析
        String pattern = "d-MM-yyyy HH:mm";
        DateTimeFormatter dtF1 = DateTimeFormatter.ofPattern(pattern);

        LocalDateTime ldp1 = LocalDateTime.parse("2014-03-25T01:30"), //默认格式
        ldp2 = LocalDateTime.parse("15-05-2016 13:55",dtF1); //自定义格式

        System.out.println(ldp1 + "" + ldp2); //将以默认格式打印//格式化

        DateTimeFormatter dtF2 = DateTimeFormatter.ofPattern("EEE d, MMMM, yyyy HH:mm");

        DateTimeFormatter dtF3 = DateTimeFormatter.ISO_LOCAL_DATE_TIME;

        LocalDateTime ldtf1 = LocalDateTime.now();

        System.out.println(ldtf1.format(dtF2) +""+ldtf1.format(dtF3));
    }
}
```

一个重要提示，建议不要使用自定义模式，而是使用预定义的格式化器。你的代码看起来更清晰，长期来看使用ISO8601绝对会对你有帮助。

第17.7节：简单日期操作

获取当前日期。

```
LocalDate.now()
```

获取昨天的日期。

* A date time with Time zone details in ISO-8601 standards.

```
/*
public static void checkZonedDateTime() {
    ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId
        .of(ZoneId.SHORT_IDS.get("CST")));
    System.out.println(zonedDateTime);
}
}
```

Section 17.6: Date Time Formatting

Before Java 8, there was [DateFormat](#) and [SimpleDateFormat](#) classes in the package `java.text` and this legacy code will be continued to be used for sometime.

But, Java 8 offers a modern approach to handling Formatting and Parsing.

In formatting and parsing first you pass a [String](#) object to [DateTimeFormatter](#), and in turn use it for formatting or parsing.

```
import java.time.*;
import java.time.format.*;

class DateTimeFormat
{
    public static void main(String[] args) {

        //Parsing
        String pattern = "d-MM-yyyy HH:mm";
        DateTimeFormatter dtF1 = DateTimeFormatter.ofPattern(pattern);

        LocalDateTime ldp1 = LocalDateTime.parse("2014-03-25T01:30"), //Default format
        ldp2 = LocalDateTime.parse("15-05-2016 13:55",dtF1); //Custom format

        System.out.println(ldp1 + "\n" + ldp2); //Will be printed in Default format

        //Formatting
        DateTimeFormatter dtF2 = DateTimeFormatter.ofPattern("EEE d, MMMM, yyyy HH:mm");

        DateTimeFormatter dtF3 = DateTimeFormatter.ISO_LOCAL_DATE_TIME;

        LocalDateTime ldtf1 = LocalDateTime.now();

        System.out.println(ldtf1.format(dtF2) +"\n"+ldtf1.format(dtF3));
    }
}
```

An important notice, instead of using Custom patterns, it is good practice to use predefined formatters. Your code look more clear and usage of ISO8601 will definitely help you in the long run.

Section 17.7: Simple Date Manipulations

Get the current date.

```
LocalDate.now()
```

Get yesterday's date.

```
LocalDate y = LocalDate.now().minusDays(1);
```

获取明天的日期

```
LocalDate t = LocalDate.now().plusDays(1);
```

获取特定日期。

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

除了plus和minus方法之外，还有一组“with”方法可以用来设置LocalDate实例的特定字段。

```
LocalDate.now().withMonth(6);
```

上面的例子返回一个新的实例，月份设置为六月（这与java.util.Date不同，后者中setMonth的索引从0开始，因此六月是5）。

由于LocalDate的操作返回的是不可变的LocalDate实例，这些方法也可以链式调用。

```
LocalDate ld = LocalDate.now().plusDays(1).plusYears(1);
```

这将给出从现在起一年后的明天的日期。

```
LocalDate y = LocalDate.now() .minusDays(1);
```

Get tomorrow's date

```
LocalDate t = LocalDate.now() .plusDays(1);
```

Get a specific date.

```
LocalDate t = LocalDate.of(1974, 6, 2, 8, 30, 0, 0);
```

In addition to the plus and minus methods, there are a set of "with" methods that can be used to set a particular field on a LocalDate instance.

```
LocalDate.now() .withMonth(6);
```

The example above returns a new instance with the month set to June (this differs from java.util.Date where setMonth was indexed a 0 making June 5).

Because LocalDate manipulations return immutable LocalDate instances, these methods may also be chained together.

```
LocalDate ld = LocalDate.now() .plusDays(1) .plusYears(1);
```

This would give us tomorrow's date one year from now.

第18章：LocalTime

方法	输出
LocalTime.of(13, 12, 11)	13:12:11
LocalTime.MIDNIGHT	00:00
LocalTime.中午	12:00
LocalTime.现在()	来自系统时钟的当前时间
LocalTime.最大值	支持的最大本地时间 23:59:59.999999999
LocalTime.最小值	支持的最小本地时间 00:00
LocalTime.根据当天秒数(84399)获取时间	23:59:59, 从当天秒数值获取时间
LocalTime.ofNanoOfDay(0000000000)	00:00:02, 从当天纳秒值获取时间

第18.1节：两个LocalTime之间的时间量

计算两个LocalTime之间时间单位的量有两种等效的方法：(1) 通过 until(Temporal, TemporalUnit)方法，和 (2) 通过TemporalUnit.between(Temporal, Temporal)方法。

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;

public class AmountOfTime {

    public static void main(String[] args) {

        LocalTime start = LocalTime.of(1, 0, 0); // 小时, 分钟, 秒
        LocalTime end = LocalTime.of(2, 10, 20); // 小时, 分钟, 秒

        long halfDays1 = start.until(end, ChronoUnit.HALF_DAYS); // 0
        long halfDays2 = ChronoUnit.HALF_DAYS.between(start, end); // 0

        long hours1 = start.until(end, ChronoUnit.HOURS); // 1
        long hours2 = ChronoUnit.HOURS.between(start, end); // 1

        long minutes1 = start.until(end, ChronoUnit.MINUTES); // 70
        long minutes2 = ChronoUnit.MINUTES.between(start, end); // 70

        long seconds1 = start.until(end, ChronoUnit.SECONDS); // 4220
        long seconds2 = ChronoUnit.SECONDS.between(start, end); // 4220

        long millis1 = start.until(end, ChronoUnit.MILLIS); // 4220000
        long millis2 = ChronoUnit.MILLIS.between(start, end); // 4220000

        long microsecs1 = start.until(end, ChronoUnit.MICROS); // 4220000000
        long microsecs2 = ChronoUnit.MICROS.between(start, end); // 4220000000

        long nanosecs1 = start.until(end, ChronoUnit.NANOS); // 422000000000000
        long nanosecs2 = ChronoUnit.NANOS.between(start, end); // 422000000000000

        // 使用其他ChronoUnit会抛出UnsupportedTemporalTypeException异常。
        // 以下方法是其示例。
        long days1 = start.until(end, ChronoUnit.DAYS);
        long days2 = ChronoUnit.DAYS.between(start, end);
    }
}
```

Chapter 18: LocalTime

Method	Output
LocalTime.of(13, 12, 11)	13:12:11
LocalTime.MIDNIGHT	00:00
LocalTime.NOON	12:00
LocalTime.now()	Current time from system clock
LocalTime.MAX	The maximum supported local time 23:59:59.999999999
LocalTime.MIN	The minimum supported local time 00:00
LocalTime.ofSecondOfDay(84399)	23:59:59, Obtains Time from second-of-day value
LocalTime.ofNanoOfDay(2000000000)	00:00:02, Obtains Time from nanos-of-day value

Section 18.1: Amount of time between two LocalTime

There are two equivalent ways to calculate the amount of time unit between two LocalTime: (1) through until(Temporal, TemporalUnit) method and through (2) TemporalUnit.between(Temporal, Temporal).

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;

public class AmountOfTime {

    public static void main(String[] args) {

        LocalTime start = LocalTime.of(1, 0, 0); // hour, minute, second
        LocalTime end = LocalTime.of(2, 10, 20); // hour, minute, second

        long halfDays1 = start.until(end, ChronoUnit.HALF_DAYS); // 0
        long halfDays2 = ChronoUnit.HALF_DAYS.between(start, end); // 0

        long hours1 = start.until(end, ChronoUnit.HOURS); // 1
        long hours2 = ChronoUnit.HOURS.between(start, end); // 1

        long minutes1 = start.until(end, ChronoUnit.MINUTES); // 70
        long minutes2 = ChronoUnit.MINUTES.between(start, end); // 70

        long seconds1 = start.until(end, ChronoUnit.SECONDS); // 4220
        long seconds2 = ChronoUnit.SECONDS.between(start, end); // 4220

        long millis1 = start.until(end, ChronoUnit.MILLIS); // 4220000
        long millis2 = ChronoUnit.MILLIS.between(start, end); // 4220000

        long microsecs1 = start.until(end, ChronoUnit.MICROS); // 4220000000
        long microsecs2 = ChronoUnit.MICROS.between(start, end); // 4220000000

        long nanosecs1 = start.until(end, ChronoUnit.NANOS); // 422000000000000
        long nanosecs2 = ChronoUnit.NANOS.between(start, end); // 422000000000000

        // Using others ChronoUnit will be thrown UnsupportedTemporalTypeException.
        // The following methods are examples thereof.
        long days1 = start.until(end, ChronoUnit.DAYS);
        long days2 = ChronoUnit.DAYS.between(start, end);
    }
}
```

第18.2节：简介

LocalTime 是一个不可变且线程安全的类，用于表示时间，通常视为时-分-秒。时间以纳秒精度表示。例如，值“13:45.30.123456789”可以存储在 LocalTime 中。

该类不存储或表示日期或时区。相反，它描述的是墙上时钟显示的本地时间。没有额外的信息（如偏移量或时区），它无法表示时间线上的一个瞬间。它是一个基于值的类，比较时应使用 equals 方法。

字段

MAX - 支持的最大 LocalTime, '23:59:59.999999999'。MIDNIGHT (午夜)、MIN (最小值)、NOON (中午)

重要的静态方法

now(), now(Clock clock), now(ZonedDateTime zone), parse(CharSequence text)

重要的实例方法

isAfter(LocalTime other), isBefore(LocalTime other), minus(TemporalAmount amountToSubtract), minus(long amountToSubtract, TemporalUnit unit), plus(TemporalAmount amountToAdd), plus(long amountToAdd, TemporalUnit unit)

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");
```

时间差可以通过以下任意方式计算

```
long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);
```

你也可以从任何 LocalTime 对象中加上或减去小时、分钟或秒。

minusHours(long hoursToSubtract), minusMinutes(long hoursToMinutes), minusNanos(long nanosToSubtract),
minusSeconds(long secondsToSubtract), plusHours(long hoursToSubtract), plusMinutes(long hoursToMinutes),
plusNanos(long nanosToSubtract), plusSeconds(long secondsToSubtract)

```
now.plusHours(1L);
now1.minusMinutes(20L);
```

第18.3节：时间修改

你可以加上小时、分钟、秒和纳秒：

```
LocalTime time = LocalTime.now();
LocalTime addHours = time.plusHours(5); // 加5小时
LocalTime addMinutes = time.plusMinutes(15); // 加15分钟
LocalTime addSeconds = time.plusSeconds(30); // 加30秒
LocalTime addNanoseconds = time.plusNanos(150_000_000); // 加150,000,000纳秒 (150毫秒)
```

第18.4节：时区及其时间差

```
import java.time.LocalTime;
```

Section 18.2: Intro

LocalTime is an immutable class and thread-safe, used to represent time, often viewed as hour-min-sec. Time is represented to nanosecond precision. For example, the value "13:45.30.123456789" can be stored in a LocalTime.

This class does not store or represent a date or time-zone. Instead, it is a description of the local time as seen on a wall clock. It cannot represent an instant on the time-line without additional information such as an offset or time-zone. This is a value based class, equals method should be used for comparisons.

Fields

MAX - The maximum supported LocalTime, '23:59:59.999999999'. MIDNIGHT, MIN, NOON

Important Static Methods

now(), now(Clock clock), now(ZonedDateTime zone), parse(CharSequence text)

Important Instance Methods

isAfter(LocalTime other), isBefore(LocalTime other), minus(TemporalAmount amountToSubtract), minus(long amountToSubtract, TemporalUnit unit), plus(TemporalAmount amountToAdd), plus(long amountToAdd, TemporalUnit unit)

```
ZoneId zone = ZoneId.of("Asia/Kolkata");
LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone);
LocalTime then = LocalTime.parse("04:16:40");
```

Difference in time can be calculated in any of following ways

```
long timeDiff = Duration.between(now, now1).toMinutes();
long timeDiff1 = java.time.temporal.ChronoUnit.MINUTES.between(now2, now1);
```

You can also add/subtract hours, minutes or seconds from any object of LocalTime.

minusHours(long hoursToSubtract), minusMinutes(long hoursToMinutes), minusNanos(long nanosToSubtract),
minusSeconds(long secondsToSubtract), plusHours(long hoursToSubtract), plusMinutes(long hoursToMinutes),
plusNanos(long nanosToSubtract), plusSeconds(long secondsToSubtract)

```
now.plusHours(1L);
now1.minusMinutes(20L);
```

Section 18.3: Time Modification

You can add hours, minutes, seconds and nanoseconds:

```
LocalTime time = LocalTime.now();
LocalTime addHours = time.plusHours(5); // Add 5 hours
LocalTime addMinutes = time.plusMinutes(15); // Add 15 minutes
LocalTime addSeconds = time.plusSeconds(30); // Add 30 seconds
LocalTime addNanoseconds = time.plusNanos(150_000_000); // Add 150,000,000ns (150ms)
```

Section 18.4: Time Zones and their time difference

```
import java.time.LocalTime;
```

```
import java.time.ZoneId;
import java.time.temporal.ChronoUnit;

public class Test {
    public static void main(String[] args)
    {
ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");

    LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone1);
    LocalTime now2 = LocalTime.now(zone2);

    System.out.println("当前时间 : " + now);
    System.out.println("柏林时间 : " + now1);
    System.out.println("巴西时间 : " + now2);

long minutesBetween = ChronoUnit.MINUTES.between(now2, now1);
System.out.println("柏林和巴西之间的分钟数 : " + minutesBetween +"分钟");
    }
}
```

```
import java.time.ZoneId;
import java.time.temporal.ChronoUnit;

public class Test {
    public static void main(String[] args)
    {
ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");

    LocalTime now = LocalTime.now();
LocalTime now1 = LocalTime.now(zone1);
    LocalTime now2 = LocalTime.now(zone2);

    System.out.println("Current Time : " + now);
    System.out.println("Berlin Time : " + now1);
    System.out.println("Brazil Time : " + now2);

long minutesBetween = ChronoUnit.MINUTES.between(now2, now1);
System.out.println("Minutes Between Berlin and Brazil : " + minutesBetween +"mins");
    }
}
```

第19章：BigDecimal

BigDecimal 类提供算术运算（加、减、乘、除）、刻度操作、舍入、比较、哈希和格式转换等操作。BigDecimal 表示不可变的任意精度有符号十进制数。该类应在需要高精度计算时使用。

第19.1节：比较 BigDecimal

应使用 compareTo 方法来比较 BigDecimal：

```
BigDecimal a = new BigDecimal(5);
a.compareTo(new BigDecimal(0)); // a 大于, 返回 1
a.compareTo(new BigDecimal(5)); // a 相等, 返回 0
a.compareTo(new BigDecimal(10)); // a 较小, 返回 -1
```

通常你不应该使用 equals 方法，因为它只有在两个 BigDecimal 的值和 scale 都相等时才认为它们相等：

```
BigDecimal a = new BigDecimal(5);
a.equals(new BigDecimal(5)); // 值和 scale 都相等, 返回 true
a.equals(new BigDecimal(5.00)); // 值相等但 scale 不同, 返回 false
```

第19.2节：使用 BigDecimal 替代 float

由于 float 类型在计算机内存中的表示方式，使用该类型进行的运算结果可能不准确——某些值以近似值存储。货币计算就是很好的例子。如果需要高精度，应使用其他类型。例如，Java 7 提供了 BigDecimal。

```
import java.math.BigDecimal;

public class FloatTest {

    public static void main(String[] args) {
        float accountBalance = 10000.00f;
        System.out.println("使用 float 的运算:");
        System.out.println("1000 次 1.99 的运算");
        for(int i = 0; i<1000; i++){
            accountBalance -= 1.99f;
        }
        System.out.println(String.format("浮点运算后的账户余额: %f",
            accountBalance));

        BigDecimal accountBalanceTwo = new BigDecimal("10000.00");
        System.out.println("使用 BigDecimal 的运算:");
        System.out.println("1000 次 1.99 的运算");
        BigDecimal operation = new BigDecimal("1.99");
        for(int i = 0; i<1000; i++){
            accountBalanceTwo = accountBalanceTwo.subtract(operation);
        }
        System.out.println(String.format("使用 BigDecimal 运算后的账户余额: %f",
            accountBalanceTwo));
    }
}
```

程序输出为：

Chapter 19: BigDecimal

The [BigDecimal](#) class provides operations for arithmetic (add, subtract, multiply, divide), scale manipulation, rounding, comparison, hashing, and format conversion. The BigDecimal represents immutable, arbitrary-precision signed decimal numbers. This class shall be used in a necessity of high-precision calculation.

Section 19.1: Comparing BigDecimals

The method [compareTo](#) should be used to compare BigDecimals:

```
BigDecimal a = new BigDecimal(5);
a.compareTo(new BigDecimal(0)); // a is greater, returns 1
a.compareTo(new BigDecimal(5)); // a is equal, returns 0
a.compareTo(new BigDecimal(10)); // a is less, returns -1
```

Commonly you should **not** use the [equals](#) method since it considers two BigDecimals equal only if they are equal in value and also **scale**:

```
BigDecimal a = new BigDecimal(5);
a.equals(new BigDecimal(5)); // value and scale are equal, returns true
a.equals(new BigDecimal(5.00)); // value is equal but scale is not, returns false
```

Section 19.2: Using BigDecimal instead of float

Due to way that the float type is represented in computer memory, results of operations using this type can be inaccurate - some values are stored as approximations. Good examples of this are monetary calculations. If high precision is necessary, other types should be used. e.g. Java 7 provides BigDecimal.

```
import java.math.BigDecimal;

public class FloatTest {

    public static void main(String[] args) {
        float accountBalance = 10000.00f;
        System.out.println("Operations using float:");
        System.out.println("1000 operations for 1.99");
        for(int i = 0; i<1000; i++){
            accountBalance -= 1.99f;
        }
        System.out.println(String.format("Account balance after float operations: %f",
            accountBalance));

        BigDecimal accountBalanceTwo = new BigDecimal("10000.00");
        System.out.println("Operations using BigDecimal:");
        System.out.println("1000 operations for 1.99");
        BigDecimal operation = new BigDecimal("1.99");
        for(int i = 0; i<1000; i++){
            accountBalanceTwo = accountBalanceTwo.subtract(operation);
        }
        System.out.println(String.format("Account balance after BigDecimal operations: %f",
            accountBalanceTwo));
    }
}
```

Output of this program is:

```
使用浮点数的运算:  
1000 次 1.99 的运算  
浮点运算后的账户余额: 8009,765625  
使用 BigDecimal 的运算:  
1000 次 1.99 的运算  
使用 BigDecimal 运算后的账户余额: 8010,000000
```

对于起始余额为10000.00，经过1000次每次1.99的操作后，我们预期余额为8010.00。使用浮点数类型得到的结果约为8009.77，在货币计算的情况下这是不可接受的误差。

使用 BigDecimal 可以得到正确的结果。

第19.3节：BigDecimal.valueOf()

BigDecimal 类包含一个常用数字的内部缓存，例如 0 到 10。建议优先使用 BigDecimal.valueOf() 方法，而不是使用类似类型参数的构造函数，即在下面的例子中，a 优于 b。

```
BigDecimal a = BigDecimal.valueOf(10L); //返回缓存的对象引用  
BigDecimal b = new BigDecimal(10L); //不返回缓存的对象引用
```

```
BigDecimal a = BigDecimal.valueOf(20L); //不返回缓存的对象引用  
BigDecimal b = new BigDecimal(20L); //不返回缓存的对象引用
```

BigDecimal a = BigDecimal.valueOf(15.15); //将 double (或 float) 转换为 BigDecimal 的首选方式，因为返回的值等同于使用 Double.toString(double) 结果构造 BigDecimal 得到的值

```
BigDecimal b = new BigDecimal(15.15); //返回不可预测的结果
```

```
Operations using float:  
1000 operations for 1.99  
Account balance after float operations: 8009,765625  
Operations using BigDecimal:  
1000 operations for 1.99  
Account balance after BigDecimal operations: 8010,000000
```

For a starting balance of 10000.00, after 1000 operations for 1.99, we expect the balance to be 8010.00. Using the float type gives us an answer around 8009.77, which is unacceptably imprecise in the case of monetary calculations. Using BigDecimal gives us the proper result.

Section 19.3: BigDecimal.valueOf()

The BigDecimal class contains an internal cache of frequently used numbers e.g. 0 to 10. The BigDecimal.valueOf() methods are provided in preference to constructors with similar type parameters i.e. in the below example a is preferred to b.

```
BigDecimal a = BigDecimal.valueOf(10L); //Returns cached Object reference  
BigDecimal b = new BigDecimal(10L); //Does not return cached Object reference
```

```
BigDecimal a = BigDecimal.valueOf(20L); //Does not return cached Object reference  
BigDecimal b = new BigDecimal(20L); //Does not return cached Object reference
```

BigDecimal a = BigDecimal.valueOf(15.15); //Preferred way to convert a double (or float) into a BigDecimal, as the value returned is equal to that resulting from constructing a BigDecimal from the result of using Double.toString(double)

```
BigDecimal b = new BigDecimal(15.15); //Return unpredictable result
```

第19.4节：使用 BigDecimal 进行数学运算

此示例展示了如何使用 BigDecimal 执行基本的数学运算。

1. 加法

```
BigDecimal a = new BigDecimal("5");  
BigDecimal b = new BigDecimal("7");  
  
//等同于 result = a + b  
BigDecimal result = a.add(b);  
System.out.println(result);
```

结果 : 12

2. 减法

```
BigDecimal a = new BigDecimal("5");  
BigDecimal b = new BigDecimal("7");  
  
//等同于 result = a - b  
BigDecimal result = a.subtract(b);  
System.out.println(result);
```

结果 : -2

Section 19.4: Mathematical operations with BigDecimal

This example shows how to perform basic mathematical operations using BigDecimals.

1.Addition

```
BigDecimal a = new BigDecimal("5");  
BigDecimal b = new BigDecimal("7");  
  
//Equivalent to result = a + b  
BigDecimal result = a.add(b);  
System.out.println(result);
```

Result : 12

2.Subtraction

```
BigDecimal a = new BigDecimal("5");  
BigDecimal b = new BigDecimal("7");  
  
//Equivalent to result = a - b  
BigDecimal result = a.subtract(b);  
System.out.println(result);
```

Result : -2

3. 乘法

当两个BigDecimal相乘时，结果的精度(scale)等于操作数精度之和。

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

//等同于 result = a * b
BigDecimal result = a.multiply(b);
System.out.println(result);
```

结果 : 36.89931

要更改结果的小数位数，可以使用重载的 multiply 方法，该方法允许传入MathContext——一个描述运算规则的对象，特别是结果的精度和舍入模式。有关可用舍入模式的更多信息，请参阅 Oracle 文档。

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

MathContext returnRules = new MathContext(4, RoundingMode.HALF_DOWN);

//等同于 result = a * b
BigDecimal result = a.multiply(b, returnRules);
System.out.println(result);
```

结果 : 36.90

4. 除法

除法比其他算术运算稍微复杂一些，例如考虑以下示例：

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

BigDecimal result = a.divide(b);
System.out.println(result);
```

我们预计这会得到类似于 : 0.7142857142857143 的结果，但实际上会得到：

结果 : java.lang.ArithmeticException: 非终止小数扩展；没有精确可表示的小数结果。

当结果是终止小数时，比如我想用5除以2，这样的操作会完美运行，但对于那些除法结果是非终止小数的数字，我们会得到一个ArithmaticException异常。

在实际场景中，无法预测除法过程中会遇到的数值，因此我们需要为BigDecimal除法指定Scale和Rounding Mode。有关 Scale 和 Rounding Mode 的更多信息，请参阅 Oracle 文档。

例如，我可以这样写：

```
BigDecimal a = new BigDecimal("5");
```

3. Multiplication

When multiplying two `BigDecimals` the result is going to have scale equal to the sum of the scales of operands.

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

//Equivalent to result = a * b
BigDecimal result = a.multiply(b);
System.out.println(result);
```

Result : 36.89931

To change the scale of the result use the overloaded multiply method which allows passing MathContext - an object describing the rules for operators, in particular the precision and rounding mode of the result. For more information about available rounding modes please refer to the Oracle Documentation.

```
BigDecimal a = new BigDecimal("5.11");
BigDecimal b = new BigDecimal("7.221");

MathContext returnRules = new MathContext(4, RoundingMode.HALF_DOWN);

//Equivalent to result = a * b
BigDecimal result = a.multiply(b, returnRules);
System.out.println(result);
```

Result : 36.90

4. Division

Division is a bit more complicated than the other arithmetic operations, for instance consider the below example:

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");

BigDecimal result = a.divide(b);
System.out.println(result);
```

We would expect this to give something similar to : 0.7142857142857143, but we would get:

Result: java.lang.ArithmaticException: Non-terminating decimal expansion; no exact representable decimal result.

This would work perfectly well when the result would be a terminating decimal say if I wanted to divide 5 by 2, but for those numbers which upon dividing would give a non terminating result we would get an `ArithmaticException`. In the real world scenario, one cannot predict the values that would be encountered during the division, so we need to specify the **Scale** and the **Rounding Mode** for BigDecimal division. For more information on the Scale and Rounding Mode, refer the [Oracle Documentation](#).

For example, I could do:

```
BigDecimal a = new BigDecimal("5");
```

```
BigDecimal b = new BigDecimal("7");
```

```
//相当于 result = a / b (最多10位小数, 采用四舍五入)
BigDecimal result = a.divide(b,10,RoundingMode.HALF_UP);
System.out.println(result);
```

结果 : 0.7142857143

5. 余数或模数

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");
```

```
// 等同于 result = a % b
BigDecimal result = a.remainder(b);
System.out.println(result);
```

结果 : 5

6. 幂

```
BigDecimal a = new BigDecimal("5");
```

```
// 等同于 result = a^10
BigDecimal result = a.pow(10);
System.out.println(result);
```

结果 : 9765625

7. 最大值

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");
```

```
//等同于 result = MAX(a,b)
BigDecimal result = a.max(b);
System.out.println(result);
```

结果 : 7

8. 取最小值

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");
```

```
//等同于 result = MIN(a,b)
BigDecimal result = a.min(b);
System.out.println(result);
```

结果 : 5

9. 小数点左移

```
BigDecimal b = new BigDecimal("7");
```

```
//Equivalent to result = a / b (Upto 10 Decimal places and Round HALF_UP)
BigDecimal result = a.divide(b,10,RoundingMode.HALF_UP);
System.out.println(result);
```

Result : 0.7142857143

5.Remainder or Modulus

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");
```

```
//Equivalent to result = a % b
BigDecimal result = a.remainder(b);
System.out.println(result);
```

Result : 5

6.Power

```
BigDecimal a = new BigDecimal("5");
```

```
//Equivalent to result = a^10
BigDecimal result = a.pow(10);
System.out.println(result);
```

Result : 9765625

7.Max

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");
```

```
//Equivalent to result = MAX(a,b)
BigDecimal result = a.max(b);
System.out.println(result);
```

Result : 7

8.Min

```
BigDecimal a = new BigDecimal("5");
BigDecimal b = new BigDecimal("7");
```

```
//Equivalent to result = MIN(a,b)
BigDecimal result = a.min(b);
System.out.println(result);
```

Result : 5

9.Move Point To Left

```
BigDecimal a = new BigDecimal("5234.49843776");
//将小数点向左移动2位
BigDecimal result = a.movePointLeft(2);
System.out.println(result);
```

结果：52.3449843776

10. 将小数点向右移动

```
BigDecimal a = new BigDecimal("5234.49843776");
// 将小数点向当前位置的右侧移动3位
BigDecimal result = a.movePointRight(3);
System.out.println(result);
```

结果：5234498.43776

上述示例还有更多选项和参数组合（例如，divide方法有6种变体），此集合并非详尽无遗，仅涵盖一些基本示例。

第19.5节：使用值为零、一或十初始化BigDecimal

BigDecimal提供了表示数字零、一和十的静态属性。建议使用这些属性，而不是直接使用数字：

- [BigDecimal.ZERO](#)
- [BigDecimal.ONE](#)
- [BigDecimal.TEN](#)

通过使用静态属性，可以避免不必要的实例化，同时代码中使用了字面量，而不是“魔法数字”。

```
//不好的示例：
BigDecimal bad0 = new BigDecimal(0);
BigDecimal bad1 = new BigDecimal(1);
BigDecimal bad10 = new BigDecimal(10);
```

```
//好的示例：
BigDecimal good0 = BigDecimal.ZERO;
BigDecimal good1 = BigDecimal.ONE;
BigDecimal good10 = BigDecimal.TEN;
```

第19.6节：BigDecimal对象是不可变的

如果你想用BigDecimal进行计算，必须使用返回的值，因为BigDecimal对象是不可变的：

```
BigDecimal a = new BigDecimal("42.23");
BigDecimal b = new BigDecimal("10.001");

a.add(b); // a仍然是42.23
```

```
BigDecimal a = new BigDecimal("5234.49843776");
```

```
//Moves the decimal point to 2 places left of current position
BigDecimal result = a.movePointLeft(2);
System.out.println(result);
```

Result : 52.3449843776

10.Move Point To Right

```
BigDecimal a = new BigDecimal("5234.49843776");
//Moves the decimal point to 3 places right of current position
BigDecimal result = a.movePointRight(3);
System.out.println(result);
```

Result : 5234498.43776

There are many more options and combination of parameters for the above mentioned examples (For instance, there are 6 variations of the divide method), this set is a non-exhaustive list and covers a few basic examples.

Section 19.5: Initialization of BigDecimals with value zero, one or ten

BigDecimal provides static properties for the numbers zero, one and ten. It's good practise to use these instead of using the actual numbers:

- [BigDecimal.ZERO](#)
- [BigDecimal.ONE](#)
- [BigDecimal.TEN](#)

By using the static properties, you avoid an unnecessary instantiation, also you've got a literal in your code instead of a 'magic number'.

```
//Bad example:
BigDecimal bad0 = new BigDecimal(0);
BigDecimal bad1 = new BigDecimal(1);
BigDecimal bad10 = new BigDecimal(10);
```

```
//Good Example:
BigDecimal good0 = BigDecimal.ZERO;
BigDecimal good1 = BigDecimal.ONE;
BigDecimal good10 = BigDecimal.TEN;
```

Section 19.6: BigDecimal objects are immutable

If you want to calculate with BigDecimal you have to use the returned value because BigDecimal objects are immutable:

```
BigDecimal a = new BigDecimal("42.23");
BigDecimal b = new BigDecimal("10.001");

a.add(b); // a will still be 42.23
```

```
BigDecimal c = a.add(b); // c 将是 52.231
```

```
BigDecimal c = a.add(b); // c will be 52.231
```

第20章：大整数（BigInteger）

`BigInteger` 类用于涉及数值过大，超出原始数据类型表示范围的大整数的数学运算。例如，100的阶乘有158位数字——远远大于 `long` 类型能表示的范围。`BigInteger` 提供了与Java所有原始整数运算符对应的功能，以及 `java.lang.Math` 中所有相关方法和其他一些操作的类似功能。

第20.1节：初始化

`java.math.BigInteger` 类提供了与Java所有原始整数运算符对应的操作，以及 `java.lang.Math` 中所有相关方法的类似功能。由于 `java.math` 包不会自动导入，您可能需要先导入 `java.math.BigInteger` 才能使用该类的简单名称。

将 `long` 或 `int` 值转换为 `BigInteger` 使用：

```
long longValue = Long.MAX_VALUE;
BigInteger valueFromLong = BigInteger.valueOf(longValue);
```

或者，对于整数：

```
int intValue = Integer.MIN_VALUE; // 负数
BigInteger valueFromInt = BigInteger.valueOf(intValue);
```

这会将 `intValue` 整数扩展为 `long`，使用符号位扩展以保持负值的负号不变。

将数字型 `String` 转换为 `BigInteger` 使用：

```
String decimalString = "-1";
BigInteger valueFromDecimalString = new BigInteger(decimalString);
```

以下构造函数用于将指定进制的`BigInteger`的字符串表示转换为 `BigInteger`。

```
String binaryString = "10";
int binaryRadix = 2;
BigInteger valueFromBinaryString = new BigInteger(binaryString, binaryRadix);
```

Java 还支持将字节直接转换为`BigInteger`实例。目前仅可使用有符号和无符号的大端编码：

```
byte[] bytes = new byte[] { (byte) 0x80 };
BigInteger valueFromBytes = new BigInteger(bytes);
```

这将生成一个值为 -128 的`BigInteger`实例，因为第一个位被解释为符号位。

```
byte[] unsignedBytes = new byte[] { (byte) 0x80 };
int sign = 1; // positive
BigInteger valueFromUnsignedBytes = new BigInteger(sign, unsignedBytes);
```

这将生成一个`BigInteger`实例，值为128，因为字节被解释为无符号数，并且符号被显式设置为1，即正数。

有一些预定义的常用值常量：

Chapter 20: BigInteger

The `BigInteger` class is used for mathematical operations involving large integers with magnitudes too large for primitive data types. For example 100-factorial is 158 digits - much larger than a `long` can represent. `BigInteger` provides analogues to all of Java's primitive integer operators, and all relevant methods from `java.lang.Math` as well as few other operations.

Section 20.1: Initialization

The `java.math.BigInteger` class provides operations analogues to all of Java's primitive integer operators and for all relevant methods from `java.lang.Math`. As the `java.math` package is not automatically made available you may have to import `java.math.BigInteger` before you can use the simple class name.

To convert `long` or `int` values to `BigInteger` use:

```
long longValue = Long.MAX_VALUE;
BigInteger valueFromLong = BigInteger.valueOf(longValue);
```

or, for integers:

```
int intValue = Integer.MIN_VALUE; // negative
BigInteger valueFromInt = BigInteger.valueOf(intValue);
```

which will widen the `intValue` integer to `long`, using sign bit extension for negative values, so that negative values will stay negative.

To convert a numeric `String` to `BigInteger` use:

```
String decimalString = "-1";
BigInteger valueFromDecimalString = new BigInteger(decimalString);
```

Following constructor is used to translate the String representation of a `BigInteger` in the specified radix into a `BigInteger`.

```
String binaryString = "10";
int binaryRadix = 2;
BigInteger valueFromBinaryString = new BigInteger(binaryString, binaryRadix);
```

Java also supports direct conversion of bytes to an instance of `BigInteger`. Currently only signed and unsigned big endian encoding may be used:

```
byte[] bytes = new byte[] { (byte) 0x80 };
BigInteger valueFromBytes = new BigInteger(bytes);
```

This will generate a `BigInteger` instance with value -128 as the first bit is interpreted as the sign bit.

```
byte[] unsignedBytes = new byte[] { (byte) 0x80 };
int sign = 1; // positive
BigInteger valueFromUnsignedBytes = new BigInteger(sign, unsignedBytes);
```

This will generate a `BigInteger` instance with value 128 as the bytes are interpreted as unsigned number, and the sign is explicitly set to 1, a positive number.

There are predefined constants for common values:

- BigInteger.ZERO — 值为“0”。
- BigInteger.ONE — 值为“1”。
- BigInteger.TEN — 值为“10”。

还有BigInteger.TWO（值为“2”），但你不能在代码中使用它，因为它是private的。

第20.2节：BigInteger数学运算示例

BigInteger是不可变对象，因此你需要将任何数学运算的结果赋值给一个新的BigInteger实例。

加法： $10 + 10 = 20$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("10");

BigInteger sum = value1.add(value2);
System.out.println(sum);
```

输出：20

减法： $10 - 9 = 1$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("9");

BigInteger sub = value1.subtract(value2);
System.out.println(sub);
```

输出：1

除法： $10 / 5 = 2$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

输出：2

除法： $17/4 = 4$

```
BigInteger value1 = new BigInteger("17");
BigInteger value2 = new BigInteger("4");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

输出：4

- BigInteger.ZERO — value of "0".
- BigInteger.ONE — value of "1".
- BigInteger.TEN — value of "10".

There's also BigInteger.TWO (value of "2"), but you can't use it in your code because it's **private**.

Section 20.2: BigInteger Mathematical Operations Examples

BigInteger is an immutable object, so you need to assign the results of any mathematical operation, to a new BigInteger instance.

Addition: $10 + 10 = 20$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("10");

BigInteger sum = value1.add(value2);
System.out.println(sum);
```

output: 20

Subtraction: $10 - 9 = 1$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("9");

BigInteger sub = value1.subtract(value2);
System.out.println(sub);
```

output: 1

Division: $10 / 5 = 2$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

output: 2

Division: $17/4 = 4$

```
BigInteger value1 = new BigInteger("17");
BigInteger value2 = new BigInteger("4");

BigInteger div = value1.divide(value2);
System.out.println(div);
```

output: 4

乘法： $10 * 5 = 50$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger mul = value1.multiply(value2);
System.out.println(mul);
```

输出：50

幂： $10 ^ 3 = 1000$

```
BigInteger value1 = new BigInteger("10");
BigInteger power = value1.pow(3);
System.out.println(power);
```

输出：1000

余数： $10 \% 6 = 4$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("6");

BigInteger power = value1.remainder(value2);
System.out.println(power);
```

输出：4

最大公约数 (GCD) : 12和18的最大公约数是6。

```
BigInteger value1 = new BigInteger("12");
BigInteger value2 = new BigInteger("18");

System.out.println(value1.gcd(value2));
```

输出：6

两个 BigInteger 的最大值：

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.max(value2));
```

输出：11

两个 BigInteger 的最小值：

Multiplication: $10 * 5 = 50$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("5");

BigInteger mul = value1.multiply(value2);
System.out.println(mul);
```

output: 50

Power: $10 ^ 3 = 1000$

```
BigInteger value1 = new BigInteger("10");
BigInteger power = value1.pow(3);
System.out.println(power);
```

output: 1000

Remainder: $10 \% 6 = 4$

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("6");

BigInteger power = value1.remainder(value2);
System.out.println(power);
```

output: 4

GCD: Greatest Common Divisor (GCD) for 12and 18 is 6.

```
BigInteger value1 = new BigInteger("12");
BigInteger value2 = new BigInteger("18");

System.out.println(value1.gcd(value2));
```

Output: 6

Maximum of two BigIntegers:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.max(value2));
```

Output: 11

Minimum of two BigIntegers:

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.min(value2));
```

输出：10

第20.3节：比较 BigInteger

你可以像比较 Java 中的 String 或其他对象一样比较 BigInteger。

例如：

```
BigInteger one = BigInteger.valueOf(1);
BigInteger two = BigInteger.valueOf(2);

if(one.equals(two)){
    System.out.println("Equal");
}
else{
    System.out.println("Not Equal");
}
```

输出：

不相等

注意：

通常情况下，不要使用==运算符来比较BigInteger

- ==运算符：比较引用；即两个值是否引用同一个对象
- equals()方法：比较两个BigInteger的内容。

例如，BigInteger不应该用以下方式比较：

```
if (firstBigInteger == secondBigInteger) {
    // 仅检查引用相等，不检查内容相等！
}
```

这样做可能导致意外行为，因为==运算符只检查引用相等。如果两个 BigInteger包含相同内容，但不指向同一个对象，这将失败。相反，应使用上述说明的equals方法比较 BigInteger。

你也可以将你的BigInteger与常量值如0、1、10进行比较。

例如：

```
BigInteger reallyBig = BigInteger.valueOf(1);
if(BigInteger.ONE.equals(reallyBig)){
    // 当它们相等时的代码。
}
```

你也可以使用compareTo()方法比较两个BigInteger，方法如下：compareTo()返回3个值。

```
BigInteger value1 = new BigInteger("10");
BigInteger value2 = new BigInteger("11");

System.out.println(value1.min(value2));
```

Output: 10

Section 20.3: Comparing BigIntegers

You can compare BigIntegers same as you compare String or other objects in Java.

For example:

```
BigInteger one = BigInteger.valueOf(1);
BigInteger two = BigInteger.valueOf(2);

if(one.equals(two)){
    System.out.println("Equal");
}
else{
    System.out.println("Not Equal");
}
```

Output:

Not Equal

Note:

In general, do **not** use the == operator to compare BigIntegers

- == operator: compares references; i.e. whether two values refer to the same object
- equals() method: compares the content of two BigIntegers.

For example, BigIntegers should **not** be compared in the following way:

```
if (firstBigInteger == secondBigInteger) {
    // Only checks for reference equality, not content equality!
}
```

Doing so may lead to unexpected behavior, as the == operator only checks for reference equality. If both BigIntegers contain the same content, but do not refer to the same object, **this will fail**. Instead, compare BigIntegers using the equals methods, as explained above.

You can also compare your BigInteger to constant values like 0,1,10.

for example:

```
BigInteger reallyBig = BigInteger.valueOf(1);
if(BigInteger.ONE.equals(reallyBig)){
    //code when they are equal.
}
```

You can also compare two BigIntegers by using compareTo() method, as following: compareTo() returns 3 values.

- 0: 当两者相等时。
- 1: 当第一个大于第二个（括号内的那个）时。
- -1: 当第一个小于第二个时。

```
BigInteger reallyBig = BigInteger.valueOf(10);
BigInteger reallyBig1 = BigInteger.valueOf(100);

if(reallyBig.compareTo(reallyBig1) == 0){
    //当两者相等时的代码。
}
else if(reallyBig.compareTo(reallyBig1) == 1){
    //当 reallyBig 大于 reallyBig1 时的代码。
}
else if(reallyBig.compareTo(reallyBig1) == -1){
    //当 reallyBig 小于 reallyBig1 时的代码。
}
```

- 0: When both are **equal**.
- 1: When first is **greater than** second (the one in brackets).
- -1: When first is **less than** second.

```
BigInteger reallyBig = BigInteger.valueOf(10);
BigInteger reallyBig1 = BigInteger.valueOf(100);

if(reallyBig.compareTo(reallyBig1) == 0){
    //code when both are equal.
}
else if(reallyBig.compareTo(reallyBig1) == 1){
    //code when reallyBig is greater than reallyBig1.
}
else if(reallyBig.compareTo(reallyBig1) == -1){
    //code when reallyBig is less than reallyBig1.
}
```

第20.4节：BigInteger上的二进制逻辑运算

BigInteger 也支持 Number 类型可用的二进制逻辑运算。与所有操作一样，它们是通过调用方法来实现的。

二进制或运算：

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.or(val2);
```

输出：11 (相当于 10 | 9)

二进制与运算：

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.and(val2);
```

输出：8 (相当于 10 & 9)

二进制异或：

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.xor(val2);
```

输出：3 (相当于 10 ^ 9)

右移：

```
BigInteger val1 = new BigInteger("10");
```

Section 20.4: Binary Logic Operations on BigInteger

BigInteger supports the binary logic operations that are available to **Number** types as well. As with all operations they are implemented by calling a method.

Binary Or:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.or(val2);
```

Output: 11 (which is equivalent to 10 | 9)

Binary And:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.and(val2);
```

Output: 8 (which is equivalent to 10 & 9)

Binary Xor:

```
BigInteger val1 = new BigInteger("10");
BigInteger val2 = new BigInteger("9");

val1.xor(val2);
```

Output: 3 (which is equivalent to 10 ^ 9)

RightShift:

```
BigInteger val1 = new BigInteger("10");
```

```
val1.shiftRight(1); // 参数为整数
```

输出：5 (相当于 $10 \gg 1$)

左移：

```
BigInteger val1 = new BigInteger("10");
```

```
val1.shiftLeft(1); // 这里参数应该是 Integer
```

输出：20 (相当于 $10 \ll 1$)

二进制取反 (非)：

```
BigInteger val1 = new BigInteger("10");
```

```
val1.not();
```

输出：5

NAND (与非)：*

```
BigInteger val1 = new BigInteger("10");  
BigInteger val2 = new BigInteger("9");
```

```
val1.andNot(val2);
```

输出：7

第20.5节：生成随机BigInteger

BigInteger类有一个专门用于生成随机BigInteger的构造函数，给定一个
`java.util.Random`的实例和一个int，指定BigInteger将拥有多少位。它的用法非常简单——
当你这样调用构造函数`BigInteger(int, Random)`时：

```
BigInteger randomBigInt = new BigInteger(bitCount, sourceOfRandomness);
```

那么你最终会得到一个值在0 (含) 和`2bitCount` (不含) 之间的BigInteger。

这也意味着`new BigInteger(2147483647, sourceOfRandomness)`可能会在足够时间内返回所有正的BigInteger。

什么是`sourceOfRandomness`由你决定。例如，`new Random()`在大多数情况下已经足够好：

```
new BigInteger(32, new Random());
```

如果你愿意牺牲速度以获得更高质量的随机数，可以使用`new SecureRandom()`代替：
<https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html>

```
val1.shiftRight(1); // the argument be an Integer
```

Output: 5 (equivalent to $10 \gg 1$)

LeftShift:

```
BigInteger val1 = new BigInteger("10");
```

```
val1.shiftLeft(1); // here parameter should be Integer
```

Output: 20 (equivalent to $10 \ll 1$)

Binary Inversion (Not):

```
BigInteger val1 = new BigInteger("10");
```

```
val1.not();
```

Output: 5

*NAND (And-Not):**

```
BigInteger val1 = new BigInteger("10");  
BigInteger val2 = new BigInteger("9");
```

```
val1.andNot(val2);
```

Output: 7

Section 20.5: Generating random BigIntegers

The `BigInteger` class has a constructor dedicated to generate random BigIntegers, given an instance of `java.util.Random` and an int that specifies how many bits will the `BigInteger` have. Its usage is quite simple - when you call the constructor `BigInteger(int, Random)` like this:

```
BigInteger randomBigInt = new BigInteger(bitCount, sourceOfRandomness);
```

then you'll end up with a `BigInteger` whose value is between 0 (inclusive) and `2bitCount` (exclusive).

This also means that `new BigInteger(2147483647, sourceOfRandomness)` may return all positive `BigIntegers` given enough time.

What will the `sourceOfRandomness` be is up to you. For example, a `new Random()` is good enough in most cases:

```
new BigInteger(32, new Random());
```

If you're willing to give up speed for higher-quality random numbers, you can use a `new SecureRandom()` instead:
<https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html>

```
import java.security.SecureRandom;  
  
// 代码中的某处...  
new BigInteger(32, new SecureRandom());
```

你甚至可以用匿名类即时实现一个算法！注意，自己实现的RNG算法通常会导致低质量的随机性，因此除非你想让生成的BigInteger可预测，否则务必使用经过验证的合适算法。

```
new BigInteger(32, new Random() {  
    int seed = 0;  
  
    @Override  
    protected int next(int bits) {  
        seed = ((22695477 * seed) + 1) & 2147483647; // 这些数值是无耻地从  
        ="https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use" rel="nofollow  
        norefferrer">Wikipedia 返回 seed; }  
}
```

```
import java.security.SecureRandom;  
  
// somewhere in the code...  
new BigInteger(32, new SecureRandom());
```

You can even implement an algorithm on-the-fly with an anonymous class! Note that **rolling out your own RNG algorithm will end you up with low quality randomness**, so always be sure to use an algorithm that is proven to be decent unless you want the resulting BigInteger(s) to be predictable.

```
new BigInteger(32, new Random() {  
    int seed = 0;  
  
    @Override  
    protected int next(int bits) {  
        seed = ((22695477 * seed) + 1) & 2147483647; // Values shamelessly stolen from  
        ="https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use" rel="nofollow  
        norefferrer">Wikipedia return seed; } });
```

第21章：数字格式 (NumberFormat)

第21.1节：数字格式 (NumberFormat)

不同国家有不同的数字格式，考虑到这一点，我们可以使用Java的Locale来实现不同的格式。使用Locale可以帮助格式化

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

使用上述格式，你可以执行各种操作

1. 格式化数字

```
numberFormat.format(10000000.99);
```

2. 格式化货币

```
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
currencyFormat.format(10340.999);
```

3. 格式化百分比

```
NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
percentageFormat.format(10929.999);
```

4. 控制数字位数

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

Chapter 21: NumberFormat

Section 21.1: NumberFormat

Different countries have different number formats and considering this we can have different formats using Locale of java. Using locale can help in formatting

```
Locale locale = new Locale("en", "IN");
NumberFormat numberFormat = NumberFormat.getInstance(locale);
```

using above format you can perform various tasks

1. Format Number

```
numberFormat.format(10000000.99);
```

2. Format Currency

```
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
currencyFormat.format(10340.999);
```

3. Format Percentage

```
NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
percentageFormat.format(10929.999);
```

4. Control Number of Digits

```
numberFormat.setMinimumIntegerDigits(int digits)
numberFormat.setMaximumIntegerDigits(int digits)
numberFormat.setMinimumFractionDigits(int digits)
numberFormat.setMaximumFractionDigits(int digits)
```

第22章：位操作

第22.1节：检查、设置、清除和切换单个位。使用long作为位掩码

假设我们想要修改整数原始类型 n 的位， i (byte、short、char、int或long) :

```
(i & 1 << n) != 0 // 检查第'n'位  
i |= 1 << n;      // 将第'n'位设置为1  
i &= ~(1 << n); // 将第'n'位设置为0  
i ^= 1 << n;      // 切换第'n'位的值
```

使用 long/int/short/byte 作为位掩码：

```
public class BitMaskExample {  
    private static final long FIRST_BIT = 1L << 0;  
    private static final long SECOND_BIT = 1L << 1;  
    private static final long THIRD_BIT = 1L << 2;  
    private static final long FOURTH_BIT = 1L << 3;  
    private static final long FIFTH_BIT = 1L << 4;  
    private static final long BIT_55 = 1L << 54;  
  
    public static void main(String[] args) {  
        checkBitMask(FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55);  
    }  
  
    private static void checkBitMask(long bitmask) {  
        System.out.println("FIRST_BIT: " + ((bitmask & FIRST_BIT) != 0));  
        System.out.println("SECOND_BIT: " + ((bitmask & SECOND_BIT) != 0));  
        System.out.println("THIRD_BIT: " + ((bitmask & THIRD_BIT) != 0));  
        System.out.println("FOURTH_BIT: " + ((bitmask & FOURTH_BIT) != 0));  
        System.out.println("FIFTH_BIT: " + ((bitmask & FIFTH_BIT) != 0));  
        System.out.println("BIT_55: " + ((bitmask & BIT_55) != 0));  
    }  
}
```

打印结果

```
FIRST_BIT: true  
SECOND_BIT: false  
THIRD_BIT: true  
FOURTH_BIT: false  
FIFTH_BIT: true  
BIT_55: true
```

这与我们作为checkBitMask参数传入的掩码匹配：FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55。

第22.2节：java.util.BitSet类

从1.7版本开始，有一个[java.util.BitSet](#)类，提供了简单且用户友好的位存储和操作接口：

```
final BitSet bitSet = new BitSet(8); // 默认所有位都未设置  
  
IntStream.range(0, 8).filter(i -> i % 2 == 0).forEach(bitSet::set); // {0, 2, 4, 6}  
  
bitSet.set(3); // {0, 2, 3, 4, 6}
```

Chapter 22: Bit Manipulation

Section 22.1: Checking, setting, clearing, and toggling individual bits. Using long as bit mask

Assuming we want to modify bit n of an integer primitive, i (byte, short, char, int, or long):

```
(i & 1 << n) != 0 // checks bit 'n'  
i |= 1 << n;      // sets bit 'n' to 1  
i &= ~(1 << n); // sets bit 'n' to 0  
i ^= 1 << n;      // toggles the value of bit 'n'
```

Using long/int/short/byte as a bit mask:

```
public class BitMaskExample {  
    private static final long FIRST_BIT = 1L << 0;  
    private static final long SECOND_BIT = 1L << 1;  
    private static final long THIRD_BIT = 1L << 2;  
    private static final long FOURTH_BIT = 1L << 3;  
    private static final long FIFTH_BIT = 1L << 4;  
    private static final long BIT_55 = 1L << 54;  
  
    public static void main(String[] args) {  
        checkBitMask(FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55);  
    }  
  
    private static void checkBitMask(long bitmask) {  
        System.out.println("FIRST_BIT: " + ((bitmask & FIRST_BIT) != 0));  
        System.out.println("SECOND_BIT: " + ((bitmask & SECOND_BIT) != 0));  
        System.out.println("THIRD_BIT: " + ((bitmask & THIRD_BIT) != 0));  
        System.out.println("FOURTH_BIT: " + ((bitmask & FOURTH_BIT) != 0));  
        System.out.println("FIFTH_BIT: " + ((bitmask & FIFTH_BIT) != 0));  
        System.out.println("BIT_55: " + ((bitmask & BIT_55) != 0));  
    }  
}
```

Prints

```
FIRST_BIT: true  
SECOND_BIT: false  
THIRD_BIT: true  
FOURTH_BIT: false  
FIFTH_BIT: true  
BIT_55: true
```

which matches that mask we passed as checkBitMask parameter: FIRST_BIT | THIRD_BIT | FIFTH_BIT | BIT_55.

Section 22.2: java.util.BitSet class

Since 1.7 there's a [java.util.BitSet](#) class that provides simple and user-friendly bit storage and manipulation interface:

```
final BitSet bitSet = new BitSet(8); // by default all bits are unset  
  
IntStream.range(0, 8).filter(i -> i % 2 == 0).forEach(bitSet::set); // {0, 2, 4, 6}  
  
bitSet.set(3); // {0, 2, 3, 4, 6}
```

```

bitSet.set(3, false); // {0, 2, 4, 6}

final boolean b = bitSet.get(3); // b = false

bitSet.flip(6); // {0, 2, 4}

bitSet.set(100); // {0, 2, 4, 100} - 自动扩展

```

`BitSet` 实现了 `Cloneable` 和 `Serializable`, 底层所有位值都存储在 `long[] words` 字段中, 该字段会自动扩展。

它还支持全集合的逻辑操作 `and`、`or`、`xor`、`andNot`:

```

bitSet.and(new BitSet(8));
bitSet.or(new BitSet(8));
bitSet.xor(new BitSet(8));
bitSet.andNot(new BitSet(8));

```

第22.3节：检查一个数是否是2的幂

如果一个整数 `x` 是2的幂, 则只有一位被设置, 而 `x-1` 在该位之后的所有位都被设置。例如: 4 的二进制是 100, 3的二进制是 011, 满足上述条件。零不是2的幂, 必须显式检查。

```

boolean isPowerOfTwo(int x)
{
    return (x != 0) && ((x & (x - 1)) == 0);
}

```

左移和右移的用法

假设我们有三种权限, `READ`、`WRITE` 和 `EXECUTE`。每种权限的范围是0到7。 (假设是4位数字系统)

```

资源 = 读 写 执行 (12位数字)

资源 = 0100 0110 0101 = 4 6 5 (12位数字)

```

我们如何获取上面设置的 (12位数字) 权限?

```

0100 0110 0101

0000 0000 0111 (&)

0000 0000 0101 = 5

```

所以, 这就是我们如何获取执行权限的资源。现在, 如果我们想获取读取权限的资源, 该怎么办?

```

0100 0110 0101

0111 0000 0000 (&)

```

```

bitSet.set(3, false); // {0, 2, 4, 6}

final boolean b = bitSet.get(3); // b = false

bitSet.flip(6); // {0, 2, 4}

bitSet.set(100); // {0, 2, 4, 100} - expands automatically

```

`BitSet` implements `Cloneable` and `Serializable`, and under the hood all bit values are stored in `long[] words` field, that expands automatically.

It also supports whole-set logical operations `and`, `or`, `xor`, `andNot`:

```

bitSet.and(new BitSet(8));
bitSet.or(new BitSet(8));
bitSet.xor(new BitSet(8));
bitSet.andNot(new BitSet(8));

```

Section 22.3: Checking if a number is a power of 2

If an integer `x` is a power of 2, only one bit is set, whereas `x-1` has all bits set after that. For example: 4 is 100 and 3 is 011 as binary number, which satisfies the aforementioned condition. Zero is not a power of 2 and has to be checked explicitly.

```

boolean isPowerOfTwo(int x)
{
    return (x != 0) && ((x & (x - 1)) == 0);
}

```

Usage for Left and Right Shift

Let's suppose, we have three kind of permissions, `READ`, `WRITE` and `EXECUTE`. Each permission can range from 0 to 7. (Let's assume 4 bit number system)

```

RESOURCE = READ WRITE EXECUTE (12 bit number)

RESOURCE = 0100 0110 0101 = 4 6 5 (12 bit number)

```

How can we get the (12 bit number) permissions, set on above (12 bit number)?

```

0100 0110 0101

0000 0000 0111 (&)

0000 0000 0101 = 5

```

So, this is how we can get the `EXECUTE` permissions of the `RESOURCE`. Now, what if we want to get `READ` permissions of the `RESOURCE`?

```

0100 0110 0101

0111 0000 0000 (&)

```

0100 0000 0000 = 1024

对吧？你可能是这么认为的？但是，权限结果是1024。我们只想获取资源的读取权限。别担心，这就是我们使用移位运算符的原因。如果我们看，读取权限比实际结果落后8位，那么如果应用某个移位运算符，将读取权限移到结果的最右边会怎样？如果我们这样做：

0100 0000 0000 >> 8 => 0000 0000 0100 (因为它是正数，所以用0替代，如果你不关心符号，直接使用无符号右移运算符)

我们现在实际上得到了读取权限，其值为4。

现在，例如，我们获得了读取、写入、执行权限的资源，我们可以做些什么来为这个资源设置权限？

让我们先以二进制权限为例。（仍假设4位数字系统）

读取 = 0001

写入 = 0100

执行 = 0110

如果你认为我们只需简单地做：

读取 | 写入 | 执行，你的想法有一定道理，但不完全正确。看看如果我们执行 读取 |
写入 | 执行 会发生什么

0001 | 0100 | 0110 => 0111

但权限实际上（在我们的例子中）表示为 0001 0100 0110

所以，为了做到这一点，我们知道 读取 位于8位之后，写入 位于4位之后，权限 位于最后。用于 资源 权限的数字系统实际上是12位（在我们的例子中）。它 在不同系统中可能（会）不同。

(读取 << 8) | (写入 << 4) | (执行)
0000 0000 0001 << 8 (读取)
0001 0000 0000 (左移8位)
0000 0000 0100 << 4 (写入)
0000 0100 0000 (左移4位)
0000 0000 0001 (执行)

现在如果我们将上述移位结果相加，结果将类似于；

0100 0000 0000 = 1024

Right? You are probably assuming this? But, permissions are resulted in 1024. We want to get only READ permissions for the resource. Don't worry, that's why we had the shift operators. If we see, READ permissions are 8 bits behind the actual result, so if apply some shift operator, which will bring READ permissions to the very right of the result? What if we do:

0100 0000 0000 >> 8 => 0000 0000 0100 (Because it's a positive number so replaced with 0's, if you don't care about sign, just use unsigned right shift operator)

We now actually have the **READ** permissions which is 4.

Now, for example, we are given **READ, WRITE, EXECUTE** permissions for a **RESOURCE**, what can we do to make permissions for this **RESOURCE**?

Let's first take the example of binary permissions. (Still assuming 4 bit number system)

READ = 0001

WRITE = 0100

EXECUTE = 0110

If you are thinking that we will simply do:

READ | WRITE | EXECUTE, you are somewhat right but not exactly. See, what will happen if we will perform READ |
WRITE | EXECUTE

0001 | 0100 | 0110 => 0111

But permissions are actually being represented (in our example) as 0001 0100 0110

So, in order to do this, we know that **READ** is placed 8 bits behind, **WRITE** is placed 4 bits behind and **PERMISSIONS** is placed at the last. The number system being used for **RESOURCE** permissions is actually 12 bit (in our example). It can(will) be different in different systems.

(READ << 8) | (WRITE << 4) | (EXECUTE)
0000 0000 0001 << 8 (READ)
0001 0000 0000 (Left shift by 8 bits)
0000 0000 0100 << 4 (WRITE)
0000 0100 0000 (Left shift by 4 bits)
0000 0000 0001 (EXECUTE)

Now if we add the results of above shifting, it will be something like;

```
0001 0000 0000 (读取)  
0000 0100 0000 (写入)  
0000 0000 0001 (执行)  
0001 0100 0001 (权限)
```

```
0001 0000 0000 (READ)  
0000 0100 0000 (WRITE)  
0000 0000 0001 (EXECUTE)  
0001 0100 0001 (PERMISSIONS)
```

第22.4节：有符号与无符号移位

在Java中，所有数字基本类型都是有符号的。例如，int总是表示 $[-2^{31} - 1, 2^{31}]$ 范围内的值，保留第一位作为符号位——1表示负值，0表示正值。

基本移位运算符`>>`和`<<`是有符号运算符。它们会保持值的符号。

但程序员常常使用数字来存储无符号值。对于int来说，这意味着将范围移位到 $[0, 2^{32} - 1]$ ，使得值的范围是有符号int的两倍。

对于这些高级用户来说，符号位没有意义。这就是为什么Java添加了`>>>`，一个无符号右移运算符，忽略了符号位。

```
初始值: 4 (  
    有符号左-移位: 4 << 1 8 (  
        有符号右-移位: 4 >> 1 2 (  
            无符号右-移位: 4 >>> 1 2 (  
                初始值: -4 ( 111111111111111111111111111100 )  
                有符号左-移位: -4 << 1 -8 ( 111111111111111111111111111100 )  
                有符号右-移位: -4 >> 1 -2 ( 111111111111111111111111111100 )  
                无符号右-移位: -4 >>> 1 2147483646 ( 111111111111111111111111111100 )
```

Section 22.4: Signed vs unsigned shift

In Java, all number primitives are signed. For example, an int always represent values from $[-2^{31} - 1, 2^{31}]$, keeping the first bit to sign the value - 1 for negative value, 0 for positive.

Basic shift operators `>>` and `<<` are signed operators. They will conserve the sign of the value.

But it is common for programmers to use numbers to store *unsigned values*. For an int, it means shifting the range to $[0, 2^{32} - 1]$, to have twice as much value as with a signed int.

For those power users, the bit for sign as no meaning. That's why Java added `>>>`, a left-shift operator, disregarding that sign bit.

```
initial value: 4 (  
    signed left-shift: 4 << 1 8 (  
        signed right-shift: 4 >> 1 2 (  
            unsigned right-shift: 4 >>> 1 2 (  
                initial value: -4 ( 111111111111111111111111111100 )  
                signed left-shift: -4 << 1 -8 ( 111111111111111111111111111100 )  
                signed right-shift: -4 >> 1 -2 ( 111111111111111111111111111100 )  
                unsigned right-shift: -4 >>> 1 2147483646 ( 111111111111111111111111111100 )
```

为什么没有`<<<`？

这是因为右移的定义。由于它在左侧填充空位，不需要考虑符号位。因此，不需要两个不同的运算符。

详见此问题以获取更详细的答案。

第22.5节：表示2的幂

对于表示整数的2的幂 (2^n)，可以使用位移操作来显式指定

语法基本如下：

```
int pow2 = 1<<n;
```

示例：

```
int twoExp4 = 1<<4; //2^4  
int twoExp5 = 1<<5; //2^5  
int twoExp6 = 1<<6; //2^6  
...  
int twoExp31 = 1<<31; //2^31
```

这在定义常量值时尤其有用，可以明确表示使用的是2的幂，

Why is there no `<<<`?

This comes from the intended definition of right-shift. As it fills the emptied places on the left, there are no decision to take regarding the bit of sign. As a consequence, there is no need for 2 different operators.

See this [question](#) for a more detailed answer.

Section 22.5: Expressing the power of 2

For expressing the power of 2 (2^n) of integers, one may use a bitshift operation that allows to explicitly specify the n .

The syntax is basically:

```
int pow2 = 1<<n;
```

Examples:

```
int twoExp4 = 1<<4; //2^4  
int twoExp5 = 1<<5; //2^5  
int twoExp6 = 1<<6; //2^6  
...  
int twoExp31 = 1<<31; //2^31
```

This is especially useful when defining constant values that should make it apparent, that a power of 2 is used,

而不是使用十六进制或十进制值。

```
int twoExp4 = 0x10; //十六进制
int twoExp5 = 0x20; //十六进制
int twoExp6 = 64; //十进制
...
int twoExp31 = -2147483648; //这是2的幂吗？
```

计算2的整数次幂的简单方法是

```
int pow2(int exp){
    return 1<<exp;
}
```

第22.6节：作为位片段的值的打包/解包

为了提高内存性能，通常会将多个值压缩到一个基本类型的值中。这对于将各种信息传递到单个变量中可能很有用。

例如，可以将3个字节——例如RGB颜色代码——打包到一个int中。

打包这些值

```
// 原始字节作为输入
byte[] b = {(byte)0x65, (byte)0xFF, (byte)0x31};

// 以大端序打包：x == 0x65FF31
int x = (b[0] & 0xFF) << 16 // 红色
| (b[1] & 0xFF) << 8 // 绿色
| (b[2] & 0xFF) << 0; // 蓝色

// 以小端序打包：y == 0x31FF65
int y = (b[0] & 0xFF) << 0
| (b[1] & 0xFF) << 8
| (b[2] & 0xFF) << 16;
```

解包数值

```
// 原始 int32 作为输入
int x = 0x31FF65;

// 大端序解包: {0x65, 0xFF, 0x31}
byte[] c = {
    (byte)(x >> 16),
    (byte)(x >> 8),
    (byte)(x & 0xFF)
};

// 小端序解包: {0x31, 0xFF, 0x65}
byte[] d = {
    (byte)(x & 0xFF),
    (byte)(x >> 8),
    (byte)(x >> 16)
};
```

instead of using hexadecimal or decimal values.

```
int twoExp4 = 0x10; //hexadecimal
int twoExp5 = 0x20; //hexadecimal
int twoExp6 = 64; //decimal
...
int twoExp31 = -2147483648; //is that a power of 2?
```

A simple method to calculate the int power of 2 would be

```
int pow2(int exp){
    return 1<<exp;
}
```

Section 22.6: Packing / unpacking values as bit fragments

It is common for memory performance to compress multiple values into a single primitive value. This may be useful to pass various information into a single variable.

For example, one can pack 3 bytes - such as color code in [RGB](#) - into a single int.

Packing the values

```
// Raw bytes as input
byte[] b = {(byte)0x65, (byte)0xFF, (byte)0x31};

// Packed in big endian: x == 0x65FF31
int x = (b[0] & 0xFF) << 16 // Red
| (b[1] & 0xFF) << 8 // Green
| (b[2] & 0xFF) << 0; // Blue

// Packed in little endian: y == 0x31FF65
int y = (b[0] & 0xFF) << 0
| (b[1] & 0xFF) << 8
| (b[2] & 0xFF) << 16;
```

Unpacking the values

```
// Raw int32 as input
int x = 0x31FF65;

// Unpacked in big endian: {0x65, 0xFF, 0x31}
byte[] c = {
    (byte)(x >> 16),
    (byte)(x >> 8),
    (byte)(x & 0xFF)
};

// Unpacked in little endian: {0x31, 0xFF, 0x65}
byte[] d = {
    (byte)(x & 0xFF),
    (byte)(x >> 8),
    (byte)(x >> 16)
};
```

第23章：数组

参数	详情
ArrayType	ArrayType数组的类型。可以是基本类型 (int, long, byte) 或对象类型 (String, MyObject 等)。
索引	索引指的是数组中某个对象的位置。

长度	每个数组在创建时都需要指定一个固定长度。这可以在创建空数组时完成 (new int[3])，也可以在指定值时隐含确定 ({1, 2, 3})。
----	---

数组允许存储和检索任意数量的值。它们类似于数学中的向量。数组的数组类似于矩阵，充当多维数组。数组可以存储任何类型的数据：基本类型如int或引用类型如Object。

第23.1节：创建和初始化数组

基本情况

```
int[] numbers1 = new int[3]; // 用于3个int值的数组, 默认值为0
int[] numbers2 = { 1, 2, 3 }; // 3个int值的数组字面量
int[] numbers3 = new int[] { 1, 2, 3 }; // 初始化了3个int值的数组
int[][] numbers4 = { { 1, 2 }, { 3, 4, 5 } }; // 锯齿状数组字面量
int[][] numbers5 = new int[5][]; // 锯齿状数组, 一维长度为5
int[][] numbers6 = new int[5][4]; // 多维数组: 5x4
```

数组可以使用任何原始类型或引用类型创建。

```
float[] boats = new float[5]; // 五个32位浮点数的数组。
double[] header = new double[] { 4.56, 332.267, 7.0, 0.3367, 10.0 };
String[] theory = new String[] { "a", "b", "c" };
Object[] dArt = new Object[] { new Object(), "We love Stack Overflow.", new Integer(3) };
// 三个字符串的数组 (引用类型)。
// 三个对象的数组 (引用类型)。
```

对于最后一个例子，请注意数组中允许声明数组类型的子类型。

用户自定义类型的数组也可以类似于原始类型构建。

```
UserDefinedClass[] udType = new UserDefinedClass[5];
```

数组、集合和流

版本 ≥ Java SE 1.2

// 参数需要对象，而非基本类型

```
// 这里对 int 127 进行了自动装箱
Integer[] initial = { 127, Integer.valueOf( 42 ) };
List<Integer> toList = Arrays.asList( initial ); // 固定大小!
```

```
// 注意：适用于所有集合
Integer[] fromCollection = toList.toArray( new Integer[toList.size()] );
```

```
// Java 不允许创建参数化类型的数组
List<String>[] list = new ArrayList<String>[2]; // 编译错误！
```

版本 ≥ Java SE 8

```
// 流 - JDK 8 及以上
Stream<Integer> toStream = Arrays.stream( initial );
Integer[] fromStream = toStream.toArray( Integer[]::new );
```

介绍

Chapter 23: Arrays

Parameter

ArrayType Type of the array. This can be primitive (`int, long, byte`) or Objects (`String, MyObject, etc.`).

index Index refers to the position of a certain Object in an array.

length Every array, when being created, needs a set length specified. This is either done when creating an empty array (`new int[3]`) or implied when specifying values ({1, 2, 3}).

Arrays allow for the storage and retrieval of an arbitrary quantity of values. They are analogous to vectors in mathematics. Arrays of arrays are analogous to matrices, and act as multidimensional arrays. Arrays can store any data of any type: primitives such as `int` or reference types such as `Object`.

Section 23.1: Creating and Initializing Arrays

Basic cases

```
int[] numbers1 = new int[3]; // Array for 3 int values, default value is 0
int[] numbers2 = { 1, 2, 3 }; // Array literal of 3 int values
int[] numbers3 = new int[] { 1, 2, 3 }; // Array of 3 int values initialized
int[][] numbers4 = { { 1, 2 }, { 3, 4, 5 } }; // Jagged array literal
int[][] numbers5 = new int[5][]; // Jagged array, one dimension 5 long
int[][] numbers6 = new int[5][4]; // Multidimensional array: 5x4
```

Arrays may be created using any primitive or reference type.

```
float[] boats = new float[5]; // Array of five 32-bit floating point numbers.
double[] header = new double[] { 4.56, 332.267, 7.0, 0.3367, 10.0 };
String[] theory = new String[] { "a", "b", "c" };
Object[] dArt = new Object[] { new Object(), "We love Stack Overflow.", new Integer(3) };
// Array of five 64-bit floating point numbers.
// Array of three strings (reference type).
// Array of three Objects (reference type).
```

For the last example, note that subtypes of the declared array type are allowed in the array.

Arrays for user defined types can also be built similar to primitive types

```
UserDefinedClass[] udType = new UserDefinedClass[5];
```

Arrays, Collections, and Streams

Version ≥ Java SE 1.2

// Parameters require objects, not primitives

```
// Auto-boxing happening for int 127 here
Integer[] initial = { 127, Integer.valueOf( 42 ) };
List<Integer> toList = Arrays.asList( initial ); // Fixed size!
```

```
// Note: Works with all collections
Integer[] fromCollection = toList.toArray( new Integer[toList.size()] );
```

```
// Java doesn't allow you to create an array of a parameterized type
List<String>[] list = new ArrayList<String>[2]; // Compilation error!
```

Version ≥ Java SE 8

```
// Streams - JDK 8+
Stream<Integer> toStream = Arrays.stream( initial );
Integer[] fromStream = toStream.toArray( Integer[]::new );
```

Intro

数组是一种数据结构，用于存储固定数量的原始值或对象实例的引用。

数组中的每个项称为元素，每个元素通过其数字索引访问。数组的长度在创建时确定：

```
int size = 42;  
int[] array = new int[size];
```

数组的大小在初始化时运行时固定。初始化后不能更改。如果大小必须在运行时可变，应使用如ArrayList的Collection类。ArrayList将元素存储在数组中，并通过分配新数组并从旧数组复制元素来支持调整大小。

如果数组是原始类型，即

```
int[] array1 = { 1, 2, 3 };  
int[] array2 = new int[10];
```

值存储在数组本身中。如果没有初始化器（如上面的array2），则每个元素的默认值为0（零）。

如果数组类型是对象引用，如

```
SomeClassOrInterface[] array = new SomeClassOrInterface[10];
```

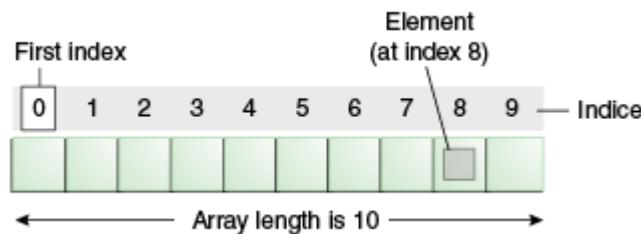
那么数组包含对SomeClassOrInterface类型对象的引用。这些引用可以指向SomeClassOrInterface的实例，或其任何子类（对于类）或实现类（对于接口）。如果数组声明没有初始化器，则每个元素将被赋予默认值null。

因为所有数组都是用int索引的，数组的大小必须由一个int指定。数组的大小不能被指定为long类型：

```
long size = 23L;  
int[] array = new int[size]; // 编译时错误：  
// 类型不兼容：可能从  
long转换到int时发生精度丢失
```

数组使用零基索引系统，这意味着索引从0开始，到length - 1结束。

例如，下面的图表示一个大小为10的数组。这里，第一个元素的索引是0，最后一个元素的索引是9，而不是第一个元素的索引是1，最后一个元素的索引是10（见下图）。



对数组元素的访问是在常数时间内完成的。这意味着访问数组的第一个元素与访问第二个元素、第三个元素等的时间成本相同。

Java提供了多种定义和初始化数组的方法，包括字面量和构造函数表示法。当

An array is a data structure that holds a fixed number of primitive values or references to object instances.

Each item in an array is called an element, and each element is accessed by its numerical index. The length of an array is established when the array is created:

```
int size = 42;  
int[] array = new int[size];
```

The size of an array is fixed at runtime when initialized. It cannot be changed after initialization. If the size must be mutable at runtime, a [Collection](#) class such as [ArrayList](#) should be used instead. [ArrayList](#) stores elements in an array and supports resizing by allocating a new array and copying elements from the old array.

If the array is of a primitive type, i.e.

```
int[] array1 = { 1, 2, 3 };  
int[] array2 = new int[10];
```

the values are stored in the array itself. In the absence of an initializer (as in array2 above), the default value assigned to each element is 0 (zero).

If the array type is an object reference, as in

```
SomeClassOrInterface[] array = new SomeClassOrInterface[10];
```

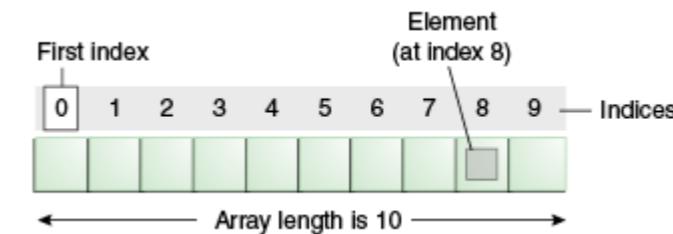
then the array contains *references* to objects of type SomeClassOrInterface. Those references can refer to an instance of SomeClassOrInterface or any subclass (for classes) or implementing class (for interfaces) of SomeClassOrInterface. If the array declaration has no initializer then the default value of [null](#) is assigned to each element.

Because all arrays are **int-indexed**, the size of an array must be specified by an **int**. The size of the array cannot be specified as a **long**:

```
long size = 23L;  
int[] array = new int[size]; // Compile-time error:  
// incompatible types: possible lossy conversion from  
// long to int
```

Arrays use a **zero-based index** system, which means indexing starts at 0 and ends at length - 1.

For example, the following image represents an array with size 10. Here, the first element is at index 0 and the last element is at index 9, instead of the first element being at index 1 and the last element at index 10 (see figure below).



Accesses to elements of arrays are done in **constant time**. That means accessing to the first element of the array has the same cost (in time) of accessing the second element, the third element and so on.

Java offers several ways of defining and initializing arrays, including **literal** and **constructor** notations. When

使用`new Type[length]`构造函数声明数组时，每个元素将被初始化为以下默认值：

- 0针对原始数值类型：`byte`、`short`、`int`、`long`、`float`和`double`。
- 字符类型`char`的'`\u0000`'（空字符）。
- 布尔类型`boolean`的`false`。
- 引用类型`reference types`的`null`。

创建和初始化原始类型数组

```
int[] array1 = new int[] { 1, 2, 3 }; // 使用new操作符和  
// 数组初始化器创建数组。  
int[] array2 = { 1, 2, 3 }; // 使用数组初始化器的简写语法。  
int[] array3 = new int[3]; // 等同于 { 0, 0, 0 }  
int[] array4 = null; // 数组本身是对象，因此  
// 可以被赋值为null。
```

声明数组时，`[]`会作为类型的一部分出现在声明开头（类型名之后），或者作为特定变量的声明符一部分（变量名之后），也可能两者都有：

```
int array5[]; /* 等同于 */ int[] array5;  
int a, b[], c[][]; /* 等同于 */ int a; int[] b; int[][] c;  
int[] a, b[]; /* 等同于 */ int[] a; int[][] b;  
int a, []b, c[][]; /* 编译错误，因为 [] 不是声明开头类型的一部分，而是在 'b' 之前。 */  
  
// 声明返回数组的方法时，适用相同的规则：  
int foo()[] { ... } /* 等同于 */ int[] foo() { ... }
```

在下面的示例中，两种声明都是正确的，可以编译并正常运行。然而，Java 编码规范和Google Java 风格指南都不推荐在变量名后面加括号的形式——括号用于标识数组类型，应与类型声明一起出现。方法返回签名也应遵循相同的规则。

```
float array[]; /* 和 */ int foo()[] { ... } /* 不推荐 */  
float[] array; /* 和 */ int[] foo() { ... } /* 推荐 */
```

不推荐的类型是为了适应习惯于 C 语法规的用户设计的，C 语言中括号位于变量名之后。

在 Java 中，可以创建大小为0的数组：

```
int[] array = new int[0]; // 可以编译并正常运行。  
int[] array2 = {};// 等效语法。
```

但是，由于这是一个空数组，无法从中读取或赋值任何元素：

```
array[0] = 1; // 抛出 java.lang.ArrayIndexOutOfBoundsException 异常。  
int i = array2[0]; // 也会抛出 ArrayIndexOutOfBoundsException。
```

这种空数组通常用作返回值，这样调用代码只需处理一个数组，而不必担心可能导致 `NullPointerException` 的空值（`null`）。

数组的长度必须是非负整数：

```
int[] array = new int[-1]; // 抛出 java.lang.NegativeArraySizeException
```

declaring arrays using the `new Type[length]` constructor, each element will be initialized with the following default values:

- 0 for primitive numerical types: `byte`, `short`, `int`, `long`, `float`, and `double`.
- '`\u0000`' (null character) for the `char` type.
- `false` for the `boolean` type.
- `null` for `reference types`.

Creating and initializing primitive type arrays

```
int[] array1 = new int[] { 1, 2, 3 }; // Create an array with new operator and  
// array initializer.  
int[] array2 = { 1, 2, 3 }; // Shortcut syntax with array initializer.  
int[] array3 = new int[3]; // Equivalent to { 0, 0, 0 }  
int[] array4 = null; // The array itself is an object, so it  
// can be set as null.
```

When declaring an array, `[]` will appear as part of the type at the beginning of the declaration (after the type name), or as part of the declarator for a particular variable (after variable name), or both:

```
int array5[]; /* equivalent to */ int[] array5;  
int a, b[], c[][]; /* equivalent to */ int a; int[] b; int[][] c;  
int[] a, b[]; /* equivalent to */ int[] a; int[][] b;  
int a, []b, c[][]; /* Compilation Error, because [] is not part of the type at beginning  
of the declaration, rather it is before 'b'. */  
// The same rules apply when declaring a method that returns an array:  
int foo()[] { ... } /* equivalent to */ int[] foo() { ... }
```

In the following example, both declarations are correct and can compile and run without any problems. However, both the [Java Coding Convention](#) and the [Google Java Style Guide](#) discourage the form with brackets after the variable name—the brackets identify the array type and should appear with the type designation. The same should be used for method return signatures.

```
float array[]; /* and */ int foo()[] { ... } /* are discouraged */  
float[] array; /* and */ int[] foo() { ... } /* are encouraged */
```

The discouraged type is [meant to accommodate transitioning C users](#), who are familiar with the syntax for C which has the brackets after the variable name.

In Java, it is possible to have arrays of size 0:

```
int[] array = new int[0]; // Compiles and runs fine.  
int[] array2 = {};// Equivalent syntax.
```

However, since it's an empty array, no elements can be read from it or assigned to it:

```
array[0] = 1; // Throws java.lang.ArrayIndexOutOfBoundsException.  
int i = array2[0]; // Also throws ArrayIndexOutOfBoundsException.
```

Such empty arrays are typically useful as return values, so that the calling code only has to worry about dealing with an array, rather than a potential `null` value that may lead to a `NullPointerException`.

The length of an array must be a non-negative integer:

```
int[] array = new int[-1]; // Throws java.lang.NegativeArraySizeException
```

数组大小可以通过一个名为length的公共最终字段来确定：

```
System.out.println(array.length); // 在此情况下打印 0。
```

注意：array.length 返回的是数组的实际大小，而不是被赋值的数组元素数量，这与ArrayList.size()不同，后者返回的是被赋值的数组元素数量。

创建和初始化多维数组

创建多维数组的最简单方法如下：

```
int[][] a = new int[2][3];
```

它将创建两个长度为三的int数组——a[0]和a[1]。这与经典的C风格矩形多维数组初始化非常相似。

你可以同时创建和初始化：

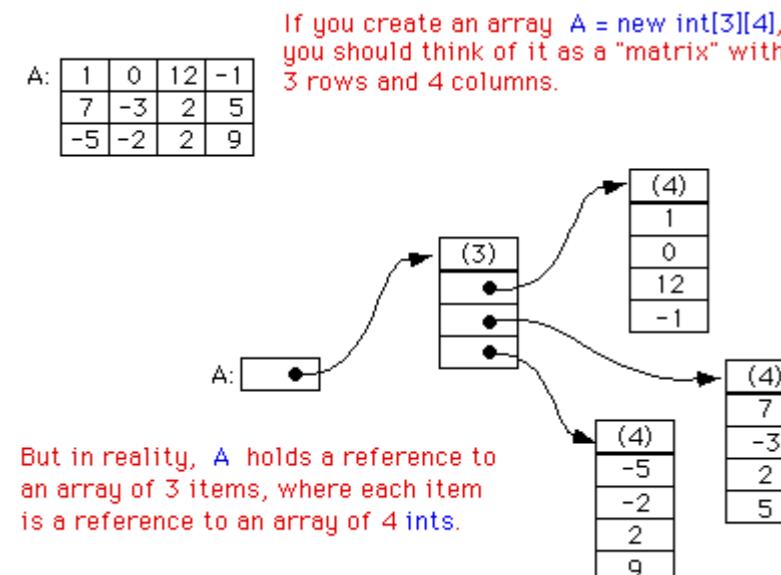
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

与C语言不同，C语言只支持矩形多维数组，内部数组不需要具有相同长度，甚至可以未定义：

```
int[][] a = { {1}, {2, 3}, null };
```

这里，a[0]是一个长度为一的int数组，而a[1]是一个长度为二的int数组，a[2]是null。这样的数组称为锯齿状数组或参差数组，即它们是数组的数组。Java中的多维数组是作为数组的数组实现的，即array[i][j][k]等价于((array[i])[j])[k]。与C#不同，Java不支持语法array[i,j]。

Java中的多维数组表示



来源 - 在Ideone上实时运行

创建和初始化引用类型数组

```
String[] array6 = new String[] { "Laurel", "Hardy" }; // 使用new操作符和数组初始化器创建数组。
```

```
String[] array7 = { "劳雷尔", "哈代" }; // 使用数组的快捷语法初始化
```

The array size can be determined using a public final field called length:

```
System.out.println(array.length); // Prints 0 in this case.
```

Note: array.length returns the actual size of the array and not the number of array elements which were assigned a value, unlike `ArrayList.size()` which returns the number of array elements which were assigned a value.

Creating and initializing multi-dimensional arrays

The simplest way to create a multi-dimensional array is as follows:

```
int[][] a = new int[2][3];
```

It will create two three-length int arrays—a[0] and a[1]. This is very similar to the classical, C-style initialization of rectangular multi-dimensional arrays.

You can create and initialize at the same time:

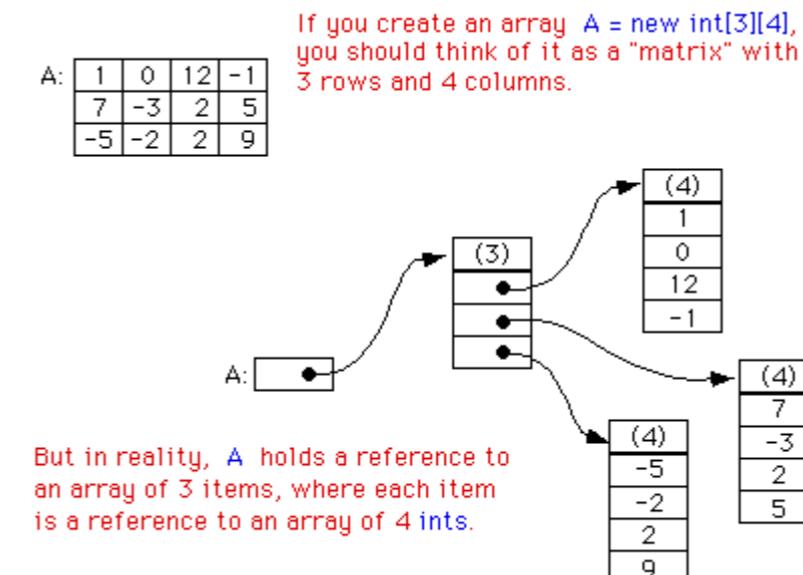
```
int[][] a = { {1, 2}, {3, 4}, {5, 6} };
```

Unlike C, where only rectangular multi-dimensional arrays are supported, inner arrays do not need to be of the same length, or even defined:

```
int[][] a = { {1}, {2, 3}, null };
```

Here, a[0] is a one-length int array, whereas a[1] is a two-length int array and a[2] is null. Arrays like this are called jagged arrays or ragged arrays, that is, they are arrays of arrays. Multi-dimensional arrays in Java are implemented as arrays of arrays, i.e. `array[i][j][k]` is equivalent to `((array[i])[j])[k]`. Unlike C#, the syntax `array[i, j]` is not supported in Java.

Multidimensional array representation in Java



Source - Live on Ideone

Creating and initializing reference type arrays

```
String[] array6 = new String[] { "Laurel", "Hardy" }; // Create an array with new operator and array initializer.  
String[] array7 = { "Laurel", "Hardy" }; // Shortcut syntax with array
```

```
// 初始化器。  
String[] array8 = new String[3];  
// { null, null, null }  
String[] array9 = null;  
// null
```

[在 Ideone 上运行](#)

除了上面展示的String字面量和基本类型外，数组初始化的快捷语法也适用于标准的Object类型：

```
Object[] array10 = { new Object(), new Object() };
```

由于数组是协变的，引用类型数组可以被初始化为子类数组，尽管如果尝试将元素设置为非String类型时，会抛出ArrayStoreException异常：

```
Object[] array11 = new String[] { "foo", "bar", "baz" };  
array11[1] = "qux"; // 正常  
array11[1] = new StringBuilder(); // 抛出 ArrayStoreException
```

快捷语法不能用于此，因为快捷语法会隐式地具有Object[]类型。

可以使用String[] emptyArray = new String[0]来初始化一个零元素的数组。例如，像这样的零长度数组用于从Collection创建Array时，当方法需要对象的运行时类型。

在基本类型和引用类型中，空数组初始化（例如String[] array8 = new String[3]）将使用每种数据类型的默认值初始化数组。

创建和初始化泛型类型数组

在泛型类中，由于类型擦除，泛型类型的数组不能像这样初始化：

```
public class MyGenericClass<T> {  
    private T[] a;  
  
    public MyGenericClass() {  
        a = new T[5]; // 编译时错误：泛型数组创建  
    }  
}
```

相反，可以使用以下方法之一创建它们：（注意这些会产生未经检查的警告）

1. 通过创建一个Object数组，并将其强制转换为泛型类型：

```
a = (T[]) new Object[5];
```

这是最简单的方法，但由于底层数组仍然是 Object[] 类型，该方法不提供类型安全。因此，这种创建数组的方法最好仅在泛型类内部使用——不对外公开。

2. 通过使用 `Array.newInstance` 并传入类参数：

```
public MyGenericClass(Class<T> clazz) {  
    a = (T[]) Array.newInstance(clazz, 5);
```

```
// initializer.  
String[] array8 = new String[3];  
String[] array9 = null;  
// null
```

[Live on Ideone](#)

In addition to the `String` literals and primitives shown above, the shortcut syntax for array initialization also works with canonical `Object` types:

```
Object[] array10 = { new Object(), new Object() };
```

Because arrays are covariant, a reference type array can be initialized as an array of a subclass, although an `ArrayStoreException` will be thrown if you try to set an element to something other than a `String`:

```
Object[] array11 = new String[] { "foo", "bar", "baz" };  
array11[1] = "qux"; // fine  
array11[1] = new StringBuilder(); // throws ArrayStoreException
```

The shortcut syntax cannot be used for this because the shortcut syntax would have an implicit type of `Object[]`.

An array can be initialized with zero elements by using `String[] emptyArray = new String[0]`. For example, an array with zero length like this is used for Creating an `Array` from a `Collection` when the method needs the runtime type of an object.

In both primitive and reference types, an empty array initialization (for example `String[] array8 = new String[3]`) will initialize the array with the default value for each data type.

Creating and initializing generic type arrays

In generic classes, arrays of generic types **cannot** be initialized like this due to type erasure:

```
public class MyGenericClass<T> {  
    private T[] a;  
  
    public MyGenericClass() {  
        a = new T[5]; // Compile time error: generic array creation  
    }  
}
```

Instead, they can be created using one of the following methods: (note that these will generate unchecked warnings)

1. By creating an `Object` array, and casting it to the generic type:

```
a = (T[]) new Object[5];
```

This is the simplest method, but since the underlying array is still of type `Object[]`, this method does not provide type safety. Therefore, this method of creating an array is best used only within the generic class - not exposed publicly.

2. By using `Array.newInstance` with a class parameter:

```
public MyGenericClass(Class<T> clazz) {  
    a = (T[]) Array.newInstance(clazz, 5);
```

```
}
```

这里必须显式将 T 的类传递给构造函数。 `Array.newInstance` 的返回类型始终是 `Object`。但该方法更安全，因为新创建的数组始终是 `T[]` 类型，因此可以安全地对外暴露。

```
}
```

Here the class of T has to be explicitly passed to the constructor. The return type of `Array.newInstance` is always `Object`. However, this method is safer because the newly created array is always of type `T[]`, and therefore can be safely externalized.

初始化后填充数组

版本 ≥ Java SE 1.2

可以使用 `Arrays.fill()` 在初始化后用相同的值填充数组：

```
Arrays.fill(array8, "abc"); // { "abc", "abc", "abc" }
```

[在 Ideone 上运行](#)

`fill()` 也可以为数组指定范围内的每个元素赋值：

```
Arrays.fill(array8, 1, 2, "aaa"); // 从索引1到2填充"aaa"。
```

[在 Ideone 上运行](#)

版本 ≥ Java SE 8

自Java 8版本起，方法`setAll`及其`Concurrent`等效方法`parallelSetAll`可用于将数组的每个元素设置为生成的值。这些方法接收一个生成器函数，该函数接受一个索引并返回该位置所需的值。

以下示例创建一个整数数组，并将其所有元素设置为各自的索引值：

```
int[] array = new int[5];
Arrays.setAll(array, i -> i); // 数组变为 { 0, 1, 2, 3, 4 }.
```

[在 Ideone 上运行](#)

数组的声明和初始化分开

数组元素的索引值必须是整数（0, 1, 2, 3, 4, ...），且小于数组的长度（索引从零开始）。否则，将抛出`ArrayIndexOutOfBoundsException`异常：

```
int[] array9; // 数组声明 - 未初始化
array9 = new int[3]; // 初始化数组 - { 0, 0, 0 }
array9[0] = 10; // 设置索引0的值 - { 10, 0, 0 }
array9[1] = 20; // 设置索引1的值 - { 10, 20, 0 }
array9[2] = 30; // 设置索引2的值 - { 10, 20, 30 }
```

数组不能用数组初始化快捷语法重新初始化

由于数组初始化器只能在字段声明、本地变量声明或作为数组创建表达式的一部分中指定，因此不能通过数组初始化快捷语法重新初始化数组。

但是，可以创建一个新数组并将其赋值给用于引用旧数组的变量。

虽然这会导致该变量引用的数组被重新初始化，但变量内容是一个全新的数组。为此，可以使用`new`操作符配合数组初始化器，并赋值给数组变量：

```
// 数组的第一次初始化
int[] array = new int[] { 1, 2, 3 };
```

Filling an array after initialization

Version ≥ Java SE 1.2

`Arrays.fill()` 可以用于在初始化后用 **相同值** 填充数组：

```
Arrays.fill(array8, "abc"); // { "abc", "abc", "abc" }
```

[Live on Ideone](#)

`fill()` 也可以为数组指定范围内的每个元素赋值：

```
Arrays.fill(array8, 1, 2, "aaa"); // Placing "aaa" from index 1 to 2.
```

[Live on Ideone](#)

Version ≥ Java SE 8

Since Java version 8, the method `setAll`, and its Concurrent equivalent `parallelSetAll`, can be used to set every element of an array to generated values. These methods are passed a generator function which accepts an index and returns the desired value for that position.

The following example creates an integer array and sets all of its elements to their respective index value:

```
int[] array = new int[5];
Arrays.setAll(array, i -> i); // The array becomes { 0, 1, 2, 3, 4 }.
```

[Live on Ideone](#)

Separate declaration and initialization of arrays

The value of an index for an array element must be a whole number (0, 1, 2, 3, 4, ...) and less than the length of the array (indexes are zero-based). Otherwise, an `ArrayIndexOutOfBoundsException` will be thrown:

```
int[] array9; // Array declaration - uninitialized
array9 = new int[3]; // Initialize array - { 0, 0, 0 }
array9[0] = 10; // Set index 0 value - { 10, 0, 0 }
array9[1] = 20; // Set index 1 value - { 10, 20, 0 }
array9[2] = 30; // Set index 2 value - { 10, 20, 30 }
```

Arrays may not be re-initialized with array initializer shortcut syntax

It is not possible to re-initialize an array via a shortcut syntax with an array initializer since an array initializer can only be specified in a field declaration or local variable declaration, or as a part of an array creation expression.

However, it is possible to create a new array and assign it to the variable being used to reference the old array. While this results in the array referenced by that variable being re-initialized, the variable contents are a completely new array. To do this, the `new` operator can be used with an array initializer and assigned to the array variable:

```
// First initialization of array
int[] array = new int[] { 1, 2, 3 };
```

```
// 输出"1 2 3".
for (int i : array) {
    System.out.print(i + " ");
}

// 将数组重新初始化为一个新的 int[] 数组。
array = new int[] { 4, 5, 6 };

// 输出"4 5 6".
for (int i : array) {
    System.out.print(i + " ");
}

array = { 1, 2, 3, 4 }; // 编译时错误！不能通过快捷语法和数组初始化器重新初始化数组。
```

[在 Ideone 上运行](#)

第23.2节：从数组创建列表

`Arrays.asList()` 方法可用于返回包含给定数组元素的固定大小的 `List`。生成的 `List` 将与数组的基类型具有相同的参数类型。

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = Arrays.asList(stringArray);
```

注意：该列表由原始数组的（视图）支持，这意味着对列表的任何更改都会更改数组反之亦然。然而，任何会改变列表大小（从而改变数组长度）的更改都会抛出异常。

要创建列表的副本，请使用接受 `Collection` 作为参数的 `java.util.ArrayList` 构造函数：

```
版本 ≥ Java SE 5
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));
版本 ≥ Java SE 7
```

在 Java SE 7 及更高版本中，可以使用一对尖括号`<>`（空类型参数集），称为Diamond。编译器可以从上下文中确定类型参数。这意味着在调用 `ArrayList` 构造函数时可以省略类型信息，编译时会自动推断。这称为类型推断，是 Java 泛型的一部分。

```
// 使用 Arrays.asList()

String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<>(Arrays.asList(stringArray));
```

// 使用 `ArrayList.addAll()`

```
String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
list.addAll(Arrays.asList(stringArray));
```

// 使用 `Collections.addAll()`

```
String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, stringArray);
```

```
// Prints "1 2 3".
for (int i : array) {
    System.out.print(i + " ");
}

// Re-initializes array to a new int[] array.
array = new int[] { 4, 5, 6 };

// Prints "4 5 6".
for (int i : array) {
    System.out.print(i + " ");
}

array = { 1, 2, 3, 4 }; // Compile-time error! Can't re-initialize an array via shortcut
// syntax with array initializer.
```

[Live on Ideone](#)

Section 23.2: Creating a List from an Array

The `Arrays.asList()` method can be used to return a fixed-size `List` containing the elements of the given array. The resulting `List` will be of the same parameter type as the base type of the array.

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = Arrays.asList(stringArray);
```

Note: This list is backed by (*a view of*) the original array, meaning that any changes to the list will change the array and vice versa. However, changes to the list that would change its size (and hence the array length) will throw an exception.

To create a copy of the list, use the constructor of `java.util.ArrayList` taking a `Collection` as an argument:

```
Version ≥ Java SE 5
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<String>(Arrays.asList(stringArray));
Version ≥ Java SE 7
```

In Java SE 7 and later, a pair of angle brackets `<>` (empty set of type arguments) can be used, which is called the Diamond. The compiler can determine the type arguments from the context. This means the type information can be left out when calling the constructor of `ArrayList` and it will be inferred automatically during compilation. This is called `Type Inference` which is a part of Java Generics.

// Using `Arrays.asList()`

```
String[] stringArray = {"foo", "bar", "baz"};
List<String> stringList = new ArrayList<>(Arrays.asList(stringArray));
```

// Using `ArrayList.addAll()`

```
String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
list.addAll(Arrays.asList(stringArray));
```

// Using `Collections.addAll()`

```
String[] stringArray = {"foo", "bar", "baz"};
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list, stringArray);
```

关于菱形操作符值得注意的一点是，它不能与匿名类一起使用。

版本 ≥ Java SE 8

```
// 使用流 (Streams)

int[] ints = {1, 2, 3};
List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

String[] stringArray = {"foo", "bar", "baz"};
List<Object> list = Arrays.stream(stringArray).collect(Collectors.toList());
```

关于使用 `Arrays.asList()` 方法的重要注意事项

- 该方法返回List，它是Arrays的静态内部类ArrayList的一个实例，而不是java.util.ArrayList。返回的List是固定大小的。这意味着，不支持添加或删除元素，且会抛出UnsupportedOperationException异常：

```
stringList.add("something"); // 抛出 java.lang.UnsupportedOperationException
```

- 可以通过将基于数组的List传递给新List的构造函数来创建一个新的List。这会创建一个数据的新副本，该副本大小可变且不依赖于原始数组：

```
List<String> modifiableList = new ArrayList<>(Arrays.asList("foo", "bar"));
```

- 在原始数组（如int[]）上调用<T> List<T> asList(T... a)时，会生成一个List<int[]>，其唯一元素是源原始数组，而不是源数组的实际元素。

这种行为的原因是原始类型不能用作泛型类型参数，因此在这种情况下，整个原始数组替代了泛型类型参数。为了将原始数组转换为List，首先需要将原始数组转换为对应包装类型的数组（即调用
Arrays.asList 在Integer[]上，而不是在int[]上）。

因此，这将打印false：

```
int[] arr = {1, 2, 3};      // int的原始数组
System.out.println(Arrays.asList(arr).contains(1));
```

[查看演示](#)

另一方面，这将打印true：

```
Integer[] arr = {1, 2, 3}; // Integer对象数组 (int的包装类)
System.out.println(Arrays.asList(arr).contains(1));
```

[查看演示](#)

这也会打印true，因为数组将被解释为Integer[]:]

```
System.out.println(Arrays.asList(1,2,3).contains(1));
```

[查看演示](#)

A point worth noting about the Diamond is that it cannot be used with Anonymous Classes.

Version ≥ Java SE 8

```
// Using Streams

int[] ints = {1, 2, 3};
List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

String[] stringArray = {"foo", "bar", "baz"};
List<Object> list = Arrays.stream(stringArray).collect(Collectors.toList());
```

Important notes related to using `Arrays.asList()` method

- This method returns List, which is an instance of `Arrays$ArrayList`(static inner class of `Arrays`) and not `java.util.ArrayList`. The resulting List is of fixed-size. That means, adding or removing elements is not supported and will throw an `UnsupportedOperationException`:

```
stringList.add("something"); // throws java.lang.UnsupportedOperationException
```

- A new List can be created by passing an array-backed List to the constructor of a new List. This creates a new copy of the data, which has changeable size and that is not backed by the original array:

```
List<String> modifiableList = new ArrayList<>(Arrays.asList("foo", "bar"));
```

- Calling <T> List<T> asList(T... a) on a primitive array, such as an int[], will produce a List<int[]> whose only element is the source primitive array instead of the actual elements of the source array.

The reason for this behavior is that primitive types cannot be used in place of generic type parameters, so the entire primitive array replaces the generic type parameter in this case. In order to convert a primitive array to a List, first of all, convert the primitive array to an array of the corresponding wrapper type (i.e. call `Arrays.asList` on an Integer[] instead of an int[]).

Therefore, this will print false:

```
int[] arr = {1, 2, 3};      // primitive array of int
System.out.println(Arrays.asList(arr).contains(1));
```

[View Demo](#)

On the other hand, this will print true:

```
Integer[] arr = {1, 2, 3}; // object array of Integer (wrapper for int)
System.out.println(Arrays.asList(arr).contains(1));
```

[View Demo](#)

This will also print true, because the array will be interpreted as an Integer[]:]

```
System.out.println(Arrays.asList(1,2,3).contains(1));
```

[View Demo](#)

第23.3节：从集合创建数组

java.util.Collection中有两种方法可以从集合创建数组：

- [Object\[\] toArray\(\)](#)
- [<T> T\[\] toArray\(T\[\] a\)](#)

Object[] toArray()可以如下使用：

```
版本 ≥ Java SE 5
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// 虽然 set 是一个 Set<String>, 但 toArray() 返回的是 Object[] 而不是 String[]
Object[] objectArray = set.toArray();
```

<T> T[] toArray(T[] a) 可以按如下方式使用：

```
版本 ≥ Java SE 5
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// 不需要预先创建正确大小的数组。
// 只要数组类型正确即可。 (如果大小不对, 将会创建一个相同类型的新数组。)

String[] stringArray = set.toArray(new String[0]);

// 如果提供的数组大小与集合相同或更大, 数组将被填充集合的值并返回 (不会分配新数组)
)

String[] stringArray2 = set.toArray(new String[set.size()]);
```

它们之间的区别不仅仅是无类型与有类型结果的差异。它们的性能也可能不同（详情请阅读此性能分析部分）：

- Object[] toArray() 使用了矢量化的arraycopy，比T[] toArray(T[] a)中使用的类型检查arraycopy快得多。
- T[] toArray(new T[non-zero-size]) 需要在运行时将数组清零，而T[] toArray(new T[0])则不需要。避免清零使得后者调用比前者更快。详细分析见：[Arrays of Wisdom of the Ancients](#)。

版本 ≥ Java SE 8

从 Java SE 8+ 开始，引入了Stream的概念，可以使用集合产生的Stream来通过[Stream.toArray](#)方法创建一个新的数组。

```
String[] strings = list.stream().toArray(String[]::new);
```

示例取自 [Stack Overflow](#) 上关于将 [ArrayList](#) 转换为 [String\[\]](#) 的两个回答 (1, 2)。

第 23.4 节：多维数组和锯齿状数组

可以定义多维数组。多维数组不是通过提供单个索引访问，而是通过为每个维度指定一个索引来访问。

Section 23.3: Creating an Array from a Collection

Two methods in [java.util.Collection](#) create an array from a collection:

- [Object\[\] toArray\(\)](#)
- [<T> T\[\] toArray\(T\[\] a\)](#)

Object[] toArray() can be used as follows:

```
Version ≥ Java SE 5
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// although set is a Set<String>, toArray() returns an Object[] not a String[]
Object[] objectArray = set.toArray();
```

<T> T[] toArray(T[] a) can be used as follows:

```
Version ≥ Java SE 5
Set<String> set = new HashSet<String>();
set.add("red");
set.add("blue");

// The array does not need to be created up front with the correct size.
// Only the array type matters. (If the size is wrong, a new array will
// be created with the same type.)
String[] stringArray = set.toArray(new String[0]);

// If you supply an array of the same size as collection or bigger, it
// will be populated with collection values and returned (new array
// won't be allocated)
String[] stringArray2 = set.toArray(new String[set.size()]);
```

The difference between them is more than just having untyped vs typed results. Their performance can differ as well (for details please read this [performance analysis section](#)):

- Object[] toArray() uses vectorized arraycopy, which is much faster than the type-checked arraycopy used in T[] toArray(T[] a).
- T[] toArray(new T[non-zero-size]) needs to zero-out the array at runtime, while T[] toArray(new T[0]) does not. Such avoidance makes the latter call faster than the former. Detailed analysis here : [Arrays of Wisdom of the Ancients](#).

Version ≥ Java SE 8

Starting from Java SE 8+, where the concept of Stream has been introduced, it is possible to use the Stream produced by the collection in order to create a new Array using the [Stream.toArray](#) method.

```
String[] strings = list.stream().toArray(String[]::new);
```

Examples taken from two answers (1, 2) to [Converting 'ArrayList' to 'String\[\]'](#) in Java on Stack Overflow.

Section 23.4: Multidimensional and Jagged Arrays

It is possible to define an array with more than one dimension. Instead of being accessed by providing a single index, a multidimensional array is accessed by specifying an index for each dimension.

多维数组的声明可以通过在普通数组声明中为每个维度添加[]来完成。例如，要创建一个二维int数组，可以在声明中添加另一对方括号，如int[][]。三维数组 (int[][][]) 依此类推。

定义一个具有三行三列的二维数组：

```
int rows = 3;  
int columns = 3;  
int[][] table = new int[rows][columns];
```

可以使用此结构对数组进行索引并赋值。请注意，未赋值的元素是该数组类型的默认值，在本例中，int 类型的默认值为 0。

```
table[0][0] = 0;  
table[0][1] = 1;  
table[0][2] = 2;
```

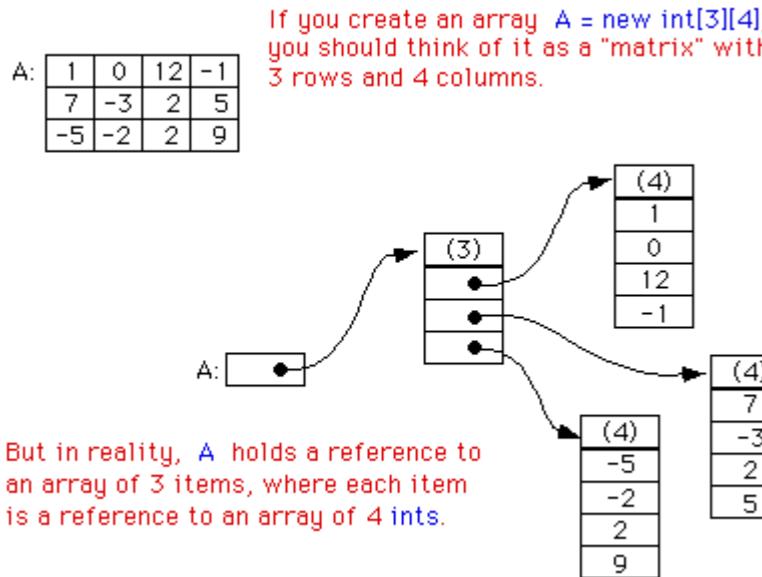
也可以一次实例化一个维度，甚至可以创建非矩形数组。这些通常被称为锯齿状数组。

```
int[][] nonRect = new int[4][];
```

需要注意的是，虽然可以定义锯齿状数组的任意维度，但其前面的维度**必须**先被定义。

```
// 有效  
String[][] employeeGraph = new String[30][];  
  
// 无效  
int[][] unshapenMatrix = new int[][10];  
  
// 也无效  
int[][][] misshapenGrid = new int[100][][10];
```

Java中多维数组的表示方式



图片来源：<http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html>

The declaration of multidimensional array can be done by adding [] for each dimension to a regular array declaration. For instance, to make a 2-dimensional int array, add another set of brackets to the declaration, such as int[][] . This continues for 3-dimensional arrays (int[][][]) and so forth.

To define a 2-dimensional array with three rows and three columns:

```
int rows = 3;  
int columns = 3;  
int[][] table = new int[rows][columns];
```

The array can be indexed and assign values to it with this construct. Note that the unassigned values are the default values for the type of an array, in this case 0 for int.

```
table[0][0] = 0;  
table[0][1] = 1;  
table[0][2] = 2;
```

It is also possible to instantiate a dimension at a time, and even make non-rectangular arrays. These are more commonly referred to as jagged arrays.

```
int[][] nonRect = new int[4][];
```

It is important to note that although it is possible to define any dimension of jagged array, its preceding level **must** be defined.

```
// valid  
String[][] employeeGraph = new String[30][];  
  
// invalid  
int[][] unshapenMatrix = new int[][10];  
  
// also invalid  
int[][][] misshapenGrid = new int[100][][10];
```

How Multidimensional Arrays are represented in Java

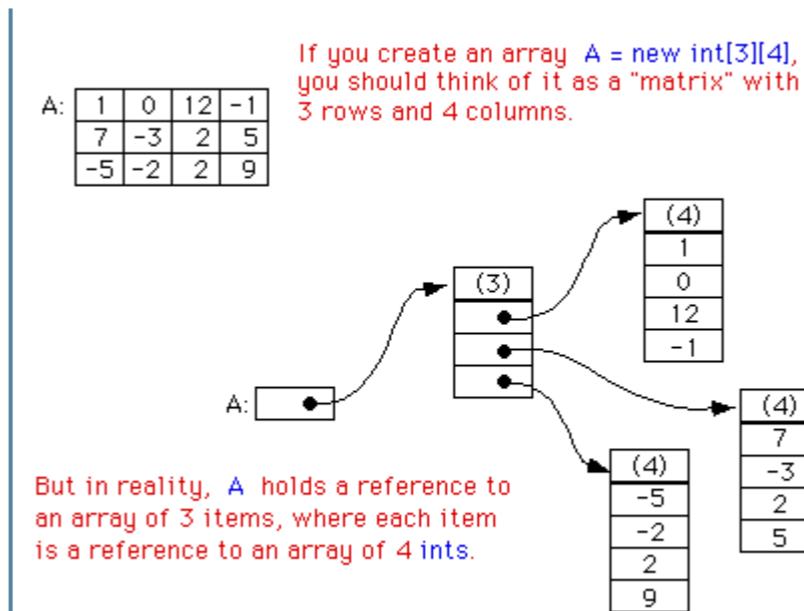


Image source: <http://math.hws.edu/eck/cs124/javanotes3/c8/s5.html>

锯齿状数组字面量初始化

多维数组和锯齿状数组也可以通过字面量表达式进行初始化。以下声明并填充了一个2x3的int数组：

```
int[][] table = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

注意：锯齿状子数组也可以是null。例如，以下代码声明并填充了一个二维int数组，其第一个子数组为null，第二个子数组长度为零，第三个子数组长度为一，最后一个子数组长度为二：

```
int[][] table = {  
    null,  
    {},  
    {1},  
    {1, 2}  
};
```

对于多维数组，可以通过索引提取低维度的数组：

```
int[][][] arr = new int[3][3][3];  
int[][] arr1 = arr[0]; // 从 arr 中获取第一个 3x3 维数组  
int[] arr2 = arr1[0]; // 从 arr1 中获取第一个 3 维数组  
int[] arr3 = arr[0]; // 错误：无法将 int[][] 转换为 int[]
```

第23.5节：数组索引越界异常 (ArrayIndexOutOfBoundsException)

当访问数组中不存在的索引时，会抛出ArrayIndexOutOfBoundsException异常。

数组是从零开始索引的，因此第一个元素的索引是0，最后一个元素的索引是数组容量减去1（即array.length - 1）。

因此，任何通过索引 i 访问数组元素的请求必须满足条件 `0 <= i < array.length`，否则将抛出ArrayIndexOutOfBoundsException异常。

下面的代码是一个简单示例，其中抛出了ArrayIndexOutOfBoundsException异常。

```
String[] people = new String[] { "Carol", "Andy" };  
  
// 创建了一个数组：  
// people[0]: "Carol"  
// people[1]: "Andy"  
  
// 注意：索引2处没有元素。尝试访问它会触发异常：  
System.out.println(people[2]); // 抛出 ArrayIndexOutOfBoundsException 异常。
```

输出：

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2  
at your.package.path.method(YourClass.java:15)
```

注意，访问的非法索引也包含在异常中（示例中的2）；这些信息可能

Jagged array literal initialization

Multidimensional arrays and jagged arrays can also be initialized with a literal expression. The following declares and populates a 2x3 int array:

```
int[][] table = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

Note: Jagged subarrays may also be null. For instance, the following code declares and populates a two dimensional int array whose first subarray is null, second subarray is of zero length, third subarray is of one length and the last subarray is a two length array:

```
int[][] table = {  
    null,  
    {},  
    {1},  
    {1, 2}  
};
```

For multidimensional array it is possible to extract arrays of lower-level dimension by their indices:

```
int[][][] arr = new int[3][3][3];  
int[][] arr1 = arr[0]; // get first 3x3-dimensional array from arr  
int[] arr2 = arr1[0]; // get first 3-dimensional array from arr1  
int[] arr3 = arr[0]; // error: cannot convert from int[][] to int[]
```

Section 23.5: ArrayIndexOutOfBoundsException

The [ArrayIndexOutOfBoundsException](#) is thrown when a non-existing index of an array is being accessed.

Arrays are zero-based indexed, so the index of the first element is 0 and the index of the last element is the array capacity minus 1 (i.e. `array.length - 1`).

Therefore, any request for an array element by the index i has to satisfy the condition `0 <= i < array.length`, otherwise the [ArrayIndexOutOfBoundsException](#) will be thrown.

The following code is a simple example where an [ArrayIndexOutOfBoundsException](#) is thrown.

```
String[] people = new String[] { "Carol", "Andy" };  
  
// An array will be created:  
// people[0]: "Carol"  
// people[1]: "Andy"  
  
// Notice: no item on index 2. Trying to access it triggers the exception:  
System.out.println(people[2]); // throws an ArrayIndexOutOfBoundsException.
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2  
at your.package.path.method(YourClass.java:15)
```

Note that the illegal index that is being accessed is also included in the exception (2 in the example); this information could

有助于查找异常的原因。

为避免这种情况，只需检查索引是否在数组的范围内：

```
int index = 2;
if (index >= 0 && index < people.length) {
    System.out.println(people[index]);
}
```

第23.6节：数组协变

对象数组是协变的，这意味着正如Integer是Number的子类，Integer[]是Number[]的子类。这看起来很直观，但可能导致令人惊讶的行为：

```
Integer[] integerArray = {1, 2, 3};
Number[] numberArray = integerArray; // 有效
Number firstElement = numberArray[0]; // 有效
numberArray[0] = 4L; // 运行时抛出 ArrayStoreException
```

虽然Integer[]是Number[]的子类，但它只能存放Integer，尝试赋值一个Long元素会抛出运行时异常。

注意，这种行为是数组特有的，可以通过使用泛型List来避免：

```
List<Integer> integerList = Arrays.asList(1, 2, 3);
//List<Number> numberList = integerList; // 编译错误
List<? extends Number> numberList = integerList;
Number firstElement = numberList.get(0);
//numberList.set(0, 4L); // 编译错误
```

数组中的所有元素不必共享相同类型，只要它们是数组类型的子类即可：

```
interface I {}

class A implements I {}
class B implements I {}
class C 实现 I {}

I[] array10 = new I[] { new A(), new B(), new C() }; // 使用 new
// 操作符和数组初始化器创建数组。

I[] array11 = { new A(), new B(), new C() }; // 使用数组初始化器的快捷语法。

I[] array12 = new I[3]; // { null, null, null }

I[] array13 = new A[] { new A(), new A() }; // 可行，因为 A 实现了 I。

Object[] array14 = new Object[] { "Hello, World!", 3.14159, 42 }; // 使用 new
// 操作符和数组初始化器创建数组。

Object[] array15 = { new A(), 64, "My String" }; // 使用数组初始化器的快捷语法。
```

be useful to find the cause of the exception.

To avoid this, simply check that the index is within the limits of the array:

```
int index = 2;
if (index >= 0 && index < people.length) {
    System.out.println(people[index]);
}
```

Section 23.6: Array Covariance

Object arrays are covariant, which means that just as Integer is a subclass of Number, Integer[] is a subclass of Number[]. This may seem intuitive, but can result in surprising behavior:

```
Integer[] integerArray = {1, 2, 3};
Number[] numberArray = integerArray; // valid
Number firstElement = numberArray[0]; // valid
numberArray[0] = 4L; // throws ArrayStoreException at runtime
```

Although Integer[] is a subclass of Number[], it can only hold Integers, and trying to assign a Long element throws a runtime exception.

Note that this behavior is unique to arrays, and can be avoided by using a generic List instead:

```
List<Integer> integerList = Arrays.asList(1, 2, 3);
//List<Number> numberList = integerList; // compile error
List<? extends Number> numberList = integerList;
Number firstElement = numberList.get(0);
//numberList.set(0, 4L); // compile error
```

It's not necessary for all of the array elements to share the same type, as long as they are a subclass of the array's type:

```
interface I {}

class A implements I {}
class B implements I {}
class C implements I {}

I[] array10 = new I[] { new A(), new B(), new C() }; // Create an array with new
// operator and array initializer.

I[] array11 = { new A(), new B(), new C() }; // Shortcut syntax with array
// initializer.

I[] array12 = new I[3]; // { null, null, null }

I[] array13 = new A[] { new A(), new A() }; // Works because A implements I.

Object[] array14 = new Object[] { "Hello, World!", 3.14159, 42 }; // Create an array with
// new operator and array initializer.

Object[] array15 = { new A(), 64, "My String" }; // Shortcut syntax
// with array initializer.
```

第23.7节：数组转流

版本 ≥ Java SE 8

将对象数组转换为Stream：

```
String[] arr = new String[] {"str1", "str2", "str3"};
Stream<String> stream = Arrays.stream(arr);
```

使用Arrays.stream()将原始类型数组转换为Stream时，会将数组转换为Stream的原始类型特化版本：

```
int[] intArr = {1, 2, 3};
IntStream intStream = Arrays.stream(intArr);
```

你也可以将Stream限制为数组中元素的一个范围。起始索引是包含的，结束索引是不包含的：

```
int[] values = {1, 2, 3, 4};
IntStream intStream = Arrays.stream(values, 2, 4);
```

一个类似于Arrays.stream()的方法出现在Stream类中：[Stream.of\(\)](#)。区别在于Stream.of()使用可变参数，因此你可以写成：

```
Stream<Integer> intStream = Stream.of(1, 2, 3);
Stream<String> stringStream = Stream.of("1", "2", "3");
Stream<Double> doubleStream = Stream.of(new Double[]{1.0, 2.0});
```

第23.8节：遍历数组

你可以通过使用增强型for循环（也称foreach）或者使用数组索引来遍历数组：

```
int[] array = new int[10];

// 使用索引：读写
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}
```

版本 ≥ Java SE 5

```
// 增强型for：只读
for (int e : array) {
    System.out.println(e);
}
```

这里值得注意的是，不能直接对数组使用Iterator，但通过Arrays库可以很容易地将数组转换为列表，从而获得一个Iterable对象。

对于装箱数组，使用Arrays.asList：

```
Integer[] 装箱数组 = {1, 2, 3};
Iterable<Integer> 装箱迭代器 = Arrays.asList(装箱数组); // 基于列表的可迭代对象
Iterator<Integer> 来自装箱1 = 装箱迭代器.iterator();
```

对于原始类型数组（使用java 8），使用流（本例中特指 Arrays.stream -> IntStream）：

```
int[] 原始数组 = {1, 2, 3};
```

Section 23.7: Arrays to Stream

Version ≥ Java SE 8

Converting an array of objects to Stream:

```
String[] arr = new String[] {"str1", "str2", "str3"};
Stream<String> stream = Arrays.stream(arr);
```

Converting an array of primitives to Stream using [Arrays.stream\(\)](#) will transform the array to a primitive specialization of Stream:

```
int[] intArr = {1, 2, 3};
IntStream intStream = Arrays.stream(intArr);
```

You can also limit the Stream to a range of elements in the array. The start index is inclusive and the end index is exclusive:

```
int[] values = {1, 2, 3, 4};
IntStream intStream = Arrays.stream(values, 2, 4);
```

A method similar to [Arrays.stream\(\)](#) appears in the Stream class: [Stream.of\(\)](#). The difference is that Stream.of() uses a varargs parameter, so you can write something like:

```
Stream<Integer> intStream = Stream.of(1, 2, 3);
Stream<String> stringStream = Stream.of("1", "2", "3");
Stream<Double> doubleStream = Stream.of(new Double[]{1.0, 2.0});
```

Section 23.8: Iterating over arrays

You can iterate over arrays either by using enhanced for loop (aka foreach) or by using array indices:

```
int[] array = new int[10];

// using indices: read and write
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}

Version ≥ Java SE 5

// extended for: read only
for (int e : array) {
    System.out.println(e);
}
```

It is worth noting here that there is no direct way to use an Iterator on an Array, but through the Arrays library it can be easily converted to a list to obtain an Iterable object.

For boxed arrays use [Arrays.asList](#):

```
Integer[] boxed = {1, 2, 3};
Iterable<Integer> boxedIt = Arrays.asList(boxed); // list-backed iterable
Iterator<Integer> fromBoxed1 = boxedIt.iterator();
```

For primitive arrays (using java 8) use streams (specifically in this example - [Arrays.stream -> IntStream](#)):

```
int[] primitives = {1, 2, 3};
```

```
IntStream 原始流 = Arrays.stream(原始数组); // 基于列表的可迭代对象  
PrimitiveIterator.OfInt 来自原始1 = 原始流.iterator();
```

如果不能使用流（没有 Java 8），可以选择使用谷歌的guava库：

```
Iterable<Integer> 来自原始2 = Ints.asList(原始数组);
```

在二维数组或更高维数组中，这两种技术都可以以稍微复杂的方式使用。

示例：

```
int[][] 数组 = new int[10][10];  
  
for (int 外层索引 = 0; 外层索引 < 数组.length; 外层索引++) {  
    for (int 内层索引 = 0; 内层索引 < 数组[外层索引].length; 内层索引++) {  
        数组[外层索引][内层索引] = 外层索引 + 内层索引;  
    }  
}  
  
版本 ≥ Java SE 5  
  
for (int[] 数组元素 : 数组) {  
    for (int 值 : 数组元素) {  
        System.out.println(值);  
    }  
}
```

不使用基于索引的循环，无法将数组设置为任何非统一的值。

当然，在使用索引迭代时，你也可以使用while或do-while循环。

一个注意事项：使用数组索引时，确保索引在0和数组.length - 1之间（两者均包含）。不要对数组长度做硬编码假设，否则如果数组长度改变而你的硬编码值不变，可能会导致代码出错。

示例：

```
int[] 数组 = {1, 2, 3, 4};  
  
public void 增加数字() {  
    // 这样做：  
    for (int i = 0; i < 数组.length; i++) {  
        数组[i] += 1; // 或者这样： 数组[i] = 数组[i] + 1; 或 数组[i]++;
    }  
  
    // 不要这样做：  
    for (int i = 0; i < 4; i++) {  
        numbers[i] += 1;
    }
}
```

最好不要使用复杂的计算来获取索引，而是使用索引进行迭代，如果需要不同的值，再进行计算。

示例：

```
public void fillArrayWithDoubleIndex(int[] array) {  
    // 这样做：  
    for (int i = 0; i < array.length; i++) {  
        array[i] = i * 2;
    }
}
```

```
IntStream primitiveStream = Arrays.stream(primitives); // list-backed iterable  
PrimitiveIterator.OfInt fromPrimitive1 = primitiveStream.iterator();
```

If you can't use streams (no java 8), you can choose to use google's [guava](#) library:

```
Iterable<Integer> fromPrimitive2 = Ints.asList(primitives);
```

In two-dimensional arrays or more, both techniques can be used in a slightly more complex fashion.

Example:

```
int[][] array = new int[10][10];  
  
for (int indexOuter = 0; indexOuter < array.length; indexOuter++) {  
    for (int indexInner = 0; indexInner < array[indexOuter].length; indexInner++) {  
        array[indexOuter][indexInner] = indexOuter + indexInner;
    }
}  
  
Version ≥ Java SE 5  
  
for (int[] numbers : array) {  
    for (int value : numbers) {  
        System.out.println(value);
    }
}
```

It is impossible to set an Array to any non-uniform value without using an index based loop.

Of course you can also use **while** or **do-while** loops when iterating using indices.

One note of caution: when using array indices, make sure the index is between 0 and `array.length - 1` (both inclusive). Don't make hard coded assumptions on the array length otherwise you might break your code if the array length changes but your hard coded values don't.

Example:

```
int[] numbers = {1, 2, 3, 4};  
  
public void incrementNumbers() {  
    // DO THIS :  
    for (int i = 0; i < numbers.length; i++) {  
        numbers[i] += 1; // or this: numbers[i] = numbers[i] + 1; or numbers[i]++;
    }  
  
    // DON'T DO THIS :  
    for (int i = 0; i < 4; i++) {  
        numbers[i] += 1;
    }
}
```

It's also best if you don't use fancy calculations to get the index but use the index to iterate and if you need different values calculate those.

Example:

```
public void fillArrayWithDoubleIndex(int[] array) {  
    // DO THIS :  
    for (int i = 0; i < array.length; i++) {  
        array[i] = i * 2;
    }
}
```

```

}

// 不要这样做：
int doubleLength = array.length * 2;
for (int i = 0; i < doubleLength; i += 2) {
    array[i / 2] = i;
}

```

反向访问数组

```

int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
    System.out.println(array[i]);
}

```

使用临时数组减少代码重复

遍历临时数组而不是重复代码可以使代码更简洁。它可用于对多个变量执行相同操作的情况。

```

// 我们想打印出所有这些
String name = "Margaret";
int eyeCount = 16;
double height = 50.2;
int legs = 9;
int arms = 5;

// 复制粘贴方法：
System.out.println(name);
System.out.println(eyeCount);
System.out.println(height);
System.out.println(legs);
System.out.println(arms);

// 临时数组方法：
for(Object attribute : new Object[]{name, eyeCount, height, legs, arms})
    System.out.println(attribute);

// 仅使用数字
for(double number : new double[]{eyeCount, legs, arms, height})
    System.out.println(Math.sqrt(number));

```

请记住，这段代码不应用于性能关键的部分，因为每次进入循环时都会创建一个数组，且基本类型变量会被复制到数组中，因此无法被修改。

第23.9节：数组转字符串

版本 ≥ Java SE 5

自Java 1.5起，你可以获得指定数组内容的String表示，而无需遍历其每个元素。只需使用`Arrays.toString(Object[])`或`Arrays.deepToString(Object[])`来处理多维数组：

```

int[] arr = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(arr)); // [1, 2, 3, 4, 5]

```

```

}

// DON'T DO THIS :
int doubleLength = array.length * 2;
for (int i = 0; i < doubleLength; i += 2) {
    array[i / 2] = i;
}

```

Accessing Arrays in reverse order

```

int[] array = {0, 1, 1, 2, 3, 5, 8, 13};
for (int i = array.length - 1; i >= 0; i--) {
    System.out.println(array[i]);
}

```

Using temporary Arrays to reduce code repetition

Iterating over a temporary array instead of repeating code can make your code cleaner. It can be used where the same operation is performed on multiple variables.

```

// we want to print out all of these
String name = "Margaret";
int eyeCount = 16;
double height = 50.2;
int legs = 9;
int arms = 5;

```

```

// copy-paste approach:
System.out.println(name);
System.out.println(eyeCount);
System.out.println(height);
System.out.println(legs);
System.out.println(arms);

```

```

// temporary array approach:
for(Object attribute : new Object[]{name, eyeCount, height, legs, arms})
    System.out.println(attribute);

// using only numbers
for(double number : new double[]{eyeCount, legs, arms, height})
    System.out.println(Math.sqrt(number));

```

Keep in mind that this code should not be used in performance-critical sections, as an array is created every time the loop is entered, and that primitive variables will be copied into the array and thus cannot be modified.

Section 23.9: Arrays to a String

Version ≥ Java SE 5

Since Java 1.5 you can get a `String` representation of the contents of the specified array without iterating over its every element. Just use `Arrays.toString(Object[])` or `Arrays.deepToString(Object[])` for multidimensional arrays:

```

int[] arr = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(arr)); // [1, 2, 3, 4, 5]

```

```

int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(Arrays.deepToString(arr)); // [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```

`Arrays.toString()` 方法使用 `Object.toString()` 方法来生成数组中每个元素的 `String` 值，除了基本类型数组外，它可以用于所有类型的数组。例如：

```

public class 猫 { /* 隐式继承自 Object */
    @Override
    public String toString() {
        return "猫!";
    }
}

猫[] 数组 = { new 猫(), new 猫() };
System.out.println(Arrays.toString(数组)); // [猫!, 猫!]

```

如果类没有重写 `toString()`，则会使用继承自 `Object` 的 `toString()`。通常输出不是很有用，例如：

```

public class 狗 {
    /* 隐式继承自 Object */
}

狗[] 数组 = { new 狗() };
System.out.println(Arrays.toString(数组)); // [Dog@17ed40e0]

```

第23.10节：数组排序

可以使用 `Arrays API` 轻松完成数组排序。

```

import java.util.Arrays;

// 创建一个整数数组
int[] array = {7, 4, 2, 1, 19};
// 这是排序部分，只需一个函数即可使用
Arrays.sort(array);
// 输出 [1, 2, 4, 7, 19]
System.out.println(Arrays.toString(array));

```

排序字符串数组：

字符串不是数值数据，它定义了自己的顺序，称为字典序，也称为字母顺序

当你使用 `sort()` 方法对字符串数组进行排序时，它会按照

`Comparable` 接口定义的自然顺序进行排序，如下所示：

升序

```

String[] names = {"John", "Steve", "Shane", "Adam", "Ben"};
System.out.println("排序前的字符串数组：" + Arrays.toString(names));
Arrays.sort(names);
System.out.println("升序排序后的字符串数组：" + Arrays.toString(names));

```

输出：

```

int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println(Arrays.deepToString(arr)); // [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

```

`Arrays.toString()` 方法使用 `Object.toString()` 方法来生产数组中每个元素的 `String` 值，除了基本类型数组外，它可以用于所有类型的数组。例如：

```

public class Cat { /* implicitly extends Object */
    @Override
    public String toString() {
        return "CAT!";
    }
}

Cat[] arr = { new Cat(), new Cat() };
System.out.println(Arrays.toString(arr)); // [CAT!, CAT!]

```

If no overridden `toString()` exists for the class, then the inherited `toString()` from `Object` will be used. Usually the output is then not very useful, for example:

```

public class Dog {
    /* implicitly extends Object */
}

Dog[] arr = { new Dog() };
System.out.println(Arrays.toString(arr)); // [Dog@17ed40e0]

```

Section 23.10: Sorting arrays

Sorting arrays can be easily done with the [Arrays api](#).

```

import java.util.Arrays;

// creating an array with integers
int[] array = {7, 4, 2, 1, 19};
// this is the sorting part just one function ready to be used
Arrays.sort(array);
// prints [1, 2, 4, 7, 19]
System.out.println(Arrays.toString(array));

```

Sorting String arrays:

`String` is not a numeric data, it defines its own order which is called lexicographic order, also known as alphabetic order. When you sort an array of `String` using `sort()` method, it sorts array into natural order defined by `Comparable` interface, as shown below :

Increasing Order

```

String[] names = {"John", "Steve", "Shane", "Adam", "Ben"};
System.out.println("String array before sorting：" + Arrays.toString(names));
Arrays.sort(names);
System.out.println("String array after sorting in ascending order：" + Arrays.toString(names));

```

Output:

排序前的字符串数组：[约翰, 史蒂夫, 谢恩, 亚当, 本]
排序后升序的字符串数组：[亚当, 本, 约翰, 谢恩, 史蒂夫]

递减顺序

```
Arrays.sort(names, 0, names.length, Collections.reverseOrder());  
System.out.println("字符串数组按降序排序后：" + Arrays.toString(names));
```

输出：

降序排序后的字符串数组：[Steve, Shane, John, Ben, Adam]

对对象数组进行排序

为了对对象数组进行排序，所有元素必须实现 Comparable 或 Comparator 接口来定义排序的顺序。

我们可以使用 sort(Object[]) 方法对对象数组进行自然顺序排序，但必须确保数组中的所有元素都实现了 Comparable 接口。

此外，它们还必须相互可比较，例如，对于数组中的任意元素 e1 和 e2，e1.compareTo(e2) 不应抛出 ClassCastException。或者，您可以使用如下示例中所示的 sort(T[], Comparator) 方法按自定义顺序对 Object 数组进行排序。

// 如何使用 Comparator 和 Comparable 对象数组进行排序

```
Course[] courses = new Course[4];  
courses[0] = new Course(101, "Java", 200);  
courses[1] = new Course(201, "Ruby", 300);  
courses[2] = new Course(301, "Python", 400);  
courses[3] = new Course(401, "Scala", 500);
```

```
System.out.println("排序前的对象数组：" + Arrays.toString(courses));
```

```
Arrays.sort(courses);  
System.out.println("按自然顺序排序后的对象数组：" + Arrays.toString(courses));
```

```
Arrays.sort(courses, new Course.PriceComparator());  
System.out.println("按价格排序后的对象数组：" + Arrays.toString(courses));
```

```
Arrays.sort(courses, new Course.NameComparator());  
System.out.println("按名称排序后的对象数组：" + Arrays.toString(courses));
```

输出：

排序前的对象数组：[#101 Java@200, #201 Ruby@300, #301 Python@400, #401 Scala@500]
按自然顺序排序后的对象数组：[#101 Java@200, #201 Ruby@300, #301 Python@400, #401 Scala@500]
按价格排序后的对象数组：[#101 Java@200, #201 Ruby@300, #301 Python@400, #401 Scala@500]
按名称排序后的对象数组：[#101 Java@200, #301 Python@400, #201 Ruby@300, #401 Scala@500]

String array before sorting : [John, Steve, Shane, Adam, Ben]
String array after sorting in ascending order : [Adam, Ben, John, Shane, Steve]

Decreasing Order

```
Arrays.sort(names, 0, names.length, Collections.reverseOrder());  
System.out.println("String array after sorting in descending order : " + Arrays.toString(names));
```

Output:

String array after sorting in descending order : [Steve, Shane, John, Ben, Adam]

Sorting an Object array

In order to sort an object array, all elements must implement either Comparable or Comparator interface to define the order of the sorting.

We can use either sort(Object[]) method to sort an object array on its natural order, but you must ensure that all elements in the array must implement Comparable.

Furthermore, they must be mutually comparable as well, for example e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array. Alternatively you can sort an Object array on custom order using sort(T[], Comparator) method as shown in following example.

// How to Sort Object Array in Java using Comparator and Comparable

```
Course[] courses = new Course[4];  
courses[0] = new Course(101, "Java", 200);  
courses[1] = new Course(201, "Ruby", 300);  
courses[2] = new Course(301, "Python", 400);  
courses[3] = new Course(401, "Scala", 500);
```

```
System.out.println("Object array before sorting : " + Arrays.toString(courses));
```

```
Arrays.sort(courses);  
System.out.println("Object array after sorting in natural order : " + Arrays.toString(courses));
```

```
Arrays.sort(courses, new Course.PriceComparator());  
System.out.println("Object array after sorting by price : " + Arrays.toString(courses));
```

```
Arrays.sort(courses, new Course.NameComparator());  
System.out.println("Object array after sorting by name : " + Arrays.toString(courses));
```

Output:

Object array before sorting : [#101 Java@200, #201 Ruby@300, #301 Python@400, #401 Scala@500]
Object array after sorting in natural order : [#101 Java@200, #201 Ruby@300, #301 Python@400, #401 Scala@500]
Object array after sorting by price : [#101 Java@200, #201 Ruby@300, #301 Python@400, #401 Scala@500]
Object array after sorting by name : [#101 Java@200, #301 Python@400, #201 Ruby@300, #401 Scala@500]

第23.11节：获取数组长度

数组是对象，提供空间以存储指定类型的元素，最多可存储其大小的元素。数组创建后，其大小不可修改。

```
int[] arr1 = new int[0];
int[] arr2 = new int[2];
int[] arr3 = new int[]{1, 2, 3, 4};
int[] arr4 = {1, 2, 3, 4, 5, 6, 7};

int len1 = arr1.length; // 0
int len2 = arr2.length; // 2
int len3 = arr3.length; // 4
int len4 = arr4.length; // 7
```

数组中的length字段存储数组的大小。它是一个final字段，不能被修改。

这段代码展示了数组的length与数组中存储的对象数量之间的区别。

```
public static void main(String[] args) {
    Integer arr[] = new Integer[] {1,2,3,null,5,null,7,null,null,11,null,13};

    int arrayLength = arr.length;
    int nonEmptyElementsCount = 0;

    for (int i=0; i<arrayLength; i++) {
        Integer arrEl = arr[i];
        if (arrEl != null) {
            nonEmptyElementsCount++;
        }
    }

    System.out.println("数组 'arr' 的长度为 "+arrayLength+
        " 并且它包含 "+nonEmptyElementsCount+" 个非空值");
}
```

结果：

```
数组 'arr' 的长度为 13
并且它包含 7 个非空值
```

第23.12节：在数组中查找元素

有许多方法可以找到数组中某个值的位置。以下示例代码片段均假设该数组是以下之一：

```
String[] strings = new String[] { "A", "B", "C" };
int[] ints = new int[] { 1, 2, 3, 4 };
```

此外，每个都会将index或index2设置为所需元素的索引，若元素不存在则设置为-1。

使用`Arrays.binarySearch`（仅适用于已排序数组）

```
int index = Arrays.binarySearch(strings, "A");
int index2 = Arrays.binarySearch(ints, 1);
```

Section 23.11: Getting the Length of an Array

Arrays are objects which provide space to store up to its size of elements of specified type. An array's size can not be modified after the array is created.

```
int[] arr1 = new int[0];
int[] arr2 = new int[2];
int[] arr3 = new int[]{1, 2, 3, 4};
int[] arr4 = {1, 2, 3, 4, 5, 6, 7};

int len1 = arr1.length; // 0
int len2 = arr2.length; // 2
int len3 = arr3.length; // 4
int len4 = arr4.length; // 7
```

The length field in an array stores the size of an array. It is a `final` field and cannot be modified.

This code shows the difference between the length of an array and amount of objects an array stores.

```
public static void main(String[] args) {
    Integer arr[] = new Integer[] {1,2,3,null,5,null,7,null,null,11,null,13};

    int arrayLength = arr.length;
    int nonEmptyElementsCount = 0;

    for (int i=0; i<arrayLength; i++) {
        Integer arrEl = arr[i];
        if (arrEl != null) {
            nonEmptyElementsCount++;
        }
    }

    System.out.println("Array 'arr' has a length of "+arrayLength+"\n"
        + "and it contains "+nonEmptyElementsCount+" non-empty values");
}
```

Result:

```
Array 'arr' has a length of 13
and it contains 7 non-empty values
```

Section 23.12: Finding an element in an array

There are many ways find the location of a value in an array. The following example snippets all assume that the array is one of the following:

```
String[] strings = new String[] { "A", "B", "C" };
int[] ints = new int[] { 1, 2, 3, 4 };
```

In addition, each one sets index or index2 to either the index of required element, or -1 if the element is not present.

Using `Arrays.binarySearch` (for sorted arrays only)

```
int index = Arrays.binarySearch(strings, "A");
int index2 = Arrays.binarySearch(ints, 1);
```

使用Arrays.asList（仅适用于非原始类型数组）

```
int index = Arrays.asList(strings).indexOf("A");
int index2 = Arrays.asList(ints).indexOf(1); // 编译错误
```

使用Stream

版本 ≥ Java SE 8

```
int index = IntStream.range(0, strings.length)
    .filter(i -> "A".equals(strings[i]))
    .findFirst()
.orElse(-1); // 如果不存在，返回-1。
// 原始类型数组类似
```

使用循环的线性搜索

```
int index = -1;
for (int i = 0; i < array.length; i++) {
    if ("A".equals(array[i])) {
        index = i;
        break;
    }
}
// 原始类型数组类似
```

使用第三方库如org.apache.commons进行线性搜索

```
int index = org.apache.commons.lang3.ArrayUtils.contains(strings, "A");
int index2 = org.apache.commons.lang3.ArrayUtils.contains(ints, 1);
```

注意：直接使用线性搜索比封装成列表更高效。

测试数组是否包含某个元素

上面的示例可以通过简单地测试计算出的索引是否大于或等于零来判断数组是否包含某个元素。

另外，也有一些更简洁的变体：

```
boolean isPresent = Arrays.asList(strings).contains("A");
版本 ≥ Java SE 8
boolean isPresent = Stream<String>.of(strings).anyMatch(x -> "A".equals(x));
boolean isPresent = false;
for (String s : strings) {
    if ("A".equals(s)) {
        isPresent = true;
        break;
    }
}
boolean isPresent = org.apache.commons.lang3.ArrayUtils.contains(ints, 4);
```

第23.13节：如何更改数组的大小？

简单的答案是你不能这样做。一旦数组被创建，它的大小就不能改变。相反，数组只能通过创建一个具有适当大小的新数组，并将元素从现有数组复制到新数组来“调整大小”。

```
String[] listOfCities = new String[3]; // 创建了大小为3的数组。
listOfCities[0] = "纽约";
```

Using a `Arrays.asList` (for non-primitive arrays only)

```
int index = Arrays.asList(strings).indexOf("A");
int index2 = Arrays.asList(ints).indexOf(1); // compilation error
```

Using a Stream

Version ≥ Java SE 8

```
int index = IntStream.range(0, strings.length)
    .filter(i -> "A".equals(strings[i]))
    .findFirst()
.orElse(-1); // If not present, gives us -1.
// Similar for an array of primitives
```

Linear search using a loop

```
int index = -1;
for (int i = 0; i < array.length; i++) {
    if ("A".equals(array[i])) {
        index = i;
        break;
    }
}
// Similar for an array of primitives
```

Linear search using 3rd-party libraries such as `org.apache.commons`

```
int index = org.apache.commons.lang3.ArrayUtils.contains(strings, "A");
int index2 = org.apache.commons.lang3.ArrayUtils.contains(ints, 1);
```

Note: Using a direct linear search is more efficient than wrapping in a list.

Testing if an array contains an element

The examples above can be adapted to test if the array contains an element by simply testing to see if the index computed is greater or equal to zero.

Alternatively, there are also some more concise variations:

```
boolean isPresent = Arrays.asList(strings).contains("A");
Version ≥ Java SE 8
boolean isPresent = Stream<String>.of(strings).anyMatch(x -> "A".equals(x));
boolean isPresent = false;
for (String s : strings) {
    if ("A".equals(s)) {
        isPresent = true;
        break;
    }
}
boolean isPresent = org.apache.commons.lang3.ArrayUtils.contains(ints, 4);
```

Section 23.13: How do you change the size of an array?

The simple answer is that you cannot do this. Once an array has been created, its size cannot be changed. Instead, an array can only be “resized” by creating a new array with the appropriate size and copying the elements from the existing array to the new one.

```
String[] listOfCities = new String[3]; // array created with size 3.
listOfCities[0] = "New York";
```

```
listOfCities[1] = "伦敦";
listOfCities[2] = "柏林";
```

假设（例如）需要向上述定义的listOfCities数组中添加一个新元素。为此，您需要：

1. 创建一个大小为4的新数组，
2. 将旧数组中现有的3个元素复制到新数组的偏移量0、1和2处，
3. 并将新元素添加到新数组的偏移量3处。

有多种方法可以实现上述操作。在 Java 6 之前，最简洁的方法是：

```
String[] newArray = new String[listOfCities.length + 1];
System.arraycopy(listOfCities, 0, newArray, 0, listOfCities.length);
newArray[listOfCities.length] = "Sydney";
```

从 Java 6 开始，`Arrays.copyOf` 和 `Arrays.copyOfRange` 方法可以更简单地实现这一点：

```
String[] newArray = Arrays.copyOf(listOfCities, listOfCities.length + 1);
newArray[listOfCities.length] = "Sydney";
```

有关其他复制数组的方法，请参阅以下示例。请记住，在调整大小时，您需要一个长度不同于原始数组的数组副本。

- 复制数组

调整数组大小的更好替代方案

上述调整数组大小有两个主要缺点：

- 效率低下。增大（或缩小）数组涉及复制许多或全部现有数组元素，并分配一个新的数组对象。数组越大，开销越大。
- 您需要能够更新包含对旧数组引用的任何“活动”变量。

一种选择是从一开始就创建一个足够大的数组。只有在你能够准确确定该大小在分配数组之前，这种方法才可行。如果你做不到这一点，那么调整数组大小的问题又会出现。

另一种选择是使用Java SE类库或第三方库提供的数据结构类。例如，Java SE的“collections”框架提供了多种List、Set和Map接口的实现，具有不同的运行时特性。ArrayList类的性能特性最接近普通数组（例如O(N) 查找，O(1) 获取和设置，O(N) 随机插入和删除），同时提供了更高效的调整大小方式，避免了引用更新的问题。

（ArrayList的调整大小效率来自其在每次调整时将底层数组大小加倍的策略。

对于典型的使用场景，这意味着你只需偶尔调整大小。当你将成本摊销到列表的整个生命周期时，每次插入的调整大小成本是O(1)。调整普通数组大小时，也可能采用相同的策略。）

第23.14节：在原始类型和装箱类型之间转换数组

有时需要将原始类型转换为装箱类型。

要转换数组，可以使用流（Java 8及以上版本）：

```
listOfCities[1] = "London";
listOfCities[2] = "Berlin";
```

Suppose (for example) that a new element needs to be added to the `listOfCities` array defined as above. To do this, you will need to:

1. create a new array with size 4,
2. copy the existing 3 elements of the old array to the new array at offsets 0, 1 and 2, and
3. add the new element to the new array at offset 3.

There are various ways to do the above. Prior to Java 6, the most concise way was:

```
String[] newArray = new String[listOfCities.length + 1];
System.arraycopy(listOfCities, 0, newArray, 0, listOfCities.length);
newArray[listOfCities.length] = "Sydney";
```

From Java 6 onwards, the `Arrays.copyOf` and `Arrays.copyOfRange` methods can do this more simply:

```
String[] newArray = Arrays.copyOf(listOfCities, listOfCities.length + 1);
newArray[listOfCities.length] = "Sydney";
```

For other ways to copy an array, refer to the following example. Bear in mind that you need an array copy with a different length to the original when resizing.

- Copying arrays

A better alternatives to array resizing

There two major drawbacks with resizing an array as described above:

- It is inefficient. Making an array bigger (or smaller) involves copying many or all of the existing array elements, and allocating a new array object. The larger the array, the more expensive it gets.
- You need to be able to update any “live” variables that contain references to the old array.

One alternative is to create the array with a large enough size to start with. This is only viable if you can determine that size accurately *before allocating the array*. If you cannot do that, then the problem of resizing the array arises again.

The other alternative is to use a data structure class provided by the Java SE class library or a third-party library. For example, the Java SE “collections” framework provides a number of implementations of the `List`, `Set` and `Map` APIs with different runtime properties. The `ArrayList` class is closest to performance characteristics of a plain array (e.g. O(N) lookup, O(1) get and set, O(N) random insertion and deletion) while providing more efficient resizing without the reference update problem.

(The resize efficiency for `ArrayList` comes from its strategy of doubling the size of the backing array on each resize. For a typical use-case, this means that you only resize occasionally. When you amortize over the lifetime of the list, the resize cost per insert is O(1). It may be possible to use the same strategy when resizing a plain array.)

Section 23.14: Converting arrays between primitives and boxed types

Sometimes conversion of `primitive` types to `boxed` types is necessary.

To convert the array, it's possible to use streams (in Java 8 and above):

版本 ≥ Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};  
Integer[] boxedArray =  
    Arrays.stream(primitiveArray).boxed().toArray(Integer[]::new);
```

在较低版本中，可以通过遍历原始数组并显式地将其复制到装箱数组来实现：

版本 < Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};  
Integer[] boxedArray = new Integer[primitiveArray.length];  
for (int i = 0; i < primitiveArray.length; ++i) {  
    boxedArray[i] = primitiveArray[i]; // 每个元素在这里自动装箱  
}
```

类似地，装箱数组可以转换为其对应的基本类型数组：

版本 ≥ Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};  
int[] primitiveArray =  
    Arrays.stream(boxedArray).mapToInt(Integer::intValue).toArray();
```

版本 < Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};  
int[] primitiveArray = new int[boxedArray.length];  
for (int i = 0; i < boxedArray.length; ++i) {  
    primitiveArray[i] = boxedArray[i]; // 每个元素在这里拆箱  
}
```

第23.15节：从数组中移除元素

Java的java.util.Arrays中没有提供直接从数组中移除元素的方法。要实现此操作，您可以将原数组复制到一个不包含要移除元素的新数组，或者将数组转换为允许移除操作的其他结构。

使用ArrayList

您可以将数组转换为java.util.List，移除元素后再将列表转换回数组，方法如下：

```
String[] 数组 = new String[]{"foo", "bar", "baz"};  
  
List<String> 列表 = new ArrayList<>(Arrays.asList(数组));  
列表.remove("foo");  
  
// 创建一个与列表大小相同的新数组，并将列表元素复制到其中。  
  
数组 = 列表.toArray(new String[列表.size()]);  
  
System.out.println(Arrays.toString(数组)); // [bar, baz]
```

使用 System.arraycopy

[System.arraycopy\(\)](#) 可以用来复制原始数组并移除你想要的元素。下面是一个示例：

```
int[] 数组 = new int[] { 1, 2, 3, 4 }; // 原始数组。  
int[] 结果 = new int[数组.length - 1]; // 用于存放结果的数组。  
int 索引 = 1; // 移除值 "2"。
```

Version ≥ Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};  
Integer[] boxedArray =  
    Arrays.stream(primitiveArray).boxed().toArray(Integer[]::new);
```

With lower versions it can be by iterating the primitive array and explicitly copying it to the boxed array:

Version < Java SE 8

```
int[] primitiveArray = {1, 2, 3, 4};  
Integer[] boxedArray = new Integer[primitiveArray.length];  
for (int i = 0; i < primitiveArray.length; ++i) {  
    boxedArray[i] = primitiveArray[i]; // Each element is autoboxed here  
}
```

Similarly, a boxed array can be converted to an array of its primitive counterpart:

Version ≥ Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};  
int[] primitiveArray =  
    Arrays.stream(boxedArray).mapToInt(Integer::intValue).toArray();
```

Version < Java SE 8

```
Integer[] boxedArray = {1, 2, 3, 4};  
int[] primitiveArray = new int[boxedArray.length];  
for (int i = 0; i < boxedArray.length; ++i) {  
    primitiveArray[i] = boxedArray[i]; // Each element is outboxed here  
}
```

Section 23.15: Remove an element from an array

Java doesn't provide a direct method in [java.util.Arrays](#) to remove an element from an array. To perform it, you can either copy the original array to a new one without the element to remove or convert your array to another structure allowing the removal.

Using ArrayList

You can convert the array to a [java.util.List](#), remove the element and convert the list back to an array as follows:

```
String[] array = new String[]{"foo", "bar", "baz"};  
  
List<String> list = new ArrayList<>(Arrays.asList(array));  
list.remove("foo");  
  
// Creates a new array with the same size as the list and copies the list  
// elements to it.  
array = list.toArray(new String[list.size()]);  
  
System.out.println(Arrays.toString(array)); // [bar, baz]
```

Using System.arraycopy

[System.arraycopy\(\)](#) can be used to make a copy of the original array and remove the element you want. Below an example:

```
int[] array = new int[] { 1, 2, 3, 4 }; // Original array.  
int[] result = new int[array.length - 1]; // Array which will contain the result.  
int index = 1; // Remove the value "2".
```

```
// 复制索引左侧的元素。
System.arraycopy(数组, 0, 结果, 0, 索引);
// 复制索引右侧的元素。
System.arraycopy(array, index + 1, result, index, array.length - index - 1);

System.out.println(Arrays.toString(result)); // [1, 3, 4]
```

使用 Apache Commons Lang

为了轻松移除元素，你可以使用Apache Commons Lang库，特别是静态方法类ArrayUtils的removeElement()。下面是一个示例：

```
int[] array = new int[]{1,2,3,4};
array = ArrayUtils.removeElement(array, 2); // 移除第一个出现的2
System.out.println(Arrays.toString(array)); // [1, 3, 4]
```

第23.16节：比较数组是否相等

数组类型继承自 java.lang.Object 的equals() (和 hashCode()) 实现，因此equals()只会在比较完全相同的数组对象时返回true。要基于数组的值比较数组是否相等，请使用java.util.Arrays.equals，该方法对所有数组类型都进行了重载。

```
int[] a = new int[]{1, 2, 3};
int[] b = new int[]{1, 2, 3};
System.out.println(a.equals(b)); // 打印“false”，因为a和b引用不同的对象
System.out.println(Arrays.equals(a, b)); // 打印“true”，因为a和b的元素具有相同的值
```

当元素类型是引用类型时，Arrays.equals() 会调用数组元素的 equals() 来判断不等性。特别地，如果元素类型本身是数组类型，则会使用身份比较。要比较多维数组的相等性，请改用下面的 Arrays.deepEquals()：

```
int a[] = { 1, 2, 3 };
int b[] = { 1, 2, 3 };

Object[] aObject = { a }; // aObject 包含一个元素
Object[] bObject = { b }; // bObject 包含一个元素

System.out.println(Arrays.equals(aObject, bObject)); // false
System.out.println(Arrays.deepEquals(aObject, bObject)); // true
```

因为集合和映射使用 equals() 和 hashCode()，数组通常不适合作为集合元素或映射键。要么将它们包装在一个辅助类中，该类基于数组元素实现 equals() 和 hashCode()，要么将它们转换为 List 实例并存储这些列表。

第23.17节：复制数组

Java 提供了多种复制数组的方法。

for 循环

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

```
// Copy the elements at the left of the index.
System.arraycopy(array, 0, result, 0, index);
// Copy the elements at the right of the index.
System.arraycopy(array, index + 1, result, index, array.length - index - 1);

System.out.println(Arrays.toString(result)); // [1, 3, 4]
```

Using Apache Commons Lang

To easily remove an element, you can use the [Apache Commons Lang](#) library and especially the static method [removeElement\(\)](#) of the class [ArrayUtils](#). Below an example:

```
int[] array = new int[]{1, 2, 3, 4};
array = ArrayUtils.removeElement(array, 2); // remove first occurrence of 2
System.out.println(Arrays.toString(array)); // [1, 3, 4]
```

Section 23.16: Comparing arrays for equality

Array types inherit their equals() (and hashCode()) implementations from java.lang.Object, so equals() will only return true when comparing against the exact same array object. To compare arrays for equality based on their values, use [java.util.Arrays.equals](#), which is overloaded for all array types.

```
int[] a = new int[]{1, 2, 3};
int[] b = new int[]{1, 2, 3};
System.out.println(a.equals(b)); // prints "false" because a and b refer to different objects
System.out.println(Arrays.equals(a, b)); // prints "true" because the elements of a and b have the same values
```

When the element type is a reference type, [Arrays.equals\(\)](#) calls equals() on the array elements to determine equality. In particular, if the element type is itself an array type, identity comparison will be used. To compare multidimensional arrays for equality, use [Arrays.deepEquals\(\)](#) instead as below:

```
int a[] = { 1, 2, 3 };
int b[] = { 1, 2, 3 };

Object[] aObject = { a }; // aObject contains one element
Object[] bObject = { b }; // bObject contains one element

System.out.println(Arrays.equals(aObject, bObject)); // false
System.out.println(Arrays.deepEquals(aObject, bObject)); // true
```

Because sets and maps use equals() and hashCode(), arrays are generally not useful as set elements or map keys. Either wrap them in a helper class that implements equals() and hashCode() in terms of the array elements, or convert them to [List](#) instances and store the lists.

Section 23.17: Copying arrays

Java provides several ways to copy an array.

for loop

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

请注意，使用此选项对对象数组而非原始数组进行操作时，复制的内容将是对原始内容的引用，而不是其副本。

[Object.clone\(\)](#)

由于数组在Java中是Object类型的，因此你可以使用[Object.clone\(\)](#)。

```
int[] a = { 4, 1, 3, 2 };
int[] b = a.clone(); // [4, 1, 3, 2]
```

请注意，数组的Object.clone方法执行的是浅拷贝，即它返回一个新数组的引用，该新数组引用与源数组相同的元素。

[Arrays.copyOf\(\)](#)

[java.util.Arrays](#)提供了一种简便的方法将一个数组复制到另一个数组。以下是基本用法：

```
int[] a = {4, 1, 3, 2};
int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]
```

请注意，[Arrays.copyOf](#)还提供了一个重载版本，允许你更改数组的类型：

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

[System.arraycopy\(\)](#)

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

从指定的源数组中，从指定位置开始，复制数组到目标数组的指定位置。

以下是使用示例

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
System.arraycopy(a, 0, b, 0, a.length); // [4, 1, 3, 2]
```

[Arrays.copyOfRange\(\)](#)

主要用于复制数组的一部分，也可以用来复制整个数组到另一个数组，如下所示：

```
int[] a = { 4, 1, 3, 2 };
int[] b = Arrays.copyOfRange(a, 0, a.length); // [4, 1, 3, 2]
```

第23.18节：数组类型转换

数组是对象，但它们的类型由所包含对象的类型决定。因此，不能直接将A[]转换为T[]，而必须将特定A[]中的每个A成员转换为T对象。通用示例：

```
public static <T, A> T[] castArray(T[] target, A[] array) {
    for (int i = 0; i < array.length; i++) {
        target[i] = (T) array[i];
```

Note that using this option with an Object array instead of primitive array will fill the copy with reference to the original content instead of copy of it.

[Object.clone\(\)](#)

Since arrays are [Objects](#) in Java, you can use [Object.clone\(\)](#).

```
int[] a = { 4, 1, 3, 2 };
int[] b = a.clone(); // [4, 1, 3, 2]
```

Note that the [Object.clone](#) method for an array performs a **shallow copy**, i.e. it returns a reference to a new array which references the **same** elements as the source array.

[Arrays.copyOf\(\)](#)

[java.util.Arrays](#) provides an easy way to perform the copy of an array to another. Here is the basic usage:

```
int[] a = {4, 1, 3, 2};
int[] b = Arrays.copyOf(a, a.length); // [4, 1, 3, 2]
```

Note that [Arrays.copyOf](#) also provides an overload which allows you to change the type of the array:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

[System.arraycopy\(\)](#)

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

Below an example of use

```
int[] a = { 4, 1, 3, 2 };
int[] b = new int[a.length];
System.arraycopy(a, 0, b, 0, a.length); // [4, 1, 3, 2]
```

[Arrays.copyOfRange\(\)](#)

Mainly used to copy a part of an Array, you can also use it to copy whole array to another as below:

```
int[] a = { 4, 1, 3, 2 };
int[] b = Arrays.copyOfRange(a, 0, a.length); // [4, 1, 3, 2]
```

Section 23.18: Casting Arrays

Arrays are objects, but their type is defined by the type of the contained objects. Therefore, one cannot just cast A[] to T[], but each A member of the specific A[] must be cast to a T object. Generic example:

```
public static <T, A> T[] castArray(T[] target, A[] array) {
    for (int i = 0; i < array.length; i++) {
        target[i] = (T) array[i];
```

```
    }
    return target;
}
```

因此，给定一个A[]数组：

```
T[] target = new T[array.Length];
target = castArray(target, array);
```

Java SE 提供了方法`Arrays.copyOf(original, newLength, newType)`来实现此目的：

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf(doubles, doubles.length, Number[].class);
```

```
    }
    return target;
}
```

Thus, given an A[] array:

```
T[] target = new T[array.Length];
target = castArray(target, array);
```

Java SE provides the method [`Arrays.copyOf\(original, newLength, newType\)`](#) for this purpose:

```
Double[] doubles = { 1.0, 2.0, 3.0 };
Number[] numbers = Arrays.copyOf\(doubles, doubles.length, Number\[\].class\);
```

第24章：集合

Java.util中的集合框架提供了许多通用的集合类，用于处理数据集合，具备常规数组无法提供的功能。

集合框架包含Collection<O>接口，主要的子接口有List<O>和Set<O>，以及映射集合Map<K,V>。Collections是根接口，许多其他集合框架都实现了它。

第24.1节：在循环中从列表中移除元素

在循环中从列表中移除元素比较棘手，这是因为列表的索引和长度会发生变化。

给定以下列表，下面是一些会产生意外结果的示例，以及一些会产生正确结果的示例。

```
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Strawberry");
```

错误示范

在for语句的迭代中移除元素跳过了"Banana"：

代码示例只会打印Apple和Strawberry。由于删除Apple后，Banana会移动到索引0，但同时i被递增到1，导致Banana被跳过。

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println(fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
        fruits.remove(i);
    }
}
```

在增强的for语句中删除元素会抛出异常：

因为在遍历集合的同时修改了它。

抛出异常：java.util.ConcurrentModificationException

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

正确的做法

使用Iterator在while循环中删除元素

```
Iterator<String> fruitIterator = fruits.iterator();
while(fruitIterator.hasNext()) {
    String fruit = fruitIterator.next();
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
```

Chapter 24: Collections

The collections framework in java.util provides a number of generic classes for sets of data with functionality that can't be provided by regular arrays.

Collections framework contains interfaces for Collection<O>, with main sub-interfaces List<O> and Set<O>, and mapping collection Map<K, V>. Collections are the root interface and are being implemented by many other collection frameworks.

Section 24.1: Removing items from a List within a loop

It is tricky to remove items from a list while within a loop, this is due to the fact that the index and length of the list gets changed.

Given the following list, here are some examples that will give an unexpected result and some that will give the correct result.

```
List<String> fruits = new ArrayList<String>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Strawberry");
```

INCORRECT

Removing in iteration of for statement Skips "Banana":

The code sample will only print Apple and Strawberry. Banana is skipped because it moves to index 0 once Apple is deleted, but at the same time i gets incremented to 1.

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println(fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
        fruits.remove(i);
    }
}
```

Removing in the enhanced for statement Throws Exception:

Because of iterating over collection and modifying it at the same time.

Throws: java.util.ConcurrentModificationException

```
for (String fruit : fruits) {
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
        fruits.remove(fruit);
    }
}
```

CORRECT

Removing in while loop using an Iterator

```
Iterator<String> fruitIterator = fruits.iterator();
while(fruitIterator.hasNext()) {
    String fruit = fruitIterator.next();
    System.out.println(fruit);
    if ("Apple".equals(fruit)) {
```

```
fruitIterator.remove();
}
}
```

Iterator接口内置了一个remove()方法，专门用于这种情况。然而，该方法在文档中被标记为“可选”，并且可能会抛出UnsupportedOperationException异常。

抛出：UnsupportedOperationException - 如果该迭代器不支持删除操作

因此，建议查看文档以确保支持此操作（实际上，除非集合是通过第三方库获得的不可变集合，或者使用了Collections.unmodifiable...()方法，否则该操作几乎总是被支持的）。

在使用Iterator时，如果List的modCount自Iterator创建以来发生了变化，会抛出ConcurrentModificationException异常。这种变化可能发生在同一线程，也可能发生在共享同一列表的多线程应用中。

modCount是一个int变量，用于计数该列表被结构性修改的次数。结构性修改本质上指对Collection对象调用了add()或remove()操作（通过Iterator进行的修改不计入）。当Iterator被创建时，它会存储当前的modCount值，并在每次迭代List时检查当前的modCount是否与创建时相同。如果modCount值发生变化，则抛出ConcurrentModificationException异常。

因此，对于上述声明的列表，下面的操作不会抛出任何异常：

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

但是，在初始化Iterator之后向List添加新元素会抛出ConcurrentModificationException：

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next()); //这里会抛出ConcurrentModificationException
}
```

反向迭代

```
for (int i = (fruits.size() - 1); i >= 0; i--) {
    System.out.println(fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
        fruits.remove(i);
    }
}
```

这样不会跳过任何元素。这种方法的缺点是输出顺序是反向的。不过，在大多数删除元素的场景中，这并不重要。你绝对不应该对LinkedList使用这种方法。

正向迭代，调整循环索引

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println(fruits.get(i));
```

```
fruitIterator.remove();
}
}
```

The Iterator interface has a remove() method built in just for this case. However, this method is marked as “optional” in the documentation, and it might throw an UnsupportedOperationException.

Throws: UnsupportedOperationException - if the remove operation is not supported by this iterator

Therefore, it is advisable to check the documentation to make sure this operation is supported (in practice, unless the collection is an immutable one obtained through a 3rd party library or the use of one of the Collections.unmodifiable...() method, the operation is almost always supported).

While using an Iterator a ConcurrentModificationException is thrown when the modCount of the List is changed from when the Iterator was created. This could have happened in the same thread or in a multi-threaded application sharing the same list.

A modCount is an int variable which counts the number of times this list has been structurally modified. A structural change essentially means an add() or remove() operation being invoked on Collection object (changes made by Iterator are not counted). When the Iterator is created, it stores this modCount and on every iteration of the List checks if the current modCount is same as and when the Iterator was created. If there is a change in the modCount value it throws a ConcurrentModificationException.

Hence for the above-declared list, an operation like below will not throw any exception:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.set(0, "Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next());
}
```

But adding a new element to the List after initializing an Iterator will throw a ConcurrentModificationException:

```
Iterator<String> fruitIterator = fruits.iterator();
fruits.add("Watermelon");
while(fruitIterator.hasNext()){
    System.out.println(fruitIterator.next()); //ConcurrentModificationException here
}
```

Iterating backwards

```
for (int i = (fruits.size() - 1); i >= 0; i--) {
    System.out.println(fruits.get(i));
    if ("Apple".equals(fruits.get(i))) {
        fruits.remove(i);
    }
}
```

This does not skip anything. The downside of this approach is that the output is reverse. However, in most cases where you remove items that will not matter. You should never do this with LinkedList.

Iterating forward, adjusting the loop index

```
for (int i = 0; i < fruits.size(); i++) {
    System.out.println(fruits.get(i));
```

```
if ("Apple".equals(fruits.get(i))) {  
    fruits.remove(i);  
    i--;  
}  
}
```

这不会跳过任何元素。当从列表（List）中移除第 *i* 个元素时，原本位于索引 *i+1* 的元素会变成新的第 *i* 个元素。因此，循环可以将 *i* 减一，以便下一次迭代处理下一个元素，而不会跳过任何元素。

使用“应移除”列表

```
ArrayList shouldBeRemoved = new ArrayList();  
for (String str : currentArrayList) {  
    if (condition) {  
        shouldBeRemoved.add(str);  
    }  
}  
currentArrayList.removeAll(shouldBeRemoved);
```

该方案使开发者能够以更简洁的方式检查是否移除了正确的元素。

版本 ≥ Java SE 8

在 Java 8 中，可以使用以下替代方案。如果不需要在循环中进行移除操作，这些方法更简洁明了。

过滤流（Filtering a Stream）

一个列表可以被流式处理和过滤。可以使用合适的过滤器来移除所有不需要的元素。

```
List<String> filteredList =  
fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());
```

注意，与这里的其他示例不同，此示例会生成一个新的List实例，并保持原始 List不变。

使用removeIf

如果只需要移除一组元素，可以避免构造流的开销。

```
fruits.removeIf(p -> "Apple".equals(p));
```

第24.2节：从现有数据构建集合

标准集合

Java集合框架

从单个数据值构建List的简单方法是使用java.util.Arrays类的Arrays.asList方法：

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

所有标准集合实现都提供了以另一个集合作为参数的构造函数，在构造时将所有元素添加到新集合中：

```
List<String> list = new ArrayList<>(data); // 将按原样添加数据  
Set<String> set1 = new HashSet<>(data); // 将添加数据，仅保留唯一值
```

```
if ("Apple".equals(fruits.get(i))) {  
    fruits.remove(i);  
    i--;  
}  
}
```

This does not skip anything. When the *i*th element is removed from the List, the element originally positioned at index *i+1* becomes the new *i*th element. Therefore, the loop can decrement *i* in order for the next iteration to process the next element, without skipping.

Using a "should-be-removed" list

```
ArrayList shouldBeRemoved = new ArrayList();  
for (String str : currentArrayList) {  
    if (condition) {  
        shouldBeRemoved.add(str);  
    }  
}  
currentArrayList.removeAll(shouldBeRemoved);
```

This solution enables the developer to check if the correct elements are removed in a cleaner way.

Version ≥ Java SE 8

In Java 8 the following alternatives are possible. These are cleaner and more straightforward if the removing does not have to happen in a loop.

Filtering a Stream

A List can be streamed and filtered. A proper filter can be used to remove all undesired elements.

```
List<String> filteredList =  
fruits.stream().filter(p -> !"Apple".equals(p)).collect(Collectors.toList());
```

Note that unlike all the other examples here, this example produces a new List instance and keeps the original List unchanged.

Using removeIf

Saves the overhead of constructing a stream if all that is needed is to remove a set of items.

```
fruits.removeIf(p -> "Apple".equals(p));
```

Section 24.2: Constructing collections from existing data

Standard Collections

Java Collections framework

A simple way to construct a List from individual data values is to use java.util.Arrays method Arrays.asList:

```
List<String> data = Arrays.asList("ab", "bc", "cd", "ab", "bc", "cd");
```

All standard collection implementations provide constructors that take another collection as an argument adding all elements to the new collection at the time of construction:

```
List<String> list = new ArrayList<>(data); // will add data as is  
Set<String> set1 = new HashSet<>(data); // will add data keeping only unique values
```

```
SortedSet<String> set2 = new TreeSet<>(data); // 将添加数据，保留唯一值并排序
Set<String> set3 = new LinkedHashSet<>(data); // 将添加数据，仅保留唯一值并
保持原始顺序
```

Google Guava 集合框架

另一个很棒的框架是Google Guava，它是一个非常实用的工具类（提供便捷的静态方法），用于构造不同类型的标准集合Lists和Sets：

```
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
...
List<String> list1 = Lists.newArrayList("ab", "bc", "cd");
List<String> list2 = Lists.newArrayList(data);
Set<String> set4 = Sets.newHashSet(data);
SortedSet<String> set5 = Sets.newTreeSet("bc", "cd", "ab", "bc", "cd");
```

映射集合

Java集合框架

同样对于映射，给定一个Map<String, Object> map，可以构造一个包含所有元素的新映射，如下所示：

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Apache Commons Collections 框架

使用Apache Commons，你可以通过ArrayUtils.toMap以及MapUtils.toMap用数组创建Map：

```
import org.apache.commons.lang3.ArrayUtils;
...
// 取自 org.apache.commons.lang.ArrayUtils#toMap JavaDoc

// 创建一个映射颜色的Map。
Map colorMap = MapUtils.toMap(new String[][] {
    {"RED", "#FF0000"},
    {"GREEN", "#00FF00"},
    {"BLUE", "#0000FF"});
```

数组的每个元素必须是Map.Entry或数组，且包含至少两个元素，其中第一个元素用作键，第二个元素用作值。

Google Guava 集合框架

来自Google Guava框架的工具类名为Maps：

```
import com.google.common.collect.Maps;
...
void howToCreateMapsMethod(Function<? super K, V> valueFunction,
    Iterable<K> keys1,
Set<K> keys2,
SortedSet<K> keys3) {
    ImmutableMap<K, V> map1 = toMap(keys1, valueFunction); // 不可变副本
    Map<K, V> map2 = asMap(keys2, valueFunction); // 实时 Map 视图
    SortedMap<K, V> map3 = toMap(keys3, valueFunction); // 实时 Map 视图
}
```

版本 ≥ Java SE 8

```
SortedSet<String> set2 = new TreeSet<>(data); // will add data keeping unique values and sorting
Set<String> set3 = new LinkedHashSet<>(data); // will add data keeping only unique values and
preserving the original order
```

Google Guava Collections framework

Another great framework is Google Guava that is amazing utility class (providing convenience static methods) for construction of different types of standard collections Lists and Sets:

```
import com.google.common.collect.Lists;
import com.google.common.collect.Sets;
...
List<String> list1 = Lists.newArrayList("ab", "bc", "cd");
List<String> list2 = Lists.newArrayList(data);
Set<String> set4 = Sets.newHashSet(data);
SortedSet<String> set5 = Sets.newTreeSet("bc", "cd", "ab", "bc", "cd");
```

Mapping Collections

Java Collections framework

Similarly for maps, given a Map<String, Object> map a new map can be constructed with all elements as follows:

```
Map<String, Object> map1 = new HashMap<>(map);
SortedMap<String, Object> map2 = new TreeMap<>(map);
```

Apache Commons Collections framework

Using Apache Commons you can create Map using array in ArrayUtils.toMap as well as MapUtils.toMap:

```
import org.apache.commons.lang3.ArrayUtils;
...
// Taken from org.apache.commons.lang.ArrayUtils#toMap JavaDoc

// Create a Map mapping colors.
Map colorMap = MapUtils.toMap(new String[][] {
    {"RED", "#FF0000"},
    {"GREEN", "#00FF00"},
    {"BLUE", "#0000FF"});
```

Each element of the array must be either a Map.Entry or an Array, containing at least two elements, where the first element is used as key and the second as value.

Google Guava Collections framework

Utility class from Google Guava framework is named Maps:

```
import com.google.common.collect.Maps;
...
void howToCreateMapsMethod(Function<? super K, V> valueFunction,
    Iterable<K> keys1,
Set<K> keys2,
SortedSet<K> keys3) {
    ImmutableMap<K, V> map1 = toMap(keys1, valueFunction); // Immutable copy
    Map<K, V> map2 = asMap(keys2, valueFunction); // Live Map view
    SortedMap<K, V> map3 = toMap(keys3, valueFunction); // Live Map view
}
```

Version ≥ Java SE 8

使用 Stream,

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

或者

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

第24.3节：声明一个ArrayList并添加对象

我们可以创建一个ArrayList（遵循List接口）：

```
List aListOfFruits = new ArrayList();  
版本 ≥ Java SE 5  
List<String> aListOfFruits = new ArrayList<String>();  
版本 ≥ Java SE 7  
List<String> aListOfFruits = new ArrayList<>();
```

现在，使用方法add来添加一个String：

```
aListOfFruits.add("Melon");  
aListOfFruits.add("Strawberry");
```

在上述示例中，ArrayList将在索引0处包含String"Melon"，在索引1处包含String"Strawberry"。

我们也可以使用addAll(Collection<? extends E> c)方法添加多个元素

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();  
aListOfFruitsAndVeggies.add("Onion");  
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

现在“洋葱”被放置在 aListOfFruitsAndVeggies 的索引0位置，“哈密瓜”在索引1，“草莓”在索引2。

第24.4节：遍历集合

遍历列表

```
List<String> names = new ArrayList<>(Arrays.asList("克莱门汀", "杜兰", "迈克"));  
版本 ≥ Java SE 8  
names.forEach(System.out::println);
```

如果需要并行处理，使用

```
names.parallelStream().forEach(System.out::println);  
版本 ≥ Java SE 5  
for (String name : names) {  
    System.out.println(name);  
}  
Java SE 5之前的版本  
for (int i = 0; i < names.size(); i++) {  
    System.out.println(names.get(i));  
}  
版本 ≥ Java SE 1.2
```

Using Stream,

```
Stream.of("xyz", "abc").collect(Collectors.toList());
```

or

```
Arrays.stream("xyz", "abc").collect(Collectors.toList());
```

Section 24.3: Declaring an ArrayList and adding objects

We can create an ArrayList (following the List interface):

```
List aListOfFruits = new ArrayList();  
Version ≥ Java SE 5  
List<String> aListOfFruits = new ArrayList<String>();  
Version ≥ Java SE 7  
List<String> aListOfFruits = new ArrayList<>();
```

Now, use the method add to add a String:

```
aListOfFruits.add("Melon");  
aListOfFruits.add("Strawberry");
```

In the above example, the ArrayList will contain the String "Melon" at index 0 and the String "Strawberry" at index 1.

Also we can add multiple elements with addAll(Collection<? extends E> c) method

```
List<String> aListOfFruitsAndVeggies = new ArrayList<String>();  
aListOfFruitsAndVeggies.add("Onion");  
aListOfFruitsAndVeggies.addAll(aListOfFruits);
```

Now "Onion" is placed at 0 index in aListOfFruitsAndVeggies, "Melon" is at index 1 and "Strawberry" is at index 2.

Section 24.4: Iterating over Collections

Iterating over List

```
List<String> names = new ArrayList<>(Arrays.asList("Clementine", "Duran", "Mike"));  
Version ≥ Java SE 8  
names.forEach(System.out::println);
```

If we need parallelism use

```
names.parallelStream().forEach(System.out::println);  
Version ≥ Java SE 5  
for (String name : names) {  
    System.out.println(name);  
}  
Version < Java SE 5  
for (int i = 0; i < names.size(); i++) {  
    System.out.println(names.get(i));  
}  
Version ≥ Java SE 1.2
```

```
//创建支持正向和反向遍历的ListIterator  
ListIterator<String> listIterator = names.listIterator();
```

```
//正向遍历列表  
while(listIterator.hasNext()) {  
    System.out.println(listIterator.next());  
}
```

//当从上面的正向迭代器遍历到最后一个元素后，反向遍历列表

```
while(listIterator.hasPrevious()) {  
    System.out.println(listIterator.previous());  
}
```

遍历Set集合

```
Set<String> names = new HashSet<>(Arrays.asList("Clementine", "Duran", "Mike"));  
版本 ≥ Java SE 8  
names.forEach(System.out::println);  
版本 ≥ Java SE 5  
for (Iterator<String> iterator = names.iterator(); iterator.hasNext(); ) {  
    System.out.println(iterator.next());  
}
```

```
for (String name : names) {  
    System.out.println(name);  
}
```

Java SE 5之前的版本

```
Iterator iterator = names.iterator();  
while (iterator.hasNext()) {  
    System.out.println(iterator.next());  
}
```

遍历 Map

```
Map<Integer, String> names = new HashMap<>();  
names.put(1, "Clementine");  
names.put(2, "Duran");  
names.put(3, "Mike");  
版本 ≥ Java SE 8  
names.forEach((key, value) -> System.out.println("Key: " + key + " Value: " + value));  
版本 ≥ Java SE 5
```

```
for (Map.Entry<Integer, String> entry : names.entrySet()) {  
    System.out.println(entry.getKey());  
    System.out.println(entry.getValue());  
}
```

```
// 仅遍历键  
for (Integer key : names.keySet()) {  
    System.out.println(key);  
}  
// 仅遍历值  
for (String value : names.values()) {  
    System.out.println(value);  
}
```

Java SE 5之前的版本

```
Iterator entries = names.entrySet().iterator();  
while (entries.hasNext()) {  
    Map.Entry entry = (Map.Entry) entries.next();  
    System.out.println(entry.getKey());  
    System.out.println(entry.getValue());
```

```
//Creates ListIterator which supports both forward as well as backward traversal  
ListIterator<String> listIterator = names.listIterator();
```

```
//Iterates list in forward direction  
while(listIterator.hasNext()) {  
    System.out.println(listIterator.next());  
}
```

```
//Iterates list in backward direction once reaches the last element from above iterator in forward  
direction  
while(listIterator.hasPrevious()) {  
    System.out.println(listIterator.previous());  
}
```

Iterating over Set

```
Set<String> names = new HashSet<>(Arrays.asList("Clementine", "Duran", "Mike"));  
Version ≥ Java SE 8  
names.forEach(System.out::println);  
Version ≥ Java SE 5  
for (Iterator<String> iterator = names.iterator(); iterator.hasNext(); ) {  
    System.out.println(iterator.next());  
}
```

```
for (String name : names) {  
    System.out.println(name);  
}
```

Version < Java SE 5

```
Iterator iterator = names.iterator();  
while (iterator.hasNext()) {  
    System.out.println(iterator.next());  
}
```

Iterating over Map

```
Map<Integer, String> names = new HashMap<>();  
names.put(1, "Clementine");  
names.put(2, "Duran");  
names.put(3, "Mike");  
Version ≥ Java SE 8  
names.forEach((key, value) -> System.out.println("Key: " + key + " Value: " + value));  
Version ≥ Java SE 5
```

```
for (Map.Entry<Integer, String> entry : names.entrySet()) {  
    System.out.println(entry.getKey());  
    System.out.println(entry.getValue());  
}
```

```
// Iterating over only keys  
for (Integer key : names.keySet()) {  
    System.out.println(key);  
}  
// Iterating over only values  
for (String value : names.values()) {  
    System.out.println(value);  
}
```

```
Version < Java SE 5  
Iterator entries = names.entrySet().iterator();  
while (entries.hasNext()) {  
    Map.Entry entry = (Map.Entry) entries.next();  
    System.out.println(entry.getKey());  
    System.out.println(entry.getValue());
```

}

第24.5节：不可变的空集合

有时使用不可变的空集合是合适的。Collections类提供了以高效方式获取此类集合的方法：

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

这些方法是泛型的，会自动将返回的集合转换为赋值时的类型。也就是说，例如调用emptyList()可以赋值给任何类型的List，emptySet()和emptyMap()同理。

这些方法返回的集合是不可变的，如果尝试调用会改变其内容的方法（如add、put等），将抛出UnsupportedOperationException异常。这些集合主要用作空方法结果或其他默认值的替代，而不是使用null或通过new创建对象。

}

Section 24.5: Immutable Empty Collections

Sometimes it is appropriate to use an immutable empty collection. The [Collections](#) class provides methods to get such collections in an efficient way:

```
List<String> anEmptyList = Collections.emptyList();
Map<Integer, Date> anEmptyMap = Collections.emptyMap();
Set<Number> anEmptySet = Collections.emptySet();
```

These methods are generic and will automatically convert the returned collection to the type it is assigned to. That is, an invocation of e.g. emptyList() can be assigned to any type of [List](#) and likewise for emptySet() and emptyMap().

The collections returned by these methods are immutable in that they will throw [UnsupportedOperationException](#) if you attempt to call methods which would change their contents (add, put, etc.). These collections are primarily useful as substitutes for empty method results or other default values, instead of using [null](#) or creating objects with [new](#).

第24.6节：子集合

List subList(int fromIndex, int toIndex)

这里的 fromIndex 是包含的，toIndex 是不包含的。

```
List list = new ArrayList();
List list1 = list.subList(fromIndex, toIndex);
```

1. 如果列表在给定范围内不存在，则会抛出 [IndexOutOfBoundsException](#)。
2. 对 list1 所做的任何更改都会影响 list 中的相同更改。这称为备份集合（backed collections）。
3. 如果 fromIndex 大于 toIndex (`fromIndex > toIndex`)，则会抛出 [IllegalArgumentException](#)。

示例：

```
List<String> list = new ArrayList<String>();
List<String> list = new ArrayList<String>();
list.add("Hello1");
list.add("Hello2");
System.out.println("Before Sublist "+list);
List<String> list2 = list.subList(0, 1);
list2.add("Hello3");
System.out.println("After sublist changes "+list);
```

输出：

```
Before Sublist [Hello1, Hello2]
After sublist changes [Hello1, Hello3, Hello2]
```

设置子集(fromIndex,toIndex)

这里的 fromIndex 是包含的，toIndex 是不包含的。

```
Set set = new TreeSet();
```

Section 24.6: Sub Collections

List subList(int fromIndex, int toIndex)

Here fromIndex is inclusive and toIndex is exclusive.

```
List list = new ArrayList();
List list1 = list.subList(fromIndex, toIndex);
```

1. If the list doesn't exist in the give range, it throws [IndexOutOfBoundsException](#).
2. What ever changes made on the list1 will impact the same changes in the list. This is called backed collections.
3. If the fromnIndex is greater than the toIndex (`fromIndex > toIndex`) it throws [IllegalArgumentException](#).

Example:

```
List<String> list = new ArrayList<String>();
List<String> list = new ArrayList<String>();
list.add("Hello1");
list.add("Hello2");
System.out.println("Before Sublist "+list);
List<String> list2 = list.subList(0, 1);
list2.add("Hello3");
System.out.println("After sublist changes "+list);
```

Output:

```
Before Sublist [Hello1, Hello2]
After sublist changes [Hello1, Hello3, Hello2]
```

Set subSet(fromIndex,toIndex)

Here fromIndex is inclusive and toIndex is exclusive.

```
Set set = new TreeSet();
```

```
Set set1 = set.subSet(fromIndex,toIndex);
```

尝试插入超出其范围的元素时，返回的集合将抛出 `IllegalArgumentException` 异常。

Map subMap(fromKey,toKey)

`fromKey` 包含在内，`toKey` 不包含在内

```
Map map = new TreeMap();
Map map1 = map.get(fromKey,toKey);
```

如果 `fromKey` 大于 `toKey`，或者该映射本身有受限范围，且 `fromKey` 或 `toKey` 位于该范围的边界之外，则会抛出 `IllegalArgumentException` 异常。

所有集合都支持备份集合，意味着对子集合所做的更改将在主集合中反映相同的更改。

第 24.7 节：不可修改集合

有时暴露内部集合并不是一个好做法，因为它的可变特性可能导致恶意代码漏洞。为了提供“只读”集合，Java 提供了其不可修改版本。

不可修改集合通常是可修改集合的副本，保证集合本身不能被更改。尝试修改它将导致抛出 `UnsupportedOperationException` 异常。

需要注意的是，集合内部存在的对象仍然可以被修改。

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MyPojoClass {
    private List<Integer> intList = new ArrayList<>();

    public void addValueToIntList(Integer value){
        intList.add(value);
    }

    public List<Integer> getIntList() {
        return Collections.unmodifiableList(intList);
    }
}
```

以下尝试修改不可修改的集合将抛出异常：

```
import java.util.List;

public class App {

    public static void main(String[] args) {
        MyPojoClass pojo = new MyPojoClass();
        pojo.addValueToIntList(42);

        List<Integer> list = pojo.getIntList();
        list.add(69);
    }
}
```

```
Set set1 = set.subSet(fromIndex,toIndex);
```

The returned set will throw an `IllegalArgumentException` on an attempt to insert an element outside its range.

Map subMap(fromKey,toKey)

`fromKey` is inclusive and `toKey` is exclusive

```
Map map = new TreeMap();
Map map1 = map.get(fromKey,toKey);
```

If `fromKey` is greater than `toKey` or if this map itself has a restricted range, and `fromKey` or `toKey` lies outside the bounds of the range then it throws `IllegalArgumentException`.

All the collections support backed collections means changes made on the sub collection will have same change on the main collection.

Section 24.7: Unmodifiable Collection

Sometimes it's not a good practice expose an internal collection since it can lead to a malicious code vulnerability due to its mutable characteristic. In order to provide "read-only" collections java provides its unmodifiable versions.

An unmodifiable collection is often a copy of a modifiable collection which guarantees that the collection itself cannot be altered. Attempts to modify it will result in an `UnsupportedOperationException` exception.

It is important to notice that objects which are present inside the collection can still be altered.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class MyPojoClass {
    private List<Integer> intList = new ArrayList<>();

    public void addValueToIntList(Integer value){
        intList.add(value);
    }

    public List<Integer> getIntList() {
        return Collections.unmodifiableList(intList);
    }
}
```

The following attempt to modify an unmodifiable collection will throw an exception:

```
import java.util.List;

public class App {

    public static void main(String[] args) {
        MyPojoClass pojo = new MyPojoClass();
        pojo.addValueToIntList(42);

        List<Integer> list = pojo.getIntList();
        list.add(69);
    }
}
```

输出：

```
主线程中出现异常 "main" java.lang.UnsupportedOperationException  
位于 java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)  
位于 App.main(App.java:12)
```

第24.8节：陷阱：并发修改异常

当使用迭代器对象以外的方法在迭代集合时修改该集合，就会发生此异常。例如，我们有一个帽子列表，想要移除所有带有耳罩的帽子：

```
List<IHat> hats = new ArrayList<>();  
hats.add(new Ushanka()); // 这顶有耳罩  
hats.add(new Fedora());  
hats.add(new Sombrero());  
for (IHat hat : hats) {  
    if (hat.hasEarFlaps()) {  
        hats.remove(hat);  
    }  
}
```

如果运行这段代码，会抛出ConcurrentModificationException异常，因为代码在迭代集合时修改了集合。如果多个线程同时操作同一个列表，其中一个线程在迭代时尝试修改集合，也可能出现相同的异常。多线程中集合的并发修改是常见情况，但应使用并发编程工具箱中的常用工具来处理，如同步锁、专门支持并发修改的集合、修改克隆集合等。

输出：

```
Exception in thread "main" java.lang.UnsupportedOperationException  
at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)  
at App.main(App.java:12)
```

Section 24.8: Pitfall: concurrent modification exceptions

This exception occurs when a collection is modified while iterating over it using methods other than those provided by the iterator object. For example, we have a list of hats and we want to remove all those that have ear flaps:

```
List<IHat> hats = new ArrayList<>();  
hats.add(new Ushanka()); // that one has ear flaps  
hats.add(new Fedora());  
hats.add(new Sombrero());  
for (IHat hat : hats) {  
    if (hat.hasEarFlaps()) {  
        hats.remove(hat);  
    }  
}
```

If we run this code, **ConcurrentModificationException** will be raised since the code modifies the collection while iterating it. The same exception may occur if one of the multiple threads working with the same list is trying to modify the collection while others iterate over it. Concurrent modification of collections in multiple threads is a natural thing, but should be treated with usual tools from the concurrent programming toolbox such as synchronization locks, special collections adopted for concurrent modification, modifying the cloned collection from initial etc.

第24.9节：使用Iterator从列表中移除匹配项

上面我提到了一个在循环中从列表中移除项的例子，这次我想举另一个可能有用的例子，使用Iterator接口。

这是一个演示技巧，当处理列表中想要去除的重复项时可能会用到。

注意：这只是对在循环中从列表中移除项示例的补充：

那么我们像往常一样定义我们的列表

```
String[] names = {"James", "Smith", "Sonny", "Huckle", "Berry", "Finn", "Allan"};  
List<String> nameList = new ArrayList<>();  
  
//从数组创建列表  
nameList.addAll(Arrays.asList(names));  
  
String[] removeNames = {"Sonny", "Huckle", "Berry"};  
List<String> removeNameList = new ArrayList<>();  
  
//从数组创建列表  
removeNameList.addAll(Arrays.asList(removeNames));
```

以下方法接收两个集合对象，并执行从removeNameList中移除与nameList中元素匹配的元素的操作。

Section 24.9: Removing matching items from Lists using Iterator

Above I noticed an example to remove items from a List within a Loop and I thought of another example that may come in handy this time using the [Iterator](#) interface.

This is a demonstration of a trick that might come in handy when dealing with duplicate items in lists that you want to get rid of.

Note: This is only adding on to the [Removing items from a List within a loop](#) example:

So let's define our lists as usual

```
String[] names = {"James", "Smith", "Sonny", "Huckle", "Berry", "Finn", "Allan"};  
List<String> nameList = new ArrayList<>();  
  
//Create a List from an Array  
nameList.addAll(Arrays.asList(names));  
  
String[] removeNames = {"Sonny", "Huckle", "Berry"};  
List<String> removeNameList = new ArrayList<>();  
  
//Create a List from an Array  
removeNameList.addAll(Arrays.asList(removeNames));
```

The following method takes in two Collection objects and performs the magic of removing the elements in our removeNameList that match with elements in nameList.

```

private static void removeNames(Collection<String> collection1, Collection<String> collection2) {
    //获取迭代器。
    Iterator<String> iterator = collection1.iterator();

    //当集合中有项目时循环
    while(iterator.hasNext()){
        if (collection2.contains(iterator.next()))
            iterator.remove(); //移除当前的姓名或项目
    }
}

```

调用该方法并传入nameList和removeNameList，如下所示
`removeNames(nameList,removeNameList);`
将产生以下输出：

移除姓名前的数组列表：**James Smith Sonny Huckle Berry Finn Allan**
移除姓名后的数组列表：**James Smith Finn Allan**

Collections的一个简单实用用法，可用于删除列表中重复的元素。

第24.10节：合并列表

以下方法可用于合并列表而不修改源列表。

第一种方法。代码行数较多，但易于理解

```

List<String> newList = new ArrayList<String>();
newList.addAll(listOne);
newList.addAll(listTwo);

```

第二种方法。少一行代码，但可读性较差。

```

List<String> newList = new ArrayList<String>(listOne);
newList.addAll(listTwo);

```

第三种方法。需要第三方Apache commons-collections库。

```
ListUtils.union(listOne,listTwo);
```

版本 ≥ Java SE 8

使用流（Streams）也可以实现相同功能

```

List<String> newList = Stream.concat(listOne.stream(),
listTwo.stream()).collect(Collectors.toList());

```

参考文献。[接口 List](#)

第24.11节：创建你自己的Iterable结构以供Iterator或for-each循环使用

为了确保我们的集合可以使用迭代器或for-each循环进行迭代，我们必须注意以下步骤：

1. 我们想要迭代的东西必须是Iterable并且暴露iterator()。
2. 通过重写 hasNext()、next() 和 remove() 来设计一个 java.util.Iterator。

```

private static void removeNames(Collection<String> collection1, Collection<String> collection2) {
    //get Iterator.
    Iterator<String> iterator = collection1.iterator();

    //Loop while collection has items
    while(iterator.hasNext()){
        if (collection2.contains(iterator.next()))
            iterator.remove(); //remove the current Name or Item
    }
}

```

Calling the method and passing in the nameList and the removeNameList as follows
`removeNames(nameList,removeNameList);`
Will produce the following output:

Array List before removing names: **James Smith Sonny Huckle Berry Finn Allan**
Array List after removing names: **James Smith Finn Allan**

A simple neat use for Collections that may come in handy to remove repeating elements within lists.

Section 24.10: Join lists

Following ways can be used for joining lists without modifying source list(s).

First approach. Has more lines but easy to understand

```

List<String> newList = new ArrayList<String>();
newList.addAll(listOne);
newList.addAll(listTwo);

```

Second approach. Has one less line but less readable.

```

List<String> newList = new ArrayList<String>(listOne);
newList.addAll(listTwo);

```

Third approach. Requires third party [Apache commons-collections](#) library.

```
ListUtils.union(listOne,listTwo);
```

Version ≥ Java SE 8

Using Streams the same can be achieved by

```

List<String> newList = Stream.concat(listOne.stream(),
listTwo.stream()).collect(Collectors.toList());

```

References. [Interface List](#)

Section 24.11: Creating your own Iterable structure for use with Iterator or for-each loop

To ensure that our collection can be iterated using iterator or for-each loop, we have to take care of following steps:

1. The stuff we want to iterate upon has to be Iterable and expose iterator().
2. Design a [java.util.Iterator](#) by overriding hasNext(), next() and remove().

我在下面添加了一个简单的泛型链表实现，使用上述实体使链表可迭代。

```
包 org.algorithms.linkedlist;

导入 java.util.Iterator;
导入 java.util.NoSuchElementException;

公共类 LinkedList<T> 实现 Iterable<T> {

    节点<T> head, current;

    私有静态类 Node<T> {
        T 数据;
        节点<T> next;

        节点(T 数据) {
            this.data = data;
        }
    }

    public LinkedList(T data) {
        head = new Node<>(data);
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<T> {

        Node<T> node = head;

        @Override
        public boolean hasNext() {
            return node != null;
        }

        @Override
        public T next() {
            if (!hasNext())
                throw new NoSuchElementException();
            Node<T> prevNode = node;
            node = node.next;
            return prevNode.data;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Removal logic not implemented.");
        }

        public void add(T data) {
            Node current = head;
```

I have added a simple generic linked list implementation below that uses above entities to make the linked list iterable.

```
package org.algorithms.linkedlist;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class LinkedList<T> implements Iterable<T> {

    Node<T> head, current;

    private static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
        }
    }

    public LinkedList(T data) {
        head = new Node<>(data);
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator<T> {

        Node<T> node = head;

        @Override
        public boolean hasNext() {
            return node != null;
        }

        @Override
        public T next() {
            if (!hasNext())
                throw new NoSuchElementException();
            Node<T> prevNode = node;
            node = node.next;
            return prevNode.data;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException("Removal logic not implemented.");
        }

        public void add(T data) {
            Node current = head;
```

```

while (current.next != null)
    current = current.next;
current.next = new Node<>(data);
}

class A {
    public static void main(String[] args) {

        LinkedList<Integer> list = new LinkedList<>(1);
        list.add(2);
        list.add(4);
        list.add(3);

        //测试 #1
        System.out.println("使用迭代器:");
        Iterator<Integer> itr = list.iterator();
        while (itr.hasNext()) {
            Integer i = itr.next();
            System.out.print(i + " ");
        }

        //测试 #2
        System.out.println("使用增强for循环:");
        for (Integer data : list) {
            System.out.print(data + " ");
        }
    }
}

```

输出

```

使用迭代器:
1 2 4 3
使用增强for循环:
1 2 4 3

```

这将在 Java 7 及以上版本运行。您也可以通过替换使其在 Java 5 和 Java 6 上运行：

```
LinkedList<Integer> list = new LinkedList<>(1);
```

with

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

或者通过合并兼容的更改使用任何其他版本。

第24.12节：集合与原始值

Java中的集合只适用于对象。也就是说，Java中没有`Map<int, int>`。相反，原始值需要被装箱成对象，如`Map<Integer, Integer>`。Java的自动装箱功能使得这些集合的使用变得透明：

```
Map<Integer, Integer> map = new HashMap<>();
```

```

while (current.next != null)
    current = current.next;
current.next = new Node<>(data);
}

class App {
    public static void main(String[] args) {

        LinkedList<Integer> list = new LinkedList<>(1);
        list.add(2);
        list.add(4);
        list.add(3);

        //Test #1
        System.out.println("using Iterator:");
        Iterator<Integer> itr = list.iterator();
        while (itr.hasNext()) {
            Integer i = itr.next();
            System.out.print(i + " ");
        }

        //Test #2
        System.out.println("\n\nusing for-each:");
        for (Integer data : list) {
            System.out.print(data + " ");
        }
    }
}

```

Output

```

using Iterator:
1 2 4 3
using for-each:
1 2 4 3

```

This will run in Java 7+. You can make it run on Java 5 and Java 6 also by substituting:

```
LinkedList<Integer> list = new LinkedList<>(1);
```

with

```
LinkedList<Integer> list = new LinkedList<Integer>(1);
```

or just any other version by incorporating the compatible changes.

Section 24.12: Collections and Primitive Values

Collections in Java only work for objects. I.e. there is no `Map<int, int>` in Java. Instead, primitive values need to be *boxed* into objects, as in `Map<Integer, Integer>`. Java auto-boxing will enable transparent use of these collections:

```
Map<Integer, Integer> map = new HashMap<>();
```

```
map.put(1, 17); // int到Integer对象的自动装箱  
int a = map.get(1); // 自动拆箱。
```

不幸的是，这带来了相当大的开销。一个HashMap<Integer, Integer>每个条目大约需要72字节（例如，在64位JVM上使用压缩指针，假设整数大于256，且假设映射的负载为50%）。由于实际数据只有8字节，这导致了巨大的开销。此外，它需要两级间接寻址（Map -> Entry -> Value），因此速度不必要地慢。

存在几个针对原始数据类型优化的库（在50%负载时每个条目仅需约16字节，即内存使用减少4倍，且少一级间接寻址），在Java中使用大量原始值集合时，可以带来显著的性能提升。

```
map.put(1, 17); // Automatic boxing of int to Integer objects  
int a = map.get(1); // Automatic unboxing.
```

Unfortunately, the overhead of this is *substantial*. A HashMap<Integer, Integer> will require about 72 bytes per entry (e.g. on 64-bit JVM with compressed pointers, and assuming integers larger than 256, and assuming 50% load of the map). Because the actual data is only 8 bytes, this yields a massive overhead. Furthermore, it requires two level of indirection (Map -> Entry -> Value) it is unnecessarily slow.

There exist several libraries with optimized collections for primitive data types (that require only ~16 bytes per entry at 50% load, i.e. 4x less memory, and one level of indirection less), that can yield substantial performance benefits when using large collections of primitive values in Java.

第25章：列表

列表（list）是一个有序的值集合。在Java中，列表是Java集合框架（Java Collections Framework）的一部分。列表实现了[java.util.List接口](#)，该接口继承自[java.util.Collection](#)。

第25.1节：对泛型列表进行排序

Collections类提供了两个标准的静态方法用于排序列表：

- `sort(List<T> list)` 适用于T扩展Comparable<? super T>的列表，
- `sort(List<T> list, Comparator<? super T> c)` 适用于任何类型的列表。

使用前者需要修改被排序列表元素的类，这并不总是可行的。且这可能不理想，因为虽然它提供了默认排序，但在不同情况下可能需要其他排序顺序，或者排序只是一次性任务。

假设我们有一个任务，需要对以下类的实例对象进行排序：

```
public class User {  
    public final Long id;  
    public final String username;  
  
    public User(Long id, String username) {  
        this.id = id;  
        this.username = username;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%s:%d", username, id);  
    }  
}
```

为了使用 `Collections.sort(List<User> list)`，我们需要修改 `User` 类以实现 `Comparable` 接口。例如

```
public class User implements Comparable<User> {  
    public final Long id;  
    public final String username;  
  
    public User(Long id, String username) {  
        this.id = id;  
        this.username = username;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%s:%d", username, id);  
    }  
  
    @Override  
    /** 'User' 对象的自然排序是基于 'id' 字段。 */  
    public int compareTo(User o) {  
        return id.compareTo(o.id);  
    }  
}
```

Chapter 25: Lists

A *list* is an *ordered* collection of values. In Java, lists are part of the [Java Collections Framework](#). Lists implement the [java.util.List](#) interface, which extends [java.util.Collection](#).

Section 25.1: Sorting a generic list

The `Collections` class offers two standard static methods to sort a list:

- `sort(List<T> list)` applicable to lists where T `extends Comparable<? super T>`, and
- `sort(List<T> list, Comparator<? super T> c)` applicable to lists of any type.

Applying the former requires amending the class of list elements being sorted, which is not always possible. It might also be undesirable as although it provides the default sorting, other sorting orders may be required in different circumstances, or sorting is just a one off task.

Consider we have a task of sorting objects that are instances of the following class:

```
public class User {  
    public final Long id;  
    public final String username;  
  
    public User(Long id, String username) {  
        this.id = id;  
        this.username = username;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%s:%d", username, id);  
    }  
}
```

In order to use `Collections.sort(List<User> list)` we need to modify the `User` class to implement the `Comparable` interface. For example

```
public class User implements Comparable<User> {  
    public final Long id;  
    public final String username;  
  
    public User(Long id, String username) {  
        this.id = id;  
        this.username = username;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%s:%d", username, id);  
    }  
  
    @Override  
    /** The natural ordering for 'User' objects is by the 'id' field. */  
    public int compareTo(User o) {  
        return id.compareTo(o.id);  
    }  
}
```

(顺便说一下：许多标准的 Java 类如 String、Long、Integer 都实现了 Comparable 接口。这使得这些元素的列表默认可排序，并简化了其他类中 compare 或 compareTo 的实现。)通过上述修改，我们可以轻松地根据类的自然排序对 User 对象列表进行排序。（在本例中，我们定义的排序依据是 id 值）。例如：

```
List<User> users = Lists.newArrayList(  
    new User(33L, "A"),  
    new User(25L, "B"),  
    new User(28L, "")  
);  
  
Collections.sort(users);  
  
System.out.print(users);  
// [B:25, C:28, A:33]
```

但是，假设我们想按name而不是id对User对象进行排序。或者，假设我们无法修改该类以实现 Comparable 接口。

这时，带有Comparator参数的sort方法就很有用：

```
Collections.sort(users, new Comparator<User>() {  
    @Override  
    /* 根据名字对两个'User'对象进行排序。 */  
    public int compare(User left, User right) {  
        return left.username.compareTo(right.username);  
    }  
});  
System.out.print(users);  
// [A:33, B:25, C:28]
```

版本 ≥ Java SE 8

在Java 8中，你可以使用lambda表达式代替匿名类。后者可以简化为一行代码：

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

此外，Java 8 在 List 接口上添加了一个默认的 sort方法，使排序更加简便。

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

第25.2节：将整数列表转换为字符串列表

```
List<Integer> nums = Arrays.asList(1, 2, 3);  
List<String> strings = nums.stream()  
.map(Object::toString)  
.collect(Collectors.toList());
```

即：

1. 从列表创建一个流
2. 使用Object::toString映射每个元素
3. 使用Collectors.toList()将String值收集到List中

第25.3节：实现 List 的类——优缺点

List 接口由不同的类实现。它们各自采用不同的策略实现该接口，并提供不同的优缺点。

(Aside: many standard Java classes such as String, Long, Integer implement the Comparable interface. This makes lists of those elements sortable by default, and simplifies implementation of compare or compareTo in other classes.)

With the modification above, we can easily sort a list of User objects based on the classes *natural ordering*. (In this case, we have defined that to be ordering based on id values). For example:

```
List<User> users = Lists.newArrayList(  
    new User(33L, "A"),  
    new User(25L, "B"),  
    new User(28L, "")  
);  
  
Collections.sort(users);  
  
System.out.print(users);  
// [B:25, C:28, A:33]
```

However, suppose that we wanted to sort User objects by name rather than by id. Alternatively, suppose that we had not been able to change the class to make it implement Comparable.

This is where the sort method with the Comparator argument is useful:

```
Collections.sort(users, new Comparator<User>() {  
    @Override  
    /* Order two 'User' objects based on their names. */  
    public int compare(User left, User right) {  
        return left.username.compareTo(right.username);  
    }  
});  
System.out.print(users);  
// [A:33, B:25, C:28]
```

Version ≥ Java SE 8

In Java 8 you can use a lambda instead of an anonymous class. The latter reduces to a one-liner:

```
Collections.sort(users, (l, r) -> l.username.compareTo(r.username));
```

Further, Java 8 adds a default sort method on the List interface, which simplifies sorting even more.

```
users.sort((l, r) -> l.username.compareTo(r.username))
```

Section 25.2: Convert a list of integers to a list of strings

```
List<Integer> nums = Arrays.asList(1, 2, 3);  
List<String> strings = nums.stream()  
.map(Object::toString)  
.collect(Collectors.toList());
```

That is:

1. Create a stream from the list
2. Map each element using Object::toString
3. Collect the String values into a List using Collectors.toList()

Section 25.3: Classes implementing List - Pros and Cons

The List interface is implemented by different classes. Each of them has its own way for implementing it with different strategies and providing different pros and cons.

实现 List 的类

以下是 Java SE 8 中所有实现了 `java.util.List` 接口的公共类：

1. 抽象类：

- `AbstractList`
- `AbstractSequentialList`

2. 具体类：

- `ArrayList`
- `AttributeList`
- `CopyOnWriteArrayList`
- `LinkedList`
- `RoleList`
- `RoleUnresolvedList`
- `Stack`
- 向量

每种实现的时间复杂度优缺点

ArrayList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

`ArrayList` 是 `List` 接口的可调整大小的数组实现。将列表存储到数组中，`ArrayList` 提供了除实现 `List` 接口的方法外，用于操作数组大小的方法。

初始化大小为100的 Integer 类型 ArrayList

```
List<Integer> myList = new ArrayList<Integer>(100); // 构造一个具有指定初始容量的空列表。
```

- 优点：

`size`、`isEmpty`、`get`、`set`、`iterator` 和 `listIterator` 操作的运行时间是常数时间。因此，获取和设置列表中的每个元素具有相同的时间成本：

```
int e1 = myList.get(0); // \
int e2 = myList.get(10); // / => 全部相同的常数时间成本 => O(1)
myList.set(2,10); // /
```

- 缺点：

由于使用数组（静态结构）实现，添加超过数组大小的元素代价很大，因为需要为整个数组重新分配空间。然而，根据文档：

添加操作的摊销时间是常数时间，也就是说，添加 n 个元素需要 $O(n)$ 时间

删除一个元素需要 $O(n)$ 时间。

属性列表

Classes implementing List

These are all of the **public** classes in Java SE 8 that implement the `java.util.List` interface:

1. Abstract Classes:

- `AbstractList`
- `AbstractSequentialList`

2. Concrete Classes:

- `ArrayList`
- `AttributeList`
- `CopyOnWriteArrayList`
- `LinkedList`
- `RoleList`
- `RoleUnresolvedList`
- `Stack`
- `Vector`

Pros and Cons of each implementation in term of time complexity

ArrayList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

`ArrayList` is a resizable-array implementation of the `List` interface. Storing the list into an array, `ArrayList` provides methods (in addition to the methods implementing the `List` interface) for manipulating the size of the array.

Initialize ArrayList of Integer with size 100

```
List<Integer> myList = new ArrayList<Integer>(100); // Constructs an empty list with the specified initial capacity.
```

- PROS:

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. So getting and setting each element of the List has the same *time cost*:

```
int e1 = myList.get(0); // \
int e2 = myList.get(10); // / => All the same constant cost => O(1)
myList.set(2,10); // /
```

- CONS:

Being implemented with an array (static structure) adding elements over the size of the array has a big cost due to the fact that a new allocation need to be done for all the array. However, from [documentation](#):

The add operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time

Removing an element requires $O(n)$ time.

AttributeList

即将推出

CopyOnWriteArrayList

即将推出

链表

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, Serializable
```

[LinkedList](#) 是由一个 [双向链表](#) 实现的，这是一种由一组顺序链接的记录称为节点组成的链式数据结构。

初始化 Integer 类型的 LinkedList

```
List<Integer> myList = new LinkedList<Integer>(); // 构造一个空列表。
```

- 优点：

在列表的前端或末端添加或移除元素的时间是常数时间。

```
myList.add(10); // \
myList.add(0,2); // | => 常数时间 => O(1)
myList.remove(); //
```

- 缺点：来自 [文档](#)：

索引列表的操作将从列表的开头或结尾遍历，取决于哪个位置离指定索引更近。

诸如以下的操作：

```
myList.get(10); // \
myList.add(11,25); // | => 最坏情况在 O(n/2) 内完成
myList.set(15,35); //
```

RoleList

即将推出

RoleUnresolvedList

即将推出

Stack

即将推出

On coming

CopyOnWriteArrayList

On coming

LinkedList

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, Serializable
```

[LinkedList](#) is implemented by a [doubly-linked list](#) a linked data structure that consists of a set of sequentially linked records called nodes.

Initialize LinkedList of Integer

```
List<Integer> myList = new LinkedList<Integer>(); // Constructs an empty list.
```

- PROS:

Adding or removing an element to the front of the list or to the end has constant time.

```
myList.add(10); // \
myList.add(0,2); // | => constant time => O(1)
myList.remove(); //
```

- CONS: From [documentation](#):

Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Operations such as:

```
myList.get(10); // \
myList.add(11,25); // | => worst case done in O(n/2)
myList.set(15,35); //
```

RoleList

On coming

RoleUnresolvedList

On coming

Stack

On coming

第25.4节：查找两个列表之间的公共元素

假设你有两个列表：A 和 B，你需要找到同时存在于两个列表中的元素。

你可以通过调用方法 `List retainAll()` 来实现。

示例：

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll((Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 })));
    numbersB.addAll((Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 })));

    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);
    List<Integer> numbersC = new ArrayList<>();
    numbersC.addAll(numbersA);
    numbersC.retainAll(numbersB);

    System.out.println("列表 A : " + numbersA);
    System.out.println("列表 B : " + numbersB);
    System.out.println("A 和 B 之间的公共元素: " + numbersC);
}
```

Vector

On coming

Section 25.4: Finding common elements between 2 lists

Suppose you have two lists: A and B, and you need to find the elements that exist in both lists.

You can do it by just invoking the method `List retainAll()`.

Example:

```
public static void main(String[] args) {
    List<Integer> numbersA = new ArrayList<>();
    List<Integer> numbersB = new ArrayList<>();
    numbersA.addAll((Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 })));
    numbersB.addAll((Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 })));

    System.out.println("A: " + numbersA);
    System.out.println("B: " + numbersB);
    List<Integer> numbersC = new ArrayList<>();
    numbersC.addAll(numbersA);
    numbersC.retainAll(numbersB);

    System.out.println("List A : " + numbersA);
    System.out.println("List B : " + numbersB);
    System.out.println("Common elements between A and B: " + numbersC);
}
```

第25.5节：列表元素的原地替换

本例介绍如何替换List元素，同时确保替换元素与被替换元素处于相同的位置。

这可以通过以下方法实现：

- `set(int index, T type)`
- `int indexOf(T type)`

考虑一个ArrayList，包含元素“程序启动！”，“你好，世界！”和“再见，世界！”

```
List<String> strings = new ArrayList<String>();
strings.add("程序启动！");
strings.add("你好，世界！");
strings.add("再见，世界！");
```

如果我们知道要替换元素的索引，可以直接使用`set`，示例如下：

```
strings.set(1, "嗨，世界");
```

如果不知道索引，可以先搜索。例如：

```
int pos = strings.indexOf("Goodbye world!");
if (pos >= 0) {
```

Section 25.5: In-place replacement of a List element

This example is about replacing a `List` element while ensuring that the replacement element is at the same position as the element that is replaced.

This can be done using these methods:

- `set(int index, T type)`
- `int indexOf(T type)`

Consider an `ArrayList` containing the elements "Program starting!", "Hello world!" and "Goodbye world!"

```
List<String> strings = new ArrayList<String>();
strings.add("Program starting!");
strings.add("Hello world!");
strings.add("Goodbye world!");
```

If we know the index of the element we want to replace, we can simply use `set` as follows:

```
strings.set(1, "Hi world");
```

If we don't know the index, we can search for it first. For example:

```
int pos = strings.indexOf("Goodbye world!");
if (pos >= 0) {
```

```
strings.set(pos, "Goodbye cruel world!");
}
```

备注：

1. set操作不会引发ConcurrentModificationException。
2. 对于ArrayList，set操作速度很快 ($O(1)$)，但对于LinkedList则较慢 ($O(N)$)。
3. 在ArrayList或LinkedList上进行indexOf搜索速度较慢 ($O(N)$)。

第25.6节：使列表不可修改

Collections类提供了一种使列表不可修改的方法：

```
List<String> ls = new ArrayList<String>();
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

如果你想要一个只包含一个元素的不可修改列表，可以使用：

```
List<String> unmodifiableList = Collections.singletonList("列表中唯一的字符串");
```

第25.7节：在列表中移动对象

Collections类允许你使用各种方法在列表中移动对象 (ls是List)：

反转列表：

```
Collections.reverse(ls);
```

旋转列表中元素的位置

rotate方法需要一个整数参数。这个参数表示沿着列表移动的位数。下面是一个示例：

```
List<String> ls = new ArrayList<String>();
ls.add(" how");
ls.add(" are");
ls.add(" you?");
ls.add("hello,");
Collections.rotate(ls, 1);

for(String line : ls) System.out.print(line);
System.out.println();
```

这将打印出 "hello, how are you?"

在列表中重新排列元素

使用上面相同的列表，我们可以对列表中的元素进行洗牌：

```
Collections.shuffle(ls);
```

我们也可以给它一个 `java.util.Random` 对象，用于随机地将对象放置在各个位置：

```
Random random = new Random(12);
Collections.shuffle(ls, random);
```

```
strings.set(pos, "Goodbye cruel world!");
}
```

Notes:

1. The set operation will not cause a `ConcurrentModificationException`.
2. The set operation is fast ($O(1)$) for `ArrayList` but slow ($O(N)$) for a `LinkedList`.
3. An `indexOf` search on an `ArrayList` or `LinkedList` is slow ($O(N)$).

Section 25.6: Making a list unmodifiable

The Collections class provides a way to make a list unmodifiable:

```
List<String> ls = new ArrayList<String>();
List<String> unmodifiableList = Collections.unmodifiableList(ls);
```

If you want an unmodifiable list with one item you can use:

```
List<String> unmodifiableList = Collections.singletonList("Only string in the list");
```

Section 25.7: Moving objects around in the list

The Collections class allows for you to move objects around in the list using various methods (ls is the List):

Reversing a list:

```
Collections.reverse(ls);
```

Rotating positions of elements in a list

The rotate method requires an integer argument. This is how many spots to move it along the line by. An example of this is below:

```
List<String> ls = new ArrayList<String>();
ls.add(" how");
ls.add(" are");
ls.add(" you?");
ls.add("hello,");
Collections.rotate(ls, 1);

for(String line : ls) System.out.print(line);
System.out.println();
```

This will print "hello, how are you?"

Shuffling elements around in a list

Using the same list above, we can shuffle the elements in a list:

```
Collections.shuffle(ls);
```

We can also give it a `java.util.Random` object that it uses to randomly place objects in spots:

```
Random random = new Random(12);
Collections.shuffle(ls, random);
```

第25.8节：创建、添加和删除 ArrayList中的元素

ArrayList 是Java中内置的数据结构之一。它是一个动态数组（不需要预先声明数据结构的大小），用于存储元素（对象）。

它继承了 AbstractList 类并实现了 List 接口。一个 ArrayList 可以包含重复元素，并且保持插入顺序。需要注意的是，ArrayList 类是非同步的，因此在处理 ArrayList 的并发时应当小心。由于数组基于索引工作，ArrayList 允许随机访问。由于删除元素时经常发生的移动操作，ArrayList 中的操作速度较慢。

可以按如下方式创建一个 ArrayList：

```
List<T> myArrayList = new ArrayList<>();
```

其中T（泛型）是将存储在ArrayList中的类型。

ArrayList 的类型可以是任何对象。类型不能是原始类型（请使用它们的包装类代替）。

要向 ArrayList 添加元素，使用 add() 方法：

```
myArrayList.add(element);
```

或者向指定索引添加元素：

```
myArrayList.add(index, element); // 元素的索引应为 int 类型（从 0 开始）
```

要从 ArrayList 中移除元素，使用 remove() 方法：

```
myArrayList.remove(element);
```

或者从指定索引移除元素：

```
myArrayList.remove(index); // 元素的索引应为 int 类型（从 0 开始）
```

第 25.9 节：创建列表

给你的列表指定类型

要创建列表，你需要一个类型（任何类，例如String）。这就是你的List的类型。该List只会存储指定类型的对象。例如：

```
List<String> strings;
```

可以存储"string1"、"hello world!"、"goodbye"等，但不能存储9.2，然而：

```
List<Double> doubles;
```

可以存储9.2，但不能存储"hello world!"。

初始化你的列表

如果你尝试向上述列表添加内容，会得到NullPointerException，因为strings和doubles

Section 25.8: Creating, Adding and Removing element from an ArrayList

ArrayList is one of the inbuilt data structures in Java. It is a dynamic array (where the size of the data structure not needed to be declared first) for storing elements (Objects).

It extends AbstractList class and implements List interface. An ArrayList can contain duplicate elements where it maintains insertion order. It should be noted that the class ArrayList is non-synchronized, so care should be taken when handling concurrency with ArrayList. ArrayList allows random access because array works at the index basis. Manipulation is slow in ArrayList because of shifting that often occurs when an element is removed from the array list.

An ArrayList can be created as follows:

```
List<T> myArrayList = new ArrayList<>();
```

Where T (Generics) is the type that will be stored inside ArrayList.

The type of the ArrayList can be any Object. The type can't be a primitive type (use their [wrapper classes](#) instead).

To add an element to the ArrayList, use add() method:

```
myArrayList.add(element);
```

Or to add item to a certain index:

```
myArrayList.add(index, element); //index of the element should be an int (starting from 0)
```

To remove an item from the ArrayList, use the remove() method:

```
myArrayList.remove(element);
```

Or to remove an item from a certain index:

```
myArrayList.remove(index); //index of the element should be an int (starting from 0)
```

Section 25.9: Creating a List

Giving your list a type

To create a list you need a type (any class, e.g. String). This is the type of your List. The List will only store objects of the specified type. For example:

```
List<String> strings;
```

Can store "string1", "hello world!", "goodbye", etc, but it can't store 9.2, however:

```
List<Double> doubles;
```

Can store 9.2, but not "hello world!".

Initialising your list

If you try to add something to the lists above you will get a NullPointerException, because strings and doubles

都等于null !

有两种方法可以初始化列表：

选项1：使用实现了List接口的类

List 是一个接口，这意味着它没有构造函数，而是一个类必须重写的方法集合。
ArrayList 是最常用的List，虽然LinkedList也很常见。所以我们这样初始化列表：

```
List<String> strings = new ArrayList<String>();
```

或者

```
List<String> strings = new LinkedList<String>();
```

版本 ≥ Java SE 7

从 Java SE 7 开始，你可以使用diamond operator (菱形操作符)：

```
List<String> strings = new ArrayList<>();
```

或者

```
List<String> strings = new LinkedList<>();
```

选项 2：使用 Collections 类

Collections 类提供了两个有用的方法，用于创建没有List变量的列表：

- emptyList()：返回一个空列表。
- singletonList(T)：创建一个类型为T的列表并添加指定的元素。

以及一个使用现有List来填充数据的方法：

- addAll(L, T...)：将所有指定的元素添加到作为第一个参数传入的列表中。

示例：

```
import java.util.List; import java.util.Collections; List<Integer> l = Collections.emptyList(); List<Integer> l1 = Collections.singletonList(42); Collections.addAll(l1, 1, 2, 3);
```

第25.10节：位置访问操作

List API有八个用于位置访问操作的方法：

- add(T type)
- add(int index, T type)
- remove(Object o)
- remove(int index)
- get(int index)
- set(int index, E element)
- int indexOf(Object o)
- int lastIndexOf(Object o)

所以，如果我们有一个List：

both equal null!

There are two ways to initialise a list:

Option 1: Use a class that implements List

List is an interface, which means that does not have a constructor, rather methods that a class must override.
ArrayList is the most commonly used List, though LinkedList is also common. So we initialise our list like this:

```
List<String> strings = new ArrayList<String>();
```

或者

```
List<String> strings = new LinkedList<String>();
```

Version ≥ Java SE 7

Starting from Java SE 7, you can use a diamond operator:

```
List<String> strings = new ArrayList<>();
```

或者

```
List<String> strings = new LinkedList<>();
```

Option 2: Use the Collections class

The Collections class provides two useful methods for creating Lists without a List variable:

- emptyList(): returns an empty list.
- singletonList(T): creates a list of type T and adds the element specified.

And a method which uses an existing List to fill data in:

- addAll(L, T...): adds all the specified elements to the list passed as the first parameter.

Examples:

```
import java.util.List; import java.util.Collections; List<Integer> l = Collections.emptyList(); List<Integer> l1 = Collections.singletonList(42); Collections.addAll(l1, 1, 2, 3);
```

Section 25.10: Positional Access Operations

The List API has eight methods for positional access operations:

- add(T type)
- add(int index, T type)
- remove(Object o)
- remove(int index)
- get(int index)
- set(int index, E element)
- int indexOf(Object o)
- int lastIndexOf(Object o)

So, if we have a List:

```
List<String> strings = new ArrayList<String>();
```

如果我们想向其中添加字符串“Hello world!”和“Goodbye world!”，我们会这样做：

```
strings.add("你好，世界！");  
strings.add("再见，世界！");
```

我们的列表将包含这两个元素。现在假设我们想在列表的前面添加“程序开始！”，我们可以这样做：

```
strings.add(0, "Program starting!");
```

注意：第一个元素的索引是0。

现在，如果我们想删除“再见，世界！”这一行，可以这样做：

```
strings.remove("Goodbye world!");
```

如果我们想删除第一行（在本例中是“程序开始！”），可以这样做：

```
strings.remove(0);
```

注意：

1. 添加和删除列表元素会修改列表，这可能导致ConcurrentModificationException
如果列表正在被并发迭代。
2. 添加和删除元素的时间复杂度可以是O(1)或O(N)，这取决于列表类、所用方法以及无论是在列表的开头、结尾还是中间添加/删除元素。

为了在指定位置检索列表中的元素，可以使用List API的E get(int index);方法。例如：

```
strings.get(0);
```

将返回列表的第一个元素。

你可以使用set(int index, E element);方法替换指定位置的任何元素。例如：

```
strings.set(0, "This is a replacement");
```

这将把字符串 “This is a replacement” 设置为列表的第一个元素。

注意：set方法会覆盖位置0的元素。它不会在位置0添加新的字符串并将旧元素推到位置1。

int indexOf(Object o);返回传入对象首次出现的位置。如果列表中没有该对象，则返回-1。接着前面的例子，如果你调用：

```
strings.indexOf("This is a replacement")
```

预期返回0，因为我们在列表的第0个位置设置了字符串“This is a replacement”。如果列表中存在多个匹配项，当调用int indexOf(Object o);时，如前所述

```
List<String> strings = new ArrayList<String>();
```

And we wanted to add the strings "Hello world!" and "Goodbye world!" to it, we would do it as such:

```
strings.add("Hello world!");  
strings.add("Goodbye world!");
```

And our list would contain the two elements. Now lets say we wanted to add "Program starting!" at the **front** of the list. We would do this like this:

```
strings.add(0, "Program starting!");
```

NOTE: The first element is 0.

Now, if we wanted to remove the "Goodbye world!" line, we could do it like this:

```
strings.remove("Goodbye world!");
```

And if we wanted to remove the first line (which in this case would be "Program starting!", we could do it like this:

```
strings.remove(0);
```

Note:

1. Adding and removing list elements modify the list, and this can lead to a **ConcurrentModificationException** if the list is being iterated concurrently.
2. Adding and removing elements can be O(1) or O(N) depending on the list class, the method used, and whether you are adding / removing an element at the start, the end, or in the middle of the list.

In order to retrieve an element of the list at a specified position you can use the E `get(int index)` ; method of the List API. For example:

```
strings.get(0);
```

will return the first element of the list.

You can replace any element at a specified position by using the set(`int index, E element`) . For example:

```
strings.set(0, "This is a replacement");
```

This will set the String "This is a replacement" as the first element of the list.

Note: The set method will overwrite the element at the position 0. It will not add the new String at the position 0 and push the old one to the position 1.

The `int indexOf(Object o)` ; returns the position of the first occurrence of the object passed as argument. If there are no occurrences of the object in the list then the -1 value is returned. In continuation of the previous example if you call:

```
strings.indexOf("This is a replacement")
```

the 0 is expected to be returned as we set the String "This is a replacement" in the position 0 of our list. In case where there are more than one occurrence in the list when `int indexOf(Object o)` ; is called then as mentioned

将返回第一次出现的索引。通过调用int lastIndexOf(Object o)可以获取列表中最后一次出现的索引。所以如果我们再添加另一个 "This is a replacement" :

```
strings.add("This is a replacement");
strings.lastIndexOf("This is a replacement");
```

这次将返回1，而不是0；

第25.11节：遍历列表中的元素

举个例子，假设我们有一个类型为String的列表，包含四个元素："hello, ", "how ", "are ", "you?"

遍历每个元素的最佳方式是使用for-each循环：

```
public void printEachElement(List<String> list){
    for(String s : list){
        System.out.println(s);
    }
}
```

输出结果将是：

```
hello,
how
are
you?
```

要将它们全部打印在同一行，可以使用 StringBuilder：

```
public void printAsLine(List<String> list){
    StringBuilder builder = new StringBuilder();
    for(String s : list){
        builder.append(s);
    }
    System.out.println(builder.toString());
}
```

将打印：

```
hello, how are you?
```

或者，你可以使用元素索引（如“从 ArrayList 访问第 i 个元素”中所述）来遍历列表。警告：这种方法对链表效率较低。

第 25.12 节：从列表 B 中移除存在于列表 A 中的元素

假设你有两个列表 A 和 B，想要从B中移除所有在A中存在的元素，此时的方法是

```
List.removeAll(Collection c);
```

#示例：

the index of the first occurrence will be returned. By calling the int lastIndexOf(Object o) you can retrieve the index of the last occurrence in the list. So if we add another "This is a replacement":

```
strings.add("This is a replacement");
strings.lastIndexOf("This is a replacement");
```

This time the 1 will be returned and not the 0;

Section 25.11: Iterating over elements in a list

For the example, lets say that we have a List of type String that contains four elements: "hello, ", "how ", "are ", "you?"

The best way to iterate over each element is by using a for-each loop:

```
public void printEachElement(List<String> list){
    for(String s : list){
        System.out.println(s);
    }
}
```

Which would print:

```
hello,
how
are
you?
```

To print them all in the same line, you can use a StringBuilder:

```
public void printAsLine(List<String> list){
    StringBuilder builder = new StringBuilder();
    for(String s : list){
        builder.append(s);
    }
    System.out.println(builder.toString());
}
```

Will print:

```
hello, how are you?
```

Alternatively, you can use element indexing (as described in Accessing element at ith Index from ArrayList) to iterate a list. Warning: this approach is inefficient for linked lists.

Section 25.12: Removing elements from list B that are present in the list A

Lets suppose you have 2 Lists A and B, and you want to remove from B all the elements that you have in A the method in this case is

```
List.removeAll(Collection c);
```

#Example:

```
public static void main(String[] args) {  
    List<Integer> numbersA = new ArrayList<>();  
    List<Integer> numbersB = new ArrayList<>();  
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));  
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));  
    System.out.println("A: " + numbersA);  
    System.out.println("B: " + numbersB);  
  
    numbersB.removeAll(numbersA);  
    System.out.println("B cleared: " + numbersB);  
}
```

这将打印

```
A: [1, 3, 4, 7, 5, 2]  
B: [13, 32, 533, 3, 4, 2]  
B cleared: [13, 32, 533]
```

```
public static void main(String[] args) {  
    List<Integer> numbersA = new ArrayList<>();  
    List<Integer> numbersB = new ArrayList<>();  
    numbersA.addAll(Arrays.asList(new Integer[] { 1, 3, 4, 7, 5, 2 }));  
    numbersB.addAll(Arrays.asList(new Integer[] { 13, 32, 533, 3, 4, 2 }));  
    System.out.println("A: " + numbersA);  
    System.out.println("B: " + numbersB);  
  
    numbersB.removeAll(numbersA);  
    System.out.println("B cleared: " + numbersB);  
}
```

this will print

```
A: [1, 3, 4, 7, 5, 2]  
B: [13, 32, 533, 3, 4, 2]  
B cleared: [13, 32, 533]
```

第26章：集合

第26.1节：初始化

集合（Set）是一种不能包含重复元素的集合（Collection）。它模拟了数学中的集合抽象。

集合的实现存在于多个类中，如`HashSet`、`TreeSet`、`LinkedHashSet`。

例如：

HashSet：

```
Set<T> set = new HashSet<T>();
```

这里T可以是`String`、`Integer`或任何其他`object`。`HashSet`允许 $O(1)$ 的快速查找，但不会对添加的数据进行排序，并且会丢失元素的插入顺序。

TreeSet：

它以排序的方式存储数据，牺牲了一些速度，基本操作的时间复杂度为 $O(\lg(n))$ 。它不维护元素的插入顺序。

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet：

它是`HashSet`的链表实现，可以按添加顺序迭代元素。其内容不提供排序。基本操作为 $O(1)$ ，但维护底层链表的开销比`HashSet`更高。

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

第26.2节：集合基础

集合是什么？

集合是一种数据结构，包含一组元素，其重要特性是集合中没有两个元素是相等的。

集合的类型：

1. `HashSet`：由哈希表支持的集合（实际上是一个`HashMap`实例）
2. `Linked HashSet`：由哈希表和链表支持的集合，具有可预测的迭代顺序
3. `TreeSet`：基于`TreeMap`的`NavigableSet`实现。

创建集合

```
Set<Integer> set = new HashSet<Integer>(); // 创建一个空的整数集合
```

```
t = new LinkedHashSet<Integer>(); // 创建一个空的整数集合，具有可预测的迭代顺序
```

向集合中添加元素

Chapter 26: Sets

Section 26.1: Initialization

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction.

`Set` have its implementation in various classes like `HashSet`, `TreeSet`, `LinkedHashSet`.

For example:

HashSet:

```
Set<T> set = new HashSet<T>();
```

Here T can be `String`, `Integer` or any other `object`. `HashSet` allows for quick lookup of $O(1)$ but does not sort the data added to it and loses the insertion order of items.

TreeSet:

It stores data in a sorted manner sacrificing some speed for basic operations which take $O(\lg(n))$. It does not maintain the insertion order of items.

```
TreeSet<T> sortedSet = new TreeSet<T>();
```

LinkedHashSet:

It is a linked list implementation of `HashSet`. Once can iterate over the items in the order they were added. Sorting is not provided for its contents. $O(1)$ basic operations are provided, however there is higher cost than `HashSet` in maintaining the backing linked list.

```
LinkedHashSet<T> linkedhashset = new LinkedHashSet<T>();
```

Section 26.2: Basics of Set

What is a Set?

A set is a data structure which contains a set of elements with an important property that no two elements in the set are equal.

Types of Set:

1. **HashSet**: A set backed by a hash table (actually a `HashMap` instance)
2. **Linked HashSet**: A Set backed by Hash table and linked list, with predictable iteration order
3. **TreeSet**: A `NavigableSet` implementation based on a `TreeMap`.

Creating a set

```
Set<Integer> set = new HashSet<Integer>(); // Creates an empty Set of Integers
```

```
Set<Integer> linkedHashSet = new LinkedHashSet<Integer>(); //Creates a empty Set of Integers, with predictable iteration order
```

Adding elements to a Set

可以使用add()方法向集合中添加元素

```
set.add(12); // - 向集合中添加元素12  
set.add(13); // - 向集合中添加元素13
```

执行此方法后我们的集合：

```
set = [12,13]
```

删除集合中的所有元素

```
set.clear(); //从集合中移除所有对象。
```

此时集合将为：

```
set = []
```

检查元素是否属于集合

可以使用contains()方法检查元素是否存在与集合中

```
set.contains(0); //如果指定对象是集合中的元素，则返回true。
```

输出：False

检查集合是否为空

isEmpty() 方法可用于检查集合是否为空。

```
set.isEmpty(); //如果集合没有元素则返回 true
```

输出： True

从集合中移除元素

```
set.remove(0); // 从集合中移除指定对象的第一次出现
```

检查集合的大小

```
set.size(); // 返回集合中元素的数量
```

输出： 0

第26.3节：集合的类型和用法

通常，集合是一种存储唯一值的集合类型。唯一性由 equals() 和 hashCode() 方法决定。

排序由集合类型决定。

HashSet - 随机排序

版本 ≥ Java SE 7

```
Set<String> set = new HashSet<>();  
set.add("Banana");
```

Elements can be added to a set using the add() method

```
set.add(12); // - Adds element 12 to the set  
set.add(13); // - Adds element 13 to the set
```

Our set after executing this method:

```
set = [12,13]
```

Delete all the elements of a Set

```
set.clear(); //Removes all objects from the collection.
```

After this set will be:

```
set = []
```

Check whether an element is part of the Set

Existence of an element in the set can be checked using the contains() method

```
set.contains(0); //Returns true if a specified object is an element within the set.
```

Output: False

Check whether a Set is empty

isEmpty() method can be used to check whether a Set is empty.

```
set.isEmpty(); //Returns true if the set has no elements
```

Output: True

Remove an element from the Set

```
set.remove(0); // Removes first occurrence of a specified object from the collection
```

Check the Size of the Set

```
set.size(); //Returns the number of elements in the collection
```

Output: 0

Section 26.3: Types and Usage of Sets

Generally, sets are a type of collection which stores unique values. Uniqueness is determined by the equals() and hashCode() methods.

Sorting is determined by the type of set.

HashSet - Random Sorting

Version ≥ Java SE 7

```
Set<String> set = new HashSet<>();  
set.add("Banana");
```

```
set.add("Banana");
set.add("Apple");
set.add("Strawberry");

// 集合元素: ["Strawberry", "Banana", "Apple"]
```

LinkedHashSet - 插入顺序

版本 ≥ Java SE 7

```
Set<String> set = new LinkedHashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");
```

// 集合元素: ["Banana", "Apple", "Strawberry"]

TreeSet - 通过 compareTo() 或 Comparator

版本 ≥ Java SE 7

```
Set<String> set = new TreeSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");
```

// 设置元素: ["苹果", "香蕉", "草莓"]

版本 ≥ Java SE 7

```
Set<String> set = new TreeSet<> ((string1, string2) -> string2.compareTo(string1));
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");
```

// 集合元素: ["Strawberry", "Banana", "Apple"]

第26.4节：从现有的Set创建列表

使用新的List

```
List<String> list = new ArrayList<String>(listOfElements);
```

使用List.addAll()方法

```
Set<String> set = new HashSet<String>();
set.add("foo");
set.add("boo");

List<String> list = new ArrayList<String>();
list.addAll(set);
```

使用Java 8 Stream API

```
List<String> list = set.stream().collect(Collectors.toList());
```

第26.5节：使用Set去除重复项

假设你有一个集合elements，想要创建另一个包含相同元素但去除所有重复项的集合：

```
set.add("Banana");
set.add("Apple");
set.add("Strawberry");
```

// Set Elements: ["Strawberry", "Banana", "Apple"]

LinkedHashSet - Insertion Order

Version ≥ Java SE 7

```
Set<String> set = new LinkedHashSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");
```

// Set Elements: ["Banana", "Apple", "Strawberry"]

TreeSet - By compareTo() or Comparator

Version ≥ Java SE 7

```
Set<String> set = new TreeSet<> ();
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");
```

// Set Elements: ["Apple", "Banana", "Strawberry"]

Version ≥ Java SE 7

```
Set<String> set = new TreeSet<> ((string1, string2) -> string2.compareTo(string1));
set.add("Banana");
set.add("Banana");
set.add("Apple");
set.add("Strawberry");
```

// Set Elements: ["Strawberry", "Banana", "Apple"]

Section 26.4: Create a list from an existing Set

Using a new List

```
List<String> list = new ArrayList<String>(listOfElements);
```

Using List.addAll() method

```
Set<String> set = new HashSet<String>();
set.add("foo");
set.add("boo");

List<String> list = new ArrayList<String>();
list.addAll(set);
```

Using Java 8 Steam API

```
List<String> list = set.stream().collect(Collectors.toList());
```

Section 26.5: Eliminating duplicates using Set

Suppose you have a collection elements, and you want to create another collection containing the same elements but with all **duplicates eliminated**:

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

示例：

```
List<String> names = new ArrayList<>(  
    Arrays.asList("John", "Marco", "Jenny", "Emily", "Jenny", "Emily", "John"));  
Set<String> noDuplicates = new HashSet<>(names);  
System.out.println("noDuplicates = " + noDuplicates);
```

输出：

```
noDuplicates = [Marco, Emily, John, Jenny]
```

第26.6节：声明带有初始值的HashSet

你可以创建一个继承自HashSet的新类：

```
Set<String> h = new HashSet<String>() {{  
    add("a");  
    add("b");  
}};
```

一行解决方案：

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

使用 guava：

```
Sets.newHashSet("a", "b", "c")
```

使用 Streams：

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

```
Collection<Type> noDuplicates = new HashSet<Type>(elements);
```

Example:

```
List<String> names = new ArrayList<>(  
    Arrays.asList("John", "Marco", "Jenny", "Emily", "Jenny", "Emily", "John"));  
Set<String> noDuplicates = new HashSet<>(names);  
System.out.println("noDuplicates = " + noDuplicates);
```

Output:

```
noDuplicates = [Marco, Emily, John, Jenny]
```

Section 26.6: Declaring a HashSet with values

You can create a new class that inherits from HashSet:

```
Set<String> h = new HashSet<String>() {{  
    add("a");  
    add("b");  
}};
```

One line solution:

```
Set<String> h = new HashSet<String>(Arrays.asList("a", "b"));
```

Using guava:

```
Sets.newHashSet("a", "b", "c")
```

Using Streams:

```
Set<String> set3 = Stream.of("a", "b", "c").collect(toSet());
```

第27章：List 与 Set

List 和 Set 集合在顶层的区别是什么，以及在 Java 中何时选择使用 List，何时选择使用 Set

第27.1节：List 与 Set

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
public class SetAndListExample
{
    public static void main( String[] args )
    {
        System.out.println("List example ....");
        List list = new ArrayList();
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("1");
        for (String temp : list){
            System.out.println(temp);
        }

        System.out.println("Set example ....");
        Set<String> set = new HashSet<String>();
        set.add("1");
        set.add("2");
        set.add("3");
        set.add("4");
        set.add("1");
        set.add("2");
        set.add("5");

        for (String temp : set){
            System.out.println(temp);
        }
    }
}
```

Output List example 1 2 3 4 1 Set example 3 2 10 5 4

Chapter 27: List vs Set

What are differences between List and Set collection at the top level and How to choose when to use List in java and when to use Set in Java

Section 27.1: List vs Set

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
public class SetAndListExample
{
    public static void main( String[] args )
    {
        System.out.println("List example ....");
        List list = new ArrayList();
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("1");
        for (String temp : list){
            System.out.println(temp);
        }

        System.out.println("Set example ....");
        Set<String> set = new HashSet<String>();
        set.add("1");
        set.add("2");
        set.add("3");
        set.add("4");
        set.add("1");
        set.add("2");
        set.add("5");

        for (String temp : set){
            System.out.println(temp);
        }
    }
}
```

Output List example 1 2 3 4 1 Set example 3 2 10 5 4

第28章：映射

java.util.Map接口表示键与其值之间的映射。映射不能包含重复的键；且每个键最多映射到一个值。

由于Map是一个接口，因此您需要实例化该接口的具体实现才能使用它；有多种Map实现，最常用的是java.util.HashMap和java.util.TreeMap

第28.1节：高效迭代Map条目

本节提供了十个独特示例实现的代码和基准测试，这些实现迭代一个Map<Integer, Integer>的条目并计算Integer值的总和。所有示例的算法复杂度均为 $\Theta(n)$ ，然而基准测试仍然有助于提供关于哪些实现更适合“真实世界”环境的效率见解。

1. 使用Iterator和Map.Entry的实现

```
Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry<Integer, Integer> pair = it.next();
    sum += pair.getKey() + pair.getValue();
}
```

2. 使用for和Map.Entry的实现

```
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    sum += pair.getKey() + pair.getValue();
}
```

3. 使用Map.forEach (Java 8及以上) 实现

```
map.forEach((k, v) -> sum[0] += k + v);
```

4. 使用Map.keySet配合for的实现

```
for (Integer key : map.keySet()) {
    sum += key + map.get(key);
}
```

5. 使用Map.keySet配合Iterator的实现

```
Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    Integer key = it.next();
    sum += key + map.get(key);
}
```

6. 使用for配合Iterator和Map.Entry的实现

```
for (Iterator<Map.Entry<Integer, Integer>> entries =
     map.entrySet().iterator(); entries.hasNext(); ) {
    Map.Entry<Integer, Integer> entry = entries.next();
    sum += entry.getKey() + entry.getValue();
}
```

7. 使用Stream.forEach (Java 8及以上) 实现

Chapter 28: Maps

The [java.util.Map interface](#) represents a mapping between keys and their values. A map cannot contain duplicate keys; and each key can map to at most one value.

Since Map is an interface, then you need to instantiate a concrete implementation of that interface in order to use it; there are several Map implementations, and mostly used are the [java.util.HashMap](#) and [java.util.TreeMap](#)

Section 28.1: Iterating Map Entries Efficiently

This section provides code and benchmarks for ten unique example implementations which iterate over the entries of a Map<Integer, Integer> and generate the sum of the Integer values. All of the examples have an algorithmic complexity of $\Theta(n)$, however, the benchmarks are still useful for providing insight on which implementations are more efficient in a "real world" environment.

1. Implementation using [Iterator](#) with [Map.Entry](#)

```
Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
while (it.hasNext()) {
    Map.Entry<Integer, Integer> pair = it.next();
    sum += pair.getKey() + pair.getValue();
}
```

2. Implementation using [for](#) with [Map.Entry](#)

```
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    sum += pair.getKey() + pair.getValue();
}
```

3. Implementation using [Map.forEach](#) (Java 8+)

```
map.forEach((k, v) -> sum[0] += k + v);
```

4. Implementation using [Map.keySet](#) with [for](#)

```
for (Integer key : map.keySet()) {
    sum += key + map.get(key);
}
```

5. Implementation using [Map.keySet](#) with [Iterator](#)

```
Iterator<Integer> it = map.keySet().iterator();
while (it.hasNext()) {
    Integer key = it.next();
    sum += key + map.get(key);
}
```

6. Implementation using [for](#) with [Iterator](#) and [Map.Entry](#)

```
for (Iterator<Map.Entry<Integer, Integer>> entries =
     map.entrySet().iterator(); entries.hasNext(); ) {
    Map.Entry<Integer, Integer> entry = entries.next();
    sum += entry.getKey() + entry.getValue();
}
```

7. Implementation using [Stream.forEach](#) (Java 8+)

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. 使用Stream.forEach结合Stream.parallel的实现 (Java 8及以上)

```
map.entrySet()
    .stream()
    .parallel()
    .forEach(e -> sum += e.getKey() + e.getValue());
```

9. 使用Apache Collections中的IterableMap实现

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. 使用Eclipse Collections中的MutableMap实现

```
mutableMap.forEachKeyValue((key, value) -> {
    sum += key + value;
});
```

性能测试 (代码可在[Github](#)获取)

测试环境：Windows 8.1 64位, Intel i7-4790 3.60GHz, 16 GB

1. 10次试验的平均性能 (100个元素) 最佳：308±21 ns/次操作

基准测试	分数	误差	单位
test3_UsingForEachAndJava8	308	± 21	ns/op
test10_UsingEclipseMutableMap	309	± 9	ns/op
test1_UsingWhileAndMapEntry	380	± 14	ns/op
test6_UsingForAndIterator	387	± 16	ns/op
test2_UsingForEachAndMapEntry	391	± 23	ns/op
test7_UsingJava8StreamAPI	510	± 14	ns/op
test9_UsingApacheIterableMap	524	± 8	ns/op
test4_UsingKeySetAndForEach	816	± 26	ns/op
test5_UsingKeySetAndIterator	863	± 25	ns/op
test8_UsingJava8StreamAPIParallel	5552	± 185	ns/op

2. 10次试验的平均性能 (10000个元素) 最佳：37.606±0.790 μs/次操作

基准测试	分数	误差	单位
test10_UsingEclipseMutableMap	37606	± 790	ns/op
test3_UsingForEachAndJava8	50368	± 887	ns/op
test6_UsingForAndIterator	50332	± 507	ns/op
test2_UsingForEachAndMapEntry	51406	± 1032	ns/op
test1_UsingWhileAndMapEntry	52538	± 2431	ns/op
test7_UsingJava8StreamAPI	54464	± 712	ns/op
test4_UsingKeySetAndForEach	79016	± 25345	ns/op
test5_UsingKeySetAndIterator	91105	± 10220	ns/op
test8_UsingJava8StreamAPIParallel	112511	± 365	ns/op
test9_UsingApacheIterableMap	125714	± 1935	ns/op

3. 10次试验的平均性能 (100000个元素) 最佳：1184.767±332.968 μs/操作

基准测试	分数	误差	单位
------	----	----	----

```
map.entrySet().stream().forEach(e -> sum += e.getKey() + e.getValue());
```

8. Implementation using [Stream.forEach](#) with [Stream.parallel](#) (Java 8+)

```
map.entrySet()
    .stream()
    .parallel()
    .forEach(e -> sum += e.getKey() + e.getValue());
```

9. Implementation using [IterableMap](#) from [Apache Collections](#)

```
MapIterator<Integer, Integer> mit = iterableMap.mapIterator();
while (mit.hasNext()) {
    sum += mit.next() + it.getValue();
}
```

10. Implementation using [MutableMap](#) from [Eclipse Collections](#)

```
mutableMap.forEachKeyValue((key, value) -> {
    sum += key + value;
});
```

Performance Tests (Code available on [Github](#))

Test Environment: Windows 8.1 64-bit, Intel i7-4790 3.60GHz, 16 GB

1. Average Performance of 10 Trials (100 elements) Best: 308±21 ns/op

Benchmark	Score	Error	Units
test3_UsingForEachAndJava8	308	± 21	ns/op
test10_UsingEclipseMutableMap	309	± 9	ns/op
test1_UsingWhileAndMapEntry	380	± 14	ns/op
test6_UsingForAndIterator	387	± 16	ns/op
test2_UsingForEachAndMapEntry	391	± 23	ns/op
test7_UsingJava8StreamAPI	510	± 14	ns/op
test9_UsingApacheIterableMap	524	± 8	ns/op
test4_UsingKeySetAndForEach	816	± 26	ns/op
test5_UsingKeySetAndIterator	863	± 25	ns/op
test8_UsingJava8StreamAPIParallel	5552	± 185	ns/op

2. Average Performance of 10 Trials (10000 elements) Best: 37.606±0.790 μs/op

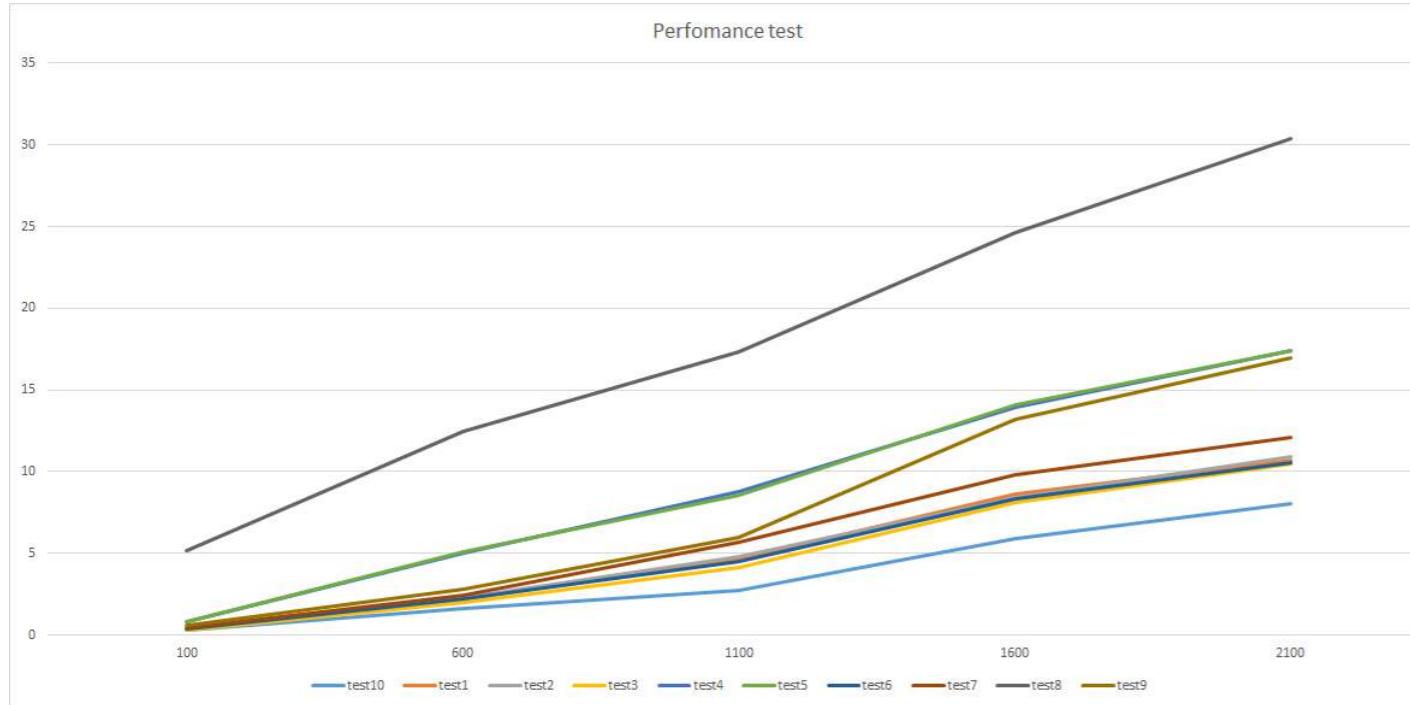
Benchmark	Score	Error	Units
test10_UsingEclipseMutableMap	37606	± 790	ns/op
test3_UsingForEachAndJava8	50368	± 887	ns/op
test6_UsingForAndIterator	50332	± 507	ns/op
test2_UsingForEachAndMapEntry	51406	± 1032	ns/op
test1_UsingWhileAndMapEntry	52538	± 2431	ns/op
test7_UsingJava8StreamAPI	54464	± 712	ns/op
test4_UsingKeySetAndForEach	79016	± 25345	ns/op
test5_UsingKeySetAndIterator	91105	± 10220	ns/op
test8_UsingJava8StreamAPIParallel	112511	± 365	ns/op
test9_UsingApacheIterableMap	125714	± 1935	ns/op

3. Average Performance of 10 Trials (100000 elements) Best: 1184.767±332.968 μs/op

Benchmark	Score	Error	Units
-----------	-------	-------	-------

test1_使用While和MapEntry	1184.767	± 332.968	μs/操作
test10_使用EclipseMutableMap	1191.735	± 304.273	μs/操作
test2_使用ForEach和MapEntry	1205.815	± 366.043	μs/操作
test6_使用For和Iterator	1206.873	± 367.272	μs/操作
test8_使用Java8StreamAPI并行	1485.895	± 233.143	μs/操作
test5_使用KeySet和Iterator	1540.281	± 357.497	μs/操作
test4_使用KeySet和ForEach	1593.342	± 294.417	μs/操作
test3_使用ForEach和Java8	1666.296	± 126.443	μs/操作
test7_使用Java8StreamAPI	1706.676	± 436.867	μs/操作
test9_使用ApacheIterableMap	3289.866	± 1445.564	μs/操作

4. 各映射大小对应的性能变化比较



x: 映射大小
f(x): 基准测试分数 (μs/操作)

	100	600	1100	1600	2100
10	0.333	1.631	2.752	5.937	8.024
3	0.309	1.971	4.147	8.147	10.473
6	0.372	2.190	4.470	8.322	10.531
1	0.405	2.237	4.616	8.645	10.707
2	0.376	2.267	4.809	8.403	10.910
f(x)	0.473	2.448	5.668	9.790	12.125
9	0.565	2.830	5.952	13.22	16.965
4	0.808	5.012	8.813	13.939	17.407
5	0.81	5.104	8.533	14.064	17.422
8	5.173	12.499	17.351	24.671	30.403

第28.2节：HashMap的使用

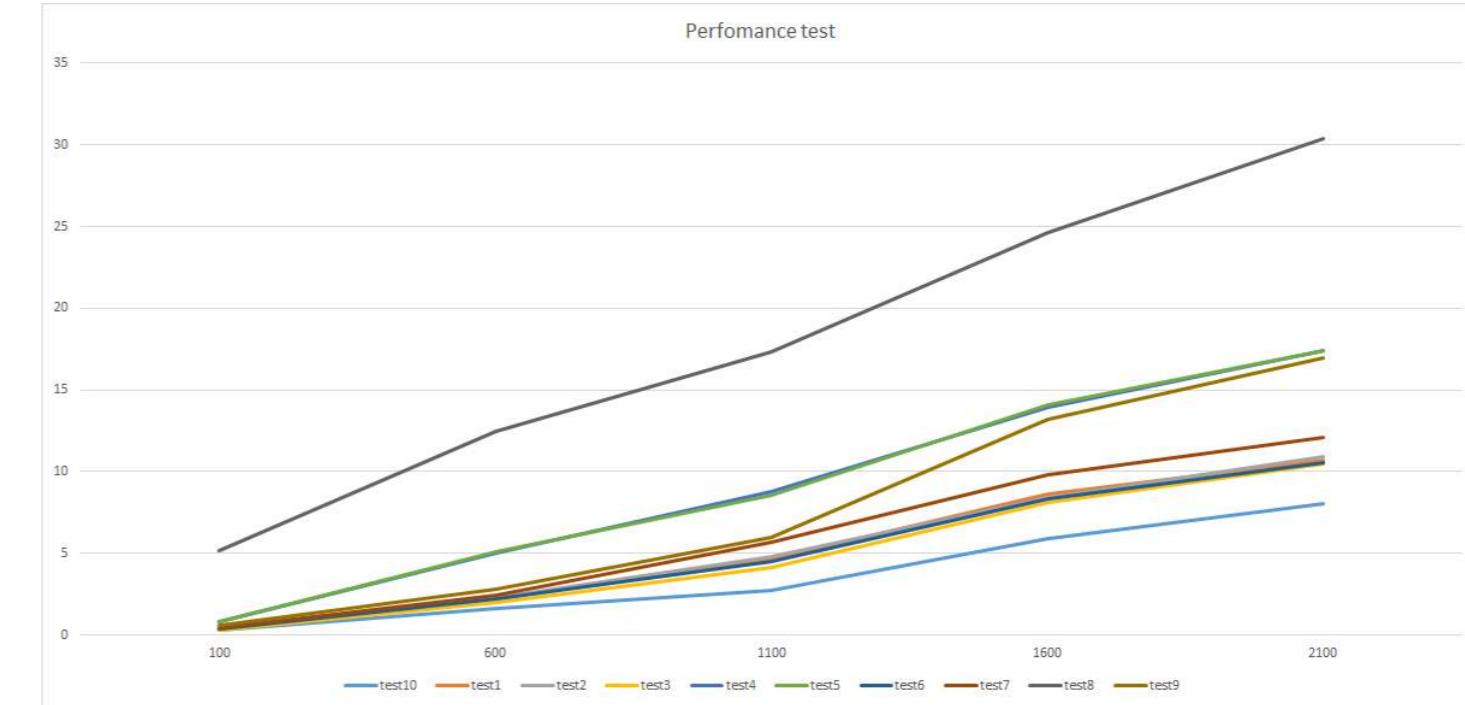
HashMap是Map接口的一个实现，提供了一种以键值对形式存储数据的数据结构。

1. 声明HashMap

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

test1_UsingWhileAndMapEntry	1184.767	± 332.968	μs/op
test10_UsingEclipseMutableMap	1191.735	± 304.273	μs/op
test2_UsingForEachAndMapEntry	1205.815	± 366.043	μs/op
test6_UsingForAndIterator	1206.873	± 367.272	μs/op
test8_UsingJava8StreamAPIParallel	1485.895	± 233.143	μs/op
test5_UsingKeySetAndIterator	1540.281	± 357.497	μs/op
test4_UsingKeySetAndForEach	1593.342	± 294.417	μs/op
test3_UsingForEachAndJava8	1666.296	± 126.443	μs/op
test7_UsingJava8StreamAPI	1706.676	± 436.867	μs/op
test9_UsingApacheIterableMap	3289.866	± 1445.564	μs/op

4. A Comparison of Performance Variations Respective to Map Size



x: Size of Map
f(x): Benchmark Score (μs/op)

	100	600	1100	1600	2100
10	0.333	1.631	2.752	5.937	8.024
3	0.309	1.971	4.147	8.147	10.473
6	0.372	2.190	4.470	8.322	10.531
1	0.405	2.237	4.616	8.645	10.707
2	0.376	2.267	4.809	8.403	10.910
f(x)	0.473	2.448	5.668	9.790	12.125
9	0.565	2.830	5.952	13.22	16.965
4	0.808	5.012	8.813	13.939	17.407
5	0.81	5.104	8.533	14.064	17.422
8	5.173	12.499	17.351	24.671	30.403

Section 28.2: Usage of HashMap

HashMap is an implementation of the Map interface that provides a Data Structure to store data in Key-Value pairs.

1. Declaring HashMap

```
Map<KeyType, ValueType> myMap = new HashMap<KeyType, ValueType>();
```

KeyType和ValueType必须是Java中的有效类型，例如String、Integer、Float或任何自定义类，如Employee、Student等。

例如：Map<String, Integer> myMap = new HashMap<String, Integer>();

2. 向HashMap中放入值。

要向HashMap中放入值，我们必须调用HashMap对象的put方法，传入键和值作为参数。

```
myMap.put("key1", 1);  
myMap.put("key2", 2);
```

如果你使用已经存在于Map中的键调用put方法，该方法将覆盖其值并返回旧值。

3. 从HashMap中获取值。

要从HashMap中获取值，你必须调用get方法，并传入键作为参数。

```
myMap.get("key1"); //返回1 (Integer类)
```

如果你传入一个HashMap中不存在的键，该方法将返回null

4. 检查键是否存在于Map中。

```
myMap.containsKey(varKey);
```

5. 检查值是否存在于Map中。

```
myMap.containsValue(varValue);
```

上述方法将返回一个boolean值，表示键或值是否存在于Map中，返回true或false。

第28.3节：使用Java 8中Map的默认方法

使用Java 8中Map接口引入的默认方法示例

1. 使用getOrDefault

返回映射到该键的值，如果键不存在，则返回默认值

```
Map<Integer, String> map = new HashMap<>();  
map.put(1, "第一个元素");  
map.get(1); // => 第一个元素  
map.get(2); // => null  
map.getOrDefault(2, "默认元素"); // => 默认元素
```

2. 使用forEach

允许对每个Map条目执行'Action'中指定的操作

```
Map<Integer, String> map = new HashMap<Integer, String>();  
map.put(1, "one");  
map.put(2, "two");  
map.put(3, "three");
```

KeyType and ValueType must be valid types in Java, such as - String, Integer, Float or any custom class like Employee, Student etc..

For Example : Map<String, Integer> myMap = new HashMap<String, Integer>();

2. Putting values in HashMap.

To put a value in the HashMap, we have to call put method on the HashMap object by passing the Key and the Value as parameters.

```
myMap.put("key1", 1);  
myMap.put("key2", 2);
```

If you call the put method with the Key that already exists in the Map, the method will override its value and return the old value.

3. Getting values from HashMap.

For getting the value from a HashMap you have to call the get method, by passing the Key as a parameter.

```
myMap.get("key1"); //return 1 (class Integer)
```

If you pass a key that does not exist in the HashMap, this method will return null

4. Check whether the Key is in the Map or not.

```
myMap.containsKey(varKey);
```

5. Check whether the Value is in the Map or not.

```
myMap.containsValue(varValue);
```

The above methods will return a boolean value true or false if key, value exists in the Map or not.

Section 28.3: Using Default Methods of Map from Java 8

Examples of using Default Methods introduced in Java 8 in Map interface

1. Using getOrDefault

Returns the value mapped to the key, or if the key is not present, returns the default value

```
Map<Integer, String> map = new HashMap<>();  
map.put(1, "First element");  
map.get(1); // => First element  
map.get(2); // => null  
map.getOrDefault(2, "Default element"); // => Default element
```

2. Using forEach

Allows to perform the operation specified in the 'action' on each Map Entry

```
Map<Integer, String> map = new HashMap<Integer, String>();  
map.put(1, "one");  
map.put(2, "two");  
map.put(3, "three");
```

```
map.forEach((key, value) -> System.out.println("Key: "+key+ " :: Value: "+value));
```

```
// Key: 1 :: Value: one  
// Key: 2 :: Value: two  
// Key: 3 :: Value: three
```

3. 使用 replaceAll

只有当键存在时才会替换为 new-value

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);  
map.replaceAll((key,value)->value+10); //{john=30, paul=40, peter=50}
```

4. 使用 putIfAbsent

如果键不存在或映射为 null，则将键值对添加到映射中

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);  
map.putIfAbsent("kelly", 50); //{john=20, paul=30, peter=40, kelly=50}
```

5. 使用remove

仅当键与给定值关联时才移除该键

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);  
map.remove("peter",40); //{john=30, paul=40}
```

6. 使用replace

如果键存在，则将值替换为新值。如果键不存在，则不执行任何操作。

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);  
map.replace("peter",50); //{john=20, paul=30, peter=50}  
map.replace("jack",60); //{john=20, paul=30, peter=50}
```

7. 使用computeIfAbsent

此方法在Map中添加一个条目。键由函数指定，值是映射函数应用的结果

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);
```

```
map.forEach((key, value) -> System.out.println("Key: "+key+ " :: Value: "+value));
```

```
// Key: 1 :: Value: one  
// Key: 2 :: Value: two  
// Key: 3 :: Value: three
```

3. Using replaceAll

Will replace with new-value only if key is present

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);  
map.replaceAll((key,value)->value+10); //{john=30, paul=40, peter=50}
```

4. Using putIfAbsent

Key-Value pair is added to the map, if the key is not present or mapped to null

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);  
map.putIfAbsent("kelly", 50); //{john=20, paul=30, peter=40, kelly=50}
```

5. Using remove

Removes the key only if its associated with the given value

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);  
map.remove("peter",40); //{john=30, paul=40}
```

6. Using replace

If the key is present then the value is replaced by new-value. If the key is not present, does nothing.

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);  
map.replace("peter",50); //{john=20, paul=30, peter=50}  
map.replace("jack",60); //{john=20, paul=30, peter=50}
```

7. Using computeIfAbsent

This method adds an entry in the Map. the key is specified in the function and the value is the result of the application of the mapping function

```
Map<String, Integer> map = new HashMap<String, Integer>();  
map.put("john", 20);  
map.put("paul", 30);  
map.put("peter", 40);
```

```
map.computeIfAbsent("kelly", k->map.get("john")+10); // {john=20, paul=30, peter=40, kelly=30}
map.computeIfAbsent("peter", k->map.get("john")+10); // {john=20, paul=30, peter=40, kelly=30}
//peter 已存在
```

8. 使用computeIfPresent

此方法添加或修改Map中的条目。如果该键的条目不存在，则不执行任何操作

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.computeIfPresent("kelly", (k,v)->v+10); // {john=20, paul=30, peter=40} //kelly 不存在    map.computeIfPresent("peter", (k,v)->v+10); // {john=20, paul=30, peter=50} // peter 存在, 因此增加值
```

9. 使用compute

此方法用新计算的值替换键的值

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.compute("peter", (k,v)->v+50); // {john=20, paul=30, peter=90} //增加该值
```

10. 使用merge

如果键不存在或键对应的值为null，则向映射中添加键值对；如果键存在，则用新计算的值替换该值；如果新计算的值为null，则从映射中移除该键

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);

//如果键不存在或键对应的值为null，则向映射中添加键值对
map.merge("kelly", 50, (k,v)->map.get("john")+10); // {john=20, paul=30, peter=40, kelly=50}

//如果键存在，则用新计算的值替换该值
map.merge("peter", 50, (k,v)->map.get("john")+10); // {john=20, paul=30, peter=30, kelly=50}

//如果新计算的值为null，则从映射中移除该键
map.merge("peter", 30, (k,v)->map.get("nancy")); // {john=20, paul=30, kelly=50}
```

第28.4节：遍历Map的内容

Map提供了方法，让你可以将映射的键、值或键值对作为集合访问。你可以遍历这些集合。例如，给定以下映射：

```
Map<String, Integer> repMap = new HashMap<>();
repMap.put("Jon Skeet", 927_654);
repMap.put("BalusC", 708_826);
repMap.put("Darin Dimitrov", 715_567);
```

遍历映射的键：

```
map.computeIfAbsent("kelly", k->map.get("john")+10); // {john=20, paul=30, peter=40, kelly=30}
map.computeIfAbsent("peter", k->map.get("john")+10); // {john=20, paul=30, peter=40, kelly=30}
//peter already present
```

8. Using computeIfPresent

This method adds an entry or modifies an existing entry in the Map. Does nothing if an entry with that key is not present

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.computeIfPresent("kelly", (k,v)->v+10); // {john=20, paul=30, peter=40} //kelly not present
map.computeIfPresent("peter", (k,v)->v+10); // {john=20, paul=30, peter=50} // peter present, so increase the value
```

9. Using compute

This method replaces the value of a key by the newly computed value

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);
map.compute("peter", (k,v)->v+50); // {john=20, paul=30, peter=90} //Increase the value
```

10. Using merge

Adds the key-value pair to the map, if key is not present or value for the key is null Replaces the value with the newly computed value, if the key is present Key is removed from the map, if new value computed is null

```
Map<String, Integer> map = new HashMap<String, Integer>();
map.put("john", 20);
map.put("paul", 30);
map.put("peter", 40);

//Adds the key-value pair to the map, if key is not present or value for the key is null
map.merge("kelly", 50, (k,v)->map.get("john")+10); // {john=20, paul=30, peter=40, kelly=50}

//Replaces the value with the newly computed value, if the key is present
map.merge("peter", 50, (k,v)->map.get("john")+10); // {john=20, paul=30, peter=30, kelly=50}

//Key is removed from the map, if new value computed is null
map.merge("peter", 30, (k,v)->map.get("nancy")); // {john=20, paul=30, kelly=50}
```

Section 28.4: Iterating through the contents of a Map

Maps provide methods which let you access the keys, values, or key-value pairs of the map as collections. You can iterate through these collections. Given the following map for example:

```
Map<String, Integer> repMap = new HashMap<>();
repMap.put("Jon Skeet", 927_654);
repMap.put("BalusC", 708_826);
repMap.put("Darin Dimitrov", 715_567);
```

Iterating through map keys:

```
for (String key : repMap.keySet()) {  
    System.out.println(key);  
}
```

输出：

```
Darin Dimitrov  
Jon Skeet  
BalusC
```

keySet() 提供映射的键作为一个 Set。由于键不能包含重复值，因此使用 Set。遍历该集合会依次返回每个键。HashMap 是无序的，因此在此示例中，键可能以任意顺序返回。

遍历映射的值：

```
for (Integer value : repMap.values()) {  
    System.out.println(value);  
}
```

输出：

```
---
```

values() 返回映射的值作为一个 Collection。遍历该集合会依次得到每个值。同样，返回的值顺序可能是任意的。

同时遍历键和值

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {System.out.println("%s = %d", entry.getKey(), entry.getValue());}
```

输出：

```
Darin Dimitrov = 715567  
Jon Skeet = 927654  
BalusC = 708826
```

entrySet() 返回一个 Map.Entry 对象的集合。Map.Entry 允许访问每个条目的键和值。

第28.5节：合并、组合和构造映射

使用 putAll 将一个映射的所有成员放入另一个映射。映射中已存在的键将被其对应的值覆盖。

```
Map<String, Integer> numbers = new HashMap<>();  
numbers.put("One", 1)
```

```
for (String key : repMap.keySet()) {  
    System.out.println(key);  
}
```

Prints:

```
Darin Dimitrov  
Jon Skeet  
BalusC
```

keySet() provides the keys of the map as a Set. Set is used as the keys cannot contain duplicate values. Iterating through the set yields each key in turn. HashMaps are not ordered, so in this example the keys may be returned in any order.

Iterating through map values:

```
for (Integer value : repMap.values()) {  
    System.out.println(value);  
}
```

Prints:

```
715567  
927654  
708826
```

values() returns the values of the map as a Collection. Iterating through the collection yields each value in turn. Again, the values may be returned in any order.

Iterating through keys and values together

```
for (Map.Entry<String, Integer> entry : repMap.entrySet()) {  
    System.out.printf("%s = %d\n", entry.getKey(), entry.getValue());  
}
```

Prints:

```
Darin Dimitrov = 715567  
Jon Skeet = 927654  
BalusC = 708826
```

entrySet() returns a collection of Map.Entry objects. Map.Entry gives access to the key and value for each entry.

Section 28.5: Merging, combine and composing Maps

Use putAll to put every member of one map into another. Keys already present in the map will have their corresponding values overwritten.

```
Map<String, Integer> numbers = new HashMap<>();  
numbers.put("One", 1)
```

```

numbers.put("Three", 3)
Map<String, Integer> other_numbers = new HashMap<>();
other_numbers.put("Two", 2)
other_numbers.put("Three", 4)

numbers.putAll(other_numbers)

```

这将在 numbers 中产生以下映射：

```

"One" -> 1
"Two" -> 2
"Three" -> 4 //旧值 3 被新值 4 覆盖

```

如果你想合并值而不是覆盖它们，可以使用 Java 8 中新增的 Map.merge，它使用用户提供的 BiFunction 来合并重键的值。merge 针对单个键和值操作，因此你需要使用循环或 Map.forEach。这里我们对重复键的字符串进行连接：

```

for (Map.Entry<String, Integer> e : other_numbers.entrySet())
    numbers.merge(e.getKey(), e.getValue(), Integer::sum);
//或者代替上面的循环
other_numbers.forEach((k, v) -> numbers.merge(k, v, Integer::sum));

```

如果你想强制约束没有重复的键，可以使用一个合并函数，该函数会抛出一个 AssertionError：

```

mapA.forEach((k, v) ->
mapB.merge(k, v, (v1, v2) ->
    {throw new AssertionError("duplicate values for key: "+k);}));

```

组合 Map<X,Y> 和 Map<Y,Z> 以获得 Map<X,Z>

如果你想组合两个映射，可以按如下方式操作

```

Map<String, Integer> map1 = new HashMap<String, Integer>();
map1.put("key1", 1);
map1.put("key2", 2);
map1.put("key3", 3);

Map<Integer, Double> map2 = new HashMap<Integer, Double>();
map2.put(1, 1.0);
map2.put(2, 2.0);
map2.put(3, 3.0);

Map<String, Double> map3 = new new HashMap<String, Double>();
map1.forEach((key,value)->map3.put(key,map2.get(value)));

```

这将产生以下映射

```

"key1" -> 1.0
"key2" -> 2.0
"key3" -> 3.0

```

第28.6节：添加多个项目

我们可以使用 V put(K key,V value)：

```

numbers.put("Three", 3)
Map<String, Integer> other_numbers = new HashMap<>();
other_numbers.put("Two", 2)
other_numbers.put("Three", 4)

numbers.putAll(other_numbers)

```

This yields the following mapping in numbers:

```

"One" -> 1
"Two" -> 2
"Three" -> 4 //old value 3 was overwritten by new value 4

```

If you want to combine values instead of overwriting them, you can use `Map.merge`, added in Java 8, which uses a user-provided BiFunction to merge values for duplicate keys. `merge` operates on individual keys and values, so you'll need to use a loop or `Map.forEach`. Here we concatenate strings for duplicate keys:

```

for (Map.Entry<String, Integer> e : other_numbers.entrySet())
    numbers.merge(e.getKey(), e.getValue(), Integer::sum);
//or instead of the above loop
other_numbers.forEach((k, v) -> numbers.merge(k, v, Integer::sum));

```

If you want to enforce the constraint there are no duplicate keys, you can use a merge function that throws an `AssertionError`:

```

mapA.forEach((k, v) ->
mapB.merge(k, v, (v1, v2) ->
    {throw new AssertionError("duplicate values for key: "+k);}));

```

Composing Map<X,Y> and Map<Y,Z> to get Map<X,Z>

If you want to compose two mappings, you can do it as follows

```

Map<String, Integer> map1 = new HashMap<String, Integer>();
map1.put("key1", 1);
map1.put("key2", 2);
map1.put("key3", 3);

Map<Integer, Double> map2 = new HashMap<Integer, Double>();
map2.put(1, 1.0);
map2.put(2, 2.0);
map2.put(3, 3.0);

Map<String, Double> map3 = new new HashMap<String, Double>();
map1.forEach((key,value)->map3.put(key,map2.get(value)));

```

This yields the following mapping

```

"key1" -> 1.0
"key2" -> 2.0
"key3" -> 3.0

```

Section 28.6: Add multiple items

We can use V put(K key,V value):

将指定的值与此映射中的指定键关联（可选操作）。如果映射之前包含该键的映射，则旧值将被指定值替换。

```
String currentVal;
Map<Integer, String> map = new TreeMap<>();
currentVal = map.put(1, "First element.");
System.out.println(currentVal); // 将打印 null
currentVal = map.put(2, "Second element.");
System.out.println(currentVal); // 仍将打印 null
currentVal = map.put(2, "This will replace 'Second element'");
System.out.println(currentVal); // 将打印 Second element.
System.out.println(map.size()); // 将打印 2，因为键 2 的值被替换了。
```

```
Map<Integer, String> map2 = new HashMap<>();
map2.put(2, "Element 2");
map2.put(3, "Element 3");

map.putAll(map2);

System.out.println(map.size());
```

输出：

3

要添加多个元素，可以使用如下的内部类：

```
Map<Integer, String> map = new HashMap<>() {{
    // 这现在是一个匿名内部类，带有一个无名的实例构造函数
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}}
```

请记住，创建匿名内部类并不总是高效的，且可能导致内存泄漏，因此在可能的情况下，使用初始化块代替：

```
static Map<Integer, String> map = new HashMap<>();

static {
    // 现在不再创建内部类，因此我们可以避免内存泄漏
    put(5, "高");
    put(4, "低");
    put(1, "太慢");
}
```

上面的示例使地图变为静态。通过移除所有出现的内容，也可以在非静态环境中使用它静态。

此外，大多数实现支持putAll方法，可以像这样将一个映射中的所有条目添加到另一个映射中：

```
another.putAll(one);
```

Associates the specified value with the specified key in this map (optional operation). If the map previously contained a mapping for the key, the old value is replaced by the specified value.

```
String currentVal;
Map<Integer, String> map = new TreeMap<>();
currentVal = map.put(1, "First element.");
System.out.println(currentVal); // Will print null
currentVal = map.put(2, "Second element.");
System.out.println(currentVal); // Will print null yet again
currentVal = map.put(2, "This will replace 'Second element'");
System.out.println(currentVal); // will print Second element.
System.out.println(map.size()); // Will print 2 as key having
// value 2 was replaced.
```

```
Map<Integer, String> map2 = new HashMap<>();
map2.put(2, "Element 2");
map2.put(3, "Element 3");

map.putAll(map2);

System.out.println(map.size());
```

Output:

3

To add many items you can use an inner classes like this:

```
Map<Integer, String> map = new HashMap<>() {{
    // This is now an anonymous inner class with an unnamed instance constructor
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}}
```

Keep in mind that creating an anonymous inner class is not always efficient and can lead to memory leaks so when possible, use an initializer block instead:

```
static Map<Integer, String> map = new HashMap<>();

static {
    // Now no inner classes are created so we can avoid memory leaks
    put(5, "high");
    put(4, "low");
    put(1, "too slow");
}
```

The example above makes the map static. It can also be used in a non-static context by removing all occurrences of **static**.

In addition to that most implementations support putAll, which can add all entries in one map to another like this:

```
another.putAll(one);
```

第28.7节：创建和初始化映射

介绍

映射存储键/值对，其中每个键都有一个关联的值。给定特定的键，映射可以非常快速地查找关联的值。

映射，也称为关联数组，是一种以键和值形式存储数据的对象。在Java中，映射通过Map接口表示，该接口不是集合接口的扩展。

- 方法一：

```
/*J2SE 5.0之前*/
Map map = new HashMap();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- 方法二：

```
/*J2SE 5.0及以后风格（使用泛型）：*/
Map<String, Object> map = new HashMap<>();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- 方法三：

```
Map<String, Object> map = new HashMap<String, Object>(){{
    put("name", "A");
    put("address", "Malviya-Nagar");
    put("city", "Jaipur");
}};
System.out.println(map);
```

- 方式4：

```
Map<String, Object> map = new TreeMap<String, Object>();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- 方式5：

```
//Java 8
final Map<String, String> map =
    Arrays.stream(new String[][] {
        { "name", "A" },
        { "address", "Malviya-Nagar" },
        { "city", "jaipur" },
    }).collect(Collectors.toMap(m -> m[0], m -> m[1]));
System.out.println(map);
```

Section 28.7: Creating and Initializing Maps

Introduction

Maps stores key/value pairs, where each key has an associated value. Given a particular key, the map can look up the associated value very quickly.

Maps, also known as associate array, is an object that stores the data in form of keys and values. In Java, maps are represented using Map interface which is not an extension of the collection interface.

- Way 1:

```
/*J2SE < 5.0*/
Map map = new HashMap();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Way 2:

```
/*J2SE 5.0+ style (use of generics):*/
Map<String, Object> map = new HashMap<>();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Way 3:

```
Map<String, Object> map = new HashMap<String, Object>(){{
    put("name", "A");
    put("address", "Malviya-Nagar");
    put("city", "Jaipur");
}};
System.out.println(map);
```

- Way 4:

```
Map<String, Object> map = new TreeMap<String, Object>();
map.put("name", "A");
map.put("address", "Malviya-Nagar");
map.put("city", "Jaipur");
System.out.println(map);
```

- Way 5:

```
//Java 8
final Map<String, String> map =
    Arrays.stream(new String[][] {
        { "name", "A" },
        { "address", "Malviya-Nagar" },
        { "city", "jaipur" },
    }).collect(Collectors.toMap(m -> m[0], m -> m[1]));
System.out.println(map);
```

- 方式6：

```
//这种方式是在函数外部初始化一个映射
final static Map<String, String> map;
static
{
    map = new HashMap<String, String>();
    map.put("a", "b");
    map.put("c", "d");
}
```

- 方式7：创建一个不可变的单键值映射。

```
//不可变的单键值映射
Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

请注意，无法修改这样的映射。

任何试图修改地图的操作都会导致抛出 `UnsupportedOperationException` 异常。

```
//不可变的单个键值对
Map<String, String> singletonMap = Collections.singletonMap("key", "value");
singletonMap.put("newKey", "newValue"); //将抛出 UnsupportedOperationException
singletonMap.putAll(new HashMap<>()); //将抛出 UnsupportedOperationException
singletonMap.remove("key"); //将抛出 UnsupportedOperationException
singletonMap.replace("key", "value", "newValue"); //将抛出
UnsupportedOperationException
//等等
```

- Way 6:

```
//This way for initial a map in outside the function
final static Map<String, String> map;
static
{
    map = new HashMap<String, String>();
    map.put("a", "b");
    map.put("c", "d");
}
```

- Way 7: Creating an immutable single key-value map.

```
//Immutable single key-value map
Map<String, String> singletonMap = Collections.singletonMap("key", "value");
```

Please note, that **it is impossible to modify such map**.

Any attempts to modify the map will result in throwing the `UnsupportedOperationException`.

```
//Immutable single key-value pair
Map<String, String> singletonMap = Collections.singletonMap("key", "value");
singletonMap.put("newKey", "newValue"); //will throw UnsupportedOperationException
singletonMap.putAll(new HashMap<>()); //will throw UnsupportedOperationException
singletonMap.remove("key"); //will throw UnsupportedOperationException
singletonMap.replace("key", "value", "newValue"); //will throw
UnsupportedOperationException
//and etc
```

第28.8节：检查键是否存在

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

映射可以包含null值

对于映射，必须小心不要将“包含键”与“拥有值”混淆。例如，[HashMaps](#) 可以包含null，这意味着以下行为是完全正常的：

```
Map<String, String> map = new HashMap<>();
map.put("one", null);
if (map.containsKey("one")) {
    System.out.println("这会打印！"); // 这行代码会被执行
}
if (map.get("one") != null) {
    System.out.println("这行代码永远不会被执行！"); // 这行代码永远不会被执行
}
```

更正式地说，不能保证 `map.contains(key) <=> map.get(key)!=null`

Section 28.8: Check if key exists

```
Map<String, String> num = new HashMap<>();
num.put("one", "first");

if (num.containsKey("one")) {
    System.out.println(num.get("one")); // => first
}
```

Maps can contain null values

For maps, one has to be carrefull not to confuse "containing a key" with "having a value". For example, [HashMaps](#) can contain null which means the following is perfectly normal behavior :

```
Map<String, String> map = new HashMap<>();
map.put("one", null);
if (map.containsKey("one")) {
    System.out.println("This prints !"); // This line is reached
}
if (map.get("one") != null) {
    System.out.println("This is never reached !"); // This line is never reached
}
```

More formally, there is no guarantee that `map.contains(key) <=> map.get(key)!=null`

第28.9节：添加元素

1. 加法

1. Addition

```
Map<Integer, String> map = new HashMap<>();  
map.put(1, "第一个元素。");  
System.out.println(map.get(1));
```

输出：第一个元素。

2.重写

```
Map<Integer, String> map = new HashMap<>();  
map.put(1, "第一个元素。");  
map.put(1, "新元素。");  
System.out.println(map.get(1));
```

输出：新元素。

以HashMap为例。也可以使用实现了Map接口的其他实现。

第28.10节：清空映射

```
Map<Integer, String> map = new HashMap<>();  
  
map.put(1, "第一个元素。");  
map.put(2, "第二个元素。");  
map.put(3, "第三个元素。");  
  
map.clear();  
  
System.out.println(map.size()); // => 0
```

第28.11节：使用自定义对象作为键

在使用自己的对象作为键之前，必须重写对象的hashCode()和equals()方法。

在简单情况下，你会有类似如下代码：

```
class MyKey {  
    private String name;  
    MyKey(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof MyKey) {  
            return this.name.equals(((MyKey)obj).name);  
        }  
        return false;  
    }  
  
    @Override  
    public int hashCode() {  
        return this.name.hashCode();  
    }  
}
```

hashCode 将决定键属于哪个哈希桶，equals 将决定该哈希桶内的哪个对象。

```
Map<Integer, String> map = new HashMap<>();  
map.put(1, "First element.");  
System.out.println(map.get(1));
```

Output: First element.

2. Override

```
Map<Integer, String> map = new HashMap<>();  
map.put(1, "First element.");  
map.put(1, "New element.");  
System.out.println(map.get(1));
```

Output: New element.

HashMap is used as an example. Other implementations that implement the Map interface may be used as well.

Section 28.10: Clear the map

```
Map<Integer, String> map = new HashMap<>();  
  
map.put(1, "First element.");  
map.put(2, "Second element.");  
map.put(3, "Third element.");  
  
map.clear();  
  
System.out.println(map.size()); // => 0
```

Section 28.11: Use custom object as key

Before using your own object as key you must override hashCode() and equals() method of your object.

In simple case you would have something like:

```
class MyKey {  
    private String name;  
    MyKey(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof MyKey) {  
            return this.name.equals(((MyKey)obj).name);  
        }  
        return false;  
    }  
  
    @Override  
    public int hashCode() {  
        return this.name.hashCode();  
    }  
}
```

hashCode will decide which hash bucket the key belongs to and equals will decide which object inside that hash bucket.

如果没有这些方法，上述比较将使用对象的引用，这在每次不使用相同对象引用时将无法工作。

Without these method, the reference of your object will be used for above comparison which will not work unless you use the same object reference every time.

第29章：LinkedHashMap

LinkedHashMap类是Map接口的哈希表和链表实现，具有可预测的迭代顺序。它继承自HashMap类并实现了Map接口。

关于Java LinkedHashMap类的重要点有：LinkedHashMap根据键包含值。它只包含唯一的元素。它可能有一个null键和多个null值。它与HashMap相同，但维护插入顺序。

第29.1节：Java LinkedHashMap类

要点：

- 是Map接口的哈希表和链表实现，具有可预测的迭代顺序。
- 继承HashMap类并实现Map接口。
- 基于键包含值。
- 仅包含唯一元素。
- 可以有一个null键和多个null值。
- 与HashMap相同，但维护插入顺序。

方法：

- void clear()。
- boolean containsKey(Object key)。
- Object get(Object key)。
- protected boolean removeEldestEntry(Map.Entry eldest)

示例：

```
public static void main(String arg[])
{
    LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();
    lhm.put("拉梅什", "中级");
    lhm.put("希瓦", "B-Tech");
    lhm.put("桑托什", "B-Com");
    lhm.put("阿莎", "Msc");
    lhm.put("拉古", "M-Tech");

    Set set = lhm.entrySet();
    Iterator i = set.iterator();
    while (i.hasNext()) {
        Map.Entry me = (Map.Entry) i.next();
        System.out.println(me.getKey() + " : " + me.getValue());
    }

    System.out.println("键包含：" + lhm.containsKey("希瓦"));
    System.out.println("对应键的值：" + lhm.get("阿莎"));
}
```

Chapter 29: LinkedHashMap

LinkedHashMap class is Hash table and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

The important points about Java LinkedHashMap class are: A LinkedHashMap contains values based on the key. It contains only unique elements. It may have one null key and multiple null values. It is same as HashMap instead maintains insertion order.

Section 29.1: Java LinkedHashMap class

Key Points:

- Is Hash table and Linked list implementation of the Map interface, with predictable iteration order.
- inherits HashMap class and implements the Map interface.
- contains values based on the key.
- only unique elements.
- may have one null key and multiple null values.
- same as HashMap instead maintains insertion order.

Methods:

- void clear().
- boolean containsKey(Object key).
- Object get(Object key).
- protected boolean removeEldestEntry(Map.Entry eldest)

Example:

```
public static void main(String arg[])
{
    LinkedHashMap<String, String> lhm = new LinkedHashMap<String, String>();
    lhm.put("Ramesh", "Intermediate");
    lhm.put("Shiva", "B-Tech");
    lhm.put("Santosh", "B-Com");
    lhm.put("Asha", "Msc");
    lhm.put("Raghu", "M-Tech");

    Set set = lhm.entrySet();
    Iterator i = set.iterator();
    while (i.hasNext()) {
        Map.Entry me = (Map.Entry) i.next();
        System.out.println(me.getKey() + " : " + me.getValue());
    }

    System.out.println("The Key Contains :" + lhm.containsKey("Shiva"));
    System.out.println("The value to the corresponding to key :" + lhm.get("Asha"));
}
```

第30章：WeakHashMap

弱哈希映射的概念

第30.1节：WeakHashMap的概念

要点：

- Map的实现。
- 仅存储对其键的弱引用。

弱引用：仅被弱引用引用的对象会被垃圾回收器积极回收；在这种情况下，GC不会等待直到需要内存才回收。

HashMap和WeakHashMap的区别：

如果Java内存管理器不再对作为键指定的对象有强引用，那么WeakHashMap中的映射条目将被移除。

示例：

```
public class WeakHashMapTest {  
    public static void main(String[] args) {  
        Map hashMap= new HashMap();  
  
        Map weakHashMap = new WeakHashMap();  
  
        String keyHashMap = new String("keyHashMap");  
        String keyWeakHashMap = new String("keyWeakHashMap");  
  
        hashMap.put(keyHashMap, "Ankita");  
        weakHashMap.put(keyWeakHashMap, "Atul");  
        System.gc();  
        System.out.println("之前：哈希映射值：" + hashMap.get("keyHashMap") + " 和弱哈希映射  
        值：" + weakHashMap.get("keyWeakHashMap"));  
  
        keyHashMap = null;  
        keyWeakHashMap = null;  
  
        System.gc();  
  
        System.out.println("After: hash map value:" +hashMap.get("keyHashMap")+" and weak hash map  
        value:" +weakHashMap.get("keyWeakHashMap"));  
    }  
}
```

大小差异 (HashMap 与 WeakHashMap)：

调用 HashMap 对象的 size() 方法将返回相同数量的键值对。只有在显式调用 HashMap 对象的 remove() 方法时，大小才会减少。

由于垃圾回收器可能随时丢弃键，WeakHashMap 的行为可能就像有一个未知的线程在悄悄地移除条目一样。因此，size 方法返回的值可能会随着时间的推移变小。所以，**WeakHashMap 的大小会自动减少**。

Chapter 30: WeakHashMap

Concepts of weak Hashmap

Section 30.1: Concepts of WeakHashMap

Key Points:

- Implementation of Map.
- stores only weak references to its keys.

Weak References : The objects that are referenced only by weak references are garbage collected eagerly; the GC won't wait until it needs memory in that case.

Difference between Hashmap and WeakHashMap:

If the Java memory manager no longer has a strong reference to the object specified as a key, then the entry in the map will be removed in WeakHashMap.

Example:

```
public class WeakHashMapTest {  
    public static void main(String[] args) {  
        Map hashMap= new HashMap();  
  
        Map weakHashMap = new WeakHashMap();  
  
        String keyHashMap = new String("keyHashMap");  
        String keyWeakHashMap = new String("keyWeakHashMap");  
  
        hashMap.put(keyHashMap, "Ankita");  
        weakHashMap.put(keyWeakHashMap, "Atul");  
        System.gc();  
        System.out.println("Before: hash map value:" +hashMap.get("keyHashMap")+" and weak hash map  
        value:" +weakHashMap.get("keyWeakHashMap"));  
  
        keyHashMap = null;  
        keyWeakHashMap = null;  
  
        System.gc();  
  
        System.out.println("After: hash map value:" +hashMap.get("keyHashMap")+" and weak hash map  
        value:" +weakHashMap.get("keyWeakHashMap"));  
    }  
}
```

Size differences (HashMap vs WeakHashMap):

Calling size() method on HashMap object will return the same number of key-value pairs. size will decrease only if remove() method is called explicitly on the HashMap object.

Because the garbage collector may discard keys at anytime, a WeakHashMap may behave as though an unknown thread is silently removing entries. So it is possible for the size method to return smaller values over time. So, in **WeakHashMap size decrease happens automatically**.

第31章：SortedMap

Sorted Map 介绍。

第31.1节：排序映射简介

要点：

- SortedMap接口继承自Map。
- 条目按键的升序顺序维护。

SortedMap的方法：

- Comparator comparator().
- Object firstKey().
- SortedMap headMap(Object end).
- Object lastKey().
- SortedMap subMap(Object start, Object end).
- SortedMap tailMap(Object start).

示例

```
public static void main(String args[]) {  
    // 创建一个哈希映射  
    TreeMap tm = new TreeMap();  
  
    // 向映射中添加元素  
    tm.put("Zara", new Double(3434.34));  
    tm.put("Mahnaz", new Double(123.22));  
    tm.put("Ayan", new Double(1378.00));  
    tm.put("Daisy", new Double(99.22));  
    tm.put("Qadir", new Double(-19.08));  
  
    // 获取条目集合  
    Set set = tm.entrySet();  
  
    // 获取迭代器  
    Iterator i = set.iterator();  
  
    // 显示元素  
    while(i.hasNext()) {  
        Map.Entry me = (Map.Entry)i.next();  
        System.out.print(me.getKey() + ": ");  
        System.out.println(me.getValue());  
    }  
    System.out.println();  
  
    // 向Zara的账户存入1000  
    double balance = ((Double)tm.get("Zara")).doubleValue();  
    tm.put("Zara", new Double(balance + 1000));  
    System.out.println("Zara的新余额: " + tm.get("Zara"));  
}
```

Chapter 31: SortedMap

Introduction to sorted Map.

Section 31.1: Introduction to sorted Map

Keypoint:

- SortedMap interface extends Map.
- entries are maintained in an ascending key order.

Methods of sorted Map :

- Comparator comparator().
- Object firstKey().
- SortedMap headMap(Object end).
- Object lastKey().
- SortedMap subMap(Object start, Object end).
- SortedMap tailMap(Object start).

Example

```
public static void main(String args[]) {  
    // Create a hash map  
    TreeMap tm = new TreeMap();  
  
    // Put elements to the map  
    tm.put("Zara", new Double(3434.34));  
    tm.put("Mahnaz", new Double(123.22));  
    tm.put("Ayan", new Double(1378.00));  
    tm.put("Daisy", new Double(99.22));  
    tm.put("Qadir", new Double(-19.08));  
  
    // Get a set of the entries  
    Set set = tm.entrySet();  
  
    // Get an iterator  
    Iterator i = set.iterator();  
  
    // Display elements  
    while(i.hasNext()) {  
        Map.Entry me = (Map.Entry)i.next();  
        System.out.print(me.getKey() + ": ");  
        System.out.println(me.getValue());  
    }  
    System.out.println();  
  
    // Deposit 1000 into Zara's account  
    double balance = ((Double)tm.get("Zara")).doubleValue();  
    tm.put("Zara", new Double(balance + 1000));  
    System.out.println("Zara's new balance: " + tm.get("Zara"));  
}
```

第32章：TreeMap和TreeSet

TreeMap和TreeSet是Java 1.2中新增的基本集合。TreeMap是一个可变的、有序的Map实现。同样，TreeSet是一个可变的、有序的Set实现。

TreeMap实现为红黑树，提供 $O(\log n)$ 的访问时间。TreeSet是使用带有虚拟值的TreeMap实现的。

这两个集合都不是线程安全的。

第32.1节：简单Java类型的TreeMap

首先，我们创建一个空的映射，并向其中插入一些元素：

```
版本 ≥ Java SE 7
TreeMap<Integer, String> treeMap = new TreeMap<>();
版本 < Java SE 7
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
treeMap.put(10, "ten");
treeMap.put(4, "four");
treeMap.put(1, "one");
treeSet.put(12, "twelve");
```

当映射中有了几个元素后，我们可以执行一些操作：

```
System.out.println(treeMap.firstEntry()); // 打印 1=one
System.out.println(treeMap.lastEntry()); // 打印 12=twelve
System.out.println(treeMap.size()); // 打印 4, 因为映射中有4个元素
System.out.println(treeMap.get(12)); // 打印 twelve
System.out.println(treeMap.get(15)); // 打印 null, 因为映射中未找到该键
```

我们也可以使用 `Iterator` 或 `foreach` 循环遍历映射元素。注意，条目是根据它们的自然排序打印的，而不是插入顺序：

```
版本 ≥ Java SE 7
for (Entry<Integer, String> entry : treeMap.entrySet()) {
    System.out.print(entry + " ");
}
Iterator<Entry<Integer, String>> iter = treeMap.entrySet().iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " ");
}
```

第32.2节：简单Java类型的TreeSet

首先，我们创建一个空集合，并向其中插入一些元素：

```
版本 ≥ Java SE 7
TreeSet<Integer> treeSet = new TreeSet<>();
版本 < Java SE 7
TreeSet<Integer> treeSet = new TreeSet<Integer>();
treeSet.add(10);
treeSet.add(4);
treeSet.add(1);
```

Chapter 32: TreeMap and TreeSet

`TreeMap` 和 `TreeSet` 是基本 Java 集合，添加于 Java 1.2。`TreeMap` 是一个 **mutable, ordered, Map** 实现。同样，`TreeSet` 是一个 **mutable, ordered Set** 实现。

`TreeMap` 实现为红黑树，提供 $O(\log n)$ 访问时间。`TreeSet` 使用带有虚拟值的 `TreeMap` 实现。

两个集合都不是线程安全的。

Section 32.1: TreeMap of a simple Java type

首先，我们创建一个空的映射，并向其中插入一些元素：

```
Version ≥ Java SE 7
TreeMap<Integer, String> treeMap = new TreeMap<>();
Version < Java SE 7
TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();
treeMap.put(10, "ten");
treeMap.put(4, "four");
treeMap.put(1, "one");
treeSet.put(12, "twelve");
```

当映射中有了几个元素后，我们可以执行一些操作：

```
System.out.println(treeMap.firstEntry()); // Prints 1=one
System.out.println(treeMap.lastEntry()); // Prints 12=twelve
System.out.println(treeMap.size()); // Prints 4, since there are 4 elements in the map
System.out.println(treeMap.get(12)); // Prints twelve
System.out.println(treeMap.get(15)); // Prints null, since the key is not found in the map
```

我们也可以迭代遍历映射元素，使用 `Iterator` 或 `foreach` 循环。注意，条目是根据它们的自然排序打印的，而不是插入顺序：

```
Version ≥ Java SE 7
for (Entry<Integer, String> entry : treeMap.entrySet()) {
    System.out.print(entry + " ");
}
Iterator<Entry<Integer, String>> iter = treeMap.entrySet().iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " ");
}
```

Section 32.2: TreeSet of a simple Java Type

首先，我们创建一个空集合，并向其中插入一些元素：

```
Version ≥ Java SE 7
TreeSet<Integer> treeSet = new TreeSet<>();
Version < Java SE 7
TreeSet<Integer> treeSet = new TreeSet<Integer>();
treeSet.add(10);
treeSet.add(4);
treeSet.add(1);
```

```
treeSet.add(12);
```

一旦集合中有了几个元素，我们就可以执行一些操作：

```
System.out.println(treeSet.first()); // 输出 1
System.out.println(treeSet.last()); // 输出 12
System.out.println(treeSet.size()); // 输出 4, 因为集合中有 4 个元素
System.out.println(treeSet.contains(12)); // 输出 true
System.out.println(treeSet.contains(15)); // 输出 false
```

我们也可以使用 `Iterator` 或 `foreach` 循环遍历映射元素。注意，条目是根据它们的自然排序打印的，而不是插入顺序：

```
版本 ≥ Java SE 7
for (Integer i : treeSet) {
    System.out.print(i + " "); // 输出 1 4 10 12
}

Iterator<Integer> iter = treeSet.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); // 输出 1 4 10 12
}
```

第32.3节：自定义Java类型的TreeMap/TreeSet

由于TreeMap和TreeSet根据它们的自然排序维护键/元素，因此TreeMap的键和TreeSet的元素必须彼此可比较。

假设我们有一个自定义的Person类：

```
public class Person {
    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... 构造函数, getter, setter 及各种方法
}
```

如果我们将其按原样存储在TreeSet（或作为TreeMap中的键）：

```
TreeSet<Person2> set = ...
set.add(new Person(1,"first","last",Date.from(Instant.now())));
```

那么我们会遇到如下异常：

```
Exception in thread "main" java.lang.ClassCastException: Person cannot be cast to
java.lang.Comparable
at java.util.TreeMap.compare(TreeMap.java:1294)
    at java.util.TreeMap.put(TreeMap.java:538)
    at java.util.TreeSet.add(TreeSet.java:255)
```

为了解决这个问题，假设我们想根据它们的id（`private int id`）的顺序对Person实例进行排序。

我们可以通过两种方式来实现：

1. 一种解决方案是修改Person，使其实现Comparable接口：

```
treeSet.add(12);
```

Once we have a few elements in the set, we can perform some operations:

```
System.out.println(treeSet.first()); // Prints 1
System.out.println(treeSet.last()); // Prints 12
System.out.println(treeSet.size()); // Prints 4, since there are 4 elements in the set
System.out.println(treeSet.contains(12)); // Prints true
System.out.println(treeSet.contains(15)); // Prints false
```

We can also iterate over the map elements using either an Iterator, or a foreach loop. Note that the entries are printed according to their [natural ordering](#), not the insertion order:

```
Version ≥ Java SE 7
for (Integer i : treeSet) {
    System.out.print(i + " "); //prints 1 4 10 12
}

Iterator<Integer> iter = treeSet.iterator();
while (iter.hasNext()) {
    System.out.print(iter.next() + " "); //prints 1 4 10 12
}
```

Section 32.3: TreeMap/TreeSet of a custom Java type

Since `TreeMaps` and `TreeSets` maintain keys/elements according to their [natural ordering](#). Therefor `TreeMap` keys and `TreeSet` elements have to comparable to one another.

Say we have a custom Person class:

```
public class Person {
    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constructors, getters, setters and various methods
}
```

If we store it as-is in a `TreeSet` (or a Key in a `TreeMap`):

```
TreeSet<Person2> set = ...
set.add(new Person(1,"first","last",Date.from(Instant.now())));
```

Then we'd run into an Exception such as this one:

```
Exception in thread "main" java.lang.ClassCastException: Person cannot be cast to
java.lang.Comparable
at java.util.TreeMap.compare(TreeMap.java:1294)
    at java.util.TreeMap.put(TreeMap.java:538)
    at java.util.TreeSet.add(TreeSet.java:255)
```

To fix that, let's assume that we want to order Person instances based on the order of their ids (`private int id`). We could do it in one of two ways:

1. One solution is to modify Person so it would implement the [Comparable interface](#):

```

public class Person implements Comparable<Person> {
    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... 构造函数, getter, setter 及各种方法

    @Override
    public int compareTo(Person o) {
        return Integer.compare(this.id, o.id); //按id比较
    }
}

```

2. 另一种解决方案是为TreeSet提供一个Comparator : _____

```

版本 ≥ Java SE 8
TreeSet<Person> treeSet = new TreeSet<>((personA, personB) -> Integer.compare(personA.getId(),
personB.getId()));

TreeSet<Person> treeSet = new TreeSet<>(new Comparator<Person>(){
    @Override
    public int compare(Person personA, Person personB) {
        return Integer.compare(personA.getId(), personB.getId());
    }
});

```

然而，这两种方法都有两个注意事项：

- 一旦实例被插入到集合中，非常重要的是不要修改用于排序的任何字段 TreeSet/TreeMap。在上述示例中，如果我们更改已插入集合中的某个人的id，可能会遇到意外的行为。
- 正确且一致地实现比较方法非常重要。根据Javadoc :

实现者必须确保对于所有的x和y， $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$ 。（这意味着只有当 $y.\text{compareTo}(x)$ 抛出异常时， $x.\text{compareTo}(y)$ 才必须抛出异常。）实现者还必须确保该关系是传递的： $(x.\text{compareTo}(y)>0 \&& y.\text{compareTo}(z)>0)$ 意味着 $x.\text{compareTo}(z)>0$ 。

最后，实现者必须确保当 $x.\text{compareTo}(y)==0$ 时，对于所有z，有 $\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$ 。

第32.4节：TreeMap和TreeSet的线程安全性

TreeMap和TreeSet不是线程安全的集合，因此在多线程程序中使用时必须小心。

当由多个线程同时读取时，TreeMap和TreeSet都是安全的。因此，如果它们由单个线程创建并填充（例如，在程序开始时），然后仅被多个线程读取而不修改，则无需同步或加锁。

然而，如果同时读取和修改，或被多个线程同时修改，集合可能会抛出ConcurrentModificationException异常或表现出意外行为。在这些情况下，必须使用以下方法之一对集合的访问进行同步/加锁：

```

public class Person implements Comparable<Person> {
    private int id;
    private String firstName, lastName;
    private Date birthday;

    //... Constructors, getters, setters and various methods

    @Override
    public int compareTo(Person o) {
        return Integer.compare(this.id, o.id); //Compare by id
    }
}

```

2. Another solution is to provide the TreeSet with a [Comparator](#):

```

Version ≥ Java SE 8
TreeSet<Person> treeSet = new TreeSet<>((personA, personB) -> Integer.compare(personA.getId(),
personB.getId()));

TreeSet<Person> treeSet = new TreeSet<>(new Comparator<Person>(){
    @Override
    public int compare(Person personA, Person personB) {
        return Integer.compare(personA.getId(), personB.getId());
    }
});

```

However, there are two caveats to both approaches:

- It's **very important** not to modify any fields used for ordering once an instance has been inserted into a TreeSet/TreeMap. In the above example, if we change the id of a person that's already inserted into the collection, we might run into unexpected behavior.
- It's important to implement the comparison properly and consistently. As per the [Javadoc](#):

The implementor must ensure $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$ for all x and y. (This implies that $x.\text{compareTo}(y)$ must throw an exception iff $y.\text{compareTo}(x)$ throws an exception.)

The implementor must also ensure that the relation is transitive: $(x.\text{compareTo}(y)>0 \&& y.\text{compareTo}(z)>0)$ implies $x.\text{compareTo}(z)>0$.

Finally, the implementor must ensure that $x.\text{compareTo}(y)==0$ implies that $\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$, for all z.

Section 32.4: TreeMap and TreeSet Thread Safety

TreeMap and TreeSet are **not** thread-safe collections, so care must be taken to ensure when used in multi-threaded programs.

Both TreeMap and TreeSet are safe when read, even concurrently, by multiple threads. So if they have been created and populated by a single thread (say, at the start of the program), and only then read, but not modified by multiple threads, there's no reason for synchronization or locking.

However, if read and modified concurrently, or modified concurrently by more than one thread, the collection might throw a [ConcurrentModificationException](#) or behave unexpectedly. In these cases, it's imperative to synchronize/lock access to the collection using one of the following approaches:

1. 使用Collections.synchronizedSorted..:

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());  
SortedMap<Integer, String> map = Collections.synchronizedSortedMap(new  
TreeMap<Integer, String>());
```

这将提供一个由实际集合支持的SortedSet/SortedMap实现，并在某个互斥对象上进行同步。注意，这会在单个锁上同步对集合的所有读写访问，因此即使是并发读取也不可能。

2. 通过手动在某个对象上同步，比如集合本身：

```
TreeSet<Integer> set = new TreeSet<>();
```

...

```
//线程1  
synchronized (set) {  
    set.add(4);  
}
```

...

```
//线程2  
synchronized (set) {  
    set.remove(5);  
}
```

3. 通过使用锁，比如ReentrantReadWriteLock：

```
TreeSet<Integer> set = new TreeSet<>();  
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

...

```
//线程1  
lock.writeLock().lock();  
set.add(4);  
lock.writeLock().unlock();
```

...

```
//线程2  
lock.readLock().lock();  
set.contains(5);  
lock.readLock().unlock();
```

与之前的同步方法不同，使用ReadWriteLock允许多个线程同时从映射中读取数据。

1. Using Collections.synchronizedSorted..:

```
SortedSet<Integer> set = Collections.synchronizedSortedSet(new TreeSet<Integer>());  
SortedMap<Integer, String> map = Collections.synchronizedSortedMap(new  
TreeMap<Integer, String>());
```

This will provide a [SortedSet/SortedMap](#) implementation backed by the actual collection, and synchronized on some mutex object. Note that this will synchronize all read and write access to the collection on a single lock, so even concurrent reads would not be possible.

2. By manually synchronizing on some object, like the collection itself:

```
TreeSet<Integer> set = new TreeSet<>();
```

...

```
//Thread 1  
synchronized (set) {  
    set.add(4);  
}
```

...

```
//Thread 2  
synchronized (set) {  
    set.remove(5);  
}
```

3. By using a lock, such as a [ReentrantReadWriteLock](#):

```
TreeSet<Integer> set = new TreeSet<>();  
ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
```

...

```
//Thread 1  
lock.writeLock().lock();  
set.add(4);  
lock.writeLock().unlock();
```

...

```
//Thread 2  
lock.readLock().lock();  
set.contains(5);  
lock.readLock().unlock();
```

As opposed to the previous synchronization methods, using a [ReadWriteLock](#) allows multiple threads to read from the map concurrently.

第33章：队列和双端队列

第33.1节：PriorityQueue的使用

PriorityQueue是一种数据结构。与SortedSet类似，PriorityQueue也根据元素的优先级对其进行排序。优先级较高的元素排在前面。PriorityQueue的类型应实现comparable或comparator接口，其方法决定数据结构中元素的优先级。

```
//优先队列的类型是Integer。  
PriorityQueue<Integer> queue = new PriorityQueue<Integer>();  
  
// 元素被添加到 PriorityQueue 中  
queue.addAll( Arrays.asList( 9, 2, 3, 1, 3, 8 ) );  
  
// PriorityQueue 使用 Integer 类的 compareTo 方法对元素进行排序  
// 该队列的头部是根据指定顺序的最小元素  
System.out.println( queue ); // 输出: [1, 2, 3, 9, 3, 8]  
queue.remove();  
System.out.println( queue ); // 输出: [2, 3, 3, 9, 8]  
queue.remove();  
System.out.println( queue ); // 输出: [3, 8, 3, 9]  
queue.remove();  
System.out.println( queue ); // 输出: [3, 8, 9]  
queue.remove();  
System.out.println( queue ); // 输出: [8, 9]  
queue.remove();  
System.out.println( queue ); // 输出: [9]  
queue.remove();  
System.out.println( queue ); //输出: []
```

第33.2节：双端队列（Deque）

双端队列（Deque）是一种“双端队列”，意味着元素可以添加到队列的前端或尾端。队列只能向队列的尾部添加元素。

双端队列（Deque）继承了队列（Queue）接口，这意味着常规方法依然存在，然而双端队列接口提供了额外的方法，使队列更加灵活。如果你了解队列的工作原理，这些额外的方法本身就很直观，因为这些方法旨在增加更多的灵活性：

方法	简要说明
getFirst()	获取队列头部（head）的第一个元素，但不移除它。
getLast()	获取队列尾部（tail）的第一个元素，但不移除它。
addFirst(E e)	向队列的头部添加一个元素
addLast(E e)	向队列的尾部添加一个元素
removeFirst()	移除队列头部的第一个元素
removeLast()	移除队列尾部的第一个元素

当然，offer、poll 和 peek 也有相同的选项，但它们不适用于异常，而是适用于特殊值。在这里展示它们的作用没有意义。

添加和访问元素

要向双端队列（Deque）的尾部添加元素，可以调用它的 add() 方法。你也可以使用 addFirst() 和 addLast() 方法，分别向双端队列的头部和尾部添加元素。

Chapter 33: Queues and Deques

Section 33.1: The usage of the PriorityQueue

PriorityQueue is a data structure. Like [SortedSet](#), PriorityQueue sorts also its elements based on their priorities. The elements, which have a higher priority, comes first. The type of the PriorityQueue should implement comparable or comparator interface, whose methods decides the priorities of the elements of the data structure.

```
//The type of the PriorityQueue is Integer.  
PriorityQueue<Integer> queue = new PriorityQueue<Integer>();  
  
//The elements are added to the PriorityQueue  
queue.addAll( Arrays.asList( 9, 2, 3, 1, 3, 8 ) );  
  
//The PriorityQueue sorts the elements by using compareTo method of the Integer Class  
//The head of this queue is the least element with respect to the specified ordering  
System.out.println( queue ); //The Output: [1, 2, 3, 9, 3, 8]  
queue.remove();  
System.out.println( queue ); //The Output: [2, 3, 3, 9, 8]  
queue.remove();  
System.out.println( queue ); //The Output: [3, 8, 3, 9]  
queue.remove();  
System.out.println( queue ); //The Output: [3, 8, 9]  
queue.remove();  
System.out.println( queue ); //The Output: [8, 9]  
queue.remove();  
System.out.println( queue ); //The Output: [9]  
queue.remove();  
System.out.println( queue ); //The Output: []
```

Section 33.2: Deque

A Deque is a "double ended queue" which means that elements can be added at the front or the tail of the queue. The queue only can add elements to the tail of a queue.

The Deque inherits the Queue interface which means the regular methods remain, however the Deque interface offers additional methods to be more flexible with a queue. The additional methods really speak for them self if you know how a queue works, since those methods are intended to add more flexibility:

Method	Brief description
getFirst()	Gets the first item of the head of the queue without removing it.
getLast()	Gets the first item of the tail of the queue without removing it.
addFirst(E e)	Adds an item to the head of the queue
addLast(E e)	Adds an item to the tail of the queue
removeFirst()	Removes the first item at the head of the queue
removeLast()	Removes the first item at the tail of the queue

Of course the same options for offer, poll and peek are available, however they do not work with exceptions but rather with special values. There is no point in showing what they do here.

Adding and Accessing Elements

To add elements to the tail of a Deque you call its add() method. You can also use the addFirst() and addLast() methods, which add elements to the head and tail of the deque.

```

Deque<String> dequeA = new LinkedList<>();

dequeA.add("element 1"); //在尾部添加元素
dequeA.addFirst("element 2"); //在头部添加元素
dequeA.addLast("element 3"); //在尾部添加元素

```

你可以查看队列头部的元素而不将其取出。这可以通过 `element()` 方法实现。你也可以使用 `getFirst()` 和 `getLast()` 方法，它们分别返回双端队列中的第一个和最后一个元素。示例如下：

```

String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();

```

移除元素

要从双端队列中移除元素，可以调用 `remove()`、`removeFirst()` 和 `removeLast()` 方法。以下是一些示例：

```

String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement = dequeA.removeLast();

```

第33.3节：栈

什么是栈？

在Java中，栈是一种针对对象的后进先出（LIFO）数据结构。

栈API

Java包含一个具有以下方法的栈API

<code>Stack()</code>	//创建一个空栈	
<code>isEmpty()</code>	//栈是否为空？	返回类型：Boolean
<code>push(Item item)</code>	//将一个元素压入栈中	
<code>pop()</code>	//移除栈顶元素	返回类型：Item
<code>size()</code>	//返回栈中元素数量	返回类型：Int

示例

```

import java.util.*;

public class StackExample {

    public static void main(String args[]) {
        Stack st = new Stack();
        System.out.println("stack: " + st);

        st.push(10);
        System.out.println("10 被压入栈中");
        System.out.println("栈: " + st);

        st.push(15);
        System.out.println("15 被压入栈中");
        System.out.println("栈: " + st);

        st.push(80);
        System.out.println("80 被压入栈中");
        System.out.println("栈: " + st);
    }
}

```

```

Deque<String> dequeA = new LinkedList<>();

```

```

dequeA.add("element 1"); //add element at tail
dequeA.addFirst("element 2"); //add element at head
dequeA.addLast("element 3"); //add element at tail

```

You can peek at the element at the head of the queue without taking the element out of the queue. This is done via the `element()` method. You can also use the `getFirst()` and `getLast()` methods, which return the first and last element in the Deque. Here is how that looks:

```

String firstElement0 = dequeA.element();
String firstElement1 = dequeA.getFirst();
String lastElement = dequeA.getLast();

```

Removing Elements

To remove elements from a deque, you call the `remove()`, `removeFirst()` and `removeLast()` methods. Here are a few examples:

```

String firstElement = dequeA.remove();
String firstElement = dequeA.removeFirst();
String lastElement = dequeA.removeLast();

```

Section 33.3: Stacks

What is a Stack?

In Java, Stacks are a LIFO (Last In, First Out) Data structure for objects.

Stack API

Java contains a Stack API with the following methods

<code>Stack()</code>	//Creates an empty Stack	
<code>isEmpty()</code>	//Is the Stack Empty?	Return Type: Boolean
<code>push(Item item)</code>	//push an item onto the stack	
<code>pop()</code>	//removes item from top of stack	Return Type: Item
<code>size()</code>	//returns # of items in stack	Return Type: Int

Example

```

import java.util.*;

public class StackExample {

    public static void main(String args[]) {
        Stack st = new Stack();
        System.out.println("stack: " + st);

        st.push(10);
        System.out.println("10 was pushed to the stack");
        System.out.println("stack: " + st);

        st.push(15);
        System.out.println("15 was pushed to the stack");
        System.out.println("stack: " + st);

        st.push(80);
        System.out.println("80 was pushed to the stack");
        System.out.println("stack: " + st);
    }
}

```

```

st.pop();
System.out.println("80 被弹出栈");
System.out.println("栈: " + st);

st.pop();
System.out.println("15 被弹出栈");
System.out.println("栈: " + st);

st.pop();
System.out.println("10 被弹出栈");
System.out.println("栈: " + st);

if(st.isEmpty())
{
    System.out.println("栈为空");
}
}
}

```

这将返回：

```

栈: []
10 被压入栈中
栈: [10]
15 被压入栈中
栈: [10, 15]
80 被压入栈中
栈: [10, 15, 80]
80 被弹出栈
栈: [10, 15]
15 被弹出栈
栈: [10]
10 被弹出栈
栈: []
空栈

```

```

st.pop();
System.out.println("80 was popped from the stack");
System.out.println("stack: " + st);

st.pop();
System.out.println("15 was popped from the stack");
System.out.println("stack: " + st);

st.pop();
System.out.println("10 was popped from the stack");
System.out.println("stack: " + st);

if(st.isEmpty())
{
    System.out.println("empty stack");
}
}
}

```

This returns:

```

stack: []
10 was pushed to the stack
stack: [10]
15 was pushed to the stack
stack: [10, 15]
80 was pushed to the stack
stack: [10, 15, 80]
80 was popped from the stack
stack: [10, 15]
15 was popped from the stack
stack: [10]
10 was popped from the stack
stack: []
empty stack

```

第33.4节：阻塞队列

BlockingQueue 是一个接口，它表示一种队列，当你尝试从中出队且队列为空时，或者尝试向其中入队且队列已满时，会阻塞。尝试从空队列出队的线程会被阻塞，直到其他线程向队列中插入了一个元素。尝试向满队列入队的线程会被阻塞，直到其他线程通过出队一个或多个元素或完全清空队列腾出空间。

BlockingQueue 方法有四种形式，处理无法立即满足的操作方式不同，但可能在未来某个时间点满足：一种抛出异常，第二种返回特殊值（根据操作不同，可能是 null 或 false），第三种无限期阻塞当前线程直到操作成功，第四种仅阻塞给定的最长时间后放弃。

操作	抛出异常	特殊值	阻塞	超时
插入	add()	offer(e)	put(e)	offer(e, time, unit)
删除	remove()	poll()	take()	poll(time, unit)
检查元素()		peek()	不适用	不适用

一个阻塞队列（BlockingQueue）可以是有界的或无界的。有界的阻塞队列是指初始化时带有初始

Section 33.4: BlockingQueue

A BlockingQueue is an interface, which is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up.

	Operation	Throws	Exception	Special Value	Blocks	Times out
Insert	add()		offer(e)	put(e)	offer(e, time, unit)	
Remove	remove()		poll()	take()	poll(time, unit)	
Examine	element()		peek()	N/A	N/A	

A BlockingQueue can be **bounded** or **unbounded**. A bounded BlockingQueue is one which is initialized with initial

容量。

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

如果队列的大小等于定义的初始容量，任何对put()方法的调用都会被阻塞。

无界队列是指初始化时没有容量限制，实际上默认初始化为 Integer.MAX_VALUE。

一些常见的阻塞队列（BlockingQueue）实现有：

- 1.ArrayBlockingQueue
- 2.LinkedBlockingQueue
- 3.PriorityBlockingQueue

现在让我们来看一个ArrayBlockingQueue的示例：

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<>(2);  
bQueue.put("This is entry 1");  
System.out.println("条目一完成");  
bQueue.put("这是条目二");  
System.out.println("条目二完成");  
bQueue.put("这是条目三");  
System.out.println("条目三完成");
```

这将打印：

```
条目一完成  
条目二完成
```

线程将在第二次输出后被阻塞。

第33.5节：LinkedList作为FIFO队列

java.util.LinkedList类，虽然实现了java.util.List接口，但也是java.util.Queue接口的通用实现，基于FIFO（先进先出）原则操作。

在下面的示例中，使用offer()方法，元素被插入到LinkedList中。此插入操作称为入队（enqueue）。在下面的while循环中，元素根据FIFO从Queue中移除。此操作称为出队（dequeue）。

```
Queue<String> queue = new LinkedList<String>();  
  
queue.offer("第一个元素");  
queue.offer("第二个元素");  
queue.offer("第三个元素");  
queue.offer("第四个元素");  
queue.offer("第五个元素");  
  
while ( !queue.isEmpty() ) {  
    System.out.println( queue.poll() );  
}
```

这段代码的输出是

```
第一个元素
```

capacity.

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);
```

Any calls to a put() method will be blocked if the size of the queue is equal to the initial capacity defined.

An unbounded Queue is one which is initialized without capacity, actually by default it initialized with Integer.MAX_VALUE.

Some common implementations of BlockingQueue are:

1. ArrayBlockingQueue
2. LinkedBlockingQueue
3. PriorityBlockingQueue

Now let's look at an example of ArrayBlockingQueue:

```
BlockingQueue<String> bQueue = new ArrayBlockingQueue<>(2);  
bQueue.put("This is entry 1");  
System.out.println("Entry one done");  
bQueue.put("This is entry 2");  
System.out.println("Entry two done");  
bQueue.put("This is entry 3");  
System.out.println("Entry three done");
```

This will print:

```
Entry one done  
Entry two done
```

And the thread will be blocked after the second output.

Section 33.5: LinkedList as a FIFO Queue

The [java.util.LinkedList](#) class, while implementing [java.util.List](#) is a general-purpose implementation of [java.util.Queue](#) interface too operating on a [FIFO \(First In, First Out\)](#) principle.

In the example below, with offer() method, the elements are inserted into the [LinkedList](#). This insertion operation is called enqueue. In the while loop below, the elements are removed from the Queue based on FIFO. This operation is called dequeue.

```
Queue<String> queue = new LinkedList<String>();  
  
queue.offer("first element");  
queue.offer("second element");  
queue.offer("third element");  
queue.offer("fourth. element");  
queue.offer("fifth. element");  
  
while ( !queue.isEmpty() ) {  
    System.out.println( queue.poll() );  
}
```

The output of this code is

```
first element
```

第二个元素
第三个元素
第四个元素
第五个元素

如输出所示，最先插入的元素“第一个元素”最先被移除，“第二个元素”排在第二位被移除，依此类推。

第33.6节：队列接口

基础知识

队列（Queue）是一种用于在处理前保存元素的集合。队列通常（但不一定）以先进先出（FIFO）的方式排列元素。

队列的头部是通过调用remove或poll方法将被移除的元素。在FIFO队列中，所有新元素都插入到队列的尾部。

队列接口

```
公共接口 队列<E> extends 集合<E> {
    布尔型 添加(E e);
    布尔型 提供(E e);
    E 移除();
    E 轮询();
    E 元素();
    E 查看();
}
```

每个队列方法有两种形式：

- 一种在操作失败时抛出异常；
- 另一种在操作失败时返回特殊值（根据操作不同，可能是null或false）。

操作类型抛出异常返回特殊值

插入	添加(e)	提供(e)
移除	remove()	poll()
检查	element()	peek()

second element
third element
fourth element
fifth element

As seen in the output, the first inserted element "first element" is removed firstly, "second element" is removed in the second place etc.

Section 33.6: Queue Interface

Basics

A Queue is a collection for holding elements prior to processing. Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner.

Head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue.

The Queue Interface

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);

    boolean offer(E e);

    E remove();

    E poll();

    E element();

    E peek();
}
```

Each Queue method exists in two forms:

- one throws an exception if the operation fails;
- other returns a special value if the operation fails (either **null** or **false** depending on the operation).

Type of operation Throws exception Returns special value

Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

第34章：双端队列接口

双端队列（Deque）是一种线性集合，支持在两端插入和移除元素。

deque一词是“double ended queue”（双端队列）的缩写，通常读作“deck”。

大多数双端队列实现对其包含的元素数量没有固定限制，但该接口既支持容量受限的双端队列，也支持无固定大小限制的双端队列。

双端队列接口比栈和队列这两种抽象数据类型更丰富，因为它同时实现了栈和队列的功能。

第34.1节：向双端队列添加元素

```
Deque deque = new LinkedList();

//在尾部添加元素
deque.add("Item1");

//在头部添加元素
deque.addFirst("Item2");

//在尾部添加元素
deque.addLast("Item3");
```

第34.2节：从双端队列中移除元素

```
//检索并移除此双端队列所表示队列的头部元素
Object headItem = deque.remove();

//检索并移除此双端队列的第一个元素
Object firstItem = deque.removeFirst();

//检索并移除此双端队列的最后一个元素。
Object lastItem = deque.removeLast();
```

第34.3节：检索元素但不移除

```
//检索但不移除此双端队列所表示的队列的头部元素
Object headItem = deque.element();

//检索但不移除此双端队列的第一个元素。
Object firstItem = deque.getFirst();

//检索但不移除此双端队列的最后一个元素。
Object lastItem = deque.getLast();
```

第34.4节：遍历双端队列

```
//使用迭代器
Iterator iterator = deque.iterator();
while(iterator.hasNext()){
    String Item = (String) iterator.next();
}

//使用for循环
```

Chapter 34: Dequeue Interface

A Deque is linear collection that supports element insertion and removal at both ends.

The name deque is short for "double ended queue" and is usually pronounced "deck".

Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

The Deque interface is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at same time

Section 34.1: Adding Elements to Deque

```
Deque deque = new LinkedList();

//Adding element at tail
deque.add("Item1");

//Adding element at head
deque.addFirst("Item2");

//Adding element at tail
deque.addLast("Item3");
```

Section 34.2: Removing Elements from Deque

```
//Retrieves and removes the head of the queue represented by this deque
Object headItem = deque.remove();

//Retrieves and removes the first element of this deque.
Object firstItem = deque.removeFirst();

//Retrieves and removes the last element of this deque.
Object lastItem = deque.removeLast();
```

Section 34.3: Retrieving Element without Removing

```
//Retrieves, but does not remove, the head of the queue represented by this deque
Object headItem = deque.element();

//Retrieves, but does not remove, the first element of this deque.
Object firstItem = deque.getFirst();

//Retrieves, but does not remove, the last element of this deque.
Object lastItem = deque.getLast();
```

Section 34.4: Iterating through Deque

```
//Using Iterator
Iterator iterator = deque.iterator();
while(iterator.hasNext()){
    String Item = (String) iterator.next();
}

//Using For Loop
```

```
for(Object object : deque) {  
    String Item = (String) object;  
}
```

```
for(Object object : deque) {  
    String Item = (String) object;  
}
```

第35章：枚举

Java枚举（使用 enum 关键字声明）是单个类中大量常量的简写语法。

第35.1节：声明和使用基本枚举

枚举可以被视为语法糖，用于表示一个密封类，该类只在编译时已知的次数内实例化，以定义一组常量。

一个简单的枚举用来列出不同的季节，可以这样声明：

```
public enum Season {  
    WINTER,  
    SPRING,  
    SUMMER,  
    FALL  
}
```

虽然枚举常量不一定非得全部大写，但Java的惯例是常量名全部大写，单词之间用下划线分隔。

你可以在单独的文件中声明一个枚举：

```
/**  
 * 这个枚举声明在Season.java文件中。  
 */  
public enum Season {  
    WINTER,  
    SPRING,  
    SUMMER,  
    FALL  
}
```

但你也可以把它声明在另一个类里面：

```
public class Day {  
  
    private Season season;  
  
    public String getSeason() {  
        return season.name();  
    }  
  
    public void setSeason(String season) {  
        this.season = Season.valueOf(season);  
    }  
  
    /**  
     * 该枚举声明在 Day.java 文件内，  
     * 由于声明为 private，外部无法访问。  
     */  
    private enum Season {  
        WINTER,  
        SPRING,  
        SUMMER,  
        FALL  
    }  
}
```

Chapter 35: Enums

Java enums (declared using the `enum` keyword) are shorthand syntax for sizable quantities of constants of a single class.

Section 35.1: Declaring and using a basic enum

`Enum` can be considered to be syntax sugar for a sealed class that is instantiated only a number of times known at compile-time to define a set of constants.

A simple enum to list the different seasons would be declared as follows:

```
public enum Season {  
    WINTER,  
    SPRING,  
    SUMMER,  
    FALL  
}
```

While the enum constants don't necessarily need to be in all-caps, it is Java convention that names of constants are entirely uppercase, with words separated by underscores.

You can declare an Enum in its own file:

```
/**  
 * This enum is declared in the Season.java file.  
 */  
public enum Season {  
    WINTER,  
    SPRING,  
    SUMMER,  
    FALL  
}
```

But you can also declare it inside another class:

```
public class Day {  
  
    private Season season;  
  
    public String getSeason() {  
        return season.name();  
    }  
  
    public void setSeason(String season) {  
        this.season = Season.valueOf(season);  
    }  
  
    /**  
     * This enum is declared inside the Day.java file and  
     * cannot be accessed outside because it's declared as private.  
     */  
    private enum Season {  
        WINTER,  
        SPRING,  
        SUMMER,  
        FALL  
    }  
}
```

```
}
```

最后，不能在方法体或构造函数内声明枚举：

```
public class Day {  
  
    /**  
     * 构造函数  
     */  
    public Day() {  
        // 非法。编译错误  
        enum Season {  
            冬季,  
            春季,  
            夏季,  
            秋季  
        }  
    }  
  
    public void 一个简单方法() {  
        // 合法。你可以在方法内部声明一个基本类型（或对象）。编译通过！  
        int 基本整型 = 42;  
  
        // 不合法。编译错误。  
        枚举 季节 {  
            冬季,  
            春季,  
            夏季,  
            秋季  
        }  
  
        季节 season = 季节.春季;  
    }  
}
```

不允许重复的枚举常量：

```
public enum Season {  
    WINTER,  
    冬季, // 编译时错误：重复的常量  
    春季,  
    夏季,  
    秋季  
}
```

枚举的每个常量默认都是 `public`、`static` 和 `final`。由于每个常量都是 `static`，可以直接通过枚举名访问它们。

枚举常量可以作为方法参数传递：

```
public static void display(Season s) {  
    System.out.println(s.name()); // name() 是一个内置方法，用于获取枚举常量的确切名称  
}  
  
display(Season.WINTER); // 输出 "WINTER"
```

```
}
```

```
}
```

Finally, you cannot declare an Enum inside a method body or constructor:

```
public class Day {  
  
    /**  
     * Constructor  
     */  
    public Day() {  
        // Illegal. Compilation error  
        enum Season {  
            WINTER,  
            SPRING,  
            SUMMER,  
            FALL  
        }  
    }  
}  
  
public void aSimpleMethod() {  
    // Legal. You can declare a primitive (or an Object) inside a method. Compile!  
    int primitiveInt = 42;  
  
    // Illegal. Compilation error.  
    enum Season {  
        WINTER,  
        SPRING,  
        SUMMER,  
        FALL  
    }  
  
    Season season = Season.SPRING;  
}  
}
```

Duplicate enum constants are not allowed:

```
public enum Season {  
    WINTER,  
    WINTER, // Compile Time Error : Duplicate Constants  
    SPRING,  
    SUMMER,  
    FALL  
}
```

Every constant of enum is `public`, `static` and `final` by default. As every constant is `static`, they can be accessed directly using the enum name.

Enum constants can be passed around as method parameters:

```
public static void display(Season s) {  
    System.out.println(s.name()); // name() is a built-in method that gets the exact name of the  
    enum constant  
}  
  
display(Season.WINTER); // Prints out "WINTER"
```

你可以使用 values() 方法获取枚举常量的数组。返回的数组中的值保证按声明顺序排列：

```
Season[] seasons = Season.values();
```

注意：此方法每次调用都会分配一个新的值数组。

要遍历枚举常量：

```
public static void enumIterate() {  
    for (Season s : Season.values()) {  
        System.out.println(s.name());  
    }  
}
```

你可以在 switch 语句中使用枚举：

```
public static void enumSwitchExample(Season s) {  
    switch(s) {  
        case WINTER:  
            System.out.println("天气相当冷");  
            break;  
        case SPRING:  
            System.out.println("天气逐渐变暖");  
            break;  
        case SUMMER:  
            System.out.println("天气相当热");  
            break;  
        case FALL:  
            System.out.println("天气逐渐变凉");  
            break;  
    }  
}
```

你也可以使用==来比较枚举常量：

```
Season.秋季 == Season.冬季 // false  
Season.春季 == Season.春季 // true
```

比较枚举常量的另一种方法是使用equals()，如下所示，但这被认为是不好的做法，因为你很容易陷入以下陷阱：

```
Season.FALL.equals(Season.FALL); // true  
Season.FALL.equals(Season.WINTER); // false  
Season.FALL.equals("FALL"); // false 且无编译错误
```

此外，虽然enum中的实例集合在运行时不能更改，但实例本身并非天生不可变，因为像其他类一样，enum可以包含可变字段，如下所示。

```
public enum MutableExample {  
    A,  
    B;  
  
    private int count = 0;  
  
    public void increment() {
```

You can get an array of the enum constants using the values() method. The values are guaranteed to be in declaration order in the returned array:

```
Season[] seasons = Season.values();
```

Note: this method allocates a new array of values each time it is called.

To iterate over the enum constants:

```
public static void enumIterate() {  
    for (Season s : Season.values()) {  
        System.out.println(s.name());  
    }  
}
```

You can use enums in a switch statement:

```
public static void enumSwitchExample(Season s) {  
    switch(s) {  
        case WINTER:  
            System.out.println("It's pretty cold");  
            break;  
        case SPRING:  
            System.out.println("It's warming up");  
            break;  
        case SUMMER:  
            System.out.println("It's pretty hot");  
            break;  
        case FALL:  
            System.out.println("It's cooling down");  
            break;  
    }  
}
```

You can also compare enum constants using ==:

```
Season.FALL == Season.WINTER // false  
Season.SPRING == Season.SPRING // true
```

Another way to compare enum constants is by using equals() as below, which is considered bad practice as you can easily fall into pitfalls as follows:

```
Season.FALL.equals(Season.FALL); // true  
Season.FALL.equals(Season.WINTER); // false  
Season.FALL.equals("FALL"); // false and no compiler error
```

Furthermore, although the set of instances in the enum cannot be changed at run-time, the instances themselves are not inherently immutable because like any other class, an enum can contain mutable fields as is demonstrated below.

```
public enum MutableExample {  
    A,  
    B;  
  
    private int count = 0;  
  
    public void increment() {
```

```

count++;

}

public void print() {
    System.out.println("The count of " + name() + " is " + count);
}

// 用法:
MutableExample.A.print();      // 输出 0
MutableExample.A.increment();
MutableExample.A.print();      // 输出 1 -- 我们修改了一个字段
MutableExample.B.print();      // 输出 0 -- 另一个实例保持不变

```

然而，一个好的做法是使**enum**实例不可变，即当它们没有任何额外的字段或所有这些字段都被标记为**final**且本身不可变时。这将确保在应用程序的整个生命周期中，一个**enum**不会泄漏任何内存，并且其实例在所有线程中使用都是安全的。

枚举隐式实现了Serializable和Comparable，因为Enum类实现了这些接口：

```

public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable

```

第35.2节：带构造函数的枚举

枚举不能有公共构造函数；但是，私有构造函数是允许的（枚举的构造函数默认是包私有的）：

```

public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25); // 美国硬币的常用名称
    // 注意上面的括号和构造函数参数相匹配
    private int value;

    Coin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // int 值将是 5

```

建议将所有字段设为私有，并提供 getter 方法，因为枚举的实例数量是有限的。

如果你要将一个Enum实现为一个class，代码将如下所示：

```

public class Coin<T extends Coin<T>> implements Comparable<T>, Serializable{
    public static final Coin PENNY = new Coin(1);
    public static final Coin NICKEL = new Coin(5);
    public static final Coin DIME = new Coin(10);
    public static final Coin QUARTER = new Coin(25);

    private int value;
}

```

```

count++;

}

public void print() {
    System.out.println("The count of " + name() + " is " + count);
}

// Usage:
MutableExample.A.print();      // Outputs 0
MutableExample.A.increment();
MutableExample.A.print();      // Outputs 1 -- we've changed a field
MutableExample.B.print();      // Outputs 0 -- another instance remains unchanged

```

However, a good practice is to make **enum** instances immutable, i.e. when they either don't have any additional fields or all such fields are marked as **final** and are immutable themselves. This will ensure that for a lifetime of the application an **enum** won't leak any memory and that it is safe to use its instances across all threads.

Enums implicitly implement [Serializable](#) and [Comparable](#) because the **Enum** class does:

```

public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable

```

Section 35.2: Enums with constructors

An **enum** cannot have a public constructor; however, private constructors are acceptable (constructors for enums are package-private by default):

```

public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25); // usual names for US coins
    // note that the above parentheses and the constructor arguments match
    private int value;

    Coin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

int p = Coin.NICKEL.getValue(); // the int value will be 5

```

It is recommended that you keep all fields private and provide getter methods, as there are a finite number of instances for an enum.

If you were to implement an **Enum** as a **class** instead, it would look like this:

```

public class Coin<T extends Coin<T>> implements Comparable<T>, Serializable{
    public static final Coin PENNY = new Coin(1);
    public static final Coin NICKEL = new Coin(5);
    public static final Coin DIME = new Coin(10);
    public static final Coin QUARTER = new Coin(25);

    private int value;
}

```

```

private Coin(int value){
    this.value = value;
}

public int getValue() {
    return value;
}

int p = Coin.NICKEL.getValue(); // int 值将是 5

```

枚举常量在技术上是可变的，因此可以添加 `setter` 来更改枚举常量的内部结构。但这被认为是非常不好的做法，应当避免。

最佳实践是使枚举字段不可变，使用`final`修饰：

```

public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value) {
        this.value = value;
    }

    ...
}

```

您可以在同一个枚举中定义多个构造函数。当您这样做时，您在枚举声明中传入的参数决定调用哪个构造函数：

```

public enum Coin {
    PENNY(1, true), NICKEL(5, false), DIME(10), QUARTER(25);

    private final int value;
    private final boolean isCopperColored;

    Coin(int value){
        this(value, false);
    }

    Coin(int value, boolean isCopperColored){
        this.value = value;
        this.isCopperColored = isCopperColored;
    }

    ...
}

```

注意：所有非原始枚举字段都应实现`Serializable`接口，因为`Enum`类实现了该接口。

第35.3节：带有抽象方法的枚举

枚举可以定义抽象方法，每个枚举成员都必须实现该方法。

```

枚举 Action {
    躲避{

```

```

private Coin(int value){
    this.value = value;
}

public int getValue() {
    return value;
}

int p = Coin.NICKEL.getValue(); // the int value will be 5

```

Enum constants are technically mutable, so a setter could be added to change the internal structure of an enum constant. However, this is considered very bad practice and should be avoided.

Best practice is to make Enum fields immutable, with `final`:

```

public enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int value;

    Coin(int value) {
        this.value = value;
    }

    ...
}

```

You may define multiple constructors in the same enum. When you do, the arguments you pass in your enum declaration decide which constructor is called:

```

public enum Coin {
    PENNY(1, true), NICKEL(5, false), DIME(10), QUARTER(25);

    private final int value;
    private final boolean isCopperColored;

    Coin(int value){
        this(value, false);
    }

    Coin(int value, boolean isCopperColored){
        this.value = value;
        this.isCopperColored = isCopperColored;
    }

    ...
}

```

Note: All non-primitive enum fields should implement `Serializable` because the `Enum` class does.

Section 35.3: Enums with Abstract Methods

Enums can define abstract methods, which each `enum` member is required to implement.

```

enum Action {
    DODGE {

```

```

public boolean 执行(玩家 player) {
    return player.正在攻击();
}

攻击{
    public boolean 执行(玩家 player) {
        return player.有武器();
    }
}

跳跃{
    public boolean 执行(玩家 player) {
        return player.获取坐标().等于(new 坐标(0, 0));
    }
};

public abstract boolean 执行(玩家 player);
}

```

这允许每个枚举成员为给定操作定义自己的行为，而无需在顶层定义的方法中根据类型进行切换。

请注意，这种模式是通常通过多态和/或实现接口来实现的简写形式。

第35.4节：实现接口

这是一个枚举，同时也是一个可调用函数，用于测试字符串输入是否符合预编译的正则表达式模式。

```

import java.util.function.Predicate;
import java.util.regex.Pattern;

enum RegEx implements Predicate<String> {
    UPPER("[A-Z]+"), LOWER("[a-z]+"), NUMERIC("[+-]?[0-9]+");

    private final Pattern pattern;

    private RegEx(final String pattern) {
        this.pattern = Pattern.compile(pattern);
    }

    @Override
    public boolean test(final String input) {
        return this.pattern.matcher(input).matches();
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(RegEx.UPPER.test("ABC"));
        System.out.println(RegEx.LOWER.test("abc"));
        System.out.println(RegEx.NUMERIC.test("+111"));
    }
}

```

枚举的每个成员也可以实现该方法：

```
import java.util.function.Predicate;
```

```

public boolean execute(Player player) {
    return player.isAttacking();
}

ATTACK {
    public boolean execute(Player player) {
        return player.hasWeapon();
    }
}

JUMP {
    public boolean execute(Player player) {
        return player.getCoordinates().equals(new Coordinates(0, 0));
    }
};

public abstract boolean execute(Player player);
}

```

This allows for each enum member to define its own behaviour for a given operation, without having to switch on types in a method in the top-level definition.

Note that this pattern is a short form of what is typically achieved using polymorphism and/or implementing interfaces.

Section 35.4: Implements Interface

This is an **enum** that is also a callable function that tests **String** inputs against precompiled regular expression patterns.

```

import java.util.function.Predicate;
import java.util.regex.Pattern;

enum RegEx implements Predicate<String> {
    UPPER("[A-Z]+"), LOWER("[a-z]+"), NUMERIC("[+-]?[0-9]+");

    private final Pattern pattern;

    private RegEx(final String pattern) {
        this.pattern = Pattern.compile(pattern);
    }

    @Override
    public boolean test(final String input) {
        return this.pattern.matcher(input).matches();
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(RegEx.UPPER.test("ABC"));
        System.out.println(RegEx.LOWER.test("abc"));
        System.out.println(RegEx.NUMERIC.test("+111"));
    }
}

```

Each member of the enum can also implement the method:

```
import java.util.function.Predicate;
```

```

enum 接受者 实现 谓词<字符串> {
    NULL {
        @Override
        public boolean 测试(字符串 s) { 返回 s == null; }
    },
    EMPTY {
        @Override
        public boolean 测试(字符串 s) { 返回 s.equals(""); }
    },
    NULL_OR_EMPTY {
        @Override
        public boolean 测试(字符串 s) { 返回 NULL.测试(s) || EMPTY.测试(s); }
    };
}

public class Main {
    public static void 主函数(字符串[] 参数) {
        系统.输出.打印(接受者.NULL.测试(null)); // true
        系统.输出.打印(接受者.EMPTY.测试("")); // true
        系统.输出.打印(接受者.NULL_OR_EMPTY.测试(" ")); // false
    }
}

```

第35.5节：使用单元素枚举实现单例模式

枚举常量在枚举首次被引用时实例化。因此，这允许使用单元素枚举实现单例软件设计模式。

```

public enum Attendant {
    INSTANCE;

    private Attendant() {
        // 执行一些初始化操作
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}

public class Main {
    public static void main(String... args) {
        Attendant.INSTANCE.sayHello(); // 此时实例化
    }
}

```

根据Joshua Bloch的《Effective Java》一书，单元素枚举是实现单例模式的最佳方式。这种方法具有以下优点：

- 线程安全
- 保证单一实例化
- 开箱即用的序列化

如“实现接口”一节所示，该单例还可以实现一个或多个接口。

```

enum Acceptor implements Predicate<String> {
    NULL {
        @Override
        public boolean test(String s) { return s == null; }
    },
    EMPTY {
        @Override
        public boolean test(String s) { return s.equals(""); }
    },
    NULL_OR_EMPTY {
        @Override
        public boolean test(String s) { return NULL.test(s) || EMPTY.test(s); }
    };
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Acceptor.NULL.test(null)); // true
        System.out.println(Acceptor.EMPTY.test("")); // true
        System.out.println(Acceptor.NULL_OR_EMPTY.test(" ")); // false
    }
}

```

Section 35.5: Implement Singleton pattern with a single-element enum

Enum constants are instantiated when an enum is referenced for the first time. Therefore, that allows to implement Singleton software design pattern with a single-element enum.

```

public enum Attendant {
    INSTANCE;

    private Attendant() {
        // perform some initialization routine
    }

    public void sayHello() {
        System.out.println("Hello!");
    }
}

public class Main {
    public static void main(String... args) {
        Attendant.INSTANCE.sayHello(); // instantiated at this point
    }
}

```

According to "Effective Java" book by Joshua Bloch, a single-element enum is the best way to implement a singleton. This approach has following advantages:

- thread safety
- guarantee of single instantiation
- out-of-the-box serialization

And as shown in the section implements interface this singleton might also implement one or more interfaces.

第35.6节：使用方法和静态代码块

枚举可以包含方法，就像任何类一样。为了了解这是如何工作的，我们将声明如下枚举：

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
}
```

让我们写一个返回相反方向枚举值的方法：

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
  
    public Direction getOpposite(){  
        switch (this){  
            case NORTH:  
                return SOUTH;  
            case SOUTH:  
                return NORTH;  
            case WEST:  
                return EAST;  
            case EAST:  
                return WEST;  
            default: //这永远不会发生  
                return null;  
        }  
    }  
}
```

这可以通过使用字段和静态初始化块进一步改进：

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
  
    private Direction opposite;  
  
    public Direction getOpposite(){  
        return opposite;  
    }  
  
    static {  
        NORTH.opposite = SOUTH;  
        SOUTH.opposite = NORTH;  
        WEST.opposite = EAST;  
        EAST.opposite = WEST;  
    }  
}
```

在此示例中，相反方向存储在私有实例字段opposite中，该字段在第一次使用Direction时静态初始化。在这种特殊情况下（因为NORTH引用SOUTH，反之亦然），我们不能在这里使用带构造函数的枚举（构造函数NORTH(SOUTH)、SOUTH(NORTH)、EAST(WEST)、WEST(EAST)会更优雅，并且允许opposite被声明为final，但会自引用，因此不被允许）。

第35.7节：零实例枚举

```
enum Util {  
    /* 无实例 */;
```

Section 35.6: Using methods and static blocks

An enum can contain a method, just like any class. To see how this works, we'll declare an enum like this:

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
}
```

Let's have a method that returns the enum in the opposite direction:

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
  
    public Direction getOpposite(){  
        switch (this){  
            case NORTH:  
                return SOUTH;  
            case SOUTH:  
                return NORTH;  
            case WEST:  
                return EAST;  
            case EAST:  
                return WEST;  
            default: //This will never happen  
                return null;  
        }  
    }  
}
```

This can be improved further through the use of fields and static initializer blocks:

```
public enum Direction {  
    NORTH, SOUTH, EAST, WEST;  
  
    private Direction opposite;  
  
    public Direction getOpposite(){  
        return opposite;  
    }  
  
    static {  
        NORTH.opposite = SOUTH;  
        SOUTH.opposite = NORTH;  
        WEST.opposite = EAST;  
        EAST.opposite = WEST;  
    }  
}
```

In this example, the opposite direction is stored in a private instance field opposite, which is statically initialized the first time a Direction is used. In this particular case (because NORTH references SOUTH and conversely), we cannot use Enums with constructors here (Constructors NORTH(SOUTH), SOUTH(NORTH), EAST(WEST), WEST(EAST) would be more elegant and would allow opposite to be declared **final**, but would be self-referential and therefore are not allowed).

Section 35.7: Zero instance enum

```
enum Util {  
    /* No instances */;
```

```

public static int clamp(int min, int max, int i) {
    return Math.min(Math.max(i, min), max);
}

// 其他实用方法...
}

```

正如enum可以用于单例（1个实例的类），它也可以用于工具类（0个实例的类）。只需确保用;结束（空的）枚举常量列表即可。

请参阅问题[Zero instance enum vs private constructors for preventing instantiation](#)，了解与私有构造函数相比的优缺点讨论。

第35.8节：作为有界类型参数的枚举

在Java中编写泛型类时，可以确保类型参数是枚举。由于所有枚举都继承自Enum类，因此可以使用以下语法。

```

public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}

```

在此示例中，类型T必须是枚举。

第35.9节：枚举的文档编写

枚举名称并不总是足够清晰易懂。要为枚举编写文档，请使用标准的javadoc：

```

/**
 * 美国硬币
 */
public enum Coins {

    /**
     * 一美分硬币，通常称为便士，      * 是一种货币单位，等于
     * 一美元的百分之一
     *
     * PENNY(1),

    /**
     * 镍币是一种五美分硬币，等于
     * 一美元的五百分之一
     */
    NICEL(5),

    /**
     * 角币是一种十美分硬币，指的是
     * 一美元的十分之一
     */
    DIME(10),

    /**
     * 四分之一美元硬币价值25美分，
     * 是一美元的四分之一
     */
}

```

```

public static int clamp(int min, int max, int i) {
    return Math.min(Math.max(i, min), max);
}

// other utility methods...
}

```

Just as **enum** can be used for singletons (1 instance classes), it can be used for utility classes (0 instance classes). Just make sure to terminate the (empty) list of enum constants with a ;.

See the question [Zero instance enum vs private constructors for preventing instantiation](#) for a discussion on pro's and con's compared to private constructors.

Section 35.8: Enum as a bounded type parameter

When writing a class with generics in java, it is possible to ensure that the type parameter is an enum. Since all enums extend the **Enum** class, the following syntax may be used.

```

public class Holder<T extends Enum<T>> {
    public final T value;

    public Holder(T init) {
        this.value = init;
    }
}

```

In this example, the type T *must* be an enum.

Section 35.9: Documenting enums

Not always the **enum** name is clear enough to be understood. To document an **enum**, use standard javadoc:

```

/**
 * United States coins
 */
public enum Coins {

    /**
     * One-cent coin, commonly known as a penny,
     * is a unit of currency equaling one-hundredth
     * of a United States dollar
     */
    PENNY(1),

    /**
     * A nickel is a five-cent coin equaling
     * five-hundredth of a United States dollar
     */
    NICEL(5),

    /**
     * The dime is a ten-cent coin refers to
     * one tenth of a United States dollar
     */
    DIME(10),

    /**
     * The quarter is a US coin worth 25 cents,
     * one-fourth of a United States dollar
     */
}

```

```

*/  

QUARTER(25);  
  

private int value;  
  

Coins(int value){  
    this.value = value;  
}  
  

public int getValue(){  
    return value;  
}  
}

```

第35.10节：枚举常量特定主体

在一个枚举中，可以为枚举的某个特定常量定义特定行为，以覆盖枚举的默认行为，这种技术称为常量特定主体。

假设设有三个钢琴学生——约翰、班和卢克——定义在一个名为PianoClass的枚举中，如下所示：

```

enum PianoClass {  
JOHN, BEN, LUKE;  
    public String getSex() {  
        return "男";  
    }  
    public String getLevel() {  
        return "初学者";  
    }  
}

```

有一天，又来了两个学生——丽塔和汤姆——他们的性别（女）和水平（中级）与之前的不匹配：

```

enum PianoClass2 {  
JOHN, BEN, LUKE, RITA, TOM;  
    public String getSex() {  
        return "男"; // 问题，丽塔是女性  
    }  
    public String getLevel() {  
        return "初学者"; // 问题，Tom 是中级学生  
    }  
}

```

因此，简单地将新学生添加到常量声明中，如下所示，是不正确的：

```

PianoClass2 tom = PianoClass2.TOM;  
PianoClass2 rita = PianoClass2.RITA;  
System.out.println(tom.getLevel()); // 输出 初学者 -> 错误，Tom 不是初学者  
System.out.println(rita.getSex()); // 输出 男 -> 错误，Rita 不是男性

```

可以为每个常量 Rita 和 Tom 定义特定行为，覆盖PianoClass2的默认行为，如下所示：

```

enum PianoClass3 {  
JOHN, BEN, LUKE,  
    RITA {  
        @Override

```

```

*/  

QUARTER(25);  
  

private int value;  
  

Coins(int value){  
    this.value = value;  
}  
  

public int getValue(){  
    return value;  
}  
}

```

Section 35.10: Enum constant specific body

In an **enum** it is possible to define a specific behavior for a particular constant of the **enum** which overrides the default behavior of the **enum**, this technique is known as *constant specific body*.

Suppose three piano students - John, Ben and Luke - are defined in an **enum** named PianoClass, as follows:

```

enum PianoClass {  
JOHN, BEN, LUKE;  
    public String getSex() {  
        return "Male";  
    }  
    public String getLevel() {  
        return "Beginner";  
    }  
}

```

And one day two other students arrive - Rita and Tom - with a sex (Female) and level (Intermediate) that do not match the previous ones:

```

enum PianoClass2 {  
JOHN, BEN, LUKE, RITA, TOM;  
    public String getSex() {  
        return "Male"; // issue, Rita is a female  
    }  
    public String getLevel() {  
        return "Beginner"; // issue, Tom is an intermediate student  
    }  
}

```

so that simply adding the new students to the constant declaration, as follows, is not correct:

```

PianoClass2 tom = PianoClass2.TOM;  
PianoClass2 rita = PianoClass2.RITA;  
System.out.println(tom.getLevel()); // prints Beginner -> wrong Tom's not a beginner  
System.out.println(rita.getSex()); // prints Male -> wrong Rita's not a male

```

It's possible to define a specific behavior for each of the constant, Rita and Tom, which overrides the PianoClass2 default behavior as follows:

```

enum PianoClass3 {  
JOHN, BEN, LUKE,  
    RITA {  
        @Override

```

```

public String getSex() {
    return "女性";
}
TOM {
    @Override
    public String getLevel() {
        return "中级";
    }
}
public String getSex() {
    return "Male";
}
public String getLevel() {
    return "初学者";
}
}

```

现在汤姆的等级和丽塔的性别都如预期一样：

```

PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;
System.out.println(tom.getLevel()); // 输出 Intermediate
System.out.println(rita.getSex()); // 输出 Female

```

定义特定内容体的另一种方式是使用构造函数，例如：

```

enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
    }
}

```

使用方法：

```

Friend mat = Friend.MAT;
Friend john = Friend.JOHN;
Friend jane = Friend.JANE;
System.out.println(mat.getGender()); // 男性
System.out.println(john.getGender()); // 男性
System.out.println(jane.getGender()); // 女性

```

第35.11节：获取枚举的值

每个枚举类都包含一个隐式的静态方法，名为values()。该方法返回一个包含该枚举所有值的数组。你可以使用此方法来遍历这些值。但需要注意的是，每次调用该方法时都会返回一个new数组。

```

public String getSex() {
    return "Female";
}
TOM {
    @Override
    public String getLevel() {
        return "Intermediate";
    }
}
public String getSex() {
    return "Male";
}
public String getLevel() {
    return "Beginner";
}
}

```

and now Tom's level and Rita's sex are as they should be:

```

PianoClass3 tom = PianoClass3.TOM;
PianoClass3 rita = PianoClass3.RITA;
System.out.println(tom.getLevel()); // prints Intermediate
System.out.println(rita.getSex()); // prints Female

```

Another way to define content specific body is by using constructor, for instance:

```

enum Friend {
    MAT("Male"),
    JOHN("Male"),
    JANE("Female");

    private String gender;

    Friend(String gender) {
        this.gender = gender;
    }

    public String getGender() {
        return this.gender;
    }
}

```

and usage:

```

Friend mat = Friend.MAT;
Friend john = Friend.JOHN;
Friend jane = Friend.JANE;
System.out.println(mat.getGender()); // Male
System.out.println(john.getGender()); // Male
System.out.println(jane.getGender()); // Female

```

Section 35.11: Getting the values of an enum

Each enum class contains an implicit static method named values(). This method returns an array containing all values of that enum. You can use this method to iterate over the values. It is important to note however that this method returns a new array every time it is called.

```

public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    /**
     * 打印出该枚举中的所有值。
     */
    public static void printAllDays() {
        for(Day day : Day.values()) {
            System.out.println(day.name());
        }
    }
}

```

如果你需要一个Set，可以使用EnumSet.allOf(Day.class)。

第35.12节：枚举多态模式

当一个方法需要接受一组“可扩展”的枚举值时，程序员可以像对待普通类一样应用多态，通过创建一个接口，该接口将在任何需要使用枚举的地方使用：

```

公共接口 ExtensibleEnum {
    字符串 name();
}

```

通过这种方式，任何实现（实现）该接口的枚举都可以作为参数使用，允许程序员创建可变数量的枚举，这些枚举将被方法接受。例如，这在API中非常有用，其中存在一个默认的（不可修改的）枚举，而这些API的用户希望用更多的值“扩展”该枚举。

可以按如下方式定义一组默认的枚举值：

```

公共枚举 DefaultValues 实现 ExtensibleEnum {
    VALUE_ONE, VALUE_TWO;
}

```

然后可以这样定义附加值：

```

公共枚举 ExtendedValues 实现 ExtensibleEnum {
    VALUE_THREE, VALUE_FOUR;
}

```

示例展示了如何使用这些枚举——注意printEnum()如何接受来自两种枚举类型的值：

```

私有 无返回值 printEnum(ExtensibleEnum val) {
    系统.输出.打印行(val.name());
}

printEnum(DefaultValues.VALUE_ONE); // VALUE_ONE
printEnum(DefaultValues.VALUE_TWO); // VALUE_TWO
printEnum(ExtendedValues.VALUE_THREE); // VALUE_THREE
printEnum(ExtendedValues.VALUE_FOUR); // VALUE_FOUR

```

注意：该模式并不能防止你在另一个枚举中重新定义已经在某个枚举中定义的枚举值。这些枚举值将是不同的实例。此外，由于我们只有接口，而不是真正的枚举，因此无法使用基于枚举的switch语句。

```

public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

    /**
     * Print out all the values in this enum.
     */
    public static void printAllDays() {
        for(Day day : Day.values()) {
            System.out.println(day.name());
        }
    }
}

```

If you need a Set you can use EnumSet.allOf(Day.class) as well.

Section 35.12: Enum Polymorphism Pattern

When a method need to accept an "extensible" set of enum values, the programmer can apply polymorphism like on a normal class by creating an interface which will be used anywhere where the enums shall be used:

```

public interface ExtensibleEnum {
    String name();
}

```

This way, any enum tagged by (implementing) the interface can be used as a parameter, allowing the programmer to create a variable amount of enums that will be accepted by the method. This can be useful, for example, in APIs where there is a default (unmodifiable) enum and the user of these APIs want to "extend" the enum with more values.

A set of default enum values can be defined as follows:

```

public enum DefaultValues implements ExtensibleEnum {
    VALUE_ONE, VALUE_TWO;
}

```

Additional values can then be defined like this:

```

public enum ExtendedValues implements ExtensibleEnum {
    VALUE_THREE, VALUE_FOUR;
}

```

Sample which shows how to use the enums - note how printEnum() accepts values from both enum types:

```

private void printEnum(ExtensibleEnum val) {
    System.out.println(val.name());
}

printEnum(DefaultValues.VALUE_ONE); // VALUE_ONE
printEnum(DefaultValues.VALUE_TWO); // VALUE_TWO
printEnum(ExtendedValues.VALUE_THREE); // VALUE_THREE
printEnum(ExtendedValues.VALUE_FOUR); // VALUE_FOUR

```

Note: This pattern does not prevent you from redefining enum values, which are already defined in one enum, in another enum. These enum values would be different instances then. Also, it is not possible to use switch-on-enum since all we have is the interface, not the real enum.

第35.13节：枚举值的比较与包含

枚举只包含常量，可以直接用`==`进行比较。因此，只需进行引用检查，无需使用`equals`方法。此外，如果错误地使用`equals`，可能会引发`NullPointerException`，而使用`==`检查则不会出现这种情况。

```
enum Day {  
    GOOD, AVERAGE, WORST;  
}  
  
public class Test {  
  
    public static void main(String[] args) {  
        Day day = null;  
  
        if (day.equals(Day.GOOD)) {//NullPointerException!  
            System.out.println("Good Day!");  
        }  
  
        if (day == Day.GOOD) {//始终使用 == 来比较枚举  
            System.out.println("Good Day!");  
        }  
    }  
}
```

为了对枚举值进行分组、补集、范围操作，我们有`EnumSet`类，它包含不同的方法。

- `EnumSet#range`：通过两个端点定义的范围获取枚举的子集
- `EnumSet#of`：特定枚举的集合，没有任何范围。有多个重载的`of`方法。
- `EnumSet#complementOf`：方法参数中提供的枚举值的补集枚举集合

```
enum 页面 {  
    A1, A2, A3, A4, A5, A6, A7, A8, A9, A10  
}  
  
public class Test {  
  
    public static void main(String[] args) {  
        EnumSet<页面> 范围 = EnumSet.范围(页面.A1, 页面.A5);  
  
        if (范围.包含(页面.A4)) {  
            System.out.println("范围包含 A4");  
        }  
  
        EnumSet<页面> of = EnumSet.of(页面.A1, 页面.A5, 页面.A3);  
  
        if (of.包含(页面.A1)) {  
            System.out.println("Of 包含 A1");  
        }  
    }  
}
```

第 35.14 节：通过名称获取枚举常量

假设我们有一个枚举 星期几：

Section 35.13: Compare and Contains for Enum values

Enums contains only constants and can be compared directly with `==`. So, only reference check is needed, no need to use `equals` method. Moreover, if `equals` used incorrectly, may raise the `NullPointerException` while that's not the case with `==` check.

```
enum Day {  
    GOOD, AVERAGE, WORST;  
}  
  
public class Test {  
  
    public static void main(String[] args) {  
        Day day = null;  
  
        if (day.equals(Day.GOOD)) {//NullPointerException!  
            System.out.println("Good Day!");  
        }  
  
        if (day == Day.GOOD) {//Always use == to compare enum  
            System.out.println("Good Day!");  
        }  
    }  
}
```

To group, complement, range the enum values we have `EnumSet` class which contains different methods.

- `EnumSet#range`：To get subset of enum by range defined by two endpoints
- `EnumSet#of`：Set of specific enums without any range. Multiple overloaded of methods are there.
- `EnumSet#complementOf`：Set of enum which is complement of enum values provided in method parameter

```
enum Page {  
    A1, A2, A3, A4, A5, A6, A7, A8, A9, A10  
}  
  
public class Test {  
  
    public static void main(String[] args) {  
        EnumSet<Page> range = EnumSet.range(Page.A1, Page.A5);  
  
        if (range.contains(Page.A4)) {  
            System.out.println("Range contains A4");  
        }  
  
        EnumSet<Page> of = EnumSet.of(Page.A1, Page.A5, Page.A3);  
  
        if (of.contains(Page.A1)) {  
            System.out.println("Of contains A1");  
        }  
    }  
}
```

Section 35.14: Get enum constant by name

Say we have an enum `DayOfWeek`:

```
enum 星期几 {  
星期日, 星期一, 星期二, 星期三, 星期四, 星期五, 星期六;  
}
```

枚举编译时带有一个内置的静态valueOf()方法，可用于通过名称查找常量：

```
String dayName = DayOfWeek.SUNDAY.name();  
assert dayName.equals("SUNDAY");  
  
DayOfWeek day = DayOfWeek.valueOf(dayName);  
assert day == DayOfWeek.SUNDAY;
```

这也可以通过动态枚举类型实现：

```
Class<DayOfWeek> enumType = DayOfWeek.class;  
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");  
assert day == DayOfWeek.SUNDAY;
```

如果指定的枚举没有匹配名称的常量，这两个valueOf()方法都会抛出IllegalArgumentException异常。

Guava库提供了一个辅助方法Enums.getIfPresent()，返回Guava的Optional以消除显式异常处理：

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;  
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);  
assert day == DayOfWeek.SUNDAY;
```

第35.15节：带属性（字段）的枚举

如果我们想使用enum来包含更多信息，而不仅仅是作为常量值，并且希望能够比较两个枚举。

考虑以下示例：

```
public enum Coin {  
PENNY(1), NICKEL(5), DIME(10), QUARTER(25);  
  
private final int value;  
  
Coin(int value){  
    this.value = value;  
}  
  
public boolean isGreaterThan(Coin other){  
    return this.value > other.value;  
}  
}
```

这里我们定义了一个名为**Coin**的Enum，表示其值。通过方法isGreaterThan我们可以比较两个enum：

```
Coin penny = Coin.PENNY;  
Coin dime = Coin.DIME;  
  
System.out.println(penny.isGreaterThan(dime)); // 输出：false
```

```
enum DayOfWeek {  
SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;  
}
```

An enum is compiled with a built-in static valueOf() method which can be used to lookup a constant by its name:

```
String dayName = DayOfWeek.SUNDAY.name();  
assert dayName.equals("SUNDAY");  
  
DayOfWeek day = DayOfWeek.valueOf(dayName);  
assert day == DayOfWeek.SUNDAY;
```

This is also possible using a dynamic enum type:

```
Class<DayOfWeek> enumType = DayOfWeek.class;  
DayOfWeek day = Enum.valueOf(enumType, "SUNDAY");  
assert day == DayOfWeek.SUNDAY;
```

Both of these valueOf() methods will throw an [IllegalArgumentException](#) if the specified enum does not have a constant with a matching name.

The Guava library provides a helper method [Enums.getIfPresent\(\)](#) that returns a Guava [Optional](#) to eliminate explicit exception handling:

```
DayOfWeek defaultDay = DayOfWeek.SUNDAY;  
DayOfWeek day = Enums.valueOf(DayOfWeek.class, "INVALID").or(defaultDay);  
assert day == DayOfWeek.SUNDAY;
```

Section 35.15: Enum with properties (fields)

In case we want to use **enum** with more information and not just as constant values, and we want to be able to compare two enums.

Consider the following example:

```
public enum Coin {  
PENNY(1), NICKEL(5), DIME(10), QUARTER(25);  
  
private final int value;  
  
Coin(int value){  
    this.value = value;  
}  
  
public boolean isGreaterThan(Coin other){  
    return this.value > other.value;  
}  
}
```

Here we defined an **Enum** called **Coin** which represent its value. With the method **isGreaterThan** we can compare two **enums**:

```
Coin penny = Coin.PENNY;  
Coin dime = Coin.DIME;  
  
System.out.println(penny.isGreaterThan(dime)); // prints: false
```

```
System.out.println(dime.isGreater Than(penny)); // 输出 : true
```

第35.16节：将枚举转换为字符串

有时你想将枚举转换为字符串，有两种方法可以做到这一点。

假设我们有：

```
public enum Fruit {  
    苹果, 橙子, 草莓, 香蕉, 柠檬, 葡萄柚;  
}
```

那么我们如何将类似Fruit.APPLE转换为"APPLE"呢？

使用name()进行转换

name()是enum中的一个内部方法，返回枚举的String表示，返回的String准确地表示了枚举值的定义方式。

例如：

```
System.out.println(Fruit.BANANA.name()); // "BANANA"  
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

使用toString()进行转换

toString()默认情况下被重写为与name()具有相同的行为

但是，toString()很可能被开发者重写，以使其打印出更友好的String

如果你想在代码中进行检查，不要使用toString()，name()更稳定。只有在你要将值输出到日志或标准输出等时才使用toString()

默认情况下：

```
System.out.println(Fruit.BANANA.toString()); // "BANANA"  
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

被覆盖的示例

```
System.out.println(Fruit.BANANA.toString()); // "Banana"  
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

第35.17节：带有静态字段的枚举

如果你的枚举类需要有静态字段，请记住它们是在枚举值本身之后创建的。这意味着，以下代码将导致NullPointerException：

```
enum 示例 {  
    ONE(1), TWO(2);  
  
    static Map<String, Integer> integers = new HashMap<>();  
  
    private 示例(int value) {
```

```
System.out.println(dime.isGreater Than(penny)); // prints: true
```

Section 35.16: Convert enum to String

Sometimes you want to convert your enum to a String, there are two ways to do that.

Assume we have:

```
public enum Fruit {  
    APPLE, ORANGE, STRAWBERRY, BANANA, LEMON, GRAPE_FRUIT;  
}
```

So how do we convert something like Fruit.APPLE to "APPLE"?

Convert using name()

name() is an internal method in enum that returns the String representation of the enum, the return String represents **exactly** how the enum value was defined.

For example:

```
System.out.println(Fruit.BANANA.name()); // "BANANA"  
System.out.println(Fruit.GRAPE_FRUIT.name()); // "GRAPE_FRUIT"
```

Convert using toString()

toString() is, by default, overridden to have the same behavior as name()

However, toString() is likely overridden by developers to make it print a more user friendly String

Don't use toString() if you want to do checking in your code, name() is much more stable for that. Only use toString() when you are going to output the value to logs or stdout or something

By default:

```
System.out.println(Fruit.BANANA.toString()); // "BANANA"  
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "GRAPE_FRUIT"
```

Example of being overridden

```
System.out.println(Fruit.BANANA.toString()); // "Banana"  
System.out.println(Fruit.GRAPE_FRUIT.toString()); // "Grape Fruit"
```

Section 35.17: Enums with static fields

If your enum class is required to have static fields, keep in mind they are created **after** the enum values themselves. That means, the following code will result in a NullPointerException:

```
enum Example {  
    ONE(1), TWO(2);  
  
    static Map<String, Integer> integers = new HashMap<>();  
  
    private Example(int value) {
```

```

integers.put(this.name(), value);
}
}

```

一种可能的修复方法：

```

enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
}

```

不要初始化静态字段：

```

enum Example {
    ONE(1), TWO(2);

    // 初始化后 integers 仍为 null ! !
    static Map<String, Integer> integers = null;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }

    // !!这可能导致空指针异常 ! !
    public int getValue(){
        return (Example.integers.get(this.name()));
    }
}

```

初始化：

- 创建枚举值
 - 作为副作用，调用了 `putValue()`，它初始化了整数
- 静态值被设置
 - `integers = null;` // 在枚举之后执行，因此 `integers` 的内容丢失

```

        integers.put(this.name(), value);
    }
}

```

A possible way to fix this:

```

enum Example {
    ONE(1), TWO(2);

    static Map<String, Integer> integers;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }
}

```

Do not initialize the static field:

```

enum Example {
    ONE(1), TWO(2);

    // after initialisation integers is null!!
    static Map<String, Integer> integers = null;

    private Example(int value) {
        putValue(this.name(), value);
    }

    private static void putValue(String name, int value) {
        if (integers == null)
            integers = new HashMap<>();
        integers.put(name, value);
    }

    // !!this may lead to null pointer exception ! !
    public int getValue(){
        return (Example.integers.get(this.name()));
    }
}

```

initialisation:

- create the enum values
 - as side effect `putValue()` called that initializes `integers`
- the static values are set
 - `integers = null;` // is executed after the enums so the content of `integers` is lost

第36章：枚举映射

Java EnumMap 类是针对枚举键的专用 Map 实现。它继承自 Enum 和 AbstractMap 类。

java.util.EnumMap 类的参数。

K : 这是该映射维护的键的类型。V : 这是映射值的类型。

第36.1节：枚举映射图书示例

```
import java.util.*;
class Book {
    int id;
    String name,author,publisher;
    int 数量;
    public 书籍(int 编号, String 名称, String 作者, String 出版社, int 数量) {
        this.编号 = 编号;
        this.名称 = 名称;
        this.作者 = 作者;
        this.出版社 = 出版社;
        this.数量 = 数量;
    }
}
public class 枚举映射示例 {
    // 创建枚举
    public enum 键{
        一, 二, 三
    };
    public static void main(String[] 参数) {
        // 创建书籍
        书籍 b1=new 书籍(101,"让我们学C","亚什旺特·卡内特卡尔","BPB",8);
        书籍 b2=new 书籍(102,"数据通信与网络","福鲁赞","麦格劳·希尔",4);
        书籍 b3=new 书籍(103,"操作系统","加尔文","威利",6);
        // 将书籍添加到映射
        映射.put(键.一, b1);
        映射.put(键.二, b2);
        映射.put(键.三, b3);
        // 遍历 EnumMap
        for(Map.Entry<Key, Book> entry:map.entrySet()){
            Book b=entry.getValue();
            System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
        }
    }
}
```

Chapter 36: Enum Map

Java EnumMap class is the specialized Map implementation for enum keys. It inherits Enum and AbstractMap classes.

the Parameters for java.util.EnumMap class.

K: It is the type of keys maintained by this map. V: It is the type of mapped values.

Section 36.1: Enum Map Book Example

```
import java.util.*;
class Book {
    int id;
    String name,author,publisher;
    int quantity;
    public Book(int id, String name, String author, String publisher, int quantity) {
        this.id = id;
        this.name = name;
        this.author = author;
        this.publisher = publisher;
        this.quantity = quantity;
    }
}
public class EnumMapExample {
    // Creating enum
    public enum Key{
        One, Two, Three
    };
    public static void main(String[] args) {
        // Creating Books
        Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
        Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
        Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
        // Adding Books to Map
        map.put(Key.One, b1);
        map.put(Key.Two, b2);
        map.put(Key.Three, b3);
        // Traversing EnumMap
        for(Map.Entry<Key, Book> entry:map.entrySet()){
            Book b=entry.getValue();
            System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
        }
    }
}
```

第37章：EnumSet类

Java EnumSet类是专门用于枚举类型的Set实现。它继承自AbstractSet类并实现了Set接口。

第37.1节：EnumSet示例

```
import java.util.*;
enum days {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
    public static void main(String[] args) {
        Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
        // 遍历元素
        Iterator<days> iter = set.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

Chapter 37: EnumSet class

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

Section 37.1: Enum Set Example

```
import java.util.*;
enum days {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
    public static void main(String[] args) {
        Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
        // Traversing elements
        Iterator<days> iter = set.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

第38章：以数字开头的枚举

Java 不允许枚举名以数字开头，如 100A、25K。在这种情况下，我们可以在代码中添加下划线（_）或任何允许的模式，并进行相应检查。

第38.1节：以名称开头的枚举

```
public enum BookCode {  
    _10A("Simon Haykin", "通信系统"),  
    _42B("Stefan Hakins", "时间简史"),  
    E1("Sedra Smith", "电子电路");  
  
    private String author;  
    private String title;  
  
    BookCode(String author, String title) {  
        this.author = author;  
        this.title = title;  
    }  
  
    public String getName() {  
        String name = name();  
        if (name.charAt(0) == '_') {  
            name = name.substring(1, name.length());  
        }  
        return name;  
    }  
  
    public static BookCode of(String code) {  
        if (Character.isDigit(code.charAt(0))) {  
            code = "_" + code;  
        }  
        return BookCode.valueOf(code);  
    }  
}
```

Chapter 38: Enum starting with number

Java does not allow the name of enum to start with number like 100A, 25K. In that case, we can append the code with _ (underscore) or any allowed pattern and make check of it.

Section 38.1: Enum with name at beginning

```
public enum BookCode {  
    _10A("Simon Haykin", "Communication System"),  
    _42B("Stefan Hakins", "A Brief History of Time"),  
    E1("Sedra Smith", "Electronics Circuits");  
  
    private String author;  
    private String title;  
  
    BookCode(String author, String title) {  
        this.author = author;  
        this.title = title;  
    }  
  
    public String getName() {  
        String name = name();  
        if (name.charAt(0) == '_') {  
            name = name.substring(1, name.length());  
        }  
        return name;  
    }  
  
    public static BookCode of(String code) {  
        if (Character.isDigit(code.charAt(0))) {  
            code = "_" + code;  
        }  
        return BookCode.valueOf(code);  
    }  
}
```

第39章：哈希表

哈希表（**Hashtable**）是Java集合中的一个类，实现了Map接口并继承了Dictionary类

只包含唯一元素且是同步的

第39.1节：哈希表

```
import java.util.*;
public class HashtableDemo {
    public static void main(String args[]) {
        // 创建并填充哈希表
        Hashtable<Integer, String> map = new Hashtable<Integer, String>();
        map.put(101, "C语言");
        map.put(102, "域");
        map.put(104, "数据库");
        System.out.println("删除前的值: "+ map);
        // 删除键102对应的值
        map.remove(102);
        System.out.println("删除后的值: "+ map);
    }
}
```

Chapter 39: Hashtable

Hashtable is a class in Java collections which implements Map interface and extends the Dictionary Class

Contains only unique elements and its synchronized

Section 39.1: Hashtable

```
import java.util.*;
public class HashtableDemo {
    public static void main(String args[]) {
        // create and populate hash table
        Hashtable<Integer, String> map = new Hashtable<Integer, String>();
        map.put(101, "C Language");
        map.put(102, "Domain");
        map.put(104, "Databases");
        System.out.println("Values before remove: "+ map);
        // Remove value for key 102
        map.remove(102);
        System.out.println("Values after remove: "+ map);
    }
}
```

第40章：运算符

Java编程语言中的运算符是执行特定操作的特殊符号，这些操作作用于一个、两个或三个操作数，然后返回一个结果。

第40.1节：自增/自减运算符 (++/--)

变量可以使用++和--运算符分别增加或减少1。

当++和--运算符跟在变量后面时，分别称为后置递增和后置递减。

```
int a = 10;  
a++; // a 现在等于 11  
a--; // a 现在又等于 10
```

当 ++ 和 -- 运算符位于变量之前时，这些操作分别称为 **前置递增** 和 **前置递减**

```
int x = 10;  
--x; // x 现在等于 9  
++x; // x 现在等于 10
```

如果运算符位于变量之前，表达式的值是变量递增或递减后的值。如果运算符位于变量之后，表达式的值是变量递增或递减之前的值。

```
int x=10;  
  
System.out.println("x=" + x + " x=" + x++ + " x=" + x); // 输出 x=10 x=10 x=11  
System.out.println("x=" + x + " x=" + ++x + " x=" + x); // 输出 x=11 x=12 x=12  
System.out.println("x=" + x + " x=" + x-- + " x=" + x); // 输出 x=12 x=12 x=11  
System.out.println("x=" + x + " x=" + --x + " x=" + x); // 输出 x=11 x=10 x=10
```

注意不要覆盖后置递增或递减。这种情况发生在你在表达式末尾使用后置递增/递减运算符，并将结果重新赋值给该递增/递减变量本身时。递增/递减操作将不会生效。尽管左侧变量被正确递增，但其值会立即被表达式右侧先前计算的结果覆盖：

```
int x = 0;  
x = x++ + 1 + x++; // x = 0 + 1 + 1  
                    // 不要这样做——最后的自增没有效果（错误！）  
System.out.println(x); // 输出 2（不是 3！）
```

正确写法：

```
int x = 0;  
x = x++ + 1 + x; // 计算结果为 x = 0 + 1 + 1  
x++; // 加 1  
System.out.println(x); // 输出 3
```

第40.2节：条件运算符 (? :)

语法

Chapter 40: Operators

Operators in Java programming language are special symbols that perform specific operations on one, two, or three operands, and then return a result.

Section 40.1: The Increment/Decrement Operators (++/--)

Variables can be incremented or decremented by 1 using the ++ and -- operators, respectively.

When the ++ and -- operators follow variables, they are called **post-increment** and **post-decrement** respectively.

```
int a = 10;  
a++; // a now equals 11  
a--; // a now equals 10 again
```

When the ++ and -- operators precede the variables the operations are called **pre-increment** and **pre-decrement** respectively.

```
int x = 10;  
--x; // x now equals 9  
++x; // x now equals 10
```

If the operator precedes the variable, the value of the expression is the value of the variable after being incremented or decremented. If the operator follows the variable, the value of the expression is the value of the variable prior to being incremented or decremented.

```
int x=10;  
  
System.out.println("x=" + x + " x=" + x++ + " x=" + x); // outputs x=10 x=10 x=11  
System.out.println("x=" + x + " x=" + ++x + " x=" + x); // outputs x=11 x=12 x=12  
System.out.println("x=" + x + " x=" + x-- + " x=" + x); // outputs x=12 x=12 x=11  
System.out.println("x=" + x + " x=" + --x + " x=" + x); // outputs x=11 x=10 x=10
```

Be careful not to overwrite post-increments or decrements. This happens if you use a post-in/decrement operator at the end of an expression which is reassigned to the in/decremented variable itself. The in/decrement will not have an effect. Even though the variable on the left hand side is incremented correctly, its value will be immediately overwritten with the previously evaluated result from the right hand side of the expression:

```
int x = 0;  
x = x++ + 1 + x++; // x = 0 + 1 + 1  
                    // do not do this - the last increment has no effect (bug!)  
System.out.println(x); // prints 2 (not 3!)
```

Correct:

```
int x = 0;  
x = x++ + 1 + x; // evaluates to x = 0 + 1 + 1  
x++; // adds 1  
System.out.println(x); // prints 3
```

Section 40.2: The Conditional Operator (? :)

Syntax

{condition-to-evaluate} ? {statement-executed-on-true} : {statement-executed-on-false}

如语法所示，条件运算符（也称为三元运算符1）使用?（问号）和:（冒号）字符来实现两个可能结果的条件表达式。它可以用来替代较长的if-else代码块，根据条件返回两个值中的一个。

```
result = testCondition ? value1 : value2
```

等价于

```
如果 (测试条件) {  
    结果 = 值1;  
} 否则 {  
    结果 = 值2;  
}
```

可以理解为“如果测试条件为真，则将结果设为值1；否则，将结果设为值2”。

例如：

```
// 使用条件运算符获取绝对值  
a = -10;  
int 绝对值 = a < 0 ? -a : a;  
System.out.println("abs = " + 绝对值); // 输出 "abs = 10"
```

等价于

```
// 使用 if/else 语句获取绝对值  
a = -10;  
int 绝对值;  
if (a < 0) {  
    绝对值 = -a;  
} 否则 {  
    absValue = a;  
}  
System.out.println("abs = " + 绝对值); // 输出 "abs = 10"
```

常见用法

你可以使用条件运算符进行条件赋值（例如空值检查）。

```
String x = y != null ? y.toString() : ""; //其中 y 是一个对象
```

此示例等同于：

```
String x = "";  
  
if (y != null) {  
    x = y.toString();  
}
```

由于条件运算符的优先级是倒数第二，仅高于赋值运算符，通常不需要在condition周围加括号，但当条件运算符与其他运算符结合使用时，整个条件运算符结构需要加括号：

操作符结构与其他运算符结合时：

```
// 三部分表达式中不需要括号
```

{condition-to-evaluate} ? {statement-executed-on-true} : {statement-executed-on-false}

As shown in the syntax, the Conditional Operator (also known as the Ternary Operator1) uses the ? (question mark) and : (colon) characters to enable a conditional expression of two possible outcomes. It can be used to replace longer if-else blocks to return one of two values based on condition.

```
result = testCondition ? value1 : value2
```

Is equivalent to

```
if (testCondition) {  
    result = value1;  
} else {  
    result = value2;  
}
```

It can be read as “**If testCondition is true, set result to value1; otherwise, set result to value2**”.

For example:

```
// get absolute value using conditional operator  
a = -10;  
int absValue = a < 0 ? -a : a;  
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Is equivalent to

```
// get absolute value using if/else loop  
a = -10;  
int absValue;  
if (a < 0) {  
    absValue = -a;  
} else {  
    absValue = a;  
}  
System.out.println("abs = " + absValue); // prints "abs = 10"
```

Common Usage

You can use the conditional operator for conditional assignments (like null checking).

```
String x = y != null ? y.toString() : ""; //where y is an object
```

This example is equivalent to:

```
String x = "";  
  
if (y != null) {  
    x = y.toString();  
}
```

Since the Conditional Operator has the second-lowest precedence, above the Assignment Operators, there is rarely a need for use parenthesis around the condition, but parenthesis is required around the entire Conditional Operator construct when combined with other operators:

```
// no parenthesis needed for expressions in the 3 parts
```

```
10 <= a && a < 19 ? b * 5 : b * 7
```

```
// 需要括号  
7 * (a > 0 ? 2 : 5)
```

条件运算符的嵌套也可以在第三部分完成，这里它更像是链式调用或类似于switch语句。

```
a ? "a 为真":  
b ? "a 为假, b 为真":  
c ? "a 和 b 都为假, c 为真":  
    "a、b 和 c 都为假"
```

```
//运算符优先级可以用括号来说明：
```

```
a ? x : (b ? y : (c ? z : w))
```

脚注：

1 - 《Java语言规范》和《Java教程》都称 (?) 运算符为条件运算符。《教程》称它“也被称为三元运算符”，因为它是（目前）Java定义的唯一三元运算符。“条件运算符”这一术语与C、C++及其他具有等效运算符的语言保持一致。

```
10 <= a && a < 19 ? b * 5 : b * 7
```

```
// parenthesis required  
7 * (a > 0 ? 2 : 5)
```

Conditional operators nesting can also be done in the third part, where it works more like chaining or like a switch statement.

```
a ? "a is true":  
b ? "a is false, b is true":  
c ? "a and b are false, c is true":  
    "a, b, and c are false"
```

```
//operator precedence can be illustrated with parenthesis:
```

```
a ? x : (b ? y : (c ? z : w))
```

Footnote:

1 - Both the [Java Language Specification](#) and the [Java Tutorial](#) call the (?) operator the *Conditional Operator*. The Tutorial says that it is "also known as the Ternary Operator" as it is (currently) the only ternary operator defined by Java. The "Conditional Operator" terminology is consistent with C and C++ and other languages with an equivalent operator.

第40.3节：按位与逻辑运算符 (~, &, |, ^)

Java语言提供了4个对整数或布尔操作数执行按位或逻辑运算的运算符。

- 补码 (~) 运算符是一个一元运算符，对一个操作数的位进行按位或逻辑取反；参见JLS 15.15.5。。
- 与 (&) 运算符是一个二元运算符，对两个操作数执行按位或逻辑“与”运算；参见JLS 15.22.2。。
- 或 (|) 运算符是一个二元运算符，对两个操作数执行按位或逻辑“包含或”运算；参见 [JLS 15.22.2](#)。。
- 异或 (^) 运算符是一个二元运算符，对两个操作数执行按位或逻辑“异或”运算；参见 [JLS 15.22.2](#)。。

当操作数为布尔值时，这些运算符执行的逻辑运算可总结如下：

A	B	~A	A & B	A B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

请注意，对于整数操作数，上表描述的是单个位的情况。实际上，运算符是对操作数的所有32位或64位同时进行操作的。

操作数类型和结果类型。

当操作数为整数时，适用通常的算术转换。按位运算符的常见用例

~ 运算符用于反转布尔值，或更改整数操作数中的所有位。

A	B	~A	A & B	A B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Note that for integer operands, the above table describes what happens for individual bits. The operators actually operate on all 32 or 64 bits of the operand or operands in parallel.

Operand types and result types.

The usual arithmetic conversions apply when the operands are integers. Common use-cases for the bitwise operators

The ~ operator is used to reverse a boolean value, or change all the bits in an integer operand.

& 运算符用于“屏蔽”整数操作数中的某些位。例如：

```
int word = 0b00101010;
int mask = 0b00000011; // 用于屏蔽除最低两位以外所有位的掩码

int lowBits = word & mask; // -> 0b00000010
int highBits = word & ~mask; // -> 0b00101000
```

| 运算符用于组合两个操作数的真值。例如：

```
int word2 = 0b01011111;
// 将 word1 的最低两位与 word2 的最高 30 位组合
int combined = (word & mask) | (word2 & ~mask); // -> 0b01011110
```

The ^ 运算符用于切换或“翻转”位：

```
int word3 = 0b00101010;
int word4 = word3 ^ mask; // -> 0b00101001
```

有关位运算符使用的更多示例，请参见位操作 (Bit Manipulation)

第40.4节：字符串连接运算符 (+)

符号 + 在Java中可以表示三种不同的运算符：

- 如果 + 前没有操作数，则它是单目加号运算符。
- 如果有两个操作数，且它们都是数字，则它是二元加法运算符。
- 如果有两个操作数，且至少有一个是String，则它是二元连接运算符。

在简单情况下，连接运算符将两个字符串连接成第三个字符串。例如：

```
String s1 = "一个字符串";
String s2 = "这是 " + s1; // s2 包含 "这是一个字符串"
```

当两个操作数中有一个不是字符串时，它会被转换为String，转换规则如下：

- 类型为原始型的操作数会被转换，仿佛通过调用装箱值的toString()方法进行转换。
- 类型为引用型的操作数通过调用操作数的toString()方法进行转换。如果操作数为null，或者toString()方法返回null，则使用字符串字面量"null"代替。

例如：

```
int one = 1;
String s3 = "一个 is " + one; // s3 包含 "一个 is 1"
String s4 = null + " 是 null"; // s4 包含 "null 是 null"
String s5 = "{1} is " + new int[]{1}; // s5 包含类似
// "{} is [I@xxxxxxxxx"
```

关于 s5示例的解释是，数组类型的toString()方法继承自java.lang.Object，其行为是生成一个由类型名和对象的身份哈希码组成的字符串。

连接运算符被规定为创建一个新的String对象，除非表达式是一个常量表达式。在后一种情况下，表达式在编译时求值，其运行时值

The & operator is used for "masking out" some of the bits in an integer operand. For example:

```
int word = 0b00101010;
int mask = 0b00000011; // Mask for masking out all but the bottom
// two bits of a word
int lowBits = word & mask; // -> 0b00000010
int highBits = word & ~mask; // -> 0b00101000
```

The | operator is used to combine the truth values of two operands. For example:

```
int word2 = 0b01011111;
// Combine the bottom 2 bits of word1 with the top 30 bits of word2
int combined = (word & mask) | (word2 & ~mask); // -> 0b01011110
```

The ^ operator is used for toggling or "flipping" bits:

```
int word3 = 0b00101010;
int word4 = word3 ^ mask; // -> 0b00101001
```

For more examples of the use of the bitwise operators, see Bit Manipulation

Section 40.4: The String Concatenation Operator (+)

The + symbol can mean three distinct operators in Java:

- If there is no operand before the +, then it is the unary Plus operator.
- If there are two operands, and they are both numeric, then it is the binary Addition operator.
- If there are two operands, and at least one of them is a String, then it is the binary Concatenation operator.

In the simple case, the Concatenation operator joins two strings to give a third string. For example:

```
String s1 = "a String";
String s2 = "This is " + s1; // s2 contains "This is a String"
```

When one of the two operands is not a string, it is converted to a String as follows:

- An operand whose type is a primitive type is converted as if by calling toString() on the boxed value.
- An operand whose type is a reference type is converted by calling the operand's toString() method. If the operand is null, or if the toString() method returns null, then the string literal "null" is used instead.

For example:

```
int one = 1;
String s3 = "One is " + one; // s3 contains "One is 1"
String s4 = null + " is null"; // s4 contains "null is null"
String s5 = "{1} is " + new int[]{1}; // s5 contains something like
// "{} is [I@xxxxxxxxx"
```

The explanation for the s5 example is that the toString() method on array types is inherited from java.lang.Object, and the behavior is to produce a string that consists of the type name, and the object's identity hashcode.

The Concatenation operator is specified to create a new String object, except in the case where the expression is a Constant Expression. In the latter case, the expression is evaluated at compile type, and its runtime value is

等同于字符串字面量。这意味着在拆分像下面这样长的字符串字面量时没有运行时开销：

```
String typing = "快速的棕色狐狸 " +
    "跳过了 " +
    "lazy dog";           // 常量表达式
```

优化与效率

如上所述，除常量表达式外，每个字符串连接表达式都会创建一个新的 String 对象。考虑以下代码：

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

在上述方法中，循环的每次迭代都会创建一个比上一次迭代长一个字符的新 String。每次连接都会复制操作数字字符串中的所有字符以形成新的 String。

因此，stars(N) 将会：

- 创建 N 个新的 String 对象，并丢弃除最后一个之外的所有对象，
- 复制 $N * (N + 1) / 2$ 个字符，且
- 生成 $O(N^2)$ 字节的垃圾数据。

对于大的N来说，这非常昂贵。实际上，任何在循环中连接字符串的代码都可能遇到这个问题。

更好的写法如下：

```
public String stars(int count) {
    // 创建一个容量为 'count' 的字符串构建器
    StringBuilder sb = new StringBuilder(count);
    for (int i = 0; i < count; i++) {
        sb.append("*");
    }
    return sb.toString();
}
```

理想情况下，你应该设置StringBuilder的容量，但如果这不现实，该类会自动增长构建器用来存储字符的底层数组。（注意：实现是指数级扩展底层数组。这种策略将字符复制的数量保持在O(N)而非O(N^2)。）

有些人将这种模式应用于所有字符串连接。然而，这没有必要，因为JLS允许Java编译器优化单个表达式内的字符串连接。例如：

```
String s1 = ...;
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + ";;
```

通常会被字节码编译器优化成类似如下的代码；

```
StringBuilder tmp = new StringBuilder();
tmp.append("Hello ")
tmp.append(s1 == null ? "null" + s1);
tmp.append("Welcome to ");
```

equivalent to a string literal. This means that there is no runtime overhead in splitting a long string literal like this:

```
String typing = "The quick brown fox " +
    "jumped over the " +
    "lazy dog";           // constant expression
```

Optimization and efficiency

As noted above, with the exception of constant expressions, each string concatenation expression creates a new String object. Consider this code:

```
public String stars(int count) {
    String res = "";
    for (int i = 0; i < count; i++) {
        res = res + "*";
    }
    return res;
}
```

In the method above, each iteration of the loop will create a new String that is one character longer than the previous iteration. Each concatenation copies all of the characters in the operand strings to form the new String. Thus, stars(N) will:

- create N new String objects, and throw away all but the last one,
- copy $N * (N + 1) / 2$ characters, and
- generate $O(N^2)$ bytes of garbage.

This is very expensive for large N. Indeed, any code that concatenates strings in a loop is liable to have this problem. A better way to write this would be as follows:

```
public String stars(int count) {
    // Create a string builder with capacity 'count'
    StringBuilder sb = new StringBuilder(count);
    for (int i = 0; i < count; i++) {
        sb.append("*");
    }
    return sb.toString();
}
```

Ideally, you should set the capacity of the StringBuilder, but if this is not practical, the class will automatically grow the backing array that the builder uses to hold characters. (Note: the implementation expands the backing array exponentially. This strategy keeps that amount of character copying to a O(N) rather than O(N^2).)

Some people apply this pattern to all string concatenations. However, this is unnecessary because the JLS allows a Java compiler to optimize string concatenations within a single expression. For example:

```
String s1 = ...;
String s2 = ...;
String test = "Hello " + s1 + ". Welcome to " + s2 + "\n";
```

will typically be optimized by the bytecode compiler to something like this;

```
StringBuilder tmp = new StringBuilder();
tmp.append("Hello ")
tmp.append(s1 == null ? "null" + s1);
tmp.append("Welcome to ");
```

```
tmp.append(s2 == null ? "null" + s2);tmp.ap  
pend("");  
String test = tmp.toString();
```

(如果JIT编译器能推断出 `s1` 或 `s2` 不可能为 `null`, 可能会进一步优化这段代码。) 但请注意, 这种优化仅允许在单个表达式内进行。

简而言之, 如果你关心字符串连接的效率:

- 如果在循环中 (或类似场景) 进行重复连接, 建议手动优化。
- 不要手动优化单个连接表达式。

第40.5节 : 算术运算符 (+, -, *, /, %)

Java语言提供了7个对整数和浮点数值进行算术运算的运算符。

- 有两个+运算符:
 - 二元加法运算符将一个数字加到另一个数字上。 (还有一个二元+运算符用于字符串连接。该内容在另一个示例中描述。)
 - 一元加号运算符除了触发数值提升 (见下文) 外, 不执行任何操作。
- 有两个-运算符:
 - 二元减法运算符将一个数字从另一个数字中减去。
 - 一元减号运算符等同于用零减去其操作数。
- 二元乘法运算符 (*) 将一个数字乘以另一个数字。
- 二元除法运算符 (/) 将一个数字除以另一个数字。
- 二元取余运算符 (%) 计算一个数字除以另一个数字后的余数。

1. 这通常被错误地称为“模运算符”。“余数”是Java语言规范 (JLS) 中使用的术语。

“模”和“余数”不是同一个概念。

操作数和结果类型, 以及数值提升

运算符要求数值类型的操作数, 并产生数值类型的结果。操作数类型可以是任何原始数值类型 (即`byte`、`short`、`char`、`int`、`long`、`float`或`double`) , 也可以是`java.lang`中定义的任何数值包装类型; 即 (`Byte`、`Character`、`Short`、`Integer`、`Long`、`Float`或`Double`) 。

结果类型基于操作数的类型确定, 规则如下:

- 如果任一操作数是`double`或`Double`, 则结果类型为`double`。
- 否则, 如果任一操作数是`float`或`Float`, 则结果类型为`float`。
- 否则, 如果任一操作数是`long`或`Long`, 则结果类型为`long`。
- 否则, 结果类型为`int`。此规则适用于`byte`、`short`和`char`操作数, 以及`int`。

操作的结果类型决定了算术操作的执行方式以及操作数的处理方式。

- 如果结果类型是`double`, 操作数将提升为`double`, 操作使用64位 (双精度二进制) IEEE 754浮点算术执行。
- 如果结果类型是`float`, 操作数将提升为`float`, 操作使用32位 (单精度二进制) IEEE 754浮点算术执行。
- 如果结果类型是`long`, 操作数将提升为`long`, 并使用64位带符号二进制补码整数算术进行运算。
- 如果结果类型是`int`, 操作数将被提升为`int`, 操作使用32位有符号二进制补码整数算术进行。

```
tmp.append(s2 == null ? "null" + s2);  
tmp.append("\n");  
String test = tmp.toString();
```

(The JIT compiler may optimize that further if it can deduce that `s1` or `s2` cannot be `null`.) But note that this optimization is only permitted within a single expression.

In short, if you are concerned about the efficiency of string concatenations:

- Do hand-optimize if you are doing repeated concatenation in a loop (or similar).
- Don't hand-optimize a single concatenation expression.

Section 40.5: The Arithmetic Operators (+, -, *, /, %)

The Java language provides 7 operators that perform arithmetic on integer and floating point values.

- There are two + operators:
 - The binary addition operator adds one number to another one. (There is also a binary + operator that performs string concatenation. That is described in a separate example.)
 - The unary plus operator does nothing apart from triggering numeric promotion (see below)
- There are two - operators:
 - The binary subtraction operator subtracts one number from another one.
 - The unary minus operator is equivalent to subtracting its operand from zero.
- The binary multiply operator (*) multiplies one number by another.
- The binary divide operator (/) divides one number by another.
- The binary remainder1 operator (%) calculates the remainder when one number is divided by another.

1. This is often incorrectly referred to as the "modulus" operator. "Remainder" is the term that is used by the JLS. "Modulus" and "remainder" are not the same thing.

Operand and result types, and numeric promotion

The operators require numeric operands and produce numeric results. The operand types can be any primitive numeric type (i.e. `byte`, `short`, `char`, `int`, `long`, `float` or `double`) or any numeric wrapper type define in `java.lang`; i.e. (`Byte`, `Character`, `Short`, `Integer`, `Long`, `Float` or `Double`).

The result type is determined base on the types of the operand or operands, as follows:

- If either of the operands is a `double` or `Double`, then the result type is `double`.
- Otherwise, if either of the operands is a `float` or `Float`, then the result type is `float`.
- Otherwise, if either of the operands is a `long` or `Long`, then the result type is `long`.
- Otherwise, the result type is `int`. This covers `byte`, `short` and `char` operands as well as `int`.

The result type of the operation determines how the arithmetic operation is performed, and how the operands are handled

- If the result type is `double`, the operands are promoted to `double`, and the operation is performed using 64-bit (double precision binary) IEE 754 floating point arithmetic.
- If the result type is `float`, the operands are promoted to `float`, and the operation is performed using 32-bit (single precision binary) IEE 754 floating point arithmetic.
- If the result type is `long`, the operands are promoted to `long`, and the operation is performed using 64-bit signed twos-complement binary integer arithmetic.
- If the result type is `int`, the operands are promoted to `int`, and the operation is performed using 32-bit signed twos-complement binary integer arithmetic.

提升分两个阶段进行：

- 如果操作数类型是包装类型，操作数值将被unboxed为对应的原始类型值。
- 如有必要，原始类型将被提升为所需类型：
 - 整数提升为int或long是无损的。
 - 将float提升为double是无损的。
 - 将整数提升为浮点值可能导致精度丢失。转换使用IEEE 768“舍入到最近值”语义进行。

除法的含义

/ 运算符将左操作数 n (被除数) 除以右操作数 d (除数)，并产生结果 q (商)。

Java 整数除法向零舍入。Java 语言规范 (JLS) 第 15.17.2 节规定了 Java 整数除法的行为如下：

对于操作数 n 和 d，商 q 是一个整数值，其绝对值尽可能大，同时满足 $|d \cdot q| \leq |n|$ 。此外，当 $|n| \geq |d|$ 且 n 和 d 同号时，q 为正；当 $|n| \geq |d|$ 且 n 和 d 异号时，q 为负。

有几个特殊情况：

- 如果 n 是 MIN_VALUE，且除数为 -1，则会发生整数溢出，结果为 MIN_VALUE。此情况下不会抛出异常。
- 如果 d 为 0，则抛出 `ArithmeticException`。

Java 浮点除法有更多边界情况需要考虑。但基本思想是结果 q 是最接近满足 $d \cdot q = n$ 的值。

浮点除法永远不会导致异常。相反，除以零的操作会产生 INF 和 NaN 值；详见下文。

余数的含义

与 C 和 C++ 不同，Java 中的余数运算符既适用于整数运算，也适用于浮点运算。

对于整数情况， $a \% b$ 的结果定义为数值 r，满足 $(a / b) * b + r$ 等于 a，其中 /、* 和 + 是相应的 Java 整数运算符。除非 b 为零，其他情况均适用。若 b 为零，则余数运算会抛出 ArithmeticException。

由上述定义可知， $a \% b$ 只有在 a 为负时才可能为负，且只有在 a 为正时才可能为正。此外， $a \% b$ 的绝对值始终小于 b 的绝对值。

浮点余数运算是整数情况的推广。表达式 $a \% b$ 的结果余数 r 由数学关系定义为 $r = a - (b \cdot q)$ ，其中：

- q 是一个整数，
- 仅当 a / b 为负时才为负，仅当 a / b 为正时才为正，且其绝对值尽可能大，
- 但不超过 a 和 b 的真实数学商的绝对值。

Promotion is performed in two stages:

- If the operand type is a wrapper type, the operand value is unboxed to a value of the corresponding primitive type.
- If necessary, the primitive type is promoted to the required type:
 - Promotion of integers to int or long is loss-less.
 - Promotion of float to double is loss-less.
 - Promotion of an integer to a floating point value can lead to loss of precision. The conversion is performed using IEE 768 "round-to-nearest" semantics.

The meaning of division

The / operator divides the left-hand operand n (the dividend) and the right-hand operand d (the divisor) and produces the result q (the quotient).

Java integer division rounds towards zero. The [JLS Section 15.17.2](#) specifies the behavior of Java integer division as follows:

The quotient produced for operands n and d is an integer value q whose magnitude is as large as possible while satisfying $|d \cdot q| \leq |n|$. Moreover, q is positive when $|n| \geq |d|$ and n and d have the same sign, but q is negative when $|n| \geq |d|$ and n and d have opposite signs.

There are a couple of special cases:

- If the n is MIN_VALUE, and the divisor is -1, then integer overflow occurs and the result is MIN_VALUE. No exception is thrown in this case.
- If d is 0, then `ArithmeticException` is thrown.

Java floating point division has more edge cases to consider. However the basic idea is that the result q is the value that is closest to satisfying $d \cdot q = n$.

Floating point division will never result in an exception. Instead, operations that divide by zero result in an INF and NaN values; see below.

The meaning of remainder

Unlike C and C++, the remainder operator in Java works with both integer and floating point operations.

For integer cases, the result of $a \% b$ is defined to be the number r such that $(a / b) * b + r$ is equal to a, where /, * and + are the appropriate Java integer operators. This applies in all cases except when b is zero. That case, remainder results in an [ArithmeticException](#).

It follows from the above definition that $a \% b$ can be negative only if a is negative, and it be positive only if a is positive. Moreover, the magnitude of $a \% b$ is always less than the magnitude of b.

Floating point remainder operation is a generalization of the integer case. The result of $a \% b$ is the remainder r is defined by the mathematical relation $r = a - (b \cdot q)$ where:

- q is an integer,
- it is negative only if a / b is negative and positive only if a / b is positive, and
- its magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of a and b.

浮点余数在边界情况下（例如 b 为零时）可能产生 INF 和 NaN 值；详见下文。它不会抛出异常。

重要提示：

由 % 计算的浮点余数运算结果与 IEEE 754 定义的余数运算结果不同。IEEE 754 余数可使用 Math.IEEEremainder 库方法计算。

整数溢出

Java 32 位和 64 位整数值为有符号数，采用二进制补码表示。例如，(32 位) int 可表示的数值范围为 -2^{31} 到 $+2^{31} - 1$ 。

当你对两个 N 位整数 ($N == 32$ 或 64) 进行加、减或乘运算时，运算结果可能过大，无法用 N 位整数表示。在这种情况下，运算会导致 整数溢出，结果可按如下方式计算：

- 数学运算用于给出整个数字的中间二进制补码表示。该表示将比N位更长。
- 中间表示的最低32位或64位被用作结果。

应注意，整数溢出在任何情况下都不会导致异常。

浮点数的INF和NaN值

Java 使用 IEEE 754 浮点数表示法来表示 float 和 double。这些表示法有一些特殊值，用于表示超出实数域的值：

- “无限”或INF值表示数值过大。 $+INF$ 值表示过大且为正的数。 $-INF$ 值表示过大且为负的数。
- “未定”/“非数”或NaN表示由无意义运算产生的值。

INF值由导致溢出的浮点运算或除以零产生。

NaN值由零除以零或计算零余零产生。

令人惊讶的是，可以使用INF和NaN操作数进行算术运算而不会触发异常。例如：

- 将 $+INF$ 与有限值相加结果为 $+INF$ 。
- 将 $+INF$ 与 $+INF$ 相加得到 $+INF$ 。
- 将 $+INF$ 与 $-INF$ 相加得到 NaN 。
- 除以 INF 得到 $+0.0$ 或 -0.0 。
- 所有包含一个或多个 NaN 操作数的运算结果均为 NaN 。

有关详细信息，请参阅 JLS 15 的相关小节。请注意，这在很大程度上是“学术性”的。对于典型的计算，INF 或 NaN 表示出现了错误；例如，输入数据不完整或不正确，或者计算程序编写有误。

第 40.6 节：移位运算符 (<<、>> 和 >>>)

Java 语言提供了三种运算符，用于对 32 位和 64 位整数值执行按位移位操作。这些

Floating point remainder can produce INF and NaN values in edge-cases such as when b is zero; see below. It will not throw an exception.

Important note:

The result of a floating-point remainder operation as computed by % **is not the same** as that produced by the remainder operation defined by IEEE 754. The IEEE 754 remainder may be computed using the `Math.IEEEremainder` library method.

Integer Overflow

Java 32 和 64 位整数值是带符号数，使用二进制补码表示。例如，(32 位) int 可表示的数值范围为 -2^{31} 到 $+2^{31} - 1$ 。

当你对两个 N 位整数 ($N == 32$ 或 64) 进行加、减或乘运算时，运算结果可能过大，无法用 N 位整数表示。在这种情况下，运算会导致 *integer overflow*，结果可按如下方式计算：

- The mathematical operation is performed to give a intermediate two's-complement representation of the entire number. This representation will be larger than N bits.
- The bottom 32 or 64 bits of the intermediate representation are used as the result.

It should be noted that integer overflow does not result in exceptions under any circumstances.

Floating point INF and NAN values

Java 使用 IEEE 754 浮点数表示法来表示 float 和 double。这些表示法有一些特殊值，用于表示超出实数域的值：

- The “infinite” or INF values denote numbers that are too large. The $+INF$ value denote numbers that are too large and positive. The $-INF$ value denote numbers that are too large and negative.
- The “indefinite” / “not a number” or NaN denote values resulting from meaningless operations.

The INF values are produced by floating operations that cause overflow, or by division by zero.

The NaN values are produced by dividing zero by zero, or computing zero remainder zero.

Surprisingly, it is possible perform arithmetic using INF and NaN operands without triggering exceptions. For example:

- Adding $+INF$ and a finite value gives $+INF$.
- Adding $+INF$ and $+INF$ gives $+INF$.
- Adding $+INF$ and $-INF$ gives NaN .
- Dividing by INF gives either $+0.0$ or -0.0 .
- All operations with one or more NaN operands give NaN .

For full details, please refer to the relevant subsections of [JLS 15](#). Note that this is largely “academic”. For typical calculations, an INF or NaN means that something has gone wrong; e.g. you have incomplete or incorrect input data, or the calculation has been programmed incorrectly.

Section 40.6: The Shift Operators (<<, >> and >>>)

The Java language provides three operator for performing bitwise shifting on 32 and 64 bit integer values. These

都是二元运算符，第一个操作数是要移位的值，第二个操作数表示移位的位数。

- << 或 左移 运算符将第一个操作数给定的值向 左 移动第二个操作数指定的位数。右端空出的位置用零填充。
- '>>' 或 算术移位 运算符将第一个操作数给定的值向 右 移动第二个操作数指定的位数。左端空出的位置通过复制最左边的位来填充。此过程称为 符号扩展。
- ">>>"或逻辑右移运算符将第一个操作数给定的值向右移动由第二个操作数指定的位数。左端的空位用零填充。

备注：

1. 这些运算符要求第一个操作数为int或long类型的值，并产生与之相同类型的值第一个操作数。（当将移位结果赋值给byte时，您需要使用显式类型转换，short 或 char 变量。）
2. 如果使用移位运算符且第一个操作数是byte、char或short，则该操作数会被提升为int，且操作产生一个int。
3. 第二个操作数被对操作的位数取模以得到移位的数量。
有关模数学概念的更多信息，请参见模数示例。
4. 通过该操作从左端或右端移出的位将被丢弃。（Java 不提供原始的“旋转”操作符。）
5. 算术移位运算符等同于将一个（二进制补码）数除以2的幂。
6. 左移运算符等同于将一个（二进制补码）数乘以2的幂。

下表将帮助您了解三种移位运算符的效果。（数字已用二进制表示以便于观察。）

操作数1	操作数2	<<	>>	>>>
0b00000000000000001011	0	0b00000000000000001011	0b00000000000000001011	0b00000000000000001011
0b00000000000000001011	1	0b000000000000000010110	0b0000000000000000101	0b0000000000000000101
0b00000000000000001011	2	0b0000000000000000101100	0b000000000000000010	0b000000000000000010
0b00000000000000001011	28	0b1011000000000000	0b0000000000000000	0b0000000000000000
0b00000000000000001011	31	0b1000000000000000	0b0000000000000000	0b0000000000000000
0b00000000000000001011	32	0b00000000000000001011	0b00000000000000001011	0b00000000000000001011
...
0b100000000000001011	0	0b100000000000001011	0b100000000000001011	0b100000000000001011
0b100000000000001011	1	0b00000000000010110	0b11000000000000101	0b01000000000000101
0b100000000000001011	2	0b000000000000101100	0b1110000000000010	0b00100000000000100
0b100000000000001011	31	0b1000000000000000	0b111111111111111	0b0000000000000001

以下是位操作中移位运算符的使用示例

第40.7节： instanceof 运算符

该运算符用于检查对象是否属于特定的类/接口类型。 instanceof 运算符的写法为：

are all binary operators with the first operand being the value to be shifted, and the second operand saying how far to shift.

- The << or *left shift* operator shifts the value given by the first operand *leftwards* by the number of bit positions given by the second operand. The empty positions at the right end are filled with zeros.
- The '>>' or *arithmetic shift* operator shifts the value given by the first operand *rightwards* by the number of bit positions given by the second operand. The empty positions at the left end are filled by copying the left-most bit. This process is known as *sign extension*.
- The '>>>' or *logical right shift* operator shifts the value given by the first operand *rightwards* by the number of bit positions given by the second operand. The empty positions at the left end are filled with zeros.

Notes:

1. These operators require an **int** or **long** value as the first operand, and produce a value with the same type as the first operand. (You will need to use an explicit type cast when assigning the result of a shift to a **byte**, **short** or **char** variable.)
2. If you use a shift operator with a first operand that is a **byte**, **char** or **short**, it is promoted to an **int** and the operation produces an **int**.)
3. The second operand is reduced *modulo the number of bits of the operation* to give the amount of the shift. For more about the **mod mathematical concept**, see Modulus examples.
4. The bits that are shifted off the left or right end by the operation are discarded. (Java does not provide a primitive "rotate" operator.)
5. The arithmetic shift operator is equivalent dividing a (two's complement) number by a power of 2.
6. The left shift operator is equivalent multiplying a (two's complement) number by a power of 2.

The following table will help you see the effects of the three shift operators. (The numbers have been expressed in binary notation to aid visualization.)

Operand1	Operand2	<<	>>	>>>
0b00000000000000001011	0	0b00000000000000001011	0b00000000000000001011	0b00000000000000001011
0b00000000000000001011	1	0b000000000000000010110	0b0000000000000000101	0b0000000000000000101
0b00000000000000001011	2	0b0000000000000000101100	0b000000000000000010	0b000000000000000010
0b00000000000000001011	28	0b1011000000000000	0b0000000000000000	0b0000000000000000
0b00000000000000001011	31	0b1000000000000000	0b0000000000000000	0b0000000000000000
0b00000000000000001011	32	0b00000000000000001011	0b00000000000000001011	0b00000000000000001011
...
0b100000000000001011	0	0b100000000000001011	0b100000000000001011	0b100000000000001011
0b100000000000001011	1	0b00000000000010110	0b11000000000000101	0b01000000000000101
0b100000000000001011	2	0b000000000000101100	0b1110000000000010	0b00100000000000100
0b100000000000001011	31	0b1000000000000000	0b111111111111111	0b0000000000000001

以下是位操作中移位运算符的使用示例

Section 40.7: The instanceof Operator

This operator checks whether the object is of a particular class/interface type. **instanceof** operator is written as:

(对象引用变量)instanceof(类/接口类型)

示例：

```
public class Test {  
  
    public static void main(String args[]){  
        String name = "Buyya";  
        // 由于name是String类型，以下将返回true  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

这将产生以下结果：

true

如果被比较的对象与右侧类型在赋值上兼容，该操作符仍然会返回true。

示例：

```
class Vehicle {}  
  
public class Car extends Vehicle {  
    public static void main(String args[]){  
        Vehicle a = new Car();  
        boolean result = a instanceof Car;  
        System.out.println( result );  
    }  
}
```

这将产生以下结果：

true

第40.8节：赋值运算符 (=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, |= 和 ^=)

这些运算符的左操作数必须是非final变量或数组元素。右操作数必须与左操作数赋值兼容。这意味着类型要么相同，要么右操作数类型必须能通过装箱、拆箱或类型提升的组合转换为左操作数类型。（详细内容请参见JLS 5.2。）

“运算并赋值”运算符的具体含义由JLS 15.26.2规定如下：

形式为E1 op= E2的复合赋值表达式等价于E1 = (T) ((E1) op (E2))，
其中T是E1的类型，但E1只被求值一次。

请注意，最终赋值之前存在隐式类型转换。

1. =

(Object reference variable) instanceof (class/interface type)

Example:

```
public class Test {  
  
    public static void main(String args[]){  
        String name = "Buyya";  
        // following will return true since name is type of String  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

This would produce the following result:

true

This operator will still return true if the object being compared is the assignment compatible with the type on the right.

Example:

```
class Vehicle {}  
  
public class Car extends Vehicle {  
    public static void main(String args[]){  
        Vehicle a = new Car();  
        boolean result = a instanceof Car;  
        System.out.println( result );  
    }  
}
```

This would produce the following result:

true

Section 40.8: The Assignment Operators (=, +=, -=, *=, /=, %=, <<=, >>=, >>>=, &=, |= and ^=)

The left hand operand for these operators must be either a non-final variable or an element of an array. The right hand operand must be *assignment compatible* with the left hand operand. This means that either the types must be the same, or the right operand type must be convertible to the left operands type by a combination of boxing, unboxing or widening. (For complete details refer to [JLS 5.2](#).)

The precise meaning of the “operation and assign” operators is specified by [JLS 15.26.2](#) as:

A compound assignment expression of the form E1 op= E2 is equivalent to E1 = (T) ((E1) op (E2)), where T is the type of E1, except that E1 is evaluated only once.

Note that there is an implicit type-cast before the final assignment.

1. =

简单赋值运算符：将右操作数的值赋给左操作数。

示例：`c = a + b` 会将 `a + b` 的值加到 `c` 的值上，并赋值给 `c`

2. `+=`

“加并赋值”运算符：将右操作数的值加到左操作数的值上，并将结果赋给左操作数。如果左操作数的类型是String，则这是一个“连接并赋值”运算符。

示例：`c += a` 大致等同于 `c = c + a`

3. `-=`

“减并赋值”运算符：从左操作数的值中减去右操作数的值，并将结果赋给左操作数。

示例：`c -= a` 大致相当于 `c = c - a`

4. `*=`

“乘并赋值”运算符：将右操作数的值乘以左操作数的值，并将结果赋给左操作数。

示例：`c *= a` 大致相当于 `c = c * a`

5. `/=`

“除并赋值”运算符：将右操作数的值除以左操作数的值，并将结果赋给左操作数。

示例：`c /= a` 大致相当于 `c = c / a`

6. `%=`

“取模并赋值”运算符：计算右操作数的值对左操作数的值的模，并将结果赋给左操作数。

示例：`c %= a` 大致相当于 `c = c % a`

7. `<=`

“左移并赋值”运算符。

示例：`c <= 2` 大致等同于 `c = c << 2`

The simple assignment operator: assigns the value of the right hand operand to the left hand operand.

Example: `c = a + b` will add the value of `a + b` to the value of `c` and assign it to `c`

2. `+=`

The “add and assign” operator: adds the value of right hand operand to the value of the left hand operand and assigns the result to left hand operand. If the left hand operand has type `String`, then this a “concatenate and assign” operator.

Example: `c += a` is roughly the same as `c = c + a`

3. `-=`

The “subtract and assign” operator: subtracts the value of the right operand from the value of the left hand operand and assign the result to left hand operand.

Example: `c -= a` is roughly the same as `c = c - a`

4. `*=`

The “multiply and assign” operator: multiplies the value of the right hand operand by the value of the left hand operand and assign the result to left hand operand ..

Example: `c *= a` is roughly the same as `c = c * a`

5. `/=`

The “divide and assign” operator: divides the value of the right hand operand by the value of the left hand operand and assign the result to left hand operand.

Example: `c /= a` is roughly the same as `c = c / a`

6. `%=`

The “modulus and assign” operator: calculates the modulus of the value of the right hand operand by the value of the left hand operand and assign the result to left hand operand.

Example: `c %= a` is roughly the same as `c = c % a`

7. `<=`

The “left shift and assign” operator.

Example: `c <= 2` is roughly the same as `c = c << 2`

8. >>=

“算术右移并赋值”运算符。

示例：`c >>= 2` 大致等同于 `c = c >> 2`

9. >>>=

“逻辑右移并赋值”运算符。

示例：`c >>>= 2` 大致等同于 `c = c >>> 2`

10. &=

“按位与并赋值”运算符。

示例：`c &= 2` 大致相当于 `c = c & 2`

11. |=

“按位或赋值”运算符。

示例：`c |= 2` 大致相当于 `c = c | 2`

12. ^=

“按位异或赋值”运算符。

示例：`c ^= 2` 大致相当于 `c = c ^ 2`

第40.9节：条件与和条件或运算符（&& 和 ||）

Java 提供了条件与和条件或运算符，这两者都接受一个或两个类型为 `boolean` 的操作数，并产生一个 `boolean` 结果。它们是：

- `&&` - 条件与运算符，
- `||` - 条件或 (conditional-OR) 运算符。对 `<left-expr> && <right-expr>` 的求值等同于以下伪代码：

```
{  
    boolean L = evaluate(<left-expr>);  
    if (L) {  
        return evaluate(<right-expr>);  
    } else {  
        // 短路第二个操作数表达式的求值  
        return false;  
    }  
}
```

8. >>=

The "arithmetic right shift and assign" operator.

Example: `c >>= 2` is roughly the same as `c = c >> 2`

9. >>>=

The "logical right shift and assign" operator.

Example: `c >>>= 2` is roughly the same as `c = c >>> 2`

10. &=

The "bitwise and and assign" operator.

Example: `c &= 2` is roughly the same as `c = c & 2`

11. |=

The "bitwise or and assign" operator.

Example: `c |= 2` is roughly the same as `c = c | 2`

12. ^=

The "bitwise exclusive or and assign" operator.

Example: `c ^= 2` is roughly the same as `c = c ^ 2`

Section 40.9: The conditional-and and conditional-or Operators (&& and ||)

Java provides a conditional-and and a conditional-or operator, that both take one or two operands of type `boolean` and produce a `boolean` result. These are:

- `&&` - the conditional-AND operator,
- `||` - the conditional-OR operators. The evaluation of `<left-expr> && <right-expr>` is equivalent to the following pseudo-code:

```
{  
    boolean L = evaluate(<left-expr>);  
    if (L) {  
        return evaluate(<right-expr>);  
    } else {  
        // short-circuit the evaluation of the 2nd operand expression  
        return false;  
    }  
}
```

}

对 `<left-expr> || <right-expr>` 的求值等价于以下伪代码：

```
{
    boolean L = evaluate(<left-expr>);
    if (!L) {
        return evaluate(<right-expr>);
    } else {
        // 短路第二个操作数表达式的求值
        return true;
    }
}
```

如上伪代码所示，短路运算符的行为等同于使用 `if / else` 语句。

示例 - 在表达式中使用 `&&` 作为保护条件

下面的示例展示了 `&&` 运算符最常见的使用模式。比较这两个方法版本，用于测试传入的 `Integer` 是否为零。

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
}
```

第一个版本在大多数情况下都能正常工作，但如果 `value` 参数为 `null`，则会抛出 `NullPointerException` 异常。

在第二个版本中，我们添加了一个“保护”测试。表达式 `value != null && value == 0` 的计算顺序是先执行 `value != null` 测试。如果 `null` 测试成功（即结果为 `true`），则继续计算 `value == 0` 表达式。如果 `null` 测试失败，则跳过 `value == 0` 的计算（短路），从而避免了 `NullPointerException` 异常。

示例 - 使用 `&&` 避免昂贵的计算

下面的示例展示了如何使用 `&&` 来避免相对昂贵的计算：

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

在第一个版本中，`|` 的两个操作数都会被计算，因此（昂贵的） `isPrime` 方法会被不必要的调用。第二个版本通过使用 `||` 替代 `|` 避免了不必要的调用。

第40.10节：关系运算符 (`<`, `<=`, `>`, `>=`)

运算符 `<`、`<=`、`>` 和 `>=` 是用于比较数值类型的二元运算符。运算符的含义如下

}

The evaluation of `<left-expr> || <right-expr>` is equivalent to the following pseudo-code:

```
{
    boolean L = evaluate(<left-expr>);
    if (!L) {
        return evaluate(<right-expr>);
    } else {
        // short-circuit the evaluation of the 2nd operand expression
        return true;
    }
}
```

As the pseudo-code above illustrates, the behavior of the short-circuit operators are equivalent to using `if / else` statements.

Example - using `&&` as a guard in an expression

The following example shows the most common usage pattern for the `&&` operator. Compare these two versions of a method to test if a supplied `Integer` is zero.

```
public boolean isZero(Integer value) {
    return value == 0;
}

public boolean isZero(Integer value) {
    return value != null && value == 0;
}
```

The first version works in most cases, but if the `value` argument is `null`，then a `NullPointerException` will be thrown.

In the second version we have added a "guard" test. The `value != null && value == 0` expression is evaluated by first performing the `value != null` test. If the `null` test succeeds (i.e. it evaluates to `true`) then the `value == 0` expression is evaluated. If the `null` test fails, then the evaluation of `value == 0` is skipped (short-circuited), and we don't get a `NullPointerException`.

Example - using `&&` to avoid a costly calculation

The following example shows how `&&` can be used to avoid a relatively costly calculation:

```
public boolean verify(int value, boolean needPrime) {
    return !needPrime | isPrime(value);
}

public boolean verify(int value, boolean needPrime) {
    return !needPrime || isPrime(value);
}
```

In the first version, both operands of the `|` will always be evaluated, so the (expensive) `isPrime` method will be called unnecessarily. The second version avoids the unnecessary call by using `||` instead of `|`.

Section 40.10: The Relational Operators (`<`, `<=`, `>`, `>=`)

The operators `<`, `<=`, `>` and `>=` are binary operators for comparing numeric types. The meaning of the operators is as

你会预期的。例如，如果 a 和 b 被声明为 byte、short、char、int、long、float、double 或相应的装箱类型中的任何一种：

- `a < b` 测试 a 的值是否小于 b 的值。
- `a <= b` 测试 a 的值是否小于或等于 b 的值。
- `a > b` 测试 a 的值是否大于 b 的值。
- `a >= b` 测试 a 的值是否大于或等于 b 的值。

这些运算符的结果类型在所有情况下都是 boolean。

关系运算符可以用于比较不同类型的数字。例如：

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

当其中一个或两个数字是装箱数值类型的实例时，可以使用关系运算符。例如：

```
Integer i = 1; // 1 被自动装箱为 Integer
Integer j = 2; // 2 被自动装箱为 Integer
if (i < j) {
    System.out.println("i is smaller");
}
```

具体行为总结如下：

1. 如果其中一个操作数是装箱类型，则会进行拆箱。
2. 如果任一操作数现在是 byte、short 或 char 类型，则会提升为 int 类型。
3. 如果操作数的类型不同，则“较小”类型的操作数会提升为“较大”类型。
4. 比较将在得到的 int、long、float 或 double 值上进行。

涉及浮点数的关系比较时需要特别小心：

- 计算浮点数的表达式通常会产生舍入误差，因为计算机的浮点表示具有有限的精度。
- 当比较整数类型和浮点类型时，将整数转换为浮点数也可能导致舍入误差。

最后，Java 确实不支持使用关系运算符与上述列出的类型以外的任何类型。例如，您不能使用这些运算符来比较字符串、数字数组等。

第40.11节：相等运算符 (==, !=)

== 和 != 运算符是二元运算符，根据操作数是否相等，结果为 true 或 false。== 运算符在操作数相等时返回 true，否则返回 false。!= 运算符在操作数相等时返回 false，否则返回 true。

这些运算符可以用于原始类型和引用类型的操作数，但行为有显著差异。根据 Java 语言规范 (JLS)，实际上有三组不同的这些运算符：

- 布尔类型的 == 和 != 运算符。
- 数值类型的 == 和 != 运算符。

you would expect. For example, if a and b are declared as any of **byte, short, char, int, long, float, double** or the corresponding boxed types:

- `a < b` tests if the value of `a` is less than the value of `b`.
- `a <= b` tests if the value of `a` is less than or equal to the value of `b`.
- `a > b` tests if the value of `a` is greater than the value of `b`.
- `a >= b` tests if the value of `a` is greater than or equal to the value of `b`.

The result type for these operators is **boolean** in all cases.

Relational operators can be used to compare numbers with different types. For example:

```
int i = 1;
long l = 2;
if (i < l) {
    System.out.println("i is smaller");
}
```

Relational operators can be used when either or both numbers are instances of boxed numeric types. For example:

```
Integer i = 1; // 1 is autoboxed to an Integer
Integer j = 2; // 2 is autoboxed to an Integer
if (i < j) {
    System.out.println("i is smaller");
}
```

The precise behavior is summarized as follows:

1. If one of the operands is a boxed type, it is unboxed.
2. If either of the operands now a **byte, short** or **char**, it is promoted to an **int**.
3. If the types of the operands are not the same, then the operand with the "smaller" type is promoted to the "larger" type.
4. The comparison is performed on the resulting **int, long, float** or **double** values.

You need to be careful with relational comparisons that involve floating point numbers:

- Expressions that compute floating point numbers often incur rounding errors due to the fact that the computer floating-point representations have limited precision.
- When comparing an integer type and a floating point type, the conversion of the integer to floating point can also lead to rounding errors.

Finally, Java does not support the use of relational operators with any types other than the ones listed above. For example, you *cannot* use these operators to compare strings, arrays of numbers, and so on.

Section 40.11: The Equality Operators (==, !=)

The == and != operators are binary operators that evaluate to **true** or **false** depending on whether the operands are equal. The == operator gives **true** if the operands are equal and **false** otherwise. The != operator gives **false** if the operands are equal and **true** otherwise.

These operators can be used with operands of primitive and reference types, but the behavior is significantly different. According to the JLS, there are actually three distinct sets of these operators:

- The Boolean == and != operators.
- The Numeric == and != operators.

- 引用类型的 == 和 != 运算符。

但是，在所有情况下，== 和 != 运算符的结果类型都是 boolean。

数值型的 == 和 != 运算符

当 == 或 != 运算符的一个（或两个）操作数是原始数值类型（byte、short、char、int、long、float 或 double）时，该运算符执行数值比较。第二个操作数必须是原始数值类型或装箱数值类型。

其他数值运算符的行为如下：

- 如果其中一个操作数是装箱类型，则会进行拆箱。
- 如果任一操作数现在是 byte、short 或 char 类型，则会提升为 int 类型。
- 如果操作数的类型不同，则“较小”操作数会提升为“较大”类型。
- 比较按以下方式进行：
 - 如果提升后的操作数是 int 或 long，则测试它们的值是否相同。
 - 如果提升后的操作数是 float 或 double，则：
 - 两个零的版本 (+0.0 和 -0.0) 被视为相等
 - NaN 值被视为不等于任何值，且
 - 其他值如果其 IEEE 754 表示相同，则视为相等。

注意：使用 == 和 != 比较浮点值时需要小心。

布尔型的 == 和 != 运算符

如果两个操作数都是布尔型，或者一个是布尔型另一个是 Boolean，则这些运算符是布尔型的==和!=运算符。其行为如下：

- 如果其中一个操作数是 Boolean 类型，则会进行拆箱。
- 对拆箱后的操作数进行测试，并根据以下真值表计算布尔结果

A	B	A == B	A != B
false	false	true	false
false	true	false	true
true	false	false	true
true	true	true	false

有两个“陷阱”使得建议谨慎使用==和!=来比较布尔值：

- 如果使用==或!=来比较两个 Boolean 对象，则使用的是引用操作符。这可能会产生意想不到的结果；参见陷阱：使用==比较包装类对象（如 Integer）
- ==操作符很容易被误写成=。对于大多数操作数类型，这种错误会导致编译错误。然而，对于 boolean 和 Boolean 操作数，这种错误会导致运行时行为不正确；参见陷阱 - 使用'=='测试布尔值

引用类型的==和!=操作符

如果两个操作数都是对象引用，==和!=操作符测试两个操作数是否引用同一个对象。这通常不是你想要的。要测试两个对象是否按值相等，应使用 .equals() 方法代替。

- The Reference == and != operators.

However, in all cases, the result type of the == and != operators is boolean.

The Numeric == and != operators

When one (or both) of the operands of an == or != operator is a primitive numeric type (byte, short, char, int, long, float or double), the operator is a numeric comparison. The second operand must be either a primitive numeric type, or a boxed numeric type.

The behavior other numeric operators is as follows:

- If one of the operands is a boxed type, it is unboxed.
- If either of the operands now a byte, short or char, it is promoted to an int.
- If the types of the operands are not the same, then the operand with the "smaller" type is promoted to the "larger" type.
- The comparison is then carried out as follows:
 - If the promoted operands are int or long then the values are tested to see if they are identical.
 - If the promoted operands are float or double then:
 - the two versions of zero (+0.0 and -0.0) are treated as equal
 - a NaN value is treated as not equals to anything, and
 - other values are equal if their IEEE 754 representations are identical.

Note: you need to be careful when using == and != to compare floating point values.

The Boolean == and != operators

If both operands are boolean, or one is boolean and the other is Boolean, these operators the Boolean == and != operators. The behavior is as follows:

- If one of the operands is a Boolean, it is unboxed.
- The unboxed operands are tested and the boolean result is calculated according to the following truth table

A	B	A == B	A != B
false	false	true	false
false	true	false	true
true	false	false	true
true	true	true	false

There are two "pitfalls" that make it advisable to use == and != sparingly with truth values:

- If you use == or != to compare two Boolean objects, then the Reference operators are used. This may give an unexpected result; see Pitfall: using == to compare primitive wrappers objects such as Integer
- The == operator can easily be mistyped as =. For most operand types, this mistake leads to a compilation error. However, for boolean and Boolean operands the mistake leads to incorrect runtime behavior; see Pitfall - Using '==' to test a boolean

The Reference == and != operators

If both operands are object references, the == and != operators test if the two operands refer to the same object. This often not what you want. To test if two objects are equal by value, the .equals() method should be used instead.

```

String s1 = "我们相等";
String s2 = new String("我们相等");

s1.equals(s2); // true

// 警告 - 不要使用 == 或 != 来比较 String 值
s1 == s2;      // false

```

警告：在大多数情况下，使用 == 和 != 来比较 String 值是不正确的；详见<http://stackoverflow.com/documentation/java/4388/java-pitfalls/16290/using-to-compare-strings>。类似的问题也适用于基本类型包装类；详见<http://stackoverflow.com/documentation/java/4388/java-pitfalls/8996/using-to-compare-primitive-wrappers-objects-such-as-integer>。

关于 NaN 边界情况

[JLS 15.21.1](#) 规定如下：

如果任一操作数是 NaN，则 == 的结果为 **false**，但 != 的结果为 **true**。实际上，测试 `x != x` 当且仅当 `x` 的值是 NaN 时，结果为 **true**。

这种行为对大多数程序员来说是意料之外的。如果你测试一个 NaN 值是否等于它自身，答案是“不等于！”。换句话说，== 对于 NaN 值来说不是自反的。

然而，这并不是 Java 的“怪异”行为，这种行为是 IEEE 754 浮点标准中规定的，你会发现大多数现代编程语言都实现了这一点。（更多信息见<http://stackoverflow.com/a/1573715/139985>... 注意这篇文章的作者是“参与决策过程的人”！）

第40.12节：Lambda运算符 (->)

从 Java 8 开始，Lambda 操作符 (->) 是用于引入 Lambda 表达式的操作符。常见的两种语法如下示例所示：

```

版本 ≥ Java SE 8
a -> a + 1          // 一个对参数加一的 Lambda 表达式
a -> { return a + 1; } // 使用代码块的等价 Lambda 表达式。

```

Lambda 表达式定义了一个匿名函数，或者更准确地说，是实现了函数式接口的匿名类的一个实例。

（此示例为完整性而包含。完整内容请参见 Lambda 表达式主题。）

```

String s1 = "We are equal";
String s2 = new String("We are equal");

s1.equals(s2); // true

// WARNING - don't use == or != with String values
s1 == s2;      // false

```

Warning: using == and != to compare `String` values is **incorrect** in most cases; see <http://stackoverflow.com/documentation/java/4388/java-pitfalls/16290/using-to-compare-strings>. A similar problem applies to primitive wrapper types; see <http://stackoverflow.com/documentation/java/4388/java-pitfalls/8996/using-to-compare-primitive-wrappers-objects-such-as-integer>.

About the NaN edge-cases

[JLS 15.21.1](#) states the following:

If either operand is NaN, then the result of == is **false** but the result of != is **true**. Indeed, the test `x != x` is **true** if and only if the value of `x` is NaN.

This behavior is (to most programmers) unexpected. If you test if a NaN value is equal to itself, the answer is "No it isn't!". In other words, == is not *reflexive* for NaN values.

However, this is not a Java "oddity", this behavior is specified in the IEEE 754 floating-point standards, and you will find that it is implemented by most modern programming languages. (For more information, see <http://stackoverflow.com/a/1573715/139985>... noting that this is written by someone who was "in the room when the decisions were made"!)

Section 40.12: The Lambda operator (->)

From Java 8 onwards, the Lambda operator (->) is the operator used to introduce a Lambda Expression. There are two common syntaxes, as illustrated by these examples:

```

Version ≥ Java SE 8
a -> a + 1          // a lambda that adds one to its argument
a -> { return a + 1; } // an equivalent lambda using a block.

```

A lambda expression defines an anonymous function, or more correctly an instance of an anonymous class that implements a *functional interface*.

(This example is included here for completeness. Refer to the Lambda Expressions topic for the full treatment.)

第 41 章：构造函数

虽然不是必须的，但 Java 中的构造函数是编译器识别的方法，用于为类实例化特定的值，这些值可能对对象的作用至关重要。本主题演示了 Java 类构造函数的正确用法。

第 41.1 节：默认构造函数

构造函数的“默认”是指它们没有任何参数。如果你没有指定任何构造函数，编译器会为你生成一个默认构造函数。

这意味着以下两个代码片段在语义上是等价的：

```
public class TestClass {  
    private String test;  
}  
  
public class TestClass {  
    private String test;  
    public TestClass() {  
    }  
}
```

默认构造函数的可见性与类的可见性相同。因此，包私有定义的类具有包私有的默认构造函数

但是，如果你有非默认构造函数，编译器将不会为你生成默认构造函数。所以这些是不等价的：

```
public class TestClass {  
    private String test;  
    public TestClass(String arg) {  
    }  
  
public class TestClass {  
    private String test;  
    public TestClass() {  
    }  
    public TestClass(String arg) {  
    }  
}
```

注意，生成的构造函数不会执行任何非标准初始化。这意味着除非字段有初始化器，否则类的所有字段都将具有默认值。

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass() {  
        testData = "Test"  
    }  
}
```

构造函数的调用方式如下：

Chapter 41: Constructors

While not required, constructors in Java are methods recognized by the compiler to instantiate specific values for the class which may be essential to the role of the object. This topic demonstrates proper usage of Java class constructors.

Section 41.1: Default Constructor

The "default" for constructors is that they do not have any arguments. In case you do not specify **any** constructor, the compiler will generate a default constructor for you.

This means the following two snippets are semantically equivalent:

```
public class TestClass {  
    private String test;  
}  
  
public class TestClass {  
    private String test;  
    public TestClass() {  
    }  
}
```

The visibility of the default constructor is the same as the visibility of the class. Thus a class defined package-privately has a package-private default constructor

However, if you have non-default constructor, the compiler will not generate a default constructor for you. So these are not equivalent:

```
public class TestClass {  
    private String test;  
    public TestClass(String arg) {  
    }  
  
public class TestClass {  
    private String test;  
    public TestClass() {  
    }  
    public TestClass(String arg) {  
    }  
}
```

Beware that the generated constructor performs no non-standard initialization. This means all fields of your class will have their default value, unless they have an initializer.

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass() {  
        testData = "Test"  
    }  
}
```

Constructors are called like this:

```
TestClass testClass = new TestClass();
```

第41.2节：调用父类构造函数

假设你有一个父类和一个子类。构造子类实例时，总是需要在子类构造函数的最开始运行某个父类构造函数。
我们可以通过显式调用 `super(...)` 并传入适当的参数，作为子类构造函数的第一条语句，来选择想要的父类构造函数。这样做可以节省时间，复用父类的构造函数，而不必在子类构造函数中重复编写相同的代码。

无超级(...) 方法：

(隐式地，调用了无参数版本的 `super()`)

```
class Parent {  
    private String name;  
    private int age;  
  
    public Parent() {} // 必须，因为我们调用了无参数的 super()  
  
    public Parent(String fName, int fAge) {  
        name = fName;  
        age = fAge;  
    }  
}
```

// 这甚至无法编译，因为 `name` 和 `age` 是私有的，
// 使它们对子类也不可见。

```
class Child extends Parent {  
    public Child() {  
        // 编译器在这里隐式调用 super()  
        name = "John";  
        age = 42;  
    }  
}
```

使用 `super()` 方法：

```
类 Parent {  
    private String name;  
    私有 int age;  
    公共 Parent(String fName, int fAge) {  
        name = fName;  
        age = fAge;  
    }  
  
    class Child extends Parent {  
        public Child() {  
            super("John", 42); // 显式调用super构造函数  
        }  
    }  
}
```

注意：调用另一个构造函数（链式调用）或 `super` 构造函数 必须 是构造函数中的第一条语句。

如果你显式调用 `super(...)` 构造函数，必须存在匹配的父类构造函数（这很简单，不是吗？）。

```
TestClass testClass = new TestClass();
```

Section 41.2: Call parent constructor

Say you have a Parent class and a Child class. To construct a Child instance always requires some Parent constructor to be run at the very beginning of the Child constructor. We can select the Parent constructor we want by explicitly calling `super(...)` with the appropriate arguments as our first Child constructor statement. Doing this saves us time by reusing the Parent classes' constructor instead of rewriting the same code in the Child classes' constructor.

Without `super(...)` method:

(implicitly, the no-args version `super()` is called invisibly)

```
class Parent {  
    private String name;  
    private int age;  
  
    public Parent() {} // necessary because we call super() without arguments  
  
    public Parent(String fName, int fAge) {  
        name = fName;  
        age = fAge;  
    }  
}  
  
// This does not even compile, because name and age are private,  
// making them invisible even to the child class.  
class Child extends Parent {  
    public Child() {  
        // compiler implicitly calls super() here  
        name = "John";  
        age = 42;  
    }  
}
```

With `super()` method:

```
class Parent {  
    private String name;  
    private int age;  
    public Parent(String fName, int fAge) {  
        name = fName;  
        age = fAge;  
    }  
}  
  
class Child extends Parent {  
    public Child() {  
        super("John", 42); // explicit super-call  
    }  
}
```

Note: Calls to another constructor (chaining) or the super constructor **MUST** be the first statement inside the constructor.

If you call the `super(...)` constructor explicitly, a matching parent constructor must exist (that's straightforward, isn't it?).

如果你没有显式调用任何 `super(...)` 构造函数，父类必须有一个无参构造函数——这可以是显式编写的，也可以是编译器在父类没有提供任何构造函数时自动生成的默认构造函数。

```
class Parent{  
    public Parent(String tName, int tAge) {}  
}  
  
class Child extends Parent{  
    public Child(){}  
}
```

`Parent` 类没有默认构造函数，因此编译器无法在 `Child` 构造函数中添加 `super`。此代码无法编译。你必须修改构造函数以适应双方，或者像这样编写你自己的 `super` 调用：

```
class Child extends Parent{  
    public Child(){  
        super("",0);  
    }  
}
```

第41.3节：带参数的构造函数

构造函数可以带有任何类型的参数。

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass(String testData) {  
        this.testData = testData;  
    }  
}
```

调用方式如下：

```
TestClass testClass = new TestClass("Test Data");
```

一个类可以有多个具有不同签名的构造函数。要链式调用构造函数（在实例化时调用同一类的不同构造函数），使用`this()`。

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass(String testData) {  
        this.testData = testData;  
    }  
  
    public TestClass() {  
        this("Test"); // testData 默认为 "Test"  
    }  
}
```

调用方式如下：

```
TestClass testClass1 = new TestClass("Test Data");
```

If you don't call any `super(...)` constructor explicitly, your parent class must have a no-args constructor - and this can be either written explicitly or created as a default by the compiler if the parent class doesn't provide any constructor.

```
class Parent{  
    public Parent(String tName, int tAge) {}  
}  
  
class Child extends Parent{  
    public Child(){}  
}
```

The class `Parent` has no default constructor, so, the compiler can't add `super` in the `Child` constructor. This code will not compile. You must change the constructors to fit both sides, or write your own `super` call, like that:

```
class Child extends Parent{  
    public Child(){  
        super("",0);  
    }  
}
```

Section 41.3: Constructor with Arguments

Constructors can be created with any kinds of arguments.

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass(String testData) {  
        this.testData = testData;  
    }  
}
```

Called like this:

```
TestClass testClass = new TestClass("Test Data");
```

A class can have multiple constructors with different signatures. To chain constructor calls (call a different constructor of the same class when instantiating) use `this()`.

```
public class TestClass {  
  
    private String testData;  
  
    public TestClass(String testData) {  
        this.testData = testData;  
    }  
  
    public TestClass() {  
        this("Test"); // testData defaults to "Test"  
    }  
}
```

Called like this:

```
TestClass testClass1 = new TestClass("Test Data");
```

```
TestClass testClass2 = new TestClass();
```

```
TestClass testClass2 = new TestClass();
```

第42章：Object类的方法和构造函数

本文档页面用于展示关于Java类构造函数的详细示例，以及关于Object类方法的内容，这些方法是从任何新创建类的超类Object自动继承的。

第42.1节：hashCode()方法

当Java类重写了equals方法时，也应重写hashCode方法。正如方法契约中定义的：

- 每当在 Java 应用程序的执行过程中对同一对象多次调用时，hashCode 方法必须始终返回相同的整数，前提是用于对象 equals 比较的信息未被修改。该整数不需要在同一应用程序的不同执行之间保持一致。
- 如果两个对象根据equals(Object)方法相等，那么对这两个对象分别调用hashCode方法必须产生相同的整数结果。
- 如果两个对象根据equals(Object)方法不相等，则不要求对这两个对象分别调用hashCode方法必须产生不同的整数结果。但是，程序员应当注意，为不相等的对象产生不同的整数结果可能会提高哈希表的性能。

哈希码用于诸如HashMap、HashTable和HashSet等哈希实现中。hashCode函数的结果决定了对象将被放入的桶。这些哈希实现如果提供的hashCode实现良好，则效率更高。良好hashCode实现的一个重要特性是hashCode值的分布均匀。换句话说，多个实例存储在同一个桶中的概率较小。

计算哈希码值的算法可能类似于以下内容：

```
public class Foo {  
    private int field1, field2;  
    private String field3;  
  
    public Foo(int field1, int field2, String field3) {  
        this.field1 = field1;  
        this.field2 = field2;  
        this.field3 = field3;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj) {  
            return true;  
        }  
        if (obj == null || getClass() != obj.getClass()) {  
            return false;  
        }  
  
        Foo f = (Foo) obj;  
        return field1 == f.field1 &&  
               field2 == f.field2 &&  
               (field3 == null ? f.field3 == null : field3.equals(f.field3));  
    }  
}
```

Chapter 42: Object Class Methods and Constructor

This documentation page is for showing details with example about java class [constructors](#) and about [Object Class Methods](#) which are automatically inherited from the superclass [Object](#) of any newly created class.

Section 42.1: hashCode() method

When a Java class overrides the equals method, it should override the hashCode method as well. As defined [in the method's contract](#):

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Hash codes are used in hash implementations such as [HashMap](#), [HashTable](#), and [HashSet](#). The result of the hashCode function determines the bucket in which an object will be put. These hash implementations are more efficient if the provided hashCode implementation is good. An important property of good hashCode implementation is that the distribution of the hashCode values is uniform. In other words, there is a small probability that numerous instances will be stored in the same bucket.

An algorithm for computing a hash code value may be similar to the following:

```
public class Foo {  
    private int field1, field2;  
    private String field3;  
  
    public Foo(int field1, int field2, String field3) {  
        this.field1 = field1;  
        this.field2 = field2;  
        this.field3 = field3;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj) {  
            return true;  
        }  
        if (obj == null || getClass() != obj.getClass()) {  
            return false;  
        }  
  
        Foo f = (Foo) obj;  
        return field1 == f.field1 &&  
               field2 == f.field2 &&  
               (field3 == null ? f.field3 == null : field3.equals(f.field3));  
    }  
}
```

```

@Override
public int hashCode() {
    int hash = 1;
    hash = 31 * hash + field1;
    hash = 31 * hash + field2;
    hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
    return hash;
}

```

使用 Arrays.hashCode() 作为快捷方式

版本 ≥ Java SE 1.2

在 Java 1.2 及以上版本中，不必开发算法来计算哈希码，可以使用通过提供包含字段值的对象或原始类型数组，使用 java.util.Arrays#hashCode：

```

@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}

```

版本 ≥ Java SE 7

Java 1.7 引入了 java.util.Objects 类，该类提供了一个便捷方法 hash(Object... objects)，该方法基于传入对象的值计算哈希码。此方法的工作方式与 java.util.Arrays#hashCode 相同。

```

@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}

```

注意：这种方法效率较低，每次调用自定义的 hashCode() 方法时都会产生垃圾对象：

- 会创建一个临时的 Object[]。 (在 Objects.hash() 版本中，数组是通过“可变参数”机制创建的。)
- 如果任何字段是原始类型，则必须进行装箱，这可能会创建更多的临时对象。
- 数组必须被填充。
- 数组必须通过Arrays.hashCode或Objects.hash方法进行迭代。
- 对Object.hashCode()的调用，Arrays.hashCode或Objects.hash必须执行的调用（可能）不能被内联。

哈希码的内部缓存

由于计算对象的哈希码可能代价较高，因此首次计算时将哈希码值缓存在对象内部可能是有吸引力的。例如

```

public final class ImmutableList {
    private int[] array;
    private volatile int hash = 0;

    public ImmutableList(int[] initial) {
        array = initial.clone();
    }

    // 其他方法
}

```

```

@Override
public int hashCode() {
    int hash = 1;
    hash = 31 * hash + field1;
    hash = 31 * hash + field2;
    hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
    return hash;
}

```

Using Arrays.hashCode() as a short cut

Version ≥ Java SE 1.2

In Java 1.2 and above, instead of developing an algorithm to compute a hash code, one can be generated using java.util.Arrays#hashCode by supplying an Object or primitives array containing the field values:

```

@Override
public int hashCode() {
    return Arrays.hashCode(new Object[] {field1, field2, field3});
}

```

Version ≥ Java SE 7

Java 1.7 introduced the java.util.Objects class which provides a convenience method, hash(Object... objects), that computes a hash code based on the values of the objects supplied to it. This method works just like java.util.Arrays#hashCode.

```

@Override
public int hashCode() {
    return Objects.hash(field1, field2, field3);
}

```

Note: this approach is inefficient, and produces garbage objects each time your custom hashCode() method is called:

- A temporary Object[] is created. (In the Objects.hash() version, the array is created by the "varargs" mechanism.)
- If any of the fields are primitive types, they must be boxed and that may create more temporary objects.
- The array must be populated.
- The array must be iterated by the Arrays.hashCode or Objects.hash method.
- The calls to Object.hashCode() that Arrays.hashCode or Objects.hash has to make (probably) cannot be inlined.

Internal caching of hash codes

Since the calculation of an object's hash code can be expensive, it can be attractive to cache the hash code value within the object the first time that it is calculated. For example

```

public final class ImmutableList {
    private int[] array;
    private volatile int hash = 0;

    public ImmutableList(int[] initial) {
        array = initial.clone();
    }

    // Other methods
}

```

```

@Override
public boolean equals(Object obj) {
    // ...
}

@Override
public int hashCode() {
    int h = hash;
    if (h == 0) {
        h = Arrays.hashCode(array);
        hash = h;
    }
    return h;
}

```

这种方法在（重复）计算哈希码的成本与用于缓存哈希码的额外字段开销之间进行了权衡。是否能作为性能优化取决于给定对象被哈希（查找）的频率以及其他因素。

你还会注意到，如果一个`ImmutableArray`的真实哈希码恰好为零（概率为 $1/2^{32}$ ），则缓存将无效。

最后，如果我们哈希的对象是可变的，实现这种方法会更加困难。然而，如果哈希码发生变化，则存在更大的问题；请参见上述约定。

第42.2节：`toString()`方法

`toString()`方法用于通过对象的内容创建对象的`String`表示。编写类时应重写此方法。当对象与字符串连接时，如`"hello" + anObject`，`toString()`会被隐式调用。

请考虑以下内容：

```

public class User {
    private String firstName;
    private String lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }

    public static void main(String[] args) {
        User user = new User("John", "Doe");
        System.out.println(user.toString()); // 打印 "John Doe"
    }
}

```

这里 `toString()` 方法来自 `Object` 类，在 `User` 类中被重写，以便在打印对象时提供有意义的数据。

使用 `println()` 时，会隐式调用对象的 `toString()` 方法。因此，这些语句执行的是

```

@Override
public boolean equals(Object obj) {
    // ...
}

@Override
public int hashCode() {
    int h = hash;
    if (h == 0) {
        h = Arrays.hashCode(array);
        hash = h;
    }
    return h;
}

```

This approach trades off the cost of (repeatedly) calculating the hash code against the overhead of an extra field to cache the hash code. Whether this pays off as a performance optimization will depend on how often a given object is hashed (looked up) and other factors.

You will also notice that if the true hashCode of an `ImmutableArray` happens to be zero (one chance in 2³²), the cache is ineffective.

Finally, this approach is much harder to implement correctly if the object we are hashing is mutable. However, there are bigger concerns if hash codes change; see the contract above.

Section 42.2: `toString()` method

The `toString()` method is used to create a `String` representation of an object by using the object's content. This method should be overridden when writing your class. `toString()` is called implicitly when an object is concatenated to a string as in `"hello" + anObject`.

Consider the following:

```

public class User {
    private String firstName;
    private String lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }

    public static void main(String[] args) {
        User user = new User("John", "Doe");
        System.out.println(user.toString()); // Prints "John Doe"
    }
}

```

Here `toString()` from `Object` class is overridden in the `User` class to provide meaningful data regarding the object when printing it.

When using `println()`, the object's `toString()` method is implicitly called. Therefore, these statements do the

相同的操作：

```
System.out.println(user); // 隐式调用 `user` 的 toString()
System.out.println(user.toString());
```

如果上述 User 类中没有重写 `toString()` 方法，`System.out.println(user)` 可能会返回 `User@659e0bfd` 或类似的字符串，几乎没有有用信息，除了类名。这是因为调用使用了 Java 基础 `Object` 类的 `toString()` 实现，该实现不了解 User 类的结构或业务规则。如果你想改变类中的此功能，只需重写该方法即可。

第42.3节：equals() 方法

简要说明

`==` 用于测试引用是否相等（是否是同一个对象）`.equals()` 用于测试

值是否相等（是否在逻辑上“相等”）`equals()` 是一个用于比较两个对象是

否相等的方法。Object类中`equals()`方法的默认实现仅当两个引用指向同一个实例时返回true。因此，它的行为与`=`比较相同。

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // 输出 false
    }
}
```

尽管 `foo1` 和 `foo2` 是用相同的字段创建的，但它们指向内存中的两个不同对象。因此，默认的 `equals()` 实现会返回 `false`。

要比较对象内容是否相等，必须重写 `equals()`。

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    @Override
}
```

same thing:

```
System.out.println(user); // toString() is implicitly called on `user`
System.out.println(user.toString());
```

If the `toString()` is not overridden in the above mentioned User class, `System.out.println(user)` may return `User@659e0bfd` or a similar String with almost no useful information except the class name. This will be because the call will use the `toString()` implementation of the base Java `Object` class which does not know anything about the User class's structure or business rules. If you want to change this functionality in your class, simply override the method.

Section 42.3: equals() method

TL;DR

`==` tests for reference equality (whether they are the **same object**)

`.equals()` tests for value equality (whether they are **logically "equal"**)

`equals()` is a method used to compare two objects for equality. The default implementation of the `equals()` method in the `Object` class returns `true` if and only if both references are pointing to the same instance. It therefore behaves the same as comparison by `==`.

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    public static void main(String[] args) {
        Foo foo1 = new Foo(0, 0, "bar");
        Foo foo2 = new Foo(0, 0, "bar");

        System.out.println(foo1.equals(foo2)); // prints false
    }
}
```

Even though `foo1` and `foo2` are created with the same fields, they are pointing to two different objects in memory. The default `equals()` implementation therefore evaluates to `false`.

To compare the contents of an object for equality, `equals()` has to be overridden.

```
public class Foo {
    int field1, field2;
    String field3;

    public Foo(int i, int j, String k) {
        field1 = i;
        field2 = j;
        field3 = k;
    }

    @Override
}
```

```

public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    Foo f = (Foo) obj;
    return field1 == f.field1 &&
    field2 == f.field2 &&
        (field3 == null ? f.field3 == null : field3.equals(f.field3));
}

@Override
public int hashCode() {
    int hash = 1;
    hash = 31 * hash + this.field1;
    hash = 31 * hash + this.field2;
    hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
    return hash;
}

public static void main(String[] args) {
    Foo foo1 = new Foo(0, 0, "bar");
    Foo foo2 = new Foo(0, 0, "bar");

    System.out.println(foo1.equals(foo2)); // 输出 true
}

```

这里重写的 `equals()` 方法决定如果对象的字段相同，则对象相等。

注意 `hashCode()` 方法也被重写了。该方法的契约规定，当两个对象相等时，它们的哈希值也必须相同。这就是为什么几乎总是需要同时重写`hashCode()` 和 `equals()` 方法的原因。

特别注意 `equals` 方法的参数类型。它是 `Object obj`，而不是 `Foo obj`。如果你在方法中使用后者，那就不是对 `equals` 方法的重写。

在编写自己的类时，重写 `equals()` 和 `hashCode()` 方法时需要编写类似的逻辑。大多数集成开发环境（IDE）可以自动为你生成这些代码。

`equals()` 方法的一个示例实现可以在 `String` 类中找到，该类是 Java 核心 API 的一部分。与其比较指针，`String` 类比较的是 `String` 的内容。

版本 ≥ Java SE 7

Java 1.7 引入了 `java.util.Objects` 类，该类提供了一个方便的方法 `equals`，用于比较两个可能为 `null` 的引用，因此可以用来简化 `equals` 方法的实现。

```

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

```

```

public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

    Foo f = (Foo) obj;
    return field1 == f.field1 &&
        field2 == f.field2 &&
            (field3 == null ? f.field3 == null : field3.equals(f.field3));
}

@Override
public int hashCode() {
    int hash = 1;
    hash = 31 * hash + this.field1;
    hash = 31 * hash + this.field2;
    hash = 31 * hash + (field3 == null ? 0 : field3.hashCode());
    return hash;
}

public static void main(String[] args) {
    Foo foo1 = new Foo(0, 0, "bar");
    Foo foo2 = new Foo(0, 0, "bar");

    System.out.println(foo1.equals(foo2)); // prints true
}

```

Here the overridden `equals()` method decides that the objects are equal if their fields are the same.

Notice that the `hashCode()` method was also overwritten. The contract for that method states that when two objects are equal, their hash values must also be the same. That's why one must almost always override `hashCode()` and `equals()` together.

Pay special attention to the argument type of the `equals` method. It is `Object obj`, not `Foo obj`. If you put the latter in your method, that is not an override of the `equals` method.

When writing your own class, you will have to write similar logic when overriding `equals()` and `hashCode()`. Most IDEs can automatically generate this for you.

An example of an `equals()` implementation can be found in the `String` class, which is part of the core Java API. Rather than comparing pointers, the `String` class compares the content of the `String`.

Version ≥ Java SE 7

Java 1.7 introduced the `java.util.Objects` class which provides a convenience method, `equals`, that compares two potentially `null` references, so it can be used to simplify implementations of the `equals` method.

```

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false;
    }

```

```

Foo f = (Foo) obj;
    return field1 == f.field1 && field2 == f.field2 && Objects.equals(field3, f.field3);
}

```

类比较

由于 `equals` 方法可以针对任何对象运行，该方法通常首先做的事情之一（在检查是否为 `null` 之后）是检查被比较对象的类是否与当前类匹配。

```

@Override
public boolean equals(Object obj) {
    //...检查是否为 null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...比较字段
}

```

这通常如上所示通过比较类对象来完成。然而，在一些特殊情况下，这种做法可能会失败，这些情况可能并不明显。例如，一些框架会生成类的动态代理，而这些动态代理实际上是不同的类。以下是使用 JPA 的一个示例。

```

Foo detachedInstance = ...
Foo mergedInstance = entityManager.merge(detachedInstance);
if (mergedInstance.equals(detachedInstance)) {
    //如果用 getClass() 测试相等性，永远不会到这里
    //因为 mergedInstance 是 Foo 的代理（子类）
}

```

解决该限制的一种机制是使用 `instanceof` 比较类

```

@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...比较字段
}

```

但是，使用 `instanceof` 时必须避免一些陷阱。由于 `Foo` 可能有其他子类，并且这些子类可能重写了 `equals()`，你可能会遇到 `Foo` 等于 `FooSubclass` 但 `FooSubclass` 不等于 `Foo` 的情况。

```

Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false

```

这违反了对称性和传递性的属性，因此是对 `equals()` 方法的无效实现。因此，在使用 `instanceof` 时，良好的做法是将 `equals()` 方法设为 `final`（如上例所示）。这将确保没有子类重写 `equals()` 并破坏关键假设。

第42.4节：`wait()` 和 `notify()` 方法

`wait()` 和 `notify()` 协同工作—当一个线程在某个对象上调用 `wait()` 时，该线程将阻塞，直到另一个线程在同一对象上调用 `notify()` 或 `notifyAll()`。

```

Foo f = (Foo) obj;
    return field1 == f.field1 && field2 == f.field2 && Objects.equals(field3, f.field3);
}

```

Class Comparison

Since the `equals` method can run against any object, one of the first things the method often does (after checking for `null`) is to check if the class of the object being compared matches the current class.

```

@Override
public boolean equals(Object obj) {
    //...check for null
    if (getClass() != obj.getClass()) {
        return false;
    }
    //...compare fields
}

```

This is typically done as above by comparing the class objects. However, that can fail in a few special cases which may not be obvious. For example, some frameworks generate dynamic proxies of classes and these dynamic proxies are actually a different class. Here is an example using JPA.

```

Foo detachedInstance = ...
Foo mergedInstance = entityManager.merge(detachedInstance);
if (mergedInstance.equals(detachedInstance)) {
    //Can never get here if equality is tested with getClass()
    //as mergedInstance is a proxy (subclass) of Foo
}

```

One mechanism to work around that limitation is to compare classes using `instanceof`

```

@Override
public final boolean equals(Object obj) {
    if (!(obj instanceof Foo)) {
        return false;
    }
    //...compare fields
}

```

However, there are a few pitfalls that must be avoided when using `instanceof`. Since `Foo` could potentially have other subclasses and those subclasses might override `equals()` you could get into a case where a `Foo` is equal to a `FooSubclass` but the `FooSubclass` is not equal to `Foo`.

```

Foo foo = new Foo(7);
FooSubclass fooSubclass = new FooSubclass(7, false);
foo.equals(fooSubclass) //true
fooSubclass.equals(foo) //false

```

This violates the properties of symmetry and transitivity and thus is an invalid implementation of the `equals()` method. As a result, when using `instanceof`, a good practice is to make the `equals()` method `final` (as in the above example). This will ensure that no subclass overrides `equals()` and violates key assumptions.

Section 42.4: `wait()` and `notify()` methods

`wait()` and `notify()` work in tandem – when one thread calls `wait()` on an object, that thread will block until another thread calls `notify()` or `notifyAll()` on that same object.

(另见：wait()/notify())

```
包 com.example.examples.object;

导入 java.util.concurrent.atomic.AtomicBoolean;

公共类 WaitAndNotify {

    公共静态 void main(String[] args) 抛出 InterruptedException {
        最终 Object obj = 新建 Object();
        AtomicBoolean aHasFinishedWaiting = 新建 AtomicBoolean(false);

        线程 threadA = 新建 线程("线程 A") {
            公共 void 运行() {
                系统.输出.打印行("A1: 可能在 B1 之前或之后打印");
                系统.输出.打印行("A2: 线程 A 即将开始等待...");
                尝试 {
                    同步 (obj) { // wait() 必须在同步块中
                        // 线程 A 的执行将停止，直到调用 obj.notify()
                        obj.wait();
                    }
                    System.out.println("A3: 线程A已完成等待。"
                        + "保证在B3之后发生");
                } catch (InterruptedException e) {
                    System.out.println("线程A在等待时被中断");
                } finally {
                    aHasFinishedWaiting.set(true);
                }
            }
        };

        Thread threadB = new Thread("线程B") {
            public void run() {
                System.out.println("B1: 可能在A1之前或之后打印");

                System.out.println("B2: 线程B即将等待10秒");
                for (int i = 0; i < 10; i++) {
                    try {
                        Thread.sleep(1000); // 休眠1秒
                    } catch (InterruptedException e) {
                        System.err.println("线程B在等待时被中断");
                    }
                }

                System.out.println("B3: 由于"
                    + "A3 只能在调用 obj.notify() 之后发生，因此 B3 总是会在 A3 之前打印。");

                while (!aHasFinishedWaiting.get()) {
                    synchronized (obj) {
                        // 通知一个调用了 obj.wait() 的线程
                        obj.notify();
                    }
                }
            }
        };
    }

    threadA.start();
    threadB.start();

    threadA.join();
    threadB.join();
}
```

(See Also: wait()/notify())

```
package com.example.examples.object;

import java.util.concurrent.atomic.AtomicBoolean;

public class WaitAndNotify {

    public static void main(String[] args) throws InterruptedException {
        final Object obj = new Object();
        AtomicBoolean aHasFinishedWaiting = new AtomicBoolean(false);

        Thread threadA = new Thread("Thread A") {
            public void run() {
                System.out.println("A1: Could print before or after B1");
                System.out.println("A2: Thread A is about to start waiting...");
                try {
                    synchronized (obj) { // wait() must be in a synchronized block
                        // execution of thread A stops until obj.notify() is called
                        obj.wait();
                    }
                    System.out.println("A3: Thread A has finished waiting."
                        + "Guaranteed to happen after B3");
                } catch (InterruptedException e) {
                    System.out.println("Thread A was interrupted while waiting");
                } finally {
                    aHasFinishedWaiting.set(true);
                }
            }
        };

        Thread threadB = new Thread("Thread B") {
            public void run() {
                System.out.println("B1: Could print before or after A1");

                System.out.println("B2: Thread B is about to wait for 10 seconds");
                for (int i = 0; i < 10; i++) {
                    try {
                        Thread.sleep(1000); // sleep for 1 second
                    } catch (InterruptedException e) {
                        System.err.println("Thread B was interrupted from waiting");
                    }
                }

                System.out.println("B3: Will ALWAYS print before A3 since "
                    + "A3 can only happen after obj.notify() is called.");

                while (!aHasFinishedWaiting.get()) {
                    synchronized (obj) {
                        // notify ONE thread which has called obj.wait()
                        obj.notify();
                    }
                }
            }
        };
    }

    threadA.start();
    threadB.start();

    threadA.join();
    threadB.join();
}
```

```

        System.out.println("完成！");
    }
}

```

一些示例输出：

```

A1：可能在 B1 之前或之后打印
B1：可能在 A1 之前或之后打印
A2：线程 A 即将开始等待...
B2：线程 B 即将等待 10 秒
B3：由于 A3 只能在调用 obj.notify() 之后发生，因此 B3 总是会在 A3 之前打印。
A3：线程A已完成等待。保证发生在B3之后
完成！

```

```

B1：可能在A1之前或之后打印
B2：线程B即将等待10秒
A1：可能在B1之前或之后打印
A2：线程A即将开始等待...
B3：总是会在A3之前打印，因为A3只能在调用obj.notify()之后发生。
A3：线程A已完成等待。保证发生在B3之后
完成！

```

```

A1：可能在B1之前或之后打印
A2：线程A即将开始等待...
B1：可能在A1之前或之后打印
B2：线程 B 即将等待 10 秒
B3：由于 A3 只能在调用 obj.notify() 之后发生，因此 B3 总是会在 A3 之前打印。
A3：线程A已完成等待。保证发生在B3之后
完成！

```

```

        System.out.println("Finished!");
    }
}

```

Some example output:

```

A1: Could print before or after B1
B1: Could print before or after A1
A2: Thread A is about to start waiting...
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

```

```

B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
A1: Could print before or after B1
A2: Thread A is about to start waiting...
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

```

```

A1: Could print before or after B1
A2: Thread A is about to start waiting...
B1: Could print before or after A1
B2: Thread B is about to wait for 10 seconds
B3: Will ALWAYS print before A3 since A3 can only happen after obj.notify() is called.
A3: Thread A has finished waiting. Guaranteed to happen after B3
Finished!

```

第42.5节：getClass()方法

getClass()方法可用于查找对象的运行时类类型。见下面的示例：

```

public class User {

    private long userID;
    private String name;

    public User(long userID, String name) {
        this.userID = userID;
        this.name = name;
    }

    public class SpecificUser extends User {
        private String specificUserID;

        public SpecificUser(String specificUserID, long userID, String name) {
            super(userID, name);
            this.specificUserID = specificUserID;
        }

        public static void main(String[] args){
            User user = new User(879745, "John");
            SpecificUser specificUser = new SpecificUser("1AAAA", 877777, "Jim");
            User anotherSpecificUser = new SpecificUser("1BBBB", 812345, "Jenny");

            System.out.println(user.getClass()); //打印 "class User"
        }
    }
}

```

Section 42.5: getClass() method

The getClass() method can be used to find the runtime class type of an object. See the example below:

```

public class User {

    private long userID;
    private String name;

    public User(long userID, String name) {
        this.userID = userID;
        this.name = name;
    }

    public class SpecificUser extends User {
        private String specificUserID;

        public SpecificUser(String specificUserID, long userID, String name) {
            super(userID, name);
            this.specificUserID = specificUserID;
        }

        public static void main(String[] args){
            User user = new User(879745, "John");
            SpecificUser specificUser = new SpecificUser("1AAAA", 877777, "Jim");
            User anotherSpecificUser = new SpecificUser("1BBBB", 812345, "Jenny");

            System.out.println(user.getClass()); //Prints "class User"
        }
    }
}

```

```

System.out.println(specificUser.getClass()); //打印 "class SpecificUser"
System.out.println(anotherSpecificUser.getClass()); //打印 "class SpecificUser"
}

```

getClass() 方法将返回最具体的类类型，这就是为什么当在anotherSpecificUser上调用 getClass() 时，返回值是 class SpecificUser，因为它在继承树中比 User更低。

值得注意的是，虽然 getClass 方法声明为：

```
public final native Class<?> getClass();
```

调用 getClass 实际返回的静态类型是 Class<? extends T>，其中 T 是调用 getClass 的对象的静态类型。

即，以下代码将能编译：

```
Class<? extends String> cls = "".getClass();
```

第42.6节：clone()方法

clone()方法用于创建并返回对象的副本。这个方法有争议，应该尽量避免使用，因为它存在问题，应该使用复制构造函数或其他复制方法来代替clone()。

要使用该方法，所有调用该方法的类必须实现Cloneable接口。

Cloneable接口本身只是一个标记接口，用于改变本地clone()方法的行为，该方法会检查调用对象的类是否实现了Cloneable接口。如果调用者未实现该接口，将抛出CloneNotSupportedException异常。

Object类本身并未实现该接口，因此如果调用对象是Object类的实例，将抛出CloneNotSupportedException异常。

为了使克隆正确，克隆对象应独立于被克隆对象，因此可能需要在返回之前修改该对象。这意味着本质上要创建一个“深拷贝”，即同时复制构成被克隆对象内部结构的所有可变对象。如果未正确实现，克隆对象将不独立，并且与被克隆对象共享对可变对象的引用。这会导致不一致的行为，因为对其中一个的修改会影响另一个。

```

class Foo implements Cloneable {
    int w;
    String x;
    float[] y;
    Date z;

    public Foo clone() {
        try {
            Foo 结果 = new Foo();
            // 通过值复制基本类型
            结果.w = this.w;
            // 像 String 这样的不可变对象可以通过引用复制
            结果.x = this.x;

            // 字段 y 和 z 引用的是可变对象；递归克隆它们。
            if (this.y != null) {

```

```

System.out.println(specificUser.getClass()); //Prints "class SpecificUser"
System.out.println(anotherSpecificUser.getClass()); //Prints "class SpecificUser"
}

```

The getClass() method will return the most specific class type, which is why when getClass() is called on anotherSpecificUser, the return value is class SpecificUser because that is lower down the inheritance tree than User.

It is noteworthy that, while the getClass method is declared as:

```
public final native Class<?> getClass();
```

The actual static type returned by a call to getClass is Class<? extends T> where T is the static type of the object on which getClass is called.

i.e. the following will compile:

```
Class<? extends String> cls = "".getClass();
```

Section 42.6: clone() method

The clone() method is used to create and return a copy of an object. This method arguably should be avoided as it is problematic and a copy constructor or some other approach for copying should be used in favour of clone().

For the method to be used all classes calling the method must implement the [Cloneable](#) interface.

The [Cloneable](#) interface itself is just a tag interface used to change the behaviour of the [native](#) clone() method which checks if the calling objects class implements [Cloneable](#). If the caller does not implement this interface a [CloneNotSupportedException](#) will be thrown.

The [Object](#) class itself does not implement this interface so a [CloneNotSupportedException](#) will be thrown if the calling object is of class [Object](#).

For a clone to be correct it should be independent of the object it is being cloned from, therefore it may be necessary to modify the object before it gets returned. This means to essentially create a “deep copy” by also copying any of the *mutable* objects that make up the internal structure of the object being cloned. If this is not implemented correctly the cloned object will not be independent and have the same references to the mutable objects as the object that it was cloned from. This would result in inconsistent behaviour as any changes to those in one would affect the other.

```

class Foo implements Cloneable {
    int w;
    String x;
    float[] y;
    Date z;

    public Foo clone() {
        try {
            Foo result = new Foo();
            // copy primitives by value
            result.w = this.w;
            // immutable objects like String can be copied by reference
            result.x = this.x;

            // The fields y and z refer to a mutable objects; clone them recursively.
            if (this.y != null) {

```

```

结果.y = this.y.clone();
}
if (this.z != null) {
结果.z = this.z.clone();
}

// 完成, 返回新对象
return result;

} catch (CloneNotSupportedException e) {
// 如果任何被克隆的可变字段未实现 Cloneable
throw new AssertionError(e);
}
}
}

```

第42.7节：对象构造函数

Java中的所有构造函数必须调用Object构造函数。调用方式是 `super()`。该调用必须是构造函数中的第一行。这样做的原因是确保对象在执行任何额外初始化之前，能够实际在堆上被创建。

如果你在构造函数中没有指定调用 `super()`，编译器会自动为你添加。

因此，以下三个示例在功能上是完全相同的，都是显式调

用 `super()` 构造函数

```

public class MyClass {

    public MyClass() {
        super();
    }
}

```

隐式调用 `super()` 构造函数

```

public class MyClass {

    public MyClass() {
        // 空
    }
}

```

隐式构造函数

```

public class MyClass {

}

```

构造函数链怎么样？

可以在构造函数的第一条指令中调用其他构造函数。由于显式调用父类构造函数和调用另一个构造函数都必须是第一条指令，因此它们是互斥的。

```

public class MyClass {

```

```

    result.y = this.y.clone();
}
if (this.z != null) {
    result.z = this.z.clone();
}

// Done, return the new object
return result;

} catch (CloneNotSupportedException e) {
// in case any of the cloned mutable fields do not implement Cloneable
throw new AssertionError(e);
}
}
}

```

Section 42.7: Object constructor

All constructors in Java must make a call to the `Object` constructor. This is done with the call `super()`. This has to be the first line in a constructor. The reason for this is so that the object can actually be created on the heap before any additional initialization is performed.

If you do not specify the call to `super()` in a constructor the compiler will put it in for you.

So all three of these examples are functionally identical

with explicit call to `super()` constructor

```

public class MyClass {

    public MyClass() {
        super();
    }
}

```

with implicit call to `super()` constructor

```

public class MyClass {

    public MyClass() {
        // empty
    }
}

```

with implicit constructor

```

public class MyClass {

}

```

What about Constructor-Chaining?

It is possible to call other constructors as the first instruction of a constructor. As both the explicit call to a super constructor and the call to another constructor have to be both first instructions, they are mutually exclusive.

```

public class MyClass {

```

```

public MyClass(int size) {
    doSomethingWith(size);
}

public MyClass(Collection<?> initialValues) {
    this(initialValues.size());
    addInitialValues(initialValues);
}

```

调用 `new MyClass(Arrays.asList("a", "b", "c"))` 会调用带有 List 参数的第二个构造函数，该构造函数会转而委托给第一个构造函数（该构造函数会隐式委托给 `super()`），然后调用 `addInitialValues(int size)`，参数为列表的大小。这样做是为了减少代码重复，当多个构造函数需要执行相同的工作时使用。

如何调用特定的构造函数？

以上述示例为例，可以调用 `new MyClass("argument")` 或 `new MyClass("argument", 0)`。换句话说，类似于方法重载，您只需使用所需参数调用相应的构造函数。

Object 类的构造函数会发生什么？

不会发生比子类拥有默认空构造函数（不包括调用 `super()`）更多的事情。

默认的无参构造函数可以显式定义，但如果没有任何定义，只要没有其他构造函数，编译器会自动为你添加。

那么，如何通过构造函数创建一个对象呢？

对象的实际创建由JVM完成。Java中的每个构造函数都表现为一个名为 `<init>` 的特殊方法，负责实例初始化。该 `<init>` 方法由编译器提供，且由于 `<init>` 不是Java中的有效标识符，因此不能直接在语言中使用。

JVM如何调用这个 `<init>` 方法？

JVM会使用 `invokespecial` 指令调用 `<init>` 方法，并且只能在未初始化的类实例上调用。

更多信息请参阅JVM规范和Java语言规范：

- 特殊方法 (JVM) - [JVMS - 2.9](#)
- 构造函数 - [JLS - 8.8](#)

第42.8节：`finalize()`方法

这是 `protected` 且 `non-static` 的 `Object` 类方法。该方法用于执行一些最终操作

```

public MyClass(int size) {
    doSomethingWith(size);
}

public MyClass(Collection<?> initialValues) {
    this(initialValues.size());
    addInitialValues(initialValues);
}

```

`Calling new MyClass(Arrays.asList("a", "b", "c"))` will call the second constructor with the List-argument, which will in turn delegate to the first constructor (which will delegate implicitly to `super()`) and then call `addInitialValues(int size)` with the second size of the list. This is used to reduce code duplication where multiple constructors need to do the same work.

How do I call a specific constructor?

Given the example above, one can either call `new MyClass("argument")` or `new MyClass("argument", 0)`. In other words, much like method overloading, you just call the constructor with the parameters that are necessary for your chosen constructor.

What will happen in the Object class constructor?

Nothing more than would happen in a sub-class that has a default empty constructor (minus the call to `super()`).

The default empty constructor can be explicitly defined but if not the compiler will put it in for you as long as no other constructors are already defined.

How is an Object then created from the constructor in Object?

The actual creation of objects is down to the JVM. Every constructor in Java appears as a special method named `<init>` which is responsible for instance initializing. This `<init>` method is supplied by the compiler and because `<init>` is not a valid identifier in Java, it cannot be used directly in the language.

How does the JVM invoke this `<init>` method?

The JVM will invoke the `<init>` method using the `invokespecial` instruction and can only be invoked on uninitialized class instances.

For more information take a look at the JVM specification and the Java Language Specification:

- Special Methods (JVM) - [JVMS - 2.9](#)
- Constructors - [JLS - 8.8](#)

Section 42.8: `finalize()` method

This is a `protected` and `non-static` method of the `Object` class. This method is used to perform some final operations

或在对象从内存中移除之前进行清理操作。

根据文档，当垃圾回收器确定对象不再有任何引用时，会调用该方法。

但不能保证如果对象仍然可达，或者当对象变为可回收时没有垃圾回收器运行，`finalize()`方法一定会被调用。这就是为什么最好不要依赖该方法的原因。

在Java核心库中可以找到一些使用示例，例如在`FileInputStream.java`中：

```
受保护的 void finalize() throws IOException {
    如果 ((fd != null) && (fd != FileDescriptor.in)) {
        /* 如果fd是共享的, FileDescriptor中的引用
         * 将确保只有在安全时才调用终结器。
         * 使用该fd的所有引用都已变为不可达。
         * 我们可以调用close()
         */
        close();
    }
}
```

在这种情况下，如果资源之前未关闭，这是关闭资源的最后机会。

通常认为在任何类型的应用程序中使用`finalize()`方法是不好的做法，应当避免。

终结器不用于释放资源（例如关闭文件）。垃圾回收器在系统堆空间不足时（如果发生！）被调用。你不能指望它在系统文件句柄不足或任何其他原因时被调用。

终结器的预期用途是当一个对象即将被回收时，通知其他对象其即将消亡。现在已有更好的机制来实现这一目的——即`java.lang.ref.WeakReference<T>`类。如果你认为需要编写`finalize()`方法，那么你应该考虑是否可以使用`WeakReference`来解决同样的问题。如果这仍然无法解决你的问题，那么你可能需要从更深层次重新思考你的设计。

欲了解更多内容，可以参考Joshua Bloch的《Effective Java》一书中关于`finalize()`方法的章节。

or clean up operations on an object before it gets removed from the memory.

According to the doc, this method gets called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

But there are no guarantees that `finalize()` method would get called if the object is still reachable or no Garbage Collectors run when the object becomes eligible. That's why it's better **not rely** on this method.

In Java core libraries some usage examples could be found, for instance in [FileInputStream.java](#):

```
protected void finalize() throws IOException {
    if ((fd != null) && (fd != FileDescriptor.in)) {
        /* if fd is shared, the references in FileDescriptor
         * will ensure that finalizer is only called when
         * safe to do so. All references using the fd have
         * become unreachable. We can call close()
         */
        close();
    }
}
```

In this case it's the last chance to close the resource if that resource has not been closed before.

Generally it's considered bad practice to use `finalize()` method in applications of any kind and should be avoided.

Finalizers are *not* meant for freeing resources (e.g., closing files). The garbage collector gets called when (if!) the system runs low on heap space. You can't rely on it to be called when the system is running low on file handles or, for any other reason.

The intended use-case for finalizers is for an object that is about to be reclaimed to notify some other object about its impending doom. A better mechanism now exists for that purpose---the `java.lang.ref.WeakReference<T>` class. If you think you need write a `finalize()` method, then you should look into whether you can solve the same problem using `WeakReference` instead. If that won't solve your problem, then you may need to re-think your design on a deeper level.

For further reading [here](#) is an item about `finalize()` method from "Effective Java" book by Joshua Bloch.

第43章：注解

在Java中，annotation是一种可以添加到Java源代码中的语法元数据形式。它提供了关于程序的数据信息，但不是程序本身的一部分。注解对其所注释的代码的运行没有直接影响。类、方法、变量、参数和包都可以被注解。

第43.1节：注解背后的理念

Java语言规范对注解的描述如下：

注解是一种标记，它将信息与程序结构关联，但在运行时没有任何影响。

注解可以出现在类型或声明之前。它们也可能出现在既可应用于类型又可应用于声明的位置。

注解具体应用于何处由“元注解”@Target决定。更多信息请参见“[定义注解类型](#)”。

注解被用于多种目的。像Spring和Spring-MVC这样的框架利用注解来定义依赖注入的位置或请求路由的位置。

其他框架使用注解进行代码生成。Lombok 和 JPA 是典型的例子，它们使用注解来生成 Java (和 SQL) 代码。

本主题旨在提供以下内容的全面概述：

- 如何定义您自己的注解？
- Java语言提供了哪些注解？
- 注解在实践中是如何使用的？

第43.2节：定义注解类型

注解类型通过@interface定义。参数的定义类似于普通接口的方法。

```
@interface MyAnnotation {  
    String param1();  
    boolean param2();  
    int[] param3(); // 数组参数  
}
```

默认值

```
@interface MyAnnotation {  
    String param1() default "someValue";  
    boolean param2() default true;  
    int[] param3() default {};  
}
```

元注解

元注解是可以应用于注解类型的注解。特殊的预定义元注解定义了注解类型的使用方式。

Chapter 43: Annotations

In Java, an [annotation](#) is a form of syntactic metadata that can be added to Java source code. [It provides data](#) about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate. Classes, methods, variables, parameters and packages are allowed to be annotated.

Section 43.1: The idea behind Annotations

The [Java Language Specification](#) describes Annotations as follows:

An annotation is a marker which associates information with a program construct, but has no effect at run time.

Annotations may appear before types or declarations. It is possible for them to appear in a place where they could apply to both a type or a declaration.

What exactly an annotation applies to is governed by the "meta-annotation" @Target. See "[Defining annotation types](#)" for more information.

Annotations are used for a multitude of purposes. Frameworks like Spring and Spring-MVC make use of annotations to define where Dependencies should be injected or where requests should be routed.

Other frameworks use annotations for code-generation. Lombok and JPA are prime examples, that use annotations to generate Java (and SQL) code.

This topic aims to provide a comprehensive overview of:

- How to define your own Annotations?
- What Annotations does the Java Language provide?
- How are Annotations used in practice?

Section 43.2: Defining annotation types

Annotation types are defined with [@interface](#). Parameters are defined similar to methods of a regular interface.

```
@interface MyAnnotation {  
    String param1();  
    boolean param2();  
    int[] param3(); // array parameter  
}
```

Default values

```
@interface MyAnnotation {  
    String param1() default "someValue";  
    boolean param2() default true;  
    int[] param3() default {};  
}
```

Meta-Annotations

Meta-annotations are annotations that can be applied to annotation types. Special predefined meta-annotation define how annotation types can be used.

@Target

@Target元注解限制了注解可以应用的类型。

```
@Target(ElementType.METHOD)
@interface MyAnnotation {
    // 此注解只能应用于方法
}
```

可以使用数组表示法添加多个值，例如@Target({ElementType.FIELD, ElementType.TYPE})

可用值

元素类型	目标	在目标元素上的示例用法
注解类型 (ANNOTATION_TYPE)		@Retention(RetentionPolicy.RUNTIME) 接口 MyAnnotation
构造方法	构造方法	@MyAnnotationlic MyClass() {}
字段	字段, 枚举常量	@XmlAttributevalue int count;
局部变量	方法内部的变量声明	for (@LoopVariable int i = 0; i < 100; i++) { @Unused 字符串 resultVariable; }
包	包 (在 package-info.java)	@Deprecatedkage very.old;
方法	方法	@XmlElementlic int getCount() {...} public Rectangle(@NamedArg("width") double width, @NamedArg("height") double height) { ... }
参数	方法/构造函数参数	
类型	类, 接口, 枚举	@XmlRootElementlic class Report {}

版本 ≥ Java SE 8

元素类型 目标 在目标元素上的示例用法

类型参数声明	public <@MyAnnotation T> void f(T t) {}
类型使用	Object o = "42";ing s = (@MyAnnotation String) o;

@Retention

@Retention元注解定义了注解在应用程序编译过程或执行期间的可见性。默认情况下，注解包含在.class文件中，但在运行时不可见。要使注解在运行时可访问，必须在该注解上设置RetentionPolicy.RUNTIME。

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    // 该注解可以在运行时通过反射访问
}
```

可用值

RetentionPolicy (保留策略)	效果
CLASS (类)	该注解存在于.class文件中，但在运行时不可用
RUNTIME (运行时)	该注解在运行时可用，并且可以通过反射访问
SOURCE (源代码)	该注解在编译时可用，但不会添加到.class文件中。该注解可被例如注解处理器使用。

@Documented

@Documented 元注解用于标记那些其使用情况应由 API 文档生成器（如 javadoc）记录的注解。它没有值。使用 @Documented 后，所有使用该注解的类都会在其生成的文档页面中列出。没有 @Documented，则无法看到哪些类使用了该注解。

@Target

The @Target meta-annotation restricts the types the annotation can be applied to.

```
@Target(ElementType.METHOD)
@interface MyAnnotation {
    // this annotation can only be applied to methods
}
```

Multiple values can be added using array notation, e.g. @Target({ElementType.FIELD, ElementType.TYPE})

Available Values

ElementType	target	example usage on target element
ANNOTATION_TYPE	annotation types	@Retention(RetentionPolicy.RUNTIME) terface MyAnnotation
CONSTRUCTOR	constructors	@MyAnnotationlic MyClass() {}
FIELD	fields, enum constants	@XmlAttributevalue int count;
LOCAL_VARIABLE	variable declarations inside methods	for (@LoopVariable int i = 0; i < 100; i++) { @Unused String resultVariable;
PACKAGE	package (in package-info.java)	@Deprecatedkage very.old;
METHOD	methods	@XmlElementlic int getCount() {...} public Rectangle(@NamedArg("width") double width, @NamedArg("height") double height) { ... }
PARAMETER	method/constructor parameters	
TYPE	classes, interfaces, enums	@XmlRootElementlic class Report {}

Version ≥ Java SE 8

ElementType target example usage on target element

TYPE_PARAMETER	Type parameter declarations	public <@MyAnnotation T> void f(T t) {}
TYPE_USE	Use of a type	Object o = "42";ing s = (@MyAnnotation String) o;

@Retention

The @Retention meta-annotation defines the annotation visibility during the applications compilation process or execution. By default, annotations are included in .class files, but are not visible at runtime. To make an annotation accessible at runtime, RetentionPolicy.RUNTIME has to be set on that annotation.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    // this annotation can be accessed with reflection at runtime
}
```

Available values

RetentionPolicy	Effect
CLASS	The annotation is available in the .class file, but not at runtime
RUNTIME	The annotation is available at runtime and can be accessed via reflection
SOURCE	The annotation is available at compile time, but not added to the .class files. The annotation can be used e.g. by an annotation processor.

@Documented

The @Documented meta-annotation is used to mark annotations whose usage should be documented by API documentation generators like [javadoc](#). It has no values. With @Documented, all classes that use the annotation will list it on their generated documentation page. Without @Documented, it's not possible to see which classes use the

文档中的注释。

@Inherited

@Inherited 元注解与应用于类的注解相关。它没有值。将注解标记为@Inherited会改变注解查询的方式。

- 对于非继承注解，查询仅检查被查询的类。
- 对于继承注解，查询还会递归检查超类链，直到找到该注解的实例。

注意，只有超类会被查询：类层次结构中附加在接口上的任何注解都会被忽略。

@Repeatable

@Repeatable 元注解是在Java 8中添加的。它表示可以将多个该注解的实例附加到注解的目标上。该元注解没有值。

第43.3节：通过反射进行运行时注解检查

Java的反射API允许程序员在运行时对类的字段、方法和注解执行各种检查和操作。然而，为了使注解在运行时可见，`RetentionPolicy` 必须更改为 `RUNTIME`，如下面的示例所示：

```
@interface MyDefaultAnnotation {  
}  
  
@Retention(RetentionPolicy.RUNTIME)  
@interface MyRuntimeVisibleAnnotation {  
}  
  
public class AnnotationAtRuntimeTest {  
  
    @MyDefaultAnnotation  
    static class RuntimeCheck1 {  
    }  
  
    @MyRuntimeVisibleAnnotation  
    static class RuntimeCheck2 {  
    }  
  
    public static void main(String[] args) {  
        Annotation[] annotationsByType = RuntimeCheck1.class.getAnnotations();  
        Annotation[] annotationsByType2 = RuntimeCheck2.class.getAnnotations();  
  
        System.out.println("default retention: " + Arrays.toString(annotationsByType));  
        System.out.println("runtime retention: " + Arrays.toString(annotationsByType2));  
    }  
}
```

annotation in the documentation.

@Inherited

The @Inherited meta-annotation is relevant to annotations that are applied to classes. It has no values. Marking an annotation as @Inherited alters the way that annotation querying works.

- For a non-inherited annotation, the query only examines the class being examined.
- For an inherited annotation, the query will also check the super-class chain (recursively) until an instance of the annotation is found.

Note that only the super-classes are queried: any annotations attached to interfaces in the classes hierarchy will be ignored.

@Repeatable

The @Repeatable meta-annotation was added in Java 8. It indicates that multiple instances of the annotation can be attached to the annotation's target. This meta-annotation has no values.

Section 43.3: Runtime annotation checks via reflection

Java's Reflection API allows the programmer to perform various checks and operations on class fields, methods and annotations during runtime. However, in order for an annotation to be at all visible at runtime, the `RetentionPolicy` must be changed to `RUNTIME`, as demonstrated in the example below:

```
@interface MyDefaultAnnotation {  
}  
  
@Retention(RetentionPolicy.RUNTIME)  
@interface MyRuntimeVisibleAnnotation {  
}  
  
public class AnnotationAtRuntimeTest {  
  
    @MyDefaultAnnotation  
    static class RuntimeCheck1 {  
    }  
  
    @MyRuntimeVisibleAnnotation  
    static class RuntimeCheck2 {  
    }  
  
    public static void main(String[] args) {  
        Annotation[] annotationsByType = RuntimeCheck1.class.getAnnotations();  
        Annotation[] annotationsByType2 = RuntimeCheck2.class.getAnnotations();  
  
        System.out.println("default retention: " + Arrays.toString(annotationsByType));  
        System.out.println("runtime retention: " + Arrays.toString(annotationsByType2));  
    }  
}
```

第43.4节：内置注解

Java 标准版自带一些预定义的注解。您无需自行定义它们，即可立即使用。它们允许编译器对方法进行一些基本的检查，

Section 43.4: Built-in annotations

The Standard Edition of Java comes with some annotations predefined. You do not need to define them by yourself and you can use them immediately. They allow the compiler to enable some fundamental checking of methods,

类和代码。

@Override

该注解应用于方法，表示此方法必须重写超类的方法或实现抽象超类的方法定义。如果此注解用于任何其他类型的方法，编译器将抛出错误。

具体超类

```
public class 车辆 {  
    public void 驾驶() {  
        System.out.println("我正在驾驶");  
    }  
}  
  
class 汽车 extends 车辆 {  
    // 正确  
    @Override  
    public void 驾驶() {  
        System.out.println("轰隆，轰隆");  
    }  
}
```

抽象类

```
abstract class 动物 {  
    public abstract void 发出声音();  
}  
  
class Dog extends Animal {  
    // 正确  
    @Override  
    public void makeNoise() {  
        System.out.println("Woof");  
    }  
}
```

无法工作

```
class Logger1 {  
    public void log(String logString) {  
        System.out.println(logString);  
    }  
}  
  
class Logger2 {  
    // 这将导致编译时错误。Logger2 不是 Logger1 的子类。  
    // log 方法没有重写任何内容  
    @Override  
    public void log(String logString) {  
        System.out.println("Log 2" + logString);  
    }  
}
```

主要目的是捕捉输入错误，即你以为自己正在重写一个方法，实际上却是在定义一个新的方法。

```
class Vehicle {
```

classes and code.

@Override

This annotation applies to a method and says that this method must override a superclass' method or implement an abstract superclass' method definition. If this annotation is used with any other kind of method, the compiler will throw an error.

Concrete superclass

```
public class Vehicle {  
    public void drive() {  
        System.out.println("I am driving");  
    }  
}  
  
class Car extends Vehicle {  
    // Fine  
    @Override  
    public void drive() {  
        System.out.println("Brrrm, brrm");  
    }  
}
```

Abstract class

```
abstract class Animal {  
    public abstract void makeNoise();  
}  
  
class Dog extends Animal {  
    // Fine  
    @Override  
    public void makeNoise() {  
        System.out.println("Woof");  
    }  
}
```

Does not work

```
class Logger1 {  
    public void log(String logString) {  
        System.out.println(logString);  
    }  
}  
  
class Logger2 {  
    // This will throw compile-time error. Logger2 is not a subclass of Logger1.  
    // log method is not overriding anything  
    @Override  
    public void log(String logString) {  
        System.out.println("Log 2" + logString);  
    }  
}
```

The main purpose is to catch mistyping, where you think you are overriding a method, but are actually defining a new one.

```
class Vehicle {
```

```

public void drive() {
    System.out.println("我正在驾驶");
}

class Car extends Vehicle {
    // 编译错误。"dirve"不是正确的重写方法名。
    @Override
    public void dirve() {
        System.out.println("Brrrm, brrm");
    }
}

```

请注意，@Override 的含义随着时间发生了变化：

- 在 Java 5 中，它表示被注解的方法必须重写超类链中声明的非抽象方法。
- 从 Java 6 开始，如果被注解的方法实现了类的超类/接口层次结构中声明的抽象方法，也满足该要求。

(这有时会在将代码回移植到 Java 5 时引发问题。)

@Deprecated

此标记表示该方法已被弃用。可能有以下几种原因：

- API 存在缺陷且难以修复，
- 使用该 API 可能导致错误，
- 该 API 已被其他 API 取代，
- 该 API 已过时，
- 该 API 处于实验阶段，可能会有不兼容的更改，
- 或上述原因的任意组合。

弃用的具体原因通常可以在该 API 的文档中找到。

如果使用该注解，编译器会发出错误提示。IDE 也可能以某种方式将该方法标记为弃用

```

class ComplexAlgorithm {
    @Deprecated
    public void oldSlowUnthreadSafeMethod() {
        // 这里是代码内容
    }

    public void quickThreadSafeMethod() {
        // 客户端代码应使用此方法
    }
}

```

@SuppressWarnings

在几乎所有情况下，当编译器发出警告时，最合适的做法是修复其原因。在某些情况下（例如，使用非类型安全的预泛型代码的泛型代码），这可能无法实现，此时最好抑制那些你预期且无法修复的警告，这样你就能更清楚地看到意外的警告。

```

public void drive() {
    System.out.println("I am driving");
}

class Car extends Vehicle {
    // Compiler error. "dirve" is not the correct method name to override.
    @Override
    public void dirve() {
        System.out.println("Brrrm, brrm");
    }
}

```

Note that the meaning of @Override has changed over time:

- In Java 5, it meant that the annotated method had to override a non-abstract method declared in the superclass chain.
- From Java 6 onward, it is *also* satisfied if the annotated method implements an abstract method declared in the classes superclass / interface hierarchy.

(This can occasionally cause problems when back-porting code to Java 5.)

@Deprecated

This marks the method as deprecated. There can be several reasons for this:

- the API is flawed and is impractical to fix,
- usage of the API is likely to lead to errors,
- the API has been superseded by another API,
- the API is obsolete,
- the API is experimental and is subject to incompatible changes,
- or any combination of the above.

The specific reason for deprecation can usually be found in the documentation of the API.

The annotation will cause the compiler to emit an error if you use it. IDEs may also highlight this method somehow as deprecated

```

class ComplexAlgorithm {
    @Deprecated
    public void oldSlowUnthreadSafeMethod() {
        // stuff here
    }

    public void quickThreadSafeMethod() {
        // client code should use this instead
    }
}

```

@SuppressWarnings

In almost all cases, when the compiler emits a warning, the most appropriate action is to fix the cause. In some instances (Generics code using untype-safe pre-generics code, for example) this may not be possible and it's better to suppress those warnings that you expect and cannot fix, so you can more clearly see unexpected warnings.

该注解可以应用于整个类、方法或单行。它以警告类别作为参数。

```
@SuppressWarnings("deprecation")
public class RiddledWithWarnings {
    // 这里有多个调用已废弃代码的方法
}

@SuppressWarnings("finally")
public boolean checkData() {
    // 方法中调用了finally块内的return
}
```

最好尽可能限制注解的作用范围，以防止意外的警告也被抑制。例如，将注解的作用范围限制在单行：

```
ComplexAlgorithm algorithm = new ComplexAlgorithm();
@SuppressWarnings("deprecation") algorithm.slowUnthreadSafeMethod();
// 我们在上面的示例中将此方法标记为已废弃

@SuppressWarnings("unsafe") List<Integer> list = getUntypeSafeList();
// 旧库返回的非泛型List，仅包含整数
```

该注解支持的警告类型可能因编译器而异。JLS中特别提到了unchecked和deprecation警告。未识别的警告类型将被忽略。

@SafeVarargs

由于类型擦除，void 方法(T... t) 会被转换为 void 方法(Object[] t)，这意味着编译器并不总能验证可变参数的使用是否类型安全。例如：

```
private static <T> void generatesVarargsWarning(T... lists) {
```

有些情况下使用是安全的，这时可以用 SafeVarargs 注解该方法以抑制警告。当然，如果你的使用不安全，这也会隐藏警告。

@FunctionalInterface

这是一个可选注解，用于标记函数式接口。如果接口不符合函数式接口规范（只有一个抽象方法），编译器会报错。

```
@FunctionalInterface
public interface ITrade {
    public boolean check(Trade t);
}

@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

第43.5节：使用注解处理器进行编译时处理

此示例演示如何对带注解的元素进行编译时检查。

注释

This annotation can be applied to a whole class, method or line. It takes the category of warning as a parameter.

```
@SuppressWarnings("deprecation")
public class RiddledWithWarnings {
    // several methods calling deprecated code here
}

@SuppressWarnings("finally")
public boolean checkData() {
    // method calling return from within finally block
}
```

It is better to limit the scope of the annotation as much as possible, to prevent unexpected warnings also being suppressed. For example, confining the scope of the annotation to a single-line:

```
ComplexAlgorithm algorithm = new ComplexAlgorithm();
@SuppressWarnings("deprecation") algorithm.slowUnthreadSafeMethod();
// we marked this method deprecated in an example above

@SuppressWarnings("unsafe") List<Integer> list = getUntypeSafeList();
// old library returns, non-generic List containing only integers
```

The warnings supported by this annotation may vary from compiler to compiler. Only the unchecked and deprecation warnings are specifically mentioned in the JLS. Unrecognized warning types will be ignored.

@SafeVarargs

Because of type erasure, void method(T... t) will be converted to void method(Object[] t) meaning that the compiler is not always able to verify that the use of varargs is type-safe. For instance:

```
private static <T> void generatesVarargsWarning(T... lists) {
```

There are instances where the use is safe, in which case you can annotate the method with the SafeVarargs annotation to suppress the warning. This obviously hides the warning if your use is unsafe too.

@FunctionalInterface

This is an optional annotation used to mark a FunctionalInterface. It will cause the compiler to complain if it does not conform to the FunctionalInterface spec (has a single abstract method)

```
@FunctionalInterface
public interface ITrade {
    public boolean check(Trade t);
}

@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

Section 43.5: Compile time processing using annotation processor

This example demonstrates how to do compile time checking of an annotated element.

The annotation

@Setter 注解是一个标记，可以应用于方法。该注解在编译过程中会被丢弃，之后不可用。

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface Setter {
}
```

注解处理器

SetterProcessor 类由编译器用来处理注解。它会检查被 @Setter 注解的方法是否为 public、非 static 的方法，且方法名以 set 开头，第四个字母为大写字母。如果不满足这些条件中的任意一条，会向 Messager 写入错误信息。编译器会将此信息写入标准错误输出，但其他工具可能会以不同方式使用这些信息。例如，NetBeans IDE 允许用户指定注解处理器，用于在编辑器中显示错误信息。

```
package annotation.processor;

import annotation.Setter;
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messager;
import javax.annotation.processing.ProcessingEnvironment;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.Modifier;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic;

@SupportedAnnotationTypes({"annotation.Setter"})
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SetterProcessor extends AbstractProcessor {

    private Messager messenger;

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        // 获取带有 @Setter 注解的元素
        Set<? extends Element> annotatedElements = roundEnv.getElementsAnnotatedWith(Setter.class);

        for (Element element : annotatedElements) {
            if (element.getKind() == ElementKind.METHOD) {
                // 仅处理方法作为目标
                checkMethod((ExecutableElement) element);
            }
        }

        // 不声明注解以允许其他处理器处理它们
        return false;
    }
}
```

The @Setter annotation is a marker can be applied to methods. The annotation will be discarded during compilation not be available afterwards.

```
package annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.SOURCE)
@Target(ElementType.METHOD)
public @interface Setter {
}
```

The annotation processor

The SetterProcessor class is used by the compiler to process the annotations. It checks, if the methods annotated with the @Setter annotation are **public**, non-**static** methods with a name starting with set and having a uppercase letter as 4th letter. If one of these conditions isn't met, a error is written to the Messager. The compiler writes this to stderr, but other tools could use this information differently. E.g. the NetBeans IDE allows the user specify annotation processors that are used to display error messages in the editor.

```
package annotation.processor;

import annotation.Setter;
import java.util.Set;
import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.Messager;
import javax.annotation.processing.ProcessingEnvironment;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.annotation.processing.SupportedSourceVersion;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.Element;
import javax.lang.model.element.ElementKind;
import javax.lang.model.element.ExecutableElement;
import javax.lang.model.element.Modifier;
import javax.lang.model.element.TypeElement;
import javax.tools.Diagnostic;

@SupportedAnnotationTypes({"annotation.Setter"})
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SetterProcessor extends AbstractProcessor {

    private Messager messenger;

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment roundEnv) {
        // get elements annotated with the @Setter annotation
        Set<? extends Element> annotatedElements = roundEnv.getElementsAnnotatedWith(Setter.class);

        for (Element element : annotatedElements) {
            if (element.getKind() == ElementKind.METHOD) {
                // only handle methods as targets
                checkMethod((ExecutableElement) element);
            }
        }

        // don't claim annotations to allow other processors to process them
        return false;
    }
}
```

```

}
private void checkMethod(ExecutableElement method) {
    // 检查有效名称
    String name = method.getSimpleName().toString();
    if (!name.startsWith("set")) {
        printError(method, "setter 名称必须以 \"set\" 开头");
    } else if (name.length() == 3) {
        printError(method, "方法名称必须包含超过 \"set\" 的内容");
    } else if (Character.isLowerCase(name.charAt(3))) {
        if (method.getParameters().size() != 1) {
            printError(method, "紧跟 \"set\" 的字符必须是大写字母");
        }
    }

    // 检查 setter 是否为 public
    if (!method.getModifiers().contains(Modifier.PUBLIC)) {
        printError(method, "setter 必须是 public");
    }

    // 检查方法是否为 static
    if (method.getModifiers().contains(Modifier.STATIC)) {
        printError(method, "setter 不能是 static");
    }
}

private void printError(Element element, String message) {
    messenger.printMessage(Diagnostic.Kind.ERROR, message, element);
}

@Override
public void init(ProcessingEnvironment processingEnvironment) {
    super.init(processingEnvironment);

    // 获取用于打印错误的 messenger
    messenger = processingEnvironment.getMessager();
}
}

```

打包

为了被编译器应用，注解处理器需要向 SPI (参见 ServiceLoader) 提供。

为此，需要在包含注解处理器和注解的 jar 文件中，除了其他文件外，还添加一个文本文件 META-INF/services/javax.annotation.processing.Processor。该文件需要包含注解处理器的全限定名，即应如下所示

annotation.processor.SetterProcessor

我们假设下面的jar文件名为AnnotationProcessor.jar。

示例注解类

下面的类是默认包中的示例类，注解根据保留策略应用于正确的元素。然而，注解处理器只将第二个方法视为有效的注解目标。

import annotation.Setter;

```

}
private void checkMethod(ExecutableElement method) {
    // check for valid name
    String name = method.getSimpleName().toString();
    if (!name.startsWith("set")) {
        printError(method, "setter name must start with \"set\"");
    } else if (name.length() == 3) {
        printError(method, "the method name must contain more than just \"set\"");
    } else if (Character.isLowerCase(name.charAt(3))) {
        if (method.getParameters().size() != 1) {
            printError(method, "character following \"set\" must be upper case");
        }
    }

    // check, if setter is public
    if (!method.getModifiers().contains(Modifier.PUBLIC)) {
        printError(method, "setter must be public");
    }

    // check, if method is static
    if (method.getModifiers().contains(Modifier.STATIC)) {
        printError(method, "setter must not be static");
    }
}

private void printError(Element element, String message) {
    messenger.printMessage(Diagnostic.Kind.ERROR, message, element);
}

@Override
public void init(ProcessingEnvironment processingEnvironment) {
    super.init(processingEnvironment);

    // get messenger for printing errors
    messenger = processingEnvironment.getMessager();
}
}

```

Packaging

To be applied by the compiler, the annotation processor needs to be made available to the SPI (see ServiceLoader).

To do this a text file META-INF/services/javax.annotation.processing.Processor needs to be added to the jar file containing the annotation processor and the annotation in addition to the other files. The file needs to include the fully qualified name of the annotation processor, i.e. it should look like this

annotation.processor.SetterProcessor

We'll assume the jar file is called AnnotationProcessor.jar below.

Example annotated class

The following class is example class in the default package with the annotations being applied to the correct elements according to the retention policy. However only the annotation processor only considers the second method a valid annotation target.

import annotation.Setter;

```

public class AnnotationProcessorTest {

    @Setter
    private void setValue(String value) {}

    @Setter
    public void setString(String value) {}

    @Setter
    public static void main(String[] args) {}

}

```

使用 javac 的注解处理器

如果通过 SPI 发现注解处理器，则会自动使用它来处理带注解的元素。例如使用以下命令编译 AnnotationProcessorTest 类

```
javac -cp AnnotationProcessor.jar AnnotationProcessorTest.java
```

将产生以下输出

```

AnnotationProcessorTest.java:6: 错误: setter 必须是 public
private void setValue(String value) {}
^
AnnotationProcessorTest.java:12: 错误: setter 名称必须以 "set" 开头
public static void main(String[] args) {}
^
2 个错误

```

而不是正常编译。没有创建.class文件。

这可以通过为javac指定-proc:none选项来防止。你也可以通过指定-proc:only来放弃通常的编译过程。

IDE集成

NetBeans

注解处理器可以在NetBeans编辑器中使用。为此，需要在项目设置中指定注解处理器：

1. 进入项目属性 > 构建 > 编译
2. 勾选启用注解处理和在编辑器中启用注解处理
3. 点击注解处理器列表旁的添加按钮
4. 在弹出的窗口中输入注解处理器的全限定类名，然后点击确定。

结果

```

public class AnnotationProcessorTest {

    @Setter
    private void setValue(String value) {}

    @Setter
    public void setString(String value) {}

    @Setter
    public static void main(String[] args) {}

}

```

Using the annotation processor with javac

If the annotation processor is discovered using the SPI, it is automatically used to process annotated elements. E.g. compiling the AnnotationProcessorTest class using

```
javac -cp AnnotationProcessor.jar AnnotationProcessorTest.java
```

yields the following output

```

AnnotationProcessorTest.java:6: error: setter must be public
private void setValue(String value) {}
^
AnnotationProcessorTest.java:12: error: setter name must start with "set"
public static void main(String[] args) {}
^
2 errors

```

instead of compiling normally. No .class file is created.

This could be prevented by specifying the -proc:none option for javac. You could also forgo the usual compilation by specifying -proc:only instead.

IDE integration

Netbeans

Annotation processors can be used in the NetBeans editor. To do this the annotation processor needs to be specified in the project settings:

1. go to Project [Properties](#) > Build > Compiling
2. add check marks for Enable [Annotation Processing](#) and Enable [Annotation Processing in Editor](#)
3. click Add next to the annotation processor list
4. in the popup that appears enter the fully qualified class name of the annotation processor and click ok.

Result

```

1 import annotation.Setter;
2
3 public class Annotation {
4     ----
5     @Setter
6     private void setValue(String value) {}
7
8     @Setter
9     public void setString(String value) {}
10
11    @Setter
12    public static void main(String[] args) {}
13
14 }
15

```

第43.6节：重复注解

直到Java 8，不能在单个元素上应用两个相同的注解。标准的解决方法是使用一个容器注解，包含某个其他注解的数组：

```

// Author.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
    String value();
}

// Authors.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors {
    Author[] value();
}

// Test.java
@Authors({
    @Author("Mary"),
    @Author("Sam")
})
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotation(Authors.class).value();
        for (Author author : authors) {
            System.out.println(author.value());
            // 输出：
            // Mary
            // Sam
        }
    }
}

```

版本 ≥ Java SE 8

Java 8 提供了一种更简洁、更透明的使用容器注解的方法，使用了@Repeatable注解。
首先我们将此添加到Author类中：

```
@Repeatable(Authors.class)
```

这告诉Java将多个@Author注解视为被@Authors容器包围的情况。

我们也可以使用Class.getAnnotationsByType()通过其自身的类访问@Author数组，而不是通过其

```

1 import annotation.Setter;
2
3 public class Annotation {
4     ----
5     @Setter
6     private void setValue(String value) {}
7
8     @Setter
9     public void setString(String value) {}
10
11    @Setter
12    public static void main(String[] args) {}
13
14 }
15

```

Section 43.6: Repeating Annotations

Until Java 8, two instances of the same annotation could not be applied to a single element. The standard workaround was to use a container annotation holding an array of some other annotation:

```

// Author.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
    String value();
}

// Authors.java
@Retention(RetentionPolicy.RUNTIME)
public @interface Authors {
    Author[] value();
}

// Test.java
@Authors({
    @Author("Mary"),
    @Author("Sam")
})
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotation(Authors.class).value();
        for (Author author : authors) {
            System.out.println(author.value());
            // Output:
            // Mary
            // Sam
        }
    }
}

```

Version ≥ Java SE 8

Java 8 provides a cleaner, more transparent way of using container annotations, using the @Repeatable annotation.
First we add this to the Author class:

```
@Repeatable(Authors.class)
```

This tells Java to treat multiple @Author annotations as though they were surrounded by the @Authors container.

We can also use `Class.getAnnotationsByType()` to access the @Author array by its own class, instead of through its

容器：

```
@Author("Mary")
@Author("Sam")
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotationsByType(Author.class);
        for (Author author : authors) {
            System.out.println(author.value());
        }
    }
}
```

第43.7节：继承注解

默认情况下，类注解不适用于继承它们的类型。可以通过在注解定义中添加@Inherited注解来更改此行为

示例

考虑以下两个注解：

```
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface InheritedAnnotationType { }
```

和

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface UninheritedAnnotationType { }
```

如果三个类被这样注解：

```
@UninheritedAnnotationType
类 A {
}

@InheritedAnnotationType
类 B extends A {
}

类 C extends B { }
```

运行此代码

```
System.out.println(new A().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println("-----");
System.out.println(new A().getClass().getAnnotation(UninheritedAnnotationType.class));
```

container:

```
@Author("Mary")
@Author("Sam")
public class Test {
    public static void main(String[] args) {
        Author[] authors = Test.class.getAnnotationsByType(Author.class);
        for (Author author : authors) {
            System.out.println(author.value());
        }
    }
}
```

Section 43.7: Inherited Annotations

By default class annotations do not apply to types extending them. This can be changed by adding the @Inherited annotation to the annotation definition

Example

Consider the following 2 Annotations:

```
@Inherited
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface InheritedAnnotationType { }
```

and

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface UninheritedAnnotationType { }
```

If three classes are annotated like this:

```
@UninheritedAnnotationType
class A {
}

@InheritedAnnotationType
class B extends A {
}

class C extends B { }
```

running this code

```
System.out.println(new A().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new B().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(InheritedAnnotationType.class));
System.out.println("-----");
System.out.println(new A().getClass().getAnnotation(UninheritedAnnotationType.class));
```

```
System.out.println(new B().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(UninheritedAnnotationType.class));
```

将打印类似以下结果（取决于注解所在的包）：

```
null
@InheritedAnnotationType()
@InheritedAnnotationType()

-----
@UninheritedAnnotationType()
null
null
```

注意，注解只能从类继承，不能从接口继承。

第43.8节：运行时获取注解值

你可以通过反射获取应用了注解的方法、字段或类，然后获取所需的属性，从而获取注解的当前属性值。

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    String key() default "foo";
    String value() default "bar";
}

class AnnotationExample {
    // 将注解放在方法上，但保留默认值
    @MyAnnotation
    public void testDefaults() throws Exception {
        // 使用反射，获取无参数的公共方法 "testDefaults"，即此方法
        Method method = AnnotationExample.class.getMethod("testDefaults", null);

        // 从方法中获取类型为 MyAnnotation 的注解
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // 打印注解的设置
        print(annotation);
    }

    // 将注解放在方法上，但覆盖默认设置
    @MyAnnotation(key="baz", value="buzz")
    public void testValues() throws Exception {
        // 使用反射，获取公共方法"testValues"，即这个无参数的方法
        Method method = AnnotationExample.class.getMethod("testValues", null);

        // 从方法中获取类型为 MyAnnotation 的注解
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // 打印注解的设置
        print(annotation);
    }

    public void print(MyAnnotation annotation) {
        // 获取 MyAnnotation 的 'key' 和 'value' 属性，并打印出来
        System.out.println(annotation.key() + " = " + annotation.value());
    }

    public static void main(String[] args) {
```

```
System.out.println(new B().getClass().getAnnotation(UninheritedAnnotationType.class));
System.out.println(new C().getClass().getAnnotation(UninheritedAnnotationType.class));
```

will print a result similar to this (depending on the packages of the annotation):

```
null
@InheritedAnnotationType()
@InheritedAnnotationType()

-----
@UninheritedAnnotationType()
null
null
```

Note that annotations can only be inherited from classes, not interfaces.

Section 43.8: Getting Annotation values at run-time

You can fetch the current properties of the Annotation by using Reflection to fetch the Method or Field or Class which has an Annotation applied to it, and then fetching the desired properties.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    String key() default "foo";
    String value() default "bar";
}

class AnnotationExample {
    // Put the Annotation on the method, but leave the defaults
    @MyAnnotation
    public void testDefaults() throws Exception {
        // Using reflection, get the public method "testDefaults", which is this method with no args
        Method method = AnnotationExample.class.getMethod("testDefaults", null);

        // Fetch the Annotation that is of type MyAnnotation from the Method
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // Print out the settings of the Annotation
        print(annotation);
    }

    //Put the Annotation on the method, but override the settings
    @MyAnnotation(key="baz", value="buzz")
    public void testValues() throws Exception {
        // Using reflection, get the public method "testValues", which is this method with no args
        Method method = AnnotationExample.class.getMethod("testValues", null);

        // Fetch the Annotation that is of type MyAnnotation from the Method
        MyAnnotation annotation = (MyAnnotation)method.getAnnotation(MyAnnotation.class);

        // Print out the settings of the Annotation
        print(annotation);
    }

    public void print(MyAnnotation annotation) {
        // Fetch the MyAnnotation 'key' & 'value' properties, and print them out
        System.out.println(annotation.key() + " = " + annotation.value());
    }

    public static void main(String[] args) {
```

```

AnnotationExample example = new AnnotationExample();
try {
example.testDefaults();
    example.testValues();
} catch( Exception e ) {
    // 不应该抛出任何异常
    System.err.println("Exception [" + e.getClassName() + "] - " + e.getMessage());
e.printStackTrace(System.err);
}
}

```

输出将会是

```

foo = bar
baz = buzz

```

第43.9节：“this”和接收者参数的注解

当Java注解首次引入时，没有为实例方法的目标或内部类构造函数的隐藏构造参数提供注解的规定。这个问题在Java 8中通过添加接收者参数声明得到了修正；参见JLS 8.4.1。

接收者参数是实例方法或内部类构造函数的一个可选语法装置。对于实例方法，接收者参数表示调用该方法的对象。对于内部类的构造函数，接收者参数表示新构造对象的直接封闭实例。无论哪种情况，接收者参数的存在仅仅是为了在源代码中表示所代表对象的类型，以便该类型可以被注解。接收者参数不是形式参数；更准确地说，它不是任何变量的声明
（§4.12.3），它从不绑定到方法调用表达式或限定类实例创建表达式中传递的任何参数值，并且在运行时完全没有任何影响。

下面的示例说明了两种接收者参数的语法：

```

public class Outer {
    public class Inner {
        public Inner (Outer this) {
            // ...
        }
        public void doIt(Inner this) {
            // ...
        }
    }
}

```

接收者参数的唯一目的是允许你添加注解。例如，你可能有一个自定义注解@IsOpen，其目的是断言在调用方法时Closeable对象尚未关闭。例如：

```

public class MyResource extends Closeable {
    public void update(@IsOpen MyResource this, int value) {
        // ...
    }
}

```

```

AnnotationExample example = new AnnotationExample();
try {
    example.testDefaults();
    example.testValues();
} catch( Exception e ) {
    // Shouldn't throw any Exceptions
    System.err.println("Exception [" + e.getClassName() + "] - " + e.getMessage());
    e.printStackTrace(System.err);
}
}

```

The output will be

```

foo = bar
baz = buzz

```

Section 43.9: Annotations for 'this' and receiver parameters

When Java annotations were first introduced there was no provision for annotating the target of an instance method or the hidden constructor parameter for an inner classes constructor. This was remedied in Java 8 with addition of *receiver parameter declarations*; see [JLS 8.4.1](#).

The receiver parameter is an optional syntactic device for an instance method or an inner class's constructor. For an instance method, the receiver parameter represents the object for which the method is invoked. For an inner class's constructor, the receiver parameter represents the immediately enclosing instance of the newly constructed object. Either way, the receiver parameter exists solely to allow the type of the represented object to be denoted in source code, so that the type may be annotated. The receiver parameter is not a formal parameter; more precisely, it is not a declaration of any kind of variable (§4.12.3), it is never bound to any value passed as an argument in a method invocation expression or qualified class instance creation expression, and it has no effect whatsoever at run time.

The following example illustrates the syntax for both kinds of receiver parameter:

```

public class Outer {
    public class Inner {
        public Inner (Outer this) {
            // ...
        }
        public void doIt(Inner this) {
            // ...
        }
    }
}

```

The sole purpose of receiver parameters is to allow you to add annotations. For example, you might have a custom annotation @IsOpen whose purpose is to assert that a Closeable object has not been closed when a method is called. For example:

```

public class MyResource extends Closeable {
    public void update(@IsOpen MyResource this, int value) {
        // ...
    }
}

```

```
public void close() {  
    // ...  
}
```

在某种程度上，@IsOpen 注解在 this 上可以仅作为文档说明。然而，我们也可以做更多的事情。例如：

- 注解处理器可以插入运行时检查，确保在调用 update 时 this 不是关闭状态。
- 代码检查器可以执行静态代码分析，查找在调用 update 时 this 可能处于关闭状态的情况。

第43.10节：添加多个注解值

如果注解参数定义为数组，则可以接受多个值。例如，标准注解 @SuppressWarnings 定义如下：

```
public @interface SuppressWarnings {  
    String[] value();  
}
```

value 参数是一个字符串数组。你可以使用类似数组初始化的语法设置多个值：

```
@SuppressWarnings({"unused"})  
@SuppressWarnings({"unused", "javadoc"})
```

如果只需要设置单个值，可以省略大括号：

```
@SuppressWarnings("unused")
```

```
public void close() {  
    // ...  
}
```

At one level, the @IsOpen annotation on **this** could simply serve as documentation. However, we could potentially do more. For example:

- An annotation processor could insert a runtime check that **this** is not in closed state when update is called.
- A code checker could perform a static code analysis to find cases where **this** could be closed when update is called.

Section 43.10: Add multiple annotation values

An Annotation parameter can accept multiple values if it is defined as an array. For example the standard annotation @SuppressWarnings is defined like this:

```
public @interface SuppressWarnings {  
    String[] value();  
}
```

The value parameter is an array of Strings. You can set multiple values by using a notation similar to Array initializers:

```
@SuppressWarnings({"unused"})  
@SuppressWarnings({"unused", "javadoc"})
```

If you only need to set a single value, the brackets can be omitted:

```
@SuppressWarnings("unused")
```

第44章：不可变类

不可变对象是指其状态在初始化后不会改变的实例。例如，String是一个不可变类，一旦实例化，其值永远不会改变。

第44.1节：无可变引用的示例

```
public final class Color {  
    final private int red;  
    final private int green;  
    final private int blue;  
  
    private void check(int red, int green, int blue) {  
        if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255) {  
            throw new IllegalArgumentException();  
        }  
    }  
  
    public Color(int red, int green, int blue) {  
        check(red, green, blue);  
        this.red = red;  
        this.green = green;  
        this.blue = blue;  
    }  
  
    public Color invert() {  
        return new Color(255 - red, 255 - green, 255 - blue);  
    }  
}
```

第44.2节：不可变性的优势是什么？

不可变性的优势体现在并发性上。在可变对象中维护正确性是困难的，因为多个线程可能试图更改同一对象的状态，导致某些线程根据对该对象的读写时机，看到同一对象的不同状态。

通过使用不可变对象，可以确保所有查看该对象的线程看到的状态都是相同的，因为不可变对象的状态不会改变。

第44.3节：定义不可变类的规则

以下规则定义了创建不可变对象的简单策略。

1. 不要提供“setter”方法——即修改字段或字段引用的对象的方法。
2. 将所有字段声明为final且private。
3. 不允许子类重写方法。最简单的方法是将类声明为final。更复杂的方法是将构造函数设为private，并在工厂方法中构造实例。
4. 如果实例字段包含对可变对象的引用，不允许更改这些对象：
5. 不要提供修改可变对象的方法。
6. 不要共享对可变对象的引用。绝不存储传入构造函数的外部可变对象的引用；如有必要，创建副本，并存储对副本的引用。同样，在必要时创建内部可变对象的副本，以避免在方法中返回原始对象。

Chapter 44: Immutable Class

Immutable objects are instances whose state doesn't change after it has been initialized. For example, String is an immutable class and once instantiated its value never changes.

Section 44.1: Example without mutable refs

```
public final class Color {  
    final private int red;  
    final private int green;  
    final private int blue;  
  
    private void check(int red, int green, int blue) {  
        if (red < 0 || red > 255 || green < 0 || green > 255 || blue < 0 || blue > 255) {  
            throw new IllegalArgumentException();  
        }  
    }  
  
    public Color(int red, int green, int blue) {  
        check(red, green, blue);  
        this.red = red;  
        this.green = green;  
        this.blue = blue;  
    }  
  
    public Color invert() {  
        return new Color(255 - red, 255 - green, 255 - blue);  
    }  
}
```

Section 44.2: What is the advantage of immutability?

The advantage of immutability comes with concurrency. It is difficult to maintain correctness in mutable objects, as multiple threads could be trying to change the state of the same object, leading to some threads seeing a different state of the same object, depending on the timing of the reads and writes to the said object.

By having an immutable object, one can ensure that all threads that are looking at the object will be seeing the same state, as the state of an immutable object will not change.

Section 44.3: Rules to define immutable classes

The following rules define a simple strategy for creating immutable objects.

1. Don't provide "setter" methods - methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
5. Don't provide methods that modify the mutable objects.
6. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

第44.4节：可变引用的示例

在这种情况下，类Point是可变的，某些用户可以修改该类对象的状态。

```
类Point {
    私有int x, y;

    公共Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    公共int getX() {
        返回 x;
    }

    公共void setX(int x) {
        this.x = x;
    }

    公共int getY() {
        返回 y;
    }

    公共void setY(int y) {
        this.y = y;
    }

}

//...

public final class ImmutableCircle {
    私有 final Point center;
    私有 final double radius;

    public ImmutableCircle(Point center, double radius) {
        // 我们在这里创建新对象，因为它不应该被更改
        this.center = new Point(center.getX(), center.getY());
        this.radius = radius;
    }
}
```

Section 44.4: Example with mutable refs

In this case class Point is mutable and some user can modify state of object of this class.

```
class Point {
    私有 int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        返回 x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        返回 y;
    }

    public void setY(int y) {
        this.y = y;
    }

}

//...

public final class ImmutableCircle {
    私有 final Point center;
    私有 final double radius;

    public ImmutableCircle(Point center, double radius) {
        // we create new object here because it shouldn't be changed
        this.center = new Point(center.getX(), center.getY());
        this.radius = radius;
    }
}
```

第45章：不可变对象

第45.1节：使用防御性复制创建类型的不可变版本

Java中的一些基本类型和类本质上是可变的。例如，所有数组类型都是可变的，像java.util.Date这样的类也是如此。在需要不可变类型的情况下，这可能会很麻烦。

一种解决方法是为可变类型创建一个不可变的包装器。下面是一个整数数组的简单包装器

```
public class ImmutableListIntArray {  
    private final int[] array;  
  
    public ImmutableListIntArray(int[] array) {  
        this.array = array.clone();  
    }  
  
    public int[] getValue() {  
        return this.clone();  
    }  
}
```

该类通过使用防御性复制来隔离可变状态（即int[]）与可能修改它的任何代码：

- 构造函数使用clone()来创建参数数组的独立副本。如果构造函数的调用者随后更改了参数数组，这不会影响ImmutableIntArray的状态。
- getValue() 方法也使用 clone() 来创建返回的数组。如果调用者更改结果数组，不会影响 ImmutableListIntArray 的状态。

我们还可以为 ImmutableListIntArray 添加方法，以对封装的数组执行只读操作；例如获取其长度、获取特定索引处的值等。

请注意，以这种方式实现的不可变包装类型与原始类型在类型上不兼容。你不能简单地用前者替代后者。

第45.2节：不可变类的制作方法

不可变对象是状态不可更改的对象。不可变类是其实例在设计和实现上都是不可变的类。Java 中最常被用作不可变性示例的类是 java.lang.String。

下面是一个典型的示例：

```
public final class Person {  
    private final String name;  
    private final String ssn; // (SSN == 社会保障号码)  
  
    public Person(String name, String ssn) {  
        this.name = name;  
        this.ssn = ssn;  
    }  
}
```

Chapter 45: Immutable Objects

Section 45.1: Creating an immutable version of a type using defensive copying

Some basic types and classes in Java are fundamentally mutable. For example, all array types are mutable, and so are classes like `java.util.Date`. This can be awkward in situations where an immutable type is mandated.

One way to deal with this is to create an immutable wrapper for the mutable type. Here is a simple wrapper for an array of integers

```
public class ImmutableListIntArray {  
    private final int[] array;  
  
    public ImmutableListIntArray(int[] array) {  
        this.array = array.clone();  
    }  
  
    public int[] getValue() {  
        return this.clone();  
    }  
}
```

This class works by using *defensive copying* to isolate the mutable state (the `int[]`) from any code that might mutate it:

- The constructor uses `clone()` to create a distinct copy of the parameter array. If the caller of the constructor subsequently changes the parameter array, it would not affect the state of the `ImmutableIntArray`.
- The `getValue()` method also uses `clone()` to create the array that is returned. If the caller were to change the result array, it would not affect the state of the `ImmutableIntArray`.

We could also add methods to `ImmutableIntArray` to perform read-only operations on the wrapped array; e.g. get its length, get the value at a particular index, and so on.

Note that an immutable wrapper type implemented this way is not type compatible with the original type. You cannot simply substitute the former for the latter.

Section 45.2: The recipe for an immutable class

An immutable object is an object whose state cannot be changed. An immutable class is a class whose instances are immutable by design, and implementation. The Java class which is most commonly presented as an example of immutability is `java.lang.String`.

The following is a stereotypical example:

```
public final class Person {  
    private final String name;  
    private final String ssn; // (SSN == social security number)  
  
    public Person(String name, String ssn) {  
        this.name = name;  
        this.ssn = ssn;  
    }  
}
```

```

public String getName() {
    return name;
}

public String getSSN() {
    return ssn;
}

```

一种变体是将构造函数声明为private，并提供一个public static的工厂方法代替。

不可变类的标准做法如下：

- 所有属性必须在构造函数或工厂方法中设置。
- 不应有设置器（setter）。
- 如果出于接口兼容性原因必须包含设置器，它们要么不执行任何操作，要么抛出异常。
- 所有属性应声明为private和final。
- 对于所有引用可变类型的属性：
 - 该属性应使用通过构造函数传入值的深拷贝进行初始化，且该属性的getter应返回属性值的深拷贝。
 -
- 该类应声明为final，以防止有人创建不可变类的可变子类。

还有几点需要注意：

- 不可变性并不阻止对象为可空；例如，null 可以赋值给 String 变量。
- 如果不可变类的属性声明为 final，则其实例本质上是线程安全的。这使得不可变类成为实现多线程应用程序的良好构建模块。

第45.3节：阻止类成为不可变的典型设计缺陷

使用某些setter方法，但未在构造函数中设置所有必需的属性

```

public final class Person { // 不良不可变性的示例
    private final String name;
    private final String surname;
    public Person(String name) {
        this.name = name;
    }
    public String getName() { return name; }
    public String getSurname() { return surname; }
    public void setSurname(String surname) { this.surname = surname; }
}

```

很容易证明 Person 类不是不可变的：

```

Person person = new Person("Joe");
person.setSurname("Average"); // 不可，创建后更改姓氏字段

```

要修复它，只需删除setSurname()并按如下方式重构构造函数：

```

public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
}

```

```

public String getName() {
    return name;
}

public String getSSN() {
    return ssn;
}

```

A variation on this is to declare the constructor as **private** and provide a **public static** factory method instead.

The standard recipe for an immutable class is as follows:

- All properties must be set in the constructor(s) or factory method(s).
- There should be no setters.
- If it is necessary to include setters for interface compatibility reasons, they should either do nothing or throw an exception.
- All properties should be declared as **private** and **final**.
- For all properties that are references to mutable types:
 - the property should be initialized with a deep copy of the value passed via the constructor, and
 - the property's getter should return a deep copy of the property value.
- The class should be declared as **final** to prevent someone creating a mutable subclass of an immutable class.

A couple of other things to note:

- Immutability does not prevent object from being nullable; e.g. **null** can be assigned to a **String** variable.
- If an immutable classes properties are declared as **final**, instances are inherently thread-safe. This makes immutable classes a good building block for implementing multi-threaded applications.

Section 45.3: Typical design flaws which prevent a class from being immutable

Using some setters, without setting all needed properties in the constructor(s)

```

public final class Person { // example of a bad immutability
    private final String name;
    private final String surname;
    public Person(String name) {
        this.name = name;
    }
    public String getName() { return name; }
    public String getSurname() { return surname; }
    public void setSurname(String surname) { this.surname = surname; }
}

```

It's easy to show that Person class is not immutable:

```

Person person = new Person("Joe");
person.setSurname("Average"); // NOT OK, change surname field after creation

```

To fix it, simply delete setSurname() and refactor the constructor as follows:

```

public Person(String name, String surname) {
    this.name = name;
    this.surname = surname;
}

```

}

未将实例变量标记为private和final

看看下面的类：

```
public final class Person {
    public String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

下面的代码片段显示上述类不是不可变的：

```
Person person = new Person("Average Joe");
person.name = "Magic Mike"; // 不正确，创建后修改了person的新名字
```

要修复它，只需将name属性标记为private和final。

在getter中暴露类的可变对象

看看下面的类：

```
import java.util.List;
import java.util.ArrayList;
public final class 名字 {
    private final List<String> 名字列表;
    public 名字(List<String> 名字列表) {
        this.名字列表 = new ArrayList<String>(名字列表);
    }
    public List<String> 获取名字列表() {
        return 名字列表;
    }
    public int 大小() {
        return 名字列表.大小();
    }
}
```

名字类乍一看似乎是不可变的，但如下代码所示并非如此：

```
List<String> 名字列表 = new ArrayList<String>();
名字列表添加("Average Joe");
名字 对象 = new 名字(名字列表);
System.out.println(名字对象.大小()); // 1, 仅包含 "Average Joe"
名字列表 = 名字对象.获取名字列表();
名字列表添加("Magic Mike");
System.out.println(名字对象.大小()); // 2, 不正确，现在名字对象也包含了 "Magic Mike"
```

这是因为对getNames()返回的引用列表的更改可能会修改实际的Names列表。

为了解决这个问题，只需避免返回引用类中可变对象的引用，可以通过制作防御性拷贝，具体如下：

}

Not marking instance variables as private and final

Take a look at the following class:

```
public final class Person {
    public String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

The following snippet shows that the above class is not immutable:

```
Person person = new Person("Average Joe");
person.name = "Magic Mike"; // not OK, new name for person after creation
```

To fix it, simply mark name property as **private** and **final**.

Exposing a mutable object of the class in a getter

Take a look at the following class:

```
import java.util.List;
import java.util.ArrayList;
public final class Names {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = new ArrayList<String>(names);
    }
    public List<String> getNames() {
        return names;
    }
    public int size() {
        return names.size();
    }
}
```

Names class seems immutable at the first sight, but it is not as the following code shows:

```
List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
Names names = new Names(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList = names.getNames();
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"
```

This happened because a change to the reference List returned by getNames() can modify the actual list of Names.

To fix this, simply avoid returning references that reference class's mutable objects *either* by making defensive copies, as follows:

```
public List<String> getNames() {
    return new ArrayList<String>(this.names); // 复制元素
}
```

或者通过设计getter方法，使其只返回其他不可变对象和基本类型，具体如下：

```
public String getName(int index) {
    return names.get(index);
}
public int 大小() {
    return 名字列表.大小();
}
```

通过构造函数注入可以在不可变类外部被修改的对象

这是前面缺陷的一种变体。请看下面的类：

```
import java.util.List;
public final class NewNames {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = names;
    }
    public String getName(int index) {
        return names.get(index);
    }
    public int 大小() {
        return 名字列表.大小();
    }
}
```

和之前的Names类一样，NewNames类乍一看似乎是不可变的，但实际上并非如此，下面的代码片段证明了这一点：

```
List<String> 名字列表 = new ArrayList<String>();
名字列表添加("Average Joe");
NewNames names = new NewNames(namesList);
System.out.println(names.size()); // 1, 只包含"Average Joe"
namesList.add("Magic Mike");
System.out.println(名字对象.大小()); // 2, 不正确, 现在名字对象也包含了 "Magic Mike"
```

为了解决这个问题，与之前的缺陷类似，只需对对象进行防御性复制，而不是直接赋值给不可变类，即构造函数可以改为如下：

```
public Names(List<String> names) {
    this.names = new ArrayList<String>(names);
}
```

允许类的方法被重写

看看下面的类：

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() { return name; }
```

```
public List<String> getNames() {
    return new ArrayList<String>(this.names); // copies elements
}
```

or by designing getters in way that only other *immutable objects* and *primitives* are returned, as follows:

```
public String getName(int index) {
    return names.get(index);
}
public int size() {
    return names.size();
}
```

Injecting constructor with object(s) that can be modified outside the immutable class

This is a variation of the previous flaw. Take a look at the following class:

```
import java.util.List;
public final class NewNames {
    private final List<String> names;
    public Names(List<String> names) {
        this.names = names;
    }
    public String getName(int index) {
        return names.get(index);
    }
    public int size() {
        return names.size();
    }
}
```

As Names class before, also NewNames class seems immutable at the first sight, but it is not, in fact the following snippet proves the contrary:

```
List<String> namesList = new ArrayList<String>();
namesList.add("Average Joe");
NewNames names = new NewNames(namesList);
System.out.println(names.size()); // 1, only containing "Average Joe"
namesList.add("Magic Mike");
System.out.println(names.size()); // 2, NOT OK, now names also contains "Magic Mike"
```

To fix this, as in the previous flaw, simply make defensive copies of the object without assigning it directly to the immutable class, i.e. constructor can be changed as follows:

```
public Names(List<String> names) {
    this.names = new ArrayList<String>(names);
}
```

Letting the methods of the class being overridden

Take a look at the following class:

```
public class Person {
    private final String name;
    public Person(String name) {
        this.name = name;
    }
    public String getName() { return name; }
```

}

Person 类乍一看似乎是不可变的，但假设定义了一个新的 Person 子类：

```
public class MutablePerson extends Person {  
    private String newName;  
    public MutablePerson(String name) {  
        super(name);  
    }  
    @Override  
    public String getName() {  
        return newName;  
    }  
    public void setName(String name) {  
        newName = name;  
    }  
}
```

}

Person class seems immutable at the first sight, but suppose a new subclass of Person is defined:

```
public class MutablePerson extends Person {  
    private String newName;  
    public MutablePerson(String name) {  
        super(name);  
    }  
    @Override  
    public String getName() {  
        return newName;  
    }  
    public void setName(String name) {  
        newName = name;  
    }  
}
```

现在Person (不可变性) 可以通过使用新的子类来利用多态性：

```
Person person = new MutablePerson("普通乔");  
System.out.println(person.getName()); 打印 普通乔  
person.setName("魔术麦克"); // 不允许, person 现在有了一个新名字!  
System.out.println(person.getName()); // 打印 魔术麦克
```

为了解决这个问题，要么将类标记为final以防止被继承，要么将所有构造函数声明为private。

now Person (im)mutability can be exploited through polymorphism by using the new subclass:

```
Person person = new MutablePerson("Average Joe");  
System.out.println(person.getName()); prints Average Joe  
person.setName("Magic Mike"); // NOT OK, person has now a new name!  
System.out.println(person.getName()); // prints Magic Mike
```

To fix this, either mark the class as **final** so it cannot be extended or declare all of its constructor(s) as **private**.

第46章：可见性（控制类成员的访问权限）

第46.1节：私有可见性

private 可见性允许变量只能被其所属类访问。它们通常与 **public** getter 和 setter 一起使用。

```
class SomeClass {  
    private int variable;  
  
    public int getVariable() {  
        return variable;  
    }  
  
    public void setVariable(int variable) {  
        this.variable = variable;  
    }  
}  
  
public class SomeOtherClass {  
    public static void main(String[] args) {  
        SomeClass sc = new SomeClass();  
  
        // 这些语句无法编译，因为 SomeClass#variable 是私有的：  
        sc.variable = 7;  
        System.out.println(sc.variable);  
  
        // 相反，你应该使用公共的 getter 和 setter：  
        sc.setVariable(7);  
        System.out.println(sc.getVariable());  
    }  
}
```

第46.2节：公共可见性

对类、包和子类可见。

让我们看一个 Test 类的例子。

```
public class Test{  
    public int number = 2;  
  
    public Test(){  
    }  
}
```

现在让我们尝试创建该类的一个实例。在这个例子中，我们可以访问 number，因为它是 public 的。

```
public class Other{  
  
    public static void main(String[] args){  
        Test t = new Test();  
        System.out.println(t.number);  
    }  
}
```

Chapter 46: Visibility (controlling access to members of a class)

Section 46.1: Private Visibility

private visibility allows a variable to only be accessed by its class. They are often used in **conjunction** with **public** getters and setters.

```
class SomeClass {  
    private int variable;  
  
    public int getVariable() {  
        return variable;  
    }  
  
    public void setVariable(int variable) {  
        this.variable = variable;  
    }  
}  
  
public class SomeOtherClass {  
    public static void main(String[] args) {  
        SomeClass sc = new SomeClass();  
  
        // These statement won't compile because SomeClass#variable is private:  
        sc.variable = 7;  
        System.out.println(sc.variable);  
  
        // Instead, you should use the public getter and setter:  
        sc.setVariable(7);  
        System.out.println(sc.getVariable());  
    }  
}
```

Section 46.2: Public Visibility

Visible to the class, package, and subclass.

Let's see an example with the class Test.

```
public class Test{  
    public int number = 2;  
  
    public Test(){  
    }  
}
```

Now let's try to create an instance of the class. In this example, **we can** access number because it is **public**.

```
public class Other{  
  
    public static void main(String[] args){  
        Test t = new Test();  
        System.out.println(t.number);  
    }  
}
```

}

第46.3节：包可见性

无修饰符时，默认是包可见性。根据Java文档，“[包可见性]表示同一包中的类（无论其继承关系如何）是否可以访问该成员。”在这个来自javax.swing的例子中，

```
package javax.swing;
public abstract class JComponent extends Container ... {
    ...
    static boolean DEBUG_GRAPHICS_LOADED;
    ...
}
```

DebugGraphics 位于同一包中，因此 DEBUG_GRAPHICS_LOADED 是可访问的。

```
package javax.swing;
public class DebugGraphics extends Graphics {
    ...
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    ...
}
```

本文介绍了该主题的一些背景知识。

第46.4节：受保护的可见性

受保护的可见性意味着该成员对其包内以及其所有子类可见。

举个例子：

```
包com.stackexchange.docs;
公共类 MyClass{
    受保护的int变量；//这是我们尝试访问的变量
    公共 MyClass(){
        变量= 2;
    }
}
```

现在我们将扩展此类并尝试访问其一个受保护成员。

```
包some.other.pack;
导入com.stackexchange.docs.MyClass;
公共类 SubClass 继承 MyClass{
    公共 SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

如果你从同一个包中访问，也可以访问一个受保护的成员，而无需继承它。

}

Section 46.3: Package Visibility

With **no modifier**, the default is package visibility. *From the Java Documentation*, "[package visibility] indicates whether classes in the same package as the class (regardless of their parentage) have access to the member." In this example from [javax.swing](#),

```
package javax.swing;
public abstract class JComponent extends Container ... {
    ...
    static boolean DEBUG_GRAPHICS_LOADED;
    ...
}
```

DebugGraphics is in the same package, so DEBUG_GRAPHICS_LOADED is accessible.

```
package javax.swing;
public class DebugGraphics extends Graphics {
    ...
    static {
        JComponent.DEBUG_GRAPHICS_LOADED = true;
    }
    ...
}
```

This [article](#) gives some background on the topic.

Section 46.4: Protected Visibility

Protected visibility causes means that this member is visible to its package, along with any of its subclasses.

As an example:

```
package com.stackexchange.docs;
public class MyClass{
    protected int variable; //This is the variable that we are trying to access
    public MyClass(){
        variable = 2;
    }
}
```

Now we'll extend this class and try to access one of its **protected** members.

```
package some.other.pack;
import com.stackexchange.docs.MyClass;
public class SubClass extends MyClass{
    public SubClass(){
        super();
        System.out.println(super.variable);
    }
}
```

You would also be able to access a **protected** member without extending it if you are accessing it from the same package.

请注意，该修饰符仅适用于类的成员，而不适用于类本身。

第46.5节：类成员访问修饰符总结

访问修饰符	可见性	继承性
私有	仅限类内部不能被继承	
无修饰符 / 包内可用如果子类在包内则可用		
受保护	在包内子类可用	
公共的	在子类中随处可用	

曾经有一个private protected（两个关键字同时使用）的修饰符，可以应用于方法或变量，使它们对子类（包外）可访问，但对该包内的类则是私有的。然而，这个修饰符在Java 1.0版本发布时被移除了。

第46.6节：接口成员

```
公共接口 MyInterface {  
    公共 void foo();  
    int bar();  
  
    公共 String TEXT = "Hello";  
    int ANSWER = 42;  
  
    公共类 X {  
    }  
  
    类 Y {  
    }  
}
```

接口成员始终具有公共可见性，即使省略了public关键字。因此，foo()、bar()、TEXT、ANSWER、X和Y都具有公共可见性。然而，访问权限仍可能受包含接口的限制——因为MyInterface具有公共可见性，其成员可以从任何地方访问，但如果MyInterface具有包可见性，其成员将只能从同一包内访问。

Note that this modifier only works on members of a class, not on the class itself.

Section 46.5: Summary of Class Member Access Modifiers

Access Modifier	Visibility	Inheritance
Private	Class only	Can't be inherited
No modifier / Package	In package	Available if subclass in package
Protected	In package	Available in subclass
Public	Everywhere	Available in subclass

There was once a **private protected** (both keywords at once) modifier that could be applied to methods or variables to make them accessible from a subclass outside the package, but make them private to the classes in that package. However, this was [removed in Java 1.0's release](#).

Section 46.6: Interface members

```
public interface MyInterface {  
    public void foo();  
    int bar();  
  
    public String TEXT = "Hello";  
    int ANSWER = 42;  
  
    public class X {  
    }  
  
    class Y {  
    }  
}
```

Interface members always have public visibility, even if the **public** keyword is omitted. So both foo(), bar(), TEXT, ANSWER, X, and Y have public visibility. However, access may still be limited by the containing interface - since MyInterface has public visibility, its members may be accessed from anywhere, but if MyInterface had had package visibility, its members would only have been accessible from within the same package.

第47章：泛型

泛型是泛型编程的一种机制，扩展了Java的类型系统，使类型或方法能够操作各种类型的对象，同时提供编译时类型安全。特别是，Java集合框架支持泛型，以指定存储在集合实例中的对象类型。

第47.1节：创建泛型类

泛型使类、接口和方法能够接受其他类和接口作为类型参数。

此示例使用泛型类Param来接受单个类型参数T，参数用尖括号(<>)括起：

```
public class Param<T> {  
    private T value;  
  
    public T getValue() {  
        return value;  
    }  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
}
```

要实例化此类，需要用类型参数替代T。例如，Integer：

```
Param<Integer> integerParam = new Param<Integer>();
```

类型参数可以是任何引用类型，包括数组和其他泛型类型：

```
Param<String[]> stringArrayParam;  
Param<int[][]> int2dArrayParam;  
Param<Param<Object>> objectNestedParam;
```

在 Java SE 7 及更高版本中，类型参数可以用一对空的类型参数(<>)，称为菱形操作符(diamond)，来替代：

版本 ≥ Java SE 7
Param<Integer> integerParam = new Param<>();

与其他标识符不同，类型参数没有命名限制。但它们的名称通常是其用途的首字母大写。（这在官方 JavaDocs 中也是如此。）

例如，T 表示“类型”，E 表示“元素”，K/V 表示“键”/“值”。

扩展泛型类

```
public abstract class AbstractParam<T> {  
    private T value;  
  
    public T getValue() {  
        return value;  
    }  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
}
```

Chapter 47: Generics

[Generics](#) are a facility of generic programming that extend Java's type system to allow a type or method to operate on objects of various types while providing compile-time type safety. In particular, the Java collections framework supports generics to specify the type of objects stored in a collection instance.

Section 47.1: Creating a Generic Class

[Generics](#) enable classes, interfaces, and methods to take other classes and interfaces as type parameters.

This example uses generic class Param to take a single **type parameter** T, delimited by angle brackets(<>)：

```
public class Param<T> {  
    private T value;  
  
    public T getValue() {  
        return value;  
    }  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
}
```

To instantiate this class, provide a **type argument** in place of T. For example, [Integer](#):

```
Param<Integer> integerParam = new Param<Integer>();
```

The type argument can be any reference type, including arrays and other generic types:

```
Param<String[]> stringArrayParam;  
Param<int[][]> int2dArrayParam;  
Param<Param<Object>> objectNestedParam;
```

In Java SE 7 and later, the type argument can be replaced with an empty set of type arguments(<>) called the **diamond**:

Version ≥ Java SE 7
Param<Integer> integerParam = new Param<>();

Unlike other identifiers, type parameters have no naming constraints. However their names are commonly the first letter of their purpose in upper case. (This is true even throughout the official JavaDocs.) Examples include [T for "type"](#), [E for "element"](#) and [K/V for "key"/"value"](#).

Extending a generic class

```
public abstract class AbstractParam<T> {  
    private T value;  
  
    public T getValue() {  
        return value;  
    }  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
}
```

```
}
```

AbstractParam 是一个带有类型参数 T 的抽象类。扩展该类时，可以用写在 <> 中的类型参数替代该类型参数，或者保持类型参数不变。在下面的第一个和第二个示例中，String 和 Integer 替代了类型参数。第三个示例中，类型参数保持不变。第四个示例完全不使用泛型，因此类似于该类有一个 Object 参数。编译器会警告 AbstractParam 是原始类型，但代码仍然可以编译。

ObjectParam 类。第五个示例有2个类型参数（见下文“多个类型参数”），选择第二个参数作为传递给超类的类型参数。

```
public class Email extends AbstractParam<String> {  
    // ...  
}  
  
public class Age extends AbstractParam<Integer> {  
    // ...  
}  
  
public class Height<T> extends AbstractParam<T> {  
    // ...  
}  
  
public class ObjectParam extends AbstractParam {  
    // ...  
}  
  
public class MultiParam<T, E> extends AbstractParam<E> {  
    // ...  
}
```

以下是用法：

```
Email email = new Email();  
email.setValue("test@example.com");  
String retrievedEmail = email.getValue();  
  
Age age = new Age();  
age.setValue(25);  
Integer retrievedAge = age.getValue();  
int autounboxedAge = age.getValue();  
  
Height<Integer> heightInInt = new Height<>();  
heightInInt.setValue(125);  
  
Height<Float> heightInFloat = new Height<>();  
heightInFloat.setValue(120.3f);  
  
MultiParam<String, Double> multiParam = new MultiParam<>();  
multiParam.setValue(3.3);
```

请注意，在Email类中，T getValue()方法的行为就像它的签名是String getValue()，而 void setValue(T)方法的行为就像它被声明为void setValue(String)。

也可以使用带有空花括号 {} 的匿名内部类来实例化：

```
AbstractParam<Double> height = new AbstractParam<Double>(){ {} };  
height.setValue(198.6);
```

```
}
```

```
}
```

AbstractParam is an abstract class declared with a type parameter of T. When extending this class, that type parameter can be replaced by a type argument written inside <>, or the type parameter can remain unchanged. In the first and second examples below, String and Integer replace the type parameter. In the third example, the type parameter remains unchanged. The fourth example doesn't use generics at all, so it's similar to if the class had an Object parameter. The compiler will warn about AbstractParam being a raw type, but it will compile the ObjectParam class. The fifth example has 2 type parameters (see "multiple type parameters" below), choosing the second parameter as the type parameter passed to the superclass.

```
public class Email extends AbstractParam<String> {  
    // ...  
}  
  
public class Age extends AbstractParam<Integer> {  
    // ...  
}  
  
public class Height<T> extends AbstractParam<T> {  
    // ...  
}  
  
public class ObjectParam extends AbstractParam {  
    // ...  
}  
  
public class MultiParam<T, E> extends AbstractParam<E> {  
    // ...  
}
```

The following is the usage:

```
Email email = new Email();  
email.setValue("test@example.com");  
String retrievedEmail = email.getValue();  
  
Age age = new Age();  
age.setValue(25);  
Integer retrievedAge = age.getValue();  
int autounboxedAge = age.getValue();  
  
Height<Integer> heightInInt = new Height<>();  
heightInInt.setValue(125);  
  
Height<Float> heightInFloat = new Height<>();  
heightInFloat.setValue(120.3f);  
  
MultiParam<String, Double> multiParam = new MultiParam<>();  
multiParam.setValue(3.3);
```

Notice that in the Email class, the T getValue() method acts as if it had a signature of String getValue(), and the void setValue(T) method acts as if it was declared void setValue(String).

It is also possible to instantiate with anonymous inner class with an empty curly braces {}:

```
AbstractParam<Double> height = new AbstractParam<Double>(){ {} };  
height.setValue(198.6);
```

请注意，不允许在匿名内部类中使用菱形语法。

多个类型参数

Java 提供了在泛型类或接口中使用多个类型参数的能力。多个类型参数可以通过在尖括号中放置逗号分隔的类型列表来用于类或接口。示例：

```
public class MultiGenericParam<T, S> {
    private T firstParam;
    private S secondParam;

    public MultiGenericParam(T firstParam, S secondParam) {
        this.firstParam = firstParam;
        this.secondParam = secondParam;
    }

    public T getFirstParam() {
        return firstParam;
    }

    public void setFirstParam(T firstParam) {
        this.firstParam = firstParam;
    }

    public S getSecondParam() {
        return secondParam;
    }

    public void setSecondParam(S secondParam) {
        this.secondParam = secondParam;
    }
}
```

用法如下：

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
    "value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer, Double>(1,
    2.6);
```

第47.2节：在 `T`、`? super T` 和 `? extends T`

Java 泛型有界通配符的语法，表示未知类型的符号为 ?，如下：

- ? extends T 表示一个上界通配符。未知类型表示必须是 T 的子类型，或者是类型 T 本身。
- ? super T 表示一个下界通配符。未知类型表示必须是 T 的超类型，或者是类型 T 本身。

作为一个经验法则，你应该使用

- ? extends T 如果你只需要“读取”访问（“输入”）
- ? super T 如果你需要“写入”访问（“输出”）

Note that using the diamond with anonymous inner classes is not allowed.

Multiple type parameters

Java provides the ability to use more than one type parameter in a generic class or interface. Multiple type parameters can be used in a class or interface by placing a **comma-separated list** of types between the angle brackets. Example:

```
public class MultiGenericParam<T, S> {
    private T firstParam;
    private S secondParam;

    public MultiGenericParam(T firstParam, S secondParam) {
        this.firstParam = firstParam;
        this.secondParam = secondParam;
    }

    public T getFirstParam() {
        return firstParam;
    }

    public void setFirstParam(T firstParam) {
        this.firstParam = firstParam;
    }

    public S getSecondParam() {
        return secondParam;
    }

    public void setSecondParam(S secondParam) {
        this.secondParam = secondParam;
    }
}
```

The usage can be done as below:

```
MultiGenericParam<String, String> aParam = new MultiGenericParam<String, String>("value1",
    "value2");
MultiGenericParam<Integer, Double> dayOfWeekDegrees = new MultiGenericParam<Integer, Double>(1,
    2.6);
```

Section 47.2: Deciding between `T`、`? super T`，and `? extends T`

The syntax for Java generics bounded wildcards, representing the unknown type by ? is:

- ? extends T represents an upper bounded wildcard. The unknown type represents a type that must be a subtype of T, or type T itself.
- ? super T represents a lower bounded wildcard. The unknown type represents a type that must be a supertype of T, or type T itself.

As a rule of thumb, you should use

- ? extends T if you only need "read" access ("input")
- ? super T if you need "write" access ("output")

- T 如果你需要两者 ("修改")

使用 `extends` 或 `super` 通常更好，因为它使你的代码更灵活（例如：允许使用子类型和超类型），正如你下面将看到的。

```
class 鞋子 {}
class 苹果手机 {}
interface 水果 {}
class 苹果 implements 水果 {}
class 香蕉 implements 水果 {}
class 格兰尼史密斯 extends 苹果 {}

public class FruitHelper {

    public void eatAll(Collection<? extends Fruit> fruits) {}

    public void addApple(Collection<? super Apple> apples) {}
}
```

编译器现在能够检测某些错误的用法：

```
public class 泛型测试 {
    public static void main(String[] args){
        FruitHelper fruitHelper = new FruitHelper();
        List<Fruit> fruits = new ArrayList<Fruit>();
        fruits.add(new Apple()); // 允许, 因为苹果是水果
        fruits.add(new Banana()); // 允许, 因为香蕉是水果
        fruitHelper.addApple(fruits); // 允许, 因为"fruit super Apple"
        fruitHelper.eatAll(fruits); // 允许

        Collection<Banana> bananas = new ArrayList<>();
        bananas.add(new Banana()); // 允许
        //fruitHelper.addApple(bananas); // 编译错误: 只能包含香蕉!
        fruitHelper.eatAll(bananas); // 允许, 因为所有香蕉都是水果

        Collection<Apple> apples = new ArrayList<>();
        fruitHelper.addApple(apples); // 允许
        apples.add(new GrannySmith()); // 允许, 因为这是苹果
        fruitHelper.eatAll(apples); // 允许, 因为所有苹果都是水果。

        Collection<GrannySmith> grannySmithApples = new ArrayList<>();
        fruitHelper.addApple(grannySmithApples); // 编译错误: 不允许。
        // GrannySmith 不是 Apple 的超类型
        apples.add(new 格兰尼史密斯()); // 仍然允许, 格兰尼史密斯是苹果
        fruitHelper.eatAll(grannySmithApples); // 仍然允许, 格兰尼史密斯是水果

        Collection<Object> objects = new ArrayList<>();
        fruitHelper.addApple(objects); // 允许, 因为Object是Apple的超类
        objects.add(new 鞋子()); // 不是水果
        objects.add(new 苹果手机()); // 不是水果
        //fruitHelper.eatAll(objects); // 编译错误: 可能包含鞋子!
    }
}
```

选择正确的T、`? super T`或`? extends T`是必要的，以允许与子类型一起使用。编译器随后可以确保类型安全；如果正确使用它们，您不应需要进行强制转换（强制转换不安全，且可能导致编程错误）。

如果不理解，请记住PECS规则：

- T if you need both ("modify")

Using `extends` or `super` is usually *better* because it makes your code more flexible (as in: allowing the use of subtypes and supertypes), as you will see below.

```
class Shoe {}
class IPhone {}
interface Fruit {}
class Apple implements Fruit {}
class Banana implements Fruit {}
class GrannySmith extends Apple {}

public class FruitHelper {

    public void eatAll(Collection<? extends Fruit> fruits) {}

    public void addApple(Collection<? super Apple> apples) {}
}
```

The compiler will now be able to detect certain bad usage:

```
public class GenericsTest {
    public static void main(String[] args){
        FruitHelper fruitHelper = new FruitHelper();
        List<Fruit> fruits = new ArrayList<Fruit>();
        fruits.add(new Apple()); // Allowed, as Apple is a Fruit
        fruits.add(new Banana()); // Allowed, as Banana is a Fruit
        fruitHelper.addApple(fruits); // Allowed, as "Fruit super Apple"
        fruitHelper.eatAll(fruits); // Allowed

        Collection<Banana> bananas = new ArrayList<>();
        bananas.add(new Banana()); // Allowed
        //fruitHelper.addApple(bananas); // Compile error: may only contain Bananas!
        fruitHelper.eatAll(bananas); // Allowed, as all Bananas are Fruits

        Collection<Apple> apples = new ArrayList<>();
        fruitHelper.addApple(apples); // Allowed
        apples.add(new GrannySmith()); // Allowed, as this is an Apple
        fruitHelper.eatAll(apples); // Allowed, as all Apples are Fruits.

        Collection<GrannySmith> grannySmithApples = new ArrayList<>();
        fruitHelper.addApple(grannySmithApples); //Compile error: Not allowed.
        // GrannySmith is not a supertype of Apple
        apples.add(new GrannySmith()); //Still allowed, GrannySmith is an Apple
        fruitHelper.eatAll(grannySmithApples); //Still allowed, GrannySmith is a Fruit

        Collection<Object> objects = new ArrayList<>();
        fruitHelper.addApple(objects); // Allowed, as Object super Apple
        objects.add(new Shoe()); // Not a fruit
        objects.add(new IPhone()); // Not a fruit
        //fruitHelper.eatAll(objects); // Compile error: may contain a Shoe, too!
    }
}
```

Choosing the right T, `? super T` or `? extends T` is necessary to allow the use with subtypes. The compiler can then ensure type safety; you should not need to cast (which is not type safe, and may cause programming errors) if you use them properly.

If it is not easy to understand, please remember **PECS** rule:

生产者使用"Extends"，消费者使用"Super"。

(生产者只有写权限，消费者只有读权限)

第47.3节：菱形操作符

版本 \geq Java SE 7

Java 7引入了Diamond1，以消除泛型类实例化时的一些样板代码。在Java 7及以上版本中，您可以写：

```
List<String> list = new LinkedList<>();
```

在之前的版本中你必须写这个：

```
List<String> list = new LinkedList<String>();
```

一个限制是对于匿名类，你仍然必须在实例化时提供类型参数：

// 这将会编译通过：

```
Comparator<String> caseInsensitiveComparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};
```

// 但这将不会：

```
Comparator<String> caseInsensitiveComparator = new Comparator<>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};
```

版本 $>$ Java SE 8

虽然在 Java 7 和 8 中不支持在匿名内部类中使用菱形操作符，但它将在 Java 9 中作为新特性被包含进来。

脚注：

1 - 有些人称 `<>` 的用法为“菱形操作符”。这是不正确的。菱形并不作为操作符存在，也没有在 JLS 或（官方）Java 教程中被描述或列为操作符。实际上，`<>` 甚至不是一个独立的 Java 记号。它只是一个 `<` 记号后跟一个 `>` 记号，且在两者之间允许（虽然不推荐）有空白或注释。JLS 和教程一致将 `<>` 称为“菱形”，因此这是正确的称呼。

第47.4节：声明泛型方法

方法也可以有泛型类型参数。

```
public class Example {
```

Producer uses "Extends" and Consumer uses "Super".

(Producer has only write access, and Consumer has only read access)

Section 47.3: The Diamond

Version \geq Java SE 7

Java 7 introduced the [Diamond1](#) to remove some boiler-plate around generic class instantiation. With Java 7+ you can write:

```
List<String> list = new LinkedList<>();
```

Where you had to write in previous versions, this:

```
List<String> list = new LinkedList<String>();
```

One limitation is for Anonymous Classes, where you still must provide the type parameter in the instantiation:

// This will compile:

```
Comparator<String> caseInsensitiveComparator = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};
```

// But this will not:

```
Comparator<String> caseInsensitiveComparator = new Comparator<>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
};
```

Version $>$ Java SE 8

Although using the diamond with Anonymous Inner Classes is not supported in Java 7 and 8, [it will be included as a new feature in Java 9](#).

Footnote:

1 - Some people call the `<>` usage the "diamond operator". This is incorrect. The diamond does not behave as an operator, and is not described or listed anywhere in the JLS or the (official) Java Tutorials as an operator. Indeed, `<>` is not even a distinct Java token. Rather it is a `<` token followed by a `>` token, and it is legal (though bad style) to have whitespace or comments between the two. The JLS and the Tutorials consistently refer to `<>` as "the diamond", and that is therefore the correct term for it.

Section 47.4: Declaring a Generic Method

Methods can also have [generic](#) type parameters.

```
public class Example {
```

```
// 类型参数T的作用域限定在方法内
// 并且独立于其他方法的类型参数。
public <T> List<T> makeList(T t1, T t2) {
    List<T> 结果 = new ArrayList<T>();
    结果.add(t1);
    结果.add(t2);
    return result;
}

public void 使用() {
List<String> 字符串列表 = makeList("Jeff", "Atwood");
    List<Integer> 整数列表 = makeList(1, 2);
}
}
```

注意，我们不必向泛型方法传递实际的类型参数。编译器会根据目标类型（例如我们将结果赋值给的变量）或实际参数的类型为我们推断类型参数。它通常会推断出使调用类型正确的最具体的类型参数。

有时，虽然很少见，但可能需要用显式类型参数来覆盖这种类型推断：

```
void usage() {
consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }
```

在此示例中这是必要的，因为编译器无法“预先查看”以确定调用stream()后T需要的是Object，否则会根据makeList的参数推断为String。注意，Java语言不支持在显式提供类型参数时省略调用方法的类或对象（上述示例中的this）。

第47.5节：要求多个上界 ("extends A & B")

你可以要求泛型类型继承多个上界。

示例：我们想对数字列表进行排序，但Number没有实现Comparable接口。

```
public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
    Collections.sort( n );
}
```

在此示例中，T必须继承Number并实现Comparable<T>，这应适用于所有“正常”的内置数字实现，如Integer或BigDecimal，但不适用于更特殊的实现，如Striped64。

由于不允许多重继承，最多只能使用一个类作为边界，并且它必须是列表中的第一个。

例如，<T extends Comparable<T> & Number>是不允许的，因为Comparable是接口，而不是类。

第47.6节：在运行时获取满足泛型参数的类

许多未绑定的泛型参数，比如在静态方法中使用的，无法在运行时恢复（参见Other Threads on Erasure）。然而，有一种常用策略用于在运行时访问满足类泛型参数的类型。这允许依赖于类型访问的泛型代码无需通过每次调用传递类型信息。

```
// The type parameter T is scoped to the method
// and is independent of type parameters of other methods.
public <T> List<T> makeList(T t1, T t2) {
    List<T> result = new ArrayList<T>();
    result.add(t1);
    result.add(t2);
    return result;
}

public void usage() {
    List<String> listString = makeList("Jeff", "Atwood");
    List<Integer> listInteger = makeList(1, 2);
}
}
```

Notice that we don't have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the target type (e.g. the variable we assign the result to), or on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

Sometimes, albeit rarely, it can be necessary to override this type inference with explicit type arguments:

```
void usage() {
    consumeObjects(this.<Object>makeList("Jeff", "Atwood").stream());
}

void consumeObjects(Stream<Object> stream) { ... }
```

It's necessary in this example because the compiler can't "look ahead" to see that `Object` is desired for T after calling `stream()` and it would otherwise infer `String` based on the `makeList` arguments. Note that the Java language doesn't support omitting the class or object on which the method is called (`this` in the above example) when type arguments are explicitly provided.

Section 47.5: Requiring multiple upper bounds ("extends A & B")

You can require a generic type to extend multiple upper bounds.

Example: we want to sort a list of numbers but `Number` doesn't implement `Comparable`.

```
public <T extends Number & Comparable<T>> void sortNumbers( List<T> n ) {
    Collections.sort( n );
}
```

In this example T must extend `Number` and implement `Comparable<T>` which should fit all "normal" built-in number implementations like `Integer` or `BigDecimal` but doesn't fit the more exotic ones like `Striped64`.

Since multiple inheritance is not allowed, you can use at most one class as a bound and it must be the first listed. For example, `<T extends Comparable<T> & Number>` is not allowed because `Comparable` is an interface, and not a class.

Section 47.6: Obtain class that satisfies generic parameter at runtime

Many unbound generic parameters, like those used in a static method, cannot be recovered at runtime (see Other Threads on Erasure). However there is a common strategy employed for accessing the type satisfying a generic parameter on a class at runtime. This allows for generic code that depends on access to type without having to

线程类型信息。

背景

可以通过创建匿名内部类来检查类的泛型参数化。该类将捕获类型信息。通常，这种机制被称为超类型令牌（super type tokens），详见Neal Gafter的博客文章。

实现方式

Java中三种常见的实现方式是：

- [Guava的TypeToken](#)
- [Spring的ParameterizedTypeReference](#)
- [Jackson的TypeReference](#)

示例用法

```
public class DataService<MODEL_TYPE> {  
    private final DataDao dataDao = new DataDao();  
    private final Class<MODEL_TYPE> type = (Class<MODEL_TYPE>) new TypeToken<MODEL_TYPE>()  
        (getClass()){}.getRawType();  
  
    public List<MODEL_TYPE> getAll() {  
        return dataDao.getAllOfType(type);  
    }  
  
    // 子类明确绑定参数化为 User  
    // 对于该类的所有实例，这样的信息可以  
    // 在运行时恢复  
    public class UserService extends DataService<User> {}  
  
    public class Main {  
        public static void main(String[] args) {  
            UserService service = new UserService();  
            List<User> users = service.getAll();  
        }  
    }  
}
```

第47.7节：泛型类和接口的优点

使用泛型的代码相比非泛型代码有许多优点。以下是主要优点

编译时更强的类型检查

Java 编译器对泛型代码应用强类型检查，如果代码违反类型安全则发出错误。修复编译时错误比修复运行时错误更容易，后者可能难以发现。

消除类型转换

以下代码片段在不使用泛型的情况下需要进行类型转换：

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

thread type information through every call.

Background

Generic parameterization on a class can be inspected by creating an anonymous inner class. This class will capture the type information. In general this mechanism is referred to as **super type tokens**, which are detailed in [Neal Gafter's blog post](#).

Implementations

Three common implementations in Java are:

- [Guava's TypeToken](#)
- [Spring's ParameterizedTypeReference](#)
- [Jackson's TypeReference](#)

Example usage

```
public class DataService<MODEL_TYPE> {  
    private final DataDao dataDao = new DataDao();  
    private final Class<MODEL_TYPE> type = (Class<MODEL_TYPE>) new TypeToken<MODEL_TYPE>()  
        (getClass()){}.getRawType();  
  
    public List<MODEL_TYPE> getAll() {  
        return dataDao.getAllOfType(type);  
    }  
}  
  
// the subclass definitively binds the parameterization to User  
// for all instances of this class, so that information can be  
// recovered at runtime  
public class UserService extends DataService<User> {}  
  
public class Main {  
    public static void main(String[] args) {  
        UserService service = new UserService();  
        List<User> users = service.getAll();  
    }  
}
```

Section 47.7: Benefits of Generic class and interface

Code that uses generics has many benefits over non-generic code. Below are the main benefits

Stronger type checks at compile time

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

Elimination of casts

The following code snippet without generics requires casting:

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

当改写为使用泛型时，代码不再需要类型转换：

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0); // 无需类型转换
```

使程序员能够实现泛型算法

通过使用泛型，程序员可以实现适用于不同类型集合的泛型算法，这些算法可以被定制，且类型安全且更易于阅读。

第47.8节：实例化泛型类型

由于类型擦除，以下代码将无法工作：

```
public <T> void genericMethod() {
    T t = new T(); // 无法实例化类型 T。
}
```

类型T被擦除。由于在运行时，JVM不知道T最初是什么类型，因此它不知道调用哪个构造函数。

解决方法

1. 调用genericMethod时传递T的类：

```
public <T> void genericMethod(Class<T> cls) {
    try {
        T t = cls.newInstance();
    } catch (InstantiationException | IllegalAccessException e) {
        System.err.println("无法实例化: " + cls.getName());
    }
}
genericMethod(String.class);
```

这会抛出异常，因为无法确定传入的类是否有可访问的默认构造函数。

版本 ≥ Java SE 8

2. 传递T构造函数的引用：

```
public <T> void genericMethod(Supplier<T> cons) {
    T t = cons.get();
}
genericMethod(String::new);
```

第47.9节：创建有界泛型类

您可以通过在类定义中对类型进行限定，来限制泛型类中可用的有效类型。给定以下简单的类型层次结构：

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0); // no cast
```

Enabling programmers to implement generic algorithms

By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Section 47.8: Instantiating a generic type

Due to type erasure the following will not work:

```
public <T> void genericMethod() {
    T t = new T(); // Can not instantiate the type T.
}
```

The type T is erased. Since, at runtime, the JVM does not know what T originally was, it does not know which constructor to call.

Workarounds

1. Passing T's class when calling genericMethod:

```
public <T> void genericMethod(Class<T> cls) {
    try {
        T t = cls.newInstance();
    } catch (InstantiationException | IllegalAccessException e) {
        System.err.println("Could not instantiate: " + cls.getName());
    }
}
genericMethod(String.class);
```

Which throws exceptions, since there is no way to know if the passed class has an accessible default constructor.

Version ≥ Java SE 8

2. Passing a reference to T's constructor:

```
public <T> void genericMethod(Supplier<T> cons) {
    T t = cons.get();
}
genericMethod(String::new);
```

Section 47.9: Creating a Bounded Generic Class

You can restrict the valid types used in a **generic class** by bounding that type in the class definition. Given the following simple type hierarchy:

```

public abstract class Animal {
    public abstract String getSound();
}

public class Cat extends Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Dog extends Animal {
    public String getSound() {
        return "Woof";
    }
}

```

没有有界泛型，我们无法创建一个既是泛型又能知道每个元素都是动物的容器类：

```

public class AnimalContainer<T> {

    private Collection<T> col;

    public AnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // 非法，类型 T 没有 makeSound()
            // 这里它被当作 java.lang.Object 使用
            System.out.println(t.makeSound());
        }
    }
}

```

在类定义中使用泛型边界后，现在这是可能的。

```

public class 有界动物容器<T extends 动物> { // 注意这里的边界。

    private Collection<T> col;

    public 有界动物容器() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // 现在可行，因为 T 继承了动物
            System.out.println(t.makeSound());
        }
    }
}

```

```

public abstract class Animal {
    public abstract String getSound();
}

public class Cat extends Animal {
    public String getSound() {
        return "Meow";
    }
}

public class Dog extends Animal {
    public String getSound() {
        return "Woof";
    }
}

```

Without **bounded generics**, we cannot make a container class that is both generic and knows that each element is an animal:

```

public class AnimalContainer<T> {

    private Collection<T> col;

    public AnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Illegal, type T doesn't have makeSound()
            // it is used as an java.lang.Object here
            System.out.println(t.makeSound());
        }
    }
}

```

With generic bound in class definition, this is now possible.

```

public class BoundedAnimalContainer<T extends Animal> { // Note bound here.

    private Collection<T> col;

    public BoundedAnimalContainer() {
        col = new ArrayList<T>();
    }

    public void add(T t) {
        col.add(t);
    }

    public void printAllSounds() {
        for (T t : col) {
            // Now works because T is extending Animal
            System.out.println(t.makeSound());
        }
    }
}

```

```
}
```

这也限制了泛型类型的有效实例化：

```
// 合法
动物容器<猫> a = new 动物容器<猫>();

// 合法
AnimalContainer<String> a = new AnimalContainer<String>();

// 合法, 因为 Cat 继承自 Animal
BoundedAnimalContainer<Cat> b = new BoundedAnimalContainer<Cat>();

// 非法, 因为 String 没有继承 Animal
BoundedAnimalContainer<String> b = new BoundedAnimalContainer<String>();
```

第47.10节：在自身声明中引用声明的泛型类型

如何在泛型类型自身声明的方法中使用（可能进一步继承的）泛型类型的实例？这是深入研究泛型时会遇到的问题之一，但仍然相当常见。

假设我们有一个`DataSeries<T>`类型（此处为接口），它定义了包含类型为`T`的值的通用数据序列。当我们想对其执行大量操作时，直接使用此类型会很麻烦，例如

双精度值，因此我们定义了`DoubleSeries extends DataSeries<Double>`。现在假设，原始的`DataSeries<T>`类型有一个方法`add(values)`，该方法添加另一个相同长度的序列并返回一个新的序列。我们如何在派生类中强制`values`的类型和返回值的类型为`DoubleSeries`，而不是`DataSeries<Double>`？

该问题可以通过添加一个泛型类型参数来解决，该参数引用并扩展正在声明的类型（这里应用于接口，但对类同样适用）：

```
public interface DataSeries<T, DS extends DataSeries<T, DS>> {
    DS add(DS values);
    List<T> 数据();
}
```

这里`T`表示序列所持有的数据类型，例如`Double`，`DS`表示序列本身。通过用相应的派生类型替换上述参数，现在可以轻松实现继承类型，从而生成基于`Double`的具体定义，形式如下：

```
public interface DoubleSeries extends DataSeries<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> 数据) {
        return new DoubleSeriesImpl(数据);
    }
}
```

此时即使是IDE也会用正确的类型实现上述接口，稍作内容填充后可能如下所示：

```
class DoubleSeriesImpl implements DoubleSeries {
    private final List<Double> 数据;

    DoubleSeriesImpl(Collection<Double> 数据) {
        this.数据 = new ArrayList<>(数据);
    }
}
```

```
}
```

This also restricts the valid instantiations of the generic type:

```
// Legal
AnimalContainer<Cat> a = new AnimalContainer<Cat>();

// Legal
AnimalContainer<String> a = new AnimalContainer<String>();

// Legal because Cat extends Animal
BoundedAnimalContainer<Cat> b = new BoundedAnimalContainer<Cat>();

// Illegal because String doesn't extend Animal
BoundedAnimalContainer<String> b = new BoundedAnimalContainer<String>();
```

Section 47.10: Referring to the declared generic type within its own declaration

How do you go about using an instance of a (possibly further) inherited generic type within a method declaration in the generic type itself being declared? This is one of the problems you will face when you dig a bit deeper into generics, but still a fairly common one.

Assume we have a `DataSeries<T>` type (interface here), which defines a generic data series containing values of type `T`. It is cumbersome to work with this type directly when we want to perform a lot of operations with e.g. double values, so we define `DoubleSeries extends DataSeries<Double>`. Now assume, the original `DataSeries<T>` type has a method `add(values)` which adds another series of the same length and returns a new one. How do we enforce the type of `values` and the type of the return to be `DoubleSeries` rather than `DataSeries<Double>` in our derived class?

The problem can be solved by adding a generic type parameter referring back to and extending the type being declared (applied to an interface here, but the same stands for classes):

```
public interface DataSeries<T, DS extends DataSeries<T, DS>> {
    DS add(DS values);
    List<T> data();
}
```

Here `T` represents the data type the series holds, e.g. `Double` and `DS` the series itself. An inherited type (or types) can now be easily implemented by substituting the above mentioned parameter by a corresponding derived type, thus, yielding a concrete `Double`-based definition of the form:

```
public interface DoubleSeries extends DataSeries<Double, DoubleSeries> {
    static DoubleSeries instance(Collection<Double> data) {
        return new DoubleSeriesImpl(data);
    }
}
```

At this moment even an IDE will implement the above interface with correct types in place, which, after a bit of content filling may look like this:

```
class DoubleSeriesImpl implements DoubleSeries {
    private final List<Double> data;

    DoubleSeriesImpl(Collection<Double> data) {
        this.data = new ArrayList<>(data);
    }
}
```

```

}
@Override
public DoubleSeries add(DoubleSeries 值) {
    List<Double> 传入 = 值 != null ? 值.数据() : null;
    if (传入 == null || 传入.大小() != 数据.大小()) {
        throw new IllegalArgumentException("bad series");
    }
List<Double> 新数据 = new ArrayList<>(数据.大小());
    for (int i = 0; i < 数据.大小(); i++) {
newdata.add(this.data.get(i) + incoming.get(i)); // 注意自动装箱
    }
    return DoubleSeries.instance(newdata);
}

@Override
public List<Double> data() {
    return Collections.unmodifiableList(data);
}
}

```

如你所见，add方法声明为DoubleSeries add(DoubleSeries values)，编译器对此没有问题。

如果需要，该模式可以进一步嵌套。

第47.11节：将泛型参数绑定到多个类型

泛型参数也可以使用T extends Type1 & Type2 & ...语法绑定到多个类型。

假设你想创建一个泛型类型同时实现Flushable和Closeable的类，你可以这样写

```

class ExampleClass<T extends Flushable & Closeable> {
}

```

现在，ExampleClass只接受同时实现了Flushable和

Closeable的类型作为泛型参数。

ExampleClass<BufferedWriter> arg1; //之所以可行，是因为 BufferedWriter 同时实现了 Flushable 和 Closeable

*ExampleClass<Console> arg4; //不起作用，因为 Console 只实现了 Flushable
ExampleClass<ZipFile> arg5; //不起作用，因为 ZipFile 只实现了 Closeable*

*ExampleClass<Flushable> arg2; //不起作用，因为未满足 Closeable 约束。
ExampleClass<Closeable> arg3; //无法工作，因为未满足 Flushable 约束。*

类方法可以选择推断泛型类型参数为Closeable或Flushable。

```

class ExampleClass<T extends Flushable & Closeable> {
    /* 根据需要将其分配给有效类型。 */
    public void test (T param) {
        Flushable arg1 = param; //可行
        Closeable arg2 = param; //也可行。
    }

    /* 你甚至可以直接调用任何有效类型的方法。 */
}

```

```

}
@Override
public DoubleSeries add(DoubleSeries values) {
    List<Double> incoming = values != null ? values.data() : null;
    if (incoming == null || incoming.size() != data.size()) {
        throw new IllegalArgumentException("bad series");
    }
List<Double> newdata = new ArrayList<>(data.size());
    for (int i = 0; i < data.size(); i++) {
        newdata.add(this.data.get(i) + incoming.get(i)); // beware autoboxing
    }
    return DoubleSeries.instance(newdata);
}

@Override
public List<Double> data() {
    return Collections.unmodifiableList(data);
}
}

```

As you can see the add method is declared as DoubleSeries add(DoubleSeries values) and the compiler is happy.

The pattern can be further nested if required.

Section 47.11: Binding generic parameter to more than 1 type

Generic parameters can also be bound to more than one type using the T extends Type1 & Type2 & ... syntax.

Let's say you want to create a class whose Generic type should implement both Flushable and Closeable, you can write

```

class ExampleClass<T extends Flushable & Closeable> {
}

```

Now, the ExampleClass only accepts as generic parameters, types which implement both Flushable **and** Closeable.

ExampleClass<BufferedWriter> arg1; // Works because BufferedWriter implements both Flushable and Closeable

*ExampleClass<Console> arg4; // Does NOT work because Console only implements Flushable
ExampleClass<ZipFile> arg5; // Does NOT work because ZipFile only implements Closeable*

*ExampleClass<Flushable> arg2; // Does NOT work because Closeable bound is not satisfied.
ExampleClass<Closeable> arg3; // Does NOT work because Flushable bound is not satisfied.*

The class methods can choose to infer generic type arguments as either Closeable or Flushable.

```

class ExampleClass<T extends Flushable & Closeable> {
    /* Assign it to a valid type as you want. */
    public void test (T param) {
        Flushable arg1 = param; // Works
        Closeable arg2 = param; // Works too.
    }

    /* You can even invoke the methods of any valid type directly. */
}

```

```

public void test2 (T param) {
    param.flush(); // 在 T 上调用 Flushable 的方法，运行正常。
    param.close(); // 在 T 上调用 Closeable 的方法，也能正常工作。
}

```

注意：

您不能使用OR (|) 子句将泛型参数绑定到任一类型。仅支持AND (&) 子句。泛型类型只能继承一个类和多个接口。类必须放在列表的开头。

第47.12节：使用泛型实现自动类型转换

使用泛型，可以返回调用者期望的任何类型：

```

private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}

```

该方法会带有编译警告。实际上代码比看起来更安全，因为Java运行时会在使用时进行类型转换：

```
Bar bar = foo.get("bar");
```

当使用泛型类型时，安全性会降低：

```
List<Bar> bars = foo.get("bars");
```

这里，当返回类型是任何类型的List时，类型转换会成功（例如返回List<String>不会触发ClassCastException；但在从列表中取出元素时最终会出现异常）。

为了解决这个问题，你可以创建一个使用类型化键的API：

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

以及这个put()方法：

```
public <T> T put(Key<T> key, T value);
```

通过这种方法，你不能将错误的类型放入映射中，因此结果将始终是正确的（除非你不小心创建了两个名称相同但类型不同的键）。

相关：

- [类型安全的映射](#)

第47.13节：在泛型中使用instanceof

使用泛型来定义instanceof中的类型

考虑以下用形式参数<T>声明的泛型类Example：

```
class Example<T> {
```

```

public void test2 (T param) {
    param.flush(); // Method of Flushable called on T and works fine.
    param.close(); // Method of Closeable called on T and works fine too.
}

```

Note:

You cannot bind the generic parameter to either of the type using OR (|) clause. Only the AND (&) clause is supported. Generic type can extends only one class and many interfaces. Class must be placed at the beginning of the list.

Section 47.12: Using Generics to auto-cast

With generics, it's possible to return whatever the caller expects:

```

private Map<String, Object> data;
public <T> T get(String key) {
    return (T) data.get(key);
}

```

The method will compile with a warning. The code is actually more safe than it looks because the Java runtime will do a cast when you use it:

```
Bar bar = foo.get("bar");
```

It's less safe when you use generic types:

```
List<Bar> bars = foo.get("bars");
```

Here, the cast will work when the returned type is any kind of List (i.e. returning List<String> would not trigger a ClassCastException; you'd eventually get it when taking elements out of the list).

To work around this problem, you can create an API which uses typed keys:

```
public final static Key<List<Bar>> BARS = new Key<>("BARS");
```

along with this put() method:

```
public <T> T put(Key<T> key, T value);
```

With this approach, you can't put the wrong type into the map, so the result will always be correct (unless you accidentally create two keys with the same name but different types).

Related:

- [Type-safe Map](#)

Section 47.13: Use of instanceof with Generics

Using generics to define the type in instanceof

Consider the following generic class Example declared with the formal parameter <T>:

```
class Example<T> {
```

```

public boolean isTypeAString(String s) {
    return s instanceof T; // 编译错误, 不能在这里将T用作类类型
}

```

这将始终导致编译错误，因为编译器在将Java源代码编译成Java字节码时，会应用一种称为类型擦除的过程，该过程将所有泛型代码转换为非泛型代码，使得在运行时无法区分T类型。与instanceof一起使用的类型必须是可重构的，这意味着所有关于该类型的信息必须在运行时可用，而这通常不适用于泛型类型。

以下类表示两种不同的Example类，Example<String>和Example<Number>，在泛型被类型擦除剥离后看起来的样子：

```

class Example { // 形式参数已消失
    public boolean isTypeAString(String s) {
        return s instanceof Object; // <String> 和 <Number> 现在都是 Object
    }
}

```

由于类型已消失，JVM 无法知道哪个类型是T。

前述规则的例外

你总是可以使用无限制通配符 (?) 来指定instanceof中的类型，如下所示：

```

public boolean isAList(Object obj) {
    return obj instanceof List<?>;
}

```

这对于判断实例obj是否为List非常有用：

```

System.out.println(isAList("foo")); // 输出 false
System.out.println(isAList(new ArrayList<String>())); // 输出 true
System.out.println(isAList(new ArrayList<Float>())); // 输出 true

```

事实上，无限制通配符被视为一种可重构类型。

使用带有 instanceof 的通用实例

另一方面，使用 T 类型的实例 t 与 instanceof 是合法的，如以下示例所示：

```

class Example<T> {
    public boolean isTypeAString(T t) {
        return t instanceof String; // 这次没有编译错误
    }
}

```

因为在类型擦除之后，类将变成如下形式：

```

class Example { // 形式参数消失了
    public boolean isTypeAString(Object t) {
        return t instanceof String; // 这次没有编译错误
    }
}

```

```

public boolean isTypeAString(String s) {
    return s instanceof T; // Compilation error, cannot use T as class type here
}

```

This will always give a Compilation error because as soon as the compiler compiles the Java source into Java bytecode it applies a process known as *type erasure*, which converts all generic code into non-generic code, making impossible to distinguish among T types at runtime. The type used with `instanceof` has to be [reifiable](#), which means that all information about the type has to be available at runtime, and this is usually not the case for generic types.

The following class represents what two different classes of Example, Example<String> and Example<Number>, look like after generics has stripped off by *type erasure*:

```

class Example { // formal parameter is gone
    public boolean isTypeAString(String s) {
        return s instanceof Object; // Both <String> and <Number> are now Object
    }
}

```

Since types are gone, it's not possible for the JVM to know which type is T.

Exception to the previous rule

You can always use *unbounded wildcard* (?) for specifying a type in the `instanceof` as follows:

```

public boolean isAList(Object obj) {
    return obj instanceof List<?>;
}

```

This can be useful to evaluate whether an instance obj is a List or not:

```

System.out.println(isAList("foo")); // prints false
System.out.println(isAList(new ArrayList<String>())); // prints true
System.out.println(isAList(new ArrayList<Float>())); // prints true

```

In fact, unbounded wildcard is considered a reifiable type.

Using a generic instance with instanceof

The other side of the coin is that using an instance t of T with `instanceof` is legal, as shown in the following example:

```

class Example<T> {
    public boolean isTypeAString(T t) {
        return t instanceof String; // No compilation error this time
    }
}

```

because after the type erasure the class will look like the following:

```

class Example { // formal parameter is gone
    public boolean isTypeAString(Object t) {
        return t instanceof String; // No compilation error this time
    }
}

```

因为即使发生了类型擦除，JVM 现在也能区分内存中的不同类型，即使它们使用相同的引用类型（Object），如下代码片段所示：

```
Object obj1 = new String("foo"); // 引用类型 Object, 对象类型 String
Object obj2 = new Integer(11); // 引用类型 Object, 对象类型 Integer
System.out.println(obj1 instanceof String); // true
System.out.println(obj2 instanceof String); // false, 它是 Integer, 不是 String
```

第 47.14 节：实现泛型接口（或继承泛型类）的不同方式

假设声明了以下泛型接口：

```
public interface MyGenericInterface<T> {
    public void foo(T t);
}
```

以下列出了实现它的可能方法。

使用特定类型的非泛型类实现

选择一个特定类型来替换MyGenericClass的形式类型参数<T>，并实现它，如以下示例所示：

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // 类型T已被替换为String
}
```

该类仅处理String，这意味着使用不同参数（例如 Integer、Object等）的MyGenericInterface将无法编译，如以下代码片段所示：

```
NonGenericClass myClass = new NonGenericClass();
myClass.foo("foo_string"); // 正确, 合法
myClass.foo(11); // 不正确, 无法编译
myClass.foo(new Object()); // 不正确, 无法编译
```

泛型类实现

声明另一个带有形式类型参数<T>的泛型接口，该接口实现MyGenericInterface，如下所示：

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
    public void foo(T t) { } // 类型 T 仍然相同
    // 其他方法...
}
```

请注意，可能使用了不同的形式类型参数，如下所示：

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // 等同于之前的声明
    public void foo(U t) { }
    // 其他方法...
}
```

原始类型类的实现

Since, even if the type erasure happen anyway, now the JVM can distinguish among different types in memory, even if they use the same reference type (Object), as the following snippet shows:

```
Object obj1 = new String("foo"); // reference type Object, object type String
Object obj2 = new Integer(11); // reference type Object, object type Integer
System.out.println(obj1 instanceof String); // true
System.out.println(obj2 instanceof String); // false, it's an Integer, not a String
```

Section 47.14: Different ways for implementing a Generic Interface (or extending a Generic Class)

Suppose the following generic interface has been declared:

```
public interface MyGenericInterface<T> {
    public void foo(T t);
}
```

Below are listed the possible ways to implement it.

Non-generic class implementation with a specific type

Choose a specific type to replace the formal type parameter <T> of MyGenericClass and implement it, as the following example does:

```
public class NonGenericClass implements MyGenericInterface<String> {
    public void foo(String t) { } // type T has been replaced by String
}
```

This class only deals with String, and this means that using MyGenericInterface with different parameters (e.g. Integer, Object etc.) won't compile, as the following snippet shows:

```
NonGenericClass myClass = new NonGenericClass();
myClass.foo("foo_string"); // OK, legal
myClass.foo(11); // NOT OK, does not compile
myClass.foo(new Object()); // NOT OK, does not compile
```

Generic class implementation

Declare another generic interface with the formal type parameter <T> which implements MyGenericInterface, as follows:

```
public class MyGenericSubclass<T> implements MyGenericInterface<T> {
    public void foo(T t) { } // type T is still the same
    // other methods...
}
```

Note that a different formal type parameter may have been used, as follows:

```
public class MyGenericSubclass<U> implements MyGenericInterface<U> { // equivalent to the previous declaration
    public void foo(U t) { }
    // other methods...
}
```

Raw type class implementation

声明一个非泛型类，实现MyGenericInterface作为原始类型（完全不使用泛型），如下所示：

```
public class MyGenericSubclass implements MyGenericInterface {  
    public void foo(Object t) { } // 类型T已被Object替代  
    // 其他可能的方法  
}
```

这种方式不推荐，因为在运行时不完全安全，原因是它混淆了子类的原始类型和接口的泛型，同时也容易引起混淆。现代Java编译器会对这种实现方式发出警告，尽管如此，为了兼容旧版JVM（1.4或更早版本），代码仍然可以编译。

上述所有方式在使用泛型类作为超类型而非泛型接口时也同样适用。

Declare a non-generic class which implements MyGenericInterface as a *raw type* (not using generic at all), as follows:

```
public class MyGenericSubclass implements MyGenericInterface {  
    public void foo(Object t) { } // type T has been replaced by Object  
    // other possible methods  
}
```

This way is **not** recommended, since it is not 100% safe at runtime because it mixes up *raw type* (of the subclass) with *generics* (of the interface) and it is also confusing. Modern Java compilers will raise a warning with this kind of implementation, nevertheless the code - for compatibility reasons with older JVM (1.4 or earlier) - will compile.

All the ways listed above are also allowed when using a generic class as a supertype instead of a generic interface.

第48章：类与对象

对象具有状态和行为。例如：一只狗有状态——颜色、名字、品种，以及行为-摇尾巴、吠叫、吃东西。对象是类的一个实例。

类-类可以定义为描述其类型对象所支持的行为/状态的模板/蓝图。

第48.1节：方法重载

有时相同的功能需要针对不同类型的输入编写。这时，可以使用相同的方法名但参数集不同。每个不同的参数集称为方法签名。

如示例所示，一个方法可以有多个签名。

```
public class Displayer {  
  
    public void displayName(String firstName) {  
        System.out.println("Name is: " + firstName);  
    }  
  
    public void displayName(String firstName, String lastName) {  
        System.out.println("Name is: " + firstName + " " + lastName);  
    }  
  
    public static void main(String[] args) {  
        Displayer displayer = new Displayer();  
        displayer.displayName("Ram"); //打印 "Name is: Ram"  
        displayer.displayName("Jon", "Skeet"); //打印 "Name is: Jon Skeet"  
    }  
}
```

优点是相同的功能可以通过不同数量的输入调用。在根据传入的输入调用方法时（在本例中是一个字符串值或两个字符串值），会执行相应的方法。

方法可以重载：

1. 基于传入参数的数量。

示例：method(String s) 和 method(String s1, String s2)。

2. 基于参数的顺序。

示例：method(int i, float f) 和 method(float f, int i)。

注意：方法不能仅通过改变返回类型来重载 (`int method()` 被视为与 `String method()` 相同，尝试这样做会抛出 `RuntimeException`)。如果更改返回类型，必须同时更改参数才能实现重载。

Chapter 48: Classes and Objects

Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

Class – A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Section 48.1: Overloading Methods

Sometimes the same functionality has to be written for different kinds of inputs. At that time, one can use the same method name with a different set of parameters. Each different set of parameters is known as a method signature. As seen per the example, a single method can have multiple signatures.

```
public class Displayer {  
  
    public void displayName(String firstName) {  
        System.out.println("Name is: " + firstName);  
    }  
  
    public void displayName(String firstName, String lastName) {  
        System.out.println("Name is: " + firstName + " " + lastName);  
    }  
  
    public static void main(String[] args) {  
        Displayer displayer = new Displayer();  
        displayer.displayName("Ram"); //prints "Name is: Ram"  
        displayer.displayName("Jon", "Skeet"); //prints "Name is: Jon Skeet"  
    }  
}
```

The advantage is that the same functionality is called with two different numbers of inputs. While invoking the method according to the input we are passing, (In this case either one string value or two string values) the corresponding method is executed.

Methods can be overloaded:

1. Based on the **number of parameters** passed.

Example: `method(String s)` and `method(String s1, String s2)`.

2. Based on the **order of parameters**.

Example: `method(int i, float f)` and `method(float f, int i)`.

Note: Methods cannot be overloaded by changing just the return type (`int method()` is considered the same as `String method()` and will throw a `RuntimeException` if attempted). If you change the return type you must also change the parameters in order to overload.

第48.2节：解释什么是方法重载和重写

方法重写和重载是Java支持的两种多态形式。

方法重载

方法重载（也称为静态多态）是指在同一个类中可以有两个（或更多）同名的方法（函数）。是的，就是这么简单。

```
public class Shape{
    // 它可以是圆形、矩形或正方形
    private String type;

    // 计算矩形面积
    public Double area(Long length, Long breadth){
        return (Double) length * breadth;
    }

    // 计算圆形面积
    public Double area(Long radius){
        return (Double) 3.14 * r * r;
    }
}
```

这样用户可以根据形状的类型调用相同的方法来计算面积。

但真正的问题是，Java 编译器如何区分执行哪个方法体？

Java 已经明确指出，尽管方法名（在我们的例子中是area()）可以相同，但方法所接受的参数必须不同。

重载方法必须有不同的参数列表（数量和类型）。

话虽如此，我们不能像这样添加另一个计算正方形面积的方法：`public Double area(Long side)`，因为在这种情况下，它会与圆的area方法冲突，并且会导致Java编译器产生歧义。

谢天谢地，编写重载方法时有一些放宽的规定，比如

可以有不同的返回类型。

可以有不同的访问修饰符。

可以抛出不同的异常。

为什么这被称为静态多态？

这是因为调用哪个重载方法是在编译时决定的，基于实际的参数数量和参数的编译时类型。

Section 48.2: Explaining what is method overloading and overriding

Method Overriding and Overloading are two forms of polymorphism supported by Java.

Method Overloading

Method overloading (also known as static Polymorphism) is a way you can have two (or more) methods (functions) with same name in a single class. Yes its as simple as that.

```
public class Shape{
    //It could be a circle or rectangle or square
    private String type;

    //To calculate area of rectangle
    public Double area(Long length, Long breadth){
        return (Double) length * breadth;
    }

    //To calculate area of a circle
    public Double area(Long radius){
        return (Double) 3.14 * r * r;
    }
}
```

This way user can call the same method for area depending on the type of shape it has.

But the real question now is, how will java compiler will distinguish which method body is to be executed?

Well Java have made it clear that even though the **method names** (area() in our case) **can be same but the arguments method is taking should be different**.

Overloaded methods must have different arguments list (quantity and types).

That being said we cannot add another method to calculate area of a square like this : `public Double area(Long side)` because in this case, it will conflict with area method of circle and will cause **ambiguity** for java compiler.

Thank god, there are some relaxations while writing overloaded methods like

May have different return types.

May have different access modifiers.

May throw different exceptions.

Why is this called static polymorphism?

Well that's because which overloaded methods is to be invoked is decided at compile time, based on the actual number of arguments and the compile-time types of the arguments.

使用方法重载的一个常见原因是它提供的代码简洁性。例如，记住String.valueOf()方法，它几乎接受任何类型的参数？其背后写的可能是类似这样的代码：

```
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(char[] data)
static String valueOf(char[] data, int offset, int count)
static String valueOf(double d)
static String valueOf(float f)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(Object obj)
```

方法重写

嗯，方法重写（是的，你猜对了，它也被称为动态多态）是一个更有趣且复杂的话题。

在方法重写中，我们覆盖父类提供的方法体。明白了吗？不明白？让我们通过一个例子来讲解。

```
public abstract class Shape{
    public abstract Double area(){
        return 0.0;
    }
}
```

所以我们有一个名为Shape的类，它有一个名为area的方法，可能会返回该形状的面积。

假设现在我们有两个类，分别叫Circle和Rectangle。

```
public class Circle extends Shape {
    private Double radius = 5.0;

    // 看这个注解@Override, 它表示这个方法来自父类
    // Shape, 并且在这里被重写了
    @Override
    public Double area(){
        return 3.14 * radius * radius;
    }
}
```

类似的矩形类：

```
public class Rectangle extends Shape {
    private Double length = 5.0;
    private Double breadth= 10.0;

    // 看这个注解@Override, 它表示这个方法来自父类
    // Shape, 并且在这里被重写了
    @Override
    public Double area(){
        return length * breadth;
    }
}
```

One of common reasons of using method overloading is the simplicity of code it provides. For example remember `String.valueOf()` which takes almost any type of argument? What is written behind the scene is probably something like this:

```
static String valueOf(boolean b)
static String valueOf(char c)
static String valueOf(char[] data)
static String valueOf(char[] data, int offset, int count)
static String valueOf(double d)
static String valueOf(float f)
static String valueOf(int i)
static String valueOf(long l)
static String valueOf(Object obj)
```

Method Overriding

Well, method overriding (yes you guess it right, it is also known as dynamic polymorphism) is somewhat more interesting and complex topic.

In method overriding we overwrite the method body provided by the parent class. Got it? No? Let's go through an example.

```
public abstract class Shape{
    public abstract Double area(){
        return 0.0;
    }
}
```

So we have a class called Shape and it has method called area which will probably return the area of the shape.

Let's say now we have two classes called Circle and Rectangle.

```
public class Circle extends Shape {
    private Double radius = 5.0;

    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return 3.14 * radius * radius;
    }
}
```

Similarly rectangle class:

```
public class Rectangle extends Shape {
    private Double length = 5.0;
    private Double breadth= 10.0;

    // See this annotation @Override, it is telling that this method is from parent
    // class Shape and is overridden here
    @Override
    public Double area(){
        return length * breadth;
    }
}
```

}

所以，现在你们两个子类都拥有了父类 (Shape) 提供的更新方法体。现在的问题是如何查看结果？好吧，我们用老式的psvm方式来做。

```
public class AreaFinder{

    public static void main(String[] args){

        //这将创建一个circle类的对象
        Shape circle = new Circle();
        //这将创建一个Rectangle类的对象
        Shape rectangle = new Rectangle();

        // 鼓点.....
        //这应该打印78.5
        System.out.println("圆的形状 : "+circle.area());

        //这应该打印50.0
        System.out.println("矩形的形状: "+rectangle.area());
    }
}
```

哇！这不是很棒吗？两个相同类型的对象调用相同的方法却返回不同的值。朋友，这就是动态多态的力量。

这里有一张图表，更好地比较这两者之间的差异：

方法重载

方法重载用于提高程序的可读性。

方法重载在类内部进行。

在方法重载的情况下，参数必须不同。

方法重载是编译时多态性的例子。

在Java中，方法重载不能仅通过改变方法的返回类型来实现。返回类型在方法重载中可以相同或不同，但必须更改参数。

方法重写

方法重写用于提供其超类已经提供的方法的具体实现。

方法重写发生在具有IS-A (继承) 关系的两个类之间。

在方法重写的情况下，参数必须相同。

方法重写是运行时多态性的例子。

方法重写中返回类型必须相同或协变。

}

So, now both of your children classes have updated method body provided by the parent (Shape) class. Now question is how to see the result? Well lets do it the old psvm way.

```
public class AreaFinder{

    public static void main(String[] args){

        //This will create an object of circle class
        Shape circle = new Circle();
        //This will create an object of Rectangle class
        Shape rectangle = new Rectangle();

        // Drumbeats .....
        //This should print 78.5
        System.out.println("Shape of circle : "+circle.area());

        //This should print 50.0
        System.out.println("Shape of rectangle: "+rectangle.area());
    }
}
```

Wow! isn't it great? Two objects of same type calling same methods and returning different values. My friend, that's the power of dynamic polymorphism.

Here's a chart to better compare the differences between these two:

Method Overloading

Method overloading is used to increase the readability of the program.

Method overloading is performed within class.

In case of method overloading, parameter must be different.

Method overloading is the example of compile time polymorphism.

In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.

Method Overriding

Method overriding is used to provide the specific implementation of the method that is already provided by its super class.

Method overriding occurs in two classes that have IS-A (inheritance) relationship.

In case of method overriding, parameter must be same.

Method overriding is the example of run time polymorphism.

Return type must be same or covariant in method overriding.

第48.3节：构造函数

构造函数是以类名命名且没有返回类型的特殊方法，用于构造对象。构造函数像方法一样，可以接受输入参数。构造函数用于初始化对象。抽象类也可以有构造函数。

```
public class Hello{
    // 构造函数
    public Hello(String wordToPrint){
        printHello(wordToPrint);
    }
    public void printHello(String word){
        System.out.println(word);
    }
}
```

Section 48.3: Constructors

Constructors are special methods named after the class and without a return type, and are used to construct objects. Constructors, like methods, can take input parameters. Constructors are used to initialize objects. Abstract classes can have constructors also.

```
public class Hello{
    // constructor
    public Hello(String wordToPrint){
        printHello(wordToPrint);
    }
    public void printHello(String word){
        System.out.println(word);
    }
}
```

```
}
```

```
// 在创建对象时实例化并打印出wordToPrint的内容
```

理解构造函数与方法在多个方面的不同是很重要的：

1. 构造函数只能使用修饰符public、private和protected，不能声明为abstract，final、static或synchronized。
2. 构造函数没有返回类型。
3. 构造函数的名称必须与类名相同。在Hello示例中，Hello对象的构造函数名称与类名相同。
4. this关键字在构造函数中有额外的用法。this.method(...)调用当前实例的方法，而this(...)指当前类中具有不同签名的另一个构造函数。

构造函数也可以通过继承使用关键字super调用。

```
public class SuperManClass{  
  
    public 超人类(){  
        // 一些实现  
    }  
  
    // ... 方法  
}  
  
  
public class 蝙蝠侠类 extends 超人类{  
    public 蝙蝠侠类(){  
        super();  
    }  
    //... 方法...  
}
```

参见 Java 语言规范 #8.8 和 #15.9

第48.4节：使用静态

初始化器初始化静态常量字段

要初始化需要使用多个表达式的静态常量字段，可以使用静态初始化器来赋值。以下示例初始化了一个不可修改的String集合：

```
public class MyClass {  
  
    public static final Set<String> WORDS;  
  
    static {  
        Set<String> set = new HashSet<>();  
        set.add("Hello");  
        set.add("World");  
        set.add("foo");  
        set.add("bar");  
        set.add("42");  
        WORDS = Collections.unmodifiableSet(set);  
    }  
}
```

```
}  
}  
// instantiates the object during creating and prints out the content  
// of wordToPrint
```

It is important to understand that constructors are different from methods in several ways:

1. Constructors can only take the modifiers **public**, **private**, and **protected**, and cannot be declared **abstract**, **final**, **static**, or **synchronized**.
2. Constructors do not have a return type.
3. Constructors MUST be named the same as the class name. In the Hello example, the Hello object's constructor name is the same as the class name.
4. The **this** keyword has an additional usage inside constructors. **this.method(...)** calls a method on the current instance, while **this(...)** refers to another constructor in the current class with different signatures.

Constructors also can be called through inheritance using the keyword **super**.

```
public class SuperManClass{  
  
    public SuperManClass(){  
        // some implementation  
    }  
  
    // ... methods  
}  
  
  
public class BatmanClass extends SupermanClass{  
    public BatmanClass(){  
        super();  
    }  
    //... methods...  
}
```

See [Java Language Specification #8.8 and #15.9](#)

Section 48.4: Initializing static final fields using a static initializer

To initialize a **static final** fields that require using more than a single expression, a **static** initializer can be used to assign the value. The following example initializes a unmodifiable set of **String**s:

```
public class MyClass {  
  
    public static final Set<String> WORDS;  
  
    static {  
        Set<String> set = new HashSet<>();  
        set.add("Hello");  
        set.add("World");  
        set.add("foo");  
        set.add("bar");  
        set.add("42");  
        WORDS = Collections.unmodifiableSet(set);  
    }  
}
```

第48.5节：基本对象的构造与使用

对象属于它们自己的类，因此一个简单的例子是汽车（详见下文说明）：

```
public class Car {

    //描述单个汽车特征的变量，每个对象不同
    private int milesPerGallon;
    private String name;
    private String color;
    public int numGallonsInTank;

    public Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }

    //这是创建单个对象的地方
    public Car(int mpg, int gallonsInTank, String carName, String carColor){
        milesPerGallon = mpg;
        name = carName;
        color = carColor;
        numGallonsInTank = gallonsInTank;
    }

    //使对象更易用的方法

    //汽车需要行驶
    public void drive(int distanceInMiles){
        //获取汽车剩余的行驶里程
        int miles = numGallonsInTank * milesPerGallon;

        //检查汽车是否有足够的油行驶distanceInMiles
        if (miles <= distanceInMiles){
            numGallonsInTank = numGallonsInTank - (distanceInMiles / milesPerGallon)
            System.out.println("开了 " + numGallonsInTank + " 英里！");
        } else {
            System.out.println("无法行驶！");
        }
    }

    public void paintCar(String newColor){
        color = newColor;
    }

    //设置新的每加仑英里数
    public void setMPG(int newMPG){
        milesPerGallon = newMPG;
    }

    //设置油箱中的新加仑数
    public void setGallonsInTank(int numGallons){
        numGallonsInTank = numGallons;
    }

    public void nameCar(String newName){
        name = newName;
    }
}
```

Section 48.5: Basic Object Construction and Use

Objects come in their own class, so a simple example would be a car (detailed explanations below):

```
public class Car {

    //Variables describing the characteristics of an individual car, varies per object
    private int milesPerGallon;
    private String name;
    private String color;
    public int numGallonsInTank;

    public Car(){
        milesPerGallon = 0;
        name = "";
        color = "";
        numGallonsInTank = 0;
    }

    //this is where an individual object is created
    public Car(int mpg, int gallonsInTank, String carName, String carColor){
        milesPerGallon = mpg;
        name = carName;
        color = carColor;
        numGallonsInTank = gallonsInTank;
    }

    //methods to make the object more usable

    //Cars need to drive
    public void drive(int distanceInMiles){
        //get miles left in car
        int miles = numGallonsInTank * milesPerGallon;

        //check that car has enough gas to drive distanceInMiles
        if (miles <= distanceInMiles){
            numGallonsInTank = numGallonsInTank - (distanceInMiles / milesPerGallon)
            System.out.println("Drove " + numGallonsInTank + " miles!");
        } else {
            System.out.println("Could not drive!");
        }
    }

    public void paintCar(String newColor){
        color = newColor;
    }

    //set new Miles Per Gallon
    public void setMPG(int newMPG){
        milesPerGallon = newMPG;
    }

    //set new number of Gallon In Tank
    public void setGallonsInTank(int numGallons){
        numGallonsInTank = numGallons;
    }

    public void nameCar(String newName){
        name = newName;
    }
}
```

```

//获取汽车颜色
public String getColor(){
    return color;
}

//获取汽车名称
public String getName(){
    return name;
}

//获取加仑数
public String getGallons(){
    return numGallonsInTank;
}

}

```

对象是其类的实例。因此，创建对象的方法是在主类（Java中的main方法或Android中的onCreate）中通过两种方式之一调用Car类。

选项1

```
`Car newCar = new Car(30, 10, "Ferrari", "Red");
```

选项1是指在创建对象时，基本上告诉程序关于汽车的所有信息。更改汽车的任何属性都需要调用某个方法，例如repaintCar方法。示例：

```
newCar.repaintCar("Blue");
```

注意：确保传递给方法的数据类型正确。在上面的示例中，只要数据类型正确，也可以传递变量给repaintCar方法。

这是一个更改对象属性的示例，获取对象属性则需要使用Car类中带有返回值的方法（即非void的方法）。示例：

```
String myCarName = newCar.getName(); //返回字符串 "Ferrari"
```

当你在创建时拥有所有对象的数据时，选项1是最佳选择。

选项2

```
`Car newCar = new Car();
```

选项2实现了相同的效果，但需要更多工作来正确创建对象。我想回顾一下Car类中的这个构造函数：

```

public void Car(){
    milesPerGallon = 0;
    name = "";
    color = "";
    numGallonsInTank = 0;
}

```

注意，你实际上不必传递任何参数来创建对象。当你没有对象的所有属性但需要使用已有部分时，这非常有用。它将通用数据设置到对象的每个实例变量中，这样如果你调用不存在的数据，就不会抛出错误。

```

//Get the Car color
public String getColor(){
    return color;
}

//Get the Car name
public String getName(){
    return name;
}

//Get the number of Gallons
public String getGallons(){
    return numGallonsInTank;
}

```

Objects are **instances of** their class. So, the way you would **create an object** would be by calling the Car class in **one of two ways** in your main class (main method in Java or onCreate in Android).

Option 1

```
`Car newCar = new Car(30, 10, "Ferrari", "Red");
```

Option 1 is where you essentially tell the program everything about the Car upon creation of the object. Changing any property of the car would require calling one of the methods such as the repaintCar method. Example:

```
newCar.repaintCar("Blue");
```

Note: Make sure you pass the correct data type to the method. In the example above, you may also pass a variable to the repaintCar method **as long as the data type is correct**.

That was an example of changing properties of an object, receiving properties of an object would require using a method from the Car class that has a return value (meaning a method that is not **void**). Example:

```
String myCarName = newCar.getName(); //returns string "Ferrari"
```

Option 1 is the **best** option when you have **all the object's data** at the time of creation.

Option 2

```
`Car newCar = new Car();
```

Option 2 gets the same effect but required more work to create an object correctly. I want to recall this Constructor in the Car class:

```

public void Car(){
    milesPerGallon = 0;
    name = "";
    color = "";
    numGallonsInTank = 0;
}

```

Notice that you do not have to actually pass any parameters into the object to create it. This is very useful for when you do not have all the aspects of the object but you need to use the parts that you do have. This sets generic data into each of the instance variables of the object so that, if you call for a piece of data that does not exist, no errors

不会抛出错误。

注意：不要忘记稍后设置那些你未初始化的对象部分。例如，

```
Car myCar = new Car();
String color = Car.getColor(); //返回空字符串
```

这是一个常见错误，发生在对象未用所有数据初始化的情况下。错误被避免是因为存在一个构造函数允许创建带有占位变量的空Car对象（public Car(){}），但myCar的任何部分实际上都没有被自定义。**创建Car对象的正确示例：**

```
Car myCar = new Car();
myCar.nameCar("Ferrari");
myCar.paintCar("Purple");
myCar.setGallonsInTank(10);
myCar.setMPG(30);
```

另外，提醒一下，通过在主类中调用方法来获取对象的属性。示例：

```
String myCarName = myCar.getName(); //返回字符串 "Ferrari"
```

第48.6节：最简单的类

```
class TrivialClass {}
```

一个类至少由class关键字、名称和主体组成，主体可以为空。

你可以使用new操作符来实例化一个类。

```
TrivialClass tc = new TrivialClass();
```

第48.7节：对象成员与静态成员

使用以下类：

```
类 ObjectMemberVsStaticMember {
    静态 int staticCounter = 0;
    int memberCounter = 0;

    void increment() {
        staticCounter++;
        memberCounter++;
    }
}
```

以下代码片段：

```
final ObjectMemberVsStaticMember o1 = new ObjectMemberVsStaticMember();
final ObjectMemberVsStaticMember o2 = new ObjectMemberVsStaticMember();

o1.increment();

o2.increment();
o2.increment();
```

are thrown.

Note: Do not forget that you have to set the parts of the object later that you did not initialize it with. For example,

```
Car myCar = new Car();
String color = Car.getColor(); //returns empty string
```

This is a common mistake amongst objects that are not initialized with all their data. Errors were avoided because there is a Constructor that allows an empty Car object to be created with **stand-in variables** (public Car(){}), but no part of the myCar was actually customized. **Correct example of creating Car Object:**

```
Car myCar = new Car();
myCar.nameCar("Ferrari");
myCar.paintCar("Purple");
myCar.setGallonsInTank(10);
myCar.setMPG(30);
```

And, as a reminder, get an object's properties by calling a method in your main class. Example:

```
String myCarName = myCar.getName(); //returns string "Ferrari"
```

Section 48.6: Simplest Possible Class

```
class TrivialClass {}
```

A class consists at a minimum of the **class** keyword, a name, and a body, which might be empty.

You instantiate a class with the **new** operator.

```
TrivialClass tc = new TrivialClass();
```

Section 48.7: Object Member vs Static Member

With this class:

```
class ObjectMemberVsStaticMember {
    static int staticCounter = 0;
    int memberCounter = 0;

    void increment() {
        staticCounter++;
        memberCounter++;
    }
}
```

the following code snippet:

```
final ObjectMemberVsStaticMember o1 = new ObjectMemberVsStaticMember();
final ObjectMemberVsStaticMember o2 = new ObjectMemberVsStaticMember();

o1.increment();

o2.increment();
o2.increment();
```

```

System.out.println("o1 静态计数器 " + o1.staticCounter);
System.out.println("o1 成员计数器 " + o1.memberCounter);
System.out.println();

System.out.println("o2 静态计数器 " + o2.staticCounter);
System.out.println("o2 成员计数器 " + o2.memberCounter);
System.out.println();

System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.staticCounter);

// 以下代码无法编译。你需要一个对象
// 来访问它的成员
//System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
//ObjectMemberVsStaticMember.memberCounter);

```

输出结果如下：

```

o1 静态计数器 3
o1 成员计数器 1

o2 静态计数器 3
o2 成员计数器 2

ObjectMemberVsStaticMember.staticCounter = 3

```

注意：你不应该通过对象调用**static**成员，而应该通过类来调用。虽然这对JVM来说没有区别，但人类读者会更容易理解。

静态成员是类的一部分，每个类只存在一份。非静态成员存在于实例中，每个实例都有一份独立的副本。这也意味着你需要访问该类的对象才能访问它的成员。

```

System.out.println("o1 static counter " + o1.staticCounter);
System.out.println("o1 member counter " + o1.memberCounter);
System.out.println();

System.out.println("o2 static counter " + o2.staticCounter);
System.out.println("o2 member counter " + o2.memberCounter);
System.out.println();

System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
ObjectMemberVsStaticMember.staticCounter);

// the following line does not compile. You need an object
// to access its members
//System.out.println("ObjectMemberVsStaticMember.staticCounter = " +
//ObjectMemberVsStaticMember.memberCounter);

```

produces this output:

```

o1 static counter 3
o1 member counter 1

o2 static counter 3
o2 member counter 2

ObjectMemberVsStaticMember.staticCounter = 3

```

Note: You should not call **static** members on objects, but on classes. While it does not make a difference for the JVM, human readers will appreciate it.

static members are part of the class and exists only once per class. Non-**static** members exist on instances, there is an independent copy for each instance. This also means that you need access to an object of that class to access its members.

第49章：局部内部类

在方法内部创建的类称为Java中的局部内部类。如果你想调用局部内部类的方法，必须在该方法内部实例化这个类。

第49.1节：局部内部类

```
public class localInner1{  
    private int data=30;//实例变量  
    void display(){  
        class Local{  
            void msg(){System.out.println(data);}  
        }  
        Local l=new Local();  
        l.msg();  
    }  
    public static void main(String args[]){  
        localInner1 obj=new localInner1();  
        obj.display();  
    }  
}
```

Chapter 49: Local Inner Class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Section 49.1: Local Inner Class

```
public class localInner1{  
    private int data=30;//instance variable  
    void display(){  
        class Local{  
            void msg(){System.out.println(data);}  
        }  
        Local l=new Local();  
        l.msg();  
    }  
    public static void main(String args[]){  
        localInner1 obj=new localInner1();  
        obj.display();  
    }  
}
```

第50章：嵌套类和内部类

使用Java，开发者可以在一个类中定义另一个类。这样的类称为嵌套类。

如果嵌套类被声明为非静态，则称为内部类；否则，称为静态嵌套类。本页旨在通过示例记录并详细说明如何使用Java嵌套类和内部类。

第50.1节：使用嵌套类实现的简单栈

```
public class IntStack {  
  
    private IntStackNode head;  
  
    // IntStackNode是类IntStack的内部类  
    // 该内部类的每个实例作为栈中的一个链接  
    // 共同表示整个栈结构  
    private static class IntStackNode {  
  
        private int val;  
        private IntStackNode next;  
  
        private IntStackNode(int v, IntStackNode n) {  
            val = v;  
            next = n;  
        }  
  
        public IntStack push(int v) {  
            head = new IntStackNode(v, head);  
            return this;  
        }  
  
        public int pop() {  
            int x = head.val;  
            head = head.next;  
            return x;  
        }  
    }  
}
```

以及其使用，值得注意的是，完全没有承认嵌套类的存在。

```
public class Main {  
    public static void main(String[] args) {  
  
        IntStack s = new IntStack();  
        s.push(4).push(3).push(2).push(1).push(0);  
  
        //打印：0, 1, 2, 3, 4,  
        for(int i = 0; i < 5; i++) {  
            System.out.print(s.pop() + ", ");  
        }  
    }  
}
```

第50.2节：静态与非静态嵌套类

创建嵌套类时，您需要选择该嵌套类是否为静态：

Chapter 50: Nested and Inner Classes

Using Java, developers have the ability to define a class within another class. Such a class is called a [Nested Class](#). Nested Classes are called Inner Classes if they were declared as non-static, if not, they are simply called Static Nested Classes. This page is to document and provide details with examples on how to use Java Nested and Inner Classes.

Section 50.1: A Simple Stack Using a Nested Class

```
public class IntStack {  
  
    private IntStackNode head;  
  
    // IntStackNode is the inner class of the class IntStack  
    // Each instance of this inner class functions as one link in the  
    // Overall stack that it helps to represent  
    private static class IntStackNode {  
  
        private int val;  
        private IntStackNode next;  
  
        private IntStackNode(int v, IntStackNode n) {  
            val = v;  
            next = n;  
        }  
  
        public IntStack push(int v) {  
            head = new IntStackNode(v, head);  
            return this;  
        }  
  
        public int pop() {  
            int x = head.val;  
            head = head.next;  
            return x;  
        }  
    }  
}
```

And the use thereof, which (notably) does not at all acknowledge the existence of the nested class.

```
public class Main {  
    public static void main(String[] args) {  
  
        IntStack s = new IntStack();  
        s.push(4).push(3).push(2).push(1).push(0);  
  
        //prints: 0, 1, 2, 3, 4,  
        for(int i = 0; i < 5; i++) {  
            System.out.print(s.pop() + ", ");  
        }  
    }  
}
```

Section 50.2: Static vs Non Static Nested Classes

When creating a nested class, you face a choice of having that nested class static:

```
public class 外部类1 {
    private static class 静态嵌套类 {
    }
}
```

或者非静态：

```
public class 外部类2 {
    private class 嵌套类 {
    }
}
```

本质上，静态嵌套类没有外围外部类的实例，而非静态嵌套类则有。这影响了嵌套类实例化的时间和地点，以及这些嵌套类实例允许访问的内容。补充上述示例：

```
public class 外部类1 {
    private int 一个字段;
    public void 一个方法() {}

    private static class 静态嵌套类 {
        private int 内部字段;

        private 静态嵌套类() {
            innerField = aField; //非法, 不能从静态上下文访问 aField
            aMethod();          //非法, 不能从静态上下文调用 aMethod
        }

        private StaticNestedClass(OuterClass1 instance) {
            innerField = instance.aField; //合法
        }
    }

    public static void aStaticMethod() {
        StaticNestedClass s = new StaticNestedClass(); //合法, 能够在静态上下文中构造
        //执行涉及 s 的操作...
    }
}

public class 外部类2 {
    private int aField;

    public void aMethod() {}

    private class NestedClass {
        private int innerField;

        private NestedClass() {
            innerField = aField; //合法
            aMethod(); //合法
        }
    }
}
```

```
public class OuterClass1 {
```

```
    private static class StaticNestedClass {
    }
}
```

Or non-static:

```
public class OuterClass2 {
    private class NestedClass {
    }
}
```

At its core, static nested classes *do not have a surrounding instance* of the outer class, whereas non-static nested classes do. This affects both where/when one is allowed to instantiate a nested class, and what instances of those nested classes are allowed to access. Adding to the above example:

```
public class OuterClass1 {
    private int aField;
    public void aMethod() {}

    private static class StaticNestedClass {
        private int innerField;

        private StaticNestedClass() {
            innerField = aField; //Illegal, can't access aField from static context
            aMethod();          //Illegal, can't call aMethod from static context
        }

        private StaticNestedClass(OuterClass1 instance) {
            innerField = instance.aField; //Legal
        }
    }

    public static void aStaticMethod() {
        StaticNestedClass s = new StaticNestedClass(); //Legal, able to construct in static context
        //Do stuff involving s...
    }
}

public class OuterClass2 {
    private int aField;

    public void aMethod() {}

    private class NestedClass {
        private int innerField;

        private NestedClass() {
            innerField = aField; //Legal
            aMethod(); //Legal
        }
    }
}
```

```

    }

public void aNonStaticMethod() {
    NestedClass s = new NestedClass(); //合法
}

public static void aStaticMethod() {
NestedClass s = new NestedClass(); //非法。没有外围的
OuterClass2 实例，无法构造。
                                //由于这是静态上下文，没有外围的
OuterClass2 实例
}

```

因此，你选择静态还是非静态主要取决于是否需要能够直接访问外围类的字段和方法，尽管这也会影响你何时何地可以构造嵌套类。

一般来说，除非需要访问外部类的字段和方法，否则应将嵌套类设为静态。类似于除非需要将字段设为公共，否则应将其设为私有，这样可以减少嵌套类的可见性（通过不允许访问外部实例），从而降低出错的可能性。

第50.3节：内部类的访问修饰符

这里可以找到Java中访问修饰符的完整解释。但它们如何与内部类交互呢？

public，像往常一样，允许任何能够访问该类型的作用域进行无限制访问。

```

public class 外部类 {

    public class 内部类 {
        public int x = 5;
    }

    public 内部类 createInner() {
        return new 内部类();
    }
}

public class 其他类 {

    public static void main(String[] args) {
        int x = new 外部类().createInner().x; //直接访问字段是合法的
    }
}

```

protected和默认修饰符（无修饰符）也表现如预期，与非嵌套类的行为相同。

private 有趣的是，它的限制范围并不是所属的类，而是限制在编译单元——.java 文件。这意味着外部类可以完全访问内部类的字段和方法，即使它们被标记为 **private**。

```

public class 外部类 {

```

```

    }

public void aNonStaticMethod() {
    NestedClass s = new NestedClass(); //Legal
}

public static void aStaticMethod() {
NestedClass s = new NestedClass(); //Illegal. Can't construct without surrounding
OuterClass2 instance.
                                //As this is a static context, there is no surrounding
OuterClass2 instance
}

```

Thus, your decision of static vs non-static mainly depends on whether or not you need to be able to directly access fields and methods of the outer class, though it also has consequences for when and where you can construct the nested class.

As a rule of thumb, make your nested classes static unless you need to access fields and methods of the outer class. Similar to making your fields private unless you need them public, this decreases the visibility available to the nested class (by not allowing access to an outer instance), reducing the likelihood of error.

Section 50.3: Access Modifiers for Inner Classes

A full explanation of Access Modifiers in Java can be found here. But how do they interact with Inner classes?

public, as usual, gives unrestricted access to any scope able to access the type.

```

public class OuterClass {

    public class InnerClass {
        public int x = 5;
    }

    public InnerClass createInner() {
        return new InnerClass();
    }
}

public class SomeOtherClass {

    public static void main(String[] args) {
        int x = new OuterClass().createInner().x; //Direct field access is legal
    }
}

```

both **protected** and the default modifier (of nothing) behave as expected as well, the same as they do for non-nested classes.

private, interestingly enough, does not restrict to the class it belongs to. Rather, it restricts to the compilation unit - the .java file. This means that Outer classes have full access to Inner class fields and methods, even if they are marked **private**.

```

public class OuterClass {

```

```

public class 内部类 {

    private int x;
    private void anInnerMethod() {}

}

public InnerClass aMethod() {
    InnerClass a = new InnerClass();
    a.x = 5; //合法
    a.anInnerMethod(); //合法
    return a;
}
}

```

内部类本身可以有除 public 以外的可见性。通过将其标记为 private 或其他受限访问修饰符，其他（外部）类将无法导入和赋值该类型。但它们仍然可以获得该类型对象的引用。

```

public class 外部类 {

    private class InnerClass{}

    public InnerClass makeInnerClass() {
        return new InnerClass();
    }
}

public class AnotherClass {

    public static void main(String[] args) {
        OuterClass o = new OuterClass();

        InnerClass x = o.makeInnerClass(); //非法, 找不到类型
        OuterClass.InnerClass x = o.makeInnerClass(); //非法, InnerClass 的可见性为私有
        Object x = o.makeInnerClass(); //合法
    }
}

```

第50.4节：匿名内部类

匿名内部类是一种内部类形式，它通过单条语句声明并实例化。因此，该类没有可在程序其他地方使用的名称；即它是匿名的。

匿名类通常用于需要创建轻量级类并作为参数传递的场景。这通常通过接口来实现。例如：

```

public static Comparator<String> CASE_INSENSITIVE =
    new Comparator<String>() {
        @Override
        public int compare(String string1, String string2) {
            return string1.toUpperCase().compareTo(string2.toUpperCase());
        }
    };

```

该匿名类定义了一个Comparator<String>对象 (CASE_INSENSITIVE)，用于比较两个字符串时忽略大小写差异。

其他经常通过匿名类实现和实例化的接口包括Runnable和

```

public class InnerClass {

    private int x;
    private void anInnerMethod() {}

}

public InnerClass aMethod() {
    InnerClass a = new InnerClass();
    a.x = 5; //Legal
    a.anInnerMethod(); //Legal
    return a;
}
}

```

The Inner Class itself can have a visibility other than **public**. By marking it **private** or another restricted access modifier, other (external) classes will not be allowed to import and assign the type. They can still get references to objects of that type, however.

```

public class OuterClass {

    private class InnerClass{}

    public InnerClass makeInnerClass() {
        return new InnerClass();
    }
}

public class AnotherClass {

    public static void main(String[] args) {
        OuterClass o = new OuterClass();

        InnerClass x = o.makeInnerClass(); //Illegal, can't find type
        OuterClass.InnerClass x = o.makeInnerClass(); //Illegal, InnerClass has visibility private
        Object x = o.makeInnerClass(); //Legal
    }
}

```

Section 50.4: Anonymous Inner Classes

An anonymous inner class is a form of inner class that is declared and instantiated with a single statement. As a consequence, there is no name for the class that can be used elsewhere in the program; i.e. it is anonymous.

Anonymous classes are typically used in situations where you need to be able to create a light-weight class to be passed as a parameter. This is typically done with an interface. For example:

```

public static Comparator<String> CASE_INSENSITIVE =
    new Comparator<String>() {
        @Override
        public int compare(String string1, String string2) {
            return string1.toUpperCase().compareTo(string2.toUpperCase());
        }
    };

```

This anonymous class defines a Comparator<String> object (CASE_INSENSITIVE) that compares two strings ignoring differences in case.

Other interfaces that are frequently implemented and instantiated using anonymous classes are **Runnable** and

Callable。例如：

```
// 使用匿名Runnable类提供一个实例，线程启动时将运行该实例。
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello world");
    }
});
t.start(); // 输出 "Hello world"
```

匿名内部类也可以基于类。在这种情况下，匿名类隐式地继承了现有的类。如果被继承的类是抽象类，那么匿名类必须实现所有抽象方法。它也可以重写非抽象方法。

构造函数

匿名类不能有显式构造函数。取而代之的是，定义了一个隐式构造函数，使用 `super(...)` 将任何参数传递给被继承类中的构造函数。例如：

```
SomeClass anon = new SomeClass(1, "幸福") {
    @Override
    public int someMethod(int arg) {
        // 执行某些操作
    }
};
```

我们匿名子类 `SomeClass` 的隐式构造函数将调用一个与调用签名 `SomeClass(int, String)` 匹配的 `SomeClass` 构造函数。如果没有可用的构造函数，将会出现编译错误。匹配的构造函数抛出的任何异常也会由隐式构造函数抛出。

自然地，当扩展接口时，这种方法不起作用。当你从接口创建匿名类时，该类的超类是 `java.lang.Object`，只有一个无参构造函数。

第50.5节：从外部创建非静态内部类的实例

任何外部类可见的内部类也可以从该类中创建。

内部类依赖于外部类，并且需要对外部类实例的引用。要创建内部类的实例，`new` 操作符只需在外部类的实例上调用即可。

```
class 外部类 {
    class 内部类 {
    }
}

class 外部类2 {

    外部类 outer = new 外部类();
    外部类.内部类 createInner() {
        return outer.new 内部类();
    }
}
```

Callable. For example:

```
// An anonymous Runnable class is used to provide an instance that the Thread
// will run when started.
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Hello world");
    }
});
t.start(); // Prints "Hello world"
```

Anonymous inner classes can also be based on classes. In this case, the anonymous class implicitly **extends** the existing class. If the class being extended is abstract, then the anonymous class must implement all abstract methods. It may also override non-abstract methods.

Constructors

An anonymous class cannot have an explicit constructor. Instead, an implicit constructor is defined that uses `super(...)` to pass any parameters to a constructor in the class that is being extended. For example:

```
SomeClass anon = new SomeClass(1, "happiness") {
    @Override
    public int someMethod(int arg) {
        // do something
    }
};
```

The implicit constructor for our anonymous subclass of `SomeClass` will call a constructor of `SomeClass` that matches the call signature `SomeClass(int, String)`. If no constructor is available, you will get a compilation error. Any exceptions that are thrown by the matched constructor are also thrown by the implicit constructor.

Naturally, this does not work when extending an interface. When you create an anonymous class from an interface, the classes superclass is `java.lang.Object` which only has a no-args constructor.

Section 50.5: Create instance of non-static inner class from outside

An inner class which is visible to any outside class can be created from this class as well.

The inner class depends on the outside class and requires a reference to an instance of it. To create an instance of the inner class, the `new` operator only needs to be called on an instance of the outer class.

```
class OuterClass {
    class InnerClass {
    }
}

class OutsideClass {

    OuterClass outer = new OuterClass();
    OuterClass.InnerClass createInner() {
        return outer.new InnerClass();
    }
}
```

注意用法为outer.new。

第50.6节：方法局部内部类

在方法内部定义的类称为**方法局部内部类**。在这种情况下，内部类的作用域限制在该方法内。

方法局部内部类只能在定义该内部类的方法内实例化。

使用方法局部内部类的示例：

```
public class 外部类 {
    private void 外部方法() {
        final int 外部整数 = 1;
        // 方法局部内部类
        class 方法局部内部类 {
            private void 打印() {
                System.out.println("方法局部内部类 " + 外部整数);
            }
        }
        // 访问内部类
        MethodLocalInnerClass inner = new MethodLocalInnerClass();
        inner.print();
    }

    public static void main(String args[]) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

执行将输出：

方法局部内部类 1

第50.7节：从非静态

内部类访问外部类

对外部类的引用使用类名和this

```
public class 外部类 {
    public class 内部类 {
        public void method() {
            System.out.println("我可以访问我的封闭类: " + OuterClass.this);
        }
    }
}
```

你可以直接访问外部类的字段和方法。

```
public class OuterClass {
    private int counter;

    public class 内部类 {
        public void method() {
            System.out.println("我可以访问 " + counter);
        }
    }
}
```

Note the usage as outer.new.

Section 50.6: Method Local Inner Classes

A class written within a method called **method local inner class**. In that case the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined.

The example of using method local inner class:

```
public class OuterClass {
    private void outerMethod() {
        final int outerInt = 1;
        // Method Local Inner Class
        class MethodLocalInnerClass {
            private void print() {
                System.out.println("Method local inner class " + outerInt);
            }
        }
        // Accessing the inner class
        MethodLocalInnerClass inner = new MethodLocalInnerClass();
        inner.print();
    }

    public static void main(String args[]) {
        OuterClass outer = new OuterClass();
        outer.outerMethod();
    }
}
```

Executing will give an output:

Method local inner class 1

Section 50.7: Accessing the outer class from a non-static inner class

The reference to the outer class uses the class name and this

```
public class OuterClass {
    public class InnerClass {
        public void method() {
            System.out.println("I can access my enclosing class: " + OuterClass.this);
        }
    }
}
```

You can access fields and methods of the outer class directly.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        public void method() {
            System.out.println("I can access " + counter);
        }
    }
}
```

```
    }
}
```

但如果发生名称冲突，你可以使用外部类的引用。

```
public class OuterClass {
    private int counter;

    public class 内部类 {
        private int 计数器;

        public void method() {
            System.out.println("我的计数器: " + 计数器);
            System.out.println("外部计数器: " + 外部类.this.计数器);

            // 更新我的计数器
            计数器 = 外部类.this.计数器;
        }
    }
}
```

```
    }
}
}
```

But in case of name collision you can use the outer class reference.

```
public class OuterClass {
    private int counter;

    public class InnerClass {
        private int counter;

        public void method() {
            System.out.println("My counter: " + counter);
            System.out.println("Outer counter: " + OuterClass.this.counter);

            // updating my counter
            counter = OuterClass.this.counter;
        }
    }
}
```

第51章：java.util.Objects类

第51.1节：对象空值检查的基本用法

方法中的空值检查

```
Object 可空对象 = 方法返回对象();
if (Objects.isNull(可空对象)) {
    return;
}
```

方法中的非空检查

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

第51.2节：Objects.nonNull()方法在

stream API中的引用使用

传统的集合空值检查方式

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

使用Objects.nonNull方法和Java8 Stream API，我们可以这样做上述操作：

```
List<Object> someObjects = methodGetList();
someObjects.stream()
.filter(Objects::nonNull)
.forEach(this::doSomething);
```

Chapter 51: The java.util.Objects Class

Section 51.1: Basic use for object null check

For null check in method

```
Object nullableObject = methodReturnObject();
if (Objects.isNull(nullableObject)) {
    return;
}
```

For not null check in method

```
Object nullableObject = methodReturnObject();
if (Objects.nonNull(nullableObject)) {
    return;
}
```

Section 51.2: Objects.nonNull() method reference use in stream api

In the old fashion way for collection null check

```
List<Object> someObjects = methodGetList();
for (Object obj : someObjects) {
    if (obj == null) {
        continue;
    }
    doSomething(obj);
}
```

With the `Objects.nonNull` method and Java8 Stream API, we can do the above in this way:

```
List<Object> someObjects = methodGetList();
someObjects.stream()
.filter(Objects::nonNull)
.forEach(this::doSomething);
```

第52章：默认方法

Java 8 引入的默认方法（Default Method），允许开发者向接口添加新方法，而不会破坏该接口的现有实现。它提供了灵活性，使接口能够定义一个实现，当实现该接口的类未提供该方法的实现时，将使用该默认实现。

第52.1节：默认方法的基本用法

```
/**  
 * 带有默认方法的接口  
 */  
public interface Printable {  
    default void printString() {  
        System.out.println( "默认实现" );  
    }  
}  
  
/**  
 * 依赖于 {@link #printString()} 默认实现的类  
 */  
public class WithDefault  
    implements Printable  
{  
}  
  
/**  
 * {@link #printString()} 的自定义实现  
 */  
public class OverrideDefault  
    implements Printable {  
    @Override  
    public void printString() {  
        System.out.println( "overridden implementation" );  
    }  
}
```

以下语句

```
new WithDefault().printString();  
new OverrideDefault().printString();
```

将产生以下输出：

默认实现

重写的实现

第52.2节：从实现类访问被重写的默认方法

在类中，super.foo() 只会在超类中查找。如果你想调用超接口中的默认实现，需要用接口名来限定 super : Fooable.super.foo()。

```
public interface Fooable {
```

Chapter 52: Default Methods

Default Method introduced in Java 8, allows developers to add new methods to an interface without breaking the existing implementations of this interface. It provides flexibility to allow the interface to define an implementation which will be used as default when a class which implements that interface fails to provide an implementation of that method.

Section 52.1: Basic usage of default methods

```
/**  
 * Interface with default method  
 */  
public interface Printable {  
    default void printString() {  
        System.out.println( "default implementation" );  
    }  
}  
  
/**  
 * Class which falls back to default implementation of {@link #printString()}  
 */  
public class WithDefault  
    implements Printable  
{  
}  
  
/**  
 * Custom implementation of {@link #printString()}  
 */  
public class OverrideDefault  
    implements Printable {  
    @Override  
    public void printString() {  
        System.out.println( "overridden implementation" );  
    }  
}
```

The following statements

```
new WithDefault().printString();  
new OverrideDefault().printString();
```

Will produce this output:

default implementation

overridden implementation

Section 52.2: Accessing overridden default methods from implementing class

In classes, `super.foo()` will look in superclasses only. If you want to call a default implementation from a superinterface, you need to qualify `super` with the interface name: `Fooable.super.foo()`.

```
public interface Fooable {
```

```

default int foo() {return 3;}

}

public class A extends Object implements Fooable {
    @Override
    public int foo() {
        //return super.foo() + 1; //错误：java.lang.Object 中没有 foo() 方法
        return Fooable.super.foo() + 1; //正确，返回 4
    }
}

```

第 52.3 节：为什么使用默认方法？

简单的答案是，它允许你在不破坏现有实现的情况下演进已有接口。

例如，你有一个 20 年前发布的 Swim 接口。

```

public interface Swim {
    void backStroke();
}

```

我们做得非常出色，我们的界面非常受欢迎，全球有许多基于此的实现，但你无法控制它们的源代码。

```

public class FooSwimmer implements Swim {
    public void backStroke() {
        System.out.println("Do backstroke");
    }
}

```

20年后，你决定为接口添加新功能，但看起来我们的接口被冻结了
因为这会破坏现有的实现。

幸运的是，Java 8 引入了一个全新的特性，称为默认方法。

我们现在可以向Swim接口添加新方法。

```

public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("默认的侧泳实现。可以被重写");
    }
}

```

现在我们接口的所有现有实现仍然可以正常工作。但最重要的是，它们可以在自己的时间内实现
新添加的方法。

这种变化的最大原因之一，也是其最大用途之一，是在Java集合框架中。Oracle无法向现有的Iterable接口添加foreach
方法，而不破坏所有实现了Iterable的现有代码。通过添加默认方法，现有的Iterable实现将继承默认实现。

第52.4节：在默认方法中访问其他接口方法

你也可以在默认方法中访问其他接口方法。

```

default int foo() {return 3;}

}

public class A extends Object implements Fooable {
    @Override
    public int foo() {
        //return super.foo() + 1; //error: no method foo() in java.lang.Object
        return Fooable.super.foo() + 1; //okay, returns 4
    }
}

```

Section 52.3: Why use Default Methods?

The simple answer is that it allows you to evolve an existing interface without breaking existing implementations.

For example, you have Swim interface that you published 20 years ago.

```

public interface Swim {
    void backStroke();
}

```

We did a great job, our interface is very popular, there are many implementation on that around the world and you
don't have control over their source code.

```

public class FooSwimmer implements Swim {
    public void backStroke() {
        System.out.println("Do backstroke");
    }
}

```

After 20 years, you've decided to add new functionality to the interface, but it looks like our interface is frozen
because it will break existing implementations.

Luckily Java 8 introduces brand new feature called [Default method](#).

We can now add new method to the Swim interface.

```

public interface Swim {
    void backStroke();
    default void sideStroke() {
        System.out.println("Default sidestroke implementation. Can be overridden");
    }
}

```

Now all existing implementations of our interface can still work. But most importantly they can implement the
newly added method in their own time.

One of the biggest reasons for this change, and one of its biggest uses, is in the Java Collections framework. Oracle
could not add a **foreach** method to the existing Iterable interface without breaking all existing code which
implemented Iterable. By adding default methods, existing Iterable implementation will inherit the default
implementation.

Section 52.4: Accessing other interface methods within default method

You can as well access other interface methods from within your default method.

```

public interface Summable {
    int getA();

    int getB();

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {
    @Override
    public int getA() {
        return 1;
    }

    @Override
    public int getB() {
        return 2;
    }
}

```

以下语句将打印3：

```
System.out.println(new Sum().calculateSum());
```

默认方法也可以与接口静态方法一起使用：

```

public interface Summable {
    static int getA() {
        return 1;
    }

    static int getB() {
        return 2;
    }

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {}

```

以下语句也将打印3：

```
System.out.println(new Sum().calculateSum());
```

第52.5节：默认方法多重继承冲突

考虑以下示例：

```

public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
    default void foo() { System.out.println("B.foo"); }
}

```

```

public interface Summable {
    int getA();

    int getB();

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {
    @Override
    public int getA() {
        return 1;
    }

    @Override
    public int getB() {
        return 2;
    }
}

```

The following statement will print 3:

```
System.out.println(new Sum().calculateSum());
```

Default methods could be used along with interface static methods as well:

```

public interface Summable {
    static int getA() {
        return 1;
    }

    static int getB() {
        return 2;
    }

    default int calculateSum() {
        return getA() + getB();
    }
}

public class Sum implements Summable {}

```

The following statement will also print 3:

```
System.out.println(new Sum().calculateSum());
```

Section 52.5: Default method multiple inheritance collision

Consider next example:

```

public interface A {
    default void foo() { System.out.println("A.foo"); }
}

public interface B {
    default void foo() { System.out.println("B.foo"); }
}

```

}

这里有两个接口声明了具有相同签名的default方法foo。

如果你想在新的接口中继承这两个接口，你必须在两者之间做出选择，因为Java强制你显式解决这个冲突。

首先，你可以将方法foo声明为具有相同签名的抽象方法，这将覆盖A和B的行为。

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

当你在类中实现ABExtendsAbstract时，你必须提供foo的实现：

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

或者第二种，你可以提供一个全新的default实现。你也可以通过从实现类访问被覆盖的默认方法来重用A和B foo方法的代码。

```
public interface ABExtends extends A, B {
    @Override
    default void foo() { System.out.println("ABExtends.foo"); }
}
```

当你在类中实现ABExtends时，不需要提供foo的实现：

```
public class ABExtendsImpl implements ABExtends {}
```

第52.6节：类、抽象类和接口方法优先级

类中的实现，包括抽象声明，优先于所有接口默认方法。

- 抽象类方法优先于接口默认方法。

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
    public void backStroke() {
        System.out.println("AbstractSwimmer.backStroke");
    }
}

public class FooSwimmer extends AbstractSwimmer { }
```

以下语句

}

Here are two interfaces declaring **default** method foo with the same signature.

If you will try to extend these both interfaces in the new interface you have to make choice of two, because Java forces you to resolve this collision explicitly.

First, you can declare method foo with the same signature as **abstract**, which will override A and B behaviour.

```
public interface ABExtendsAbstract extends A, B {
    @Override
    void foo();
}
```

And when you will implement ABExtendsAbstract in the **class** you will have to provide foo implementation:

```
public class ABExtendsAbstractImpl implements ABExtendsAbstract {
    @Override
    public void foo() { System.out.println("ABImpl.foo"); }
}
```

Or **second**, you can provide a completely new **default** implementation. You also may reuse code of A and B foo methods by Accessing overridden default methods from implementing class.

```
public interface ABExtends extends A, B {
    @Override
    default void foo() { System.out.println("ABExtends.foo"); }
}
```

And when you will implement ABExtends in the **class** you will not have to provide foo implementation:

```
public class ABExtendsImpl implements ABExtends {}
```

Section 52.6: Class, Abstract class and Interface method precedence

Implementations in classes, including abstract declarations, take precedence over all interface defaults.

- Abstract class method takes precedence over [Interface Default Method](#).

```
public interface Swim {
    default void backStroke() {
        System.out.println("Swim.backStroke");
    }
}

public abstract class AbstractSwimmer implements Swim {
    public void backStroke() {
        System.out.println("AbstractSwimmer.backStroke");
    }
}

public class FooSwimmer extends AbstractSwimmer { }
```

The following statement

```
new FooSwimmer().backStroke();
```

将产生

```
AbstractSwimmer.backStroke
```

- 类方法优先于接口默认方法

```
public interface Swim {  
    default void backStroke() {  
        System.out.println("Swim.backStroke");  
    }  
  
    public abstract class AbstractSwimmer implements Swim {  
    }  
  
    public class FooSwimmer extends AbstractSwimmer {  
        public void backStroke() {  
            System.out.println("FooSwimmer.backStroke");  
        }  
    }  
}
```

以下语句

```
new FooSwimmer().backStroke();
```

将产生

```
FooSwimmer.仰泳
```

```
new FooSwimmer().backStroke();
```

Will produce

```
AbstractSwimmer.backStroke
```

- Class method takes precedence over [Interface Default Method](#)

```
public interface Swim {  
    default void backStroke() {  
        System.out.println("Swim.backStroke");  
    }  
}  
  
public abstract class AbstractSwimmer implements Swim {  
}  
  
public class FooSwimmer extends AbstractSwimmer {  
    public void backStroke() {  
        System.out.println("FooSwimmer.backStroke");  
    }  
}
```

The following statement

```
new FooSwimmer().backStroke();
```

Will produce

```
FooSwimmer.backStroke
```

第53章：包

Java中的包用于对类和接口进行分组。这有助于开发者在类数量庞大时避免冲突。如果使用包，我们可以在不同的包中创建同名的类/接口。通过使用包，我们可以在另一个类中导入这部分内容。Java中有许多内置包，如 > 1.java.util > 2.java.lang > 3.java.io 我们也可以定义自己的用户自定义包。

第53.1节：使用包创建同名类

第一个 Test.class :

```
package foo.bar

public class Test { }
```

另一个包中的 Test.class

```
package foo.bar.baz

public class Test { }
```

上述是可以的，因为这两个类存在于不同的包中。

第53.2节：使用包保护范围

在Java中，如果不提供访问修饰符，变量的默认作用域是包保护级别。这意味着同一包内的类可以访问其他类的变量，就好像这些变量是公开可用的一样。

```
package foo.bar

public class 示例类 {
    double 示例数字;
    String 示例字符串;

    public 示例类() {
        示例数字 = 3;
        示例字符串 = "Test String";
    }
    //无getter或setter方法
}

package foo.bar

public class 另一个类 {
    示例类 clazz = new 示例类();

    System.out.println("示例数字: " + clazz.示例数字);
    //打印示例数字: 3
    System.out.println("示例字符串: " + clazz.示例字符串);
    //打印示例字符串: Test String
```

Chapter 53: Packages

package in java is used to group class and interfaces. This helps developer to avoid conflict when there are huge numbers of classes. If we use this package the classes we can create a class/interface with same name in different packages. By using packages we can import the piece of again in another class. There many *built in packages* in java like > 1.java.util > 2.java.lang > 3.java.io We can define our own *user defined packages*.

Section 53.1: Using Packages to create classes with the same name

First Test.class:

```
package foo.bar

public class Test { }
```

Also Test.class in another package

```
package foo.bar.baz

public class Test { }
```

The above is fine because the two classes exist in different packages.

Section 53.2: Using Package Protected Scope

In Java if you don't provide an access modifier the default scope for variables is package-protected level. This means that classes can access the variables of other classes within the same package as if those variables were publicly available.

```
package foo.bar

public class ExampleClass {
    double exampleNumber;
    String exampleString;

    public ExampleClass() {
        exampleNumber = 3;
        exampleString = "Test String";
    }
    //No getters or setters
}

package foo.bar

public class AnotherClass {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Prints Example Number: 3
    System.out.println("Example String: " + clazz.exampleString);
    //Prints Example String: Test String
```

```
}
```

此方法不适用于另一个包中的类：

```
package baz.foo

public class ThisShouldNotWork {
    ExampleClass clazz = new ExampleClass();

    System.out.println("示例数字: " + clazz.示例数字);
    //抛出异常
    System.out.println("示例字符串: " + clazz.示例字符串);
    //抛出异常
}
```

```
}
```

This method will not work for a class in another package:

```
package baz.foo

public class ThisShouldNotWork {
    ExampleClass clazz = new ExampleClass();

    System.out.println("Example Number: " + clazz.exampleNumber);
    //Throws an exception
    System.out.println("Example String: " + clazz.exampleString);
    //Throws an exception
}
```

第54章：继承

继承是面向对象的基本特性之一，其中一个类通过使用关键字 `extends` 获取并扩展另一个类的属性。有关接口和关键字 `implements`，请参见接口部分。

第54.1节：继承

通过类之间使用 `extends` 关键字，父类（也称为 父类 或 基类）的所有属性都会出现在子类（也称为 子类 或 派生类）中

```
public class BaseClass {  
  
    public void baseMethod(){  
        System.out.println("执行基类操作");  
    }  
}  
  
public class SubClass extends BaseClass {  
  
}
```

子类SubClass的实例已经继承了方法baseMethod()：

```
SubClass s = new SubClass();  
s.baseMethod(); //有效，打印“执行基类操作”
```

可以向子类添加额外内容。这样做允许子类具有额外功能，而无需对基类或同一基类的其他子类进行任何更改：

```
public class Subclass2 extends BaseClass {  
  
    public void anotherMethod() {  
        System.out.println("执行子类2操作");  
    }  
}  
  
Subclass2 s2 = new Subclass2();  
s2.baseMethod(); //仍然有效，打印“执行基类操作”  
s2.anotherMethod(); //也有效，打印“执行子类2操作”
```

字段也会被继承：

```
public class BaseClassWithField {  
  
    public int x;  
}  
  
public class SubClassWithField extends BaseClassWithField {  
  
    public SubClassWithField(int x) {  
        this.x = x; //可以访问字段  
    }  
}
```

私有字段和方法仍然存在于子类中，但无法访问：

Chapter 54: Inheritance

Inheritance is a basic object oriented feature in which one class acquires and extends upon the properties of another class, using the keyword `extends`. For Interfaces and the keyword `implements`, see interfaces.

Section 54.1: Inheritance

With the use of the `extends` keyword among classes, all the properties of the superclass (also known as the *Parent Class* or *Base Class*) are present in the subclass (also known as the *Child Class* or *Derived Class*)

```
public class BaseClass {  
  
    public void baseMethod(){  
        System.out.println("Doing base class stuff");  
    }  
}  
  
public class SubClass extends BaseClass {  
  
}
```

Instances of SubClass have *inherited* the method `baseMethod()`:

```
SubClass s = new SubClass();  
s.baseMethod(); //Valid, prints "Doing base class stuff"
```

Additional content can be added to a subclass. Doing so allows for additional functionality in the subclass without any change to the base class or any other subclasses from that same base class:

```
public class Subclass2 extends BaseClass {  
  
    public void anotherMethod() {  
        System.out.println("Doing subclass2 stuff");  
    }  
}  
  
Subclass2 s2 = new Subclass2();  
s2.baseMethod(); //Still valid, prints "Doing base class stuff"  
s2.anotherMethod(); //Also valid, prints "Doing subclass2 stuff"
```

Fields are also inherited:

```
public class BaseClassWithField {  
  
    public int x;  
}  
  
public class SubClassWithField extends BaseClassWithField {  
  
    public SubClassWithField(int x) {  
        this.x = x; //Can access fields  
    }  
}
```

`private` fields and methods still exist within the subclass, but are not accessible:

```

public class BaseClassWithPrivateField {
    private int x = 5;
    public int getX() {
        return x;
    }
}

public class SubClassInheritsPrivateField extends BaseClassWithPrivateField {
    public void printX() {
        System.out.println(x); //非法, 无法访问私有字段 x
        System.out.println(getX()); //合法, 打印 5
    }
}

SubClassInheritsPrivateField s = new SubClassInheritsPrivateField();
int x = s.getX(); //x 的值将是 5。

```

在Java中，每个类最多只能继承一个其他类。

```

public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //非法

```

这被称为多重继承，虽然在某些语言中是合法的，但Java不允许类进行多重继承。

因此，每个类都有一条通向Object的单一祖先类链，所有类都继承自Object。

```

public class BaseClassWithPrivateField {
    private int x = 5;
    public int getX() {
        return x;
    }
}

public class SubClassInheritsPrivateField extends BaseClassWithPrivateField {
    public void printX() {
        System.out.println(x); //Illegal, can't access private field x
        System.out.println(getX()); //Legal, prints 5
    }
}

SubClassInheritsPrivateField s = new SubClassInheritsPrivateField();
int x = s.getX(); //x will have a value of 5.

```

In Java, each class may extend at most one other class.

```

public class A{}
public class B{}
public class ExtendsTwoClasses extends A, B {} //Illegal

```

This is known as multiple inheritance, and while it is legal in some languages, Java does not permit it with classes.

As a result of this, every class has an unbranching ancestral chain of classes leading to `Object`, from which all classes descend.

Section 54.2: Abstract Classes

An abstract class is a class marked with the `abstract` keyword. It, contrary to non-abstract class, may contain abstract - implementation-less - methods. It is, however, valid to create an abstract class without abstract methods.

An abstract class cannot be instantiated. It can be sub-classed (extended) as long as the sub-class is either also abstract, or implements all methods marked as abstract by super classes.

An example of an abstract class:

```

public abstract class Component {
    private int x, y;

    public setPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void render();
}

```

当类至少有一个抽象方法时，必须将该类标记为抽象类。抽象方法是没有实现的方法。抽象类中可以声明其他具有实现的方法，以便为任何子类提供通用代码。

尝试实例化该类将导致编译错误：

```

public abstract class Component {
    private int x, y;

    public setPosition(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void render();
}

```

The class must be marked abstract, when it has at least one abstract method. An abstract method is a method that has no implementation. Other methods can be declared within an abstract class that have implementation in order to provide common code for any sub-classes.

Attempting to instantiate this class will provide a compile error:

```
//错误：Component 是抽象的；无法实例化  
Component myComponent = new Component();
```

但是，继承自 Component 并为其所有抽象方法提供实现的类可以被实例化。

```
public class Button extends Component {  
  
    @Override  
    public void render() {  
        //渲染一个按钮  
    }  
  
}  
  
public class TextBox extends Component {  
  
    @Override  
    public void render() {  
        //渲染一个文本框  
    }  
}
```

继承类的实例也可以被强制转换为父类（普通继承），当调用抽象方法时，它们会提供多态效果。

```
Component myButton = new Button();  
Component myTextBox = new TextBox();  
  
myButton.render(); //渲染一个按钮  
myTextBox.render(); //渲染一个文本框
```

抽象类与接口

抽象类和接口都提供了一种定义方法签名的方式，同时要求继承/实现的类提供具体实现。

抽象类和接口之间有两个关键区别：

- 一个类只能继承一个类，但可以实现多个接口。
- 抽象类可以包含实例（非static）字段，但接口只能包含static字段。

版本 < Java SE 8

接口中声明的方法不能包含实现，因此当需要提供额外的方法且这些方法的实现会调用抽象方法时，会使用抽象类。

版本 ≥ Java SE 8

Java 8 允许接口包含默认方法，通常使用接口的其他方法来实现，使得接口和抽象类在这方面同样强大。

```
//error: Component is abstract; cannot be instantiated  
Component myComponent = new Component();
```

However a class that extends Component, and provides an implementation for all of its abstract methods and can be instantiated.

```
public class Button extends Component {  
  
    @Override  
    public void render() {  
        //render a button  
    }  
}  
  
public class TextBox extends Component {  
  
    @Override  
    public void render() {  
        //render a textbox  
    }  
}
```

Instances of inheriting classes also can be cast as the parent class (normal inheritance) and they provide a polymorphic effect when the abstract method is called.

```
Component myButton = new Button();  
Component myTextBox = new TextBox();  
  
myButton.render(); //renders a button  
myTextBox.render(); //renders a text box
```

Abstract classes vs Interfaces

Abstract classes and interfaces both provide a way to define method signatures while requiring the extending/implementing class to provide the implementation.

There are two key differences between abstract classes and interfaces:

- A class may only extend a single class, but may implement many interfaces.
- An abstract class can contain instance (non-static) fields, but interfaces may only contain static fields.

Version < Java SE 8

Methods declared in interfaces could not contain implementations, so abstract classes were used when it was useful to provide additional methods which implemented the abstract methods.

Version ≥ Java SE 8

Java 8 allows interfaces to contain default methods, usually implemented using the other methods of the interface, making interfaces and abstract classes equally powerful in this regard.

抽象类的匿名子类

作为一种便利，Java 允许实例化抽象类子类的匿名实例，这些匿名实例在创建新对象时为抽象方法提供实现。以上述示例为例，这可能如下所示：

Anonymous subclasses of Abstract Classes

As a convenience java allows for instantiation of anonymous instances of subclasses of abstract classes, which provide implementations for the abstract methods upon creating the new object. Using the above example this could look like this:

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // 渲染一个快速一次性使用的组件
    }
}
```

第54.3节：使用 'final' 限制继承和重写

最终类

当在类声明中使用时，final 修饰符阻止其他类声明继承该类。final 类是继承类层次结构中的“叶子”类。

```
// 这声明了一个最终类
final class MyFinalClass {
    /* 一些代码 */
}

// 编译错误：不能继承自最终类 MyFinalClass
class MySubClass extends MyFinalClass {
    /* 更多代码 */
}
```

最终类的使用场景

最终类可以与private构造函数结合使用，以控制或防止类的实例化。这可以用来创建所谓的“工具类”，该类仅定义静态成员；即常量和静态方法。

```
public final class UtilityClass {

    // 私有构造函数，用以替代默认的可见构造函数
    private UtilityClass() {}

    // 静态成员仍然可以照常使用
    public static int doSomethingCool() {
        return 123;
    }
}
```

不可变类也应声明为final。（不可变类是指其实例在创建后不能被更改的类；参见不可变对象主题。）通过这样做，可以防止创建不可变类的可变子类。这将违反里氏替换原则，该原则要求子类型应遵守其超类型的“行为契约”。

从实际角度来看，将不可变类声明为final使得推理程序行为更加容易。它还解决了在执行不受信任代码的安全沙箱场景中的安全问题。（例如，由于String被声明为final，受信任的类无需担心它可能被欺骗接受可变子类，而不受信任的调用者随后可能偷偷更改该子类。）

final类的一个缺点是它们无法与某些模拟框架（如Mockito）配合使用。

更新：Mockito 2 版本现在支持对最终类的模拟。

最终方法

```
Component myAnonymousComponent = new Component() {
    @Override
    public void render() {
        // render a quick 1-time use component
    }
}
```

Section 54.3: Using 'final' to restrict inheritance and overriding

Final classes

When used in a **class** declaration, the **final** modifier prevents other classes from being declared that extend the class. A **final** class is a "leaf" class in the inheritance class hierarchy.

```
// This declares a final class
final class MyFinalClass {
    /* some code */
}

// Compilation error: cannot inherit from final MyFinalClass
class MySubClass extends MyFinalClass {
    /* more code */
}
```

Use-cases for final classes

Final classes can be combined with a **private** constructor to control or prevent the instantiation of a class. This can be used to create a so-called "utility class" that only defines static members; i.e. constants and static methods.

```
public final class UtilityClass {

    // Private constructor to replace the default visible constructor
    private UtilityClass() {}

    // Static members can still be used as usual
    public static int doSomethingCool() {
        return 123;
    }
}
```

Immutable classes should also be declared as **final**. (An immutable class is one whose instances cannot be changed after they have been created; see the Immutable Objects topic.) By doing this, you make it impossible to create a mutable subclass of an immutable class. That would violate the Liskov Substitution Principle which requires that a subtype should obey the "behavioral contract" of its supertypes.

From a practical perspective, declaring an immutable class to be **final** makes it easier to reason about program behavior. It also addresses security concerns in the scenario where untrusted code is executed in a security sandbox. (For instance, since **String** is declared as **final**, a trusted class does not need to worry that it might be tricked into accepting mutable subclass, which the untrusted caller could then surreptitiously change.)

One disadvantage of **final** classes is that they do not work with some mocking frameworks such as Mockito. Update: Mockito version 2 now support mocking of final classes.

Final methods

`final` 修饰符也可以应用于方法，以防止它们在子类中被重写：

```
public class MyClassWithFinalMethod {  
  
    public final void someMethod() {  
    }  
  
}  
  
public class MySubClass extends MyClassWithFinalMethod {  
  
    @Override  
    public void someMethod() { // 编译错误 (重写的方法是 final)  
    }  
  
}
```

最终方法通常用于当你想限制子类可以更改类中的内容，但又不完全禁止子类时。

`final` 修饰符也可以应用于变量，但 `final` 对变量的含义与继承无关。

第54.4节：里氏替换原则

可替代性是面向对象编程中的一个原则，由芭芭拉·利斯科夫（Barbara Liskov）在1987年一次会议的主题演讲中提出，指出如果类B是类A的子类，那么在任何期望使用A的地方，都可以使用B代替：

```
class A {...}  
class B extends A {...}  
  
public void method(A obj) {...}  
  
A a = new B(); // 赋值合法  
method(new B()); // 作为参数传递合法
```

当类型是接口时，这一原则同样适用，此时对象之间不需要任何层级关系：

```
interface Foo {  
    void bar();  
}  
  
class A implements Foo {  
    void bar() {...}  
}  
  
class B implements Foo {  
    void bar() {...}  
}  
  
List<Foo> foos = new ArrayList<>();  
foos.add(new A()); // 合法  
foos.add(new B()); // 合法
```

现在列表中包含不属于同一类层次结构的对象。

The `final` modifier can also be applied to methods to prevent them being overridden in sub-classes:

```
public class MyClassWithFinalMethod {  
  
    public final void someMethod() {  
    }  
  
}  
  
public class MySubClass extends MyClassWithFinalMethod {  
  
    @Override  
    public void someMethod() { // Compiler error (overridden method is final)  
    }  
  
}
```

Final methods are typically used when you want to restrict what a subclass can change in a class without forbidding subclasses entirely.

The `final` modifier can also be applied to variables, but the meaning of `final` for variables is unrelated to inheritance.

Section 54.4: The Liskov Substitution Principle

Substitutability is a principle in object-oriented programming introduced by Barbara Liskov in a 1987 conference keynote stating that, if class B is a subclass of class A, then wherever A is expected, B can be used instead:

```
class A {...}  
class B extends A {...}  
  
public void method(A obj) {...}  
  
A a = new B(); // Assignment OK  
method(new B()); // Passing as parameter OK
```

This also applies when the type is an interface, where there doesn't need to any hierarchical relationship between the objects:

```
interface Foo {  
    void bar();  
}  
  
class A implements Foo {  
    void bar() {...}  
}  
  
class B implements Foo {  
    void bar() {...}  
}  
  
List<Foo> foos = new ArrayList<>();  
foos.add(new A()); // OK  
foos.add(new B()); // OK
```

Now the list contains objects that are not from the same class hierarchy.

第54.5节：抽象类和接口的使用：“是一个”关系与“拥有”能力

何时使用抽象类：在多个相关对象之间实现相同或不同的行为

何时使用接口：由多个无关对象实现一个契约

抽象类创建“是一个”关系，而接口提供“拥有”能力。

下面的代码可以看出这一点：

```
public class InterfaceAndAbstractClassDemo{
    public static void main(String args[]){
        Dog dog = new Dog("Jack", 16);
        Cat cat = new Cat("Joe", 20);

        System.out.println("Dog:" + dog);
        System.out.println("Cat:" + cat);

        dog.remember();
        dog.protectOwner();
        Learn dl = dog;
        dl.learn();

        cat.remember();
        cat.protectOwner();

        Climb c = cat;
        c.climb();

        Man man = new Man("Ravindra", 40);
        System.out.println(man);

        Climb cm = man;
        cm.climb();
        Think t = man;
        t.think();
        Learn l = man;
        l.learn();
        Apply a = man;
        a.apply();
    }
}

抽象类 Animal{
    字符串 name;
    整数 lifeExpentency;
    公共 Animal(字符串 name, 整数 lifeExpentency ){
        this.name = name;
        this.lifeExpentency = lifeExpentency;
    }
    公共 抽象 无返回值 remember();
    公共 抽象 无返回值 protectOwner();

    公共 字符串 toString(){
        返回 this.getClass().getSimpleName() + ":" + name + ":" + lifeExpentency;
    }
}
类 Dog 延伸 Animal 实现 Learn{
```

Section 54.5: Abstract class and Interface usage: "Is-a" relation vs "Has-a" capability

When to use abstract classes: To implement the same or different behaviour among multiple related objects

When to use interfaces: to implement a contract by multiple unrelated objects

Abstract classes create "is a" relations while interfaces provide "has a" capability.

This can be seen in the code below:

```
public class InterfaceAndAbstractClassDemo{
    public static void main(String args[]){
        Dog dog = new Dog("Jack", 16);
        Cat cat = new Cat("Joe", 20);

        System.out.println("Dog:" + dog);
        System.out.println("Cat:" + cat);

        dog.remember();
        dog.protectOwner();
        Learn dl = dog;
        dl.learn();

        cat.remember();
        cat.protectOwner();

        Climb c = cat;
        c.climb();

        Man man = new Man("Ravindra", 40);
        System.out.println(man);

        Climb cm = man;
        cm.climb();
        Think t = man;
        t.think();
        Learn l = man;
        l.learn();
        Apply a = man;
        a.apply();
    }
}

abstract class Animal{
    String name;
    int lifeExpentency;
    public Animal(String name, int lifeExpentency ){
        this.name = name;
        this.lifeExpentency = lifeExpentency;
    }
    public abstract void remember();
    public abstract void protectOwner();

    public String toString(){
        return this.getClass().getSimpleName() + ":" + name + ":" + lifeExpentency;
    }
}
class Dog extends Animal implements Learn{
```

```

public Dog(String 名字,int 年龄){
    super(名字,年龄);
}
public void 记忆(){
    System.out.println(this.getClass().getSimpleName()+" 可以记忆5分钟");
}
public void 保护主人(){
    System.out.println(this.getClass().getSimpleName()+" 会保护主人");
}
public void 学习(){
    System.out.println(this.getClass().getSimpleName()+" 可以学习:");
}
}
class Cat extends Animal implements Climb {
    public Cat(String 名字,int 年龄){
        super(名字,年龄);
    }
    public void 记忆(){
        System.out.println(this.getClass().getSimpleName() + " 可以记忆16小时");
    }
    public void 保护主人(){
        System.out.println(this.getClass().getSimpleName()+" 不会保护主人");
    }
    public void 爬(){
        System.out.println(this.getClass().getSimpleName()+" 可以爬");
    }
}
接口 爬{
    void 爬();
}
接口 思考 {
    void 思考();
}

接口 学习 {
    void 学习();
}
接口 应用{
    void apply();
}

class 人类 implements 思考,学习,应用,攀登{
    String 名字;
    int age;

    public 人類(String 名字,int 年齡){
        this.名字 = 名字;
        this.年齡 = 年齡;
    }
    public void 思考(){
        System.out.println("我能思考:"+this.getClass().getSimpleName());
    }
    public void 学习(){
        System.out.println("我能学习:"+this.getClass().getSimpleName());
    }
    public void 应用(){
        System.out.println("我能应用:"+this.getClass().getSimpleName());
    }
    public void 爬(){
        System.out.println("我能攀登:"+this.getClass().getSimpleName());
    }
    public String toString(){
}

```

```

public Dog(String name,int age){
    super(name,age);
}
public void remember(){
    System.out.println(this.getClass().getSimpleName()+" can remember for 5 minutes");
}
public void protectOwner(){
    System.out.println(this.getClass().getSimpleName()+" will protect owner");
}
public void learn(){
    System.out.println(this.getClass().getSimpleName()+" can learn:");
}
}
class Cat extends Animal implements Climb {
    public Cat(String name,int age){
        super(name,age);
    }
    public void remember(){
        System.out.println(this.getClass().getSimpleName() + " can remember for 16 hours");
    }
    public void protectOwner(){
        System.out.println(this.getClass().getSimpleName()+" won't protect owner");
    }
    public void climb(){
        System.out.println(this.getClass().getSimpleName()+" can climb");
    }
}
interface Climb{
    void climb();
}
interface Think {
    void think();
}

interface Learn {
    void learn();
}
interface Apply{
    void apply();
}

class Man implements Think,Learn,Apply,Climb{
    String name;
    int age;

    public Man(String name,int age){
        this.name = name;
        this.age = age;
    }
    public void think(){
        System.out.println("I can think:"+this.getClass().getSimpleName());
    }
    public void learn(){
        System.out.println("I can learn:"+this.getClass().getSimpleName());
    }
    public void apply(){
        System.out.println("I can apply:"+this.getClass().getSimpleName());
    }
    public void climb(){
        System.out.println("I can climb:"+this.getClass().getSimpleName());
    }
    public String toString(){
}

```

```
    return "人 :" + name + ":年龄:" + age;  
}  
}
```

输出：

```
狗:狗:杰克:16  
猫:猫:乔:20  
狗能记忆5分钟  
狗会保护主人  
狗能学习：  
猫能记忆16小时  
猫不会保护主人  
猫会爬树  
人 :拉文德拉:年龄:40  
我会爬树:人  
我会思考:人  
我会学习:人  
我会应用:人
```

重点笔记：

1. Animal 是一个具有共享属性的抽象类：name 和 lifeExpectancy 以及抽象方法：

remember() 和 protectOwner()。Dog 和 Cat 是实现了 remember() 和 protectOwner() 方法的 Animals。

2. Cat 可以 climb() 但 Dog 不能。Dog 可以 think() 但 Cat 不能。这些特定能力是通过实现添加到 Cat 和 Dog 中的。

3. Man 不是 Animal 但他可以 Think、Learn、Apply 和 Climb。

4. Cat 不是 Man 但它可以 Climb。

5. Dog 不是 Man 但它可以 Learn

6. Man 既不是 Cat 也不是 Dog，但可以拥有后两者的一些能力而无需继承 Animal、Cat 或 Dog。这是通过接口实现的。

7. 尽管 Animal 是一个抽象类，但它有构造函数，这与接口不同。

简而言之：

不相关的类可以通过接口拥有能力，但相关的类通过扩展基类来改变行为。

请参阅Java文档页面以了解在特定用例中应使用哪一种。

如果.....考虑使用抽象类

1. 您想在几个密切相关的类之间共享代码。
2. 您预计扩展您的抽象类的类具有许多公共方法或字段，或需要除public之外的访问修饰符（例如protected和private）。
3. 您想声明非静态或非最终字段。

如果.....考虑使用接口

1. 您预计不相关的类会实现您的接口。例如，许多不相关的对象可以实现Serializable接口。

```
    return "Man :" + name + ":Age:" + age;  
}
```

输出：

```
Dog:Dog:Jack:16  
Cat:Cat:Joe:20  
Dog can remember for 5 minutes  
Dog will protect owner  
Dog can learn:  
Cat can remember for 16 hours  
Cat won't protect owner  
Cat can climb  
Man :Ravindra:Age:40  
I can climb:Man  
I can think:Man  
I can learn:Man  
I can apply:Man
```

Key notes:

1. Animal is an abstract class with shared attributes: name and lifeExpectancy and abstract methods: remember() and protectOwner(). Dog and Cat are Animals that have implemented the remember() and protectOwner() methods.

2. Cat can climb() but Dog cannot. Dog can think() but Cat cannot. These specific capabilities are added to Cat and Dog by implementation.

3. Man is not an Animal but he can Think , Learn, Apply, and Climb.

4. Cat is not a Man but it can Climb.

5. Dog is not a Man but it can Learn

6. Man is neither a Cat nor a Dog but can have some of the capabilities of the latter two without extending Animal, Cat, or Dog. This is done with Interfaces.

7. Even though Animal is an abstract class, it has a constructor, unlike an interface.

TL;DR:

Unrelated classes can have capabilities through interfaces, but related classes change the behaviour through extension of base classes.

Refer to the Java documentation [page](#) to understand which one to use in a specific use case.

Consider using abstract classes if...

1. You want to share code among several closely related classes.
2. You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
3. You want to declare non-static or non-final fields.

Consider using interfaces if...

1. You expect that unrelated classes would implement your interface. For example, many unrelated objects can implement the [Serializable](#) interface.

2. 您想指定特定数据类型的行为，但不关心谁实现了该行为。
3. 您想利用类型的多重继承。

第54.6节：静态继承

静态方法可以像普通方法一样被继承，然而与普通方法不同的是，不可能创建“抽象”方法来强制静态方法的重写。编写一个与超类中静态方法签名相同的方法看似是一种重写，但实际上这只是创建了一个新的函数，隐藏了另一个。

```
public class BaseClass {
    public static int num = 5;

    public static void sayHello() {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        BaseClass.sayHello();
        System.out.println("BaseClass's num: " + BaseClass.num);

        SubClass.sayHello();
        //这将与上述语句的输出不同，因为它运行的是
        //不同的方法
        SubClass.sayHello(true);

        StaticOverride.sayHello();
        System.out.println("StaticOverride's num: " + StaticOverride.num);
    }
}

public class 子类 extends 基类 {
    //继承了 sayHello 函数，但没有重写它
    public static void sayHello(boolean 测试) {
        System.out.println("嘿");
    }
}

public static class 静态重写 extends 基类 {
    //隐藏了基类中的 num 字段
    //你甚至可以更改类型，因为这不会影响签名
    public static String num = "测试";

    //不能使用 @Override 注解，因为这是静态的
    //这重写了基类中的 sayHello 方法
    public static void sayHello() {
        System.out.println("静态说你好");
    }
}
```

运行这些类中的任何一个都会产生以下输出：

```
你好
基类的 num: 5
```

2. You want to specify the behaviour of a particular data type but are not concerned about who implements its behaviour.
3. You want to take advantage of multiple inheritance of type.

Section 54.6: Static Inheritance

Static method can be inherited similar to normal methods, however unlike normal methods it is impossible to create "abstract" methods in order to force static method overriding. Writing a method with the same signature as a static method in a super class appears to be a form of overriding, but really this simply creates a new function hides the other.

```
public class BaseClass {
    public static int num = 5;

    public static void sayHello() {
        System.out.println("Hello");
    }

    public static void main(String[] args) {
        BaseClass.sayHello();
        System.out.println("BaseClass's num: " + BaseClass.num);

        SubClass.sayHello();
        //This will be different than the above statement's output, since it runs
        //A different method
        SubClass.sayHello(true);

        StaticOverride.sayHello();
        System.out.println("StaticOverride's num: " + StaticOverride.num);
    }
}

public class SubClass extends BaseClass {
    //Inherits the sayHello function, but does not override it
    public static void sayHello(boolean test) {
        System.out.println("Hey");
    }
}

public static class StaticOverride extends BaseClass {
    //Hides the num field from BaseClass
    //You can even change the type, since this doesn't affect the signature
    public static String num = "test";

    //Cannot use @Override annotation, since this is static
    //This overrides the sayHello method from BaseClass
    public static void sayHello() {
        System.out.println("Static says Hi");
    }
}
```

Running any of these classes produces the output:

```
Hello
BaseClass's num: 5
```

```
你好  
嘿  
静态说你好  
StaticOverride 的数字 : 测试
```

请注意，与普通继承不同，在静态继承中方法不会被隐藏。你总是可以通过 `BaseClass.sayHello()` 调用基类的方法 `sayHello` 方法，但是如果子类中没有找到相同签名的方法，类确实会继承静态方法。如果两个方法的签名不同，子类可以运行这两个方法，即使名称相同。

静态字段以类似的方式相互隐藏。

第54.7节：面向接口编程

面向接口编程的理念是主要基于接口编写代码，只有在实例化时才使用具体类。在这种情况下，处理例如 Java 集合的良好代码看起来大致如下（方法本身并无实际用途，仅作示例）：

```
public <T> Set<T> toSet(Collection<T> collection) {  
    return Sets.newHashSet(collection);  
}
```

而糟糕的代码可能是这样的：

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {  
    return Sets.newHashSet(collection);  
}
```

不仅前者可以应用于更广泛的参数选择，其结果也更兼容其他开发者提供的通常遵循面向接口编程概念的代码。然而，使用前者的最重要原因是：

- 大多数情况下，结果被使用的上下文并不需要，也不应该需要像具体实现那样多的细节；
- 遵循接口可以强制代码更整洁，减少各种临时解决方案，比如又给某个类添加一个公共方法来应对某个特定场景；
- 代码更易于测试，因为接口容易被模拟（mock）；
- 最后，即使只期望有一个实现（至少为了测试性），这个概念也很有帮助。

那么，在编写新代码时，如何在心中只有一个特定实现的情况下，轻松地应用面向接口编程的概念呢？我们常用的一个方案是结合以下模式：

- 面向接口编程
- 工厂模式
- 建造者模式

以下基于这些原则的示例，是一个针对多种不同协议编写的RPC实现的简化和截断版本：

```
public interface RemoteInvoker {  
    <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);  
}
```

上述接口不应通过工厂直接实例化，而是进一步派生更多实现

```
Hello  
Hey  
Static says Hi  
StaticOverride's num: test
```

Note that unlike normal inheritance, in static inheritance methods are not hidden. You can always call the base `sayHello` method by using `BaseClass.sayHello()`. But classes do inherit static methods if no methods with the same signature are found in the subclass. If two method's signatures vary, both methods can be run from the subclass, even if the name is the same.

Static fields hide each other in a similar way.

Section 54.7: Programming to an interface

The idea behind programming to an interface is to base the code primarily on interfaces and only use concrete classes at the time of instantiation. In this context, good code dealing with e.g. Java collections will look something like this (not that the method itself is of any use at all, just illustration):

```
public <T> Set<T> toSet(Collection<T> collection) {  
    return Sets.newHashSet(collection);  
}
```

while bad code might look like this:

```
public <T> HashSet<T> toSet(ArrayList<T> collection) {  
    return Sets.newHashSet(collection);  
}
```

Not only the former can be applied to a wider choice of arguments, its results will be more compatible with code provided by other developers that generally adhere to the concept of programming to an interface. However, the most important reasons to use the former are:

- most of the time the context, in which the result is used, does not and should not need that many details as the concrete implementation provides;
- adhering to an interface forces cleaner code and less hacks such as yet another public method gets added to a class serving some specific scenario;
- the code is more testable as interfaces are easily mockable;
- finally, the concept helps even if only one implementation is expected (at least for testability).

So how can one easily apply the concept of programming to an interface when writing new code having in mind one particular implementation? One option that we commonly use is a combination of the following patterns:

- programming to an interface
- factory
- builder

The following example based on these principles is a simplified and truncated version of an RPC implementation written for a number of different protocols:

```
public interface RemoteInvoker {  
    <RQ, RS> CompletableFuture<RS> invoke(RQ request, Class<RS> responseClass);  
}
```

The above interface is not supposed to be instantiated directly via a factory, instead we derive further more

具体接口，一个用于HTTP调用，一个用于AMQP，每个接口都有一个工厂和一个构建器来构建实例，这些实例又是上述接口的实例：

```
public interface AmqpInvoker extends RemoteInvoker {  
    static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {  
        return new AmqpInvokerBuilder(instanceId, factory);  
    }  
}
```

现在可以像下面这样轻松构建用于AMQP的RemoteInvoker实例（或者根据构建器的不同，过程更复杂）：

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)  
    .requestRouter(router)  
    .build();
```

调用请求也同样简单：

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

由于Java 8允许将静态方法直接放入接口，上述代码中的中间工厂已隐式替换为AmqpInvoker.with()。在Java 8之前的版本，可以通过内部Factory类实现相同效果：

```
public interface AmqpInvoker extends RemoteInvoker {  
    class Factory {  
        public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {  
            return new AmqpInvokerBuilder(instanceId, factory);  
        }  
    }  
}
```

相应的实例化将变为：

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)  
    .requestRouter(router)  
    .build();
```

上面使用的构建器可能如下所示（尽管这是简化版，实际版本允许定义多达15个偏离默认值的参数）。注意该构造函数不是公共的，因此仅能从上述AmqpInvoker接口中使用：

```
public class AmqpInvokerBuilder {  
    ...  
    AmqpInvokerBuilder(String instanceId, ConnectionFactory factory) {  
        this.instanceId = instanceId;  
        this.factory = factory;  
    }  
  
    public AmqpInvokerBuilder requestRouter(RequestRouter requestRouter) {  
        this.requestRouter = requestRouter;  
        return this;  
    }  
  
    public AmqpInvoker build() throws TimeoutException, IOException {  
        return new AmqpInvokerImpl(instanceId, factory, requestRouter);  
    }  
}
```

concrete interfaces, one for HTTP invocation and one for AMQP, each then having a factory and a builder to construct instances, which in turn are also instances of the above interface:

```
public interface AmqpInvoker extends RemoteInvoker {  
    static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {  
        return new AmqpInvokerBuilder(instanceId, factory);  
    }  
}
```

Instances of RemoteInvoker for the use with AMQP can now be constructed as easy as (or more involved depending on the builder):

```
RemoteInvoker invoker = AmqpInvoker.with(instanceId, factory)  
    .requestRouter(router)  
    .build();
```

And an invocation of a request is as easy as:

```
Response res = invoker.invoke(new Request(data), Response.class).get();
```

Due to Java 8 permitting placing of static methods directly into interfaces, the intermediate factory has become implicit in the above code replaced with AmqpInvoker.with(). In Java prior to version 8, the same effect can be achieved with an inner Factory class:

```
public interface AmqpInvoker extends RemoteInvoker {  
    class Factory {  
        public static AmqpInvokerBuilder with(String instanceId, ConnectionFactory factory) {  
            return new AmqpInvokerBuilder(instanceId, factory);  
        }  
    }  
}
```

The corresponding instantiation would then turn into:

```
RemoteInvoker invoker = AmqpInvoker.Factory.with(instanceId, factory)  
    .requestRouter(router)  
    .build();
```

The builder used above could look like this (although this is a simplification as the actual one permits defining up to 15 parameters deviating from defaults). Note that the construct is not public, so it is specifically usable only from the above AmqpInvoker interface:

```
public class AmqpInvokerBuilder {  
    ...  
    AmqpInvokerBuilder(String instanceId, ConnectionFactory factory) {  
        this.instanceId = instanceId;  
        this.factory = factory;  
    }  
  
    public AmqpInvokerBuilder requestRouter(RequestRouter requestRouter) {  
        this.requestRouter = requestRouter;  
        return this;  
    }  
  
    public AmqpInvoker build() throws TimeoutException, IOException {  
        return new AmqpInvokerImpl(instanceId, factory, requestRouter);  
    }  
}
```

```
}
```

通常，也可以使用像 FreeBuilder 这样的工具生成构建器。

最后，该接口的标准（也是唯一预期的）实现被定义为包内类，以强制使用该接口、工厂和构建器：

```
class AmqpInvokerImpl 实现 AmqpInvoker {
    AmqpInvokerImpl(String instanceId, ConnectionFactory factory, RequestRouter requestRouter) {
        ...
    }

    @Override
    public <RQ, RS> CompletableFuture<RS> invoke(final RQ request, final Class<RS> respClass) {
        ...
    }
}
```

与此同时，这种模式被证明在开发我们所有新代码时非常高效，无论功能多么简单或复杂。

第54.8节：继承中的重写

继承中的重写用于当你在子类中使用超类中已定义的方法，但以不同于超类中原始设计的方式使用该方法。重写允许用户通过使用现有代码并对其进行修改，更好地满足用户需求，从而实现代码重用。

以下示例演示了ClassB如何通过更改打印方法中发送的内容来重写ClassA的功能：

示例：

```
public static void main(String[] args) {
    ClassA a = new ClassA();
    ClassA b = new ClassB();
    a.printing();
    b.printing();
}

class ClassA {
    public void printing() {
        System.out.println("A");
    }
}

class ClassB extends ClassA {
    public void printing() {
        System.out.println("B");
    }
}
```

输出：

```
A
```

```
}
```

Generally, a builder can also be generated using a tool like FreeBuilder.

Finally, the standard (and the only expected) implementation of this interface is defined as a package-local class to enforce the use of the interface, the factory and the builder:

```
class AmqpInvokerImpl implements AmqpInvoker {
    AmqpInvokerImpl(String instanceId, ConnectionFactory factory, RequestRouter requestRouter) {
        ...
    }

    @Override
    public <RQ, RS> CompletableFuture<RS> invoke(final RQ request, final Class<RS> respClass) {
        ...
    }
}
```

Meanwhile, this pattern proved to be very efficient in developing all our new code not matter how simple or complex the functionality is.

Section 54.8: Overriding in Inheritance

Overriding in Inheritance is used when you use a already defined method from a super class in a sub class, but in a different way than how the method was originally designed in the super class. Overriding allows the user to reuse code by using existing material and modifying it to suit the user's needs better.

The following example demonstrates how ClassB overrides the functionality of ClassA by changing what gets sent out through the printing method:

Example:

```
public static void main(String[] args) {
    ClassA a = new ClassA();
    ClassA b = new ClassB();
    a.printing();
    b.printing();
}

class ClassA {
    public void printing() {
        System.out.println("A");
    }
}

class ClassB extends ClassA {
    public void printing() {
        System.out.println("B");
    }
}
```

Output:

```
A
```

第54.9节：变量遮蔽

变量是被遮蔽的，方法是被重写的。使用哪个变量取决于变量声明的类。使用哪个方法取决于变量所引用对象的实际类。

```
class Car {
    public int gearRatio = 8;

    public String accelerate() {
        return "加速 : 汽车";
    }
}

class SportsCar extends Car {
    public int gearRatio = 9;

    public String accelerate() {
        return "加速 : SportsCar";
    }

    public void test() {
    }

    public static void main(String[] args) {
        Car car = new SportsCar();
        System.out.println(car.gearRatio + " " + car.accelerate());
        // 将打印出 8 Accelerate : SportsCar
    }
}
```

第54.10节：对象引用的缩小和扩展

将基类实例强制转换为子类，如：`b = (B) a;` 称为缩小（因为你试图将基类对象缩小为更具体的类对象），需要显式类型转换。

将子类实例赋值给基类，如：`A a = b;` 称为扩展，不需要类型转换。

为说明这一点，考虑以下类声明和测试代码：

```
class Vehicle {}

class 车 extends 车辆 {}

class 卡车 extends 车辆 {}

class 摩托车 extends 车辆 {}

class Test {
```

Section 54.9: Variable shadowing

Variables are SHADOWED and methods are OVERRIDDEN. Which variable will be used depends on the class that the variable is declared of. Which method will be used depends on the actual class of the object that is referenced by the variable.

```
class Car {
    public int gearRatio = 8;

    public String accelerate() {
        return "Accelerate : Car";
    }
}

class SportsCar extends Car {
    public int gearRatio = 9;

    public String accelerate() {
        return "Accelerate : SportsCar";
    }

    public void test() {
    }

    public static void main(String[] args) {
        Car car = new SportsCar();
        System.out.println(car.gearRatio + " " + car.accelerate());
        // will print out 8 Accelerate : SportsCar
    }
}
```

Section 54.10: Narrowing and Widening of object references

Casting an instance of a base class to a subclass as in `b = (B) a;` is called *narrowing* (as you are trying to narrow the base class object to a more specific class object) and needs an explicit type-cast.

Casting an instance of a subclass to a base class as in `A a = b;` is called *widening* and does not need a type-cast.

To illustrate, consider the following class declarations, and test code:

```
class Vehicle {}

class Car extends Vehicle {}

class Truck extends Vehicle {}

class MotorCycle extends Vehicle {}

class Test {
```

```

public static void main(String[] args) {
    车辆 vehicle = new 车();
    车 car = new 车();

    vehicle = car; // 有效, 无需强制类型转换

    车 c = vehicle // 无效
    车 c = (车) vehicle; // 有效
}

```

语句 `车辆 vehicle = new 车();` 是一个有效的Java语句。每个 `车` 实例也是一个 `车辆`。
因此，赋值是合法的，无需显式类型转换。

另一方面，`车 c = vehicle;` 是无效的。变量 `vehicle` 的静态类型是 `车辆`，这意味着
它可能引用一个 `车`、`卡车`、`摩托车`，或任何其他当前或未来的子类实例
`车辆`的实例。（或者实际上，是车辆本身的实例，因为我们没有将其声明为抽象类。）
该赋值不被允许，因为这可能导致`car`引用一个卡车实例。

为防止这种情况，我们需要添加显式类型转换：

```
汽车 c = (汽车) 车辆;
```

类型转换告诉编译器，我们期望`vehicle`的值是`Car`或`Car`的子类。如果有必要，编译器会插入代码以执行运行时类型检查。如果检查失败，则在代码执行时会抛出`ClassCastException`异常。

注意，并非所有类型转换都是有效的。例如：

```
String s = (String) vehicle; // 无效
```

Java编译器知道，与`Vehicle`类型兼容的实例永远不可能与
`String`类型兼容。该类型转换永远不会成功，JLS规定这会导致编译错误。

第54.11节：继承与静态方法

在Java中，父类和子类都可以有同名的静态方法。但在这种情况下，子类中静态方法的实现是隐藏父类的实现，而不是方法重写。例如：

```

class StaticMethodTest {

    // 静态方法与继承
    public static void main(String[] args) {
        Parent p = new Child();
        p.staticMethod(); // 输出 Inside Parent
        ((Child) p).staticMethod(); // 输出 Inside Child
    }

    静态类 Parent {
        public static void staticMethod() {
            System.out.println("Inside Parent");
        }
    }

    静态类 Child extends Parent {
        public static void staticMethod() {
    }
}

```

```

public static void main(String[] args) {
    Vehicle vehicle = new Car();
    Car car = new Car();

    vehicle = car; // is valid, no cast needed

    Car c = vehicle // not valid
    Car c = (Car) vehicle; // valid
}

```

The statement `Vehicle vehicle = new Car();` is a valid Java statement. Every instance of `Car` is also a `Vehicle`.
Therefore, the assignment is legal without the need for an explicit type-cast.

On the other hand, `Car c = vehicle;` is not valid. The static type of the `vehicle` variable is `Vehicle` which means
that it could refer to an instance of `Car`, `Truck`, `MotorCycle`, or any other current or future subclass
of `Vehicle`. (Or indeed, an instance of `Vehicle` itself, since we did not declare it as an abstract class.)
The assignment cannot be allowed, since that might lead to `car` referring to a `Truck` instance.`

To prevent this situation, we need to add an explicit type-cast:

```
Car c = (Car) vehicle;
```

The type-cast tells the compiler that we expect the value of `vehicle` to be a `Car` or a subclass of `Car`. If necessary,
compiler will insert code to perform a run-time type check. If the check fails, then a `ClassCastException` will be
thrown when the code is executed.

Note that not all type-casts are valid. For example:

```
String s = (String) vehicle; // not valid
```

The Java compiler knows that an instance that is type compatible with `Vehicle` cannot ever be type compatible with
`String`. The type-cast could never succeed, and the JLS mandates that this gives in a compilation error.

Section 54.11: Inheritance and Static Methods

In Java, parent and child class both can have static methods with the same name. But in such cases implementation
of static method in child is hiding parent class' implementation, it's not method overriding. For example:

```

class StaticMethodTest {

    // static method and inheritance
    public static void main(String[] args) {
        Parent p = new Child();
        p.staticMethod(); // prints Inside Parent
        ((Child) p).staticMethod(); // prints Inside Child
    }

    static class Parent {
        public static void staticMethod() {
            System.out.println("Inside Parent");
        }
    }

    static class Child extends Parent {
        public static void staticMethod() {
    }
}

```

```
    System.out.println("Inside Child");
}
}
```

静态方法绑定到类而非实例，并且这种方法绑定发生在编译时。由于在第一次调用staticMethod()时，使用了父类引用p，因此调用了Parent类的staticMethod()。在第二种情况下，我们将p强制转换为Child类，执行了Child的staticMethod()。

```
    System.out.println("Inside Child");
}
}
```

Static methods are bind to a class not to an instance and this method binding happens at compile time. Since in the first call to staticMethod(), parent class reference p was used, Parent's version of staticMethod() is invoked. In second case, we did cast p into Child class, Child's staticMethod() executed.

第55章：引用类型

第55.1节：不同的引用类型

`java.lang.ref`包提供了引用对象类，支持与垃圾回收器有限程度的交互。

Java有四种主要的不同引用类型。它们是：

- 强引用
- 弱引用
- 软引用
- 虚引用

1. 强引用

这是创建对象的常用形式。

```
MyObject myObject = new MyObject();
```

变量持有者持有对创建对象的强引用。只要该变量存活并持有此值，`MyObject`实例就不会被垃圾回收器回收。

2. 弱引用

当你不想长时间保留一个对象，并且需要尽快清理/释放为对象分配的内存时，采用这种方式。

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

简单来说，弱引用是一种不足以强制对象留在内存中的引用。弱引用允许你利用垃圾回收器判断可达性的能力，这样你就不必自己去管理。

当你需要你创建的对象时，只需使用`.get()`方法：

```
myObjectRef.get();
```

下面的代码将对此进行示例：

```
WeakReference myObjectRef = new WeakReference(MyObject);
System.out.println(myObjectRef.get()); // 这将打印对象引用地址
System.gc();
System.out.println(myObjectRef.get()); // 如果垃圾回收清理了对象，这将打印"null"
```

3. 软引用

软引用比弱引用稍强。你可以按如下方式创建一个软引用对象：

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

它们比弱引用更强地持有内存。如果你的内存供应/资源充足，垃圾回收器不会像对待弱引用那样积极地清理软引用。

Chapter 55: Reference Types

Section 55.1: Different Reference Types

`java.lang.ref` package provides reference-object classes, which support a limited degree of interaction with the garbage collector.

Java has four main different reference types. They are:

- Strong Reference
- Weak Reference
- Soft Reference
- Phantom Reference

1. Strong Reference

This is the usual form of creating objects.

```
MyObject myObject = new MyObject();
```

The variable holder is holding a strong reference to the object created. As long as this variable is live and holds this value, the `MyObject` instance will not be collected by the garbage collector.

2. Weak Reference

When you do not want to keep an object longer, and you need to clear/free the memory allocated for an object as soon as possible, this is the way to do so.

```
WeakReference myObjectRef = new WeakReference(MyObject);
```

Simply, a weak reference is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself.

When you need the object you created, just use `.get()` method:

```
myObjectRef.get();
```

Following code will exemplify this:

```
WeakReference myObjectRef = new WeakReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the object reference address
System.gc();
System.out.println(myObjectRef.get()); // This will print 'null' if the GC cleaned up the object
```

3. Soft Reference

Soft references are slightly stronger than weak references. You can create a soft referenced object as following:

```
SoftReference myObjectRef = new SoftReference(MyObject);
```

They can hold onto the memory more strongly than the weak reference. If you have enough memory supply/resources, garbage collector will not clean the soft references as enthusiastically as weak references.

软引用在缓存中非常实用。你可以创建软引用对象作为缓存，它们会被保留，直到你的内存耗尽。当内存资源不足时，垃圾回收器会移除软引用。

```
SoftReference myObjectRef = new SoftReference(MyObject);
System.out.println(myObjectRef.get()); // 这将打印对象的引用地址
System.gc();
System.out.println(myObjectRef.get()); // 这可能会打印也可能不会打印对象的引用地址
```

4. 幻影引用

这是最弱的引用类型。如果你使用虚引用（Phantom Reference）创建了一个对象引用，`get()`方法将始终返回null！

这种引用的用途是“虚引用对象在垃圾回收器确定它们的引用对象可能被回收后会被加入队列。虚引用通常用于以比Java终结机制更灵活的方式安排预先清理操作。”——摘自Oracle的虚引用（Phantom Reference）Javadoc。

你可以按如下方式创建一个虚引用对象：

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

Soft references are handy to use in caching. You can create soft referenced objects as a cache, where they kept until your memory runs out. When your memory can't supply enough resources, garbage collector will remove soft references.

```
SoftReference myObjectRef = new SoftReference(MyObject);
System.out.println(myObjectRef.get()); // This will print the reference address of the Object
System.gc();
System.out.println(myObjectRef.get()); // This may or may not print the reference address of the Object
```

4. Phantom Reference

This is the weakest referencing type. If you created an object reference using Phantom Reference, the `get()` method will always return null!

The use of this referencing is that "Phantom reference objects, which are enqueued after the collector determines that their referents may otherwise be reclaimed. Phantom references are most often used for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism." - From [Phantom Reference Javadoc](#) from Oracle.

You can create an object of Phantom Reference as following:

```
PhantomReference myObjectRef = new PhantomReference(MyObject);
```

第56章：控制台输入输出

第56.1节：从控制台读取用户输入

使用BufferedReader：

```
System.out.println("请输入你的名字并按回车。");

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    String name = reader.readLine();
    System.out.println("你好, " + name + " !");
} catch(IOException e) {
    System.out.println("发生错误: " + e.getMessage());
}
```

以下代码需要导入：

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

使用Scanner：

版本 ≥ Java SE 5

```
System.out.println("请输入您的姓名并按回车");

Scanner scanner = new Scanner(System.in);
String name = scanner.nextLine();

System.out.println("你好, " + name + " !");
```

本示例需要导入：

```
import java.util.Scanner;
```

要读取多行，请重复调用 `scanner.nextLine()`：

```
System.out.println("请分别在不同行输入您的名字和姓氏。");

Scanner scanner = new Scanner(System.in);
String firstName = scanner.nextLine();
String lastName = scanner.nextLine();

System.out.println("你好, " + firstName + " " + lastName + " !");
```

获取字符串有两种方法，`next()`和`nextLine()`。`next()`返回直到第一个空格的文本（也称为“标记”），而`nextLine()`返回用户输入直到按下回车键的所有文本。

Scanner 还提供了读取除 String 以外的数据类型的方法。这些包括：

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
```

Chapter 56: Console I/O

Section 56.1: Reading user input from the console

Using BufferedReader:

```
System.out.println("Please type your name and press Enter.");

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
try {
    String name = reader.readLine();
    System.out.println("Hello, " + name + " !");
} catch(IOException e) {
    System.out.println("An error occurred: " + e.getMessage());
}
```

The following imports are needed for this code:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

Using Scanner:

Version ≥ Java SE 5

```
System.out.println("Please type your name and press Enter");

Scanner scanner = new Scanner(System.in);
String name = scanner.nextLine();

System.out.println("Hello, " + name + " !");
```

The following import is needed for this example:

```
import java.util.Scanner;
```

To read more than one line, invoke `scanner.nextLine()` repeatedly:

```
System.out.println("Please enter your first and your last name, on separate lines.");

Scanner scanner = new Scanner(System.in);
String firstName = scanner.nextLine();
String lastName = scanner.nextLine();

System.out.println("Hello, " + firstName + " " + lastName + " !");
```

There are two methods for obtaining Strings, `next()` and `nextLine()`. `next()` returns text up until the first space (also known as a "token"), and `nextLine()` returns all text that the user inputted until pressing enter.

Scanner also provides utility methods for reading data types other than `String`. These include:

```
scanner.nextByte();
scanner.nextShort();
scanner.nextInt();
scanner.nextLong();
scanner.nextFloat();
scanner.nextDouble();
scanner.nextBigInteger();
```

```
scanner.nextBigDecimal();
```

在这些方法前加上 has (如 hasNextLine()、hasNextInt()) 会返回 true , 如果流中还有请求类型的数据。注意：如果输入不是请求的类型（例如，nextInt() 输入了 "a"），这些方法会导致程序崩溃。你可以使用 try {} catch() {} 来防止这种情况（参见：异常处理）

```
Scanner scanner = new Scanner(System.in); //创建扫描器
scanner.useLocale(Locale.US); //设置数字格式
System.out.println("请输入一个浮点数, 小数点为 .");
if (scanner.hasNextFloat()) { //检查是否为浮点数
    float fValue = scanner.nextFloat(); //直接以浮点数形式获取值
    System.out.println(fValue + " 是一个浮点数");
} else {
    String sValue = scanner.next(); //无法以浮点数形式获取
    System.out.println(sValue + " 不是一个浮点数");
}
```

使用 System.console :

版本 ≥ Java SE 6

```
String name = System.console().readLine("请输入您的姓名并按回车%");  
  
System.out.printf("你好, %s!", name);  
  
//读取密码 (如同Unix终端中不回显)
char[] password = System.console().readPassword();
```

优点:

- 读取方法是同步的
- 可以使用格式字符串语法

注意: 只有当程序从真实命令行运行且未重定向标准输入输出流时, 此方法才有效。若程序在某些IDE中运行 (如Eclipse) , 则无效。对于在IDE中运行且支持流重定向的代码, 请参见其他示例。

第56.2节 : 控制台中字符串对齐

方法PrintWriter.format (通过System.out.format调用) 可用于在控制台打印对齐的字符串。该方法接收一个包含格式信息的String和一系列要格式化的对象：

```
String rowsStrings[] = new String[] {"1",
                                         "1234",
                                         "1234567",
                                         "123456789";}  
  
String column1Format = "%-3s"; // 最少3个字符, 左对齐
String column2Format = "%-5.8s"; // 最少5个且最多8个字符, 左对齐
String column3Format = "%6.6s"; // 固定长度6个字符, 右对齐
String formatInfo = column1Format + " " + column2Format + " " + column3Format;  
  
for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

输出：

```
scanner.nextBigDecimal();
```

Prefixing any of these methods with has (as in hasNextLine(), hasNextInt()) returns **true** if the stream has any more of the request type. Note: These methods will crash the program if the input is not of the requested type (for example, typing "a" for nextInt()). You can use a **try {} catch() {}** to prevent this (see: Exceptions)

```
Scanner scanner = new Scanner(System.in); //Create the scanner
scanner.useLocale(Locale.US); //Set number format excepted
System.out.println("Please input a float, decimal separator is .");
if (scanner.hasNextFloat()) { //Check if it is a float
    float fValue = scanner.nextFloat(); //retrieve the value directly as float
    System.out.println(fValue + " is a float");
} else {
    String sValue = scanner.next(); //We can not retrieve as float
    System.out.println(sValue + " is not a float");
}
```

Using System.console:

Version ≥ Java SE 6

```
String name = System.console().readLine("Please type your name and press Enter%");  
  
System.out.printf("Hello, %s!", name);  
  
//To read passwords (without echoing as in unix terminal)
char[] password = System.console().readPassword();
```

Advantages:

- Reading methods are synchronized
- Format string syntax can be used

Note: This will only work if the program is run from a real command line without redirecting the standard input and output streams. It does not work when the program is run from within certain IDEs, such as Eclipse. For code that works within IDEs and with stream redirection, see the other examples.

Section 56.2: Aligning strings in console

The method [PrintWriter.format](#) (called through [System.out.format](#)) can be used to print aligned strings in console. The method receives a [String](#) with the format information and a series of objects to format:

```
String rowsStrings[] = new String[] {"1",
                                         "1234",
                                         "1234567",
                                         "123456789"};  
  
String column1Format = "%-3s"; // min 3 characters, left aligned
String column2Format = "%-5.8s"; // min 5 and max 8 characters, left aligned
String column3Format = "%6.6s"; // fixed size 6 characters, right aligned
String formatInfo = column1Format + " " + column2Format + " " + column3Format;  
  
for(int i = 0; i < rowsStrings.length; i++) {
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);
    System.out.println();
}
```

Output:

```
1 1 1  
1234 1234 1234  
1234567 1234567 123456  
123456789 12345678 123456
```

使用固定大小的格式字符串可以将字符串以表格形式打印，列宽固定：

```
String rowsStrings[] = new String[] {"1",  
                                     "1234",  
                                     "1234567",  
                                     "123456789";  
  
String column1Format = "%-3.3s"; // 固定长度3个字符, 左对齐  
String column2Format = "%-8.8s"; // 固定长度8个字符, 左对齐  
String column3Format = "%6.6s"; // 固定长度6个字符, 右对齐  
String formatInfo = column1Format + " " + column2Format + " " + column3Format;  
  
for(int i = 0; i < rowsStrings.length; i++) {  
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);  
    System.out.println();  
}
```

输出：

```
1 1 1  
123 1234 1234  
123 1234567 123456  
123 12345678 123456
```

格式字符串示例

- %s：仅仅是一个不带格式的字符串
- %5s：格式化字符串，最少为5个字符；如果字符串长度不足，则填充至5个字符并右对齐
- %-5s：格式化字符串，最少为5个字符；如果字符串长度不足，则填充至5个字符并左对齐
- %5.10s：格式化字符串，最少为5个字符，最多为10个字符；如果字符串长度不足5，则填充至5个字符并右对齐；如果字符串长度超过10，则截断为10个字符并右对齐
- %-5.5s：格式化字符串，固定长度为5个字符（最小值和最大值相等）；如果字符串长度不足5，则填充至5个字符并左对齐；如果字符串长度超过5，则截断为5个字符并左对齐

第56.3节：实现基本命令行行为

对于基本的原型或基本的命令行行为，以下循环非常有用。

```
public class ExampleCli {  
  
    private static final String CLI_LINE = "example-cli>"; // 控制台样式字符串  
    private static final String CMD_QUIT = "quit"; // 退出程序的字符串  
    private static final String CMD_HELLO = "hello"; // 在屏幕上打印"Hello World!"的字符串  
    private static final String CMD_ANSWER = "answer"; // 在屏幕上打印42的字符串
```

```
1 1 1  
1234 1234 1234  
1234567 1234567 123456  
123456789 12345678 123456
```

Using format strings with fixed size permits to print the strings in a table-like appearance with fixed size columns:

```
String rowsStrings[] = new String[] {"1",  
                                     "1234",  
                                     "1234567",  
                                     "123456789";  
  
String column1Format = "%-3.3s"; // fixed size 3 characters, left aligned  
String column2Format = "%-8.8s"; // fixed size 8 characters, left aligned  
String column3Format = "%6.6s"; // fixed size 6 characters, right aligned  
String formatInfo = column1Format + " " + column2Format + " " + column3Format;  
  
for(int i = 0; i < rowsStrings.length; i++) {  
    System.out.format(formatInfo, rowsStrings[i], rowsStrings[i], rowsStrings[i]);  
    System.out.println();  
}
```

Output:

```
1 1 1  
123 1234 1234  
123 1234567 123456  
123 12345678 123456
```

Format strings examples

- %s: just a string with no formatting
- %5s: format the string with a **minimum** of 5 characters; if the string is shorter it will be **padded** to 5 characters and **right** aligned
- %-5s: format the string with a **minimum** of 5 characters; if the string is shorter it will be **padded** to 5 characters and **left** aligned
- %5.10s: format the string with a **minimum** of 5 characters and a **maximum** of 10 characters; if the string is shorter than 5 it will be **padded** to 5 characters and **right** aligned; if the string is longer than 10 it will be **truncated** to 10 characters and **right** aligned
- %-5.5s: format the string with a **fixed** size of 5 characters (minimum and maximum are equals); if the string is shorter than 5 it will be **padded** to 5 characters and **left** aligned; if the string is longer than 5 it will be **truncated** to 5 characters and **left** aligned

Section 56.3: Implementing Basic Command-Line Behavior

For basic prototypes or basic command-line behavior, the following loop comes in handy.

```
public class ExampleCli {  
  
    private static final String CLI_LINE = "example-cli>"; //console like string  
  
    private static final String CMD_QUIT = "quit"; //string for exiting the program  
    private static final String CMD_HELLO = "hello"; //string for printing "Hello World!" on  
    the screen  
    private static final String CMD_ANSWER = "answer"; //string for printing 42 on the screen
```

```

public static void main(String[] args) {
    ExampleCli claimCli = new ExampleCli(); // 创建该类的一个对象

    try {
        claimCli.start(); // 调用start函数执行类似控制台的工作
    }
    catch (IOException e) {
        e.printStackTrace(); // 如果获取用户输入失败或发生类似情况，打印异常日志
    }
}

private void start() throws IOException {
    String cmd = "";

    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    while (!cmd.equals(CMD_QUIT)) { // 如果用户输入为"quit"，则终止控制台
        System.out.print(CLI_LINE); // 打印类似控制台的字符串

        cmd = reader.readLine(); // 从用户获取输入。用户输入应以
        // "hello"、"answer"或"quit"开头
        String[] cmdArr = cmd.split(" ");

        if (cmdArr[0].equals(CMD_HELLO)) { // 当用户输入以"hello"开头时执行
            hello(cmdArr);
        }
        else if (cmdArr[0].equals(CMD_ANSWER)) { // 当用户输入以
            // "answer"开头时执行
            answer(cmdArr);
        }
    }

    // 如果用户输入以"hello"开头，则在屏幕上打印"Hello World!"
    private void hello(String[] cmdArr) {
        System.out.println("Hello World!");
    }

    // 如果用户输入以"answer"开头，则在屏幕上打印"42"
    private void answer(String[] cmdArr) {
        System.out.println("42");
    }
}

```

```

public static void main(String[] args) {
    ExampleCli claimCli = new ExampleCli(); // creates an object of this class

    try {
        claimCli.start(); // calls the start function to do the work like console
    }
    catch (IOException e) {
        e.printStackTrace(); // prints the exception log if it is failed to do get the user
        input or something like that
    }
}

private void start() throws IOException {
    String cmd = "";

    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    while (!cmd.equals(CMD_QUIT)) { // terminates console if user input is "quit"
        System.out.print(CLI_LINE); // prints the console-like string

        cmd = reader.readLine(); // takes input from user. user input should be started with
        // "hello", "answer" or "quit"
        String[] cmdArr = cmd.split(" ");

        if (cmdArr[0].equals(CMD_HELLO)) { // executes when user input starts with "hello"
            hello(cmdArr);
        }
        else if (cmdArr[0].equals(CMD_ANSWER)) { // executes when user input starts with
            // "answer"
            answer(cmdArr);
        }
    }

    // prints "Hello World!" on the screen if user input starts with "hello"
    private void hello(String[] cmdArr) {
        System.out.println("Hello World!");
    }

    // prints "42" on the screen if user input starts with "answer"
    private void answer(String[] cmdArr) {
        System.out.println("42");
    }
}

```

第57章：流

流 (Stream) 表示一系列元素，并支持对这些元素执行不同类型的操作以进行计算。使用 Java 8，集合 (Collection) 接口有两个方法来生成流 (Stream) : stream() 和 parallelStream()。流操作分为中间操作和终端操作。中间操作返回一个流，因此可以在流关闭之前链式调用多个中间操作。终端操作要么返回 void，要么返回非流结果。

第57.1节：使用流

流 (Stream) 是一系列元素，可以对其执行顺序和并行的聚合操作。

任何给定的流 (Stream) 都可能有无限量的数据流经。因此，从流中接收的数据会在到达时逐个处理，而不是对所有数据进行批处理。结合lambda表达式，它们提供了一种简洁的方式，使用函数式方法对数据序列执行操作。

示例：(请参见[Ideone上的运行效果](#))

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

fruitStream.filter(s -> s.contains("a"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

输出：

```
APPLE
BANANA
ORANGE
PEAR
```

上述代码执行的操作可以总结如下：

1. 使用静态方法创建一个包含有序序列的字符串类型流 (Stream<String>)，其中元素为水果字符串工厂方法 Stream.of(values)。
2. filter() 操作仅保留匹配给定谓词的元素（即经过谓词测试返回 true 的元素）。
在此情况下，它保留包含 "a" 的元素。谓词以 lambda 表达式的形式给出。
3. map() 操作使用给定函数（称为映射器）转换每个元素。在此情况下，每个水果字符串被映射为其大写字符串版本，使用 方法引用 String::toUpperCase。

注意，map() 操作如果映射函数返回的类型与输入参数不同，将返回不同泛型类型的流。

例如，在 Stream<String> 上调用
.map(String::isEmpty) 会返回一个 Stream<Boolean>

4. sorted() 操作根据元素的自然顺序对 Stream 中的元素进行排序

Chapter 57: Streams

A Stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements. With Java 8, Collection interface has two methods to generate a Stream: stream() and parallelStream(). Stream operations are either intermediate or terminal. Intermediate operations return a Stream so multiple intermediate operations can be chained before the Stream is closed. Terminal operations are either void or return a non-stream result.

Section 57.1: Using Streams

A Stream is a sequence of elements upon which sequential and parallel aggregate operations can be performed. Any given Stream can potentially have an unlimited amount of data flowing through it. As a result, data received from a Stream is processed individually as it arrives, as opposed to performing batch processing on the data altogether. When combined with lambda expressions they provide a concise way to perform operations on sequences of data using a functional approach.

Example: ([see it work on Ideone](#))

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

fruitStream.filter(s -> s.contains("a"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Output:

```
APPLE
BANANA
ORANGE
PEAR
```

The operations performed by the above code can be summarized as follows:

1. Create a Stream<String> containing a sequenced ordered Stream of fruit String elements using the static factory method Stream.of(values).
2. The filter() operation retains only elements that match a given predicate (the elements that when tested by the predicate return true). In this case, it retains the elements containing an "a". The predicate is given as a lambda expression.
3. The map() operation transforms each element using a given function, called a mapper. In this case, each fruit String is mapped to its uppercase String version using the method-reference String::toUpperCase.

Note that the map() operation will return a stream with a different generic type if the mapping function returns a type different to its input parameter. For example on a Stream<String> calling
.map(String::isEmpty) returns a Stream<Boolean>

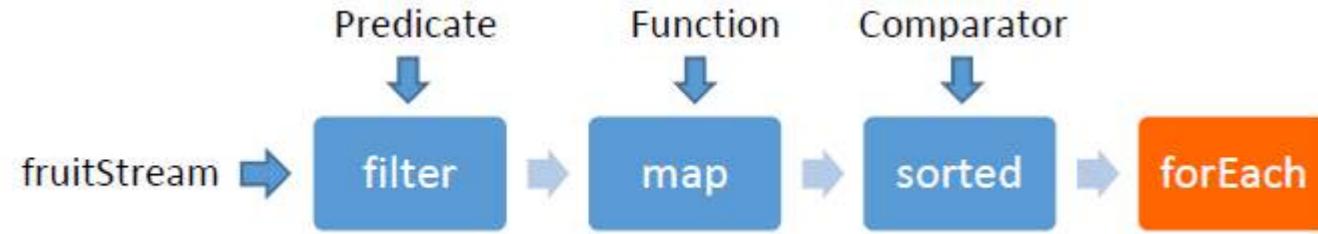
4. The sorted() operation sorts the elements of the Stream according to their natural ordering

(对于 String 来说，是按字典序排序)。

5. 最后，`forEach(action)` 操作对 Stream 的每个元素执行一个动作，将其传递给一个 `Consumer`。在示例中，每个元素只是被打印到控制台。该操作是一个终端操作，因此无法再次对其进行操作。

注意，定义在 Stream 上的操作是因为终端操作而执行的。

没有终端操作，流不会被处理。流不能被重复使用。一旦调用了终端操作，Stream 对象就变得不可用。



操作（如上所示）被链式连接，形成对数据的查询。

关闭流

注意，Stream 通常不需要关闭。只有操作 IO 通道的流才需要关闭。大多数 Stream 类型不操作资源，因此不需要关闭。

Stream 接口继承自 `AutoCloseable`。流可以通过调用 `close` 方法或使用 `try-with-resource` 语句来关闭。

一个需要关闭 Stream 的示例用例是当你从文件创建一个行流时：

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {  
    lines.forEach(System.out::println);  
}
```

Stream 接口还声明了 `Stream.onClose()` 方法，允许你注册 `Runnable` 处理器，当流关闭时会调用它们。一个示例用例是生成流的代码需要知道流何时被消费以执行一些清理操作。

```
public Stream<String> streamAndDelete(Path path) throws IOException {  
    return Files.lines(path).onClose(() -> someClass.deletePath(path));  
}
```

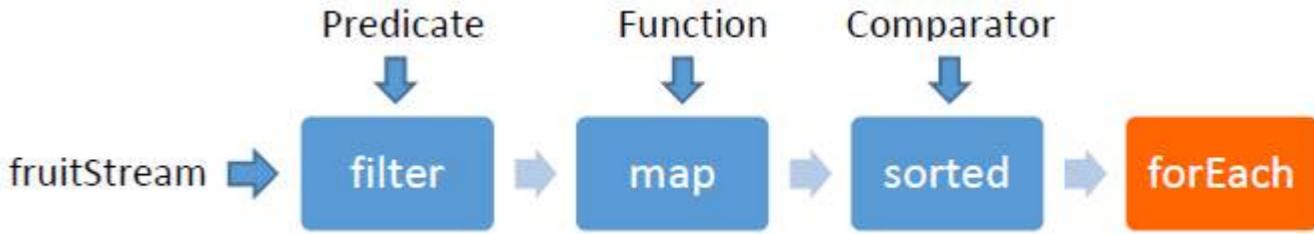
只有当调用了 `close()` 方法时，运行处理器才会执行，无论是显式调用还是通过 `try-with-resources` 语句隐式调用。

处理顺序

(lexicographically, in the case of `String`).

5. Finally, the `forEach(action)` operation performs an action which acts on each element of the Stream, passing it to a `Consumer`. In the example, each element is simply being printed to the console. This operation is a terminal operation, thus making it impossible to operate on it again.

Note that operations defined on the Stream are performed because of the terminal operation. Without a terminal operation, the stream is not processed. Streams can not be reused. Once a terminal operation is called, the Stream object becomes unusable.



Operations (as seen above) are chained together to form what can be seen as a query on the data.

Closing Streams

Note that a Stream generally does not have to be closed. It is only required to close streams that operate on IO channels. Most Stream types don't operate on resources and therefore don't require closing.

The Stream interface extends `AutoCloseable`. Streams can be closed by calling the `close` method or by using `try-with-resource` statements.

An example use case where a Stream should be closed is when you create a Stream of lines from a file:

```
try (Stream<String> lines = Files.lines(Paths.get("somePath"))) {  
    lines.forEach(System.out::println);  
}
```

The Stream interface also declares the `Stream.onClose()` method which allows you to register `Runnable` handlers which will be called when the stream is closed. An example use case is where code which produces a stream needs to know when it is consumed to perform some cleanup.

```
public Stream<String> streamAndDelete(Path path) throws IOException {  
    return Files.lines(path).onClose(() -> someClass.deletePath(path));  
}
```

The run handler will only execute if the `close()` method gets called, either explicitly or implicitly by a `try-with-resources` statement.

Processing Order

一个Stream对象的处理可以是顺序的或并行的。

在sequential模式下，元素按照Stream的源顺序进行处理。如果Stream是有序的（例如SortedMap实现或List），则处理顺序保证与源的顺序一致。

然而，在其他情况下，应注意不要依赖顺序（参见：[is the Java HashMap keySet\(\) iteration order consistent?](#)）。

示例：

```
List<Integer> integerList = Arrays.asList(0, 1, 2, 3, 42);

// 顺序处理
long howManyOddNumbers = integerList.stream()
    .filter(e -> (e % 2) == 1)
    .count();

System.out.println(howManyOddNumbers); // 输出: 2
```

[在 Ideone 上运行](#)

并行模式允许在多个核心上使用多个线程，但不保证元素处理的顺序。

如果在顺序Stream上调用多个方法，并不需要调用每个方法。例如，如果对Stream进行了过滤，元素数量减少到一个，则后续对 sort等方法的调用将不会发生。这可以提高顺序Stream的性能——这是并行Stream无法实现的优化Stream。

示例：

```
// 并行
long howManyOddNumbersParallel = integerList.parallelStream()
    .filter(e -> (e % 2) == 1)
    .count();

System.out.println(howManyOddNumbersParallel); // 输出: 2
```

[在 Ideone 上运行](#)

与容器（或集合）的区别

虽然某些操作可以同时在容器和流上执行，但它们最终服务于不同的目的并支持不同的操作。容器更关注元素的存储方式以及如何高效访问这些元素。另一方面，Stream不提供对其元素的直接访问和操作；它更专注于作为一个整体实体的对象集合，并对该实体整体执行操作。Stream和Collection是针对这些不同目的的两个独立的高级抽象。

第57.2节：消费流

只有当存在终端操作（如count()、collect()或forEach()）时，Stream才会被遍历。否则，Stream上不会执行任何操作。

在下面的示例中，Stream没有添加终端操作，因此filter()操作不会被调用，也不会产生任何输出，因为peek()不是终端操作。

A Stream object's processing can be sequential or parallel.

In a **sequential** mode, the elements are processed in the order of the source of the Stream. If the Stream is ordered (such as a [SortedMap](#) implementation or a [List](#)) the processing is guaranteed to match the ordering of the source. In other cases, however, care should be taken not to depend on the ordering (see: [is the Java HashMap keySet\(\) iteration order consistent?](#)).

Example:

```
List<Integer> integerList = Arrays.asList(0, 1, 2, 3, 42);

// sequential
long howManyOddNumbers = integerList.stream()
    .filter(e -> (e % 2) == 1)
    .count();

System.out.println(howManyOddNumbers); // Output: 2
```

[Live on Ideone](#)

Parallel mode allows the use of multiple threads on multiple cores but there is no guarantee of the order in which elements are processed.

If multiple methods are called on a sequential Stream, not every method has to be invoked. For example, if a Stream is filtered and the number of elements is reduced to one, a subsequent call to a method such as sort will not occur. This can increase the performance of a sequential Stream — an optimization that is not possible with a parallel Stream.

Example:

```
// parallel
long howManyOddNumbersParallel = integerList.parallelStream()
    .filter(e -> (e % 2) == 1)
    .count();

System.out.println(howManyOddNumbersParallel); // Output: 2
```

[Live on Ideone](#)

Differences from Containers (or Collections)

While some actions can be performed on both Containers and Streams, they ultimately serve different purposes and support different operations. Containers are more focused on how the elements are stored and how those elements can be accessed efficiently. A Stream, on the other hand, doesn't provide direct access and manipulation to its elements; it is more dedicated to the group of objects as a collective entity and performing operations on that entity as a whole. Stream and Collection are separate high-level abstractions for these differing purposes.

Section 57.2: Consuming Streams

A Stream will only be traversed when there is a *terminal operation*, like [count\(\)](#), [collect\(\)](#) or [forEach\(\)](#). Otherwise, no operation on the Stream will be performed.

In the following example, no terminal operation is added to the Stream, so the [filter\(\)](#) operation will not be invoked and no output will be produced because [peek\(\)](#) is NOT a *terminal operation*.

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

[在 Ideone 上运行](#)

这是一个带有有效终端操作的Stream序列，因此会产生输出。

你也可以使用forEach代替peek：

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

[在 Ideone 上运行](#)

输出：

2468终端操作执行后，

Stream被消费，无法重复使用。

虽然给定的流对象不能重复使用，但很容易创建一个可重用的Iterable，它委托给流管道。这对于返回实时数据集的修改视图而无需将结果收集到临时结构中非常有用。

```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

输出：

foo
bar
foo
bar

这是可行的，因为Iterable声明了一个抽象方法Iterator<T> iterator()。这样它实际上是一个函数式接口，由一个在每次调用时创建新流的lambda实现。

一般来说，Stream的操作如下面的图所示：

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).peek(System.out::println);
```

[Live on Ideone](#)

This is a Stream sequence with a valid *terminal operation*, thus an output is produced.

You could also use **forEach** instead of peek:

```
IntStream.range(1, 10).filter(a -> a % 2 == 0).forEach(System.out::println);
```

[Live on Ideone](#)

Output:

2
4
6
8

After the terminal operation is performed, the Stream is consumed and cannot be reused.

Although a given stream object cannot be reused, it's easy to create a reusable Iterable that delegates to a stream pipeline. This can be useful for returning a modified view of a live data set without having to collect results into a temporary structure.

```
List<String> list = Arrays.asList("FOO", "BAR");
Iterable<String> iterable = () -> list.stream().map(String::toLowerCase).iterator();

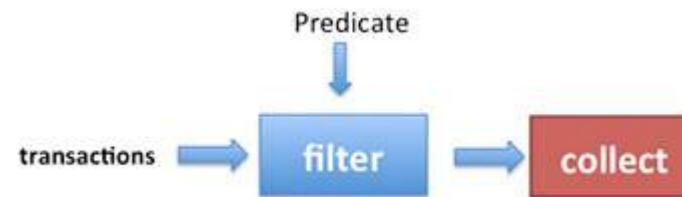
for (String str : iterable) {
    System.out.println(str);
}
for (String str : iterable) {
    System.out.println(str);
}
```

Output:

foo
bar
foo
bar

This works because Iterable declares a single abstract method Iterator<T> iterator(). That makes it effectively a functional interface, implemented by a lambda that creates a new stream on each call.

In general, a Stream operates as shown in the following image:



注意：即使没有终端操作，也会始终执行参数检查：

```

try {
    IntStream.range(1, 10).filter(null);
} catch (NullPointerException e) {
    System.out.println("由于传递了null作为filter()的参数，我们收到了NullPointerException");
}

```

[在Ideone上运行](#)

输出：

由于传递了null作为filter()的参数，我们收到了NullPointerException

第57.3节：创建频率映射

`groupingBy(classifier, downstream)`收集器允许将Stream元素收集到一个Map中，方法是将每个元素分类到一个组中，并对同一组中分类的元素执行下游操作。

这个原理的一个经典例子是使用Map来统计Stream中元素的出现次数。在这个例子中，分类器只是恒等函数，直接返回元素本身。后续操作使用`counting()`来统计相同元素的数量。

```

Stream.of("apple", "orange", "banana", "apple")
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
    .entrySet()
.forEach(System.out::println);

```

后续操作本身是一个收集器（`Collectors.counting()`），它作用于类型为String的元素，并产生类型为Long的结果。`collect`方法调用的结果是一个`Map<String, Long>`。

这将产生以下输出：

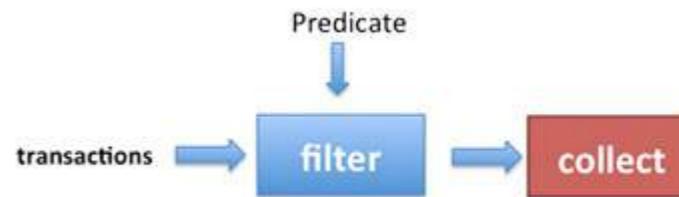
```

banana=1
orange=1
apple=2

```

第57.4节：无限流

可以生成一个不会结束的Stream。在无限Stream上调用终端方法会导致Stream进入无限循环。可以使用Stream的`limit`方法限制Java处理的Stream项数。



NOTE: Argument checks are always performed, even without a *terminal operation*:

```

try {
    IntStream.range(1, 10).filter(null);
} catch (NullPointerException e) {
    System.out.println("We got a NullPointerException as null was passed as an argument to filter()");
}

```

[Live on Ideone](#)

Output:

We got a NullPointerException as null was passed as an argument to filter()

Section 57.3: Creating a Frequency Map

The `groupingBy(classifier, downstream)` collector allows the collection of `Stream` elements into a `Map` by classifying each element in a group and performing a downstream operation on the elements classified in the same group.

A classic example of this principle is to use a `Map` to count the occurrences of elements in a `Stream`. In this example, the classifier is simply the identity function, which returns the element as-is. The downstream operation counts the number of equal elements, using `counting()`.

```

Stream.of("apple", "orange", "banana", "apple")
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
    .entrySet()
    .forEach(System.out::println);

```

The downstream operation is itself a collector (`Collectors.counting()`) that operates on elements of type `String` and produces a result of type `Long`. The result of the `collect` method call is a `Map<String, Long>`.

This would produce the following output:

```

banana=1
orange=1
apple=2

```

Section 57.4: Infinite Streams

It is possible to generate a `Stream` that does not end. Calling a terminal method on an infinite `Stream` causes the `Stream` to enter an infinite loop. The `limit` method of a `Stream` can be used to limit the number of terms of the `Stream` that Java processes.

这个例子生成了一个从数字1开始的所有自然数的Stream。每个后续项比前一项大1。
通过调用该Stream的limit方法，只考虑并打印Stream的前五项。

```
// 生成无限流 - 1, 2, 3, 4, 5, 6, 7, ...
IntStream naturalNumbers = IntStream.iterate(1, x -> x + 1);

// 仅打印前5项
naturalNumbers.limit(5).forEach(System.out::println);
```

输出：

12345生成无限流的另一种方法是使用

This example generates a Stream of all natural numbers, starting with the number 1. Each successive term of the Stream is one higher than the previous. By calling the limit method of this Stream, only the first five terms of the Stream are considered and printed.

```
// Generate infinite stream - 1, 2, 3, 4, 5, 6, 7, ...
IntStream naturalNumbers = IntStream.iterate(1, x -> x + 1);

// Print out only the first 5 terms
naturalNumbers.limit(5).forEach(System.out::println);
```

Output:

1
2
3
4
5

```
// 生成一个无限的随机数流
Stream<Double> infiniteRandomNumbers = Stream.generate(Math::random);

// 仅打印前10个随机数
infiniteRandomNumbers.limit(10).forEach(System.out::println);
```

第57.5节：将流中的元素收集到集合中

使用`toList()`和`toSet()`进行收集

可以通过使用`Stream.collect`操作轻松地将Stream中的元素收集到容器中：

```
System.out.println(Arrays
    .asList("apple", "banana", "pear", "kiwi", "orange")
    .stream()
    .filter(s -> s.contains("a"))
    .collect(Collectors.toList())
);
// 输出: [apple, banana, pear, orange]
```

其他集合实例，如`Set`，可以通过使用其他内置的`Collectors`方法来创建。例如，`Collectors.toSet()`将Stream中的元素收集到`Set`中。

对`List`或`Set`实现的显式控制根据`Collectors#toList()`和`Collectors#toSet()`

`Collectors#toList()`的文档，返回的`List`或`Set`在类型、可变性、可序列化性或线程安全性方面没有保证。

若需显式控制返回的实现，可以使用`Collectors#toCollection(Supplier)`，其中给定的供应商返回一个新的空集合。

```
// 使用方法引用的语法
```

Another way of generating an infinite stream is using the `Stream.generate` method. This method takes a lambda of type `Supplier`.

```
// Generate an infinite stream of random numbers
Stream<Double> infiniteRandomNumbers = Stream.generate(Math::random);

// Print out only the first 10 random numbers
infiniteRandomNumbers.limit(10).forEach(System.out::println);
```

Section 57.5: Collect Elements of a Stream into a Collection

Collect with `toList()` and `toSet()`

Elements from a `Stream` can be easily collected into a container by using the `Stream.collect` operation:

```
System.out.println(Arrays
    .asList("apple", "banana", "pear", "kiwi", "orange")
    .stream()
    .filter(s -> s.contains("a"))
    .collect(Collectors.toList())
);
// prints: [apple, banana, pear, orange]
```

Other collection instances, such as a `Set`, can be made by using other `Collectors` built-in methods. For example, `Collectors.toSet()` collects the elements of a Stream into a `Set`.

Explicit control over the implementation of `List` or `Set`

According to documentation of `Collectors#toList()` and `Collectors#toSet()`, there are no guarantees on the type, mutability, serializability, or thread-safety of the `List` or `Set` returned.

For explicit control over the implementation to be returned, `Collectors#toCollection(Supplier)` can be used instead, where the given supplier returns a new and empty collection.

```
// syntax with method reference
```

```

System.out.println(strings
    .stream()
    .filter(s -> s != null && s.length() <= 3)
    .collect(Collectors.toCollection(ArrayList::new))
);

// 使用 lambda 的语法
System.out.println(strings
    .stream()
    .filter(s -> s != null && s.length() <= 3)
    .collect(Collectors.toCollection(() -> new LinkedHashSet<>())))
);

```

使用 toMap 收集元素

Collector 将元素累积到一个 Map 中，其中键是学生 ID，值是学生名称。

```

List<Student> students = new ArrayList<Student>();
students.add(new Student(1, "test1"));
students.add(new Student(2, "test2"));
students.add(new Student(3, "test3"));

Map<Integer, String> IdToName = students.stream()
    .collect(Collectors.toMap(Student::getId, Student::getName));
System.out.println(IdToName);

```

输出：

```
{1=test1, 2=test2, 3=test3}
```

Collectors.toMap 还有另一种实现 Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction)。mergeFunction 主要用于当从列表中向 Map 添加新成员时，如果键重复，选择保留新值或旧值。

mergeFunction 通常写成：(s1, s2) -> s1 用于保留对应重复键的旧值，或者(s1, s2) -> s2 用于替换重复键的新值。

收集元素到集合的映射

示例：从 ArrayList 到 Map<String, List<>>

通常需要从一个主列表生成一个列表的映射。例如：从学生列表中，我们需要为每个学生成一个科目列表的映射。

```

List<Student> list = new ArrayList<>();
list.add(new Student("戴维斯", SUBJECT.MATH, 35.0));
list.add(new Student("戴维斯", SUBJECT.SCIENCE, 12.9));
list.add(new Student("戴维斯", SUBJECT.GEOGRAPHY, 37.0));

list.add(new Student("萨沙", SUBJECT.ENGLISH, 85.0));
list.add(new Student("萨沙", SUBJECT.MATH, 80.0));
list.add(new Student("萨沙", SUBJECT.SCIENCE, 12.0));
list.add(new Student("萨沙", SUBJECT.LITERATURE, 50.0));

list.add(new Student("罗伯特", SUBJECT.LITERATURE, 12.0));

```

```

System.out.println(strings
    .stream()
    .filter(s -> s != null && s.length() <= 3)
    .collect(Collectors.toCollection(ArrayList::new))
);

// syntax with lambda
System.out.println(strings
    .stream()
    .filter(s -> s != null && s.length() <= 3)
    .collect(Collectors.toCollection(() -> new LinkedHashSet<>())))
);

```

Collecting Elements using toMap

Collector accumulates elements into a Map, Where key is the Student Id and Value is Student Value.

```

List<Student> students = new ArrayList<Student>();
students.add(new Student(1, "test1"));
students.add(new Student(2, "test2"));
students.add(new Student(3, "test3"));

Map<Integer, String> IdToName = students.stream()
    .collect(Collectors.toMap(Student::getId, Student::getName));
System.out.println(IdToName);

```

Output :

```
{1=test1, 2=test2, 3=test3}
```

The Collectors.toMap has another implementation Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper, BinaryOperator<U> mergeFunction).The mergeFunction is mostly used to select either new value or retain old value if the key is repeated when adding a new member in the Map from a list.

The mergeFunction often looks like: (s1, s2) -> s1 to retain value corresponding to the repeated key, or (s1, s2) -> s2 to put new value for the repeated key.

Collecting Elements to Map of Collections

Example: from ArrayList to Map<String, List<>>

Often it requires to make a map of list out of a primary list. Example: From a student of list, we need to make a map of list of subjects for each student.

```

List<Student> list = new ArrayList<>();
list.add(new Student("Davis", SUBJECT.MATH, 35.0));
list.add(new Student("Davis", SUBJECT.SCIENCE, 12.9));
list.add(new Student("Davis", SUBJECT.GEOGRAPHY, 37.0));

list.add(new Student("Sascha", SUBJECT.ENGLISH, 85.0));
list.add(new Student("Sascha", SUBJECT.MATH, 80.0));
list.add(new Student("Sascha", SUBJECT.SCIENCE, 12.0));
list.add(new Student("Sascha", SUBJECT.LITERATURE, 50.0));

list.add(new Student("Robert", SUBJECT.LITERATURE, 12.0));

```

```

Map<String, List<SUBJECT>> map = new HashMap<>();
list.stream().forEach(s -> {
map.computeIfAbsent(s.getName(), x -> new ArrayList<>()).add(s.getSubject());
});
System.out.println(map);

```

输出：

```

{ 罗伯特=[LITERATURE],
Sascha=[英语, 数学, 科学, 文学],
Davis=[数学, 科学, 地理] }

```

示例：从 ArrayList 到 Map<String, Map<>>

```

List<Student> list = new ArrayList<>();
list.add(new Student("Davis", SUBJECT.MATH, 1, 35.0));
list.add(new Student("Davis", SUBJECT.SCIENCE, 2, 12.9));
list.add(new Student("Davis", SUBJECT.MATH, 3, 37.0));
list.add(new Student("Davis", SUBJECT.SCIENCE, 4, 37.0));

list.add(new Student("Sascha", SUBJECT.ENGLISH, 5, 85.0));
list.add(new Student("Sascha", SUBJECT.MATH, 1, 80.0));
list.add(new Student("Sascha", SUBJECT.ENGLISH, 6, 12.0));
list.add(new Student("Sascha", SUBJECT.MATH, 3, 50.0));

list.add(new Student("Robert", SUBJECT.ENGLISH, 5, 12.0));

Map<String, Map<SUBJECT, List<Double>>> map = new HashMap<>();

list.stream().forEach(student -> {
map.computeIfAbsent(student.getName(), s -> new HashMap<>())
    .computeIfAbsent(student.getSubject(), s -> new ArrayList<>())
    .add(student.getMarks());
});

System.out.println(map);

```

输出：

```

{ Robert={英语=[12.0]},
Sascha={数学=[80.0, 50.0], 英语=[85.0, 12.0]},
Davis={数学=[35.0, 37.0], 科学=[12.9, 37.0]} }

```

备忘单

目标	代码
收集到一个列表	Collectors.toList()
收集到一个预分配大小的ArrayList	Collectors.toCollection(() -> new ArrayList<>(size))
收集到一个集合	Collectors.toSet()
收集到一个集合以获得更好的迭代性能	Collectors.toCollection(() -> new LinkedHashSet<>())
收集到一个不区分大小写的集合<字符串>	Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))
收集到一个枚举集合<某枚举> (枚举的最佳性能)	Collectors.toCollection(() -> EnumSet.noneOf(某枚举.class))

```

Map<String, List<SUBJECT>> map = new HashMap<>();
list.stream().forEach(s -> {
map.computeIfAbsent(s.getName(), x -> new ArrayList<>()).add(s.getSubject());
});
System.out.println(map);

```

Output:

```

{ Robert=[LITERATURE],
Sascha=[ENGLISH, MATH, SCIENCE, LITERATURE],
Davis=[MATH, SCIENCE, GEOGRAPHY] }

```

Example: from ArrayList to Map<String, Map<>>

```

List<Student> list = new ArrayList<>();
list.add(new Student("Davis", SUBJECT.MATH, 1, 35.0));
list.add(new Student("Davis", SUBJECT.SCIENCE, 2, 12.9));
list.add(new Student("Davis", SUBJECT.MATH, 3, 37.0));
list.add(new Student("Davis", SUBJECT.SCIENCE, 4, 37.0));

list.add(new Student("Sascha", SUBJECT.ENGLISH, 5, 85.0));
list.add(new Student("Sascha", SUBJECT.MATH, 1, 80.0));
list.add(new Student("Sascha", SUBJECT.ENGLISH, 6, 12.0));
list.add(new Student("Sascha", SUBJECT.MATH, 3, 50.0));

list.add(new Student("Robert", SUBJECT.ENGLISH, 5, 12.0));

Map<String, Map<SUBJECT, List<Double>>> map = new HashMap<>();

list.stream().forEach(student -> {
map.computeIfAbsent(student.getName(), s -> new HashMap<>())
    .computeIfAbsent(student.getSubject(), s -> new ArrayList<>())
    .add(student.getMarks());
});

System.out.println(map);

```

Output:

```

{ Robert={ENGLISH=[12.0]},
Sascha={MATH=[80.0, 50.0], ENGLISH=[85.0, 12.0]},
Davis={MATH=[35.0, 37.0], SCIENCE=[12.9, 37.0]} }

```

Cheat-Sheet

Goal	Code
Collect to a List	Collectors.toList()
Collect to an ArrayList with pre-allocated size	Collectors.toCollection(() -> new ArrayList<>(size))
Collect to a Set	Collectors.toSet()
Collect to a Set with better iteration performance	Collectors.toCollection(() -> new LinkedHashSet<>())
Collect to a case-insensitive Set<String>	Collectors.toCollection(() -> new TreeSet<>(String.CASE_INSENSITIVE_ORDER))
Collect to an EnumSet<AnEnum> (best performance for enums)	Collectors.toCollection(() -> EnumSet.noneOf(AnEnum.class))

收集到具有唯一键的Map<K,V>, 使用Collectors.toMap(keyFunc,valFunc)

将MyObject.getter()映射为唯一键

Collectors.toMap(MyObject::getter, Function.identity())

将 MyObject.getter() 映射到多个

MyObjects

.Collectors.groupingBy(MyObject::getter)

Collect to a Map<K, V> with unique keys Collectors.toMap(keyFunc, valFunc)

Map MyObject.getter() to unique

MyObject

Map MyObject.getter() to multiple

MyObjects

Collectors.toMap(MyObject::getter, Function.identity())

.Collectors.groupingBy(MyObject::getter)

第57.6节：使用流实现数学函数

Stream, 尤其是IntStream, 是实现求和项 (Σ) 的优雅方式。Stream的范围可以用作求和的边界。

例如, Madhava 对圆周率的近似由公式给出 (来源 : wikipedia) :

$$\pi = \sqrt{12} \sum_{k=0}^{\infty} \frac{(-3)^{-k}}{2k+1} = \sqrt{12} \sum_{k=0}^{\infty} \frac{(-\frac{1}{3})^k}{2k+1} = \sqrt{12} \left(\frac{1}{1 \cdot 3^0} - \frac{1}{3 \cdot 3^1} + \frac{1}{5 \cdot 3^2} - \frac{1}{7 \cdot 3^3} + \dots \right)$$

这可以用任意精度计算。例如, 对于101项:

```
double pi = Math.sqrt(12) *  
IntStream.rangeClosed(0, 100)  
    .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))  
    .sum();
```

注意: 使用 double 精度时, 选择上限为 29 即足以获得与 Math.Pi 无法区分的结果。

Section 57.6: Using Streams to Implement Mathematical Functions

Streams, and especially IntStreams, are an elegant way of implementing summation terms (Σ). The ranges of the Stream can be used as the bounds of the summation.

E.g., Madhava's approximation of Pi is given by the formula (Source: [wikipedia](#)):

$$\pi = \sqrt{12} \sum_{k=0}^{\infty} \frac{(-3)^{-k}}{2k+1} = \sqrt{12} \sum_{k=0}^{\infty} \frac{(-\frac{1}{3})^k}{2k+1} = \sqrt{12} \left(\frac{1}{1 \cdot 3^0} - \frac{1}{3 \cdot 3^1} + \frac{1}{5 \cdot 3^2} - \frac{1}{7 \cdot 3^3} + \dots \right)$$

This can be calculated with an arbitrary precision. E.g., for 101 terms:

```
double pi = Math.sqrt(12) *  
IntStream.rangeClosed(0, 100)  
    .mapToDouble(k -> Math.pow(-3, -1 * k) / (2 * k + 1))  
    .sum();
```

Note: With double's precision, selecting an upper bound of 29 is sufficient to get a result that's indistinguishable from Math.Pi.

第 57.7 节：使用 flatMap() 扁平化流

一个包含可流化项目的流 (Stream) 可以被扁平化为一个连续的流 (Stream) :

项目列表数组可以转换为单个列表。

```
List<String> list1 = Arrays.asList("one", "two");  
List<String> list2 = Arrays.asList("three", "four", "five");  
List<String> list3 = Arrays.asList("six");  
List<String> finalList = Stream.of(list1, list2,  
list3).flatMap(Collection::stream).collect(Collectors.toList());  
System.out.println(finalList);  
  
// [one, two, three, four, five, six]
```

包含项目列表作为值的映射 (Map) 可以被扁平化为一个合并列表

```
Map<String, List<Integer>> map = new LinkedHashMap<>();  
map.put("a", Arrays.asList(1, 2, 3));  
map.put("b", Arrays.asList(4, 5, 6));  
  
List<Integer> allValues = map.values() // Collection<List<Integer>>  
    .stream() // Stream<List<Integer>>  
    .flatMap(List::stream) // Stream<Integer>  
    .collect(Collectors.toList());  
  
System.out.println(allValues);  
// [1, 2, 3, 4, 5, 6]
```

List的Map可以被展平成一个连续的Stream

Section 57.7: Flatten Streams with flatMap()

A Stream of items that are in turn streamable can be flattened into a single continuous Stream:

Array of List of Items can be converted into a single List.

```
List<String> list1 = Arrays.asList("one", "two");  
List<String> list2 = Arrays.asList("three", "four", "five");  
List<String> list3 = Arrays.asList("six");  
List<String> finalList = Stream.of(list1, list2,  
list3).flatMap(Collection::stream).collect(Collectors.toList());  
System.out.println(finalList);  
  
// [one, two, three, four, five, six]
```

Map containing List of Items as values can be Flattened to a Combined List

```
Map<String, List<Integer>> map = new LinkedHashMap<>();  
map.put("a", Arrays.asList(1, 2, 3));  
map.put("b", Arrays.asList(4, 5, 6));  
  
List<Integer> allValues = map.values() // Collection<List<Integer>>  
    .stream() // Stream<List<Integer>>  
    .flatMap(List::stream) // Stream<Integer>  
    .collect(Collectors.toList());  
  
System.out.println(allValues);  
// [1, 2, 3, 4, 5, 6]
```

List of Map can be flattened into a single continuous Stream

```

List<Map<String, String>> list = new ArrayList<>();
Map<String, String> map1 = new HashMap();
map1.put("1", "one");
map1.put("2", "two");

Map<String, String> map2 = new HashMap();
map2.put("3", "three");
map2.put("4", "four");
list.add(map1);
list.add(map2);

Set<String> output= list.stream() // Stream<Map<String, String>>
    .map(Map::values)           // Stream<List<String>>
    .flatMap(Collection::stream) // Stream<String>
    .collect(Collectors.toSet()); // Set<String>
// [one, two, three, four]

```

第57.8节：并行流

注意：在决定使用哪种Stream之前，请先查看并行流与顺序流的行为对比。

当你想要并发执行Stream操作时，可以使用以下任一方式。

```

List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();

```

或者：

```
Stream<String> aParallelStream = data.parallelStream();
```

要执行为并行流定义的操作，调用终端操作符：

```
aParallelStream.forEach(System.out::println);
```

并行Stream的（可能的）输出：

```

Three
Four
One
Two
Five

```

由于所有元素是并行处理的，顺序可能会改变（这可能使其更快）。使用 `parallelStream` 当顺序不重要时。

性能影响

如果涉及网络，`parallelStream` 可能会降低应用程序的整体性能，因为所有的 `parallelStream` 都使用网络的公共 fork-join 线程池。

另一方面，`parallelStream` 在许多其他情况下可能显著提高性能，这取决于运行时 CPU 中可用核心的数量。

```

List<Map<String, String>> list = new ArrayList<>();
Map<String, String> map1 = new HashMap();
map1.put("1", "one");
map1.put("2", "two");

Map<String, String> map2 = new HashMap();
map2.put("3", "three");
map2.put("4", "four");
list.add(map1);
list.add(map2);

```

```

Set<String> output= list.stream() // Stream<Map<String, String>>
    .map(Map::values)           // Stream<List<String>>
    .flatMap(Collection::stream) // Stream<String>
    .collect(Collectors.toSet()); // Set<String>
// [one, two, three, four]

```

Section 57.8: Parallel Stream

Note: Before deciding which Stream to use please have a look at [ParallelStream vs Sequential Stream behavior](#).

When you want to perform stream operations concurrently, you could use either of these ways.

```

List<String> data = Arrays.asList("One", "Two", "Three", "Four", "Five");
Stream<String> aParallelStream = data.stream().parallel();

```

Or:

```
Stream<String> aParallelStream = data.parallelStream();
```

To execute the operations defined for the parallel stream, call a terminal operator:

```
aParallelStream.forEach(System.out::println);
```

(A possible) output from the parallel Stream:

```

Three
Four
One
Two
Five

```

The order might change as all the elements are processed in parallel (Which *may* make it faster). Use `parallelStream` when ordering does not matter.

Performance impact

In case networking is involved, parallel Streams may degrade the overall performance of an application because all parallel Streams use a common fork-join thread pool for the network.

On the other hand, parallel Streams may significantly improve performance in many other cases, depending on the number of available cores in the running CPU at the moment.

第57.9节：创建流

所有javaCollection<E>都有stream()和parallelStream()方法，可以从中构造Stream<E>：

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

可以使用以下两种方法之一从数组创建Stream<E>：

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

Arrays.stream()和Stream.of()的区别在于Stream.of()有一个可变参数，因此可以像这样使用：

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

你也可以使用原始类型的Stream。例如：

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

这些原始类型流也可以使用Arrays.stream()方法构造：

```
IntStream intStream = Arrays.stream(new int[]{ 1, 2, 3 });
```

可以从数组中创建指定范围的Stream。

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStram = Arrays.stream(values, 1, 3);
```

注意，任何原始类型流都可以使用boxed方法转换为装箱类型流：

```
Stream<Integer> integerStream = intStream.boxed();
```

如果你想收集数据，这可能很有用，因为原始类型流没有接受Collector作为参数的collect方法。

重用流链的中间操作

每当调用终端操作时，流就会关闭。当只有终端操作变化时，重用中间操作的流。我们可以创建一个流供应器来构造一个新的流，所有中间操作都已设置好。

```
Supplier<Stream<String>> streamSupplier = () -> Stream.of("apple", "banana", "orange", "grapes",
"melon", "blueberry", "blackberry")
.map(String::toUpperCase).sorted();

streamSupplier.get().filter(s -> s.startsWith("A")).forEach(System.out::println);

// APPLE

streamSupplier.get().filter(s -> s.startsWith("B")).forEach(System.out::println);
```

Section 57.9: Creating a Stream

All java Collection<E>s have [stream\(\)](#) and [parallelStream\(\)](#) methods from which a Stream<E> can be constructed:

```
Collection<String> stringList = new ArrayList<>();
Stream<String> stringStream = stringList.parallelStream();
```

A Stream<E> can be created from an array using one of two methods:

```
String[] values = { "aaa", "bbbb", "ddd", "cccc" };
Stream<String> stringStream = Arrays.stream(values);
Stream<String> stringStreamAlternative = Stream.of(values);
```

The difference between [Arrays.stream\(\)](#) and [Stream.of\(\)](#) is that Stream.of() has a varargs parameter, so it can be used like:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3);
```

There are also primitive Streams that you can use. For example:

```
IntStream intStream = IntStream.of(1, 2, 3);
DoubleStream doubleStream = DoubleStream.of(1.0, 2.0, 3.0);
```

These primitive streams can also be constructed using the [Arrays.stream\(\)](#) method:

```
IntStream intStream = Arrays.stream(new int[]{ 1, 2, 3 });
```

It is possible to create a Stream from an array with a specified range.

```
int[] values= new int[]{1, 2, 3, 4, 5};
IntStream intStram = Arrays.stream(values, 1, 3);
```

Note that any primitive stream can be converted to boxed type stream using the boxed method :

```
Stream<Integer> integerStream = intStream.boxed();
```

This can be useful in some case if you want to collect the data since primitive stream does not have any collect method that takes a Collector as argument.

Reusing intermediate operations of a stream chain

Stream is closed when ever terminal operation is called. Reusing the stream of intermediate operations, when only terminal operation is only varying. we could create a stream supplier to construct a new stream with all intermediate operations already set up.

```
Supplier<Stream<String>> streamSupplier = () -> Stream.of("apple", "banana", "orange", "grapes",
"melon", "blueberry", "blackberry")
.map(String::toUpperCase).sorted();

streamSupplier.get().filter(s -> s.startsWith("A")).forEach(System.out::println);

// APPLE

streamSupplier.get().filter(s -> s.startsWith("B")).forEach(System.out::println);
```

```
// BANANA  
// BLACKBERRY  
// BLUEBERRY
```

int[] 数组可以通过流转换为 List<Integer>

```
int[] ints = {1,2,3};  
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

第57.10节：查找数值流的统计信息

Java 8 提供了名为 [IntSummaryStatistics](#)、[DoubleSummaryStatistics](#) 和 [LongSummaryStatistics](#) 的类，它们提供了一个状态对象，用于收集诸如 count、min、max、sum 和 average 等统计信息。

版本 ≥ Java SE 8

```
List<Integer> naturalNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
IntSummaryStatistics stats = naturalNumbers.stream()  
.mapToInt((x) -> x)  
.summaryStatistics();  
System.out.println(stats);
```

结果将是：

版本 ≥ Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

第57.11节：将迭代器转换为流

使用 [Spliterators.spliterator\(\)](#) 或 [Spliterators.spliteratorUnknownSize\(\)](#) 将迭代器转换为流：

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();  
Spliterator<String> spliterator = Spliterators.spliteratorUnknownSize(iterator, 0);  
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

第57.12节：使用IntStream遍历索引

元素流通常不允许访问当前项的索引值。要遍历数组或 [ArrayList](#) 同时访问索引，请使用 [IntStream.range\(start, endExclusive\)](#)。

```
String[] names = { "Jon", "Darin", "Bauke", "Hans", "Marc" };  
  
IntStream.range(0, names.length)  
.mapToObj(i -> String.format("#%d %s", i + 1, names[i]))  
.forEach(System.out::println);
```

[range\(start, endExclusive\)](#) 方法返回另一个 [IntStream](#)，[mapToObj\(mapper\)](#) 返回一个 [String](#) 流。

输出：

```
#1 乔恩  
#2 达林  
#3 鲍克  
#4 汉斯
```

```
// BANANA  
// BLACKBERRY  
// BLUEBERRY
```

[int\[\]](#) 数组可以通过流转换为 List<Integer> 使用 streams

```
int[] ints = {1,2,3};  
List<Integer> list = IntStream.of(ints).boxed().collect(Collectors.toList());
```

Section 57.10: Finding Statistics about Numerical Streams

Java 8 provides classes called [IntSummaryStatistics](#), [DoubleSummaryStatistics](#) and [LongSummaryStatistics](#) which give a state object for collecting statistics such as count, min, max, sum, and average.

Version ≥ Java SE 8

```
List<Integer> naturalNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
IntSummaryStatistics stats = naturalNumbers.stream()  
.mapToInt((x) -> x)  
.summaryStatistics();  
System.out.println(stats);
```

Which will result in:

Version ≥ Java SE 8

```
IntSummaryStatistics{count=10, sum=55, min=1, max=10, average=5.500000}
```

Section 57.11: Converting an iterator to a stream

Use [Spliterators.spliterator\(\)](#) or [Spliterators.spliteratorUnknownSize\(\)](#) to convert an iterator to a stream:

```
Iterator<String> iterator = Arrays.asList("A", "B", "C").iterator();  
Spliterator<String> spliterator = Spliterators.spliteratorUnknownSize(iterator, 0);  
Stream<String> stream = StreamSupport.stream(spliterator, false);
```

Section 57.12: Using IntStream to iterate over indexes

Streams of elements usually do not allow access to the index value of the current item. To iterate over an array or [ArrayList](#) while having access to indexes, use [IntStream.range\(start, endExclusive\)](#).

```
String[] names = { "Jon", "Darin", "Bauke", "Hans", "Marc" };  
  
IntStream.range(0, names.length)  
.mapToObj(i -> String.format("#%d %s", i + 1, names[i]))  
.forEach(System.out::println);
```

The [range\(start, endExclusive\)](#) method returns another [IntStream](#) and the [mapToObj\(mapper\)](#) returns a stream of [String](#).

Output:

```
#1 Jon  
#2 Darin  
#3 Bauke  
#4 Hans
```

这与使用带计数器的普通for循环非常相似，但具有流水线和并行化的优势：

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
    System.out.println(newName);
}
```

This is very similar to using a normal **for** loop with a counter, but with the benefit of pipelining and parallelization:

```
for (int i = 0; i < names.length; i++) {
    String newName = String.format("#%d %s", i + 1, names[i]);
    System.out.println(newName);
}
```

第57.13节：连接流

示例的变量声明：

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

示例1 - 连接两个Stream

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());
concat1.forEach(System.out::print);
// 输出: abc123
```

示例 2 - 连接超过两个Stream

```
final Stream<String> concat2 = Stream.concat(
    Stream.concat(abc.stream(), digits.stream()),
    greekAbc.stream());
System.out.println(concat2.collect(Collectors.joining(", ")));
// 输出: a, b, c, 1, 2, 3, alpha, beta, gamma
```

为了简化嵌套的concat()语法，Stream也可以通过flatMap()进行连接：

```
final Stream<String> concat3 = Stream.of(
    abc.stream(), digits.stream(), greekAbc.stream())
    .flatMap(s -> s);
// 或者使用 `flatMap(Function.identity());` (java.util.function.Function)
System.out.println(concat3.collect(Collectors.joining(", ")));
// 输出: a, b, c, 1, 2, 3, alpha, beta, gamma
```

在通过重复连接构造Stream时要小心，因为访问深度连接的Stream的元素可能导致深层调用链，甚至StackOverflowException。

第57.14节：使用流进行归约

归约是将二元运算符应用于流中的每个元素，以得到一个值的过程。

IntStream 的 sum() 方法就是归约的一个例子；它对流中的每一项进行加法运算，最终得到一个值：

Section 57.13: Concatenate Streams

Variable declaration for examples:

```
Collection<String> abc = Arrays.asList("a", "b", "c");
Collection<String> digits = Arrays.asList("1", "2", "3");
Collection<String> greekAbc = Arrays.asList("alpha", "beta", "gamma");
```

Example 1 - Concatenate two Streams

```
final Stream<String> concat1 = Stream.concat(abc.stream(), digits.stream());
concat1.forEach(System.out::print);
// prints: abc123
```

Example 2 - Concatenate more than two Streams

```
final Stream<String> concat2 = Stream.concat(
    Stream.concat(abc.stream(), digits.stream()),
    greekAbc.stream());
System.out.println(concat2.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

Alternatively to simplify the nested concat() syntax the Streams can also be concatenated with flatMap():

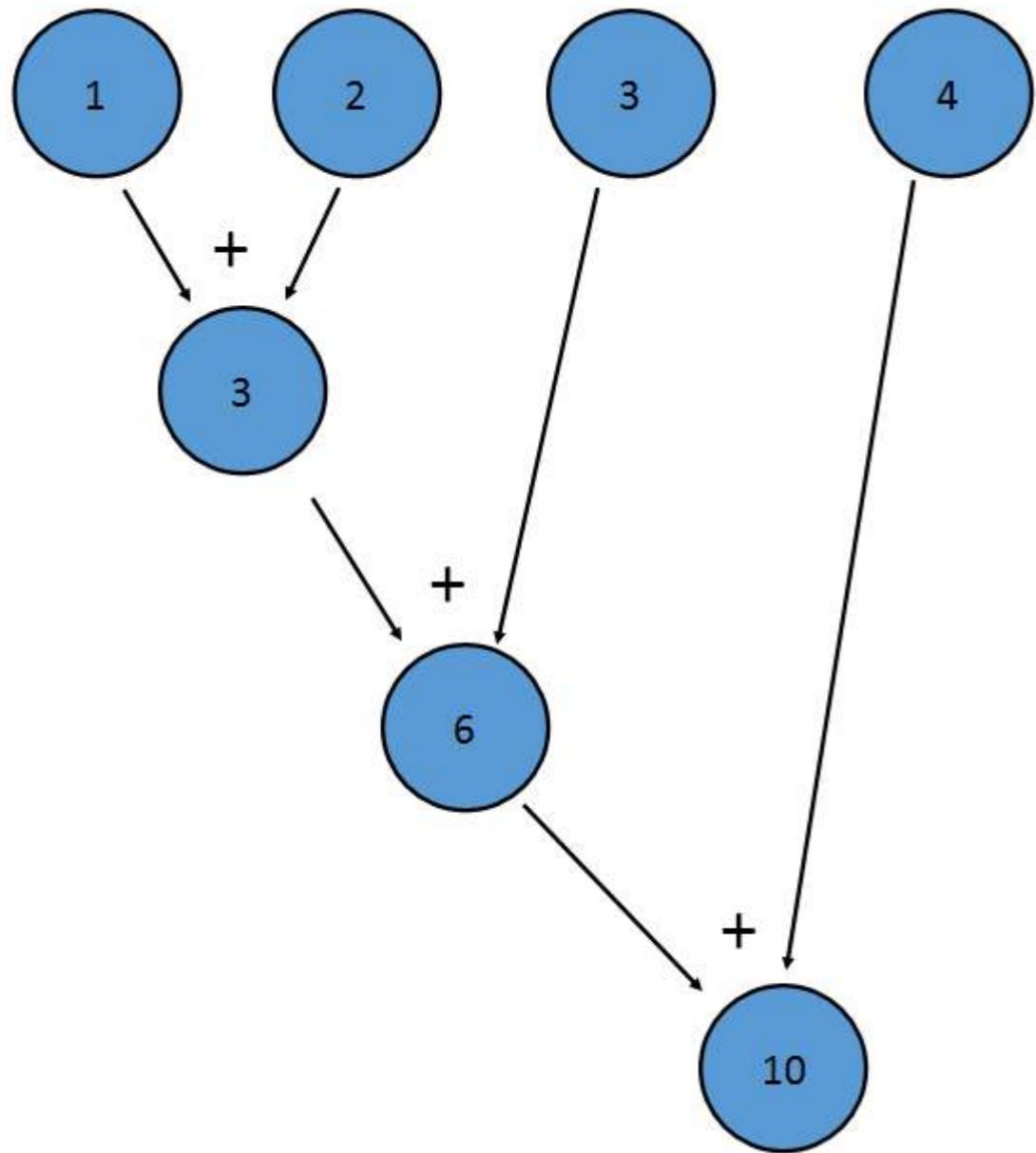
```
final Stream<String> concat3 = Stream.of(
    abc.stream(), digits.stream(), greekAbc.stream())
    .flatMap(s -> s);
// or `flatMap(Function.identity());` (java.util.function.Function)
System.out.println(concat3.collect(Collectors.joining(", ")));
// prints: a, b, c, 1, 2, 3, alpha, beta, gamma
```

Be careful when constructing Streams from repeated concatenation, because accessing an element of a deeply concatenated Stream can result in deep call chains or even a StackOverflowException.

Section 57.14: Reduction with Streams

Reduction is the process of applying a binary operator to every element of a stream to result in one value.

The sum() method of an IntStream is an example of a reduction; it applies addition to every term of the Stream, resulting in one final value:



这等同于 $((1+2)+3)+4$

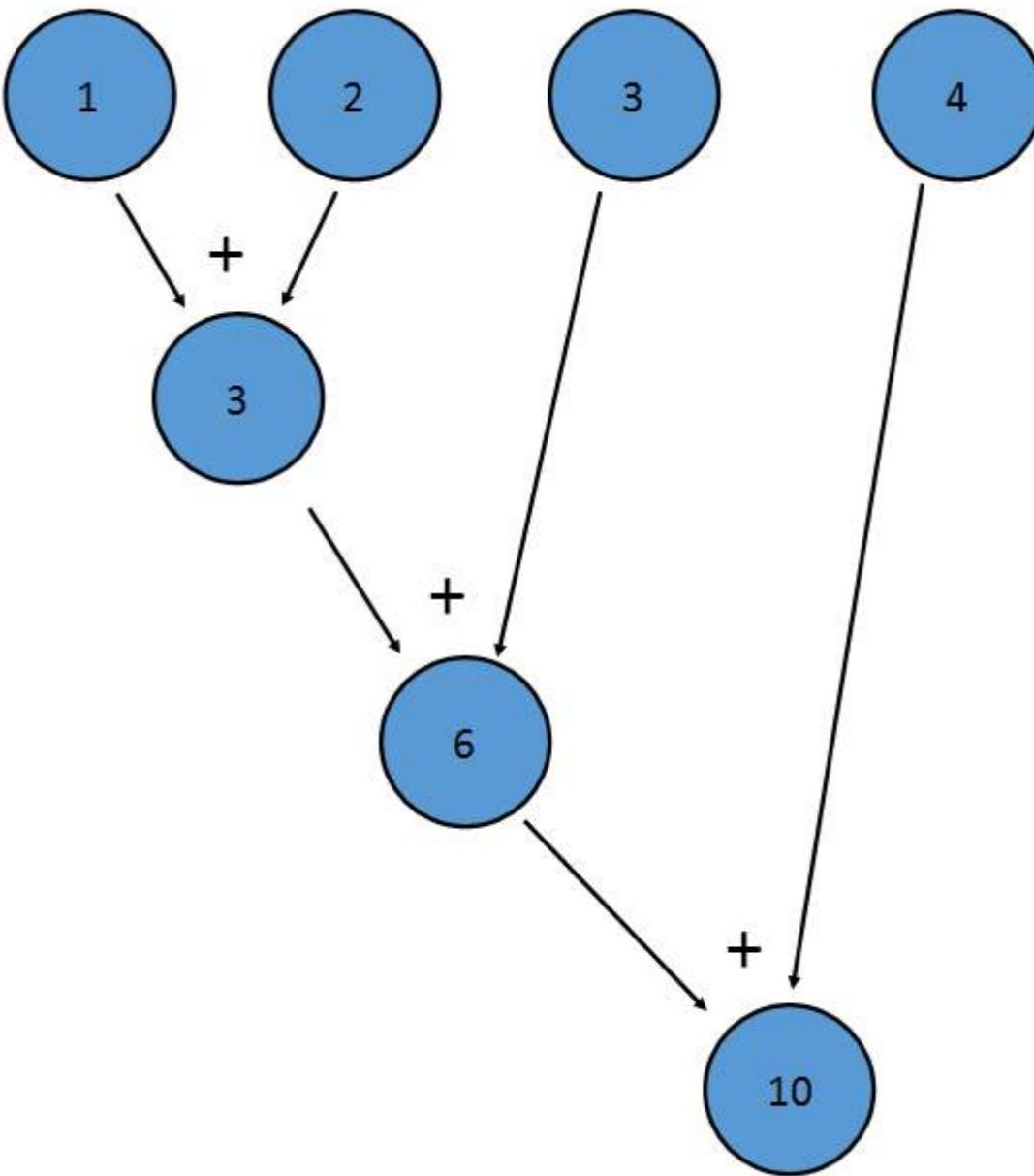
Stream 的 reduce 方法允许创建自定义归约。可以使用 reduce 方法来实现 sum() 方法：

```
IntStream istr;
//初始化 istr
OptionalInt istr.reduce((a,b)->a+b);
```

返回 Optional 版本，以便能够适当处理空流。

另一个归约的例子是将一个 Stream<LinkedList<T>> 合并成一个单一的 LinkedList<T>：

```
Stream<LinkedList<T>> listStream;
//创建一个 Stream<LinkedList<T>>
```



This is equivalent to $((1+2)+3)+4$

The reduce method of a Stream allows one to create a custom reduction. It is possible to use the reduce method to implement the sum() method:

```
IntStream istr;
//Initialize istr
OptionalInt istr.reduce((a,b)->a+b);
```

The Optional version is returned so that empty Streams can be handled appropriately.

Another example of reduction is combining a Stream<LinkedList<T>> into a single LinkedList<T>:

```
Stream<LinkedList<T>> listStream;
//Create a Stream<LinkedList<T>>
```

```
Optional<LinkedList<T>> bigList = listStream.reduce((LinkedList<T> list1, LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

你也可以提供一个 *identity element* (单位元)。例如，加法的单位元是0，因为 $x+0==x$ 。乘法的单位元是1，因为 $x*1==x$ 。上述情况下，单位元是一个空的 `LinkedList<T>`，因为如果你将一个空列表加到另一个列表上，被“加”的列表不会改变：

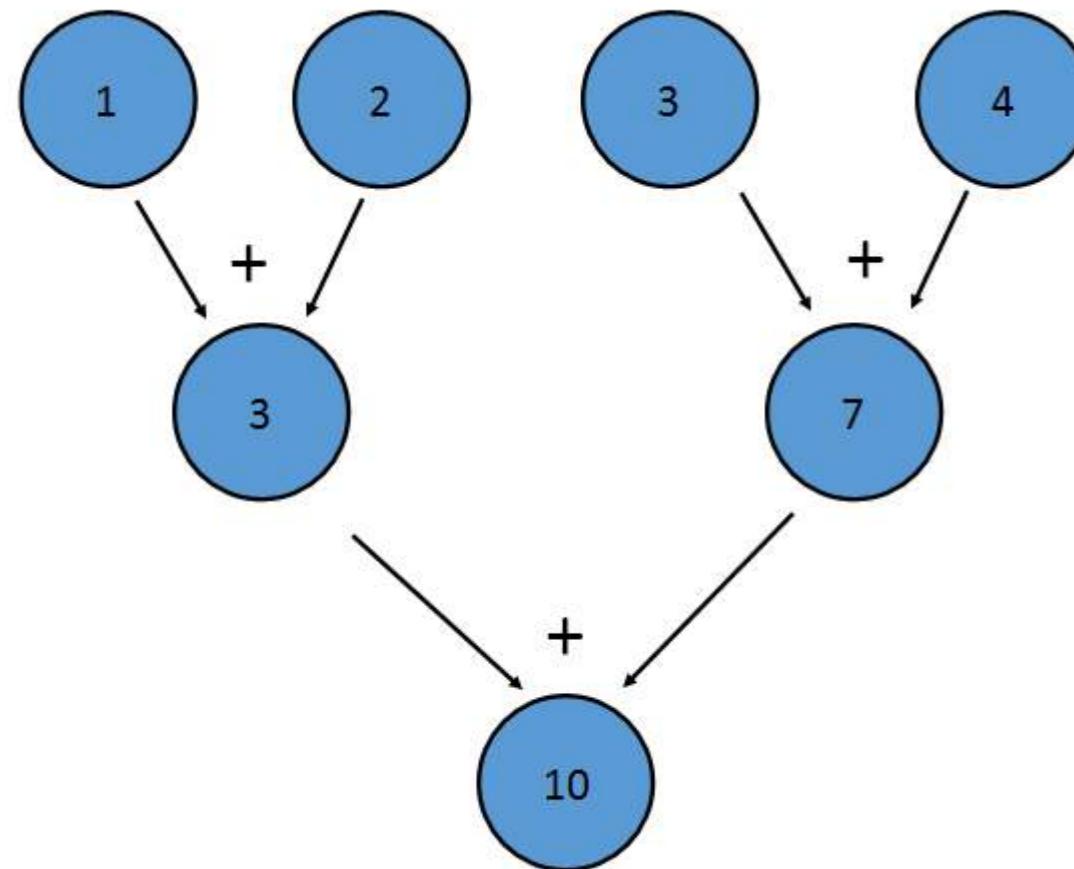
```
Stream<LinkedList<T>> listStream;

//创建一个 Stream<LinkedList<T>>

LinkedList<T> bigList = listStream.reduce(new LinkedList<T>(), (LinkedList<T> list1, LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

请注意，当提供了单位元素时，返回值不会被包装在 `Optional` 中——如果在空流上调用，`reduce()` 将返回单位元素。

二元操作符还必须是结合律的，意味着 $(a+b)+c==a+(b+c)$ 。因为元素可以以任意顺序进行归约。例如，上述加法归约可以这样执行：



```
Optional<LinkedList<T>> bigList = listStream.reduce((LinkedList<T> list1, LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

You can also provide an *identity element*. For example, the identity element for addition is 0, as $x+0==x$. For multiplication, the identity element is 1, as $x*1==x$. In the case above, the identity element is an empty `LinkedList<T>`, because if you add an empty list to another list, the list that you are "adding" to doesn't change:

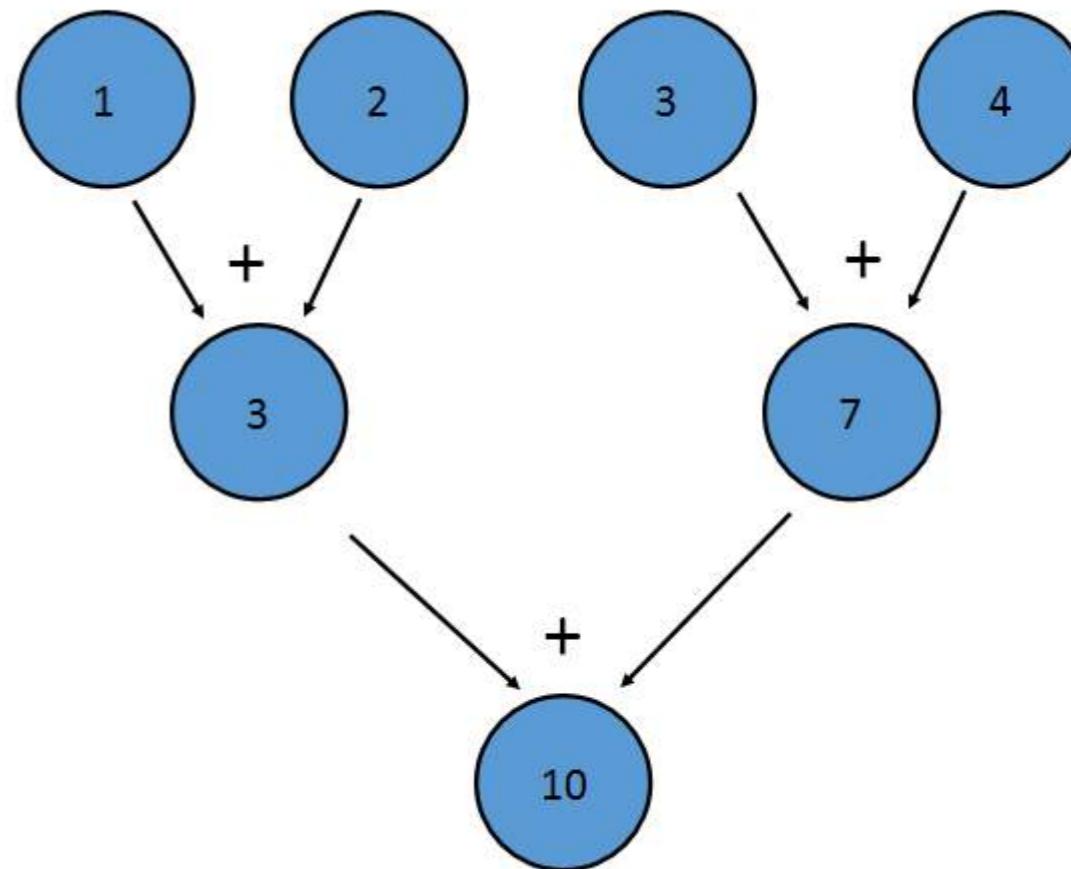
```
Stream<LinkedList<T>> listStream;

//Create a Stream<LinkedList<T>>

LinkedList<T> bigList = listStream.reduce(new LinkedList<T>(), (LinkedList<T> list1, LinkedList<T> list2)->{
    LinkedList<T> retList = new LinkedList<T>();
    retList.addAll(list1);
    retList.addAll(list2);
    return retList;
});
```

Note that when an identity element is provided, the return value is not wrapped in an `Optional`—if called on an empty stream, `reduce()` will return the identity element.

The binary operator must also be *associative*, meaning that $(a+b)+c==a+(b+c)$. This is because the elements may be reduced in any order. For example, the above addition reduction could be performed like this:



此归约等价于写作 $((1+2)+(3+4))$ 。结合律的性质还允许Java并行归约Stream——流的一部分可以由每个处理器归约，最后将每个处理器的结果合并。

第57.15节：使用Map.Entry流在映射后保留初始值

当你有一个Stream需要映射但又想保留初始值时，可以使用如下实用方法将Stream映射为Map.Entry<K,V>：

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper) {
    return (k) -> new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

然后你可以使用转换器处理Stream，同时访问原始值和映射值：

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
.map(entryMapper(transformer));
```

然后你可以继续像处理普通流那样处理该Stream。这避免了创建中间集合的开销。

第57.16节：IntStream 转 String

Java 没有字符流(Char Stream)，因此在处理字符串(String)并构建字符(Character)的流(Stream)时，一个选项是使用String的codePoints()方法获取代码点的IntStream。这样可以如下获得IntStream：

```
public IntStream stringToIntStream(String in) {
    return in.codePoints();
}
```

反向转换，即 IntStream 转 String，则稍微复杂一些。可以按如下方式实现：

```
public String intStreamToString(IntStream intStream) {
    return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
        StringBuilder::append).toString();
}
```

第57.17节：查找第一个匹配谓词的元素

可以查找流(Stream)中第一个满足条件的元素。

在此示例中，我们将查找第一个平方大于50000的整数(Integer)。

```
IntStream.iterate(1, i -> i + 1) // 生成无限流 1,2,3,4...
.filter(i -> (i*i) > 50000) // 过滤以查找平方大于50000的元素
.findFirst(); // 查找第一个符合条件的元素
```

该表达式将返回一个带有结果的OptionalInt。

This reduction is equivalent to writing $((1+2)+(3+4))$. The property of associativity also allows Java to reduce the stream in parallel—a portion of the Stream can be reduced by each processor, with a reduction combining the result of each processor at the end.

Section 57.15: Using Streams of Map.Entry to Preserve Initial Values after Mapping

When you have a Stream you need to map but want to preserve the initial values as well, you can map the Stream to a `Map.Entry<K, V>` using a utility method like the following:

```
public static <K, V> Function<K, Map.Entry<K, V>> entryMapper(Function<K, V> mapper) {
    return (k) -> new AbstractMap.SimpleEntry<>(k, mapper.apply(k));
}
```

Then you can use your converter to process Streams having access to both the original and mapped values:

```
Set<K> mySet;
Function<K, V> transformer = SomeClass::transformerMethod;
Stream<Map.Entry<K, V>> entryStream = mySet.stream()
.map(entryMapper(transformer));
```

You can then continue to process that Stream as normal. This avoids the overhead of creating an intermediate collection.

Section 57.16: IntStream to String

Java does not have a `Char Stream`, so when working with `String`s and constructing a Stream of `Characters`, an option is to get a `IntStream` of code points using `String.codePoints()` method. So `IntStream` can be obtained as below:

```
public IntStream stringToIntStream(String in) {
    return in.codePoints();
}
```

It is a bit more involved to do the conversion other way around i.e. `IntStreamToString`. That can be done as follows:

```
public String intStreamToString(IntStream intStream) {
    return intStream.collect(StringBuilder::new, StringBuilder::appendCodePoint,
        StringBuilder::append).toString();
}
```

Section 57.17: Finding the First Element that Matches a Predicate

It is possible to find the first element of a Stream that matches a condition.

For this example, we will find the first `Integer` whose square is over `50000`.

```
IntStream.iterate(1, i -> i + 1) // Generate an infinite stream 1,2,3,4...
.filter(i -> (i*i) > 50000) // Filter to find elements where the square is >50000
.findFirst(); // Find the first filtered element
```

This expression will return an `OptionalInt` with the result.

请注意，对于无限的Stream，Java会持续检查每个元素直到找到结果。对于有限的Stream，如果Java用尽元素但仍未找到结果，则返回一个空的OptionalInt。

第57.18节：使用流和方法引用编写自我说明的过程

方法引用是极好的自说明代码，使用方法引用与Stream结合使复杂的过程变得易于阅读和理解。考虑以下代码：

```
public interface Ordered {
    默认 int getOrder(){
        返回 0;
    }
}

public interface Valued<V extends Ordered> {
    boolean hasPropertyTwo();
    V getValue();
}

public interface Thing<V extends Ordered> {
    boolean hasPropertyOne();
    Valued<V> getValuedProperty();
}

public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    List<V> results = new ArrayList<V>();
    for (Thing<V> thing : things) {
        if (thing.hasPropertyOne()) {
            Valued<V> valued = thing.getValuedProperty();
            if (valued != null && valued.hasPropertyTwo()){
                V value = valued.getValue();
                if (value != null){
                    results.add(value);
                }
            }
        }
    }
    results.sort((a, b)->{
        return Integer.compare(a.getOrder(), b.getOrder());
   });
    return results;
}
```

最后这个方法使用Stream和方法引用重写后更加易读，过程中的每一步都能快速且轻松地理解——它不仅更简短，还能一目了然地显示每一步代码所对应的接口和类：

```
public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    return things.stream()
        .filter(Thing::hasPropertyOne)
        .map(Thing::getValuedProperty)
        .filter(Objects::nonNull)
        .filter(Valued::hasPropertyTwo)
        .map(Valued::getValue)
        .filter(Objects::nonNull)
        .sorted(Comparator.comparing(Ordered::getOrder))
        .collect(Collectors.toList());
}
```

Note that with an infinite Stream, Java will keep checking each element until it finds a result. With a finite Stream, if Java runs out of elements but still can't find a result, it returns an empty OptionalInt.

Section 57.18: Using Streams and Method References to Write Self-Documenting Processes

Method references make excellent self-documenting code, and using method references with Streams makes complicated processes simple to read and understand. Consider the following code:

```
public interface Ordered {
    default int getOrder(){
        return 0;
    }
}

public interface Valued<V extends Ordered> {
    boolean hasPropertyTwo();
    V getValue();
}

public interface Thing<V extends Ordered> {
    boolean hasPropertyOne();
    Valued<V> getValuedProperty();
}

public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    List<V> results = new ArrayList<V>();
    for (Thing<V> thing : things) {
        if (thing.hasPropertyOne()) {
            Valued<V> valued = thing.getValuedProperty();
            if (valued != null && valued.hasPropertyTwo()){
                V value = valued.getValue();
                if (value != null){
                    results.add(value);
                }
            }
        }
    }
    results.sort((a, b)->{
        return Integer.compare(a.getOrder(), b.getOrder());
   });
    return results;
}
```

This last method rewritten using Streams and method references is much more legible and each step of the process is quickly and easily understood - it's not just shorter, it also shows at a glance which interfaces and classes are responsible for the code in each step:

```
public <V extends Ordered> List<V> myMethod(List<Thing<V>> things) {
    return things.stream()
        .filter(Thing::hasPropertyOne)
        .map(Thing::getValuedProperty)
        .filter(Objects::nonNull)
        .filter(Valued::hasPropertyTwo)
        .map(Valued::getValue)
        .filter(Objects::nonNull)
        .sorted(Comparator.comparing(Ordered::getOrder))
        .collect(Collectors.toList());
}
```

第57.19节：将Optional流转换为值流

您可能需要将发出Optional的Stream转换为只发出存在值的Stream（即：不包含null值且不处理Optional.empty()）的值流。

```
Optional<String> op1 = Optional.empty();
Optional<String> op2 = Optional.of("Hello World");

List<String> result = Stream.of(op1, op2)
    .filter(Optional::isPresent)
    .map(Optional::get)
    .collect(Collectors.toList());

System.out.println(result); // [Hello World]
```

第57.20节：获取流的切片

示例：获取一个包含集合中第21到第50个（含）元素的30个元素的Stream。

```
final long n = 20L; // 要跳过的元素数量
final long maxSize = 30L; // 流应限制的元素数量
final Stream<T> slice = collection.stream().skip(n).limit(maxSize);
```

备注：

- 如果 n 为负数或 maxSize 为负数，则抛出 IllegalArgumentException
- skip(long) 和 limit(long) 都是中间操作
- 如果流中元素少于 n 个，则 skip(n) 返回一个空流skip(long) 和 limit(long) 在顺序
- 流管道上是廉价操作，但在有序并行管道上可能相当昂贵

第57.21节：基于流创建映射

无重复键的简单情况

```
Stream<String> characters = Stream.of("A", "B", "C");

Map<Integer, String> map = characters
    .collect(Collectors.toMap(element -> element.hashCode(), element -> element));
// map = {65=A, 66=B, 67=C}
```

为了使表达式更声明式，我们可以使用 Function 接口中的静态方法 - Function.identity()。我们可以用 Function.identity() 替换这个 lambda 表达式 element -> element。

可能存在重复键的情况

Collectors.toMap 的 javadoc 中说明：

如果映射的键包含重复项（根据Object.equals(Object)判断），在执行集合操作时会抛出IllegalStateException异常。如果映射的键可能有重复项，请改用toMap(Function, Function, BinaryOperator)。

Section 57.19: Converting a Stream of Optional to a Stream of Values

You may need to convert a Stream emitting Optional to a Stream of values, emitting only values from existing Optional. (ie: without `null` value and not dealing with `Optional.empty()`).

```
Optional<String> op1 = Optional.empty();
Optional<String> op2 = Optional.of("Hello World");

List<String> result = Stream.of(op1, op2)
    .filter(Optional::isPresent)
    .map(Optional::get)
    .collect(Collectors.toList());

System.out.println(result); // [Hello World]
```

Section 57.20: Get a Slice of a Stream

Example: Get a Stream of 30 elements, containing 21st to 50th (inclusive) element of a collection.

```
final long n = 20L; // the number of elements to skip
final long maxSize = 30L; // the number of elements the stream should be limited to
final Stream<T> slice = collection.stream().skip(n).limit(maxSize);
```

Notes:

- `IllegalArgumentException` is thrown if n is negative or maxSize is negative
- both `skip(long)` and `limit(long)` are intermediate operations
- if a stream contains fewer than n elements then `skip(n)` returns an empty stream
- both `skip(long)` and `limit(long)` are cheap operations on sequential stream pipelines, but can be quite expensive on ordered parallel pipelines

Section 57.21: Create a Map based on a Stream

Simple case without duplicate keys

```
Stream<String> characters = Stream.of("A", "B", "C");

Map<Integer, String> map = characters
    .collect(Collectors.toMap(element -> element.hashCode(), element -> element));
// map = {65=A, 66=B, 67=C}
```

To make things more declarative, we can use static method in Function interface - `Function.identity()`. We can replace this lambda element -> element with `Function.identity()`.

Case where there might be duplicate keys

The [javadoc](#) for Collectors.`toMap` states:

If the mapped keys contains duplicates (according to `Object.equals(Object)`), an `IllegalStateException` is thrown when the collection operation is performed. If the mapped keys may have duplicates, use `toMap(Function, Function, BinaryOperator)` instead.

```
Stream<String> 字符 = Stream.of("A", "B", "B", "C");
```

```
Map<Integer, String> map = characters
.collect(Collectors.toMap(
    element -> element.hashCode(),
    element -> element,
    (existingVal, newVal) -> (existingVal + newVal)));
// map = {65=A, 66=BB, 67=C}
```

传递给 `Collectors.toMap(...)` 的 `BinaryOperator` 用于在发生冲突时生成要存储的值。它可以：

- 返回旧值，使得流中的第一个值优先，
- 返回新值，使得流中的最后一个值优先，或者
- 合并旧值和新值

按值分组

当你需要执行类似数据库级联“group_by”操作时，可以使用 `Collectors.groupingBy`。举例来说，下面创建了一个映射，将人名映射到姓氏：

```
List<Person> people = Arrays.asList(
    new Person("Sam", "Rossi"),
    new Person("Sam", "Verdi"),
    new Person("John", "Bianchi"),
    new Person("John", "Rossi"),
    new Person("John", "Verdi")
);

Map<String, List<String>> map = people.stream()
    .collect(
        // 将输入元素映射到键的函数
        Collectors.groupingBy(Person::getName,
            // 将输入元素映射到值的函数,
            // 如何存储值
            Collectors.mapping(Person::getSurname, Collectors.toList())))
    );

// map = {John=[Bianchi, Rossi, Verdi], Sam=[Rossi, Verdi]}
```

[在 Ideone 上运行](#)

第57.22节：将流连接成单个字符串

一个常见的用例是从流中创建一个String，其中流中的项目由某个特定字符分隔。可以使用`Collectors.joining()`方法，如以下示例所示：

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

String result = fruitStream.filter(s -> s.contains("a"))
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.joining(", "));

System.out.println(result);
```

输出：

```
Stream<String> characters = Stream.of("A", "B", "B", "C");
```

```
Map<Integer, String> map = characters
.collect(Collectors.toMap(
    element -> element.hashCode(),
    element -> element,
    (existingVal, newVal) -> (existingVal + newVal)));
// map = {65=A, 66=BB, 67=C}
```

The `BinaryOperator` passed to `Collectors.toMap(...)` generates the value to be stored in the case of a collision. It can:

- return the old value, so that the first value in the stream takes precedence,
- return the new value, so that the last value in the stream takes precedence, or
- combine the old and new values

Grouping by value

You can use `Collectors.groupingBy` when you need to perform the equivalent of a database cascaded "group by" operation. To illustrate, the following creates a map in which people's names are mapped to surnames:

```
List<Person> people = Arrays.asList(
    new Person("Sam", "Rossi"),
    new Person("Sam", "Verdi"),
    new Person("John", "Bianchi"),
    new Person("John", "Rossi"),
    new Person("John", "Verdi")
);

Map<String, List<String>> map = people.stream()
    .collect(
        // function mapping input elements to keys
        Collectors.groupingBy(Person::getName,
            // function mapping input elements to values,
            // how to store values
            Collectors.mapping(Person::getSurname, Collectors.toList())))
    );

// map = {John=[Bianchi, Rossi, Verdi], Sam=[Rossi, Verdi]}
```

[Live on Ideone](#)

Section 57.22: Joining a stream to a single String

A use case that comes across frequently, is creating a `String` from a stream, where the stream-items are separated by a certain character. The `Collectors.joining()` method can be used for this, like in the following example:

```
Stream<String> fruitStream = Stream.of("apple", "banana", "pear", "kiwi", "orange");

String result = fruitStream.filter(s -> s.contains("a"))
    .map(String::toUpperCase)
    .sorted()
    .collect(Collectors.joining(", "));

System.out.println(result);
```

Output:

Collectors.joining() 方法也可以处理前缀和后缀：

```
String result = fruitStream.filter(s -> s.contains("e"))
    .map(String::toUpperCase)
.sorted()
.collect(Collectors.joining(", ", "Fruits: ", "."));
System.out.println(result);
```

输出：

Fruits: APPLE, ORANGE, PEAR.

[在 Ideone 上运行](#)

第57.23节：使用流排序

```
List<String> data = new ArrayList<>();
data.add("悉尼");
data.add("伦敦");
data.add("纽约");
data.add("阿姆斯特丹");
data.add("孟买");
data.add("California");
System.out.println(data);

List<String> sortedData = data.stream().sorted().collect(Collectors.toList());
System.out.println(sortedData);
```

输出：

[悉尼, 伦敦, 纽约, 阿姆斯特丹, 孟买, 加利福尼亚]
[阿姆斯特丹, 加利福尼亚, 伦敦, 孟买, 纽约, 悉尼]

也可以使用不同的比较机制，因为有一个重载的 sorted 版本，它接受一个比较器作为参数。

另外，你可以使用lambda表达式进行排序：

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

这将输出

[悉尼, 纽约, 孟买, 伦敦, 加利福尼亚, 阿姆斯特丹]

你可以使用Comparator.reverseOrder() 来获得一个施加自然顺序相反顺序的比较器。

```
List<String> reverseSortedData =
```

The Collectors.joining() method can also cater for pre- and postfixes:

```
String result = fruitStream.filter(s -> s.contains("e"))
    .map(String::toUpperCase)
.sorted()
.collect(Collectors.joining(", ", "Fruits: ", "."));
System.out.println(result);
```

Output:

Fruits: APPLE, ORANGE, PEAR.

[Live on Ideone](#)

Section 57.23: Sort Using Stream

```
List<String> data = new ArrayList<>();
data.add("Sydney");
data.add("London");
data.add("New York");
data.add("Amsterdam");
data.add("Mumbai");
data.add("California");
System.out.println(data);

List<String> sortedData = data.stream().sorted().collect(Collectors.toList());
System.out.println(sortedData);
```

Output:

[Sydney, London, New York, Amsterdam, Mumbai, California]
[Amsterdam, California, London, Mumbai, New York, Sydney]

It's also possible to use different comparison mechanism as there is a overloaded `sorted` version which takes a comparator as its argument.

Also, you can use a lambda expression for sorting:

```
List<String> sortedData2 = data.stream().sorted((s1,s2) ->
s2.compareTo(s1)).collect(Collectors.toList());
```

This would output

[Sydney, New York, Mumbai, London, California, Amsterdam]

You can use `Comparator.reverseOrder()` to have a comparator that imposes the reverse of the natural ordering.

```
List<String> reverseSortedData =
```

```
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

第57.24节：原始类型流

Java为三种原始类型提供了专门的Stream，分别是IntStream（针对int）、LongStream（针对long）和DoubleStream（针对double）。除了针对各自的原始类型进行了优化实现外，它们还提供了若干特定的终端方法，通常用于数学运算。例如：

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

第57.25节：流操作类别

流操作分为两大类，中间操作和终端操作，以及两个子类别，无状态和有状态。

中间操作：

中间操作总是惰性的，比如简单的Stream.map。它不会被调用，直到流被实际消费。可以很容易验证这一点：

```
Arrays.asList(1, 2, 3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

中间操作是流的常见构建块，连接在源之后，通常后面跟着触发流链的终端操作。

终端操作

终端操作是触发流消费的操作。一些较常见的有 Stream.forEach 或 Stream.collect。它们通常放在一系列中间操作之后，并且几乎总是急切执行的。

无状态操作

无状态意味着每个元素的处理不依赖于其他元素的上下文。无状态操作允许对流进行内存高效的处理。像 Stream.map 和 Stream.filter 这类不需要流中其他元素信息的操作被认为是无状态的。

有状态操作

有状态意味着对每个元素的操作依赖于流中的（某些）其他元素。这需要保存状态。有状态操作可能会在长流或无限流中出现问题。像 Stream.sorted 这样的操作需要在发出任何元素之前处理完整个流，这在元素足够多的流中会导致问题。可以通过一个长流来演示（自行承担风险）：

```
// 可行 - 无状态流
long BIG_ENOUGH_NUMBER = 999999999;
```

```
data.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
```

Section 57.24: Streams of Primitives

Java provides specialized Streams for three types of primitives IntStream (for ints), LongStream (for longs) and DoubleStream (for doubles). Besides being optimized implementations for their respective primitives, they also provide several specific terminal methods, typically for mathematical operations. E.g.:

```
IntStream is = IntStream.of(10, 20, 30);
double average = is.average().getAsDouble(); // average is 20.0
```

Section 57.25: Stream operations categories

Stream operations fall into two main categories, intermediate and terminal operations, and two sub-categories, stateless and stateful.

Intermediate Operations:

An intermediate operation is always *lazy*, such as a simple Stream.map. It is not invoked until the stream is actually consumed. This can be verified easily:

```
Arrays.asList(1, 2, 3).stream().map(i -> {
    throw new RuntimeException("not gonna happen");
    return i;
});
```

Intermediate operations are the common building blocks of a stream, chained after the source and are usually followed by a terminal operation triggering the stream chain.

Terminal Operations

Terminal operations are what triggers the consumption of a stream. Some of the more common are Stream.forEach or Stream.collect. They are usually placed after a chain of intermediate operations and are almost always *eager*.

Stateless Operations

Statelessness means that each item is processed without the context of other items. Stateless operations allow for memory-efficient processing of streams. Operations like Stream.map and Stream.filter that do not require information on other items of the stream are considered to be stateless.

Stateful operations

Statefulness means the operation on each item depends on (some) other items of the stream. This requires a state to be preserved. Statefulness operations may break with long, or infinite, streams. Operations like Stream.sorted require the entirety of the stream to be processed before any item is emitted which will break in a long enough stream of items. This can be demonstrated by a long stream (**run at your own risk**):

```
// works - stateless stream
long BIG_ENOUGH_NUMBER = 999999999;
```

```
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

这将由于Stream.sorted的有状态性导致内存溢出。

```
// 内存不足 - 有状态流  
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

第57.26节：将流的结果收集到数组中

类似于通过collect()获取Stream的集合，可以通过Stream.toArray()方法获得数组：

```
List<String> fruits = Arrays.asList("apple", "banana", "pear", "kiwi", "orange");  
  
String[] filteredFruits = fruits.stream()  
.filter(s -> s.contains("a"))  
.toArray(String[]::new);  
  
// 输出: [apple, banana, pear, orange]  
System.out.println(Arrays.toString(filteredFruits));
```

String[]::new 是一种特殊的方法引用：构造函数引用。

第57.27节：使用流生成随机字符串

有时创建随机Strings很有用，比如作为Web服务的会话ID或应用注册后的初始密码。这可以通过Stream轻松实现。

首先我们需要初始化一个随机数生成器。为了增强生成的String的安全性，使用SecureRandom是一个好主意。

注意：创建一个SecureRandom的开销相当大，因此最佳做法是只创建一次，并不时调用它的setSeed()方法来重新设定种子。

```
private static final SecureRandom rng = new SecureRandom(SecureRandom.generateSeed(20));  
//20字节作为种子是相当任意的，这是JavaDoc示例中使用的数字
```

在创建随机String时，我们通常希望它们只使用某些字符（例如仅字母和数字）。因此我们可以创建一个返回boolean的方法，稍后可以用来过滤Stream。

```
//对所有0-9、a-z和A-Z的字符返回true  
boolean useThisCharacter(char c){  
    //检查范围以避免使用所有Unicode字母（例如一些中文符号）  
    return c >= '0' && c <= 'z' && Character.isLetterOrDigit(c);  
}
```

接下来我们可以利用随机数生成器生成一个特定长度的随机字符串，包含通过我们useThisCharacter 检查的字符集。

```
public String generateRandomString(long length){  
    //由于没有原生的CharStream，我们改用IntStream  
    //并通过mapToObj将其转换为 Stream<Character>  
    //我们需要指定int值的边界，以确保它们可以安全地转换为char  
    Stream<Character> randomCharStream = rng.ints(Character.MIN_CODE_POINT,  
Character.MAX_CODE_POINT).mapToObj(i -> (char)i).filter(c -> this::useThisCharacter).limit(length);  
  
    //现在我们可以使用这个Stream通过collect方法构建一个字符串。  
}
```

```
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).forEach(System.out::println);
```

This will cause an out-of-memory due to statefulness of Stream.sorted:

```
// Out of memory - stateful stream  
IntStream.iterate(0, i -> i + 1).limit(BIG_ENOUGH_NUMBER).sorted().forEach(System.out::println);
```

Section 57.26: Collect Results of a Stream into an Array

Analog to get a collection for a Stream by collect() an array can be obtained by the Stream.toArray() method:

```
List<String> fruits = Arrays.asList("apple", "banana", "pear", "kiwi", "orange");  
  
String[] filteredFruits = fruits.stream()  
.filter(s -> s.contains("a"))  
.toArray(String[]::new);  
  
// prints: [apple, banana, pear, orange]  
System.out.println(Arrays.toString(filteredFruits));
```

String[]::new 是一种特殊的方法引用：构造函数引用。

Section 57.27: Generating random Strings using Streams

It is sometimes useful to create random Strings, maybe as Session-ID for a web-service or an initial password after registration for an application. This can be easily achieved using Streams.

First we need to initialize a random number generator. To enhance security for the generated Strings, it is a good idea to use SecureRandom.

Note: Creating a SecureRandom is quite expensive, so it is best practice to only do this once and call one of its setSeed() methods from time to time to reseed it.

```
private static final SecureRandom rng = new SecureRandom(SecureRandom.generateSeed(20));  
//20 Bytes as a seed is rather arbitrary, it is the number used in the JavaDoc example
```

When creating random Strings, we usually want them to use only certain characters (e.g. only letters and digits). Therefore we can create a method returning a boolean which can later be used to filter the Stream.

```
//returns true for all chars in 0-9, a-z and A-Z  
boolean useThisCharacter(char c){  
    //check for range to avoid using all unicode Letter (e.g. some chinese symbols)  
    return c >= '0' && c <= 'z' && Character.isLetterOrDigit(c);  
}
```

Next we can utilize the RNG to generate a random String of specific length containing the charset which pass our useThisCharacter check.

```
public String generateRandomString(long length){  
    //Since there is no native CharStream, we use an IntStream instead  
    //and convert it to a Stream<Character> using mapToObj.  
    //We need to specify the boundaries for the int values to ensure they can safely be cast to char  
    Stream<Character> randomCharStream = rng.ints(Character.MIN_CODE_POINT,  
Character.MAX_CODE_POINT).mapToObj(i -> (char)i).filter(c -> this::useThisCharacter).limit(length);  
  
    //now we can use this Stream to build a String utilizing the collect method.  
}
```

```
String randomString = randomCharStream.collect(StringBuilder::new, StringBuilder::append,
StringBuilder::append).toString();
return randomString;
}
```

```
String randomString = randomCharStream.collect(StringBuilder::new, StringBuilder::append,
StringBuilder::append).toString();
return randomString;
}
```

第58章：输入流和输出流

第58.1节：关闭流

大多数流在使用完毕后必须关闭，否则可能会导致内存泄漏或文件未关闭。即使抛出异常，也必须确保流被关闭。

```
版本 ≥ Java SE 7
try(FileWriter fw = new FileWriter("outfilename");
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter out = new PrintWriter(bw))
{
    out.println("the text");
    //more code
    out.println("more text");
    //more code
} catch (IOException e) {
    //handle this however you
}
```

请记住：try-with-resources 保证无论是通过正常控制流还是因为异常退出代码块，资源都会被关闭。

版本 ≤ Java SE 6

有时，try-with-resources 不是一个选项，或者你可能需要支持 Java 6 或更早版本。在这种情况下，正确的处理方式是使用 finally 块：

```
FileWriter fw = null;
BufferedWriter bw = null;
PrintWriter out = null;
try {
    fw = new FileWriter("myfile.txt");
    bw = new BufferedWriter(fw);
    out = new PrintWriter(bw);
    out.println("the text");
    out.close();
} 捕获 (IOException e) {
    //根据需要处理此异常
}
finally {
    try {
        if(out != null)
            out.close();
    } catch (IOException e) {
        //通常这里没什么可做的...
    }
}
```

注意关闭包装流也会关闭其底层流。这意味着你不能包装一个流，关闭包装流后再继续使用原始流。

第58.2节：将InputStream读取为字符串

有时你可能希望将字节输入读取为字符串。为此，你需要找到能够转换

Chapter 58: InputStreams and OutputStreams

Section 58.1: Closing Streams

Most streams must be closed when you are done with them, otherwise you could introduce a memory leak or leave a file open. It is important that streams are closed even if an exception is thrown.

```
Version ≥ Java SE 7
try(FileWriter fw = new FileWriter("outfilename");
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter out = new PrintWriter(bw))
{
    out.println("the text");
    //more code
    out.println("more text");
    //more code
} catch (IOException e) {
    //handle this however you
}
```

Remember: try-with-resources guarantees, that the resources have been closed when the block is exited, whether that happens with the usual control flow or because of an exception.

Version ≤ Java SE 6

Sometimes, try-with-resources is not an option, or maybe you're supporting older version of Java 6 or earlier. In this case, proper handling is to use a **finally** block:

```
FileWriter fw = null;
BufferedWriter bw = null;
PrintWriter out = null;
try {
    fw = new FileWriter("myfile.txt");
    bw = new BufferedWriter(fw);
    out = new PrintWriter(bw);
    out.println("the text");
    out.close();
} catch (IOException e) {
    //handle this however you want
}
finally {
    try {
        if(out != null)
            out.close();
    } catch (IOException e) {
        //typically not much you can do here...
    }
}
```

Note that closing a wrapper stream will also close its underlying stream. This means you cannot wrap a stream, close the wrapper and then continue using the original stream.

Section 58.2: Reading InputStream into a String

Sometimes you may wish to read byte-input into a String. To do this you will need to find something that converts

字节 (byte) 和作为字符使用的“原生Java”UTF-16代码点 (char) 之间的工具。这可以通过InputStreamReader来完成。

为了加快处理速度，通常会分配一个缓冲区，这样在从输入读取时不会有太多开销。

版本 ≥ Java SE 7

```
public String inputStreamToString(InputStream inputStream) throws Exception {
    StringWriter writer = new StringWriter();

    char[] buffer = new char[1024];
    try (Reader reader = new BufferedReader(new InputStreamReader(inputStream, "UTF-8"))) {
        int n;
        while ((n = reader.read(buffer)) != -1) {
            // 所有这些代码的作用是将 `reader` 的输出以
            // 1024 字节块重定向到 `writer`
            writer.write(buffer, 0, n);
        }
    }
    return writer.toString();
}
```

将此示例转换为 Java SE 6 (及更低版本) 兼容代码留作读者练习。

第58.3节：包装输入/输出流

OutputStream 和 InputStream 有许多不同的类，每个类都有独特的功能。通过将一个流包装在另一个流周围，您可以获得两个流的功能。

您可以多次包装一个流，但请注意顺序。

有用的组合

使用缓冲区向文件写入字符

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

在使用缓冲区时，压缩并加密数据后写入文件

```
Cipher cipher = ... // 初始化密码器
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

输入/输出流包装器列表

包装器	描述
BufferedOutputStream/ BufferedInputStream	虽然OutputStream一次写入一个字节， BufferedOutputStream以块的形式写入数据。这减少了 系统调用的次数，从而提高性能。
DeflaterOutputStream/ DeflaterInputStream	执行数据压缩。
InflaterOutputStream/ InflaterInputStream	执行数据解压缩。
CipherOutputStream/ CipherInputStream	加密/解密数据。
DigestOutputStream/ DigestInputStream	生成消息摘要以验证数据完整性。
CheckedOutputStream/ CheckedInputStream	生成校验和。校验和是消息摘要的一个更简单的版本。

between byte and the "native Java" UTF-16 Codepoints used as char. That is done with a InputStreamReader.

To speed the process up a bit, it's "usual" to allocate a buffer, so that we don't have too much overhead when reading from Input.

Version ≥ Java SE 7

```
public String inputStreamToString(InputStream inputStream) throws Exception {
    StringWriter writer = new StringWriter();

    char[] buffer = new char[1024];
    try (Reader reader = new BufferedReader(new InputStreamReader(inputStream, "UTF-8"))) {
        int n;
        while ((n = reader.read(buffer)) != -1) {
            // all this code does is redirect the output of `reader` to `writer` in
            // 1024 byte chunks
            writer.write(buffer, 0, n);
        }
    }
    return writer.toString();
}
```

Transforming this example to Java SE 6 (and lower)-compatible code is left out as an exercise for the reader.

Section 58.3: Wrapping Input/Output Streams

OutputStream 和 InputStream 有许多不同的类，每个类都有一个独特功能。通过将一个流包装在另一个流周围，您获得两个流的功能。

You can wrap a stream any number of times, just take note of the ordering.

Useful combinations

Writing characters to a file while using a buffer

```
File myFile = new File("targetFile.txt");
PrintWriter writer = new PrintWriter(new BufferedOutputStream(new FileOutputStream(myFile)));
```

Compressing and encrypting data before writing to a file while using a buffer

```
Cipher cipher = ... // Initialize cipher
File myFile = new File("targetFile.enc");
BufferedOutputStream outputStream = new BufferedOutputStream(new DeflaterOutputStream(new CipherOutputStream(new FileOutputStream(myFile), cipher)));
```

List of Input/Output Stream wrappers

Wrapper	Description
BufferedOutputStream/ BufferedInputStream	While OutputStream writes data one byte at a time, BufferedOutputStream writes data in chunks. This reduces the number of system calls, thus improving performance.
DeflaterOutputStream/ DeflaterInputStream	Performs data compression.
InflaterOutputStream/ InflaterInputStream	Performs data decompression.
CipherOutputStream/ CipherInputStream	Encrypts/Decrypts data.
DigestOutputStream/ DigestInputStream	Generates Message Digest to verify data integrity.
CheckedOutputStream/ CheckedInputStream	Generates a CheckSum. CheckSum is a more trivial version of Message Digest.

DataOutputStream/ DataInputStream	允许写入基本数据类型和字符串。用于写入字节。平台无关。
PrintStream	允许写入原始数据类型和字符串。用于写入字节。依赖平台。
OutputStreamWriter	将 OutputStream 转换为 Writer。OutputStream 处理字节，而 Writer 处理字符。
PrintWriter	自动调用 OutputStreamWriter。允许写入原始数据类型和字符串。专门用于写入字符，且最适合写入字符。

第58.4节：DataInputStream 示例

```
包 com.streams;
导入 java.io.*;
公共类 DataStreamDemo {
    公共静态 void main(String[] args) 抛出 IOException {
        输入流 input = 新建 FileInputStream("D:\\datastreamdemo.txt");
        DataInputStream inst = 新建 DataInputStream(input);
        int count = input.available();
        byte[] arr = new byte[count];
        inst.read(arr);
        for (byte byt : arr) {
            char ki = (char) byt;
            System.out.print(ki + "-");
        }
    }
}
```

第58.5节：向输出流写入字节

一次写入一个字节到OutputStream

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

写入字节数组

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

写入字节数组的一部分

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

第58.6节：将输入流复制到输出流

此函数用于在两个流之间复制数据 -

```
void copy(InputStream in, OutputStream out) throws IOException {
```

DataOutputStream/ DataInputStream	Allows writing of primitive data types and Strings. Meant for writing bytes. Platform independent.
PrintStream	Allows writing of primitive data types and Strings. Meant for writing bytes. Platform dependent.
OutputStreamWriter	Converts a OutputStream into a Writer. An OutputStream deals with bytes while Writers deals with characters
PrintWriter	Automatically calls OutputStreamWriter. Allows writing of primitive data types and Strings. Strictly for writing characters and best for writing characters

Section 58.4: DataInputStream Example

```
package com.streams;
import java.io.*;
public class DataStreamDemo {
    public static void main(String[] args) throws IOException {
        InputStream input = new FileInputStream("D:\\datastreamdemo.txt");
        DataInputStream inst = new DataInputStream(input);
        int count = input.available();
        byte[] arr = new byte[count];
        inst.read(arr);
        for (byte byt : arr) {
            char ki = (char) byt;
            System.out.print(ki + "-");
        }
    }
}
```

Section 58.5: Writing bytes to an OutputStream

Writing bytes to an OutputStream one byte at a time

```
OutputStream stream = object.getOutputStream();

byte b = 0x00;
stream.write( b );
```

Writing a byte array

```
byte[] bytes = new byte[] { 0x00, 0x00 };

stream.write( bytes );
```

Writing a section of a byte array

```
int offset = 1;
int length = 2;
byte[] bytes = new byte[] { 0xFF, 0x00, 0x00, 0xFF };

stream.write( bytes, offset, length );
```

Section 58.6: Copying Input Stream to Output Stream

This function copies data between two streams -

```
void copy(InputStream in, OutputStream out) throws IOException {
```

```
byte[] buffer = new byte[8192];
while ((bytesRead = in.read(buffer)) > 0) {
    out.write(buffer, 0, bytesRead);
}
```

示例 -

```
// 从 System.in 读取并写入到 System.out
copy(System.in, System.out);
```

```
byte[] buffer = new byte[8192];
while ((bytesRead = in.read(buffer)) > 0) {
    out.write(buffer, 0, bytesRead);
}
```

Example -

```
// reading from System.in and writing to System.out
copy(System.in, System.out);
```

第59章：读者与写者

读者和写者及其各自的子类为文本/基于字符的数据提供简单的输入/输出。

第59.1节：缓冲读取器

介绍

BufferedReader类是其他Reader类的包装器，主要有两个用途：

1. A BufferedReader为包装的Reader提供缓冲功能。这使得应用程序能够读取字符一次处理一个，且不会产生过多的输入/输出开销。
2. A BufferedReader 提供逐行读取文本的功能。

使用BufferedReader的基础知识

使用BufferedReader的正常模式如下。首先，获取你想要读取字符的Reader。接着实例化一个包装该Reader的BufferedReader。然后读取字符数据。

最后你关闭BufferedReader，这会关闭被包装的`Reader`。例如：

```
File someFile = new File(...);
int aCount = 0;
try (FileReader fr = new FileReader(someFile);
    BufferedReader br = new BufferedReader(fr)) {
    // 统计字符 'a' 的数量。
    int ch;
    while ((ch = br.read()) != -1) {
        if (ch == 'a') {
            aCount++;
        }
    }
    System.out.println("文件 " + someFile + " 中有 " + aCount + " 个 'a' 字符");
}
```

你可以将此模式应用于任何 Reader

注意：

1. 我们使用了 Java 7 (或更高版本) 的 try-with-resources 语法，以确保底层的读取器始终被关闭。这避免了潜在的资源泄漏。在早期版本的 Java 中，你需要在 finally 块中显式关闭 BufferedReader。
2. try 块中的代码几乎与直接从 FileReader。实际上，BufferedReader 的功能完全像它所包装的 Reader 一样。不同之处在于这个版本效率更高。

BufferedReader 缓冲区大小

BufferedReader.readLine() 方法

示例：将文件的所有行读取到一个 List 中

这是通过获取文件中的每一行，并将其添加到一个 List<String> 中完成的。然后返回该列表：

```
public List<String> getAllLines(String filename) throws IOException {
    List<String> lines = new ArrayList<String>();
```

Chapter 59: Readers and Writers

Readers and Writers and their respective subclasses provide simple I/O for text / character-based data.

Section 59.1: BufferedReader

Introduction

The `BufferedReader` class is a wrapper for other `Reader` classes that serves two main purposes:

1. A `BufferedReader` provides buffering for the wrapped `Reader`. This allows an application to read characters one at a time without undue I/O overheads.
2. A `BufferedReader` provides functionality for reading text a line at a time.

Basics of using a BufferedReader

The normal pattern for using a `BufferedReader` is as follows. First, you obtain the `Reader` that you want to read characters from. Next you instantiate a `BufferedReader` that wraps the `Reader`. Then you read character data. Finally you close the `BufferedReader` which closes the wrapped `Reader`. For example:

```
File someFile = new File(...);
int aCount = 0;
try (FileReader fr = new FileReader(someFile);
    BufferedReader br = new BufferedReader(fr)) {
    // Count the number of 'a' characters.
    int ch;
    while ((ch = br.read()) != -1) {
        if (ch == 'a') {
            aCount++;
        }
    }
    System.out.println("There are " + aCount + " 'a' characters in " + someFile);
}
```

You can apply this pattern to any Reader

Notes:

1. We have used Java 7 (or later) `try-with-resources` to ensure that the underlying reader is always closed. This avoids a potential resource leak. In earlier versions of Java, you would explicitly close the `BufferedReader` in a `finally` block.
2. The code inside the `try` block is virtually identical to what we would use if we read directly from the `FileReader`. In fact, a `BufferedReader` functions exactly like the `Reader` that it wraps would behave. The difference is that this version is a lot more efficient.

The BufferedReader buffer size

The BufferedReader.readLine() method

Example: reading all lines of a File into a List

This is done by getting each line in a file, and adding it into a `List<String>`. The list is then returned:

```
public List<String> getAllLines(String filename) throws IOException {
    List<String> lines = new ArrayList<String>();
```

```

try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        lines.add(line);
    }
}
return lines;
}

```

Java 8 提供了一种更简洁的方式来实现此功能，使用 lines() 方法：

```

public List<String> getAllLines(String filename) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        return br.lines().collect(Collectors.toList());
    }
    return Collections.emptyList();
}

```

第59.2节：StringWriter示例

Java StringWriter 类是一个字符流，用于收集来自字符串缓冲区的输出，可用于构造字符串。

StringWriter 类继承自 Writer 类。

在 StringWriter 类中，不使用网络套接字和文件等系统资源，因此不需要关闭 StringWriter。

```

import java.io.*;
public class StringWriterDemo {
    public static void main(String[] args) throws IOException {
        char[] ary = new char[1024];
        StringWriter writer = new StringWriter();
        FileInputStream input = null;
        BufferedReader buffer = null;
        input = new FileInputStream("c://stringwriter.txt");
        buffer = new BufferedReader(new InputStreamReader(input, "UTF-8"));
        int x;
        while ((x = buffer.read(ary)) != -1) {
            writer.write(ary, 0, x);
        }
        System.out.println(writer.toString());
        writer.close();
        buffer.close();
    }
}

```

上面的例子帮助我们了解了使用BufferedReader从流中读取文件数据的StringWriter简单示例。

```

try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        lines.add(line);
    }
}
return lines;
}

```

Java 8 provides a more concise way to do this using the lines() method:

```

public List<String> getAllLines(String filename) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
        return br.lines().collect(Collectors.toList());
    }
    return Collections.emptyList();
}

```

Section 59.2: StringWriter Example

Java StringWriter class is a character stream that collects output from string buffer, which can be used to construct a string.

The StringWriter class extends the Writer class.

In StringWriter class, system resources like network sockets and files are not used, therefore closing the StringWriter is not necessary.

```

import java.io.*;
public class StringWriterDemo {
    public static void main(String[] args) throws IOException {
        char[] ary = new char[1024];
        StringWriter writer = new StringWriter();
        FileInputStream input = null;
        BufferedReader buffer = null;
        input = new FileInputStream("c://stringwriter.txt");
        buffer = new BufferedReader(new InputStreamReader(input, "UTF-8"));
        int x;
        while ((x = buffer.read(ary)) != -1) {
            writer.write(ary, 0, x);
        }
        System.out.println(writer.toString());
        writer.close();
        buffer.close();
    }
}

```

The above example helps us to know simple example of StringWriter using BufferedReader to read file data from the stream.

第60章：首选项

第60.1节：使用首选项

首选项可以用来存储反映用户个人应用设置的用户设置，例如他们的编辑器字体、是否希望应用程序以全屏模式启动、是否勾选了“不再显示”复选框等。

```
public class ExitConfirm {
    private static boolean confirmExit() {
        Preferences preferences = Preferences.userNodeForPackage(ExitConfirm.class);
        boolean doShowDialog =
            preferences.getBoolean("showExitConfirmation", true); // true 是默认值

        if (!doShowDialog) {
            return true;
        }

        // 在这里显示一个对话框...
        //

        boolean exitWasConfirmed = ...; // 用户是否点击了确定或取消
        boolean doNotShowAgain = ...; // 从“不再显示”复选框获取值

        if (exitWasConfirmed && doNotShowAgain) {
            // 用户确认退出且选择不再显示对话框// 将这些设置保存到 Preferences 对象，以便下次不再显示对话框
            preferences.putBoolean("showExitConfirmation", false);
        }

        return exitWasConfirmed;
    }

    public static void exit() {
        if (confirmExit()) {
            System.exit(0);
        }
    }
}
```

第60.2节：添加事件监听器

Preferences对象会发出两种类型的事件：PreferenceChangeEvent和NodeChangeEvent。

PreferenceChangeEvent

每当节点的键值对发生变化时，Properties对象会发出一个PreferenceChangeEvent。可以通过PreferenceChangeListener监听PreferenceChangeEvent：

```
版本 ≥ Java SE 8
preferences.addPreferenceChangeListener(evt -> {
    String newValue = evt.getNewValue();
    String changedPreferenceKey = evt.getKey();
    Preferences changedNode = evt.getNode();
});
版本 < Java SE 8
```

Chapter 60: Preferences

Section 60.1: Using preferences

Preferences can be used to store user settings that reflect a user's personal application settings, e.g. their editor font, whether they prefer the application to be started in full-screen mode, whether they checked a "don't show this again" checkbox and things like that.

```
public class ExitConfirm {
    private static boolean confirmExit() {
        Preferences preferences = Preferences.userNodeForPackage(ExitConfirm.class);
        boolean doShowDialog = preferences.getBoolean("showExitConfirmation", true); // true is default value

        if (!doShowDialog) {
            return true;
        }

        //
        // Show a dialog here...
        //

        boolean exitWasConfirmed = ...; // whether the user clicked OK or Cancel
        boolean doNotShowAgain = ...; // get value from "Do not show again" checkbox

        if (exitWasConfirmed && doNotShowAgain) {
            // Exit was confirmed and the user chose that the dialog should not be shown again
            // Save these settings to the Preferences object so the dialog will not show again next time
            preferences.putBoolean("showExitConfirmation", false);
        }

        return exitWasConfirmed;
    }

    public static void exit() {
        if (confirmExit()) {
            System.exit(0);
        }
    }
}
```

Section 60.2: Adding event listeners

There are two types of events emitted by a [Preferences](#) object: [PreferenceChangeEvent](#) and [NodeChangeEvent](#).

PreferenceChangeEvent

A PreferenceChangeEvent gets emitted by a [Properties](#) object every time one of the node's key-value-pairs changes. PreferenceChangeEvents can be listened for with a [PreferenceChangeListener](#):

```
Version ≥ Java SE 8
preferences.addPreferenceChangeListener(evt -> {
    String newValue = evt.getNewValue();
    String changedPreferenceKey = evt.getKey();
    Preferences changedNode = evt.getNode();
});
Version < Java SE 8
```

```

preferences.addPreferenceChangeListener(new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        String newValue = evt.getnewValue();
        String changedPreferenceKey = evt.getKey();
        Preferences changedNode = evt.getNode();
    }
});

```

此监听器不会监听子节点的键值对变化。

NodeChangeEvent

每当向Properties节点添加或移除子节点时，将触发此事件。

```

preferences.addNodeChangeListener(new NodeChangeListener() {
    @Override
    public void childAdded(NodeChangeEvent evt) {
        Preferences addedChild = evt.getChild();
        Preferences parentOfAddedChild = evt.getParent();
    }

    @Override
    public void childRemoved(NodeChangeEvent evt) {
        Preferences removedChild = evt.getChild();
        Preferences parentOfRemovedChild = evt.getParent();
    }
});

```

第60.3节：获取偏好的子节点

Preferences对象总是表示整个Preferences树中的一个特定节点，类似于这样：

```

rRoot |-- com |   `-- mycompany |       `-- myapp |           `-- darkApplicationMode=true |   |-- 
showExitConfirmation=false |   `-- windowMaximized=true `-- org `-- myorganization `-- anotherapp `-- 
defaultFont=Helvetica `-- defaultSavePath=/home/matt/Documents `-- exporting `-- defaultFormat=pdf `-- 
openInBrowserAfterExport=false

```

要选择/com/mycompany/myapp节点：

- 按惯例，基于类的包名：

```

package com.mycompany.myapp;

// ...

// 因为该类位于 com.mycompany.myapp 包中,
// 所以将返回节点 /com/mycompany/myapp。
Preferences myApp = Preferences.userNodeForPackage(getClass());

```

- 通过相对路径：

```
Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");
```

使用相对路径（不以/开头的路径）会导致路径相对于解析它的父节点进行解析。例如，以下示例将返回路径的节点

```

preferences.addPreferenceChangeListener(new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        String newValue = evt.getnewValue();
        String changedPreferenceKey = evt.getKey();
        Preferences changedNode = evt.getNode();
    }
});

```

This listener will not listen to changed key-value pairs of child nodes.

NodeChangeEvent

This event will be fired whenever a child node of a `Properties` node is added or removed.

```

preferences.addNodeChangeListener(new NodeChangeListener() {
    @Override
    public void childAdded(NodeChangeEvent evt) {
        Preferences addedChild = evt.getChild();
        Preferences parentOfAddedChild = evt.getParent();
    }

    @Override
    public void childRemoved(NodeChangeEvent evt) {
        Preferences removedChild = evt.getChild();
        Preferences parentOfRemovedChild = evt.getParent();
    }
});

```

Section 60.3: Getting sub-nodes of Preferences

Preferences objects always represent a specific node in a whole Preferences tree, kind of like this:

```

rRoot |--- com |   `-- mycompany |       `-- myapp |           `-- darkApplicationMode=true |   |-- 
showExitConfirmation=false |   `-- windowMaximized=true `-- org `-- myorganization `-- anotherapp `-- 
defaultFont=Helvetica `-- defaultSavePath=/home/matt/Documents `-- exporting `-- defaultFormat=pdf `-- 
openInBrowserAfterExport=false

```

To select the /com/mycompany/myapp node:

- By convention, based on the package of a class:

```

package com.mycompany.myapp;

// ...

// Because this class is in the com.mycompany.myapp package, the node
// /com/mycompany/myapp will be returned.
Preferences myApp = Preferences.userNodeForPackage(getClass());

```

- By relative path:

```
Preferences myApp = Preferences.userRoot().node("com/mycompany/myapp");
```

Using a relative path (a path not starting with a /) will cause the path to be resolved relative to the parent node it is resolved on. For example, the following example will return the node of the path

/one/two/three/com/mycompany/myapp:

```
首选项前缀 = Preferences.userRoot().node("one/two/three");
首选项 myAppWithPrefix = prefix.node("com/mycompany/myapp");
// prefix 是 /one/two/three
// myAppWithPrefix 是 /one/two/three/com/mycompany/myapp
```

3.通过绝对路径：

```
首选项 myApp = Preferences.userRoot().node("/com/mycompany/myapp");
```

在根节点上使用绝对路径与使用相对路径没有区别。区别在于如果在子节点上调用，路径将相对于根节点进行解析。

```
首选项前缀 = Preferences.userRoot().node("one/two/three");
首选项 myAppWitoutPrefix = prefix.node("/com/mycompany/myapp");
// prefix 是 /one/two/three
// myAppWitoutPrefix 是 /com/mycompany/myapp
```

/one/two/three/com/mycompany/myapp:

```
Preferences prefix = Preferences.userRoot().node("one/two/three");
Preferences myAppWithPrefix = prefix.node("com/mycompany/myapp");
// prefix 是 /one/two/three
// myAppWithPrefix 是 /one/two/three/com/mycompany/myapp
```

3. By absolute path:

```
Preferences myApp = Preferences.userRoot().node("/com/mycompany/myapp");
```

Using an absolute path on the root node will not be different from using a relative path. The difference is that, if called on a sub-node, the path will be resolved relative to the root node.

```
Preferences prefix = Preferences.userRoot().node("one/two/three");
Preferences myAppWitoutPrefix = prefix.node("/com/mycompany/myapp");
// prefix 是 /one/two/three
// myAppWitoutPrefix 是 /com/mycompany/myapp
```

第60.4节：协调多个应用实例之间的首选项访问

所有 Preferences 实例在单个Java虚拟机 (JVM) 的线程之间始终是线程安全的。因为偏好设置可以在多个JVM之间共享，所以有一些特殊的方法用于处理跨虚拟机的同步更改。

如果您的应用程序只需运行单实例，则不需要外部同步。

如果您的应用程序在单个系统上运行多个实例，因此需要协调系统上JVM之间对Preferences的访问，则可以使用任何Preferences节点的sync()方法，确保对Preferences节点的更改对系统上的其他VM可见：

```
// 警告：如果您的应用程序只打算
// 在一台机器上运行单个实例，则不要使用此方法
// (这可能是大多数桌面应用程序的情况)
try {
    preferences.sync();
} catch (BackingStoreException e) {
    // 处理将首选项保存到后端存储时的任何错误
    e.printStackTrace();
}
```

Section 60.4: Coordinating preferences access across multiple application instances

All instances of Preferences are always thread-safe across the threads of a single Java Virtual Machine (JVM). Because Preferences can be shared across multiple JVMs, there are special methods that deal with synchronizing changes across virtual machines.

If you have an application which is supposed to run in a **single instance** only, then **no external synchronization** is required.

If you have an application which runs in **multiple instances** on a single system and therefore Preferences access needs to be coordinated between the JVMs on the system, then the **sync() method** of any Preferences node may be used to ensure changes to the Preferences node are visible to other JVMs on the system:

```
// Warning: don't use this if your application is intended
// to only run a single instance on a machine once
// (this is probably the case for most desktop applications)
try {
    preferences.sync();
} catch (BackingStoreException e) {
    // Deal with any errors while saving the preferences to the backing storage
    e.printStackTrace();
}
```

第60.5节：导出首选项

Preferences节点可以导出为表示该节点的XML文档。生成的XML树可以再次导入。生成的XML文档将记住它是从用户还是系统导出的首选项。

导出单个节点，但不包括其子节点：

```
版本 ≥ Java SE 7
尝试 (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
```

Section 60.5: Exporting preferences

Preferences nodes can be exported into a XML document representing that node. The resulting XML tree can be imported again. The resulting XML document will remember whether it was exported from the user or system Preferences.

To export a single node, but **not its child nodes**:

```
Version ≥ Java SE 7
try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
```

```

    // 写入数据到OutputStream时的异常
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // 从备份首选项存储读取时的异常
    bse.printStackTrace();
}
版本 < Java SE 7

OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // 写入数据到OutputStream时的异常
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // 从备份首选项存储读取时的异常
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}

```

导出单个节点及其子节点：

```

版本 ≥ Java SE 7

尝试 (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // 写入数据到OutputStream时的异常
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // 从备份首选项存储读取时的异常
    bse.printStackTrace();
}

版本 < Java SE 7

OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // 写入数据到OutputStream时的异常
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // 从备份首选项存储读取时的异常
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}

```

第60.6节：导入首选项

首选项节点可以从XML文档中导入。导入功能旨在与

```

    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}
Version < Java SE 7

OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}

```

To export a single node **with its child nodes**:

```

Version ≥ Java SE 7

try (OutputStream os = ...) {
    preferences.exportNode(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
}

Version < Java SE 7

OutputStream os = null;
try {
    os = ...;
    preferences.exportSubtree(os);
} catch (IOException ioe) {
    // Exception whilst writing data to the OutputStream
    ioe.printStackTrace();
} catch (BackingStoreException bse) {
    // Exception whilst reading from the backing preferences store
    bse.printStackTrace();
} finally {
    if (os != null) {
        try {
            os.close();
        } catch (IOException ignored) {}
    }
}

```

Section 60.6: Importing preferences

Preferences nodes can be imported from a XML document. Importing is meant to be used in conjunction with the

首选项的导出功能配合使用，因为它会创建正确对应的XML文档。

XML文档会记住它们是从用户还是系统首选项导出的。因此，它们可以再次导入到各自的首选项树中，无需您去判断或了解它们的来源。静态函数会自动判断XML文档是从用户还是系统首选项导出的，并会自动导入到它们导出时所在的树中。

```
版本 ≥ Java SE 7
try (InputStream is = ...) {
    // 这是对Preferences类的静态调用
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // 从InputStream读取数据时发生异常
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // 解析 XML 文档树时的异常
    ipfe.printStackTrace();
}

版本 < Java SE 7
InputStream is = null;
try {
    is = ...;
    // 这是对 Preferences 类的静态调用
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // 从InputStream读取数据时发生异常
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // 解析 XML 文档树时的异常
    ipfe.printStackTrace();
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException ignored) {}
    }
}
```

第60.7节：移除事件监听器

事件监听器可以再次从任何属性节点中移除，但必须保留监听器的实例。

```
版本 ≥ Java SE 8
Preferences preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = evt -> {
    System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
};
preferences.addPreferenceChangeListener(listener);

// later...
// preferences.removePreferenceChangeListener(listener);

版本 < Java SE 8
Preferences preferences = Preferences.userNodeForPackage(getClass());
```

exporting functionality of Preferences, since it creates the correct corresponding XML documents.

The XML documents will remember whether they were exported from the user or system Preferences. Therefore, they can be imported into their respective Preferences trees again, without you having to figure out or know where they came from. The static function will automatically find out whether the XML document was exported from the user or system Preferences and will automatically import them into the tree they were exported from.

```
Version ≥ Java SE 7
try (InputStream is = ...) {
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
}

Version < Java SE 7
InputStream is = null;
try {
    is = ...;
    // This is a static call on the Preferences class
    Preferences.importPreferences(is);
} catch (IOException ioe) {
    // Exception whilst reading data from the InputStream
    ioe.printStackTrace();
} catch (InvalidPreferencesFormatException ipfe) {
    // Exception whilst parsing the XML document tree
    ipfe.printStackTrace();
} finally {
    if (is != null) {
        try {
            is.close();
        } catch (IOException ignored) {}
    }
}
```

Section 60.7: Removing event listeners

Event listeners can be removed again from any `Properties` node, but the instance of the listener has to be kept around for that.

```
Version ≥ Java SE 8
Properties preferences = Preferences.userNodeForPackage(getClass());

PreferenceChangeListener listener = evt -> {
    System.out.println(evt.getKey() + " got new value " + evt.getNewValue());
};
preferences.addPreferenceChangeListener(listener);

// later...
// preferences.removePreferenceChangeListener(listener);
```

```
preferences.removePreferenceChangeListener(listener);
```

```
Version < Java SE 8
Properties preferences = Preferences.userNodeForPackage(getClass());
```

```

PreferenceChangeListener listener = new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        System.out.println(evt.getKey() + " got new value " + evt.getnewValue());
    }
};

preferences.addPreferenceChangeListener(listener);

// later...
//

preferences.removePreferenceChangeListener(listener);

```

同样适用于NodeChangeListener。

第60.8节：获取偏好设置值

一个Preferences节点的值可以是String、boolean、byte[]、double、float、int或long类型。所有调用必须提供一个默认值，以防指定的值在Preferences节点中不存在。

```

Preferences preferences = Preferences.userNodeForPackage(getClass());

String someString = preferences.get("someKey", "this is the default value");
boolean someBoolean = preferences.getBoolean("someKey", true);
byte[] someByteArray = preferences.getByteArray("someKey", new byte[0]);
double someDouble = preferences.getDouble("someKey", 887284.4d);
float someFloat = preferences.getFloat("someKey", 38723.3f);
int someInt = preferences.getInt("someKey", 13232);
long someLong = preferences.getLong("someKey", 2827637868234L);

```

第60.9节：设置偏好值

要将值存储到Preferences节点中，使用其中一个putXXX()方法。一个Preferences节点的值可以是String、boolean、byte[]、double、float、int或long类型。

```

Preferences preferences = Preferences.userNodeForPackage(getClass());

preferences.put("someKey", "some String value");
preferences.putBoolean("someKey", false);
preferences.putByteArray("someKey", new byte[0]);
preferences.putDouble("someKey", 187398123.4454d);
preferences.putFloat("someKey", 298321.445f);
preferences.putInt("someKey", 77637);
preferences.putLong("someKey", 2873984729834L);

```

```

PreferenceChangeListener listener = new PreferenceChangeListener() {
    @Override
    public void preferenceChange(PreferenceChangeEvent evt) {
        System.out.println(evt.getKey() + " got new value " + evt.getnewValue());
    }
};

preferences.addPreferenceChangeListener(listener);

// later...
//

preferences.removePreferenceChangeListener(listener);

```

The same applies for NodeChangeListener.

Section 60.8: Getting preferences values

A value of a Preferences node can be of the type **String**, **boolean**, **byte[]**, **double**, **float**, **int** or **long**. All invocations must provide a default value, in case the specified value is not present in the Preferences node.

```

Preferences preferences = Preferences.userNodeForPackage(getClass());

String someString = preferences.get("someKey", "this is the default value");
boolean someBoolean = preferences.getBoolean("someKey", true);
byte[] someByteArray = preferences.getByteArray("someKey", new byte[0]);
double someDouble = preferences.getDouble("someKey", 887284.4d);
float someFloat = preferences.getFloat("someKey", 38723.3f);
int someInt = preferences.getInt("someKey", 13232);
long someLong = preferences.getLong("someKey", 2827637868234L);

```

Section 60.9: Setting preferences values

To store a value into the Preferences node, one of the putXXX() methods is used. A value of a Preferences node can be of the type **String**, **boolean**, **byte[]**, **double**, **float**, **int** or **long**.

```

Preferences preferences = Preferences.userNodeForPackage(getClass());

preferences.put("someKey", "some String value");
preferences.putBoolean("someKey", false);
preferences.putByteArray("someKey", new byte[0]);
preferences.putDouble("someKey", 187398123.4454d);
preferences.putFloat("someKey", 298321.445f);
preferences.putInt("someKey", 77637);
preferences.putLong("someKey", 2873984729834L);

```

第61章：集合工厂方法

带参数的方法	描述
<code>List.of(E e)</code>	一个可以是类或接口的泛型类型。
<code>Set.of(E e)</code>	一个可以是类或接口的泛型类型。
<code>Map.of(K k, V v)</code>	一个键值对的泛型类型，键和值都可以是类或接口。
<code>Map.of(Map.Entry<? extends K, ? extends V> entry)</code>	一个 <code>Map.Entry</code> 实例，其键可以是 <code>K</code> 或其子类之一，其值可以是 <code>V</code> 或其任何子类。

Java 9 的到来为 Java 的集合 API 带来了许多新特性，其中之一就是集合工厂方法。这些方法允许轻松初始化**不可变**集合，无论是空集合还是非空集合。

请注意，这些工厂方法仅适用于以下接口：`List<E>`、`Set<E>` 和 `Map<K, V>`

第61.1节：List<E> 工厂方法示例

- `List<Integer> immutableEmptyList = List.of();`
 - 初始化一个空的、不可变的 `List<Integer>`。
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
 - 初始化一个包含五个初始元素的不可变 `List<Integer>`。
- `List<Integer> mutableList = new ArrayList<>(immutableList);`
 - 从不可变的 `List<Integer>` 初始化一个可变的 `List<Integer>`。

第61.2节：Set<E> 工厂方法示例

- `Set<Integer> immutableEmptySet = Set.of();`
 - 初始化一个空的、不可变的集合<整数>。
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
 - 使用五个初始元素初始化一个不可变的 `Set<Integer>`。
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
 - 从一个不可变的 `Set<Integer>` 初始化一个可变的 `Set<Integer>`。

第61.3节：Map<K, V>工厂方法示例

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
 - 初始化一个空的不可变 `Map<Integer, Integer>`。
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
 - 使用两个初始键值对初始化一个不可变的 `Map<Integer, Integer>`。
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
 - 使用两个初始键值对初始化一个不可变的 `Map<Integer, Integer>`。
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
 - 从不可变的 `Map<Integer, Integer>` 初始化一个可变的 `Map<Integer, Integer>`。

Chapter 61: Collection Factory Methods

Method w/ Parameter	Description
<code>List.of(E e)</code>	A generic type that can be a class or interface.
<code>Set.of(E e)</code>	A generic type that can be a class or interface.
<code>Map.of(K k, V v)</code>	A key-value pair of generic types that can each be a class or interface.
<code>Map.of(Map.Entry<? extends K, ? extends V> entry)</code>	A <code>Map.Entry</code> instance where its key can be <code>K</code> or one of its children, and its value can be <code>V</code> or any of its children.

The arrival of Java 9 brings many new features to Java's Collections API, one of which being collection factory methods. These methods allow for easy initialization of **immutable** collections, whether they be empty or nonempty.

Note that these factory methods are only available for the following interfaces: `List<E>`, `Set<E>`, and `Map<K, V>`

Section 61.1: List<E> Factory Method Examples

- `List<Integer> immutableEmptyList = List.of();`
 - Initializes an empty, immutable `List<Integer>`.
- `List<Integer> immutableList = List.of(1, 2, 3, 4, 5);`
 - Initializes an immutable `List<Integer>` with five initial elements.
- `List<Integer> mutableList = new ArrayList<>(immutableList);`
 - Initializes a mutable `List<Integer>` from an immutable `List<Integer>`.

Section 61.2: Set<E> Factory Method Examples

- `Set<Integer> immutableEmptySet = Set.of();`
 - Initializes an empty, immutable `Set<Integer>`.
- `Set<Integer> immutableSet = Set.of(1, 2, 3, 4, 5);`
 - Initializes an immutable `Set<Integer>` with five initial elements.
- `Set<Integer> mutableSet = new HashSet<>(immutableSet);`
 - Initializes a mutable `Set<Integer>` from an immutable `Set<Integer>`.

Section 61.3: Map<K, V> Factory Method Examples

- `Map<Integer, Integer> immutableEmptyMap = Map.of();`
 - Initializes an empty, immutable `Map<Integer, Integer>`.
- `Map<Integer, Integer> immutableMap = Map.of(1, 2, 3, 4);`
 - Initializes an immutable `Map<Integer, Integer>` with two initial key-value entries.
- `Map<Integer, Integer> immutableMap = Map.ofEntries(Map.entry(1, 2), Map.entry(3, 4));`
 - Initializes an immutable `Map<Integer, Integer>` with two initial key-value entries.
- `Map<Integer, Integer> mutableMap = new HashMap<>(immutableMap);`
 - Initializes a mutable `Map<Integer, Integer>` from an immutable `Map<Integer, Integer>`.

第62章：替代集合

第62.1节：Guava、Apache和Eclipse集合中的Multimap

该multimap允许重复的键值对。JDK中的类似实现有HashMap<K, List>、HashMap<K, Set>等。

键的顺序	值的顺序	重复	类似键	类似值	Guava	Apache	Eclipse (GS) 集合	JDK
未定义	插入顺序	是	哈希映射	ArrayList	数组列表多重映射	多值映射	快速列表多重映射	哈希映射<K, 数组列表<V>>
未定义	未定义	否	哈希映射	HashSet	HashMultimap	MultiValueMap, multiValueMap(new HashMap<K, Set<V>(), HashSet.class);	UnifiedSetMultimap	HashMap<K, HashSet<V>>
未定义	sorted	否	哈希映射	TreeSet	Multimaps.newMultimap(HashMap, Supplier<TreeSet>)	MultiValueMap, multiValueMap(new HashMap<K, Set<V>(), TreeSet.class)	TreeSortedSet-Multimap	HashMap<K, TreeSet<V>>
插入顺序	插入顺序	是		LinkedHashMap ArrayList	LinkedListMultimap	MultiValueMap, multiValueMap(new LinkedHashMap<K, List<V>(), ArrayList.class);	LinkedHashMap<K, ArrayList>	Insertion-order Insertion-order yes
插入顺序	插入顺序	否		LinkedHashMap LinkedHashSet	LinkedHashMultimap	MultiValueMap, multiValueMap(new LinkedHashMap<K, Set<V>(), LinkedHashSet.class)	LinkedHashMap<K, LinkedHashSet>	Insertion-order Insertion-order no
sorted	sorted	否	TreeMap	TreeSet	TreeMultimap	MultiValueMap, multiValueMap(new TreeMap<K, TreeSet<V>(), TreeSet.class)	TreeMap<K, TreeSet<V>>	sorted sorted no

使用Multimap的示例

任务：使用MultiMap解析字符串 "Hello World! Hello All! Hi World!"，将单词分开并打印每个单词的所有索引（例如，Hello=[0, 2]，World!=[1, 5] 等）

1. 来自Apache的MultiValueMap

```

String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// 解析文本为单词并建立索引
List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
// 创建多值映射
MultiMap<String, Integer> multiMap = new MultiValueMap<String, Integer>();

// 填充多值映射
int i = 0;
for(String word: words) {
    multiMap.put(word, i);
    i++;
}

// 打印所有单词
System.out.println(multiMap); // 打印 {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} - 顺序随机

// 打印所有唯一单词
System.out.println(multiMap.keySet()); // 打印 [Hi, Hello, World!, All!] - 顺序随机

// 打印所有索引
System.out.println("Hello = " + multiMap.get("Hello")); // 打印 [0, 2]
System.out.println("World = " + multiMap.get("World!")); // 打印 [1, 5]
System.out.println("All = " + multiMap.get("All!")); // 打印 [3]
System.out.println("Hi = " + multiMap.get("Hi")); // 打印 [4]
System.out.println("Empty = " + multiMap.get("Empty")); // 打印 null

// 打印唯一单词数量

```

Chapter 62: Alternative Collections

Section 62.1: Multimap in Guava, Apache and Eclipse Collections

This multimap allows duplicate key-value pairs. JDK analogs are HashMap<K, List>, HashMap<K, Set> and so on.

Key's order	Value's order	Duplicate	Analog key	Analog value	Guava	Apache	Eclipse (GS) Collections	JDK
not defined	Insertion-order	yes	HashMap	ArrayList	ArrayListMultimap	MultiValueMap	FastListMultimap	HashMap<K, ArrayList<V>>
not defined	not defined	no	HashMap	HashSet	HashMultimap	MultiValueMap, multiValueMap(new HashMap<K, Set<V>(), HashSet.class);	UnifiedSetMultimap	HashMap<K, HashSet<V>>
not defined	sorted	no	HashMap	TreeSet	Multimaps.newMultimap(HashMap, Supplier<TreeSet>)	MultiValueMap.multiValueMap(new HashMap<K, Set<V>(), TreeSet.class)	TreeSortedSet-Multimap	HashMap<K, TreeSet<V>>
Insertion-order	Insertion-order	yes	LinkedHashMap ArrayList	LinkedList	LinkedListMultimap	MultiValueMap, multiValueMap(new LinkedHashMap<K, List<V>(), ArrayList.class);		LinkedHashMap<K, ArrayList>
Insertion-order	Insertion-order	no	LinkedHashMap LinkedHashSet	LinkedHashMultimap		MultiValueMap, multiValueMap(new LinkedHashMap<K, Set<V>(), LinkedHashSet.class)		LinkedHashMap<K, LinkedHashSet>
sorted	sorted	no	TreeMap	TreeSet	TreeMultimap	MultiValueMap, multiValueMap(new TreeMap<K, TreeSet<V>(), TreeSet.class)		TreeMap<K, TreeSet<V>>

Examples using Multimap

Task: Parse "Hello World! Hello All! Hi World!" string to separate words and print all indexes of every word using MultiMap (for example, Hello=[0, 2], World!=[1, 5] and so on)

1. MultiValueMap from Apache

```

String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Parse text to words and index
List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
// Create Multimap
MultiMap<String, Integer> multiMap = new MultiValueMap<String, Integer>();

// Fill Multimap
int i = 0;
for(String word: words) {
    multiMap.put(word, i);
    i++;
}

// Print all words
System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} - in random orders
// Print all unique words
System.out.println(multiMap.keySet()); // print [Hi, Hello, World!, All!] - in random orders

// Print all indexes
System.out.println("Hello = " + multiMap.get("Hello")); // print [0, 2]
System.out.println("World = " + multiMap.get("World!")); // print [1, 5]
System.out.println("All = " + multiMap.get("All!")); // print [3]
System.out.println("Hi = " + multiMap.get("Hi")); // print [4]
System.out.println("Empty = " + multiMap.get("Empty")); // print null

// Print count unique words

```

```
System.out.println(multiMap.keySet().size()); // 打印 4
```

2. 来自 GS / Eclipse Collection 的 HashBiMap

```

String[] englishWords = {"one", "two", "three", "ball", "snow"};
String[] russianWords = {"jeden", "dwa", "trzy", "kula", "snieg"};

// 创建多重集合
MutableBiMap<String, String> biMap = new HashBiMap(englishWords.length);
// 创建英波词典
int i = 0;
for(String englishWord: englishWords) {
    biMap.put(englishWord, russianWords[i]);
    i++;
}

// 打印计数字词
System.out.println(biMap); // 打印 {two=dwa, ball=kula, one=jeden, snow=snieg, three=trzy} - 随机顺序

// 打印所有唯一单词
System.out.println(biMap.keySet()); // 打印 [snow, two, one, three, ball] - 随机顺序
System.out.println(biMap.valueSet()); // 打印 [dwa, kula, jeden, snieg, trzy] - 随机顺序

// 按单词打印翻译
System.out.println("one = " + biMap.get("one")); // 打印 one = jeden
System.out.println("two = " + biMap.get("two")); // 打印 two = dwa
System.out.println("kula = " + biMap.inverse().get("kula")); // 打印 kula = ball
System.out.println("snieg = " + biMap.inverse().get("snieg")); // 打印 snieg = snow
System.out.println("empty = " + biMap.get("empty")); // 打印 empty = null

// 打印词对数量
System.out.println(biMap.size()); // 打印 5

```

3. Guava 的 HashMultiMap

```

String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// 解析文本为单词并建立索引
List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
// 创建多值映射
Multimap<String, Integer> multiMap = HashMultimap.create();

// 填充多值映射
int i = 0;
for(String word: words) {
    multiMap.put(word, i);
    i++;
}

// 打印所有单词
System.out.println(multiMap); // 打印 {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} - 键和值的随机顺序

// 打印所有唯一单词
System.out.println(multiMap.keySet()); // 打印 [Hi, Hello, World!, All!] - 随机顺序

// 打印所有索引
System.out.println("Hello = " + multiMap.get("Hello")); // 打印 [0, 2]
System.out.println("World = " + multiMap.get("World!")); // 打印 [1, 5]
System.out.println("All = " + multiMap.get("All!")); // 打印 [3]
System.out.println("Hi = " + multiMap.get("Hi")); // 打印 [4]

```

```
System.out.println(multiMap.keySet().size()); // print 4
```

2. HashBiMap from GS / Eclipse Collection

```

String[] englishWords = {"one", "two", "three", "ball", "snow"};
String[] russianWords = {"jeden", "dwa", "trzy", "kula", "snieg"};

// Create Multiset
MutableBiMap<String, String> biMap = new HashBiMap(englishWords.length);
// Create English-Polish dictionary
int i = 0;
for(String englishWord: englishWords) {
    biMap.put(englishWord, russianWords[i]);
    i++;
}

// Print count words
System.out.println(biMap); // print {two=dwa, ball=kula, one=jeden, snow=snieg, three=trzy} - in random orders

// Print all unique words
System.out.println(biMap.keySet()); // print [snow, two, one, three, ball] - in random orders
System.out.println(biMap.values()); // print [dwa, kula, jeden, snieg, trzy] - in random orders

// Print translate by words
System.out.println("one = " + biMap.get("one")); // print one = jeden
System.out.println("two = " + biMap.get("two")); // print two = dwa
System.out.println("kula = " + biMap.inverse().get("kula")); // print kula = ball
System.out.println("snieg = " + biMap.inverse().get("snieg")); // print snieg = snow
System.out.println("empty = " + biMap.get("empty")); // print empty = null

// Print count word's pair
System.out.println(biMap.size()); // print 5

```

3. HashMultiMap from Guava

```

String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Parse text to words and index
List<String> words = Arrays.asList(INPUT_TEXT.split(" "));
// Create Multimap
Multimap<String, Integer> multiMap = HashMultimap.create();

// Fill Multimap
int i = 0;
for(String word: words) {
    multiMap.put(word, i);
    i++;
}

// Print all words
System.out.println(multiMap); // print {Hi=[4], Hello=[0, 2], World!=[1, 5], All!=[3]} - keys and values in random orders

// Print all unique words
System.out.println(multiMap.keySet()); // print [Hi, Hello, World!, All!] - in random orders

// Print all indexes
System.out.println("Hello = " + multiMap.get("Hello")); // print [0, 2]
System.out.println("World = " + multiMap.get("World!")); // print [1, 5]
System.out.println("All = " + multiMap.get("All!")); // print [3]
System.out.println("Hi = " + multiMap.get("Hi")); // print [4]

```

```

System.out.println("Empty = " + multiMap.get("Empty")); // 打印 []
// 打印所有单词的数量
System.out.println(multiMap.size()); // 打印 6
// 打印唯一单词的数量
System.out.println(multiMap.keySet().size()); // 打印 4

```

```

System.out.println("Empty = " + multiMap.get("Empty")); // print []
// Print count all words
System.out.println(multiMap.size()); //print 6
// Print count unique words
System.out.println(multiMap.keySet().size()); //print 4

```

更多示例：

一、Apache 集合：

- 1.[MultiValueMap](#)
- 2.[MultiValueMapLinked](#)
- 3.[MultiValueMapTree](#)

二、GS / Eclipse 集合

- 1.[FastListMultimap](#)
- 2.[HashBagMultimap](#)
- 3.[TreeSortedSetMultimap](#)
- 4.[UnifiedSetMultimap](#)

III. 番石榴 (Guava)

- 1.[HashMultiMap](#)
- 2.[LinkedHashMultimap](#)
- 3.[LinkedListMultimap](#)
- 4.[TreeMultimap](#)
- 5.[ArrayListMultimap](#)

第62.2节：Apache HashBag、Guava HashMultiset 和 Eclipse HashBag

Bag/Multiset（袋/多重集合）存储集合中的每个对象及其出现次数。接口上的额外方法允许一次添加或移除对象的多个副本。JDK中的对应是 `HashMap<T, Integer>`，其中值是该键的副本计数。

类型	Guava	Apache Commons Collections	GS Collections	JDK
顺序未定义	HashMultiset	HashBag	HashBag	HashMap
排序	TreeMultiset	TreeBag	TreeBag	TreeMap
插入顺序	LinkedHashMultiset	-	-	LinkedHashMap
并发变体	ConcurrentHashMultiset	同步包	Collections.synchronizedMap(HashMap<String, Integer>)	
并发和排序	-	同步排序包	SynchronizedSortedBag	Collections.synchronizedMap(TreeMap<String, Integer>)
不可变集合	不可变多重集合	不可修改包	不可修改包	Collections.unmodifiableMap(HashMap<String, Integer>)
不可变且已排序	不可变排序多重集合	不可修改排序包	不可修改排序包	Collections.unmodifiableSortedMap(TreeMap<String, Integer>)

示例：

1. 使用 Apache 的 SynchronizedSortedBag:

```

// 解析文本以分离单词
String INPUT_TEXT = "Hello World! Hello All! Hi World!";

```

Nore examples:

I. Apache Collection:

- 1.[MultiValueMap](#)
- 2.[MultiValueMapLinked](#)
- 3.[MultiValueMapTree](#)

II. GS / Eclipse Collection

- 1.[FastListMultimap](#)
- 2.[HashBagMultimap](#)
- 3.[TreeSortedSetMultimap](#)
- 4.[UnifiedSetMultimap](#)

III. Guava

- 1.[HashMultiMap](#)
- 2.[LinkedHashMultimap](#)
- 3.[LinkedListMultimap](#)
- 4.[TreeMultimap](#)
- 5.[ArrayListMultimap](#)

Section 62.2: Apache HashBag, Guava HashMultiset and Eclipse HashBag

A Bag/multiset stores each object in the collection together with a count of occurrences. Extra methods on the interface allow multiple copies of an object to be added or removed at once. JDK analog is `HashMap<T, Integer>`, when values is count of copies this key.

Type	Guava	Apache Commons Collections	GS Collections	JDK
Order not defined	HashMultiset	HashBag	HashBag	HashMap
Sorted	TreeMultiset	TreeBag	TreeBag	TreeMap
Insertion-order	LinkedHashMultiset	-	-	LinkedHashMap
Concurrent variant	ConcurrentHashMultiset	SynchronizedBag	SynchronizedBag	Collections.synchronizedMap(HashMap<String, Integer>)
Concurrent and sorted	-	SynchronizedSortedBag	SynchronizedSortedBag	Collections.synchronizedSortedMap(TreeMap<String, Integer>)
Immutable collection	ImmutableMultiset	UnmodifiableBag	UnmodifiableBag	Collections.unmodifiableMap(HashMap<String, Integer>)
Immutable and sorted	ImmutableSortedMultiset	UnmodifiableSortedBag	UnmodifiableSortedBag	Collections.unmodifiableSortedMap(TreeMap<String, Integer>)

Examples:

1. Using SynchronizedSortedBag from Apache:

```

// Parse text to separate words
String INPUT_TEXT = "Hello World! Hello All! Hi World!";

```

```

// 创建多重集合
Bag bag = SynchronizedSortedBag.synchronizedBag(new TreeBag(Arrays.asList(INPUT_TEXT.split(" "))));

// 打印计数字词
System.out.println(bag); // 按自然(字母)顺序打印 [1:All!,2>Hello,1:Hi,2:World!]
// 打印所有唯一单词
System.out.println(bag.uniqueSet()); // 按自然(字母)顺序打印 [All!, Hello, Hi, World!]

// 打印单词出现次数
System.out.println("Hello = " + bag.getCount("Hello")); // 打印 2
System.out.println("World = " + bag.getCount("World!")); // 打印 2
System.out.println("All = " + bag.getCount("All!")); // 打印 1
System.out.println("Hi = " + bag.getCount("Hi")); // 打印 1
System.out.println("Empty = " + bag.getCount("Empty")); // 打印 0

// 打印所有单词总数
System.out.println(bag.size()); // 打印 6

// 打印唯一单词数量
System.out.println(bag.uniqueSet().size()); // 打印 4

```

2. 使用 Eclipse(GC) 的 TreeBag :

```

// 解析文本以分离单词
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// 创建多重集合
MutableSortedBag<String> bag = TreeBag.newBag(Arrays.asList(INPUT_TEXT.split(" ")));

// 打印计数字词
System.out.println(bag); // 打印 [All!, Hello, Hello, Hi, World!, World!] - 按自然顺序
// 打印所有唯一单词
System.out.println(bag.toSortedSet()); // 打印 [All!, Hello, Hi, World!] - 按自然顺序

// 打印单词出现次数
System.out.println("Hello = " + bag.occurrencesOf("Hello")); // 打印 2
System.out.println("World = " + bag.occurrencesOf("World!")); // 打印 2
System.out.println("All = " + bag.occurrencesOf("All!")); // 打印 1
System.out.println("Hi = " + bag.occurrencesOf("Hi")); // 打印 1
System.out.println("Empty = " + bag.occurrencesOf("Empty")); // 打印 0

// 打印所有单词数量
System.out.println(bag.size()); // 打印 6

// 打印唯一单词数量
System.out.println(bag.toSet().size()); // 打印 4

```

3. 使用 Guava 的 LinkedHashMultiset:

```

// 解析文本以分离单词
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// 创建多重集合
Multiset<String> multiset = LinkedHashMultiset.create(Arrays.asList(INPUT_TEXT.split(" ")));

// 打印计数字词
System.out.println(multiset); // 打印 [Hello x 2, World! x 2, All!, Hi] - 按可预测的迭代顺序
// 打印所有唯一单词
System.out.println(multiset.elementSet()); // 打印 [Hello, World!, All!, Hi] - 按可预测的迭代顺序

// 打印单词出现次数

```

```

// Create Multiset
Bag bag = SynchronizedSortedBag.synchronizedBag(new TreeBag(Arrays.asList(INPUT_TEXT.split(" "))));

// Print count words
System.out.println(bag); // print [1:All!,2>Hello,1:Hi,2:World!]- in natural (alphabet) order
// Print all unique words
System.out.println(bag.uniqueSet()); // print [All!, Hello, Hi, World!]- in natural (alphabet) order

// Print count occurrences of words
System.out.println("Hello = " + bag.getCount("Hello")); // print 2
System.out.println("World = " + bag.getCount("World!")); // print 2
System.out.println("All = " + bag.getCount("All!")); // print 1
System.out.println("Hi = " + bag.getCount("Hi")); // print 1
System.out.println("Empty = " + bag.getCount("Empty")); // print 0

// Print count all words
System.out.println(bag.size()); //print 6

// Print count unique words
System.out.println(bag.uniqueSet().size()); //print 4

```

2. Using TreeBag from Eclipse(GC):

```

// Parse text to separate words
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Create Multiset
MutableSortedBag<String> bag = TreeBag.newBag(Arrays.asList(INPUT_TEXT.split(" ")));

// Print count words
System.out.println(bag); // print [All!, Hello, Hello, Hi, World!, World!]- in natural order
// Print all unique words
System.out.println(bag.toSortedSet()); // print [All!, Hello, Hi, World!]- in natural order

// Print count occurrences of words
System.out.println("Hello = " + bag.occurrencesOf("Hello")); // print 2
System.out.println("World = " + bag.occurrencesOf("World!")); // print 2
System.out.println("All = " + bag.occurrencesOf("All!")); // print 1
System.out.println("Hi = " + bag.occurrencesOf("Hi")); // print 1
System.out.println("Empty = " + bag.occurrencesOf("Empty")); // print 0

// Print count all words
System.out.println(bag.size()); //print 6

// Print count unique words
System.out.println(bag.toSet().size()); //print 4

```

3. Using LinkedHashMultiset from Guava:

```

// Parse text to separate words
String INPUT_TEXT = "Hello World! Hello All! Hi World!";
// Create Multiset
Multiset<String> multiset = LinkedHashMultiset.create(Arrays.asList(INPUT_TEXT.split(" ")));

// Print count words
System.out.println(multiset); // print [Hello x 2, World! x 2, All!, Hi]- in predictable iteration order
// Print all unique words
System.out.println(multiset.elementSet()); // print [Hello, World!, All!, Hi] - in predictable iteration order

// Print count occurrences of words

```

```

System.out.println("Hello = " + multiset.count("Hello")); // 打印 2
System.out.println("World = " + multiset.count("World!")); // 打印 2
System.out.println("All = " + multiset.count("All!")); // 打印 1
System.out.println("Hi = " + multiset.count("Hi")); // 打印 1
System.out.println("Empty = " + multiset.count("Empty")); // 打印 0

// 打印所有单词数量
System.out.println(multiset.size()); // 打印 6

// 打印唯一单词数量
System.out.println(multiset.elementSet().size()); // 打印 4

```

```

System.out.println("Hello = " + multiset.count("Hello")); // print 2
System.out.println("World = " + multiset.count("World!")); // print 2
System.out.println("All = " + multiset.count("All!")); // print 1
System.out.println("Hi = " + multiset.count("Hi")); // print 1
System.out.println("Empty = " + multiset.count("Empty")); // print 0

// Print count all words
System.out.println(multiset.size()); //print 6

// Print count unique words
System.out.println(multiset.elementSet().size()); //print 4

```

更多示例：

一、Apache 集合：

1. [HashBag](#) - 顺序未定义
2. [SynchronizedBag](#) - 并发且顺序未定义
3. [SynchronizedSortedBag](#) - 并发且排序顺序
4. [TreeBag](#) - 排序顺序

二、GS / Eclipse 集合

5. [MutableBag](#) - 顺序未定义
6. [MutableSortedBag](#) - 排序顺序

III. 番石榴 (Guava)

7. [HashMultiset](#) - 顺序未定义
8. [TreeMultiset](#) - 排序顺序
9. [LinkedHashMultiset](#) - 插入顺序
10. [ConcurrentHashMultiset](#) - 并发且顺序未定义

第62.3节：与集合的比较操作 - 创建集合

与集合的比较操作 - 创建集合

1. 创建列表

描述	JDK	guava	gs-collections
创建空列表	new ArrayList<>()	Lists.newArrayList()	FastList.newArrayList()
从值创建列表	Arrays.asList("1", "2", Lists.newArrayList("1", "2", "3"))		FastList.newArrayListWith("1", "2", "3")
创建列表使用容量 = 100	new ArrayList<>(100)	Lists.newArrayListWithCapacity(100)	FastList.newArrayList(100)
从任何集合创建列表	new ArrayList<>(collection) 列表。newArrayList(collection)		FastList.newArrayList(collection)
从任何可迭代对象创建列表	-	Lists.newArrayList(iterable)	FastList.newArrayList(iterable)
从迭代器	-	Lists.newArrayList(iterator)	-

More examples:

I. Apache Collection:

1. [HashBag](#) - order not defined
2. [SynchronizedBag](#) - concurrent and order not defined
3. [SynchronizedSortedBag](#) - concurrent and sorted order
4. [TreeBag](#) - sorted order

II. GS / Eclipse Collection

5. [MutableBag](#) - order not defined
6. [MutableSortedBag](#) - sorted order

III. Guava

7. [HashMultiset](#) - order not defined
8. [TreeMultiset](#) - sorted order
9. [LinkedHashMultiset](#) - insertion order
10. [ConcurrentHashMultiset](#) - concurrent and order not defined

Section 62.3: Compare operation with collections - Create collections

Compare operation with collections - Create collections

1. Create List

Description	JDK	guava	gs-collections
Create empty list	new ArrayList<>()	Lists.newArrayList()	FastList.newArrayList()
Create list from values	Arrays.asList("1", "2", Lists.newArrayList("1", "2", "3"))		FastList.newArrayListWith("1", "2", "3")
Create list with capacity = 100	new ArrayList<>(100)	Lists.newArrayListWithCapacity(100)	FastList.newArrayList(100)
Create list from any collection	new ArrayList<>(collection)	Lists.newArrayList(collection)	FastList.newArrayList(collection)
Create list from any Iterable	-	Lists.newArrayList(iterable)	FastList.newArrayList(iterable)
Create list from Iterator	-	Lists.newArrayList(iterator)	-

从数组
创建列表

Arrays.asList(array)

Lists.newArrayList(array)

FastList.newArrayListWith(array)

使用工厂
创建
列表

-

FastList.newArrayListWithNValues(10,
(() -> "1"))

示例：

```
System.out.println("createArrayList start");
// 创建空列表
List<String> emptyGuava = Lists.newArrayList(); // 使用guava
List<String> emptyJDK = new ArrayList<>(); // 使用JDK
MutableList<String> emptyGS = FastList.newArrayList(); // 使用gs

// 创建包含100个元素的列表
List < String > exactly100 = Lists.newArrayListWithCapacity(100); // 使用guava
List<String> exactly100JDK = new ArrayList<>(100); // 使用JDK
MutableList<String> exactly100GS = FastList.newArrayList(100); // 使用gs

// 创建包含约100个元素的列表
List<String> approx100 = Lists.newArrayListWithExpectedSize(100); // 使用 guava
List<String> approx100JDK = new ArrayList<>(115); // 使用 JDK
MutableList<String> approx100GS = FastList.newArrayList(115); // 使用 gs

// 创建包含一些元素的列表
List<String> withElements = Lists.newArrayList("alpha", "beta", "gamma"); // 使用 guava
List<String> withElementsJDK = Arrays.asList("alpha", "beta", "gamma"); // 使用 JDK
MutableList<String> withElementsGS = FastList.newArrayListWith("alpha", "beta", "gamma"); // 使用 gs

System.out.println(withElements);
System.out.println(withElementsJDK);
System.out.println(withElementsGS);

// 从任何 Iterable 接口 (任何集合) 创建列表
Collection<String> collection = new HashSet<>(3);
collection.add("1");
collection.add("2");
collection.add("3");

List<String> fromIterable = Lists.newArrayList(collection); // 使用 guava
List<String> fromIterableJDK = new ArrayList<>(collection); // 使用 JDK
MutableList<String> fromIterableGS = FastList.newArrayList(collection); // 使用 gs

System.out.println(fromIterable);
System.out.println(fromIterableJDK);
System.out.println(fromIterableGS);
/* 注意：JDK 仅能从 Collection 创建列表，但 guava 和 gs 可以从
Iterable 和 Collection 创建列表 */

// 从任意 Iterator 创建列表
Iterator<String> iterator = collection.iterator();
List<String> fromIterator = Lists.newArrayList(iterator); // 使用 guava
System.out.println(fromIterator);

// 从任意数组创建列表
String[] array = {"4", "5", "6"};
List<String> fromArray = Lists.newArrayList(array); // 使用 guava
List<String> fromArrayJDK = Arrays.asList(array); // 使用 JDK
MutableList<String> fromArrayGS = FastList.newArrayListWith(array); // 使用 gs
System.out.println(fromArray);
```

Create list
from array
Create list
using
factory

Arrays.asList(array)

Lists.newArrayList(array)

FastList.newArrayListWith(array)

FastList.newArrayListWithNValues(10,
(() -> "1"))

Examples:

```
System.out.println("createArrayList start");
// Create empty list
List<String> emptyGuava = Lists.newArrayList(); // using guava
List<String> emptyJDK = new ArrayList<>(); // using JDK
MutableList<String> emptyGS = FastList.newArrayList(); // using gs

// Create list with 100 element
List < String > exactly100 = Lists.newArrayListWithCapacity(100); // using guava
List<String> exactly100JDK = new ArrayList<>(100); // using JDK
MutableList<String> exactly100GS = FastList.newArrayList(100); // using gs

// Create list with about 100 element
List<String> approx100 = Lists.newArrayListWithExpectedSize(100); // using guava
List<String> approx100JDK = new ArrayList<>(115); // using JDK
MutableList<String> approx100GS = FastList.newArrayList(115); // using gs

// Create list with some elements
List<String> withElements = Lists.newArrayList("alpha", "beta", "gamma"); // using guava
List<String> withElementsJDK = Arrays.asList("alpha", "beta", "gamma"); // using JDK
MutableList<String> withElementsGS = FastList.newArrayListWith("alpha", "beta", "gamma"); // using gs

System.out.println(withElements);
System.out.println(withElementsJDK);
System.out.println(withElementsGS);

// Create list from any Iterable interface (any collection)
Collection<String> collection = new HashSet<>(3);
collection.add("1");
collection.add("2");
collection.add("3");

List<String> fromIterable = Lists.newArrayList(collection); // using guava
List<String> fromIterableJDK = new ArrayList<>(collection); // using JDK
MutableList<String> fromIterableGS = FastList.newArrayList(collection); // using gs

System.out.println(fromIterable);
System.out.println(fromIterableJDK);
System.out.println(fromIterableGS);
/* Attention: JDK create list only from Collection, but guava and gs can create list from
Iterable and Collection */

// Create list from any Iterator
Iterator<String> iterator = collection.iterator();
List<String> fromIterator = Lists.newArrayList(iterator); // using guava
System.out.println(fromIterator);

// Create list from any array
String[] array = {"4", "5", "6"};
List<String> fromArray = Lists.newArrayList(array); // using guava
List<String> fromArrayJDK = Arrays.asList(array); // using JDK
MutableList<String> fromArrayGS = FastList.newArrayListWith(array); // using gs
System.out.println(fromArray);
```

```

System.out.println(fromArrayJDK);
System.out.println(fromArrayGS);

// 使用工厂方法创建列表
MutableList<String> fromFabricGS = FastList.newWithNValues(10, () ->
String.valueOf(Math.random())); // 使用 gs
System.out.println(fromFabricGS);

System.out.println("createArrayList end");

```

2 创建集合

描述	JDK	guava	gs-collections
创建空集合	new HashSet<>()	Sets.newHashSet()	UnifiedSet.newSet()
从值创建集合	new HashSet<>(Arrays.asList("alpha", "beta", "gamma"))	Sets.newHashSet("alpha", "beta", "gamma")	UnifiedSet.newSetWith("alpha", "beta", "gamma")
从任意集合创建集合	new HashSet<>(collection)	Sets.newHashSet(collection)	UnifiedSet.newSet(collection)
从任意可迭代对象创建集合	-	Sets.newHashSet(iterable)	UnifiedSet.newSet(iterable)
从任意迭代器创建集合	-	Sets.newHashSet(iterator)	-
从数组创建集合	new HashSet<>(Arrays.asList(array))	Sets.newHashSet(array)	UnifiedSet.newSetWith(array)

示例：

```

System.out.println("createHashSet start");
// 创建空集合
Set<String> emptyGuava = Sets.newHashSet(); // 使用 guava
Set<String> emptyJDK = new HashSet<>(); // 使用 JDK
Set<String> emptyGS = UnifiedSet.newSet(); // 使用 gs

// 创建包含100个元素的集合
Set<String> approx100 = Sets.newHashSetWithExpectedSize(100); // 使用 guava
Set<String> approx100JDK = new HashSet<>(130); // 使用 JDK
Set<String> approx100GS = UnifiedSet.newSet(130); // 使用 gs

// 从一些元素创建集合
Set<String> withElements = Sets.newHashSet("alpha", "beta", "gamma"); // 使用 guava
Set<String> withElementsJDK = new HashSet<>(Arrays.asList("alpha", "beta", "gamma")); // 使用 JDK
Set<String> withElementsGS = UnifiedSet.newSetWith("alpha", "beta", "gamma"); // 使用 gs

System.out.println(withElements);
System.out.println(withElementsJDK);
System.out.println(withElementsGS);

// 从任何 Iterable 接口 (任何集合) 创建集合
Collection<String> collection = new ArrayList<>(3);
collection.add("1");
collection.add("2");
collection.add("3");

Set<String> fromIterable = Sets.newHashSet(collection); // 使用 guava
Set<String> fromIterableJDK = new HashSet<>(collection); // 使用 JDK
Set<String> fromIterableGS = UnifiedSet.newSet(collection); // 使用 gs

```

Description	JDK	guava	gs-collections
Create empty set	new HashSet<>()	Sets.newHashSet()	UnifiedSet.newSet()
Create set from values	new HashSet<>(Arrays.asList("alpha", "beta", "gamma"))	Sets.newHashSet("alpha", "beta", "gamma")	UnifiedSet.newSetWith("alpha", "beta", "gamma")
Create set from any collections	new HashSet<>(collection)	Sets.newHashSet(collection)	UnifiedSet.newSet(collection)
Create set from any Iterable	-	Sets.newHashSet(iterable)	UnifiedSet.newSet(iterable)
Create set from any Iterator	-	Sets.newHashSet(iterator)	-
Create set from Array	new HashSet<>(Arrays.asList(array))	Sets.newHashSet(array)	UnifiedSet.newSetWith(array)

Examples:

```

System.out.println("createHashSet start");
// Create empty set
Set<String> emptyGuava = Sets.newHashSet(); // using guava
Set<String> emptyJDK = new HashSet<>(); // using JDK
Set<String> emptyGS = UnifiedSet.newSet(); // using gs

// Create set with 100 element
Set<String> approx100 = Sets.newHashSetWithExpectedSize(100); // using guava
Set<String> approx100JDK = new HashSet<>(130); // using JDK
Set<String> approx100GS = UnifiedSet.newSet(130); // using gs

// Create set from some elements
Set<String> withElements = Sets.newHashSet("alpha", "beta", "gamma"); // using guava
Set<String> withElementsJDK = new HashSet<>(Arrays.asList("alpha", "beta", "gamma")); // using JDK
Set<String> withElementsGS = UnifiedSet.newSetWith("alpha", "beta", "gamma"); // using gs

System.out.println(withElements);
System.out.println(withElementsJDK);
System.out.println(withElementsGS);

// Create set from any Iterable interface (any collection)
Collection<String> collection = new ArrayList<>(3);
collection.add("1");
collection.add("2");
collection.add("3");

Set<String> fromIterable = Sets.newHashSet(collection); // using guava
Set<String> fromIterableJDK = new HashSet<>(collection); // using JDK
Set<String> fromIterableGS = UnifiedSet.newSet(collection); // using gs

```

```

System.out.println(fromIterable);
System.out.println(fromIterableJDK);
System.out.println(fromIterableGS);
/* 注意:JDK 仅能从 Collection 创建集合, 但 guava 和 gs 可以从 Iterable
和 Collection 创建集合 */

// 从任何 Iterator 创建集合
Iterator<String> iterator = collection.iterator();
Set<String> fromIterator = Sets.newHashSet(iterator); // 使用 guava
System.out.println(fromIterator);

// 从任何数组创建集合
String[] array = {"4", "5", "6"};
Set<String> fromArray = Sets.newHashSet(array); // 使用 guava
Set<String> fromArrayJDK = new HashSet<>(Arrays.asList(array)); // 使用 JDK
Set<String> fromArrayGS = UnifiedSet.newSetWith(array); // 使用 gs
System.out.println(fromArray);
System.out.println(fromArrayJDK);
System.out.println(fromArrayGS);

System.out.println("createHashSet end");

```

3 创建地图

描述	JDK	guava	gs-collections
创建空映射	new HashMap<>()	Maps.newHashMap()	UnifiedMap.newMap()
创建容量为130的映射	new HashMap<>(130)Maps.newHashMapWithExpectedSize(100)	UnifiedMap.newMap(130)	
从其他映射创建映射	new HashMap<>(map)Maps.newHashMap(map)	UnifiedMap.newMap(map)	
从键创建映射	-	UnifiedMap.newWithKeysValues("1", "a", "2", "b")	UnifiedMap.newWithKeysValues("1", "a", "2", "b")

示例：

```

System.out.println("createHashMap start");
// 创建空映射
Map<String, String> emptyGuava = Maps.newHashMap(); // 使用 guava
Map<String, String> emptyJDK = new HashMap<>(); // 使用 JDK
Map<String, String> emptyGS = UnifiedMap.newMap(); // 使用 gs

// 创建包含约100个元素的映射
Map<String, String> approx100 = Maps.newHashMapWithExpectedSize(100); // 使用 guava
Map<String, String> approx100JDK = new HashMap<>(130); // 使用 JDK
Map<String, String> approx100GS = UnifiedMap.newMap(130); // 使用 gs

// 从另一个映射创建映射
Map<String, String> map = new HashMap<>(3);
map.put("k1", "v1");
map.put("k2", "v2");
Map<String, String> withMap = Maps.newHashMap(map); // 使用 guava
Map<String, String> withMapJDK = new HashMap<>(map); // 使用 JDK
Map<String, String> withMapGS = UnifiedMap.newMap(map); // 使用 gs

System.out.println(withMap);
System.out.println(withMapJDK);
System.out.println(withMapGS);

// 从键创建映射

```

```

System.out.println(fromIterable);
System.out.println(fromIterableJDK);
System.out.println(fromIterableGS);
/* Attention: JDK create set only from Collection, but guava and gs can create set from Iterable
and Collection */

// Create set from any Iterator
Iterator<String> iterator = collection.iterator();
Set<String> fromIterator = Sets.newHashSet(iterator); // using guava
System.out.println(fromIterator);

// Create set from any array
String[] array = {"4", "5", "6"};
Set<String> fromArray = Sets.newHashSet(array); // using guava
Set<String> fromArrayJDK = new HashSet<>(Arrays.asList(array)); // using JDK
Set<String> fromArrayGS = UnifiedSet.newSetWith(array); // using gs
System.out.println(fromArray);
System.out.println(fromArrayJDK);
System.out.println(fromArrayGS);

System.out.println("createHashSet end");

```

3 Create Map

Description	JDK	guava	gs-collections
Create empty map	new HashMap<>()	Maps.newHashMap()	UnifiedMap.newMap()
Create map with capacity = 130	new HashMap<>(130)	Maps.newHashMapWithExpectedSize(100)	UnifiedMap.newMap(130)
Create map from other map	new HashMap<>(map)	Maps.newHashMap(map)	UnifiedMap.newMap(map)
Create map from keys	-	-	UnifiedMap.newWithKeysValues("1", "a", "2", "b")

Examples:

```

System.out.println("createHashMap start");
// Create empty map
Map<String, String> emptyGuava = Maps.newHashMap(); // using guava
Map<String, String> emptyJDK = new HashMap<>(); // using JDK
Map<String, String> emptyGS = UnifiedMap.newMap(); // using gs

// Create map with about 100 element
Map<String, String> approx100 = Maps.newHashMapWithExpectedSize(100); // using guava
Map<String, String> approx100JDK = new HashMap<>(130); // using JDK
Map<String, String> approx100GS = UnifiedMap.newMap(130); // using gs

// Create map from another map
Map<String, String> map = new HashMap<>(3);
map.put("k1", "v1");
map.put("k2", "v2");
Map<String, String> withMap = Maps.newHashMap(map); // using guava
Map<String, String> withMapJDK = new HashMap<>(map); // using JDK
Map<String, String> withMapGS = UnifiedMap.newMap(map); // using gs

System.out.println(withMap);
System.out.println(withMapJDK);
System.out.println(withMapGS);

// Create map from keys

```

```
Map<String, String> withKeys = UnifiedMap.newWithKeysValues("1", "a", "2", "b");
System.out.println(withKeys);

System.out.println("createHashMap end");
```

更多示例：[CreateCollectionTest](#)

1. [CollectionCompare](#)
2. [CollectionSearch](#)
3. [JavaTransform](#)

```
Map<String, String> withKeys = UnifiedMap.newWithKeysValues("1", "a", "2", "b");
System.out.println(withKeys);

System.out.println("createHashMap end");
```

More examples: [CreateCollectionTest](#)

1. [CollectionCompare](#)
2. [CollectionSearch](#)
3. [JavaTransform](#)

第63章：并发集合

并发集合是一个[集合][1]，允许多个线程同时访问。不同的线程通常可以遍历集合内容并添加或删除元素。集合负责确保集合不会被破坏。[1]:

<http://stackoverflow.com/documentation/java/90/collections#t=201612221936497298484>

第63.1节：线程安全的集合

默认情况下，各种集合类型不是线程安全的。

但是，使集合线程安全相当容易。

```
List<String> threadSafeList = Collections.synchronizedList(new ArrayList<String>());
Set<String> threadSafeSet = Collections.synchronizedSet(new HashSet<String>());
Map<String, String> threadSafeMap = Collections.synchronizedMap(new HashMap<String, String>());
```

当你创建线程安全的集合时，绝不应该通过原始集合访问它，只能通过线程安全的包装器访问。

版本 ≥ Java SE 5

从Java 5开始，`java.util.collections` 提供了几种新的线程安全集合，不需要使用各种 `Collections.synchronized` 方法。

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

第63.2节：向`ConcurrentHashMap`插入数据

```
public class InsertIntoConcurrentHashMap
{
    public static void main(String[] args)
    {
        ConcurrentHashMap<Integer, SomeObject> concurrentHashMap = new ConcurrentHashMap<>();

        SomeObject value = new SomeObject();
        Integer key = 1;

        SomeObject previousValue = concurrentHashMap.putIfAbsent(1, value);
        if (previousValue != null)
        {
            //那么其他值已经映射到了键 = 1。传递给//putIfAbsent 方法的 'value' 未被插入，因此，任何其他调用//concurrentHashMap.get(1) 的线程都不会获得你线程尝试插入的 'value' 的引用。
            //请决定如何处理这种情况。
        }
        else
        {
            //value' 引用被映射到键 = 1。
        }
    }
}
```

Chapter 63: Concurrent Collections

A *concurrent collection* is a [collection][1] which permits access by more than one thread at the same time. Different threads can typically iterate through the contents of the collection and add or remove elements. The collection is responsible for ensuring that the collection doesn't become corrupt. [1]:

<http://stackoverflow.com/documentation/java/90/collections#t=201612221936497298484>

Section 63.1: Thread-safe Collections

By default, the various Collection types are not thread-safe.

However, it's fairly easy to make a collection thread-safe.

```
List<String> threadSafeList = Collections.synchronizedList(new ArrayList<String>());
Set<String> threadSafeSet = Collections.synchronizedSet(new HashSet<String>());
Map<String, String> threadSafeMap = Collections.synchronizedMap(new HashMap<String, String>());
```

When you make a thread-safe collection, you should never access it through the original collection, only through the thread-safe wrapper.

Version ≥ Java SE 5

Starting in Java 5, `java.util.collections` has several new thread-safe collections that don't need the various `Collections.synchronized` methods.

```
List<String> threadSafeList = new CopyOnWriteArrayList<String>();
Set<String> threadSafeSet = new ConcurrentHashSet<String>();
Map<String, String> threadSafeMap = new ConcurrentHashMap<String, String>();
```

Section 63.2: Insertion into ConcurrentHashMap

```
public class InsertIntoConcurrentHashMap
{
    public static void main(String[] args)
    {
        ConcurrentHashMap<Integer, SomeObject> concurrentHashMap = new ConcurrentHashMap<>();

        SomeObject value = new SomeObject();
        Integer key = 1;

        SomeObject previousValue = concurrentHashMap.putIfAbsent(1, value);
        if (previousValue != null)
        {
            //Then some other value was mapped to key = 1. 'value' that was passed to
            //putIfAbsent method is NOT inserted, hence, any other thread which calls
            //concurrentHashMap.get(1) would NOT receive a reference to the 'value'
            //that your thread attempted to insert. Decide how you wish to handle
            //this situation.
        }
        else
        {
            //value' reference is mapped to key = 1.
        }
    }
}
```

第63.3节：并发集合

并发集合是线程安全集合的推广，允许在并发环境中更广泛的使用。

虽然线程安全集合支持多个线程安全地添加或移除元素，但它们不一定支持在相同上下文中安全迭代（例如，一个线程可能无法安全地遍历集合，而另一个线程正在通过添加/移除元素修改集合）。

这就是并发集合的用武之地。

由于迭代通常是集合中多个批量方法（如addAll、removeAll，或通过构造函数或其他方式进行集合复制、排序等）的基础实现，因此并发集合的使用场景实际上非常广泛。

例如，Java SE 5中的java.util.concurrent.CopyOnWriteArrayList是一个线程安全且支持并发的List实现，其javado c说明：

“快照”风格的迭代器方法使用对创建迭代器时数组状态的引用。该数组在迭代器的生命周期内不会改变，因此不可能发生干扰，且保证迭代器不会抛出ConcurrentModificationException异常。

因此，以下代码是安全的：

```
public class ThreadSafeAndConcurrent {

    public static final List<Integer> LIST = new CopyOnWriteArrayList<>();

    public static void main(String[] args) throws InterruptedException {
        Thread modifier = new Thread(new ModifierRunnable());
        Thread iterator = new Thread(new IteratorRunnable());
        modifier.start();
        iterator.start();
        modifier.join();
        iterator.join();
    }

    public static final class ModifierRunnable implements Runnable {
        @Override
        public void run() {
            try {
                for (int i = 0; i < 50000; i++) {
                    LIST.add(i);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static final class IteratorRunnable implements Runnable {
        @Override
        public void run() {
            try {

```

Section 63.3: Concurrent Collections

Concurrent collections are a generalization of thread-safe collections, that allow for a broader usage in a concurrent environment.

While thread-safe collections have safe element addition or removal from multiple threads, they do not necessarily have safe iteration in the same context (one may not be able to safely iterate through the collection in one thread, while another one modifies it by adding/removing elements).

This is where concurrent collections are used.

As iteration is often the base implementation of several bulk methods in collections, like addAll, removeAll, or also collection copying (through a constructor, or other means), sorting, ... the use case for concurrent collections is actually pretty large.

For example, the Java SE 5 `java.util.concurrent.CopyOnWriteArrayList` is a thread safe and concurrent List implementation, its [javadoc](#) states :

The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created. This array never changes during the lifetime of the iterator, so interference is impossible and the iterator is guaranteed not to throw ConcurrentModificationException.

Therefore, the following code is safe :

```
public class ThreadSafeAndConcurrent {

    public static final List<Integer> LIST = new CopyOnWriteArrayList<>();

    public static void main(String[] args) throws InterruptedException {
        Thread modifier = new Thread(new ModifierRunnable());
        Thread iterator = new Thread(new IteratorRunnable());
        modifier.start();
        iterator.start();
        modifier.join();
        iterator.join();
    }

    public static final class ModifierRunnable implements Runnable {
        @Override
        public void run() {
            try {
                for (int i = 0; i < 50000; i++) {
                    LIST.add(i);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static final class IteratorRunnable implements Runnable {
        @Override
        public void run() {
            try {

```

```

for (int i = 0; i < 10000; i++) {
    long total = 0;
    for(Integer inList : LIST) {
        total += inList;
    }
    System.out.println(total);
}
} catch (Exception e) {
e.printStackTrace();
}
}
}
}

```

另一个关于迭代的并发集合是ConcurrentLinkedQueue，其说明为：

迭代器是弱一致性的，返回反映队列在迭代器创建时或之后某个时间点状态的元素。它们不会抛出java.util.ConcurrentModificationException，并且可以与其他操作并发进行。自迭代器创建以来包含在队列中的元素将被准确返回一次。

应该查看javadocs以确定一个集合是否是并发的。由iterator()方法返回的迭代器的属性（“快速失败”、“弱一致性”等）是最重要的属性。

线程安全但非并发的示例

在上述代码中，将LIST声明更改为

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

可能会（并且在大多数现代多CPU/核心架构上统计上会）导致异常。

来自Collections工具方法的同步集合在添加/删除元素时是线程安全的，但在迭代时不是（除非传入的底层集合本身就是线程安全的）。

```

for (int i = 0; i < 10000; i++) {
    long total = 0;
    for(Integer inList : LIST) {
        total += inList;
    }
    System.out.println(total);
}
} catch (Exception e) {
e.printStackTrace();
}
}
}
}

```

Another concurrent collection regarding iteration is [ConcurrentLinkedQueue](#), which states :

Iterators are weakly consistent, returning elements reflecting the state of the queue at some point at or since the creation of the iterator. They do not throw java.util.ConcurrentModificationException, and may proceed concurrently with other operations. Elements contained in the queue since the creation of the iterator will be returned exactly once.

One should check the javadocs to see if a collection is concurrent, or not. The attributes of the iterator returned by the iterator() method ("fail fast", "weakly consistent", ...) is the most important attribute to look for.

Thread safe but non concurrent examples

In the above code, changing the LIST declaration to

```
public static final List<Integer> LIST = Collections.synchronizedList(new ArrayList<>());
```

Could (and statistically will on most modern, multi CPU/core architectures) lead to exceptions.

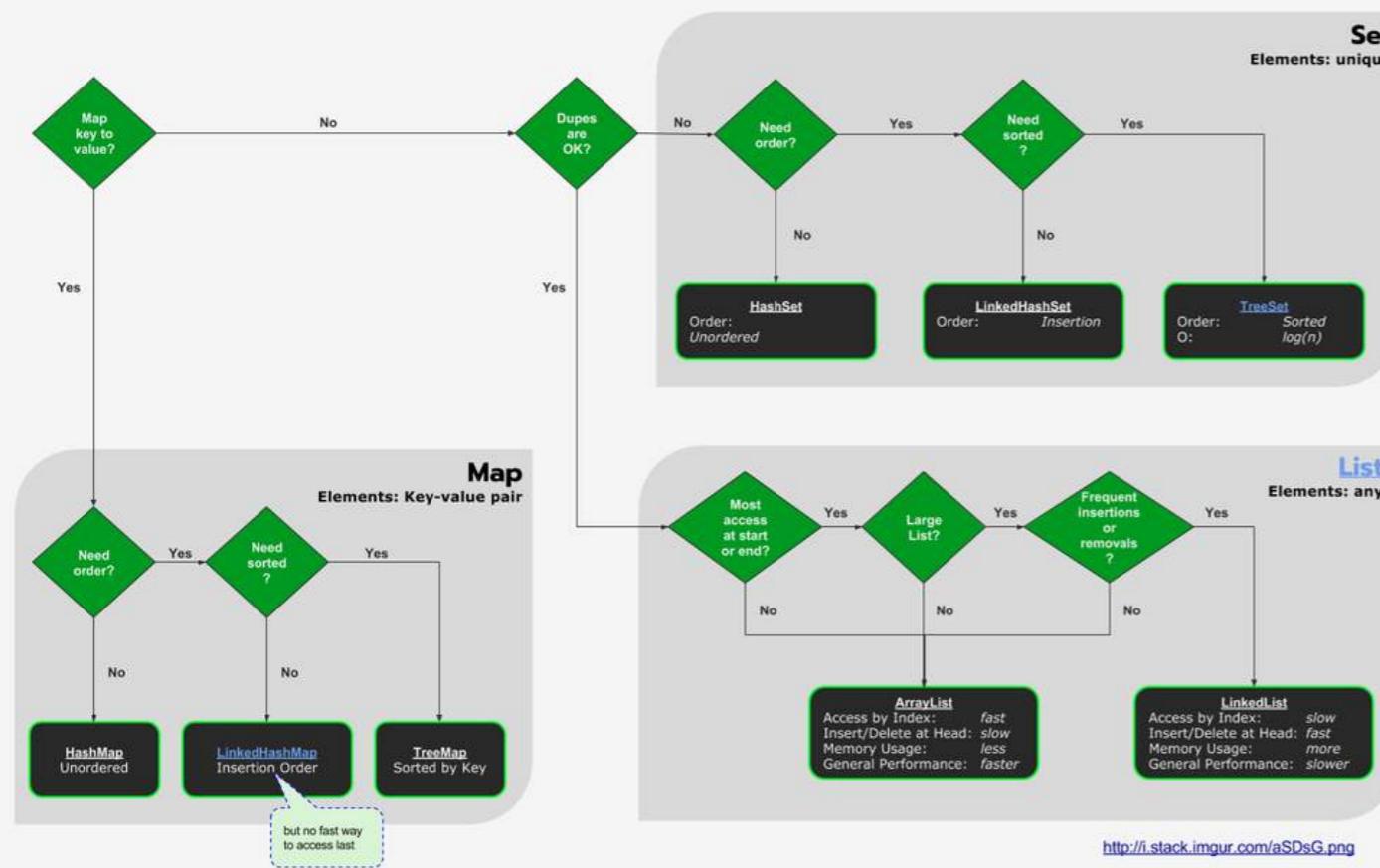
Synchronized collections from the [Collections](#) utility methods are thread safe for addition/removal of elements, but not iteration (unless the underlying collection being passed to it already is).

第64章：选择集合

Java提供了多种集合。选择使用哪种集合可能很棘手。请参见示例部分中的流程图，轻松选择适合任务的集合。

第64.1节：Java集合流程图

使用以下流程图选择适合任务的集合。



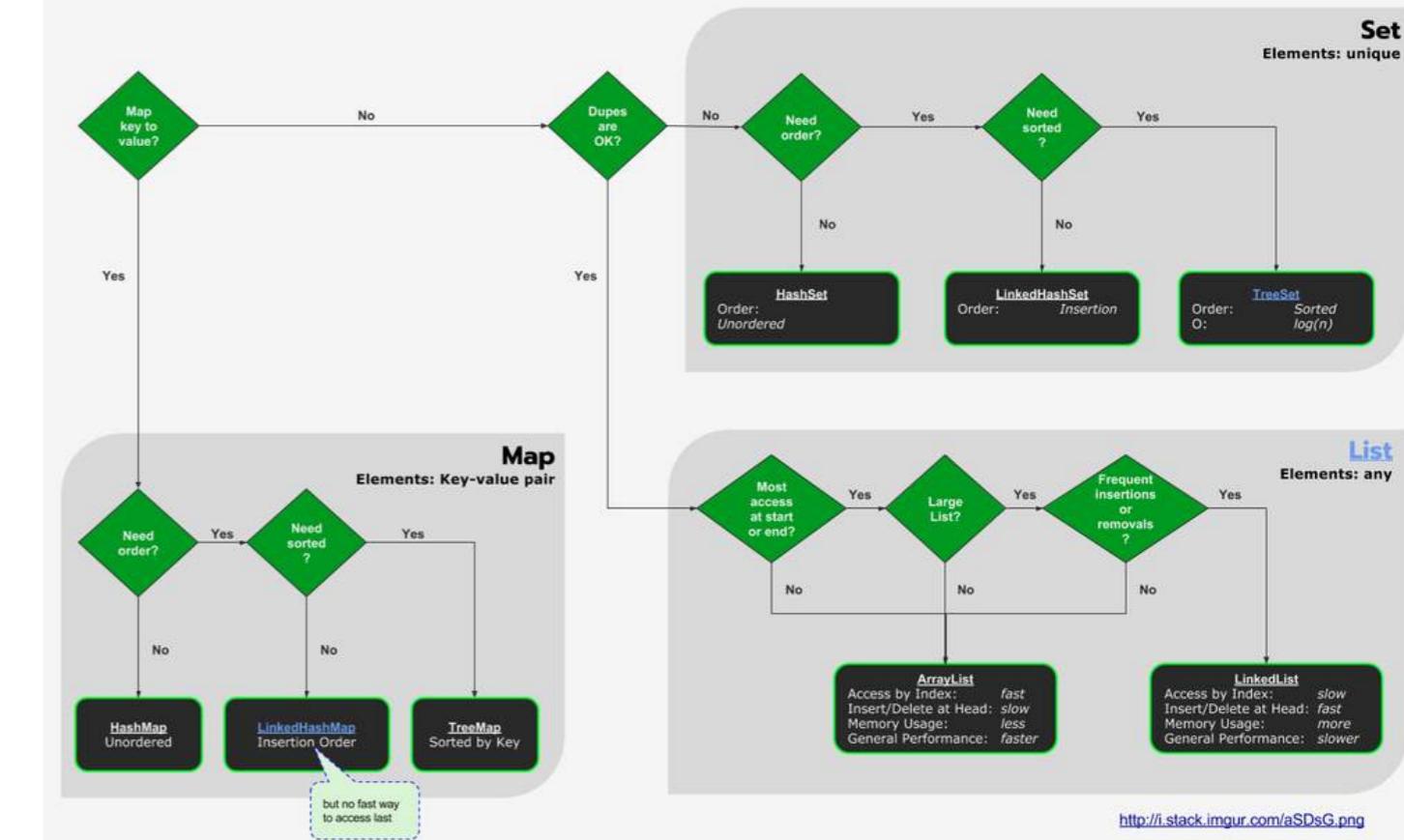
此流程图基于 [<http://i.stack.imgur.com/aSDsG.png>]。

Chapter 64: Choosing Collections

Java offers a wide variety of Collections. Choosing which Collection to use can be tricky. See the Examples section for an easy-to-follow flowchart to choose the right Collection for the job.

Section 64.1: Java Collections Flowchart

Use the following flowchart to choose the right Collection for the job.



This flowchart was based off [<http://i.stack.imgur.com/aSDsG.png>].

第65章：super关键字

第65.1节：super关键字的使用示例

super关键字在三个地方发挥重要作用

1. 构造函数级别
2. 方法级别
3. 变量级别

构造函数级别

super关键字用于调用父类构造函数。该构造函数可以是默认构造函数或带参数的构造函数。

- 默认构造函数：super();
- 带参数的构造函数：super(int no, double amount, String name);

```
class 父类
{
    父类(){
        System.out.println("父类的构造函数");
    }
}
class 子类 extends 父类
{
    子类(){
        /* 编译器会在此构造函数的第一行隐式添加 super()
         */
        System.out.println("子类的构造函数");
    }
    子类(int n1){
        /* 编译器会在此构造函数的第一行隐式添加 super()
         */
        System.out.println("带参数的构造函数");
    }
    void display(){
        System.out.println("你好");
    }
    public static void main(String args[]){
        // 使用默认构造函数创建对象
        子类 obj= new 子类();
        // 调用子类方法
        obj.display();
        // 使用带参数构造函数创建第二个对象
        子类 obj2= new 子类(10);
        obj2.display();
    }
}
```

Chapter 65: super keyword

Section 65.1: Super keyword use with examples

super keyword performs important role in three places

1. Constructor Level
2. Method Level
3. Variable Level

Constructor Level

super keyword is used to call parent class constructor. This constructor can be default constructor or parameterized constructor.

- Default constructor : super();
- Parameterized constructor : super(int no, double amount, String name);

```
class Parentclass
{
    Parentclass(){
        System.out.println("Constructor of Superclass");
    }
}
class Subclass extends Parentclass
{
    Subclass(){
        /* Compile adds super() here at the first line
         * of this constructor implicitly
         */
        System.out.println("Constructor of Subclass");
    }
    Subclass(int n1){
        /* Compile adds super() here at the first line
         * of this constructor implicitly
         */
        System.out.println("Constructor with arg");
    }
    void display(){
        System.out.println("Hello");
    }
    public static void main(String args[]){
        // Creating object using default constructor
        Subclass obj= new Subclass();
        // Calling sub class method
        obj.display();
        // Creating object 2 using arg constructor
        Subclass obj2= new Subclass(10);
        obj2.display();
    }
}
```

注意: super() 必须是构造函数中的第一条语句，否则我们将收到编译错误信息。

方法级别

Note: super() must be the first statement in constructor otherwise we will get the compilation error message.

Method Level

super关键字也可以用于方法重写的情况。super关键字可以用来调用父类的方法。

```
class 父类
{
    //重写的方法
    void display(){
        System.out.println("父类方法");
    }
}
class 子类 extends 父类
{
    //重写的方法
    void display(){
        System.out.println("子类方法");
    }
    void printMsg(){
        //这将调用重写的方法
        display();
        //这将调用被重写的方法
        super.display();
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printMsg();
    }
}
```

注意:如果没有方法重写, 则不需要使用 super 关键字来调用父类方法。

变量级别

super 用于引用直接父类的实例变量。在继承情况下, 基类和派生类可能有相似的数据成员。为了区分基类/父类和派生类/子类的数据成员, 在派生类的上下文中, 基类的数据成员必须以 super 关键字开头。

```
//父类或超类
class 父类
{
    int num=100;
}
//子类或子类
class Subclass extends Parentclass
{
    /* 我也在子类中声明了相同的变量
     * num.
     */
    int num=110;
    void printNumber(){
        System.out.println(num); // 它将打印值 110
        System.out.println(super.num); // 它将打印值 100
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
    }
}
```

super keyword can also be used in case of method overriding. super keyword can be used to invoke or call parent class method.

```
class Parentclass
{
    //Overridden method
    void display(){
        System.out.println("Parent class method");
    }
}
class Subclass extends Parentclass
{
    //Overriding method
    void display(){
        System.out.println("Child class method");
    }
    void printMsg(){
        //This would call Overriding method
        display();
        //This would call Overridden method
        super.display();
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printMsg();
    }
}
```

Note:If there is not method overriding then we do not need to use super keyword to call parent class method.

Variable Level

super is used to refer immediate parent class instance variable. In case of inheritance, there may be possibility of base class and derived class may have similar data members.In order to differentiate between the data member of base/parent class and derived/child class, in the context of derived class the base class data members must be preceded by super keyword.

```
//Parent class or Superclass
class Parentclass
{
    int num=100;
}
//Child class or subclass
class Subclass extends Parentclass
{
    /* I am declaring the same variable
     * num in child class too.
     */
    int num=110;
    void printNumber(){
        System.out.println(num); //It will print value 110
        System.out.println(super.num); //It will print value 100
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printNumber();
    }
}
```

注意: 如果我们在基类数据成员名称前没有写 `super` 关键字, 那么它将被视为当前类的数据成员, 并且基类的数据成员在派生类的上下文中被隐藏。

Note: If we are not writing `super` keyword before the base class data member name then it will be referred as current class data member and base class data member are hidden in the context of derived class.

第66章：序列化

Java 提供了一种机制，称为对象序列化，其中一个对象可以表示为一系列字节，这些字节包括对象的数据以及关于对象类型和存储在对象中的数据类型的信息。

序列化对象写入文件后，可以从文件中读取并反序列化，即可以使用表示对象及其数据的类型信息和字节在内存中重新创建该对象。

第66.1节：Java中的基本序列化

什么是序列化

序列化是将对象的状态（包括其引用）转换为字节序列的过程，同时也是在未来某个时间将这些字节重建为活动对象的过程。序列化用于当你想要持久化对象时。它也被Java RMI用于在JVM之间传递对象，无论是作为客户端到服务器的方法调用中的参数，还是作为方法调用的返回值，或作为远程方法抛出的异常。一般来说，当我们希望对象存在于JVM生命周期之外时，会使用序列化。

`java.io.Serializable` 是一个标记接口（没有方法体）。它仅用于“标记”Java类为可序列化。

序列化运行时为每个可序列化类关联一个版本号，称为 `serialVersionUID`，该版本号在 反序列化 过程中用于验证序列化对象的发送方和接收方是否加载了兼容的类。如果接收方加载的对象类的 `serialVersionUID` 与发送方对应类的不同，则反序列化将导致 `InvalidClassException`。可序列化类可以通过声明一个名为 `serialVersionUID` 的字段来显式声明自己的 `serialVersionUID`，该字段必须是 `static`、`final` 且类型为 `long`：

任何-访问-修饰符 `static final long serialVersionUID = 1L;`

如何使类具备序列化资格

要持久化对象，相应的类必须实现 `java.io.Serializable` 接口。

```
import java.io.Serializable;

public class SerialClass implements Serializable {

    private static final long serialVersionUID = 1L;
    private Date currentTime;

    public SerialClass() {
        currentTime = Calendar.getInstance().getTime();
    }

    public Date getCurrentTime() {
        return currentTime;
    }
}
```

如何将对象写入文件

现在我们需要将此对象写入文件系统。我们为此使用`java.io.ObjectOutputStream`。

```
import java.io.FileOutputStream;
```

Chapter 66: Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Section 66.1: Basic Serialization in Java

What is Serialization

Serialization is the process of converting an object's state (including its references) to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. Serialization is used when you want to persist the object. It is also used by Java RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from a method invocation, or as exceptions thrown by remote methods. In general, serialization is used when we want the object to exist beyond the lifetime of the JVM.

`java.io.Serializable` is a marker interface (has no body). It is just used to "mark" Java classes as serializable.

The serialization runtime associates with each serializable class a version number, called a `serialVersionUID`, which is used during *de-serialization* to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different `serialVersionUID` than that of the corresponding sender's class, then deserialization will result in an `InvalidClassException`. A serializable class can declare its own `serialVersionUID` explicitly by declaring a field named `serialVersionUID` that must be `static`, `final`, and of type `long`:

ANY-ACCESS-MODIFIER `static final long serialVersionUID = 1L;`

How to make a class eligible for serialization

To persist an object the respective class must implement the `java.io.Serializable` interface.

```
import java.io.Serializable;

public class SerialClass implements Serializable {

    private static final long serialVersionUID = 1L;
    private Date currentTime;

    public SerialClass() {
        currentTime = Calendar.getInstance().getTime();
    }

    public Date getCurrentTime() {
        return currentTime;
    }
}
```

How to write an object into a file

Now we need to write this object to a file system. We use `java.io.ObjectOutputStream` for this purpose.

```
import java.io.FileOutputStream;
```

```
import java.io.ObjectOutputStream;
import java.io.IOException;

public class PersistSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = new SerialClass(); //我们将把此对象写入文件系统。
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename));
            out.writeObject(time); //将字节流写入文件系统。
        out.close();
        } catch(IOException ex){
            ex.printStackTrace();
        }
    }
}
```

如何从序列化状态重新创建对象

可以使用java.io.ObjectInputStream从文件系统中读取存储的对象，示例如下：

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.lang.ClassNotFoundException;

public class ReadSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = null;

        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename));
            time = (SerialClass)in.readObject();
            in.close();
        } catch(IOException ex){
            ex.printStackTrace();
        } catch(ClassNotFoundException cnfe){
            cnfe.printStackTrace();
        }
        // 输出恢复的时间
        System.out.println("恢复的时间: " + time.getTime());
    }
}
```

序列化的类是二进制形式的。如果类定义发生变化，反序列化可能会出现问题：详情请参见Java序列化规范中的“序列化对象的版本控制”章节。

序列化一个对象会序列化以该对象为根的整个对象图，并且在存在循环图的情况下也能正确操作。提供了一个reset()方法，用于强制ObjectOutputStream忘记已经序列化过的对象。

瞬态字段 - 序列化

第66.2节：自定义序列化

在这个例子中，我们想创建一个类，该类将在初始化时接收的两个整数范围内生成一个随机数并输出到控制台。

```
import java.io.ObjectOutputStream;
import java.io.IOException;

public class PersistSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = new SerialClass(); //We will write this object to file system.
        try {
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename));
            out.writeObject(time); //Write byte stream to file system.
            out.close();
        } catch(IOException ex){
            ex.printStackTrace();
        }
    }
}
```

How to recreate an object from its serialized state

The stored object can be read from file system at later time using `java.io.ObjectInputStream` as shown below:

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.io.java.lang.ClassNotFoundException;

public class ReadSerialClass {

    public static void main(String [] args) {
        String filename = "time.ser";
        SerialClass time = null;

        try {
            ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename));
            time = (SerialClass)in.readObject();
            in.close();
        } catch(IOException ex){
            ex.printStackTrace();
        } catch(ClassNotFoundException cnfe){
            cnfe.printStackTrace();
        }
        // print out restored time
        System.out.println("Restored time: " + time.getTime());
    }
}
```

The serialized class is in binary form. The deserialization can be problematic if the class definition changes: see the [Versioning of Serialized Objects chapter of the Java Serialization Specification](#) for details.

Serializing an object serializes the entire object graph of which it is the root, and operates correctly in the presence of cyclic graphs. A `reset()` method is provided to force the `ObjectOutputStream` to forget about objects that have already been serialized.

Transient-fields - Serialization

Section 66.2: Custom Serialization

In this example we want to create a class that will generate and output to console, a random number between a

两个整数范围内生成一个随机数，该范围作为初始化时传入的参数。

```
public class SimpleRangeRandom implements Runnable {  
    private int min;  
    private int max;  
  
    private Thread thread;  
  
    public SimpleRangeRandom(int min, int max){  
        this.min = min;  
        this.max = max;  
        thread = new Thread(this);  
        thread.start();  
    }  
  
    @Override  
    private void WriteObject(ObjectOutputStream out) throws IOException;  
    private void ReadObject(ObjectInputStream in) throws IOException, ClassNotFoundException;  
    public void run() {  
        while(true){  
            Random rand = new Random();  
            System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max - min));  
            try {  
                Thread.sleep(10000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

现在如果我们想让这个类可序列化，会有一些问题。Thread 是某些系统级类之一，不可序列化。因此我们需要将 thread 声明为 transient。这样做我们可以序列化该类的对象，但仍然会有问题。如你所见，在构造函数中我们设置了随机数生成器的最小值和最大值，之后启动了负责生成和打印随机值的线程。因此，当通过调用 readObject() 恢复持久化对象时，构造函数不会再次运行，因为没有创建新对象。在这种情况下，我们需要开发一个自定义

序列化，通过在类中提供两个方法来实现。这些方法是：

```
private void writeObject(ObjectOutputStream out) throws IOException;  
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

因此，通过在 readObject() 中添加我们的实现，我们可以初始化并启动我们的线程：

```
class RangeRandom implements Serializable, Runnable {  
  
    private int min;  
    private int max;  
  
    private transient Thread thread;  
    //transient 应该用于任何无法序列化的字段，例如 Thread，或者任何你不想被序列化的字段  
  
    public RangeRandom(int min, int max){  
        this.min = min;  
        this.max = max;  
        thread = new Thread(this);  
        thread.start();  
    }
```

range of two integers which are passed as arguments during the initialization.

```
public class SimpleRangeRandom implements Runnable {  
    private int min;  
    private int max;  
  
    private Thread thread;  
  
    public SimpleRangeRandom(int min, int max){  
        this.min = min;  
        this.max = max;  
        thread = new Thread(this);  
        thread.start();  
    }  
  
    @Override  
    private void WriteObject(ObjectOutputStream out) throws IOException;  
    private void ReadObject(ObjectInputStream in) throws IOException, ClassNotFoundException;  
    public void run() {  
        while(true){  
            Random rand = new Random();  
            System.out.println("Thread: " + thread.getId() + " Random:" + rand.nextInt(max - min));  
            try {  
                Thread.sleep(10000);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Now if we want to make this class Serializable there will be some problems. The Thread is one of the certain system-level classes that are not Serializable. So we need to declare the thread as **transient**. By doing this we will be able to serialize the objects of this class but we will still have an issue. As you can see in the constructor we set the min and the max values of our randomizer and after this we start the thread which is responsible for generating and printing the random value. Thus when restoring the persisted object by calling the **readObject()** the constructor will not run again as there is no creation of a new object. In that case we need to develop a **Custom Serialization** by providing two methods inside the class. Those methods are:

```
private void writeObject(ObjectOutputStream out) throws IOException;  
private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;
```

Thus by adding our implementation in the **readObject()** we can initiate and start our thread:

```
class RangeRandom implements Serializable, Runnable {  
  
    private int min;  
    private int max;  
  
    private transient Thread thread;  
    //transient should be any field that either cannot be serialized e.g Thread or any field you do not want serialized  
  
    public RangeRandom(int min, int max){  
        this.min = min;  
        this.max = max;  
        thread = new Thread(this);  
        thread.start();  
    }
```

```

@Override
public void run() {
    while(true) {
        Random rand = new Random();
        System.out.println("线程: " + thread.getId() + " 随机数: " + rand.nextInt(max - min));
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    thread = new Thread(this);
    thread.start();
}

String[] args) {
    System.out.println("Hello");
    RangeRandom rangeRandom = new RangeRandom(1, 10);

    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try {
        fos = new FileOutputStream("test");
        out = new ObjectOutputStream(fos);
        out.writeObject(rangeRandom);
    } catch(IOException ex) {
        ex.printStackTrace();
    }

    RangeRandom rangeRandom2 = null;
    FileInputStream fis = null;
    ObjectInputStream in = null;
    try {
        fis = new FileInputStream("test");
        in = new ObjectInputStream(fis);
        rangeRandom2 = (RangeRandom)in.readObject();
        in.close();
    } catch(IOException ex) {
        ex.printStackTrace();
    } catch(ClassNotFoundException ex)
}

```

```

@Override
public void run() {
    while(true) {
        Random rand = new Random();
        System.out.println("Thread: " + thread.getId() + " Random: " + rand.nextInt(max - min));
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

private void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    thread = new Thread(this);
    thread.start();
}

```

Here is the main for our example:

```

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello");
        RangeRandom rangeRandom = new RangeRandom(1, 10);

        FileOutputStream fos = null;
        ObjectOutputStream out = null;
        try {
            fos = new FileOutputStream("test");
            out = new ObjectOutputStream(fos);
            out.writeObject(rangeRandom);
            out.close();
        } catch(IOException ex) {
            ex.printStackTrace();
        }

        RangeRandom rangeRandom2 = null;
        FileInputStream fis = null;
        ObjectInputStream in = null;
        try {
            fis = new FileInputStream("test");
            in = new ObjectInputStream(fis);
            rangeRandom2 = (RangeRandom)in.readObject();
            in.close();
        } catch(IOException ex) {
            ex.printStackTrace();
        } catch(ClassNotFoundException ex)
}

```

```
{  
ex.printStackTrace();  
}
```

```
{  
    ex.printStackTrace();  
}  
}  
}
```

If you run the main you will see that there are two threads running for each RangeRandom instance and that is because the `Thread.start()` method is now in both the constructor and the `readObject()`.

Section 66.3: Versioning and serialVersionUID

`final` 类型为 `long` 的字段，名为 `serialVersionUID`。如果类没有显式声明该字段，编译器会创建这样一个字段，并赋予一个值，该值来自于一个依赖于实现的 `serialVersionUID` 计算。这个计算依赖于类的各个方面，并遵循 Sun 提供的 [对象序列化规范 \(Object Serialization Specifications\)](#)。但是，该值不能保证在所有编译器实现中都相同。

该值用于检查类在序列化方面的兼容性，这个检查在反序列化保存的对象时进行。序列化运行时会验证从反序列化数据中读取的 `serialVersionUID` 与类中声明的 `serialVersionUID` 是否完全相同。如果不相同，则会抛出一个 `InvalidClassException` 异常。

强烈建议你在所有想要实现序列化的类中显式声明并初始化一个静态的、`final` 类型为 `long` 的字段，名为 'serialVersionUID'，而不是依赖该字段的默认计算值，即使你不打算使用版本控制。`'serialVersionUID'` 的计算非常敏感，可能因不同的编译器实现而异，因此即使是同一个类，如果在序列化过程的发送端和接收端使用了不同的编译器实现，也可能导致抛出 `InvalidClassException` 异常。

```
public class Example implements Serializable {  
    static final long serialVersionUID = 1L /*或其他某个值*/;  
    //...  
}
```

只要 `serialVersionUID` 相同，Java 序列化就能处理类的不同版本。兼容和不兼容的更改包括：

兼容的更改

- 添加字段：当被重构的类有一个字段在序列化流中不存在时，该对象中的该字段将被初始化为其类型的默认值。如果需要类特定的初始化，类可以提供一个 `readObject` 方法来将该字段初始化为非默认值。
- 添加类：流中将包含流中每个对象的类型层次结构。将流中的该层次结构与当前类进行比较可以检测到额外的类。由于流中没有用于初始化对象的信息，因此类的字段将被初始化为默认值。
- 删除类：将流中的类层次结构与当前类的层次结构进行比较可以检测到某个类已被删除。在这种情况下，将从流中读取对应于该类的字段和对象。基本类型字段将被丢弃，但被删除类引用的对象会被创建，因为它们可能在流的后续部分被引用。当流被垃圾回收或重置时，这些对象也将被垃圾回收。
- 添加 `writeObject/readObject` 方法：如果读取流的版本具有这些方法，则通常期望 `readObject` 读取由默认序列化写入流中的所需数据。

When you implement `java.io.Serializable` interface to make a class serializable, the compiler looks for a `static final` field named `serialVersionUID` of type `long`. If the class doesn't have this field declared explicitly then the compiler will create one such field and assign it with a value which comes out of a implementation dependent computation of `serialVersionUID`. This computation depends upon various aspects of the class and it follows the [Object Serialization Specifications](#) given by Sun. But, the value is not guaranteed to be the same across all compiler implementations.

This value is used for checking the compatibility of the classes with respect to serialization and this is done while de-serializing a saved object. The Serialization Runtime verifies that `serialVersionUID` read from the de-serialized data and the `serialVersionUID` declared in the class are exactly the same. If that is not the case, it throws an `InvalidClassException`.

It's highly recommended that you explicitly declare and initialize the static, final field of type long and named '`serialVersionUID`' in all your classes you want to make `Serializable` instead of relying on the default computation of the value for this field even if you are not gonna use versioning. **'serialVersionUID' computation is extremely sensitive and may vary from one compiler implementation to another and hence you may turn up getting the `InvalidClassException` even for the same class just because you used different compiler implementations on the sender and the receiver ends of the serialization process.**

```
public class Example implements Serializable {  
    static final long serialVersionUID = 1L /*or some other value*/;  
    //...  
}
```

As long as `serialVersionUID` is the same, Java Serialization can handle different versions of a class. Compatible and incompatible changes are:

Compatible Changes

- Adding fields :** When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type. If class-specific initialization is needed, the class may provide a `readObject` method that can initialize the field to nondefault values.
- Adding classes :** The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes. Since there is no information in the stream from which to initialize the object, the class's fields will be initialized to the default values.
- Removing classes :** Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted. In this case, the fields and objects corresponding to that class are read from the stream. Primitive fields are discarded, but the objects referenced by the deleted class are created, since they may be referred to later in the stream. They will be garbage-collected when the stream is garbage-collected or reset.
- Adding writeObject/readObject methods :** If the version reading the stream has these methods then `readObject` is expected, as usual, to read the required data written to the stream by the default serialization.

在读取任何可选数据之前，应先调用 `defaultReadObject`。`writeObject` 方法通常期望调用 `defaultWriteObject` 来写入所需数据，然后可以写入可选数据。

- 添加 `java.io.Serializable`：这相当于添加类型。流中不会有该类的值，因此其字段将被初始化为默认值。支持非序列化类的子类化要求该类的超类具有无参构造函数，且该类本身将被初始化为默认值。如果无参构造函数不可用，则会抛出 `InvalidClassException`。
- 更改字段的访问权限：访问修饰符 `public`、包访问权限（`package`）、`protected` 和 `private` 对序列化赋值字段的能力没有影响。
- 将字段从静态改为非静态或从瞬态改为非瞬态：当依赖默认序列化计算可序列化字段时，此更改相当于向类中添加字段。新字段将被写入流，但早期版本的类会忽略该值，因为序列化不会为静态或瞬态字段赋值。

不兼容的更改

- 删除字段：如果类中删除了字段，写入的流将不包含该字段的值。当流被早期版本的类读取时，该字段的值将被设置为默认值，因为流中没有该值。然而，这个默认值可能会不利地影响早期版本履行其契约的能力。
- 在层次结构中上下移动类：这是不允许的，因为流中的数据顺序将出现错误。
- 将非静态字段更改为静态字段或将非瞬态字段更改为瞬态字段：当依赖默认序列化时，此更改等同于从类中删除字段。该版本的类将不会将该数据写入流，因此早期版本的类将无法读取该数据。与删除字段时一样，早期版本的字段将被初始化为默认值，这可能导致类以意想不到的方式失败。
- 更改原始字段的声明类型：类的每个版本都会使用其声明的类型写入数据。早期版本的类尝试读取该字段时会失败，因为流中的数据类型与字段的类型不匹配。
- 更改 `writeObject` 或 `readObject` 方法，使其不再写入或读取默认字段数据，或更改为尝试写入或读取之前版本未处理的默认字段数据。默认字段数据必须始终一致地出现在流中或不出现。
- 将类从 `Serializable` 更改为 `Externalizable` 或反之是一个不兼容的更改，因为流中将包含与可用类的实现不兼容的数据。
- 将类从非枚举类型更改为枚举类型或反之是不兼容的更改，因为流中将包含与可用类的实现不兼容的数据。
- 移除 `Serializable` 或 `Externalizable` 中的任意一个是不兼容的更改，因为写入时将不再提供旧版本类所需的字段。
- 向类中添加 `writeReplace` 或 `readResolve` 方法是不兼容的，如果其行为会生成与任何旧版本类不兼容的对象。

第 66.4 节：使用 Gson 进行序列化

使用 Gson 进行序列化很简单，并且会输出正确的 JSON。

```
public class Employe {  
  
    private String firstName;  
    private String lastName;  
    private int age;  
    private BigDecimal salary;  
    private List<String> skills;
```

It should call `defaultReadObject` first before reading any optional data. The `writeObject` method is expected as usual to call `defaultWriteObject` to write the required data and then may write optional data.

- Adding `java.io.Serializable`**：This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class's supertype have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the `InvalidClassException` is thrown.
- Changing the access to a field**：The access modifiers `public`, `package`, `protected`, and `private` have no effect on the ability of serialization to assign values to the fields.
- Changing a field from static to nonstatic or transient to nontransient**：When relying on default serialization to compute the serializable fields, this change is equivalent to adding a field to the class. The new field will be written to the stream but earlier classes will ignore the value since serialization will not assign values to static or transient fields.

Incompatible Changes

- Deleting fields**：If a field is deleted in a class, the stream written will not contain its value. When the stream is read by an earlier class, the value of the field will be set to the default value because no value is available in the stream. However, this default value may adversely impair the ability of the earlier version to fulfill its contract.
- Moving classes up or down the hierarchy**：This cannot be allowed since the data in the stream appears in the wrong sequence.
- Changing a nonstatic field to static or a nontransient field to transient**：When relying on default serialization, this change is equivalent to deleting a field from the class. This version of the class will not write that data to the stream, so it will not be available to be read by earlier versions of the class. As when deleting a field, the field of the earlier version will be initialized to the default value, which can cause the class to fail in unexpected ways.
- Changing the declared type of a primitive field**：Each version of the class writes the data with its declared type. Earlier versions of the class attempting to read the field will fail because the type of the data in the stream does not match the type of the field.
- Changing the `writeObject` or `readObject` method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not. The default field data must consistently either appear or not appear in the stream.
- Changing a class from `Serializable` to `Externalizable` or vice versa is an incompatible change since the stream will contain data that is incompatible with the implementation of the available class.
- Changing a class from a non-enum type to an enum type or vice versa since the stream will contain data that is incompatible with the implementation of the available class.
- Removing either `Serializable` or `Externalizable` is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.
- Adding the `writeReplace` or `readResolve` method to a class is incompatible if the behavior would produce an object that is incompatible with any older version of the class.

Section 66.4: Serialization with Gson

Serialization with Gson is easy and will output correct JSON.

```
public class Employe {  
  
    private String firstName;  
    private String lastName;  
    private int age;  
    private BigDecimal salary;  
    private List<String> skills;
```

```

    //获取器和设置器
}

(序列化)

//技能
List<String> skills = new LinkedList<String>();
skills.add("领导力");
skills.add("Java经验");

//员工
Employe obj = new Employe();
obj.setFirstName("克里斯蒂安");
obj.setLastName("卢萨迪");
obj.setAge(25);
obj.setSalary(new BigDecimal("10000"));
obj.setSkills(skills);

//序列化过程
Gson gson = new Gson();
String json = gson.toJson(obj);
//{"firstName":"克里斯蒂安","lastName":"卢萨迪","age":25,"salary":10000,"skills":["领导力","Java
经验"]}

```

请注意，不能序列化具有循环引用的对象，因为这会导致无限递归。

(反序列化)

```

//这非常简单.....
//假设 json 是之前的字符串对象.....

Employe obj2 = gson.fromJson(json, Employe.class); // obj2 就像 obj 一样

```

第66.5节：使用 Jackson 进行自定义 JSON 反序列化

我们以 JSON 格式消费 REST API，然后将其解组为 POJO。Jackson's org.codehaus.jackson.map.ObjectMapper “开箱即用”，在大多数情况下我们实际上不需要做任何事情。但有时我们需要自定义反序列化器来满足自定义需求，本教程将引导你完成创建自定义反序列化器的过程。

假设我们有以下实体。

```

public class User {
    private Long id;
    private String name;
    private String email;

    //getter setter 为简洁起见省略
}

```

和

```

公共类 Program {
    私有 Long id;
    私有 String name;
    私有 User createdBy;
    私有 String contents;
}

```

```

    //getters and setters
}

(Serialization)

//Skills
List<String> skills = new LinkedList<String>();
skills.add("leadership");
skills.add("Java Experience");

//Employe
Employe obj = new Employe();
obj.setFirstName("Christian");
obj.setLastName("Lusardi");
obj.setAge(25);
obj.setSalary(new BigDecimal("10000"));
obj.setSkills(skills);

//Serialization process
Gson gson = new Gson();
String json = gson.toJson(obj);
//{"firstName":"Christian","lastName":"Lusardi","age":25,"salary":10000,"skills":["leadership","Java
Experience"]}

```

Note that you can not serialize objects with circular references since that will result in infinite recursion.

(Deserialization)

```

//it's very simple...
//Assuming that json is the previous String object.....

Employe obj2 = gson.fromJson(json, Employe.class); // obj2 is just like obj

```

Section 66.5: Custom JSON Deserialization with Jackson

We consume rest API as a JSON format and then unmarshal it to a POJO. Jackson's org.codehaus.jackson.map.ObjectMapper “just works” out of the box and we really don't do anything in most cases. But sometimes we need custom deserializer to fulfill our custom needs and this tutorial will guide you through the process of creating your own custom deserializer.

Let's say we have following entities.

```

public class User {
    private Long id;
    private String name;
    private String email;

    //getter setter are omitted for clarity
}

```

And

```

public class Program {
    private Long id;
    private String name;
    private User createdBy;
    private String contents;
}

```

```
//getter setter 为简洁起见省略
```

```
}
```

让我们先序列化/编组一个对象。

```
User user = new User();
user.setId(1L);
user.setEmail("example@example.com");
user.setName("Bazlur Rahman");

Program program = new Program();
program.setId(1L);
program.setName("Program @# 1");
program.setCreatedBy(user);
program.setContents("Some contents");

ObjectMapper objectMapper = new ObjectMapper();

final String json = objectMapper.writeValueAsString(program); System.out.println(json);
```

上述代码将生成以下 JSON-

```
{
    "id": 1,
    "name": "Program @# 1",
    "createdBy": {
        "id": 1,
        "name": "巴兹鲁尔·拉赫曼",
        "email": "example@example.com"
    },
    "contents": "一些内容"
}
```

现在可以非常容易地做相反的操作。如果我们有这个 JSON，可以使用

ObjectMapper 将其反序列化为一个程序对象，方法如下-

现在假设，这不是实际情况，我们将从一个 API 获得一个不同的 JSON，它与我们的Program类不匹配。

```
{
    "id": 1,
    "name": "Program @# 1",
    "ownerId": 1
    "contents": "一些内容"
}
```

看看这个 JSON 字符串，你可以看到，它有一个不同的字段，叫做 ownerId。

现在如果你想像之前那样序列化这个 JSON，会出现异常。

有两种方法可以避免异常并完成序列化-

忽略未知字段

忽略ownerId。在Program类中添加以下注解

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}
```

```
//getter setter are omitted for clarity
```

```
}
```

Let's serialize/marshal an object first.

```
User user = new User();
user.setId(1L);
user.setEmail("example@example.com");
user.setName("Bazlur Rahman");

Program program = new Program();
program.setId(1L);
program.setName("Program @# 1");
program.setCreatedBy(user);
program.setContents("Some contents");

ObjectMapper objectMapper = new ObjectMapper();
```

```
final String json = objectMapper.writeValueAsString(program); System.out.println(json);
```

The above code will produce following JSON-

```
{
    "id": 1,
    "name": "Program @# 1",
    "createdBy": {
        "id": 1,
        "name": "Bazlur Rahman",
        "email": "example@example.com"
    },
    "contents": "Some contents"
}
```

Now can do the opposite very easily. If we have this JSON, we can unmarshal to a program object using ObjectMapper as following -

Now let's say, this is not the real case, we are going to have a different JSON from an API which doesn't match with our Program class.

```
{
    "id": 1,
    "name": "Program @# 1",
    "ownerId": 1
    "contents": "Some contents"
}
```

Look at the JSON string, you can see, it has a different field that is ownerId.

Now if you want to serialize this JSON as we did earlier, you will have exceptions.

There are two ways to avoid exceptions and have this serialized -

Ignore the unknown fields

Ignore the ownerId. Add the following annotation in the Program class

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Program {}
```

编写自定义反序列化器

但有些情况下你实际上需要这个ownerId字段。比如你想将其作为User

类的id关联。

在这种情况下，你需要编写一个自定义反序列化器-

如你所见，首先你必须从JsonParser访问JsonNode。然后你可以使用get()方法轻松地从JsonNode中提取信息。你必须确保字段名正确，拼写错误会导致异常。

最后，你必须将你的ProgramDeserializer注册到ObjectMapper中。

```
ObjectMapper mapper = new ObjectMapper();
SimpleModule module = new SimpleModule();
module.addDeserializer(Program.class, new ProgramDeserializer());

mapper.registerModule(module);

String newJsonString = "{\"id\":1,\"name\":\"Program #@ 1\", \"ownerId\":1, \"contents\":\"Some contents\"}";
final Program program2 = mapper.readValue(newJsonString, Program.class);
```

或者，您可以使用注解直接注册反序列化器 -

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
```

Write custom deserializer

But there are cases when you actually need this ownerId field. Let's say you want to relate it as an id of the User class.

In such case, you need to write a custom deserializer-

As you can see, first you have to access the JsonNode from the JsonParser. And then you can easily extract information from a JsonNode using the get() method. and you have to make sure about the field name. It should be the exact name, spelling mistake will cause exceptions.

And finally, you have to register your ProgramDeserializer to the ObjectMapper.

```
ObjectMapper mapper = new ObjectMapper();
SimpleModule module = new SimpleModule();
module.addDeserializer(Program.class, new ProgramDeserializer());

mapper.registerModule(module);

String newJsonString = "{\"id\":1,\"name\":\"Program #@ 1\", \"ownerId\":1, \"contents\":\"Some contents\"}";
final Program program2 = mapper.readValue(newJsonString, Program.class);
```

Alternatively, you can use annotation to register the deserializer directly -

```
@JsonDeserialize(using = ProgramDeserializer.class)
public class Program {
```

第67章：Optional

Optional 是一个容器对象，可能包含也可能不包含非空值。如果值存在，`isPresent()` 将返回 `true`，`get()` 将返回该值。

还提供了依赖于值存在与否的其他方法，例如 `orElse()`，若值不存在则返回默认值，`ifPresent()` 则在值存在时执行一段代码。

第67.1节：Map

使用 Optional 的 `map()` 方法来处理可能为 `null` 的值，而无需显式进行 `null` 检查：

(请注意，`map()` 和 `filter()` 操作会立即执行，不同于它们的 Stream 对应操作，后者只有在执行 终端操作 时才会被执行。)

语法：

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

代码示例：

```
String value = null;  
  
return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");  
// 返回 "NONE"  
  
String value = "something";  
  
return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");  
// 返回 "SOMETHING"
```

因为 `Optional.map()` 当其映射函数返回 `null` 时会返回一个空的 Optional，你可以将多个 `map()` 操作链式调用，作为一种空安全的解引用方式。这也被称为 空安全链式调用。

考虑以下示例：

```
String value = foo.getBar().getBaz().toString();
```

任何 `getBar`、`getBaz` 和 `toString` 都可能抛出 `NullPointerException`。

这里有一种使用 Optional 从 `toString()` 获取值的替代方法：

```
String value = Optional.ofNullable(foo)  
.map(Foo::getBar)  
.map(Bar::getBaz)  
.map(Baz::toString)  
.orElse("");
```

如果任何映射函数返回 `null`，则此方法将返回空字符串。

下面是另一个示例，但略有不同。只有当所有映射函数都未返回 `null` 时，它才会打印值。

```
Optional.ofNullable(foo)  
.map(Foo::getBar)  
.map(Bar::getBaz)
```

Chapter 67: Optional

Optional is a container object which may or may not contain a non-null value. If a value is present, `isPresent()` will return `true` and `get()` will return the value.

Additional methods that depend on the presence of the contained value are provided, such as `orElse()`, which returns a default value if value not present, and `ifPresent()` which executes a block of code if the value is present.

Section 67.1: Map

Use the `map()` method of Optional to work with values that might be `null` without doing explicit `null` checks:

(Note that the `map()` and `filter()` operations are evaluated immediately, unlike their Stream counterparts which are only evaluated upon a *terminal operation*.)

Syntax:

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
```

Code examples:

```
String value = null;  
  
return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");  
// returns "NONE"  
  
String value = "something";  
  
return Optional.ofNullable(value).map(String::toUpperCase).orElse("NONE");  
// returns "SOMETHING"
```

Because `Optional.map()` returns an empty optional when its mapping function returns `null`, you can chain several `map()` operations as a form of null-safe dereferencing. This is also known as **Null-safe chaining**.

Consider the following example:

```
String value = foo.getBar().getBaz().toString();
```

Any of `getBar`, `getBaz`, and `toString` can potentially throw a `NullPointerException`.

Here is an alternative way to get the value from `toString()` using Optional:

```
String value = Optional.ofNullable(foo)  
.map(Foo::getBar)  
.map(Bar::getBaz)  
.map(Baz::toString)  
.orElse("");
```

This will return an empty string if any of the mapping functions returned `null`.

Below is another example, but slightly different. It will print the value only if none of the mapping functions returned `null`.

```
Optional.ofNullable(foo)  
.map(Foo::getBar)  
.map(Bar::getBaz)
```

```
.map(Baz::toString)
    .ifPresent(System.out::println);
```

第67.2节：如果 Optional 为空则返回默认值

不要仅仅使用 `Optional.get()`, 因为这可能会抛出 `NoSuchElementException`。 `Optional.orElse(T)` 和 `Optional.orElseGet(Supplier<? extends T>)` 方法提供了一种在 `Optional` 为空时提供默认值的方式。

```
String value = "something";

return Optional.ofNullable(value).orElse("defaultValue");
// 返回 "something"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// 返回 "something" (永远不会调用 getDefaultValue() 方法)

String value = null;

return Optional.ofNullable(value).orElse("defaultValue");
// 返回 "defaultValue"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// 调用 getDefaultValue() 并返回其结果
```

`orElse` 和 `orElseGet` 之间的关键区别在于, 后者仅在 `Optional` 为空时才会被计算, 而前者传入的参数即使 `Optional` 不为空也会被计算。因此, `orElse` 应仅用于常量, 绝不应用于基于任何计算提供值的情况。

第 67.3 节：如果没有值则抛出异常

使用 `orElseThrow()` 方法从 `Optional` 中获取包含的值, 或者在未设置值时抛出异常。这类似于调用 `get()`, 但允许抛出任意类型的异常。该方法接受一个供应商, 必须返回要抛出的异常。

在第一个示例中, 该方法仅返回包含的值:

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// 返回 "something" 字符串
```

在第二个示例中, 该方法抛出异常, 因为值尚未设置:

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// 抛出 IllegalArgumentException
```

如果需要抛出带有消息的异常, 也可以使用 lambda 语法:

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

第 67.4 节：使用 Supplier 延迟提供默认值

普通的 `normal orElse` 方法接受一个 `Object`, 因此你可能会想为什么这里有一个提供 `Supplier` 的选项

```
.map(Baz::toString)
    .ifPresent(System.out::println);
```

Section 67.2: Return default value if Optional is empty

Don't just use `Optional.get()` since that may throw `NoSuchElementException`. The `Optional.orElse(T)` and `Optional.orElseGet(Supplier<? extends T>)` methods provide a way to supply a default value in case the `Optional` is empty.

```
String value = "something";

return Optional.ofNullable(value).orElse("defaultValue");
// returns "something"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// returns "something" (never calls the getDefaultValue() method)

String value = null;

return Optional.ofNullable(value).orElse("defaultValue");
// returns "defaultValue"

return Optional.ofNullable(value).orElseGet(() -> getDefaultValue());
// calls getDefaultValue() and returns its results
```

The crucial difference between the `orElse` and `orElseGet` is that the latter is only evaluated when the `Optional` is empty while the argument supplied to the former one is evaluated even if the `Optional` is not empty. The `orElse` should therefore only be used for constants and never for supplying value based on any sort of computation.

Section 67.3: Throw an exception, if there is no value

Use the `orElseThrow()` method of `Optional` to get the contained value or throw an exception, if it hasn't been set. This is similar to calling `get()`, except that it allows for arbitrary exception types. The method takes a supplier that must return the exception to be thrown.

In the first example, the method simply returns the contained value:

```
Optional optional = Optional.of("something");

return optional.orElseThrow(IllegalArgumentException::new);
// returns "something" string
```

In the second example, the method throws an exception because a value hasn't been set:

```
Optional optional = Optional.empty();

return optional.orElseThrow(IllegalArgumentException::new);
// throws IllegalArgumentException
```

You can also use the lambda syntax if throwing an exception with message is needed:

```
optional.orElseThrow(() -> new IllegalArgumentException("Illegal"));
```

Section 67.4: Lazily provide a default value using a Supplier

The `normal orElse` method takes an `Object`, so you might wonder why there is an option to provide a `Supplier`

(即 orElseGet 方法)。

考虑：

```
String value = "something";
return Optional.ofNullable(value)
.orElse(getValueThatIsHardToCalculate()); // 返回 "something"
```

即使结果未被使用，因为 Optional 不为空，它仍会调用getValueThatIsHardToCalculate()

为了避免这种开销，你可以提供一个供应商：

```
String value = "something";
return Optional.ofNullable(value)
.orElseGet(() -> getValueThatIsHardToCalculate()); // 返回 "something"
```

这样只有当Optional为空时，才会调用getValueThatIsHardToCalculate()

第67.5节：过滤器

filter()用于表示你只想要满足你的谓词的值。

可以把它想象成if (!somePredicate(x)) { x = null; }。

代码示例：

```
String value = null;
Optional.ofNullable(value) // 无操作
.filter(x -> x.equals("cool string"))// 由于值为 null，此代码永远不会执行
.isPresent(); // false
```

```
String value = "cool string";
Optional.ofNullable(value) // 某些操作
.filter(x -> x.equals("cool string"))// 这段代码会执行且通过
.isPresent(); // 返回 true
```

```
String value = "hot string";
Optional.ofNullable(value) // 某些操作
.filter(x -> x.equals("cool string"))// 这段代码会执行但失败
.isPresent(); // 返回 false
```

第67.6节：使用Optional容器处理原始数字类型

OptionalDouble、OptionalInt和OptionalLong的工作方式类似于Optional，但专门设计用于包装原始类型：

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

由于数值类型确实有值，因此不需要对null进行特殊处理。可以通过以下方式检查空容器：

```
presentInt.isPresent(); // 返回 true。
absentInt.isPresent(); // 返回 false。
```

同样，也存在简写来辅助值管理：

here (the orElseGet method).

Consider:

```
String value = "something";
return Optional.ofNullable(value)
.orElse(getValueThatIsHardToCalculate()); // returns "something"
```

It would still call getValueThatIsHardToCalculate() even though its result is not used as the optional is not empty.

To avoid this penalty you supply a supplier:

```
String value = "something";
return Optional.ofNullable(value)
.orElseGet(() -> getValueThatIsHardToCalculate()); // returns "something"
```

This way getValueThatIsHardToCalculate() will only be called if the Optional is empty.

Section 67.5: Filter

filter() is used to indicate that you would like the value *only if* it matches your predicate.

Think of it like `if (!somePredicate(x)) { x = null; }`.

Code examples:

```
String value = null;
Optional.ofNullable(value) // nothing
.filter(x -> x.equals("cool string"))// this is never run since value is null
.isPresent(); // false
```

```
String value = "cool string";
Optional.ofNullable(value) // something
.filter(x -> x.equals("cool string"))// this is run and passes
.isPresent(); // true
```

```
String value = "hot string";
Optional.ofNullable(value) // something
.filter(x -> x.equals("cool string"))// this is run and fails
.isPresent(); // false
```

Section 67.6: Using Optional containers for primitive number types

OptionalDouble、OptionalInt and OptionalLong work like Optional, but are specifically designed to wrap primitive types:

```
OptionalInt presentInt = OptionalInt.of(value);
OptionalInt absentInt = OptionalInt.empty();
```

Because numeric types do have a value, there is no special handling for null. Empty containers can be checked with:

```
presentInt.isPresent(); // Is true.
absentInt.isPresent(); // Is false.
```

Similarly, shorthands exist to aid value management:

```
// 打印值，因为它在创建时就已提供。
presentInt.ifPresent(System.out::println);

// 如果原始 Optional 为空，则返回另一个值。
int finalValue = absentInt.orElseGet(this::otherValue);

// 将抛出 NoSuchElementException 异常。
int nonexistentValue = absentInt.getAsInt();
```

第67.7节：仅当存在值时运行代码

```
Optional<String> optionalWithValue = Optional.of("foo");
optionalWithValue.ifPresent(System.out::println); // 打印 "foo".

Optional<String> emptyOptional = Optional.empty();
emptyOptional.ifPresent(System.out::println); // 不执行任何操作。
```

第67.8节：FlatMap

[flatMap](#) 类似于 [map](#)。区别由 javadoc 描述如下：

该方法类似于 `map(Function)`，但提供的映射器返回的结果已经是一个 `Optional`，如果调用，`flatMap`不会再用额外的`Optional`进行包装。

换句话说，当你链式调用返回`Optional`的方法时，使用`Optional.flatMap`可以避免创建嵌套的`Optional`。

例如，给定以下类：

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar { }
```

如果你使用`Optional.map`，你将得到一个嵌套的`Optional`；即`Optional<Optional<Bar>>`。

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
    .map(Foo::getBar);
```

然而，如果你使用`Optional.flatMap`，你将得到一个简单的`Optional`；即`Optional<Bar>`。

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
    .flatMap(Foo::getBar);
```

```
// Prints the value since it is provided on creation.
presentInt.ifPresent(System.out::println);

// Gives the other value as the original Optional is empty.
int finalValue = absentInt.orElseGet(this::otherValue);

// Will throw a NoSuchElementException.
int nonexistentValue = absentInt.getAsInt();
```

Section 67.7: Run code only if there is a value present

```
Optional<String> optionalWithValue = Optional.of("foo");
optionalWithValue.ifPresent(System.out::println); // Prints "foo".

Optional<String> emptyOptional = Optional.empty();
emptyOptional.ifPresent(System.out::println); // Does nothing.
```

Section 67.8: FlatMap

[flatMap](#) is similar to [map](#). The difference is described by the javadoc as follows:

This method is similar to `map(Function)`，but the provided mapper is one whose result is already an `Optional`，and if invoked, `flatMap` does not wrap it with an additional `Optional`.

In other words, when you chain a method call that returns an `Optional`, using `Optional.flatMap` avoids creating nested `Optional`s.

For example, given the following classes:

```
public class Foo {
    Optional<Bar> getBar(){
        return Optional.of(new Bar());
    }
}

public class Bar { }
```

If you use `Optional.map`，you will get a nested `Optional`; i.e. `Optional<Optional<Bar>>`。

```
Optional<Optional<Bar>> nestedOptionalBar =
    Optional.of(new Foo())
    .map(Foo::getBar);
```

However, if you use `Optional.flatMap`，you will get a simple `Optional`; i.e. `Optional<Bar>`。

```
Optional<Bar> optionalBar =
    Optional.of(new Foo())
    .flatMap(Foo::getBar);
```

第68章：对象引用

第68.1节：作为方法参数的对象引用

本主题解释了“对象引用”的概念；面向刚开始学习Java编程的人群。你应该已经熟悉一些术语和含义：类定义、主方法、对象实例，以及在对象上调用方法和向方法传递参数。

```
public class Person {  
  
    private String name;  
  
    public void setName(String name) { this.name = name; }  
  
    public String getName() { return name; }  
  
    public static void main(String [] arguments) {  
        Person person = new Person();  
        person.setName("Bob");  
  
        int i = 5;  
        setPersonName(person, i);  
  
        System.out.println(person.getName() + " " + i);  
    }  
  
    private static void setPersonName(Person person, int num) {  
        person.setName("Linda");  
        num = 99;  
    }  
}
```

要完全掌握Java编程，你应该能够脱口而出地向别人解释这个例子。它的概念是理解Java工作原理的基础。

如你所见，我们有一个main方法，它实例化一个对象赋值给变量person，并调用一个方法将该对象中的name字段设置为"Bob"。然后它调用另一个方法，并传递person作为两个参数之一；另一个参数是一个整数变量，值为5。

被调用的方法将传入对象的name值设置为"Linda"，并将传入的整数变量设置为99，然后返回。

那么会打印出什么呢？

Linda 5

那么为什么对person所做的更改在main中生效，而对整数所做的更改却没有生效呢？

当调用发生时，main方法将person的对象引用传递给setPersonName方法；setAnotherName对该对象所做的任何更改都是该对象的一部分，因此当方法返回时，这些更改仍然是该对象的一部分。

换句话说：person指向一个对象（如果你感兴趣的话，存储在堆上）。方法对该对象所做的任何更改都是“在该对象上”进行的，且不受方法是否仍在活动或已返回的影响。当方法返回时，对对象所做的任何更改仍然存储在该对象上。

Chapter 68: Object References

Section 68.1: Object References as method parameters

This topic explains the concept of an *object reference*; it is targeted at people who are new to programming in Java. You should already be familiar with some terms and meanings: class definition, main method, object instance, and the calling of methods "on" an object, and passing parameters to methods.

```
public class Person {  
  
    private String name;  
  
    public void setName(String name) { this.name = name; }  
  
    public String getName() { return name; }  
  
    public static void main(String [] arguments) {  
        Person person = new Person();  
        person.setName("Bob");  
  
        int i = 5;  
        setPersonName(person, i);  
  
        System.out.println(person.getName() + " " + i);  
    }  
  
    private static void setPersonName(Person person, int num) {  
        person.setName("Linda");  
        num = 99;  
    }  
}
```

To be fully competent in Java programming, you should be able to explain this example to someone else off the top of your head. Its concepts are fundamental to understanding how Java works.

As you can see, we have a main that instantiates an object to the variable person, and calls a method to set the name field in that object to "Bob". Then it calls another method, and passes person as one of two parameters; the other parameter is an integer variable, set to 5.

The method called sets the name value on the passed object to "Linda", and sets the integer variable passed to 99, then returns.

So what would get printed?

Linda 5

So why does the change made to person take effect in main, but the change made to the integer does not?

When the call is made, the main method passes an *object reference* for person to the setPersonName method; any change that setAnotherName makes to that object is part of that object, and so those changes are still part of that object when the method returns.

Another way of saying the same thing: person points to an object (stored on the heap, if you're interested). Any change the method makes to that object are made "on that object", and are not affected by whether the method making the change is still active or has returned. When the method returns, any changes made to the object are still stored on that object.

与传入的整数形成对比。由于这是一个原始类型的int（而不是Integer对象实例），它是按值传递的，意味着其值被提供给方法，而不是传入的原始整数的指针。方法可以为了自身目的更改它，但这不会影响方法调用时使用的变量。

在Java中，所有原始类型都是按值传递的。对象是按引用传递的，这意味着对象的指针作为参数传递给任何接受它们的方法。

这意味着一个不太明显的事情：被调用的方法不可能创建一个新的对象并将其作为参数之一返回。方法返回一个由方法调用直接或间接创建的对象的唯一方式，是作为方法的返回值。我们先看看为什么那样做不行，然后再看看如何实现。

让我们在这里给我们的示例添加另一个方法：

```
private static void getAnotherObjectNot(Person person) {  
    person = new Person();  
    person.setName("George");  
}
```

然后，在main方法中，在调用setAnotherName之后，添加对该方法的调用和另一个println调用：

```
getAnotherObjectNot(person);  
System.out.println(person.getName());
```

现在程序将打印出：

琳达5
琳达

传入了George对象的那个对象发生了什么？传入的参数是指向Linda的指针；当getAnotherObjectNot方法创建了一个新对象时，它用指向George对象的引用替换了指向Linda对象的引用。Linda对象仍然存在（在堆上），main方法仍然可以访问它，但getAnotherObjectNot方法之后将无法对其进行任何操作，因为它没有对它的引用。看起来代码的编写者本意是让该方法创建一个新对象并返回它，但如果是这样，它并没有成功。

如果这就是编写者想做的，他需要从方法中返回新创建的对象，类似这样：

```
private static Person getAnotherObject() {  
    Person person = new Person();  
    person.setName("Mary");  
    return person;  
}
```

然后这样调用：

```
Person mary;  
mary = getAnotherObject();  
System.out.println(mary.getName());
```

整个程序的输出现在将是：

Linda 5

Contrast this with the integer that is passed. Since this is a *primitive* int (and not an Integer object instance), it is passed "by value", meaning its value is provided to the method, not a pointer to the original integer passed in. The method can change it for the method's own purposes, but that does not affect the variable used when the method call is made.

In Java, all primitives are passed by value. Objects are passed by reference, which means that a pointer to the object is passed as the parameter to any methods that take them.

One less-obvious thing this means: it is not possible for a called method to create a *new* object and return it as one of the parameters. The only way for a method to return an object that is created, directly or indirectly, by the method call, is as a return value from the method. Let's first see how that would not work, and then how it would work.

Let's add another method to our little example here:

```
private static void getAnotherObjectNot(Person person) {  
    person = new Person();  
    person.setName("George");  
}
```

And, back in the main, below the call to setAnotherName, let's put a call to this method and another println call:

```
getAnotherObjectNot(person);  
System.out.println(person.getName());
```

Now the program would print out:

Linda 5
Linda

What happened to the object that had George? Well, the parameter that was passed in was a pointer to Linda; when the getAnotherObjectNot method created a new object, it replaced the reference to the Linda object with a reference to the George object. The Linda object still exists (on the heap), the main method can still access it, but the getAnotherObjectNot method wouldn't be able to do anything with it after that, because it has no reference to it. It would appear that the writer of the code intended for the method to create a new object and pass it back, but if so, it didn't work.

If that is what the writer wanted to do, he would need to return the newly created object from the method, something like this:

```
private static Person getAnotherObject() {  
    Person person = new Person();  
    person.setName("Mary");  
    return person;  
}
```

Then call it like this:

```
Person mary;  
mary = getAnotherObject();  
System.out.println(mary.getName());
```

And the entire program output would now be:

Linda 5

这是包含两个新增内容的完整程序：

```
public class Person {
    private String name;

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }

    public static void main(String [] arguments) {
        Person person = new Person();
        person.setName("Bob");

        int i = 5;
        setPersonName(person, i);
        System.out.println(person.getName() + " " + i);

        getAnotherObjectNot(person);
        System.out.println(person.getName());

        Person person;
        person = getAnotherObject();
        System.out.println(person.getName());
    }

    private static void setPersonName(Person person, int num) {
        person.setName("Linda");
        num = 99;
    }

    private static void getAnotherObjectNot(Person person) {
        person = new Person();
        person.setMyName("乔治");
    }

    private static Person getAnotherObject() {
        Person person = new Person();
        person.setMyName("玛丽");
        return person;
    }
}
```

Here is the entire program, with both additions:

```
public class Person {
    private String name;

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }

    public static void main(String [] arguments) {
        Person person = new Person();
        person.setName("Bob");

        int i = 5;
        setPersonName(person, i);
        System.out.println(person.getName() + " " + i);

        getAnotherObjectNot(person);
        System.out.println(person.getName());

        Person person;
        person = getAnotherObject();
        System.out.println(person.getName());
    }

    private static void setPersonName(Person person, int num) {
        person.setName("Linda");
        num = 99;
    }

    private static void getAnotherObjectNot(Person person) {
        person = new Person();
        person.setMyName("George");
    }

    private static Person getAnotherObject() {
        Person person = new Person();
        person.setMyName("Mary");
        return person;
    }
}
```

第69章：异常及异常处理

类型为[Throwable](#)及其子类型的对象可以使用**throw**关键字向上抛出，并通过**try...catch**语句捕获。

第69.1节：使用try-catch捕获异常

可以使用try...catch语句来捕获和处理异常。（实际上，try语句还有其他形式，如关于try...catch...finally和try-with-resources的其他示例中所述。）

带有一个catch块的try-catch

最简单的形式如下：

```
try {  
    doSomething();  
} catch (SomeException e) {  
    handle(e);  
}  
// 下一条语句
```

简单的try...catch的行为如下：

- try块中的语句会被执行。
- 如果try块中的语句没有抛出异常，则控制权会传递到try...catch之后的下一条语句。
- 如果在try块中抛出异常。
 - 异常对象会被测试，判断它是否是SomeException或其子类型的实例。
 - 如果是，那么catch块将会catch该异常：
 - 变量e被绑定到异常对象。
 - 执行catch块中的代码。
 - 如果该代码抛出异常，则新抛出的异常将替代原始异常进行传播。
 - 否则，控制权将传递到try...catch之后的下一条语句。
 - 如果不是，原始异常将继续传播。

带多个catch块的try-catch

try...catch也可以有多个catch块。例如：

```
try {  
    doSomething();  
} catch (SomeException e) {  
    handleOneWay(e);  
} catch (SomeOtherException e) {  
    以另一种方式处理(e);  
}  
// 下一条语句
```

如果有多个catch块，它们会依次尝试，从第一个开始，直到找到与异常匹配的为止。执行相应的处理程序（如上所述），然后控制权传递到try...catch语句之后的下一条语句。匹配后的catch块之后的catch块总是被跳过，即使处理程序代码抛出异常也如此。

Chapter 69: Exceptions and exception handling

Objects of type [Throwable](#) and its subtypes can be sent up the stack with the **throw** keyword and caught with **try...catch** statements.

Section 69.1: Catching an exception with try-catch

An exception can be caught and handled using the **try...catch** statement. (In fact **try** statements take other forms, as described in other examples about **try...catch...finally** and **try-with-resources**.)

Try-catch with one catch block

The most simple form looks like this:

```
try {  
    doSomething();  
} catch (SomeException e) {  
    handle(e);  
}  
// next statement
```

The behavior of a simple **try...catch** is as follows:

- The statements in the **try** block are executed.
- If no exception is thrown by the statements in the **try** block, then control passes to the next statement after the **try...catch**.
- If an exception is thrown within the **try** block.
 - The exception object is tested to see if it is an instance of SomeException or a subtype.
 - If it is, then the **catch** block will *catch* the exception:
 - The variable e is bound to the exception object.
 - The code within the **catch** block is executed.
 - If that code throws an exception, then the newly thrown exception is propagated in place of the original one.
 - Otherwise, control passes to the next statement after the **try...catch**.
 - If it is not, the original exception continues to propagate.

Try-catch with multiple catches

A **try...catch** can also have multiple **catch** blocks. For example:

```
try {  
    doSomething();  
} catch (SomeException e) {  
    handleOneWay(e);  
} catch (SomeOtherException e) {  
    handleAnotherWay(e);  
}  
// next statement
```

If there are multiple **catch** blocks, they are tried one at a time starting with the first one, until a match is found for the exception. The corresponding handler is executed (as above), and then control is passed to the next statement after the **try...catch** statement. The **catch** blocks after the one that matches are always skipped, even if the handler code throws an exception.

“自上而下”的匹配策略对catch块中异常不互斥的情况有影响。例如：

```
try {
    throw new RuntimeException("test");
} catch (Exception e) {
    System.out.println("Exception");
} catch (RuntimeException e) {
    System.out.println("RuntimeException");
}
```

这段代码将输出“Exception”而不是“RuntimeException”。由于RuntimeException是Exception的子类型，第一个（更通用的）catch会被匹配。第二个（更具体的）catch永远不会被执行。

从中得到的教训是，最具体的catch块（就异常类型而言）应该放在最前面，最通用的应该放在最后。（某些Java编译器会警告你如果某个catch永远不会被执行，但这不是编译错误。）

多异常捕获块

版本 ≥ Java SE 7

从Java SE 7开始，单个catch块可以处理一组无关的异常。异常类型用竖线符号（|）分隔。例如：

```
try {
doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

多异常捕获的行为是对单异常情况的简单扩展。如果抛出的异常匹配列表中（至少）一个异常，catch就会匹配。

规范中还有一些额外的细微之处。变量 e 的类型是异常类型的合成union（联合类型），来自列表中的异常类型。当使用 e 的值时，其静态类型是该类型联合的最小公共超类型。然而，如果在 catch 块中重新抛出 e，则抛出的异常类型是联合中的类型。例如：

```
public void method() throws IOException, SQLException
    try {
doSomething();
    } catch (IOException | SQLException e) {
        report(e);
        throw e;
    }
}
```

上述代码中，IOException 和 SQLException 是受检异常，它们的最小公共超类型是 Exception。这意味着 report 方法必须匹配 report(Exception)。然而，编译器知道 throw 只能抛出 IOException 或 SQLException。因此，method 可以声明为 throws IOException, SQLException 而不是 throws Exception。（这是件好事：参见陷阱 - 抛出 Throwable、Exception、Error 或 RuntimeException。）

第69.2节：带资源的try语句

版本 ≥ Java SE 7

The "top down" matching strategy has consequences for cases where the exceptions in the **catch** blocks are not disjoint. For example:

```
try {
    throw new RuntimeException("test");
} catch (Exception e) {
    System.out.println("Exception");
} catch (RuntimeException e) {
    System.out.println("RuntimeException");
}
```

This code snippet will output "Exception" rather than "RuntimeException". Since **RuntimeException** is a subtype of **Exception**, the first (more general) **catch** will be matched. The second (more specific) **catch** will never be executed.

The lesson to learn from this is that the most specific **catch** blocks (in terms of the exception types) should appear first, and the most general ones should be last. (Some Java compilers will warn you if a **catch** can never be executed, but this is not a compilation error.)

Multi-exception catch blocks

Version ≥ Java SE 7

Starting with Java SE 7, a single **catch** block can handle a list of unrelated exceptions. The exception type are listed, separated with a vertical bar (|) symbol. For example:

```
try {
    doSomething();
} catch (SomeException | SomeOtherException e) {
    handleSomeException(e);
}
```

The behavior of a multi-exception catch is a simple extension for the single-exception case. The **catch** matches if the thrown exception matches (at least) one of the listed exceptions.

There is some additional subtlety in the specification. The type of e is a synthetic union of the exception types in the list. When the value of e is used, its static type is the least common supertype of the type union. However, if e is rethrown within the **catch** block, the exception types that are thrown are the types in the union. For example:

```
public void method() throws IOException, SQLException
    try {
        doSomething();
    } catch (IOException | SQLException e) {
        report(e);
        throw e;
    }
}
```

In the above, **IOException** and **SQLException** are checked exceptions whose least common supertype is **Exception**. This means that the report method must match **report(Exception)**. However, the compiler knows that the **throw** can throw only an **IOException** or an **SQLException**. Thus, **method** can be declared as **throws IOException, SQLException** rather than **throws Exception**. (Which is a good thing: see Pitfall - Throwing Throwable, Exception, Error or RuntimeException.)

Section 69.2: The try-with-resources statement

Version ≥ Java SE 7

正如try-catch-finally语句示例所示，使用 finally 子句进行资源清理需要大量的“样板”代码来正确处理边界情况。Java 7 提供了一种更简单的方式来解决这个问题，即 try-with-resources 语句。

什么是资源？

Java 7 引入了 `java.lang.AutoCloseable` 接口，以允许类使用 try-with-resources 语句进行管理。实现了 `AutoCloseable` 接口的类的实例被称为 resources。这些资源通常需要及时释放，而不是依赖垃圾回收器来处理它们。

`AutoCloseable` 接口定义了一个单一的方法：

```
public void close() throws Exception
```

`close()` 方法应以适当的方式释放资源。规范指出，对已经释放的资源调用该方法应该是安全的。此外，实现 `AutoCloseable` 的类 强烈建议 声明 `close()` 方法抛出比 `Exception` 更具体的异常，或者根本不抛出异常。

许多标准的 Java 类和接口实现了 `AutoCloseable`。这些包括：

- `InputStream`、`OutputStream` 及其子类
- `Reader`、`Writer` 及其子类
- `Socket` 和 `ServerSocket` 及其子类
- `Channel` 及其子类，以及
- JDBC 接口 `Connection`、`Statement` 和 `ResultSet` 及其子类。

应用程序和第三方类也可能实现该接口。

基本的 try-with-resource 语句

try-with-resources 的语法基于经典的 try-catch、try-finally 和 try-catch-finally 形式。以下是一个“基本”形式的示例；即没有 catch 或 finally 的形式。

```
try (PrintStream stream = new PrintStream("hello.txt")) {  
    stream.println("Hello world!");  
}
```

要管理的资源作为变量声明在 `try` 子句后面的 (...) 部分。在上面的示例中，我们声明了一个资源变量 `stream` 并将其初始化为新创建的 `PrintStream`。

资源变量初始化后，执行 `try` 块。完成后，`stream.close()` 会被自动调用，以确保资源不会泄漏。注意，无论块如何完成，`close()` 调用都会发生。

增强的 try-with-resource 语句

try-with-resources 语句可以像 Java 7 之前的 try-catch-finally 语法一样，增强为带有 catch 和 finally 块。以下代码片段为之前的代码添加了一个 catch 块来处理 `PrintStream` 构造函数可能抛出的 `FileNotFoundException`：

```
try (PrintStream stream = new PrintStream("hello.txt")) {  
    stream.println("Hello world!");  
} catch (FileNotFoundException ex) {  
    System.err.println("Cannot open the file");
```

As the try-catch-final statement example illustrates, resource cleanup using a `finally` clause requires a significant amount of “boiler-plate” code to implement the edge-cases correctly. Java 7 provides a much simpler way to deal with this problem in the form of the `try-with-resources` statement.

What is a resource?

Java 7 introduced the `java.lang.AutoCloseable` interface to allow classes to be managed using the `try-with-resources` statement. Instances of classes that implement `AutoCloseable` are referred to as `resources`. These typically need to be disposed of in a timely fashion rather than relying on the garbage collector to dispose of them.

The `AutoCloseable` interface defines a single method:

```
public void close() throws Exception
```

A `close()` method should dispose of the resource in an appropriate fashion. The specification states that it should be safe to call the method on a resource that has already been disposed of. In addition, classes that implement `AutoCloseable` are strongly encouraged to declare the `close()` method to throw a more specific exception than `Exception`, or no exception at all.

A wide range of standard Java classes and interfaces implement `AutoCloseable`. These include:

- `InputStream`、`OutputStream` 和其子类
- `Reader`、`Writer` 和其子类
- `Socket` 和 `ServerSocket` 和其子类
- `Channel` 和其子类，以及
- JDBC 接口 `Connection`、`Statement` 和 `ResultSet` 和其子类。

应用程序和第三方类可能也这样做。

The basic try-with-resource statement

The syntax of a `try-with-resources` is based on classical `try-catch`, `try-finally` and `try-catch-finally` forms. Here is an example of a “basic” form; i.e. the form without a `catch` or `finally`.

```
try (PrintStream stream = new PrintStream("hello.txt")) {  
    stream.println("Hello world!");  
}
```

要管理的资源作为变量声明在 `try` 子句后面的 (...) 部分。在上面的示例中，我们声明了一个资源变量 `stream` 并将其初始化为新创建的 `PrintStream`。

一旦资源变量初始化后，执行 `try` 块。完成后，`stream.close()` 会被自动调用，以确保资源不会泄漏。注意，无论块如何完成，`close()` 调用都会发生。

The enhanced try-with-resource statements

The `try-with-resources` 语句可以增强为带有 `catch` 和 `finally` 块，就像之前的 Java 7 `try-catch-finally` 语法一样。以下代码片段为之前的代码添加了一个 `catch` 块来处理 `FileNotFoundException`，这是 `PrintStream` 构造函数可能抛出的：

```
try (PrintStream stream = new PrintStream("hello.txt")) {  
    stream.println("Hello world!");  
} catch (FileNotFoundException ex) {  
    System.err.println("Cannot open the file");
```

```

} 最终 {
    System.err.println("All done");
}

```

如果资源初始化或try块抛出异常，则catch块将被执行。与传统的try-catch-finally语句一样，**finally**块总是会被执行。

不过，有几点需要注意：

- 资源变量在catch和finally块中是超出作用域的。
- 资源清理会在语句尝试匹配catch块之前发生。
- 如果自动资源清理抛出异常，那么该异常可能会被某个catch块捕获。

管理多个资源

上面的代码片段展示了单个资源的管理。实际上，*try-with-resources*可以在一个语句中管理多个资源。例如：

```

try (InputStream is = new FileInputStream(file1);
    OutputStream os = new FileOutputStream(file2)) {
    // 将 'is' 复制到 'os'
}

```

这表现得如你所料。*is* 和 *os* 都会在 try 块结束时自动关闭。有几点需要注意：

- 初始化按代码顺序进行，后面的资源变量初始化器可以使用前面变量的值。
- 所有成功初始化的资源变量都会被清理。
- 资源变量按照声明的逆序进行清理。

因此，在上述示例中，*is* 在 *os* 之前初始化，并在其之后清理；如果在初始化 *os* 时发生异常，*is* 也会被清理。

try-with-resource 与传统 try-catch-finally 的等价性

Java语言规范以传统的 try-catch-finally 语句来定义 *try-with-resource* 形式的行为。（详细内容请参阅JLS）

例如，这个基本的 *try-with-resource*：

```

try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}

```

被定义为等价于这个 try-catch-finally：

```

// 注意构造函数不属于 try-catch 语句的一部分
PrintStream stream = new PrintStream("hello.txt");

// 该变量用于跟踪 try 语句中抛出的主要异常
// 如果在 try 块中抛出异常，
// 由 AutoCloseable.close() 抛出的任何异常将被抑制。
Throwable primaryException = null;

// 实际的 try 块
try {

```

```

} finally {
    System.err.println("All done");
}

```

If either the resource initialization or the try block throws the exception, then the **catch** block will be executed. The **finally** block will always be executed, as with a conventional *try-catch-finally* statement.

There are a couple of things to note though:

- The resource variable is *out of scope* in the **catch** and **finally** blocks.
- The resource cleanup will happen before the statement tries to match the **catch** block.
- If the automatic resource cleanup threw an exception, then that *could* be caught in one of the **catch** blocks.

Managing multiple resources

The code snippets above show a single resource being managed. In fact, *try-with-resources* can manage multiple resources in one statement. For example:

```

try (InputStream is = new FileInputStream(file1);
    OutputStream os = new FileOutputStream(file2)) {
    // Copy 'is' to 'os'
}

```

This behaves as you would expect. Both *is* and *os* are closed automatically at the end of the **try** block. There are a couple of points to note:

- The initializations occur in the code order, and later resource variable initializers can use of the values of the earlier ones.
- All resource variables that were successfully initialized will be cleaned up.
- Resource variables are cleaned up in *reverse order* of their declarations.

Thus, in the above example, *is* is initialized before *os* and cleaned up after it, and *is* will be cleaned up if there is an exception while initializing *os*.

Equivalence of try-with-resource and classical try-catch-finally

The Java Language Specification specifies the behavior of *try-with-resource* forms in terms of the classical *try-catch-finally* statement. (Please refer to the JLS for the full details.)

For example, this basic *try-with-resource*:

```

try (PrintStream stream = new PrintStream("hello.txt")) {
    stream.println("Hello world!");
}

```

is defined to be equivalent to this *try-catch-finally*:

```

// Note that the constructor is not part of the try-catch statement
PrintStream stream = new PrintStream("hello.txt");

// This variable is used to keep track of the primary exception thrown
// in the try statement. If an exception is thrown in the try block,
// any exception thrown by AutoCloseable.close() will be suppressed.
Throwable primaryException = null;

// The actual try block
try {

```

```

stream.println("Hello world!");
} catch (Throwable t) {
    // 如果抛出异常, 记录该异常以供 finally 块使用
primaryException = t;
throw t;
} 最终 {
if (primaryException == null) {
    // 如果到目前为止没有抛出异常, close() 中抛出的异常将不会被捕获,
    // 因此会传递给外层代码。
stream.close();
} else {
    // 如果已经抛出异常, close() 中抛出的任何异常将被抑制, // 因为它们很可能与之前的
    // 异常有关。被抑制的异常可以通过// primaryException.getSuppressed() 获取。
}

try {
stream.close();
} catch (Throwable suppressedException) {
    primaryException.addSuppressed(suppressedException);
}
}
}

```

(JLS 指定实际的 t 和 primaryException 变量对普通 Java 代码是不可见的。)

增强的 *try-with-resources* 形式被规定为与基本形式等价。例如：

```

try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}

```

等价于：

```

try {
    try (PrintStream stream = new PrintStream(fileName)) {
        stream.println("Hello world!");
    }
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}

```

第69.3节：自定义异常

在大多数情况下，从代码设计的角度来看，使用现有的通用Exception类来抛出异常更为简单。尤其是在你只需要异常携带一个简单的错误信息时，通常更倾向于使用RuntimeException，因为它不是受检异常。还有其他异常类用于常见的错误类别：

- [UnsupportedOperationException](#) - 某个操作不被支持
- [Xception](#) - 向方法传递了无效的参数值
- [IllegalStateException](#) - 你的API内部达到一个不应该发生的状态，或者由于以无效方式使用API而导致的状态

```

stream.println("Hello world!");
} catch (Throwable t) {
    // If an exception is thrown, remember it for the finally block
primaryException = t;
throw t;
} finally {
if (primaryException == null) {
    // If no exception was thrown so far, exceptions thrown in close() will
    // not be caught and therefore be passed on to the enclosing code.
stream.close();
} else {
    // If an exception has already been thrown, any exception thrown in
    // close() will be suppressed as it is likely to be related to the
    // previous exception. The suppressed exception can be retrieved
    // using primaryException.getSuppressed().
try {
    stream.close();
} catch (Throwable suppressedException) {
    primaryException.addSuppressed(suppressedException);
}
}
}

```

(The JLS specifies that the actual t and primaryException variables will be invisible to normal Java code.)

The enhanced form of *try-with-resources* is specified as an equivalence with the basic form. For example:

```

try (PrintStream stream = new PrintStream(fileName)) {
    stream.println("Hello world!");
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}

```

is equivalent to:

```

try {
    try (PrintStream stream = new PrintStream(fileName)) {
        stream.println("Hello world!");
    }
} catch (NullPointerException ex) {
    System.err.println("Null filename");
} finally {
    System.err.println("All done");
}

```

Section 69.3: Custom Exceptions

Under most circumstances, it is simpler from a code-design standpoint to use existing generic [Exception](#) classes when throwing exceptions. This is especially true if you only need the exception to carry a simple error message. In that case, [RuntimeException](#) is usually preferred, since it is not a checked Exception. Other exception classes exist for common classes of errors:

- [UnsupportedOperationException](#) - a certain operation is not supported
- [IllegalArgumentException](#) - an invalid parameter value was passed to a method
- [IllegalStateException](#) - your API has internally reached a condition that should never happen, or which occurs as a result of using your API in an invalid way

以下情况你确实需要使用自定义异常类：

- 你正在编写供他人使用的API或库，并且你希望API的使用者能够专门捕获并处理来自你API的异常，且能够区分这些异常与其他更通用的异常。
- 你在程序的某个部分为特定类型的错误抛出异常，想在程序的另一部分捕获并处理这些异常，并且希望能够区分这些错误与其他更通用的错误。

你可以通过继承RuntimeException来创建未检查异常的自定义异常，或者通过继承任何Exception（但不是RuntimeException的子类）来创建检查异常，因为：

Exception的子类且不是RuntimeException的子类的是检查异常

```
public class StringTooLongException extends RuntimeException {  
    // 异常可以像其他类一样拥有方法和字段// 这些可以用来向捕获该异常的代码  
    // 传递信息  
  
    public final String value;  
    public final int maximumLength;  
  
    public StringTooLongException(String value, int maximumLength){  
        super(String.format("字符串超过最大长度 %s: %s", maximumLength, value));  
        this.value = value;  
        this.maximumLength = maximumLength;  
    }  
}
```

这些可以作为预定义异常使用：

```
void validateString(String value){  
    if (value.length() > 30){  
        throw new StringTooLongException(value, 30);  
    }  
}
```

并且这些字段可以在捕获和处理异常的地方使用：

```
void anotherMethod(String value){  
    try {  
        validateString(value);  
    } catch(StringTooLongException e){  
        System.out.println("字符串 '" + e.value +  
                           " 长度超过最大值 " + e.maximumLength );  
    }  
}
```

请记住，根据Oracle的Java文档：

如果客户端可以合理地预期从异常中恢复，则将其设为受检异常。如果客户端无法采取任何措施从异常中恢复，则将其设为非受检异常。

更多内容：

Cases where you **do** want to use a custom exception class include the following:

- You are writing an API or library for use by others, and you want to allow users of your API to be able to specifically catch and handle exceptions from your API, and *be able to differentiate those exceptions from other, more generic exceptions.*
- You are throwing exceptions for a **specific kind of error** in one part of your program, which you want to catch and handle in another part of your program, and you want to be able to differentiate these errors from other, more generic errors.

You can create your own custom exceptions by extending [RuntimeException](#) for an unchecked exception, or checked exception by extending any [Exception](#) which is not also subclass of [RuntimeException](#), because:

Subclasses of Exception that are not also subclasses of RuntimeException are checked exceptions

```
public class StringTooLongException extends RuntimeException {  
    // Exceptions can have methods and fields like other classes  
    // those can be useful to communicate information to pieces of code catching  
    // such an exception  
    public final String value;  
    public final int maximumLength;  
  
    public StringTooLongException(String value, int maximumLength){  
        super(String.format("String exceeds maximum Length of %s: %s", maximumLength, value));  
        this.value = value;  
        this.maximumLength = maximumLength;  
    }  
}
```

Those can be used just as predefined exceptions:

```
void validateString(String value){  
    if (value.length() > 30){  
        throw new StringTooLongException(value, 30);  
    }  
}
```

And the fields can be used where the exception is caught and handled:

```
void anotherMethod(String value){  
    try {  
        validateString(value);  
    } catch(StringTooLongException e){  
        System.out.println("The string '" + e.value +  
                           " was longer than the max of " + e.maximumLength );  
    }  
}
```

Keep in mind that, according to [Oracle's Java Documentation](#):

[...] If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

More:

- 为什么 `RuntimeException` 不需要显式的异常处理？

第69.4节：处理中断异常（`InterruptedException`）

`InterruptedException` 是一个令人困惑的问题——它出现在看似无害的方法中，比如 `Thread.sleep()`，但错误处理它会导致难以管理且在并发环境中表现不佳的代码。

最基本的情况是，如果捕获了 `InterruptedException`，意味着某处有人调用了当前代码运行线程的 `Thread.interrupt()`。你可能会倾向于说“这是我的代码！我永远不会中断它！”因此会做如下处理：

```
// 错误示范。不要这样做。
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    // 忽略
}
```

但这恰恰是错误处理“不可发生”事件的方式。如果你知道你的应用永远不会遇到 `InterruptedException`，应将此类事件视为对程序假设的严重违反，并尽快退出。

正确处理“不可发生”中断的方式如下：

```
// 当没有任何东西会中断你的代码时
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    throw new AssertionError(e);
}
```

这做了两件事；首先它恢复了线程的中断状态（就好像`InterruptedException`根本没有被抛出过一样），然后它抛出一个`AssertionError`，表示你的应用程序的基本不变量被破坏了。如果你确定永远不会中断该线程，那么这段代码是安全的，因为`catch`块永远不会被执行。

使用Guava的`Uninterruptibles`类可以简化这种模式；调用`Uninterruptibles.sleepUninterruptibly()`会忽略线程的中断状态，直到睡眠时间结束（此时中断状态会被恢复，以便后续调用检查并抛出它们自己的`InterruptedException`）。如果你确定永远不会中断这段代码，这样做可以安全地避免将睡眠调用包裹在`try-catch`块中。

然而，更常见的情况是，你无法保证线程永远不会被中断。特别是当你编写的代码将由`Executor`或其他线程管理器执行时，确保代码能及时响应中断非常关键，否则你的应用程序可能会停滞甚至死锁。

在这种情况下，最好的做法通常是允许`InterruptedException`向上抛出，逐层在每个方法中添加`throws InterruptedException`。这看起来可能有些笨拙，但实际上这是一个理想的特性——你的方法签名现在向调用者表明它会及时响应中断。

```
// 如果不确定如何处理中断，让调用者决定
public void myLongRunningMethod() throws InterruptedException {
    ...
}
```

- Why does `RuntimeException` not require an explicit exception handling?

Section 69.4: Handling `InterruptedException`

`InterruptedException` 是一个令人困惑的生物 - 它出现在看似无害的方法中，比如 `Thread.sleep()`，但错误处理它会导致难以管理且在并发环境中表现不佳的代码。

At its most basic, if an `InterruptedException` is caught it means someone, somewhere, called `Thread.interrupt()` on the thread your code is currently running in. You might be inclined to say "It's my code! I'll never interrupt it!" and therefore do something like this:

```
// Bad. Don't do this.
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    // disregard
}
```

But this is exactly the wrong way to handle an "impossible" event occurring. If you know your application will never encounter an `InterruptedException` you should treat such an event as a serious violation of your program's assumptions and exit as quickly as possible.

The proper way to handle an "impossible" interrupt is like so:

```
// When nothing will interrupt your code
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    throw new AssertionError(e);
}
```

This does two things; it first restores the interrupt status of the thread (as if the `InterruptedException` had not been thrown in the first place), and then it throws an `AssertionError` indicating the basic invariants of your application have been violated. If you know for certain that you'll never interrupt the thread this code runs in this is safe since the `catch` block should never be reached.

Using Guava's `Uninterruptibles` class helps simplify this pattern; calling `Uninterruptibles.sleepUninterruptibly()` disregards the interrupted state of a thread until the sleep duration has expired (at which point it's restored for later calls to inspect and throw their own `InterruptedException`). If you know you'll never interrupt such code this safely avoids needing to wrap your sleep calls in a `try-catch` block.

More often, however, you cannot guarantee that your thread will never be interrupted. In particular if you're writing code that will be executed by an `Executor` or some other thread-management it's critical that your code responds promptly to interrupts, otherwise your application will stall or even deadlock.

In such cases the best thing to do is generally to allow the `InterruptedException` to propagate up the call stack, adding a `throws InterruptedException` to each method in turn. This may seem kludgy but it's actually a desirable property - your method's signatures now indicates to callers that it will respond promptly to interrupts.

```
// Let the caller determine how to handle the interrupt if you're unsure
public void myLongRunningMethod() throws InterruptedException {
    ...
}
```

在有限的情况下（例如重写一个不throw任何受检异常的方法时），你可以重置中断状态而不抛出异常，期望接下来执行的代码来处理中断。
这会延迟处理中断，但不会完全抑制它。

```
// 抑制异常但重置中断状态，使后续代码能够检测中断并正确处理。
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    return ...; // 你的预期在此时仍然被破坏——尽量不要做更多工作。
}
```

第69.5节：try catch块中的return语句

虽然这是不好的做法，但可以在异常处理块中添加多个return语句：

```
public static int returnTest(int number){
    try{
        if(number%2 == 0) throw new Exception("Exception thrown");
        else return x;
    }
    catch(Exception e){
        return 3;
    }
    finally{
        return 7;
    }
}
```

该方法将始终返回7，因为与try/catch块关联的finally块会在任何返回之前执行。现在，由于finally中有return 7；，该值会覆盖try/catch的返回值。

如果catch块返回一个基本类型值，且该基本类型值随后在finally块中被更改，则返回的是catch块中的值，finally块中的更改将被忽略。

下面的示例将打印“0”，而不是“1”。

```
public class FinallyExample {

    public static void main(String[] args) {
        int n = returnTest(4);

        System.out.println(n);
    }

    public static int returnTest(int number) {

        int returnNumber = 0;

        try {
            if (number % 2 == 0)
                throw new Exception("Exception thrown");
            else
                return returnNumber;
        } catch (Exception e) {
            return returnNumber;
        } finally {
            returnNumber = 1;
        }
    }
}
```

In limited cases (e.g. while overriding a method that doesn't **throw** any checked exceptions) you can reset the interrupted status without raising an exception, expecting whatever code is executed next to handle the interrupt. This delays handling the interruption but doesn't suppress it entirely.

```
// Suppresses the exception but resets the interrupted state letting later code
// detect the interrupt and handle it properly.
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    return ...; // your expectations are still broken at this point - try not to do more work.
}
```

Section 69.5: Return statements in try catch block

Although it's bad practice, it's possible to add multiple return statements in a exception handling block:

```
public static int returnTest(int number){
    try{
        if(number%2 == 0) throw new Exception("Exception thrown");
        else return x;
    }
    catch(Exception e){
        return 3;
    }
    finally{
        return 7;
    }
}
```

This method will always return 7 since the finally block associated with the try/catch block is executed before anything is returned. Now, as finally has **return 7**；，this value supersedes the try/catch return values.

If the catch block returns a primitive value and that primitive value is subsequently changed in the finally block, the value returned in the catch block will be returned and the changes from the finally block will be ignored.

The example below will print "0", not "1".

```
public class FinallyExample {

    public static void main(String[] args) {
        int n = returnTest(4);

        System.out.println(n);
    }

    public static int returnTest(int number) {

        int returnNumber = 0;

        try {
            if (number % 2 == 0)
                throw new Exception("Exception thrown");
            else
                return returnNumber;
        } catch (Exception e) {
            return returnNumber;
        } finally {
            returnNumber = 1;
        }
    }
}
```

```
    }
}
```

第69.6节：介绍

异常是在程序执行时发生的错误。考虑下面的Java程序，它对两个整数进行除法运算。

```
类 Division {
    public static void main(String[] args) {
        整数 a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("输入两个整数");

        a = input.nextInt();
        b = input.nextInt();

        result = a / b;

        System.out.println("结果 = " + result);
    }
}
```

现在我们编译并执行上述代码，观察尝试除以零时的输出：

```
输入两个整数
7 0
线程“main”中的异常 java.lang.ArithmeicException: / by zero
at Division.main(Disivion.java:14)
```

除以零是一个无效操作，会产生一个无法表示为整数的值。Java通过抛出异常来处理这种情况。在本例中，异常是ArithmeicException类的一个实例。

注意：创建和读取堆栈跟踪的示例解释了两个数字之后的输出含义。

异常（exception）的作用是它允许的流程控制。不使用异常时，解决此问题的典型方法可能是先检查 `b == 0`：

```
类 Division {
    public static void main(String[] args) {
        整数 a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("请输入两个整数");

        a = input.nextInt();
        b = input.nextInt();

        if (b == 0) {
            System.out.println("不能除以零。");
            return;
        }
    }
}
```

```
    }
}
```

Section 69.6: Introduction

Exceptions are errors which occur when a program is executing. Consider the Java program below which divides two integers.

```
class Division {
    public static void main(String[] args) {
        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        result = a / b;

        System.out.println("Result = " + result);
    }
}
```

Now we compile and execute the above code, and see the output for an attempted division by zero:

```
Input two integers
7 0
Exception in thread "main" java.lang.ArithmeicException: / by zero
at Division.main(Disivion.java:14)
```

Division by zero is an invalid operation that would produce a value that cannot be represented as an integer. Java deals with this by *throwing an exception*. In this case, the exception is an instance of the *ArithmeicException* class.

Note: The example on creating and reading stack traces explains what the output after the two numbers means.

The utility of an *exception* is the flow control that it allows. Without using exceptions, a typical solution to this problem may be to first check if `b == 0`:

```
class Division {
    public static void main(String[] args) {
        int a, b, result;

        Scanner input = new Scanner(System.in);
        System.out.println("Input two integers");

        a = input.nextInt();
        b = input.nextInt();

        if (b == 0) {
            System.out.println("You cannot divide by zero.");
            return;
        }
    }
}
```

```

result = a / b;
    System.out.println("结果 = " + result);
}

```

当用户尝试除以零时，这会向控制台打印消息 不能除以零。 并以优雅的方式退出程序。通过异常处理处理此问题的等效方法是用 try-catch 块替换 if 流程控制：

```

...
a = input.nextInt();
b = input.nextInt();

try {
    result = a / b;
}
catch (ArithmeticException e) {
    System.out.println("发生了算术异常。可能你尝试除以零。");
    return;
}
...

```

try catch 块的执行过程如下：

1. 开始执行try块中的代码。
2. 如果在 try 块中发生了异常，立即中止并检查该异常是否被捕获 catch块（在本例中，当异常是ArithmeticException的实例时）。
3. 如果异常被捕获，则将其赋值给变量e，并执行catch块。
4. 如果try或catch块完成（即代码执行期间没有未捕获的异常发生），则继续执行try-catch块下面的代码。

通常认为，将异常处理作为应用程序正常流程控制的一部分是一种良好做法，当行为本来会是未定义或意外时。例如，代替返回null时，方法失败时，通常更好的做法是抛出异常，以便使用该方法的应用程序可以通过上面示例中的异常处理来定义其自身的流程控制。从某种意义上说，这绕过了必须返回特定类型的问题，因为可以抛出多种异常中的任何一种来指示发生的具体问题。

有关如何使用和不使用异常的更多建议，请参阅《Java陷阱 - 异常使用》

第69.7节：Java异常层次结构 - 非检查异常和检查异常

所有Java异常都是Exception类层次结构中类的实例。其结构可表示如下：

- [java.lang.Throwable](#) - 这是所有异常类的基类。其方法和构造函数实现了所有异常共有的一系列功能。
 - [java.lang.Exception](#) - 这是所有普通异常的超类。
 - 各种标准和自定义异常类。
 - [java.lang.RuntimeException](#) - 这是所有普通的非检查异常的超类。
 - 各种标准和自定义运行时异常类。

```

result = a / b;
    System.out.println("Result = " + result);
}

```

This prints the message You cannot divide by zero. to the console and quits the program in a graceful way when the user tries to divide by zero. An equivalent way of dealing with this problem via exception handling would be to replace the if flow control with a try-catch block:

```

...
a = input.nextInt();
b = input.nextInt();

try {
    result = a / b;
}
catch (ArithmeticException e) {
    System.out.println("An ArithmeticException occurred. Perhaps you tried to divide by zero.");
    return;
}
...

```

A try catch block is executed as follows:

1. Begin executing the code in the **try** block.
2. If an *exception* occurs in the try block, immediately abort and check to see if this exception is *caught* by the **catch** block (in this case, when the Exception is an instance of [ArithmeticException](#)).
3. If the exception is *caught*, it is assigned to the variable **e** and the **catch** block is executed.
4. If either the **try** or **catch** block is completed (i.e. no uncaught exceptions occur during code execution) then continue to execute code below the **try-catch** block.

It is generally considered good practice to use *exception handling* as part of the normal flow control of an application where behavior would otherwise be undefined or unexpected. For instance, instead of returning **null** when a method fails, it is usually better practice to *throw an exception* so that the application making use of the method can define its own flow control for the situation via *exception handling* of the kind illustrated above. In some sense, this gets around the problem of having to return a particular *type*, as any one of multiple kinds of *exceptions* may be *thrown* to indicate the specific problem that occurred.

For more advice on how and how not to use exceptions, refer to Java Pitfalls - Exception usage

Section 69.7: The Java Exception Hierarchy - Unchecked and Checked Exceptions

All Java exceptions are instances of classes in the Exception class hierarchy. This can be represented as follows:

- [java.lang.Throwable](#) - This is the base class for all exception classes. Its methods and constructors implement a range of functionality common to all exceptions.
 - [java.lang.Exception](#) - This is the superclass of all normal exceptions.
 - various standard and custom exception classes.
 - [java.lang.RuntimeException](#) - This is the superclass of all normal exceptions that are *unchecked exceptions*.
 - various standard and custom runtime exception classes.

- [java.lang.Error](#) - 这是所有“致命错误”异常的超类。

备注：

1. 下面描述了已检查异常和未检查异常之间的区别。
2. `Throwable`、`Exception` 和 `RuntimeException` 类应视为抽象类；参见陷阱 - 抛出 `Throwable`、`Exception`、`Error` 或 `RuntimeException`。
3. 错误异常由JVM在某些情况下抛出，这些情况下继续执行将是不安全或不明智的。申请尝试恢复。
4. 声明`Throwable`的自定义子类型是不明智的。Java 工具和库可能会假设`Error` and `Exception` 是 `Throwable` 唯一的直接子类型，如果该假设不正确，则会表现异常。

已检查异常与未检查异常

对某些编程语言中异常支持的批评之一是，很难知道某个方法或过程可能抛出哪些异常。鉴于未处理的异常可能导致程序崩溃，这会使异常成为脆弱性的来源。

Java语言通过受检异常机制来解决这个问题。首先，Java将异常分为两类：

- 受检异常通常表示应用程序应该能够处理的预期事件。例如，`IOException`及其子类型表示可能在输入输出操作中发生的错误情况。示例包括文件打开失败，因为文件或目录不存在，网络读写失败，因为网络连接已断开，等等。
- 未检查异常通常表示应用程序无法处理的意外事件。这些异常通常是应用程序中的错误导致的。

(以下内容中，“抛出”指的是任何显式（通过`throw`语句）或隐式（如失败的解引用、类型转换等）抛出的异常。同样，“传播”指的是在嵌套调用中抛出但未在该调用内捕获的异常。下面的示例代码将对此进行说明。)

已检查异常机制的第二部分是对可能发生已检查异常的方法有一定限制：

- 当方法中抛出或传播已检查异常时，必须要么由该方法捕获，要么在方法的`throws`子句中声明。（本示例将说明`throws`子句的重要性。）当初始化块中抛出或传播已检查异常时，必须由该块捕获。
- 已检查异常不能通过字段初始化表达式中的方法调用传播。（因为无法捕获此类异常。）

总而言之，已检查异常必须被处理或声明。

这些限制不适用于未检查异常。这包括所有隐式抛出异常的情况，因为所有此类情况都抛出未检查异常。

已检查异常示例

这些代码片段旨在说明已检查异常的限制。每种情况下，我们展示一个带有编译错误的代码版本，以及一个修正错误的版本。

```
// 这是声明一个自定义已检查异常。
public class MyException extends Exception {
    // 构造函数省略。
```

- [java.lang.Error](#) - This is the superclass of all "fatal error" exceptions.

Notes:

1. The distinction between *checked* and *unchecked* exceptions is described below.
2. The `Throwable`, `Exception` and `RuntimeException` class should be treated as **abstract**; see Pitfall - Throwing `Throwable`, `Exception`, `Error` or `RuntimeException`.
3. The `Error` exceptions are thrown by the JVM in situations where it would be unsafe or unwise for an application to attempt to recover.
4. It would be unwise to declare custom subtypes of `Throwable`. Java tools and libraries may assume that `Error` and `Exception` are the only direct subtypes of `Throwable`, and misbehave if that assumption is incorrect.

Checked versus Unchecked Exceptions

One of the criticisms of exception support in some programming languages is that is difficult to know which exceptions a given method or procedure might throw. Given that an unhandled exception is liable to cause a program to crash, this can make exceptions a source of fragility.

The Java language addresses this concern with the checked exception mechanism. First, Java classifies exceptions into two categories:

- Checked exceptions typically represent anticipated events that an application should be able to deal with. For instance, `IOException` and its subtypes represent error conditions that can occur in I/O operations. Examples include, file opens failing because a file or directory does not exist, network reads and writes failing because a network connection has been broken and so on.
- Unchecked exceptions typically represent unanticipated events that an application cannot deal with. These are typically the result of a bug in the application.

(In the following, "thrown" refers to any exception thrown explicitly (by a `throw` statement), or implicitly (in a failed dereference, type cast and so on). Similarly, "propagated" refers to an exception that was thrown in a nested call, and not caught within that call. The sample code below will illustrate this.)

The second part of the checked exception mechanism is that there are restrictions on methods where a checked exception may occur:

- When a checked exception is thrown or propagated in a method, it *must* either be caught by the method, or listed in the method's `throws` clause. (The significance of the `throws` clause is described in this example.)
- When a checked exception is thrown or propagated in an initializer block, it must be caught the the block.
- A checked exception cannot be propagated by a method call in a field initialization expression. (There is no way to catch such an exception.)

In short, a checked exception must be either handled, or declared.

These restrictions do not apply to unchecked exceptions. This includes all cases where an exception is thrown implicitly, since all such cases throw unchecked exceptions.

Checked exception examples

These code snippets are intended to illustrate the checked exception restrictions. In each case, we show a version of the code with a compilation error, and a second version with the error corrected.

```
// This declares a custom checked exception.
public class MyException extends Exception {
    // constructors omitted.
```

```

}

// 这声明了一个自定义的未检查异常。
public class MyException2 extends RuntimeException {
    // 构造函数省略。
}

```

第一个示例展示了如果显式抛出的受检异常不应在方法中处理，可以将其声明为“抛出”。

```

// 错误示范
public void methodThrowingCheckedException(boolean flag) {
    int i = 1 / 0;           // 编译通过，抛出 ArithmeticException
    if (flag) {
        throw new MyException(); // 编译错误
    } else {
        throw new MyException2(); // 编译通过
    }
}

// 修正后
public void methodThrowingCheckedException(boolean flag) throws MyException {
    int i = 1 / 0;           // 编译通过，抛出 ArithmeticException
    if (flag) {
        throw new MyException(); // 编译错误
    } else {
        throw new MyException2(); // 编译通过
    }
}

```

第二个例子展示了如何处理传播的受检异常。

```

// 错误示范
public void methodWithPropagatedCheckedException() {
    InputStream is = new FileInputStream("someFile.txt"); // 编译错误
    // 如果文件无法打开，FileInputStream 会抛出 IOException 或其子类异常
    // IOException 是一种受检异常。
    ...
}

// 修正 (版本 A)
public void methodWithPropagatedCheckedException() throws IOException {
    InputStream is = new FileInputStream("someFile.txt");
    ...
}

// 修正 (版本 B)
public void methodWithPropagatedCheckedException() {
    try {
        InputStream is = new FileInputStream("someFile.txt");
        ...
    } catch (IOException ex) {
        System.out.println("无法打开文件: " + ex.getMessage());
    }
}

```

最后一个示例展示了如何在静态字段初始化器中处理受检异常。

```

// 错误示范
public class Test {

```

```

}

// This declares a custom unchecked exception.
public class MyException2 extends RuntimeException {
    // constructors omitted.
}

```

The first example shows how explicitly thrown checked exceptions can be declared as "thrown" if they should not be handled in the method.

```

// INCORRECT
public void methodThrowingCheckedException(boolean flag) {
    int i = 1 / 0;           // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException(); // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}

// CORRECTED
public void methodThrowingCheckedException(boolean flag) throws MyException {
    int i = 1 / 0;           // Compiles OK, throws ArithmeticException
    if (flag) {
        throw new MyException(); // Compilation error
    } else {
        throw new MyException2(); // Compiles OK
    }
}

```

The second example shows how a propagated checked exception can be dealt with.

```

// INCORRECT
public void methodWithPropagatedCheckedException() {
    InputStream is = new FileInputStream("someFile.txt"); // Compilation error
    // FileInputStream throws IOException or a subclass if the file cannot
    // be opened. IOException is a checked exception.
    ...
}

// CORRECTED (Version A)
public void methodWithPropagatedCheckedException() throws IOException {
    InputStream is = new FileInputStream("someFile.txt");
    ...
}

// CORRECTED (Version B)
public void methodWithPropagatedCheckedException() {
    try {
        InputStream is = new FileInputStream("someFile.txt");
        ...
    } catch (IOException ex) {
        System.out.println("Cannot open file: " + ex.getMessage());
    }
}

```

The final example shows how to deal with a checked exception in a static field initializer.

```

// INCORRECT
public class Test {

```

```

private static final InputStream is =
    new FileInputStream("someFile.txt"); // 编译错误
}

// 修正后
public class Test {
    private static final InputStream is;
    static {
        InputStream tmp = null;
        try {
tmp = new FileInputStream("someFile.txt");
        } catch (IOException ex) {
            System.out.println("无法打开文件: " + ex.getMessage());
        }
is = tmp;
    }
}

```

请注意，在最后这种情况下，我们还必须处理 `is` 不能被赋值多次的问题，
但即使在异常情况下也必须赋值。

第69.8节：创建和读取堆栈跟踪

当异常对象被创建时（即当你 `new` 它时），`Throwable` 构造函数会捕获关于异常创建时上下文的信息。稍后，这些信息可以以堆栈跟踪的形式输出，这可以用来帮助诊断导致异常的根本问题。

打印堆栈跟踪

打印堆栈跟踪只是调用 `printStackTrace()` 方法。例如：

```

try {
    int a = 0;
    int b = 0;
    int c = a / b;
} catch (ArithmaticException ex) {
    // 这会将堆栈跟踪打印到标准输出
    ex.printStackTrace();
}

```

无参的 `printStackTrace()` 方法会打印到应用程序的标准输出；即当前的 `System.out`。还有 `printStackTrace(PrintStream)` 和 `printStackTrace(PrintWriter)` 重载方法，可以打印到指定的 Stream 或 Writer。

备注：

- 堆栈跟踪不包含异常本身的详细信息。你可以使用 `toString()` 方法来获取这些详细信息；例如

```
// 打印异常和堆栈跟踪
System.out.println(ex);
ex.printStackTrace();
```

- 堆栈跟踪打印应谨慎使用；参见陷阱 - 过多或不当的堆栈跟踪。通常更好的是使用日志框架，并传递异常对象进行记录。

理解堆栈跟踪

```

private static final InputStream is =
    new FileInputStream("someFile.txt"); // Compilation error
}

// CORRECTED
public class Test {
    private static final InputStream is;
    static {
        InputStream tmp = null;
        try {
            tmp = new FileInputStream("someFile.txt");
        } catch (IOException ex) {
            System.out.println("Cannot open file: " + ex.getMessage());
        }
        is = tmp;
    }
}

```

Note that in this last case, we also have to deal with the problems that `is` cannot be assigned to more than once, and yet also has to be assigned to, even in the case of an exception.

Section 69.8: Creating and reading stacktraces

When an exception object is created (i.e. when you `new` it), the `Throwable` constructor captures information about the context in which the exception was created. Later on, this information can be output in the form of a stacktrace, which can be used to help diagnose the problem that caused the exception in the first place.

Printing a stacktrace

Printing a stacktrace is simply a matter of calling the `printStackTrace()` method. For example:

```

try {
    int a = 0;
    int b = 0;
    int c = a / b;
} catch (ArithmaticException ex) {
    // This prints the stacktrace to standard output
    ex.printStackTrace();
}

```

The `printStackTrace()` method without arguments will print to the application's standard output; i.e. the current `System.out`. There are also `printStackTrace(PrintStream)` and `printStackTrace(PrintWriter)` overloads that print to a specified Stream or Writer.

Notes:

- 堆栈跟踪不包括异常本身的详细信息。你可以使用 `toString()` 方法来获取那些详细信息；例如

```
// Print exception and stacktrace
System.out.println(ex);
ex.printStackTrace();
```

- 堆栈跟踪打印应谨慎使用；参见陷阱 - 过多或不当的堆栈跟踪。通常更好的是使用日志框架，并传递异常对象进行记录。

Understanding a stacktrace

考虑以下由两个文件中的两个类组成的简单程序。（我们已显示文件名并添加了行号以便说明。）

```
文件: "Main.java"
1 公共类 Main {
2     公共静态 无返回值 main(字符串[] 参数) {
3         新建 Test().foo();
4     }
5 }
```

```
文件: "Test.java"
1 类 Test {
2     公共 void foo() {
3         bar();
4     }
5
6     公共 int bar() {
7         int a = 1;
8         int b = 0;
9         return a / b;
10    }
```

当这些文件被编译并运行时，我们将得到以下输出。

```
Exception in thread "main" java.lang.ArithmaticException: / by zero
at Test.bar(Test.java:9)
在 Test.foo(Test.java:3)
在 Main.main(Main.java:3)
```

让我们一行一行地阅读，弄清楚它告诉了我们什么。

第1行告诉我们，名为“main”的线程由于未捕获的异常而终止。异常的全名是java.lang.ArithmaticException，异常信息是“/ by zero”。

如果我们查阅该异常的javadoc，它写道：

当发生异常的算术条件时抛出。例如，整数“除以零”会抛出该类的一个实例。

确实，信息“/ by zero”强烈暗示异常的原因是某段代码试图将某个值除以零。但是什么呢？

剩下的3行是堆栈跟踪。每一行代表调用栈上的一个方法（或构造函数）调用，每一行告诉我们三件事：

- 正在执行的类和方法的名称，
- 源代码文件名，
- 正在执行的语句所在的源代码行号

堆栈跟踪的这些行按当前调用的栈帧从上到下列出。我们上面示例中的顶部栈帧是在Test.bar方法中，位于Test.java文件的第9行。那是以下这一行：

```
return a / b;
```

Consider the following simple program consisting of two classes in two files. (We have shown the filenames and added line numbers for illustration purposes.)

```
File: "Main.java"
1 public class Main {
2     public static void main(String[] args) {
3         new Test().foo();
4     }
5 }
```

```
File: "Test.java"
1 class Test {
2     public void foo() {
3         bar();
4     }
5
6     public int bar() {
7         int a = 1;
8         int b = 0;
9         return a / b;
10    }
```

When these files are compiled and run, we will get the following output.

```
Exception in thread "main" java.lang.ArithmaticException: / by zero
at Test.bar(Test.java:9)
at Test.foo(Test.java:3)
at Main.main(Main.java:3)
```

Let us read this one line at a time to figure out what it is telling us.

Line #1 tells us that the thread called "main" has terminated due to an uncaught exception. The full name of the exception is `java.lang.ArithmaticException`, and the exception message is "/ by zero".

If we look up the javadocs for this exception, it says:

Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class.

Indeed, the message "/ by zero" is a strong hint that the cause of the exception is that some code has attempted to divide something by zero. But what?

The remaining 3 lines are the stack trace. Each line represents a method (or constructor) call on the call stack, and each one tells us three things:

- the name of the class and method that was being executed,
- the source code filename,
- the source code line number of the statement that was being executed

These lines of a stacktrace are listed with the frame for the current call at the top. The top frame in our example above is in the `Test.bar` method, and at line 9 of the `Test.java` file. That is the following line:

```
return a / b;
```

如果我们查看文件中b初始化的前几行，很明显b的值将是零。我们可以毫无疑问地说这就是异常的原因。

如果需要进一步查看，我们可以从堆栈跟踪中看到bar()是在Test.java第3行由foo()调用的，而foo()又是由Main.main()调用的。

注意：堆栈帧中的类名和方法名是类和方法的内部名称。你需要识别以下特殊情况：

- 嵌套类或内部类看起来像“OuterClass\$InnerClass”。
- 匿名内部类看起来像“OuterClass\$1”、“OuterClass\$2”等等。
- 当构造函数、实例字段初始化器或实例初始化块中的代码正在执行时，方法名将是“”。
- 当静态字段初始化器或静态初始化块中的代码正在执行时，方法名将是“”。

(在某些版本的Java中，堆栈跟踪格式化代码会检测并省略重复的堆栈帧序列，这通常发生在应用程序因过度递归而失败时。)

异常链和嵌套堆栈跟踪

版本 ≥ Java SE 1.4

异常链发生在一段代码捕获异常后，创建并抛出一个新的异常，同时将第一个异常作为原因传递。以下是一个示例：

```
文件: Test.java
1  public class Test {
2      int foo() {
3          return 0 / 0;
4      }
5
6      public Test() {
7          try {
8              foo();
9          } catch (ArithmaticException ex) {
10             throw new RuntimeException("发生了一个错误", ex);
11         }
12     }
13
14     public static void main(String[] args) {
15         new Test();
16     }
17 }
```

当上述类被编译并运行时，我们得到以下堆栈跟踪：

```
Exception in thread "main" java.lang.RuntimeException: 发生了一个错误
    at Test.<init>(Test.java:10)
    at Test.main(Test.java:15)
Caused by: java.lang.ArithmaticException: / by zero
    at Test.foo(Test.java:3)
在 Test.<init>(Test.java:8)
... 1 more
```

堆栈跟踪以类名、方法和调用堆栈开始，表示导致（在本例中）应用程序崩溃的异常。随后是一行“Caused by:”，报告cause异常。接着报告类名和消息，随后是cause异常的堆栈帧。跟踪以“... N more”结尾，表示最后的N帧与前一个异常相同。

If we look a couple of lines earlier in the file to where b is initialized, it is apparent that b will have the value zero. We can say without any doubt that this is the cause of the exception.

If we needed to go further, we can see from the stacktrace that bar() was called from foo() at line 3 of Test.java, and that foo() was in turn called from Main.main().

Note: The class and method names in the stack frames are the internal names for the classes and methods. You will need to recognize the following unusual cases:

- A nested or inner class will look like "OuterClass\$InnerClass".
- An anonymous inner class will look like "OuterClass\$1", "OuterClass\$2", etcetera.
- When code in a constructor, instance field initializer or an instance initializer block is being executed, the method name will be "".
- When code in a static field initializer or static initializer block is being executed, the method name will be "".

(In some versions of Java, the stacktrace formatting code will detect and elide repeated stackframe sequences, as can occur when an application fails due to excessive recursion.)

Exception chaining and nested stacktraces

Version ≥ Java SE 1.4

Exception chaining happens when a piece of code catches an exception, and then creates and throws a new one, passing the first exception as the cause. Here is an example:

```
File: Test.java
1  public class Test {
2      int foo() {
3          return 0 / 0;
4      }
5
6      public Test() {
7          try {
8              foo();
9          } catch (ArithmaticException ex) {
10             throw new RuntimeException("A bad thing happened", ex);
11         }
12     }
13
14     public static void main(String[] args) {
15         new Test();
16     }
17 }
```

When the above class is compiled and run, we get the following stacktrace:

```
Exception in thread "main" java.lang.RuntimeException: A bad thing happened
    at Test.<init>(Test.java:10)
    at Test.main(Test.java:15)
Caused by: java.lang.ArithmaticException: / by zero
    at Test.foo(Test.java:3)
    at Test.<init>(Test.java:8)
... 1 more
```

The stacktrace starts with the class name, method and call stack for the exception that (in this case) caused the application to crash. This is followed by a "Caused by:" line that reports the cause exception. The class name and message are reported, followed by the cause exception's stack frames. The trace ends with an "... N more" which indicates that the last N frames are the same as for the previous exception.

只有当主异常的cause不为null时，输出中才包含“Caused by:”。异常可以无限链式连接，在这种情况下，堆栈跟踪可以包含多个“Caused by:”跟踪。

注意：cause机制仅在Java 1.4.0的Throwable API中公开。在此之前，异常链需要由应用程序通过自定义异常字段表示原因，并通过自定义的printStackTrace方法来实现。

将堆栈跟踪捕获为字符串

有时，应用程序需要能够将堆栈跟踪捕获为Java String，以便用于其他用途。实现这一点的一般方法是创建一个临时的OutputStream或Writer，将其写入内存缓冲区，并将其传递给printStackTrace(...).

Apache Commons和Guava库提供了将堆栈跟踪捕获为字符串的实用方法：

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)  
com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

如果您的代码库中无法使用第三方库，则以下方法可以完成该任务：

```
/**  
 * 返回堆栈跟踪的字符串表示。  
 *  
 * @param throwable 异常对象  
 * @return 字符串。  
 */  
public static String stackTraceToString(Throwable throwable) {  
    StringWriter stringWriter = new StringWriter();  
    throwable.printStackTrace(new PrintWriter(stringWriter));  
    return stringWriter.toString();  
}
```

请注意，如果您的目的是分析堆栈跟踪，使用getStackTrace()和getCause()比尝试解析堆栈跟踪更简单。

第69.9节：抛出异常

下面的示例展示了抛出异常的基本用法：

```
public void checkNumber(int number) throws IllegalArgumentException {  
    if (number < 0) {  
        throw new IllegalArgumentException("数字必须为正数: " + number);  
    }  
}
```

异常在第3行被抛出。该语句可以分解为两部分：

- new `IllegalArgumentException(...)` 是创建了一个`IllegalArgumentException`类的实例，该实例带有描述异常所报告错误的消息。
- `throw ...` 然后抛出异常对象。

当抛出异常时，会导致包含该异常的语句异常终止，直到异常被处理。其他示例中对此有描述。

The "Caused by:" is only included in the output when the primary exception's cause is not `null`). Exceptions can be chained indefinitely, and in that case the stacktrace can have multiple "Caused by:" traces.

Note: the cause mechanism was only exposed in the `Throwable` API in Java 1.4.0. Prior to that, exception chaining needed to be implemented by the application using a custom exception field to represent the cause, and a custom `printStackTrace` method.

Capturing a stacktrace as a String

Sometimes, an application needs to be able to capture a stacktrace as a Java `String`, so that it can be used for other purposes. The general approach for doing this is to create a temporary `OutputStream` or `Writer` that writes to an in-memory buffer and pass that to the `printStackTrace(...)`.

The [Apache Commons](#) and [Guava](#) libraries provide utility methods for capturing a stacktrace as a String:

```
org.apache.commons.lang.exception.ExceptionUtils.getStackTrace(Throwable)  
com.google.common.base.Throwables.getStackTraceAsString(Throwable)
```

If you cannot use third party libraries in your code base, then the following method will do the task:

```
/**  
 * Returns the string representation of the stack trace.  
 *  
 * @param throwable the throwable  
 * @return the string.  
 */  
public static String stackTraceToString(Throwable throwable) {  
    StringWriter stringWriter = new StringWriter();  
    throwable.printStackTrace(new PrintWriter(stringWriter));  
    return stringWriter.toString();  
}
```

Note that if your intention is to analyze the stacktrace, it is simpler to use `getStackTrace()` and `getCause()` than to attempt to parse a stacktrace.

Section 69.9: Throwing an exception

The following example shows the basics of throwing an exception:

```
public void checkNumber(int number) throws IllegalArgumentException {  
    if (number < 0) {  
        throw new IllegalArgumentException("Number must be positive: " + number);  
    }  
}
```

The exception is thrown on the 3rd line. This statement can be broken down into two parts:

- `new IllegalArgumentException(...)` is creating an instance of the `IllegalArgumentException` class, with a message that describes the error that exception is reporting.
- `throw ...` is then throwing the exception object.

When the exception is thrown, it causes the enclosing statements to *terminate abnormally* until the exception is handled. This is described in other examples.

在单个语句中同时创建并抛出异常对象是良好的编程习惯，如上所示。包含有意义的错误信息在异常中同样是良好的做法，这有助于程序员理解问题的原因。然而，这不一定是你应该向最终用户展示的信息。

(首先，Java 并不直接支持异常消息的国际化。)

还有几点需要说明：

- 我们将checkNumber声明为throwsIllegalArgumentException。这并非严格必要，因为IllegalArgumentException是一个非检查异常；参见Java异常层次结构 - 非检查异常和检查异常。不过，这样做是良好的习惯，同时也应在方法的javadoc注释中包含抛出的异常。
- 紧跟在throw语句后的代码是不可达的。因此，如果我们写成这样：

```
throw new IllegalArgumentException("it is bad");
return;
```

编译器会对return语句报编译错误。

异常链

许多标准异常除了常规的

message参数外，还有一个带有第二个cause参数的构造函数。该cause允许你链式连接异常。以下是一个示例。

首先我们定义一个未检查异常，当应用程序遇到不可恢复的错误时会抛出该异常。注意，我们包含了一个接受cause参数的构造函数。

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

接下来，这里有一些代码演示异常链的用法。

```
public String readFirstLine(String file) throws AppErrorException {
    try (Reader r = new BufferedReader(new FileReader(file))) {
        String line = r.readLine();
        if (line != null) {
            return line;
        } else {
            throw new AppErrorException("文件为空: " + file);
        }
    } catch (IOException ex) {
        throw new AppErrorException("无法读取文件: " + file, ex);
    }
}
```

It is good practice to both create and throw the exception object in a single statement, as shown above. It is also good practice to include a meaningful error message in the exception to help the programmer to understand the cause of the problem. However, this is not necessarily the message that you should be showing to the end user. (For a start, Java has no direct support for internationalizing exception messages.)

There are a couple more points to be made:

- We have declared the checkNumber as **throws** `IllegalArgumentException`. This was not strictly necessary, since `IllegalArgumentException` is a checked exception; see The Java Exception Hierarchy - Unchecked and Checked Exceptions. However, it is good practice to do this, and also to include the exceptions thrown a method's javadoc comments.
- Code immediately after a **throw** statement is *unreachable*. Hence if we wrote this:

```
throw new IllegalArgumentException("it is bad");
return;
```

the compiler would report a compilation error for the `return` statement.

Exception chaining

Many standard exceptions have a constructor with a second cause argument in addition to the conventional message argument. The cause allows you to chain exceptions. Here is an example.

First we define an unchecked exception that our application is going to throw when it encounters a non-recoverable error. Note that we have included a constructor that accepts a cause argument.

```
public class AppErrorException extends RuntimeException {
    public AppErrorException() {
        super();
    }

    public AppErrorException(String message) {
        super(message);
    }

    public AppErrorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Next, here is some code that illustrates exception chaining.

```
public String readFirstLine(String file) throws AppErrorException {
    try (Reader r = new BufferedReader(new FileReader(file))) {
        String line = r.readLine();
        if (line != null) {
            return line;
        } else {
            throw new AppErrorException("File is empty: " + file);
        }
    } catch (IOException ex) {
        throw new AppErrorException("Cannot read file: " + file, ex);
    }
}
```

位于try块内的throw检测到问题并通过带有简单消息的异常进行报告。相比之下，位于catch块内的throw通过将IOException包装在一个新的（受检）异常中来处理该异常。

但是，这并不是丢弃原始异常。通过将IOException作为cause传递，我们记录了它，以便可以在堆栈跟踪中打印出来，正如在创建和读取堆栈跟踪中所解释的那样。

第69.10节：异常的高级特性

本示例涵盖了一些异常的高级特性和使用场景。

以编程方式检查调用栈

版本 ≥ Java SE 1.4

异常堆栈跟踪的主要用途是提供有关应用程序错误及其上下文的信息，以便程序员能够诊断和修复问题。有时它也可以用于其他用途。例如，SecurityManager类可能需要检查调用栈，以决定发起调用的代码是否值得信任。

你可以使用异常以编程方式检查调用栈，方法如下：

```
Exception ex = new Exception(); // 这会捕获调用栈
StackTraceElement[] frames = ex.getStackTrace();
System.out.println("此方法是 " + frames[0].getMethodName());
System.out.println("调用自方法 " + frames[1].getMethodName());
```

这里有一些重要的注意事项：

1. StackTraceElement中可用的信息是有限的。没有比这更多的信息可用由printStackTrace显示。（该帧中局部变量的值不可用。）
2. getStackTrace()的javadoc说明JVM允许省略帧：

某些虚拟机在某些情况下，可能会从堆栈跟踪中省略一个或多个堆栈帧。在极端情况下，没有关于此异常的堆栈跟踪信息的虚拟机允许从此方法返回一个长度为零的数组。

优化异常构造

如其他地方所述，构造异常相当昂贵，因为它涉及捕获和记录当前线程上所有堆栈帧的信息。有时，我们知道对于某个异常，这些信息永远不会被使用；例如，堆栈跟踪永远不会被打印。在这种情况下，我们可以在自定义异常中使用一个实现技巧，使得信息不被捕获。

堆栈跟踪所需的堆栈帧信息，是在Throwable构造函数调用Throwable.fillInStackTrace()方法时捕获的。该方法是public的，这意味着子类可以重写它。技巧是重写继承自Throwable的方法，使其不执行任何操作；例如

```
public class MyException extends Exception {
    // 构造函数

    @Override
    public void fillInStackTrace() {
        // 不执行任何操作
    }
}
```

The **throw** within the **try** block detects a problem and reports it via an exception with a simple message. By contrast, the **throw** within the **catch** block is handling the **IOException** by wrapping it in a new (checked) exception. However, it is not throwing away the original exception. By passing the **IOException** as the cause, we record it so that it can be printed in the stacktrace, as explained in Creating and reading stacktraces.

Section 69.10: Advanced features of Exceptions

This example covers some advanced features and use-cases for Exceptions.

Examining the callstack programmatically

Version ≥ Java SE 1.4

The primary use of exception stacktraces is to provide information about an application error and its context so that the programmer can diagnose and fix the problem. Sometimes it can be used for other things. For example, a **SecurityManager** class may need to examine the call stack to decide whether the code that is making a call should be trusted.

You can use exceptions to examine the call stack programmatically as follows:

```
Exception ex = new Exception(); // this captures the call stack
StackTraceElement[] frames = ex.getStackTrace();
System.out.println("This method is " + frames[0].getMethodName());
System.out.println("Called from method " + frames[1].getMethodName());
```

There are some important caveats on this:

1. The information available in a StackTraceElement is limited. There is no more information available than is displayed by printStackTrace. (The values of the local variables in the frame are not available.)
2. The javadocs for getStackTrace() state that a JVM is permitted to leave out frames:

Some virtual machines may, under some circumstances, omit one or more stack frames from the stack trace. In the extreme case, a virtual machine that has no stack trace information concerning this throwable is permitted to return a zero-length array from this method.

Optimizing exception construction

As mentioned elsewhere, constructing an exception is rather expensive because it entails capturing and recording information about all stack frames on the current thread. Sometimes, we know that that information is never going to be used for a given exception; e.g. the stacktrace will never be printed. In that case, there is an implementation trick that we can use in a custom exception to cause the information to not be captured.

The stack frame information needed for stacktraces, is captured when the **Throwable** constructors call the **Throwable.fillInStackTrace()** method. This method is **public**, which means that a subclass can override it. The trick is to override the method inherited from **Throwable** with one that does nothing; e.g.

```
public class MyException extends Exception {
    // constructors

    @Override
    public void fillInStackTrace() {
        // do nothing
    }
}
```

```
}
```

这种方法的问题在于，重写了fillInStackTrace()的异常永远无法捕获堆栈跟踪，在需要堆栈跟踪的场景中毫无用处。

擦除或替换堆栈跟踪

版本 ≥ Java SE 1.4

在某些情况下，正常方式创建的异常的堆栈跟踪包含错误的信息，或者包含开发者不希望向用户透露的信息。针对这些情况，`Throwable.setStackTrace` 可以用来替换保存信息的 `StackTraceElement` 对象数组。

例如，下面的代码可以用来丢弃异常的堆栈信息：

```
exception.setStackTrace(new StackTraceElement[0]);
```

被抑制的异常

版本 ≥ Java SE 7

Java 7 引入了 `try-with-resources` 结构，以及相关的异常抑制概念。考虑以下代码片段：

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // 执行操作
    int temp = 0 / 0;    // 抛出 ArithmeticException
}
```

当异常被抛出时，`try` 会调用 `close()` 方法关闭 `w`，该方法会刷新任何缓冲的输出，然后关闭 `FileWriter`。但如果在刷新输出时抛出 `IOException` 会发生什么？

发生的情况是，在清理资源时抛出的任何异常都会被抑制。该异常会被捕获，并添加到主异常的被抑制异常列表中。接着，`try-with-resources` 会继续清理其他资源。最后，主异常会被重新抛出。

如果在资源初始化期间抛出异常，或者`try` 块正常完成，也会出现类似的情况。第一个抛出的异常成为主要异常，随后在清理过程中产生的异常将被抑制。

可以通过调用主要异常对象的

`getSuppressedExceptions` 方法来获取被抑制的异常。

第69.11节：try-finally和try-catch-finally语句

`try...catch...finally` 语句将异常处理与清理代码结合起来。`finally` 块包含将在所有情况下执行的代码。这使它们适用于资源管理和其他类型的清理工作。

Try-finally

下面是更简单的 (`try...finally`) 形式的示例：

```
try {
    doSomething();
} finally {
    cleanUp();
```

```
}
```

The problem with this approach is that an exception that overrides `fillInStackTrace()` can never capture the stacktrace, and is useless in scenarios where you need one.

Erasing or replacing the stacktrace

Version ≥ Java SE 1.4

In some situations, the stacktrace for an exception created in the normal way contains either incorrect information, or information that the developer does not want to reveal to the user. For these scenarios, the `Throwable.setStackTrace` can be used to replace the array of `StackTraceElement` objects that holds the information.

For example, the following can be used to discard an exception's stack information:

```
exception.setStackTrace(new StackTraceElement[0]);
```

Suppressed exceptions

Version ≥ Java SE 7

Java 7 introduced the `try-with-resources` construct, and the associated concept of exception suppression. Consider the following snippet:

```
try (Writer w = new BufferedWriter(new FileWriter(someFilename))) {
    // do stuff
    int temp = 0 / 0;    // throws an ArithmeticException
}
```

When the exception is thrown, the `try` will call `close()` on the `w` which will flush any buffered output and then close the `FileWriter`. But what happens if an `IOException` is thrown while flushing the output?

What happens is that any exception that is thrown while cleaning up a resource is suppressed. The exception is caught, and added to the primary exception's suppressed exception list. Next the `try-with-resources` will continue with the cleanup of the other resources. Finally, primary exception will be rethrown.

A similar pattern occurs if an exception is thrown during the resource initialization, or if the `try` block completes normally. The first exception thrown becomes the primary exception, and subsequent ones arising from cleanup are suppressed.

The suppressed exceptions can be retrieved from the primary exception object by calling `getSuppressedExceptions`.

Section 69.11: The try-finally and try-catch-finally statements

The `try...catch...finally` statement combines exception handling with clean-up code. The `finally` block contains code that will be executed in all circumstances. This makes them suitable for resource management, and other kinds of cleanup.

Try-finally

Here is an example of the simpler (`try...finally`) form:

```
try {
    doSomething();
} finally {
    cleanUp();
```

}

try...finally 的行为如下：

- 在try块中的代码将被执行。
- 如果try块中没有抛出异常：
 - 在finally块中的代码将被执行。
 - 如果finally块抛出异常，该异常将被传播。
 - 否则，控制权将传递到try...finally之后的下一条语句。
- 如果在try块中抛出了异常：
 - 在finally块中的代码将被执行。
 - 如果finally块抛出异常，该异常将被传播。
 - 否则，原始异常将继续传播。

finally块中的代码总是会被执行。（唯一的例外是调用了System.exit(int)或JVM崩溃。）因此，finally块是放置必须始终执行代码的正确位置；例如关闭文件和其他资源或释放锁。

try-catch-finally

我们的第二个示例展示了如何将catch和finally一起使用。它还说明了清理资源并非易事。

```
// 这段代码将文件的第一行写入字符串
String result = null;
Reader reader = null;
try {
    reader = new BufferedReader(new FileReader(fileName));
    result = reader.readLine();
} catch (IOException ex) {
    Logger.getLogger().warn("Unexpected IO error", ex); // 记录异常日志
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException ex) {
            // 忽略 / 丢弃此异常
        }
    }
}
```

本例中 try...catch...finally 的（假设的）完整行为过于复杂，无法在此详细描述。简单来说，finally 块中的代码总会被执行。

从资源管理的角度来看：

- 我们在 try 块之前声明“资源”（即 `reader` 变量），以便它在 `finally` 块中仍然可见。
- 通过使用`newFileReader(...)`，catch能够处理打开文件时抛出的任何`IOError`异常。
- 我们需要在 finally 块中调用`reader.close()`，因为存在一些异常路径，我们既无法在try块中拦截，也无法在catch块中拦截。
- 但是，由于在`reader`初始化之前可能已经抛出了异常，我们还需要显式进行`null`测试。
- 最后，调用`reader.close()`可能（假设性地）会抛出异常。我们不关心这个异常，但如果不在源头捕获它，就需要在调用栈的更高层处理它。

}

The behavior of the `try...finally` is as follows:

- The code in the `try` block is executed.
- If no exception was thrown in the `try` block:
 - The code in the `finally` block is executed.
 - If the `finally` block throws an exception, that exception is propagated.
 - Otherwise, control passes to the next statement after the `try...finally`.
- If an exception was thrown in the `try` block:
 - The code in the `finally` block is executed.
 - If the `finally` block throws an exception, that exception is propagated.
 - Otherwise, the original exception continues to propagate.

The code within `finally` block will always be executed. (The only exceptions are if `System.exit(int)` is called, or if the JVM panics.) Thus a `finally` block is the correct place code that always needs to be executed; e.g. closing files and other resources or releasing locks.

try-catch-finally

Our second example shows how `catch` and `finally` can be used together. It also illustrates that cleaning up resources is not straightforward.

```
// This code snippet writes the first line of a file to a string
String result = null;
Reader reader = null;
try {
    reader = new BufferedReader(new FileReader(fileName));
    result = reader.readLine();
} catch (IOException ex) {
    Logger.getLogger().warn("Unexpected IO error", ex); // logging the exception
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException ex) {
            // ignore / discard this exception
        }
    }
}
```

The complete set of (hypothetical) behaviors of `try...catch...finally` in this example are too complicated to describe here. The simple version is that the code in the `finally` block will always be executed.

Looking at this from the perspective of resource management:

- We declare the "resource" (i.e. `reader` variable) before the `try` block so that it will be in scope for the `finally` block.
- By putting the `new FileReader(...)`，the `catch` is able to handle any `IOError` exception from thrown when opening the file.
- We need a `reader.close()` in the `finally` block because there are some exception paths that we cannot intercept either in the `try` block or in `catch` block.
- However, since an exception *might* have been thrown before `reader` was initialized, we also need an explicit `null` test.
- Finally, the `reader.close()` call might (hypothetically) throw an exception. We don't care about that, but if we don't catch the exception at source, we would need to deal with it further up the call stack.

Java 7及以后版本提供了另一种try-with-resources语法，大大简化了资源的清理工作。

第69.12节：方法声明中的'throws'子句

Java的checked exception机制要求程序员声明某些方法可能抛出指定的已检查异常。这是通过throws子句完成的。例如：

```
public class OddNumberException extends Exception { // 一个已检查异常
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

throws OddNumberException声明调用checkEven可能会抛出类型为OddNumberException的异常。

一个throws子句可以声明一个类型列表，并且可以包含未检查异常以及已检查异常。

```
public void checkEven(Double number)
    throws OddNumberException, ArithmeticException {
    if (!Double.isFinite(number)) {
        throw new ArithmeticException("INF or NaN");
    } else if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

声明抛出未检查异常的意义是什么？

方法声明中的throws子句有两个目的：

1. 它告诉编译器哪些异常会被抛出，以便编译器可以将未捕获的（已检查）异常报告为错误。
2. 它告诉调用该方法的程序员应预期哪些异常。为此，通常将未检查异常包含在throws列表中是有意义的。

注意：throws列表也被javadoc工具用于生成API文档，并被典型的IDE的“悬停文本”方法提示所使用。

Throws与方法重写

throws子句构成方法签名的一部分，用于方法重写。重写方法可以声明与被重写方法抛出的已检查异常相同的异常集合，或者是其子集。

但是，重写方法不能添加额外的已检查异常。例如：

```
@Override
public void checkEven(int number) throws NullPointerException // OK—NullPointerException 是一个
    未检查异常
...
```

Java 7 and later provide an alternative try-with-resources syntax which significantly simplifies resource clean-up.

Section 69.12: The 'throws' clause in a method declaration

Java's *checked exception* mechanism requires the programmer to declare that certain methods *could* throw specified checked exceptions. This is done using the **throws** clause. For example:

```
public class OddNumberException extends Exception { // a checked exception
}

public void checkEven(int number) throws OddNumberException {
    if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

The **throws** OddNumberException declares that a call to checkEven *could* throw an exception that is of type OddNumberException.

A **throws** clause can declare a list of types, and can include unchecked exceptions as well as checked exceptions.

```
public void checkEven(Double number)
    throws OddNumberException, ArithmeticException {
    if (!Double.isFinite(number)) {
        throw new ArithmeticException("INF or NaN");
    } else if (number % 2 != 0) {
        throw new OddNumberException();
    }
}
```

What is the point of declaring unchecked exceptions as thrown?

The **throws** clause in a method declaration serves two purposes:

1. It tells the compiler which exceptions are thrown so that the compiler can report uncaught (checked) exceptions as errors.
2. It tells a programmer who is writing code that calls the method what exceptions to expect. For this purpose, it often makes sense to include unchecked exceptions in a **throws** list.

Note: that the **throws** list is also used by the javadoc tool when generating API documentation, and by a typical IDE's "hover text" method tips.

Throws and method overriding

The **throws** clause forms part of a method's signature for the purpose of method overriding. An override method can be declared with the same set of checked exceptions as thrown by the overridden method, or with a subset. However the override method cannot add extra checked exceptions. For example:

```
@Override
public void checkEven(int number) throws NullPointerException // OK—NullPointerException is an
    unchecked exception
...
```

```
@Override  
public void checkEven(Double number) throws OddNumberException // 正确—与超类相同
```

```
...  
  
class PrimeNumberException extends OddNumberException {}  
class NonEvenNumberException extends OddNumberException {}  
  
@Override  
public void checkEven(int number) throws PrimeNumberException, NonEvenNumberException // 正确—这两个都是子类
```

```
@Override  
public void checkEven(Double number) throws IOExcepion // 错误
```

此规则的原因是，如果重写的方法可以抛出被重写方法不能抛出的受检异常，这将破坏类型替换性。

```
@Override  
public void checkEven(Double number) throws OddNumberException // OK—identical to the superclass  
...  
  
class PrimeNumberException extends OddNumberException {}  
class NonEvenNumberException extends OddNumberException {}  
  
@Override  
public void checkEven(int number) throws PrimeNumberException, NonEvenNumberException // OK—these  
are both subclasses  
  
@Override  
public void checkEven(Double number) throws IOExcepion // ERROR
```

The reason for this rule is that if an overridden method can throw a checked exception that the overridden method could not throw, that would break type substitutability.

第70章：日历及其子类

第70.1节：创建日历对象

可以通过使用 `getInstance()` 或使用构造函数 `GregorianCalendar` 来创建日历对象。

需要注意的是，`Calendar` 中的月份是从零开始的，这意味着一月用整数值 0 表示。为了编写更好的代码，始终使用 `C` `alendar` 常量，例如 `Calendar.JANUARY`，以避免误解。

```
Calendar calendar = Calendar.getInstance();
Calendar gregorianCalendar = new GregorianCalendar();
Calendar gregorianCalendarAtSpecificDay = new GregorianCalendar(2016, Calendar.JANUARY, 1);
Calendar gregorianCalendarAtSpecificDayAndTime = new GregorianCalendar(2016, Calendar.JANUARY, 1, 6, 55, 10);
```

注意：始终使用月份常量：数字表示具有误导性，例如`Calendar.JANUARY`的值为0

第70.2节：增加/减少日历字段

`add()`和`roll()`可用于增加/减少`Calendar`字段。

```
Calendar calendar = new GregorianCalendar(2016, Calendar.MARCH, 31); // 2016年3月31日
```

`add()`方法影响所有字段，且在对日历进行实际日期的加减时表现良好

```
calendar.add(Calendar.MONTH, -6);
```

上述操作从日历中减去六个月，回到2015年9月30日。

要更改特定字段而不影响其他字段，请使用`roll()`。

```
calendar.roll(Calendar.MONTH, -6);
```

上述操作从当前month中减去六个月，因此月份被识别为九月。没有调整其他字段；年份在此操作中未发生变化。

第70.3节：日历相减

要获取两个`Calendar`之间的差异，使用`getTimeInMillis()`方法：

```
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c2.set(Calendar.DATE, c2.get(Calendar.DATE) + 1);

System.out.println(c2.getTimeInMillis() - c1.getTimeInMillis()); //输出 86400000 (24 * 60 * 60 * 1000)
```

第70.4节：查找上午/下午

使用 `Calendar` 类很容易判断是上午还是下午。

Chapter 70: Calendar and its Subclasses

Section 70.1: Creating Calendar objects

`Calendar` objects can be created by using `getInstance()` or by using the constructor `GregorianCalendar`.

It's important to notice that months in `Calendar` are zero based, which means that JANUARY is represented by an `int` value 0. In order to provide a better code, always use `Calendar` constants, such as `Calendar.JANUARY` to avoid misunderstandings.

```
Calendar calendar = Calendar.getInstance();
Calendar gregorianCalendar = new GregorianCalendar();
Calendar gregorianCalendarAtSpecificDay = new GregorianCalendar(2016, Calendar.JANUARY, 1);
Calendar gregorianCalendarAtSpecificDayAndTime = new GregorianCalendar(2016, Calendar.JANUARY, 1, 6, 55, 10);
```

Note: Always use the month constants: The numeric representation is misleading, e.g. `Calendar.JANUARY` has the value 0

Section 70.2: Increasing / Decreasing calendar fields

`add()` and `roll()` can be used to increase/decrease `Calendar` fields.

```
Calendar calendar = new GregorianCalendar(2016, Calendar.MARCH, 31); // 31 March 2016
```

The `add()` method affects all fields, and behaves effectively if one were to add or subtract actual dates from the calendar

```
calendar.add(Calendar.MONTH, -6);
```

The above operation removes six months from the calendar, taking us back to 30 September 2015.

To change a particular field without affecting the other fields, use `roll()`.

```
calendar.roll(Calendar.MONTH, -6);
```

The above operation removes six months from the current `month`, so the month is identified as September. No other fields have been adjusted; the year has not changed with this operation.

Section 70.3: Subtracting calendars

To get a difference between two `Calendars`, use `getTimeInMillis()` method:

```
Calendar c1 = Calendar.getInstance();
Calendar c2 = Calendar.getInstance();
c2.set(Calendar.DATE, c2.get(Calendar.DATE) + 1);

System.out.println(c2.getTimeInMillis() - c1.getTimeInMillis()); //outputs 86400000 (24 * 60 * 60 * 1000)
```

Section 70.4: Finding AM/PM

With `Calendar` class it is easy to find AM or PM.

```
Calendar cal = Calendar.getInstance();
cal.setTime(new Date());
if (cal.get(Calendar.AM_PM) == Calendar.PM)
    System.out.println("现在是下午");
```

```
Calendar cal = Calendar.getInstance();
cal.setTime(new Date());
if (cal.get(Calendar.AM_PM) == Calendar.PM)
    System.out.println("It is PM");
```

第71章：使用 static 关键字

第71.1节：从静态上下文引用非静态成员

静态变量和方法不是实例的一部分，无论你创建多少个特定类的对象，该变量始终只有一份副本。

例如，你可能想拥有一个不可变的常量列表，将其设为静态并只在静态方法中初始化一次是个好主意。如果你经常创建某个类的多个实例，这将显著提升性能。

此外，类中还可以有静态代码块。你可以用它为静态变量赋默认值。它们只在类被加载到内存时执行一次。

实例变量顾名思义依赖于特定对象的实例，它们的生命周期服务于该实例。你可以在对象的特定生命周期内操作它们。

在类的静态方法内部使用的所有字段和方法必须是静态的或局部的。如果你尝试使用实例（非静态）变量或方法，代码将无法编译。

```
public class Week {  
    static int daysOfTheWeek = 7; // 静态变量  
    int dayOfTheWeek; // 实例变量  
  
    public static int getDaysLeftInWeek(){  
        return Week.daysOfTheWeek-dayOfTheWeek; // 这将导致错误  
    }  
  
    public int getDaysLeftInWeek(){  
        return Week.daysOfTheWeek-dayOfTheWeek; // 这是有效的  
    }  
  
    public static int getDaysLeftInTheWeek(int today){  
        return Week.daysOfTheWeek-today; // 这是有效的  
    }  
}
```

第71.2节：使用static声明常量

由于static关键字用于在未实例化类的情况下访问字段和方法，它也可以用于声明供其他类使用的常量。这些变量在类的每个实例中保持不变。按照惯例，static变量总是使用全大写字母并用下划线代替驼峰命名。例如：

```
static E STATIC_VARIABLE_NAME
```

由于常量不可更改，static也可以与final修饰符一起使用：

例如，定义数学常数圆周率：

```
public class MathUtilities {  
  
    static final double PI = 3.14159265358
```

Chapter 71: Using the static keyword

Section 71.1: Reference to non-static member from static context

Static variables and methods are not part of an instance, There will always be a single copy of that variable no matter how many objects you create of a particular class.

For example you might want to have an immutable list of constants, it would be a good idea to keep it static and initialize it just once inside a static method. This would give you a significant performance gain if you are creating several instances of a particular class on a regular basis.

Furthermore you can also have a static block in a class as well. You can use it to assign a default value to a static variable. They are executed only once when the class is loaded into memory.

Instance variable as the name suggest are dependent on an instance of a particular object, they live to serve the whims of it. You can play around with them during a particular life cycle of an object.

All the fields and methods of a class used inside a static method of that class must be static or local. If you try to use instance (non-static) variables or methods, your code will not compile.

```
public class Week {  
    static int daysOfTheWeek = 7; // static variable  
    int dayOfTheWeek; // instance variable  
  
    public static int getDaysLeftInWeek(){  
        return Week.daysOfTheWeek-dayOfTheWeek; // this will cause errors  
    }  
  
    public int getDaysLeftInWeek(){  
        return Week.daysOfTheWeek-dayOfTheWeek; // this is valid  
    }  
  
    public static int getDaysLeftInTheWeek(int today){  
        return Week.daysOfTheWeek-today; // this is valid  
    }  
}
```

Section 71.2: Using static to declare constants

As the **static** keyword is used for accessing fields and methods without an instantiated class, it can be used to declare constants for use in other classes. These variables will remain constant across every instantiation of the class. By convention, **static** variables are always ALL_CAPS and use underscores rather than camel case. ex:

```
static E STATIC_VARIABLE_NAME
```

As constants cannot change, **static** can also be used with the **final** modifier:

For example, to define the mathematical constant of pi:

```
public class MathUtilities {  
  
    static final double PI = 3.1415926538
```

}

它可以作为常量在任何类中使用，例如：

```
public class MathCalculations {  
  
    //计算圆的周长  
    public double calculateCircumference(double radius) {  
        return (2 * radius * MathUtilities.PI);  
    }  
}
```

}

Which can be used in any class as a constant, for example:

```
public class MathCalculations {  
  
    //Calculates the circumference of a circle  
    public double calculateCircumference(double radius) {  
        return (2 * radius * MathUtilities.PI);  
    }  
}
```

第72章：Properties类

Properties对象包含键和值，二者均为字符串。java.util.Properties类是Hashtable的子类。

它可以根据属性键获取属性值。Properties 类提供了从属性文件获取数据和将数据存储到属性文件的方法。此外，它还可以用来获取系统属性。

属性文件的优点

如果属性文件中的信息发生变化，则不需要重新编译：如果任何信息发生变化

第72.1节：加载属性

加载随应用程序捆绑的属性文件：

```
public class Defaults {  
  
    public static Properties loadDefaults() {  
        try (InputStream bundledResource =  
             Defaults.class.getResourceAsStream("defaults.properties")) {  
  
            Properties defaults = new Properties();  
            defaults.load(bundledResource);  
            return defaults;  
        } catch (IOException e) {  
            // 由于资源是随应用程序捆绑的,  
            // 我们理论上永远不会到达这里。  
            throw new UncheckedIOException(  
                "defaults.properties 未正确打包"  
                + " 与应用程序", e);  
        }  
    }  
}
```

第72.2节：将属性保存为XML

将属性存储在XML文件中

将属性文件存储为XML文件的方式与存储为.properties文件的方式非常相似。只不过不是使用store()，而是使用storeToXML()。

```
public void saveProperties(String location) throws IOException{  
    // 创建属性的新实例  
    Properties prop = new Properties();  
  
    // 设置属性值  
    prop.setProperty("name", "Steve");  
    prop.setProperty("color", "green");  
    prop.setProperty("age", "23");  
  
    // 检查文件是否存在  
    File file = new File(location);  
    if (!file.exists()){  
        file.createNewFile();  
    }
```

Chapter 72: Properties Class

The properties object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from properties file and store data into properties file. Moreover, it can be used to get properties of system.

Advantage of properties file

Recompilation is not required, if information is changed from properties file: If any information is changed from

Section 72.1: Loading properties

To load a properties file bundled with your application:

```
public class Defaults {  
  
    public static Properties loadDefaults() {  
        try (InputStream bundledResource =  
             Defaults.class.getResourceAsStream("defaults.properties")) {  
  
            Properties defaults = new Properties();  
            defaults.load(bundledResource);  
            return defaults;  
        } catch (IOException e) {  
            // Since the resource is bundled with the application,  
            // we should never get here.  
            throw new UncheckedIOException(  
                "defaults.properties not properly packaged"  
                + " with application", e);  
        }  
    }  
}
```

Section 72.2: Saving Properties as XML

Storing Properties in a XML File

The way you store properties files as XML files is very similar to the way you would store them as .properties files. Just instead of using the store() you would use storeToXML().

```
public void saveProperties(String location) throws IOException{  
    // make new instance of properties  
    Properties prop = new Properties();  
  
    // set the property values  
    prop.setProperty("name", "Steve");  
    prop.setProperty("color", "green");  
    prop.setProperty("age", "23");  
  
    // check to see if the file already exists  
    File file = new File(location);  
    if (!file.exists()){  
        file.createNewFile();  
    }
```

```
// 保存属性
prop.storeToXML(new FileOutputStream(file), "使用xml测试属性");
}
```

当你打开文件时，它将显示如下内容。

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3 =<properties>
4 <comment>testing properties with xml</comment>
5 <entry key="age">23</entry>
6 <entry key="color">green</entry>
7 <entry key="name">Steve</entry>
8 </properties>
9
```

从XML文件加载属性

现在要将此文件作为properties加载，你需要调用loadFromXML()，而不是常规.properties文件中使用的load()。

```
public static void loadProperties(String location) throws FileNotFoundException, IOException{
    // 创建新的属性实例以加载文件
    Properties prop = new Properties();

    // 检查文件是否存在
    File file = new File(location);
    if (file.exists()){
        // 加载文件
        prop.loadFromXML(new FileInputStream(file));

        // 输出所有属性
        for (String name : prop.stringPropertyNames()){
            System.out.println(name + "=" + prop.getProperty(name));
        }
    } else {
        System.err.println("错误：未找到文件，路径为：" + location);
    }
}
```

运行此代码时，控制台将显示以下内容：

```
age=23
color=green
name=Steve
```

第72.3节：属性文件注意事项：尾随空白

仔细看看这两个看似完全相同的属性文件：

1 # Example 1	1 # Example 2
2	2
3 lastName=Smith	3 lastName=Smith
4	4

但它们实际上并不相同：

```
// save the properties
prop.storeToXML(new FileOutputStream(file), "testing properties with xml");
}
```

When you open the file it will look like this.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
3 =<properties>
4 <comment>testing properties with xml</comment>
5 <entry key="age">23</entry>
6 <entry key="color">green</entry>
7 <entry key="name">Steve</entry>
8 </properties>
9
```

Loading Properties from a XML File

Now to load this file as a properties you need to call the loadFromXML() instead of the load() that you would use with regular .properties files.

```
public static void loadProperties(String location) throws FileNotFoundException, IOException{
    // make new properties instance to load the file into
    Properties prop = new Properties();

    // check to make sure the file exists
    File file = new File(location);
    if (file.exists()){
        // load the file
        prop.loadFromXML(new FileInputStream(file));

        // print out all the properties
        for (String name : prop.stringPropertyNames()){
            System.out.println(name + "=" + prop.getProperty(name));
        }
    } else {
        System.err.println("Error: No file found at: " + location);
    }
}
```

When you run this code you will get the following in the console:

```
age=23
color=green
name=Steve
```

Section 72.3: Property files caveat: trailing whitespace

Take a close look at these two property files which are seemingly completely identical:

1 # Example 1	1 # Example 2
2	2
3 lastName=Smith	3 lastName=Smith
4	4

except they are really not identical:

```

1 # Example 1CR/LF
2 CR/LF
3 lastName=SmithCR/LF
4

```

```

1 # Example 2CR/LF
2 CR/LF
3 lastName=Smith CR/LF
4

```

(截图来自记事本++)

由于尾部空白被保留，lastName 的值在第一种情况下是"Smith"，在第二种情况下是"Smith "。

用户很少期望如此，且只能猜测为什么这是Properties类的默认行为。然而，创建一个改进版的Properties来解决此问题是很容易的。以下类，TrimmedProperties，正是如此。它是标准Properties类的直接替代品。

```

import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.Reader;
import java.util.Map.Entry;
import java.util.Properties;

/**
 * 属性类，如果属性是从文件加载的，则会去除值末尾的空白字符。
 *
 * <p>
 * 在标准的 {@link java.util.Properties Properties} 类中，尾部空白字符总是被保留。当从文件加载属性时，这种尾部空白几乎总是<i>无意的</i>。该类解决了这个问题。只有当输入来源是文件且输入是按行处理时（例如，从 XML 文件加载的情况<i>不会</i>被该类更改），才会去除尾部空白。
 *
 * 因此，在几乎所有情况下，该类都是标准 <tt>Properties</tt> 的安全替代品
 *
 * 类。
 *
 * <p>
 * 空白符在此定义为任意空格 (U+0020) 或制表符 (U+0009)。
 *
 */
public class TrimmedProperties extends Properties {

    /**
     * 从输入字节流读取属性列表（键和值对）。
     *
     * <p>行为完全等同于 {@link java.util.Properties#load(java.io.InputStream)}。
     * 唯一的区别是如果 <tt>inStream</tt> 是 <tt>FileInputStream</tt> 的实例，则会去除属性值末尾的空白字符。
     *
     * @see java.util.Properties#load(java.io.InputStream)
     * @param inStream 输入流。
     * @throws IOException 如果从输入流读取时发生错误。
     */
    @Override
    public void load(InputStream inStream) throws IOException {
        if (inStream instanceof FileInputStream) {
            // 首先使用标准方式读取到临时属性中
            Properties tempProps = new Properties();
            tempProps.load(inStream);

```

```

1 # Example 1CR/LF
2 CR/LF
3 lastName=SmithCR/LF
4

```

```

1 # Example 2CR/LF
2 CR/LF
3 lastName=Smith CR/LF
4

```

(screenshots are from Notepad++)

Since trailing whitespace is preserved the value of lastName would be "Smith" in the first case and "Smith " in the second case.

Very rarely this is what users expect and one can only speculate why this is the default behavior of Properties class. It is however easy to create an enhanced version of Properties that fixes this problem. The following class, TrimmedProperties, does just that. It is a drop-in replacement for standard Properties class.

```

import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.Reader;
import java.util.Map.Entry;
import java.util.Properties;

/**
 * Properties class where values are trimmed for trailing whitespace if the properties are loaded from a file.
 *
 * <p>
 * In the standard {@link java.util.Properties Properties} class trailing whitespace is always preserved. When loading properties from a file such trailing whitespace is almost always <i>unintentional</i>. This class fixes this problem. The trimming of trailing whitespace only takes place if the source of input is a file and only where the input is line oriented (meaning that for example loading from XML file is <i>not</i> changed by this class).
 * For this reason this class is almost in all cases a safe drop-in replacement for the standard <tt>Properties</tt> class.
 *
 * <p>
 * Whitespace is defined here as any of space (U+0020) or tab (U+0009).
 *
 */
public class TrimmedProperties extends Properties {

    /**
     * Reads a property list (key and element pairs) from the input byte stream.
     *
     * <p>Behaves exactly as {@link java.util.Properties#load(java.io.InputStream)} with the exception that trailing whitespace is trimmed from property values if <tt>inStream</tt> is an instance of <tt>FileInputStream</tt>.
     *
     * @see java.util.Properties#load(java.io.InputStream)
     * @param inStream the input stream.
     * @throws IOException if an error occurred when reading from the input stream.
     */
    @Override
    public void load(InputStream inStream) throws IOException {
        if (inStream instanceof FileInputStream) {
            // First read into temporary props using the standard way
            Properties tempProps = new Properties();
            tempProps.load(inStream);

```

```

    // 现在修剪并放入目标
trimAndLoad(tempProps);
} else {
    super.load(inStream);
}
}

/**
* 从输入字符流中以简单的行导向格式读取属性列表（键和值对）。
*
* <p>行为与{@link java.util.Properties#load(java.io.Reader)}完全相同，  

* <tt>reader</tt>是<tt>FileReader</tt>实例时，属性值的尾部空白会被修剪。  

*
* @see java.util.Properties#load(java.io.Reader)
* @param reader 输入字符流。
* @throws IOException 如果从输入流读取时发生错误。
*/
@Override
public void load(Reader reader) throws IOException {
    if (reader instanceof FileReader) {
        // 首先使用标准方式读取到临时属性中
        Properties tempProps = new Properties();
        tempProps.load(reader);
        // 现在修剪并放入目标
        trimAndLoad(tempProps);
    } else {
        super.load(reader);
    }
}

private void trimAndLoad(Properties p) {
    for (Entry<Object, Object> entry : p.entrySet()) {
        if (entry.getValue() instanceof String) {
            put(entry.getKey(), trimTrailing((String) entry.getValue()));
        } else {
            put(entry.getKey(), entry.getValue());
        }
    }
}

/**
* 去除字符串末尾的空格或制表符。
*
* @param str
* @return
*/
public static String trimTrailing(String str) {
    if (str != null) {
        // 从字符串尾部开始读取，直到字符不再是空白字符
        for (int i = str.length() - 1; i >= 0; i--) {
            if ((str.charAt(i) != ' ') && (str.charAt(i) != '\t')) {
                return str.substring(0, i + 1);
            }
        }
    }
    return str;
}

```

```

    // Now trim and put into target
    trimAndLoad(tempProps);
} else {
    super.load(inStream);
}

/**
* Reads a property list (key and element pairs) from the input character stream in a simple
line-oriented format.
*
* <p>Behaves exactly as {@link java.util.Properties#load(java.io.Reader)}
* with the exception that trailing whitespace is trimmed on property values
* if <tt>reader</tt> is an instance of <tt>FileReader</tt>.
*
* @see java.util.Properties#load(java.io.Reader)
* @param reader the input character stream.
* @throws IOException if an error occurred when reading from the input stream.
*/
@Override
public void load(Reader reader) throws IOException {
    if (reader instanceof FileReader) {
        // First read into temporary props using the standard way
        Properties tempProps = new Properties();
        tempProps.load(reader);
        // Now trim and put into target
        trimAndLoad(tempProps);
    } else {
        super.load(reader);
    }
}

private void trimAndLoad(Properties p) {
    for (Entry<Object, Object> entry : p.entrySet()) {
        if (entry.getValue() instanceof String) {
            put(entry.getKey(), trimTrailing((String) entry.getValue()));
        } else {
            put(entry.getKey(), entry.getValue());
        }
    }
}

/**
* Trims trailing space or tabs from a string.
*
* @param str
* @return
*/
public static String trimTrailing(String str) {
    if (str != null) {
        // read str from tail until char is no longer whitespace
        for (int i = str.length() - 1; i >= 0; i--) {
            if ((str.charAt(i) != ' ') && (str.charAt(i) != '\t')) {
                return str.substring(0, i + 1);
            }
        }
    }
    return str;
}

```

第73章：Lambda表达式

Lambda表达式提供了一种清晰简洁的方式，通过表达式实现单方法接口。它们可以减少你需要编写和维护的代码量。虽然与匿名类类似，但它们本身没有类型信息，需要进行类型推断。

方法引用使用现有方法而非表达式来实现函数式接口。它们也属于lambda家族。

第73.1节：Java Lambda表达式简介

函数式接口

Lambda 表达式只能作用于函数式接口，即仅包含一个抽象方法的接口。函数式接口可以包含任意数量的默认或静态方法。（因此，它们有时也被称为单一抽象方法接口，或 SAM 接口）。

```
接口 Foo1 {
    void bar();
}

接口 Foo2 {
    int bar(boolean baz);
}

接口 Foo3 {
    String bar(Object baz, int mink);
}

接口 Foo4 {
    默认 String bar() { // 默认方法不计入抽象方法
        return "baz";
    }
    void quux();
}
```

声明函数式接口时，可以添加@FunctionalInterface注解。该注解本身没有特殊效果，但如果将其应用于非函数式接口，编译器将报错，从而提醒接口不应被更改。

```
@FunctionalInterface
接口 Foo5 {
    void bar();
}

@FunctionalInterface
接口 BlankFoo1 extends Foo3 { // 继承自 Foo3 的抽象方法
}

@FunctionalInterface
接口 Foo6 {
    void bar();
    boolean equals(Object obj); // 重写了 Object 的一个方法，因此不计入抽象方法
}
```

相反，这不是函数式接口，因为它有超过一个抽象方法：

Chapter 73: Lambda Expressions

Lambda expressions provide a clear and concise way of implementing a single-method interface using an expression. They allow you to reduce the amount of code you have to create and maintain. While similar to anonymous classes, they have no type information by themselves. Type inference needs to happen.

Method references implement functional interfaces using existing methods rather than expressions. They belong to the lambda family as well.

Section 73.1: Introduction to Java lambdas

Functional Interfaces

Lambdas can only operate on a functional interface, which is an interface with just one abstract method. Functional interfaces can have any number of **default** or **static** methods. (For this reason, they are sometimes referred to as Single Abstract Method Interfaces, or SAM Interfaces).

```
interface Foo1 {
    void bar();
}

interface Foo2 {
    int bar(boolean baz);
}

interface Foo3 {
    String bar(Object baz, int mink);
}

interface Foo4 {
    default String bar() { // default so not counted
        return "baz";
    }
    void quux();
}
```

When declaring a functional interface the [@FunctionalInterface](#) annotation can be added. This has no special effect, but a compiler error will be generated if this annotation is applied to an interface which is not functional, thus acting as a reminder that the interface should not be changed.

```
@FunctionalInterface
interface Foo5 {
    void bar();
}

@FunctionalInterface
interface BlankFoo1 extends Foo3 { // inherits abstract method from Foo3
}

@FunctionalInterface
interface Foo6 {
    void bar();
    boolean equals(Object obj); // overrides one of Object's method so not counted
}
```

Conversely, this is **not** a functional interface, as it has more than **one abstract** method:

```
接口 BadFoo {
    void bar();
    void quux(); // <- 第二个方法阻止了使用 lambda：应该将哪个
                 // 视为 lambda？
}
```

这也不是函数式接口，因为它没有任何方法：

```
interface BlankFoo2 { }
```

请注意以下内容。假设你有

```
接口 Parent { public int parentMethod(); }
```

和

```
接口 Child extends Parent { public int ChildMethod(); }
```

那么Child 不能是一个函数式接口，因为它有两个指定的方法。

Java 8 还在包 `java.util.function` 中提供了许多泛型模板函数式接口。例如，内置接口 `Predicate<T>` 封装了一个单一方法，该方法输入类型为 `T` 的值并返回一个 `boolean`。

```
interface BadFoo {
    void bar();
    void quux(); // <- Second method prevents lambda: which one should
                 // be considered as lambda?
}
```

This is **also not** a functional interface, as it does not have any methods:

```
interface BlankFoo2 { }
```

Take note of the following. Suppose you have

```
interface Parent { public int parentMethod(); }
```

and

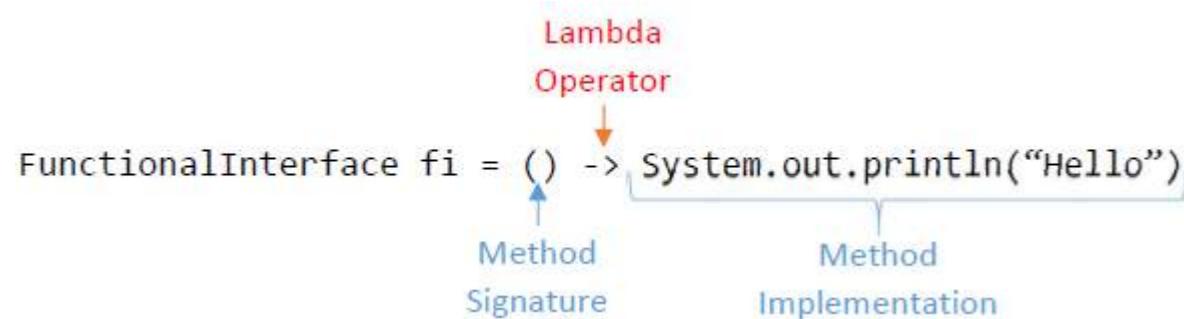
```
interface Child extends Parent { public int ChildMethod(); }
```

Then Child **cannot** be a functional interface since it has two specified methods.

Java 8 also provides a number of generic templated functional interfaces in the package `java.util.function`. For example, the built-in interface `Predicate<T>` wraps a single method which inputs a value of type `T` and returns a `boolean`.

Lambda 表达式

Lambda 表达式的基本结构是：



`fi` 将持有一个类的单例实例，类似于匿名类，该类实现了 `FunctionalInterface` 并且该方法的定义是 `{ System.out.println("Hello"); }`。换句话说，上述内容大致等同于：

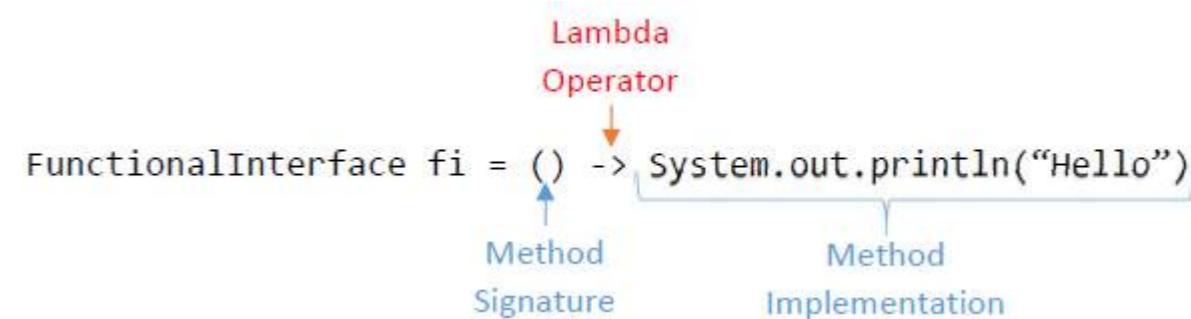
```
FunctionalInterface fi = new FunctionalInterface() {
    @Override
    public void theOneMethod() {
        System.out.println("Hello");
    }
};
```

lambda 表达式只是“基本等同”于匿名类，因为在 lambda 中，表达式如 `this`、`super` 或 `toString()` 的含义是指赋值发生的类，而不是新创建的对象。

使用 lambda 时不能指定方法名—但你也不需要这样做，因为函数式接口必须只有一个抽象方法，所以 Java 会覆盖那个方法。

Lambda Expressions

The basic structure of a Lambda expression is:



`fi` will then hold a singleton instance of a class, similar to an anonymous class, which implements `FunctionalInterface` and where the one method's definition is `{ System.out.println("Hello"); }`. In other words, the above is mostly equivalent to:

```
FunctionalInterface fi = new FunctionalInterface() {
    @Override
    public void theOneMethod() {
        System.out.println("Hello");
    }
};
```

The lambda is only "mostly equivalent" to the anonymous class because in a lambda, the meaning of expressions like `this`, `super` or `toString()` reference the class within which the assignment takes place, not the newly created object.

You cannot specify the name of the method when using a lambda—but you shouldn't need to, because a functional interface must have only one abstract method, so Java overrides that one.

在 lambda 类型不确定的情况下（例如重载方法），你可以给 lambda 添加类型转换，告诉编译器它的类型，如下所示：

```
Object fooHolder = (Foo1) () -> System.out.println("Hello");
System.out.println(fooHolder instanceof Foo1); // 返回 true
```

如果函数式接口的单个方法带有参数，这些参数的局部形式名称应出现在 lambda 的括号内。参数类型和返回类型无需声明，因为它们来自接口（虽然声明参数类型也不是错误）。因此，以下两个示例是等价的：

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

如果函数只有一个参数，参数周围的括号可以省略：

```
Foo2 np = x -> { return x ? 1 : 0; }; // 可以
Foo3 np2 = x, y -> x.toString() + y // 不可以
```

隐式返回

如果放在 lambda 中的代码是一个 Java 表达式 而不是一个 语句，则它被视为一个返回该表达式值的方法。因此，以下两者是等价的：

```
IntUnaryOperator addOneShort = (x) -> (x + 1);
IntUnaryOperator addOneLong = (x) -> { return (x + 1); };
```

访问局部变量（值闭包）

由于 lambda 是匿名类的语法简写，它们遵循访问封闭作用域中局部变量的相同规则；这些变量必须被视为 final，且不能在 lambda 内部被修改。

```
IntUnaryOperator makeAdder(int amount) {
    return (x) -> (x + amount); // 合法，尽管 amount 会超出作用域
                                // 因为 amount 没有被修改
}

IntUnaryOperator makeAccumulator(int value) {
    return (x) -> { value += x; return value; }; // 无法编译
}
```

如果必须以这种方式包装一个可变变量，应使用一个普通对象来保存该变量的副本。更多内容请参见 Java 中带有 lambda 表达式的闭包。

接受Lambda表达式

因为Lambda是接口的一个实现，所以方法接受Lambda时不需要做特殊处理：任何接受函数式接口的函数也可以接受Lambda。

In cases where the type of the lambda is not certain, (e.g. overloaded methods) you can add a cast to the lambda to tell the compiler what its type should be, like so:

```
Object fooHolder = (Foo1) () -> System.out.println("Hello");
System.out.println(fooHolder instanceof Foo1); // returns true
```

If the functional interface's single method takes parameters, the local formal names of these should appear between the brackets of the lambda. There is no need to declare the type of the parameter or return as these are taken from the interface (although it is not an error to declare the parameter types if you want to). Thus, these two examples are equivalent:

```
Foo2 longFoo = new Foo2() {
    @Override
    public int bar(boolean baz) {
        return baz ? 1 : 0;
    }
};
Foo2 shortFoo = (x) -> { return x ? 1 : 0; };
```

The parentheses around the argument can be omitted if the function only has one argument:

```
Foo2 np = x -> { return x ? 1 : 0; }; // okay
Foo3 np2 = x, y -> x.toString() + y // not okay
```

Implicit Returns

If the code placed inside a lambda is a Java *expression* rather than a *statement*, it is treated as a method which returns the value of the expression. Thus, the following two are equivalent:

```
IntUnaryOperator addOneShort = (x) -> (x + 1);
IntUnaryOperator addOneLong = (x) -> { return (x + 1); };
```

Accessing Local Variables (value closures)

Since lambdas are syntactic shorthand for anonymous classes, they follow the same rules for accessing local variables in the enclosing scope; the variables must be treated as **final** and not modified inside the lambda.

```
IntUnaryOperator makeAdder(int amount) {
    return (x) -> (x + amount); // Legal even though amount will go out of scope
                                // because amount is not modified
}

IntUnaryOperator makeAccumulator(int value) {
    return (x) -> { value += x; return value; }; // Will not compile
}
```

If it is necessary to wrap a changing variable in this way, a regular object that keeps a copy of the variable should be used. Read more in Java Closures with lambda expressions.

Accepting Lambdas

Because a lambda is an implementation of an interface, nothing special needs to be done to make a method accept a lambda: any function which takes a functional interface can also accept a lambda.

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

Lambda表达式的类型

Lambda表达式本身没有特定的类型。虽然参数的类型和数量以及返回值的类型可以传达一些类型信息，但这些信息仅限制它可以赋值给哪些类型。Lambda在被赋值给函数式接口类型时会获得类型，方式如下：

- 直接赋值给函数式类型，例如 `myPredicate = s -> s.isEmpty()`
- 将其作为具有函数式类型的参数传递，例如 `stream.filter(s -> s.isEmpty())`
- 从返回函数式类型的函数返回，例如 `return s -> s.isEmpty()`
- 将其强制转换为函数式类型，例如 `(Predicate<String>) s -> s.isEmpty()`

在赋值给函数式类型之前，Lambda没有确定的类型。举例说明，考虑Lambda表达式 `o -> o.isEmpty()`。相同的Lambda表达式可以赋值给多种不同的函数式类型：

```
Predicate<String> javaStringPred = o -> o.isEmpty();
Function<String, Boolean> javaFunc = o -> o.isEmpty();
Predicate<List> javaListPred = o -> o.isEmpty();
Consumer<String> javaStringConsumer = o -> o.isEmpty(); // 返回值被忽略!
com.google.common.base.Predicate<String> guavaPredicate = o -> o.isEmpty();
```

现在它们被赋值了，虽然lambda表达式看起来相同，但示例显示的是完全不同的类型，且它们不能相互赋值。

第73.2节：使用Lambda表达式对集合进行排序

排序列表

在Java 8之前，排序列表时需要使用匿名（或命名）类实现[java.util.Comparator](#)接口1：

```
版本 ≥ Java SE 1.2
List<Person> people = ...
Collections.sort(
    people,
    new Comparator<Person>() {
        public int compare(Person p1, Person p2){
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
);
```

从Java 8开始，匿名类可以用lambda表达式替代。注意，参数p1和p2的类型可以省略，因为编译器会自动推断它们：

```
Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);
```

这个例子可以通过使用[Comparator.comparing](#)和用::（双冒号）符号表示的方法引用来自来简化。

```
public void passMeALambda(Foo1 f) {
    f.bar();
}
passMeALambda(() -> System.out.println("Lambda called"));
```

The Type of a Lambda Expression

A lambda expression, by itself, does not have a specific type. While it is true that the types and number of parameters, along with the type of a return value can convey some type information, such information will only constrain what types it can be assigned to. The lambda receives a type when it is assigned to a functional interface type in one of the following ways:

- Direct assignment to a functional type, e.g. `myPredicate = s -> s.isEmpty()`
- Passing it as a parameter that has a functional type, e.g. `stream.filter(s -> s.isEmpty())`
- Returning it from a function that returns a functional type, e.g. `return s -> s.isEmpty()`
- Casting it to a functional type, e.g. `(Predicate<String>) s -> s.isEmpty()`

Until any such assignment to a functional type is made, the lambda does not have a definite type. To illustrate, consider the lambda expression `o -> o.isEmpty()`. The same lambda expression can be assigned to many different functional types:

```
Predicate<String> javaStringPred = o -> o.isEmpty();
Function<String, Boolean> javaFunc = o -> o.isEmpty();
Predicate<List> javaListPred = o -> o.isEmpty();
Consumer<String> javaStringConsumer = o -> o.isEmpty(); // return value is ignored!
com.google.common.base.Predicate<String> guavaPredicate = o -> o.isEmpty();
```

Now that they are assigned, the examples shown are of completely different types even though the lambda expressions looked the same, and they cannot be assigned to each other.

Section 73.2: Using Lambda Expressions to Sort a Collection

Sorting lists

Prior to Java 8, it was necessary to implement the [java.util.Comparator](#) interface with an anonymous (or named) class when sorting a list1:

```
Version ≥ Java SE 1.2
List<Person> people = ...
Collections.sort(
    people,
    new Comparator<Person>() {
        public int compare(Person p1, Person p2){
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
);
```

Starting with Java 8, the anonymous class can be replaced with a lambda expression. Note that the types for the parameters p1 and p2 can be left out, as the compiler will infer them automatically:

```
Collections.sort(
    people,
    (p1, p2) -> p1.getFirstName().compareTo(p2.getFirstName())
);
```

The example can be simplified by using [Comparator.comparing](#) and method references expressed using the ::

(双冒号)符号。

```
Collections.sort(  
    people,  
    Comparator.comparing(Person::getFirstName)  
)
```

静态导入允许我们更简洁地表达，但是否能提升整体可读性还有争议：

```
import static java.util.Collections.sort;  
import static java.util.Comparator.comparing;  
//...  
sort(people, comparing(Person::getFirstName));
```

这样构建的比较器也可以链式调用。例如，先按名字比较人，如果有同名的，则使用thenComparing方法再按姓氏比较：

```
sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));
```

1 - 注意，Collections.sort(...) 仅适用于 List 的子类型集合。 Set 和 Collection API 不保证元素的任何顺序。

排序映射

您可以以类似的方式按值对HashMap的条目进行排序。（注意，必须使用LinkedHashMap作为目标。普通的HashMap中的键是无序的。）

```
Map<String, Integer> map = new HashMap(); // ... 或任何其他 Map 类  
// 填充映射  
map = map.entrySet()  
    .stream()  
.sorted(Map.Entry.<String, Integer>comparingByValue())  
    .collect(Collectors.toMap(k -> k.getKey(), v -> v.getValue(),  
        (k, v) -> k, LinkedHashMap::new));
```

第73.3节：方法引用

方法引用允许将符合兼容函数式接口的预定义静态或实例方法作为参数传递，而不是使用匿名lambda表达式。

假设我们有一个模型：

```
类 Person {  
    私有最终String name;  
    私有最终String surname;  
  
    public Person(String name, String surname){  
        this.name = name;  
        this.surname = surname;  
    }  
  
    public String getName(){ return name; }  
    public String getSurname(){ return surname; }  
}
```

(double colon) symbol.

```
Collections.sort(  
    people,  
    Comparator.comparing(Person::getFirstName)  
)
```

A static import allows us to express this more concisely, but it is debatable whether this improves overall readability:

```
import static java.util.Collections.sort;  
import static java.util.Comparator.comparing;  
//...  
sort(people, comparing(Person::getFirstName));
```

Comparators built this way can also be chained together. For example, after comparing people by their first name, if there are people with the same first name, the thenComparing method will also compare by last name:

```
sort(people, comparing(Person::getFirstName).thenComparing(Person::getLastName));
```

1 - Note that Collections.sort(...) only works on collections that are subtypes of [List](#). The [Set](#) and [Collection](#) APIs do not imply any ordering of the elements.

Sorting maps

You can sort the entries of a [HashMap](#) by value in a similar fashion. (Note that a [LinkedHashMap](#) must be used as the target. The keys in an ordinary [HashMap](#) are unordered.)

```
Map<String, Integer> map = new HashMap(); // ... or any other Map class  
// populate the map  
map = map.entrySet()  
    .stream()  
.sorted(Map.Entry.<String, Integer>comparingByValue())  
    .collect(Collectors.toMap(k -> k.getKey(), v -> v.getValue(),  
        (k, v) -> k, LinkedHashMap::new));
```

Section 73.3: Method References

Method references allow predefined static or instance methods that adhere to a compatible functional interface to be passed as arguments instead of an anonymous lambda expression.

Assume that we have a model:

```
class Person {  
    private final String name;  
    private final String surname;  
  
    public Person(String name, String surname){  
        this.name = name;  
        this.surname = surname;  
    }  
  
    public String getName(){ return name; }  
    public String getSurname(){ return surname; }  
}
```

```
List<Person> people = getSomePeople();
```

任意实例的方法引用

```
people.stream().map(Person::getName)
```

等价的 Lambda 表达式：

```
people.stream().map(person -> person.getName())
```

在此示例中，传递了类型为 Person 的实例方法 getName() 的方法引用。由于已知它是集合类型，稍后将调用该实例上的方法。

实例方法引用（针对特定实例）

```
people.forEach(System.out::println);
```

由于 System.out 是 PrintStream 的一个实例，传递的是对该特定实例的方法引用作为参数。

等价的 Lambda 表达式：

```
people.forEach(person -> System.out.println(person));
```

静态方法引用

对于转换流，我们也可以应用对静态方法的引用：

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream().map(String::valueOf)
```

此示例传递了对 String 类型的静态 valueOf() 方法的引用。因此，集合中的实例对象作为参数传递给 valueOf()。

等价的 Lambda 表达式：

```
numbers.stream().map(num -> String.valueOf(num))
```

构造函数引用

```
List<String> strings = Arrays.asList("1", "2", "3");
strings.stream().map(Integer::new)
```

读取流中的元素到集合中，了解如何将元素收集到集合。

这里使用了 Integer 类型的单个字符串参数构造函数，根据提供的字符串参数构造一个整数。在这种情况下，只要字符串表示一个数字，流就会被映射为整数。等价的 lambda 表达式：

```
strings.stream().map(s -> new Integer(s));
```

速查表

方法引用格式

静态方法

代码

TypeName::method (args) -> TypeName.method(args)

等价的Lambda表达式

```
List<Person> people = getSomePeople();
```

Instance method reference (to an arbitrary instance)

```
people.stream().map(Person::getName)
```

The equivalent lambda:

```
people.stream().map(person -> person.getName())
```

In this example, a method reference to the instance method getName() of type Person, is being passed. Since it's known to be of the collection type, the method on the instance (known later) will be invoked.

Instance method reference (to a specific instance)

```
people.forEach(System.out::println);
```

Since System.out is an instance of PrintStream, a method reference to this specific instance is being passed as an argument.

The equivalent lambda:

```
people.forEach(person -> System.out.println(person));
```

Static method reference

Also for transforming streams we can apply references to static methods:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
numbers.stream().map(String::valueOf)
```

This example passes a reference to the static valueOf() method on the String type. Therefore, the instance object in the collection is passed as an argument to valueOf().

The equivalent lambda:

```
numbers.stream().map(num -> String.valueOf(num))
```

Reference to a constructor

```
List<String> strings = Arrays.asList("1", "2", "3");
strings.stream().map(Integer::new)
```

Read Collect Elements of a Stream into a Collection to see how to collect elements to collection.

The single String argument constructor of the Integer type is being used here, to construct an integer given the string provided as the argument. In this case, as long as the string represents a number, the stream will be mapped to Integers. The equivalent lambda:

```
strings.stream().map(s -> new Integer(s));
```

Cheat-Sheet

Method Reference Format

Static method

Code

TypeName::method (args) -> TypeName.method(args)

Equivalent Lambda

```
非静态方法 (在实例上*) instance::method(args) -> instance.method(args)
非静态方法 (无实例) TypeName::method(instance, args) -> instance.method(args)
构造函数**
    TypeName::new (args) -> new TypeName(args)
数组构造器
    TypeName[]::new(int size) -> new TypeName[size]
```

* instance 可以是任何求值为实例引用的表达式，例如 getInstance()::method,
this::method

** 如果 TypeName 是非静态内部类，构造器引用仅在外部类实例的作用域内有效

```
Non-static method (on instance*) instance::method (args) -> instance.method(args)
Non-static method (no instance) TypeName::method (instance, args) -> instance.method(args)
Constructor**
    TypeName::new (args) -> new TypeName(args)
Array constructor
    TypeName[]::new (int size) -> new TypeName[size]
```

* instance can be any expression that evaluates to a reference to an instance, e.g. getInstance()::method,
this::method

** If TypeName is a non-static inner class, constructor reference is only valid within the scope of an outer class
instance

第73.4节：实现多个接口

有时你可能希望一个lambda表达式实现多个接口。这在标记接口（如 java.io.Serializable）中最为有用，因为它们不添加抽象方法。

例如，你想创建一个带有自定义 Comparator 的 TreeSet，然后序列化并通过网络发送。简单的方法：

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

不起作用，因为比较器的lambda没有实现 Serializable。你可以通过使用交叉类型并显式指定该lambda需要可序列化来修复：

```
TreeSet<Long> ts = new TreeSet<>(
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

如果你经常使用交叉类型（例如，如果你使用像Apache Spark这样的框架，几乎所有东西都必须是可序列化的），你可以创建空接口并在代码中使用它们，示例如下：

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}  
  
public class CustomTreeSet {  
    public CustomTreeSet(SerializableComparator comparator) {}  
}
```

这样你就能保证传入的比较器是可序列化的。

第73.5节：Lambda - 监听器示例

匿名类监听器

在Java 8之前，使用匿名类来处理Button的点击事件非常常见，如下面的代码所示。此示例展示了如何在btn的作用域内实现匿名监听器
btn.addActionListener。

```
JButton btn = new JButton("我的按钮");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("按钮被按下");
    }
});
```

Section 73.4: Implementing multiple interfaces

Sometimes you may want to have a lambda expression implementing more than one interface. This is mostly useful with marker interfaces (such as [java.io.Serializable](#)) since they don't add abstract methods.

For example, you want to create a [TreeSet](#) with a custom [Comparator](#) and then serialize it and send it over the network. The trivial approach:

```
TreeSet<Long> ts = new TreeSet<>((x, y) -> Long.compare(y, x));
```

doesn't work since the lambda for the comparator does not implement [Serializable](#). You can fix this by using intersection types and explicitly specifying that this lambda needs to be serializable:

```
TreeSet<Long> ts = new TreeSet<>(
    (Comparator<Long> & Serializable) (x, y) -> Long.compare(y, x));
```

If you're frequently using intersection types (for example, if you're using a framework such as [Apache Spark](#) where almost everything has to be serializable), you can create empty interfaces and use them in your code instead:

```
public interface SerializableComparator extends Comparator<Long>, Serializable {}  
  
public class CustomTreeSet {  
    public CustomTreeSet(SerializableComparator comparator) {}  
}
```

This way you're guaranteed that the passed comparator will be serializable.

Section 73.5: Lambda - Listener Example

Anonymous class listener

Before Java 8, it's very common that an anonymous class is used to handle click event of a JButton, as shown in the following code. This example shows how to implement an anonymous listener within the scope of btn.addActionListener.

```
JButton btn = new JButton("My Button");
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button was pressed");
    }
});
```

Lambda 监听器

因为 ActionListener接口只定义了一个方法actionPerformed(), 所以它是一个函数式接口这意味着可以使用 Lambda 表达式来替代样板代码。上述示例可以使用 Lambda 表达式重新编写如下：

```
JButton btn = new JButton("我的按钮");
btn.addActionListener(e -> {
    System.out.println("按钮被按下");
});
```

第73.6节：使用 Lambda 表达式的 Java 闭包

当 Lambda 表达式引用了外部作用域（全局或局部）的变量时，就创建了一个 Lambda 闭包。对此的规则与内联方法和匿名类相同。

在 Lambda 中使用的外部作用域的局部变量必须是final。对于支持 Lambda 的最早版本 Java 8 来说，外部上下文中不需要显式声明为final，但必须以这种方式对待。例如：

```
int n = 0; // 在 Java 8 中不需要显式声明 final
Runnable r = () -> { // 使用 Lambda
    int i = n;
    // 执行某些操作
};
```

只要 n 变量的值不被改变，这种写法是合法的。如果你尝试在 Lambda 内外修改该变量，将会出现以下编译错误：

“从 lambda 表达式引用的局部变量必须是final或有效的 final”。

例如：

```
int n = 0;
Runnable r = () -> { // 使用 lambda
    int i = n;
    // 执行某些操作
};
n++; // 将会产生错误。
```

如果必须在 lambda 中使用可变变量，通常的做法是声明一个final的变量副本并使用该副本。例如

```
int n = 0;
final int k = n; // 在 Java 8 中不需要显式声明 final
Runnable r = () -> { // 使用 lambda
    int i = k;
    // 执行某些操作
};
n++; // 现在不会产生错误
r.run(); // 运行时 i = 0, 因为 k 在创建 lambda 时是 0
```

自然地，lambda 体内看不到对原始变量的更改。

请注意，Java 不支持真正的闭包。Java 的 lambda 无法以允许其观察实例化环境中变化的方式创建。

Lambda listener

Because the [ActionListener](#) interface defines only one method actionPerformed(), it is a functional interface which means there's a place to use Lambda expressions to replace the boilerplate code. The above example can be re-written using Lambda expressions as follows:

```
JButton btn = new JButton("My Button");
btn.addActionListener(e -> {
    System.out.println("Button was pressed");
});
```

Section 73.6: Java Closures with lambda expressions

A lambda closure is created when a lambda expression references the variables of an enclosing scope (global or local). The rules for doing this are the same as those for inline methods and anonymous classes.

Local variables from an enclosing scope that are used within a lambda have to be **final**. With Java 8 (the earliest version that supports lambdas), they don't need to be *declared final* in the outside context, but must be treated that way. For example:

```
int n = 0; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
```

This is legal as long as the value of the n variable is not changed. If you try to change the variable, inside or outside the lambda, you will get the following compilation error:

“local variables referenced from a lambda expression must be *final* or *effectively final*”.

For example:

```
int n = 0;
Runnable r = () -> { // Using lambda
    int i = n;
    // do something
};
n++; // Will generate an error.
```

If it is necessary to use a changing variable within a lambda, the normal approach is to declare a **final** copy of the variable and use the copy. For example

```
int n = 0;
final int k = n; // With Java 8 there is no need to explicit final
Runnable r = () -> { // Using lambda
    int i = k;
    // do something
};
n++; // Now will not generate an error
r.run(); // Will run with i = 0 because k was 0 when the lambda was created
```

Naturally, the body of the lambda does not see the changes to the original variable.

Note that Java does not support true closures. A Java lambda cannot be created in a way that allows it to see

如果你想实现一个能够观察或修改其环境变化的闭包，应该使用常规类来模拟它。例如：

```
// 无法编译 ...
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

上述示例由于之前讨论的原因无法编译。我们可以通过以下方式绕过编译错误：

```
// 可以编译，但不正确 ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

问题在于这破坏了IntUnaryOperator接口的设计契约，该接口规定实例应当是函数式且无状态的。如果将这样的闭包传递给接受函数式对象的内置函数，可能会导致崩溃或错误行为。封装可变状态的闭包应当实现为普通类。例如。

```
// 正确示例 ...
public class 累加器 {
    private int 值 = 0;

    public int 累加(int x) {
        值 += x;
        return 值;
    }
}
```

第73.7节：Lambda表达式与内存利用

由于Java的lambda是闭包，它们可以“捕获”封闭词法作用域中变量的值。虽然并非所有lambda都会捕获任何东西——像 `s -> s.length()` 这样的简单lambda不捕获任何东西，被称为无状态——捕获变量的lambda需要一个临时对象来保存捕获的变量。在这段代码片段中，`lambda () -> j` 是一个捕获变量的lambda，评估时可能会导致对象分配：

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

虽然代码片段中没有出现`new`关键字，这可能不太明显，但这段代码可能会创建10亿个独立对象来表示`() -> j` lambda表达式的实例。然而，也应注意，未来版本的Java1可能能够优化这一点，使得在运行时lambda实例被重用，或其他方式表示。

changes in the environment in which it was instantiated. If you want to implement a closure that observes or makes changes to its environment, you should simulate it using a regular class. For example:

```
// Does not compile ...
public IntUnaryOperator createAccumulator() {
    int value = 0;
    IntUnaryOperator accumulate = (x) -> { value += x; return value; };
    return accumulate;
}
```

The above example will not compile for reasons discussed previously. We can work around the compilation error as follows:

```
// Compiles, but is incorrect ...
public class AccumulatorGenerator {
    private int value = 0;

    public IntUnaryOperator createAccumulator() {
        IntUnaryOperator accumulate = (x) -> { value += x; return value; };
        return accumulate;
    }
}
```

The problem is that this breaks the design contract for the IntUnaryOperator interface which states that instances should be functional and stateless. If such a closure is passed to built-in functions that accept functional objects, it is liable to cause crashes or erroneous behavior. Closures that encapsulate mutable state should be implemented as regular classes. For example.

```
// Correct ...
public class Accumulator {
    private int value = 0;

    public int accumulate(int x) {
        value += x;
        return value;
    }
}
```

Section 73.7: Lambdas and memory utilization

Since Java lambdas are closures, they can "capture" the values of variables in the enclosing lexical scope. While not all lambdas capture anything -- simple lambdas like `s -> s.length()` capture nothing and are called *stateless* -- capturing lambdas require a temporary object to hold the captured variables. In this code snippet, the `lambda () -> j` is a capturing lambda, and may cause an object to be allocated when it is evaluated:

```
public static void main(String[] args) throws Exception {
    for (int i = 0; i < 1000000000; i++) {
        int j = i;
        doSomethingWithLambda(() -> j);
    }
}
```

Although it might not be immediately obvious since the `new` keyword doesn't appear anywhere in the snippet, this code is liable to create 1,000,000,000 separate objects to represent the instances of the `() -> j` lambda expression. However, it should also be noted that future versions of Java1 may be able to optimize this so that *at runtime* the lambda instances were reused, or were represented in some other way.

1 - 例如，Java 9引入了一个可选的“链接”阶段到Java构建序列中，这将提供进行此类全局优化的机会。

1 - For instance, Java 9 introduces an optional "link" phase to the Java build sequence which will provide the opportunity for doing global optimizations like this.

第73.8节：使用lambda表达式与您自己的函数式接口

Lambda旨在为单方法接口提供内联实现代码，并能够像使用普通变量一样传递它们。我们称之为函数式接口。

例如，使用匿名类编写Runnable并启动线程的写法如下：

```
//旧方法
new Thread(
    new Runnable(){
        public void run(){
            System.out.println("run logic...");
        }
    }
).start();

//lambdas, 来自Java 8
new Thread(
    ()-> System.out.println("运行逻辑...")
).start();
```

现在，按照上述内容，假设你有一个自定义接口：

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

如何使用lambda表达式为该接口在代码中提供实现？和上面展示的Runnable示例一样。见下面的驱动程序：

```
public class CustomLambda {
    public static void main(String[] args) {

        TwoArgInterface plusOperation = (a, b) -> a + b;
        TwoArgInterface divideOperation = (a,b)->{
            if (b==0) throw new IllegalArgumentException("除数不能为0");
            return a/b;
        };

        System.out.println("3和5的加法操作结果是: " + plusOperation.operate(3, 5));
        System.out.println("50除以25的除法操作结果是: " + divideOperation.operate(50, 25));
    }
}
```

第73.9节：传统风格到Lambda风格

传统方式

```
interface MathOperation{
    boolean unaryOperation(int num);
}
```

Section 73.8: Using lambda expression with your own functional interface

Lambdas are meant to provide inline implementation code for single method interfaces and the ability to pass them around as we have been doing with normal variables. We call them Functional Interface.

For example, writing a Runnable in anonymous class and starting a Thread looks like:

```
//Old way
new Thread(
    new Runnable(){
        public void run(){
            System.out.println("run logic...");
        }
    }
).start();

//lambdas, from Java 8
new Thread(
    ()-> System.out.println("run logic...")
).start();
```

Now, in line with above, lets say you have some custom interface:

```
interface TwoArgInterface {
    int operate(int a, int b);
}
```

How do you use lambda to give implementation of this interface in your code? Same as Runnable example shown above. See the driver program below:

```
public class CustomLambda {
    public static void main(String[] args) {

        TwoArgInterface plusOperation = (a, b) -> a + b;
        TwoArgInterface divideOperation = (a,b)->{
            if (b==0) throw new IllegalArgumentException("Divisor can not be 0");
            return a/b;
        };

        System.out.println("Plus operation of 3 and 5 is: " + plusOperation.operate(3, 5));
        System.out.println("Divide operation 50 by 25 is: " + divideOperation.operate(50, 25));
    }
}
```

Section 73.9: Traditional style to Lambda style

Traditional way

```
interface MathOperation{
    boolean unaryOperation(int num);
}
```

```

public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = new MathOperation() {
            @Override
            public boolean unaryOperation(int num) {
                return num%2 == 0;
            }
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}

```

Lambda 风格

1. 去掉类名和函数式接口体。

```

public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = (int num) -> {
            return num%2 == 0;
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}

```

2. 可选类型声明

```

MathOperation isEven = (num) -> {
    return num%2 == 0;
};

```

3. 如果参数只有一个，可省略括号

```

MathOperation isEven = num -> {
    return num%2 == 0;
};

```

4. 如果函数体只有一行，可省略大括号

5. 如果函数体只有一行，可省略 return 关键字

```

MathOperation isEven = num -> num%2 == 0;

```

第73.10节：`return` 只从 lambda 返回，而不是从外层方法返回

return 方法只从 lambda 返回，而不是从外层方法返回。

注意，这与 Scala 和 Kotlin 是不同的！

```

void threeTimes(IntConsumer r) {
    for (int i = 0; i < 3; i++) {
        r.accept(i);
    }
}

```

```

public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = new MathOperation() {
            @Override
            public boolean unaryOperation(int num) {
                return num%2 == 0;
            }
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}

```

Lambda style

1. Remove class name and functional interface body.

```

public class LambdaTry {
    public static void main(String[] args) {
        MathOperation isEven = (int num) -> {
            return num%2 == 0;
        };

        System.out.println(isEven.unaryOperation(25));
        System.out.println(isEven.unaryOperation(20));
    }
}

```

2. Optional type declaration

```

MathOperation isEven = (num) -> {
    return num%2 == 0;
};

```

3. Optional parenthesis around parameter, if it is single parameter

```

MathOperation isEven = num -> {
    return num%2 == 0;
};

```

4. Optional curly braces, if there is only one line in function body

5. Optional return keyword, if there is only one line in function body

```

MathOperation isEven = num -> num%2 == 0;

```

Section 73.10: `return` only returns from the lambda, not the outer method

The `return` method only returns from the lambda, not the outer method.

Beware that this is *different* from Scala and Kotlin!

```

void threeTimes(IntConsumer r) {
    for (int i = 0; i < 3; i++) {
        r.accept(i);
    }
}

```

```

void demo() {
    threeTimes(i -> {
        System.out.println(i);
        return; // 仅从 lambda 返回到 threeTimes !
    });
}

```

当尝试编写自定义语言结构时，可能会导致意想不到的行为，例如在内置结构如 **for** 循环中，**return** 的行为不同：

```

void demo2() {
    for (int i = 0; i < 3; i++) {
        System.out.println(i);
        return; // 完全从 'demo2' 返回
    }
}

```

在 Scala 和 Kotlin 中，**demo** 和 **demo2** 都只会打印 0。但这并不更一致。Java 的方法与重构和类的使用保持一致——顶部代码中的 **return** 和下面的代码行为相同：

```

void demo3() {
    threeTimes(new MyIntConsumer());
}

class MyIntConsumer 实现 IntConsumer {
    public void accept(int i) {
        System.out.println(i);
        return;
    }
}

```

因此，Java 中的 **return** 与类方法和重构更为一致，但与 **for** 和 **while** 内置语句则不太一致，这些仍然是特殊的。

因此，以下两种写法在 Java 中是等价的：

```

IntStream.range(1, 4)
    .map(x -> x * x)
    .forEach(System.out::println);
IntStream.range(1, 4)
    .map(x -> { return x * x; })
    .forEach(System.out::println);

```

此外，在 Java 中使用 **try-with-resources** 是安全的：

```

class Resource 实现 AutoCloseable {
    public void close() { System.out.println("close()"); }
}

void executeAround(Consumer<Resource> f) {
    try (Resource r = new Resource()) {
        System.out.print("before ");
        f.accept(r);
        System.out.print("after ");
    }
}

void demo4() {
}

```

```

void demo() {
    threeTimes(i -> {
        System.out.println(i);
        return; // Return from lambda to threeTimes only!
    });
}

```

This can lead to unexpected behavior when attempting to write own language constructs, as in builtin constructs such as **for** loops **return** behaves differently:

```

void demo2() {
    for (int i = 0; i < 3; i++) {
        System.out.println(i);
        return; // Return from 'demo2' entirely
    }
}

```

In Scala and Kotlin, **demo** and **demo2** would both only print 0. But this is *not* more consistent. The Java approach is consistent with refactoring and the use of classes - the **return** in the code at the top, and the code below behaves the same:

```

void demo3() {
    threeTimes(new MyIntConsumer());
}

class MyIntConsumer implements IntConsumer {
    public void accept(int i) {
        System.out.println(i);
        return;
    }
}

```

Therefore, the Java **return** is more consistent with class methods and refactoring, but less with the **for** and **while** builtins, these remain special.

Because of this, the following two are equivalent in Java:

```

IntStream.range(1, 4)
    .map(x -> x * x)
    .forEach(System.out::println);
IntStream.range(1, 4)
    .map(x -> { return x * x; })
    .forEach(System.out::println);

```

Furthermore, the use of **try-with-resources** is safe in Java:

```

class Resource implements AutoCloseable {
    public void close() { System.out.println("close()"); }
}

void executeAround(Consumer<Resource> f) {
    try (Resource r = new Resource()) {
        System.out.print("before ");
        f.accept(r);
        System.out.print("after ");
    }
}

void demo4() {
}

```

```

executeAround(r -> {
    System.out.print("accept() ");
    return; // 不从 demo4 返回, 但释放资源。
});
}

```

将打印 before accept() after close()。在 Scala 和 Kotlin 语义中, try-with-resources 不会关闭, 但只会打印 before accept()。

第 73.11 节 : Lambda 和 Execute-around 模式

有几个使用 lambda 作为 FunctionalInterface 的简单场景的好例子。一个相当常见且可以通过 lambda 改进的用例是所谓的 Execute-Around 模式。在此模式中, 您有一组标准的设置/拆卸代码, 这些代码是围绕特定用例代码所需的多个场景。几个常见的例子包括文件 IO、数据库 IO、try/catch 块。

```

interface 数据处理器 {
    void 处理( 连接 connection ) throws SQLException;;
}

public void 执行处理( 数据处理器 processor ) throws SQLException{
    try (连接 connection = DBUtil.获取数据库连接();) {
        processor.处理(connection);
        connection.提交();
    }
}

```

然后用 lambda 调用此方法可能如下所示：

```

public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}

```

这不仅限于输入/输出操作。它可以应用于任何类似的设置/拆卸任务场景, 只要有细微的变化。该模式的主要好处是代码重用和强制执行DRY (不要重复自己) 原则。

第73.12节：使用lambda表达式和谓词从列表中获取某个值

从Java 8开始, 您可以使用lambda表达式和谓词。

示例：使用lambda表达式和谓词从列表中获取某个值。在此示例中, 每个人都会被打印出来, 并显示他们是否已满18岁。

Person类：

```

public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.年龄 = 年龄;
    }

    public int getAge() { return age; }
    public String getName() { return name; }
}

```

```

executeAround(r -> {
    System.out.print("accept() ");
    return; // Does not return from demo4, but frees the resource.
});
}

```

will print before accept() after close(). In the Scala and Kotlin semantics, the try-with-resources would not be closed, but it would print before accept() only.

Section 73.11: Lambdas and Execute-around Pattern

There are several good examples of using lambdas as a FunctionalInterface in simple scenarios. A fairly common use case that can be improved by lambdas is what is called the Execute-Around pattern. In this pattern, you have a set of standard setup/teardown code that is needed for multiple scenarios surrounding use case specific code. A few common example of this are file io, database io, try/catch blocks.

```

interface DataProcessor {
    void process( Connection connection ) throws SQLException;
}

public void doProcessing( DataProcessor processor ) throws SQLException{
    try (Connection connection = DBUtil.getDatabaseConnection();) {
        processor.process(connection);
        connection.commit();
    }
}

```

Then to call this method with a lambda it might look like:

```

public static void updateMyDAO(MyVO vo) throws DatabaseException {
    doProcessing((Connection conn) -> MyDAO.update(conn, ObjectMapper.map(vo)));
}

```

This is not limited to I/O operations. It can apply to any scenario where similar setup/tear down tasks are applicable with minor variations. The main benefit of this Pattern is code re-use and enforcing DRY (Don't Repeat Yourself).

Section 73.12: Using lambda expressions & predicates to get a certain value(s) from a list

Starting with Java 8, you can use lambda expressions & predicates.

Example: Use a lambda expressions & a predicate to get a certain value from a list. In this example every person will be printed out with the fact if they are 18 and older or not.

Person Class:

```

public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int getAge() { return age; }
    public String getName() { return name; }
}

```

```
}
```

java.util.function.Predicate 包中的内置接口 Predicate 是一个具有 boolean test(T t) 方法。

示例用法：

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class LambdaExample {
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<Person>();
        personList.add(new Person("Jeroen", 20));
        personList.add(new Person("Jack", 5));
        personList.add(new Person("Lisa", 19));

        print(personList, p -> p.getAge() >= 18);
    }

    private static void print(List<Person> personList, Predicate<Person> checker) {
        for (Person person : personList) {
            if (checker.test(person)) {
                System.out.print(person + " 符合你的表达式。");
            } else {
                System.out.println(person + " 不符合你的表达式。");
            }
        }
    }
}
```

print(personList, p -> p.getAge() >= 18); 方法接受一个lambda表达式 (因为Predicate被用作参数) , 你可以在其中定义所需的表达式。checker的test方法检查该表达式是否正确：checker.test(person)。

您可以轻松地将其更改为其他内容，例如print(personList, p -> p.getName().startsWith("J"));。这将检查该人的名字是否以“J”开头。

```
}
```

The built-in interface Predicate from the java.util.function.Predicate packages is a functional interface with a boolean test(T t) method.

Example Usage:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class LambdaExample {
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<Person>();
        personList.add(new Person("Jeroen", 20));
        personList.add(new Person("Jack", 5));
        personList.add(new Person("Lisa", 19));

        print(personList, p -> p.getAge() >= 18);
    }

    private static void print(List<Person> personList, Predicate<Person> checker) {
        for (Person person : personList) {
            if (checker.test(person)) {
                System.out.print(person + " matches your expression.");
            } else {
                System.out.println(person + " doesn't match your expression.");
            }
        }
    }
}
```

The print(personList, p -> p.getAge() >= 18); method takes a lambda expression (because the Predicate is used a parameter) where you can define the expression that is needed. The checker's test method checks if this expression is correct or not: checker.test(person).

You can easily change this to something else, for example to print(personList, p -> p.getName().startsWith("J"));. This will check if the person's name starts with a "J".

第74章：基本控制结构

第74.1节：switch语句

switch语句是Java的多路分支语句。它用于替代冗长的**if-else if-else**链，使代码更易读。然而，与**if**语句不同，不能使用不等式；每个值必须具体定义。

switch语句有三个关键组成部分：

- **case**：这是用于与witch语句的参数进行等值比较的值。
- **default**：这是一个可选的兜底表达式，如果没有任何case语句的条件为true，则执行该表达式。
- case语句的突然完成；通常是**break**：这是防止继续执行后续case语句的必要操作。

除**continue**外，可以使用任何会导致语句突然完成的语句。这包括：

- **break**
- **return**
- **throw**

在下面的示例中，一个典型的 switch语句包含四种可能的情况，包括 **default**。

```
Scanner scan = new Scanner(System.in);
int i = scan.nextInt();
switch (i) {
    case 0:
        System.out.println("i 是零");
        break;
    case 1:
        System.out.println("i 是一");
        break;
    case 2:
        System.out.println("i 是二");
        break;
    default:
        System.out.println("i 小于零或大于二");
}
```

通过省略 **break**或任何会导致突然结束的语句，我们可以利用所谓的“贯穿（fall-through）”情况，这些情况会针对多个值进行评估。这可以用来创建一个值成功匹配的范围，但仍然不如不等式灵活。

```
Scanner scan = new Scanner(System.in);
int foo = scan.nextInt();
switch(foo) {
    case 1:
        System.out.println("我等于或大于一");
    case 2:
    case 3:
        System.out.println("我是1、2或3");
        break;
    default:
        System.out.println("我既不是1，也不是2，或3");
}
```

Chapter 74: Basic Control Structures

Section 74.1: Switch statement

The **switch** statement is Java's multi-way branch statement. It is used to take the place of long **if-else if-else** chains, and make them more readable. However, unlike **if** statements, one may not use inequalities; each value must be concretely defined.

There are three critical components to the **switch** statement:

- **case**: This is the value that is evaluated for equivalence with the argument to the **switch** statement.
- **default**: This is an optional, catch-all expression, should none of the **case** statements evaluate to **true**.
- Abrupt completion of the **case** statement; usually **break**: This is required to prevent the undesired evaluation of further **case** statements.

With the exception of **continue**, it is possible to use any statement which would cause the [abrupt completion of a statement](#). This includes:

- **break**
- **return**
- **throw**

In the example below, a typical **switch** statement is written with four possible cases, including **default**.

```
Scanner scan = new Scanner(System.in);
int i = scan.nextInt();
switch (i) {
    case 0:
        System.out.println("i is zero");
        break;
    case 1:
        System.out.println("i is one");
        break;
    case 2:
        System.out.println("i is two");
        break;
    default:
        System.out.println("i is less than zero or greater than two");
}
```

By omitting **break** or any statement which would an abrupt completion, we can leverage what are known as "fall-through" cases, which evaluate against several values. This can be used to create ranges for a value to be successful against, but is still not as flexible as inequalities.

```
Scanner scan = new Scanner(System.in);
int foo = scan.nextInt();
switch(foo) {
    case 1:
        System.out.println("I'm equal or greater than one");
    case 2:
    case 3:
        System.out.println("I'm one, two, or three");
        break;
    default:
        System.out.println("I'm not either one, two, or three");
}
```

当`foo == 1` 时，输出将是：

我等于或大于一
我是1、2或3

当`foo == 3` 时，输出将是：

我是1、2或3

版本 \geq Java SE 5

switch语句也可以与枚举一起使用。

```
enum Option {  
    蓝色药丸,  
    红色药丸  
}  
  
public void 选择一颗(Option 选项) {  
    switch(选项) {  
        case 蓝色药丸:  
            System.out.println("故事结束，醒来，相信你想相信的。");  
            break;  
        case 红色药丸:  
            System.out.println("我会向你展示兔子洞有多深。");  
            break;  
    }  
}
```

版本 \geq Java SE 7

switch 语句也可以用于String类型。

```
public void 押韵游戏(String 短语) {  
    switch (短语) {  
        case "苹果和梨":  
            System.out.println("楼梯");  
            break;  
        case "lorry":  
            System.out.println("truck");  
            break;  
        default:  
            System.out.println("Don't know any more");  
    }  
}
```

第74.2节：do...while循环

do...while循环与其他循环的不同之处在于它保证至少执行一次。它也被称为“后测试循环”结构，因为条件语句是在主循环体之后执行的。

```
int i = 0;  
do {  
    i++;  
    System.out.println(i);  
} while (i < 100); // 条件在循环内容执行后检查。
```

In case of `foo == 1` the output will be:

I'm equal or greater than one
I'm one, two, or three

In case of `foo == 3` the output will be:

I'm one, two, or three

Version \geq Java SE 5

The switch statement can also be used with enums.

```
enum Option {  
    BLUE_PILL,  
    RED_PILL  
}  
  
public void takeOne(Option option) {  
    switch(option) {  
        case BLUE_PILL:  
            System.out.println("Story ends, wake up, believe whatever you want.");  
            break;  
        case RED_PILL:  
            System.out.println("I show you how deep the rabbit hole goes.");  
            break;  
    }  
}
```

Version \geq Java SE 7

The switch statement can also be used with Strings.

```
public void rhymingGame(String phrase) {  
    switch (phrase) {  
        case "apples and pears":  
            System.out.println("Stairs");  
            break;  
        case "lorry":  
            System.out.println("truck");  
            break;  
        default:  
            System.out.println("Don't know any more");  
    }  
}
```

Section 74.2: do...while Loop

The `do...while` loop differs from other loops in that it is guaranteed to execute **at least once**. It is also called the “post-test loop” structure because the conditional statement is performed after the main loop body.

```
int i = 0;  
do {  
    i++;  
    System.out.println(i);  
} while (i < 100); // Condition gets checked AFTER the content of the loop executes.
```

在这个例子中，循环将一直运行直到数字100被打印（尽管条件是 `i < 100` 而不是 `i <= 100`），因为循环条件是在循环执行之后进行评估的。

在至少执行一次的保证下，可以在循环外声明变量，并在循环内初始化它们。

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

在这种情况下，`theWord` 在循环外定义，但由于其自然流程保证了有值，`theWord` 将被初始化。

第74.3节：增强型for循环

版本 ≥ Java SE 5

从Java 5开始，可以使用for-each循环，也称为增强型for循环：

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");

(String string : strings) { System.out.println(string); }
```

for-each 循环可用于遍历数组和实现了 `Iterable` 接口的类，后者包括集合类，如 `List` 或 `Set`。

循环变量可以是任何可从源类型赋值的类型。

增强型 for 循环中，针对 `Iterable<T>` 或 `T[]` 的循环变量可以是类型 S，如果

- `T extends S`
- 当 T 和 S 都是原始类型且无需强制转换即可赋值时
- S 是原始类型，且 T 在拆箱转换后可以转换为可赋值给 S 的类型。
- T 是原始类型，且可以通过装箱转换转换为 S。

示例：

```
T elements = ...
for (S s : elements) {
```

T	S	编译
<code>int[]</code>	长整型	是
<code>long[]</code>	整型	否
可迭代的<字节>	长整型	是
可迭代的<字符串>	字符序列	是
可迭代的<字符序列>	字符串	否
<code>整型[]</code>	长整型	否

In this example, the loop will run until the number 100 is printed (even though the condition is `i < 100` and not `i <= 100`), because the loop condition is evaluated *after* the loop executes.

With the guarantee of at least one execution, it is possible to declare variables outside of the loop and initialize them inside.

```
String theWord;
Scanner scan = new Scanner(System.in);
do {
    theWord = scan.nextLine();
} while (!theWord.equals("Bird"));

System.out.println(theWord);
```

In this context, `theWord` is defined outside of the loop, but since it's guaranteed to have a value based on its natural flow, `theWord` will be initialized.

Section 74.3: For Each

Version ≥ Java SE 5

With Java 5 and up, one can use for-each loops, also known as enhanced for-loops:

```
List strings = new ArrayList();

strings.add("This");
strings.add("is");
strings.add("a for-each loop");

(String string : strings) { System.out.println(string); }
```

For each loops can be used to iterate over Arrays and implementations of the `Iterable` interface, the latter includes Collections classes, such as `List` or `Set`.

The loop variable can be of any type that is assignable from the source type.

The loop variable for a enhanced for loop for `Iterable<T>` or `T[]` can be of type S, if

- `T extends S`
- both T and S are primitive types and assignable without a cast
- S is a primitive type and T can be converted to a type assignable to S after unboxing conversion.
- T is a primitive type and can be converted to S by autoboxing conversion.

Examples:

```
T elements = ...
for (S s : elements) {
```

T	S	Compiles
<code>int[]</code>	<code>long</code>	yes
<code>long[]</code>	<code>int</code>	no
<code>Iterable<Byte></code>	<code>long</code>	yes
<code>Iterable<String></code>	<code>CharSequence</code>	yes
<code>Iterable<CharSequence></code>	<code>String</code>	no
<code>int[]</code>	<code>Long</code>	no

第74.4节：Java中的continue语句

continue语句用于跳过当前迭代中剩余的步骤，并开始下一次循环迭代。控制权从continue语句转移到步进值（递增或递减），如果有的话。

```
String[] programmers = {"Adrian", "Paul", "John", "Harry"};

//john不会被打印出来
for (String name : programmers) {
    if (name.equals("John"))
        continue;
    System.out.println(name);
}
```

continue语句也可以使程序控制转移到具名循环的步进值（如果有的话）：

```
Outer: // 最外层循环的名称为'Outer'
for(int i = 0; i < 5; )
{
    for(int j = 0; j < 5; j++)
    {
        continue Outer;
    }
}
```

第74.5节：If / Else If / Else 控制

```
if (i < 2) {
    System.out.println("i 小于 2");
} else if (i > 2) {
    System.out.println("i 大于 2");
} else {
    System.out.println("i 既不小于 2，也不大于 2");
}
```

当 i 小于或等于 1 时，if 代码块才会执行。

只有在之前所有条件（包括之前的 else if 结构和父级 if 结构）都被判断为 false 时，才会检查 else if 条件。在本例中，只有当 i 大于或等于 2 时，才会检查 else if 条件。

如果结果为 true，则执行其代码块，之后的任何 else if 和 else 结构都会被跳过。

如果所有 if 和 else if 条件都未被判断为 true，则会执行最后的 else 代码块。

第74.6节：for循环

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
}
```

for循环的三个组成部分（由;分隔）是变量声明/初始化（这里是int i = 0）、条件（这里是i < 100）和递增语句（这里是i++）。变量声明只执行一次，就好像放在第一次运行时的{}内部一样。然后检查条件，如果为true，循环体将执行，如果

Section 74.4: Continue Statement in Java

The continue statement is used to skip the remaining steps in the current iteration and start with the next loop iteration. The control goes from the **continue** statement to the step value (increment or decrement), if any.

```
String[] programmers = {"Adrian", "Paul", "John", "Harry"};

//john is not printed out
for (String name : programmers) {
    if (name.equals("John"))
        continue;
    System.out.println(name);
}
```

The **continue** statement can also make the control of the program shift to the step value (if any) of a named loop:

```
Outer: // The name of the outermost loop is kept here as 'Outer'
for(int i = 0; i < 5; )
{
    for(int j = 0; j < 5; j++)
    {
        continue Outer;
    }
}
```

Section 74.5: If / Else If / Else Control

```
if (i < 2) {
    System.out.println("i is less than 2");
} else if (i > 2) {
    System.out.println("i is more than 2");
} else {
    System.out.println("i is not less than 2, and not more than 2");
}
```

The **if** block will only run when **i** is 1 or less.

The **else if** condition is checked only if all the conditions before it (in previous **else if** constructs, and the parent if constructs) have been tested to **false**. In this example, the **else if** condition will only be checked if **i** is greater than or equal to 2.

If its result is **true**, its block is run, and any **else if** and **else** constructs after it will be skipped.

If none of the **if** and **else if** conditions have been tested to **true**, the **else** block at the end will be run.

Section 74.6: For Loops

```
for (int i = 0; i < 100; i++) {
    System.out.println(i);
}
```

The three components of the **for** loop (separated by ;) are variable declaration/initialization (here **int i = 0**), the condition (here **i < 100**), and the increment statement (here **i++**). The variable declaration is done once as if placed just inside the {} on the first run. Then the condition is checked, if it is **true** the body of the loop will execute, if it is

false循环将停止。假设循环继续，循环体将执行，最后当达到}时，递增语句将在再次检查条件之前执行。

如果循环体只有一条语句，大括号是可选的（你可以用分号写在一行）。但建议始终使用大括号以避免误解和错误。

for循环的组成部分是可选的。如果你的业务逻辑中缺少某部分，可以在for循环中省略对应的部分。

```
int i = obj.getLastestValue(); // i的值从一个方法中获取

for ( ; i < 100; i++) { // 这里没有进行初始化
    System.out.println(i);
}
```

for (;;) { 体 }结构等同于while (true)循环。

嵌套for循环

任何包含另一个循环语句的循环语句称为嵌套循环。同样，包含多个内部循环的循环称为“嵌套for循环”。

```
for(;;){
    //外层循环语句
    for(;;){
        //内层循环语句
    }
    //外层循环语句
}
```

嵌套的for循环可以用来打印三角形形状的数字。

```
for(int i=9;i>0;i--){//外层循环
    System.out.println();
    for(int k=i;k>0;k--)//内层循环 -1
        System.out.print(" ");
    }
    for(int j=i;j<=9;j++)//内层循环 -2
        System.out.print(" "+j);
}
```

第74.7节：三元运算符

有时你需要检查一个条件并设置变量的值。

例如。

```
字符串 name;

如果 (A > B) {
    name = "比利";
} 否则 {
    name = "吉米";
}
```

这可以很容易地写成一行

false the loop will stop. Assuming the loop continues, the body will execute and finally when the } is reached the increment statement will execute just before the condition is checked again.

The curly braces are optional (you can one line with a semicolon) if the loop contains just one statement. But, it's always recommended to use braces to avoid misunderstandings and bugs.

The **for** loop components are optional. If your business logic contains one of these parts, you can omit the corresponding component from your **for** loop.

```
int i = obj.getLastestValue(); // i value is fetched from a method

for ( ; i < 100; i++) { // here initialization is not done
    System.out.println(i);
}
```

The **for (;;) { function-body }** structure is equal to a **while (true)** loop.

Nested For Loops

Any looping statement having another loop statement inside called nested loop. The same way for looping having more inner loop is called 'nested for loop'.

```
for(;;){
    //Outer Loop Statements
    for(;;){
        //Inner Loop Statements
    }
    //Outer Loop Statements
}
```

Nested for loop can be demonstrated to print triangle shaped numbers.

```
for(int i=9;i>0;i--)//Outer Loop
    System.out.println();
    for(int k=i;k>0;k--)//Inner Loop -1
        System.out.print(" ");
    }
    for(int j=i;j<=9;j++)//Inner Loop -2
        System.out.print(" "+j);
}
```

Section 74.7: Ternary Operator

Sometimes you have to check for a condition and set the value of a variable.

For ex.

```
String name;

if (A > B) {
    name = "Billy";
} else {
    name = "Jimmy";
}
```

This can be easily written in one line as

```
字符串 name = A > B ? "比利" : "吉米";
```

如果条件为真，变量的值被设置为条件后面的第一个值。如果条件为假，变量将被赋予第二个值。

第74.8节：Try ... Catch ... Finally

try { ... } catch (...) { ... } 控制结构用于处理异常。

```
字符串 age_input = "abc";
try {
    整数 age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("你可以投票！");
    } else {
        System.out.println("抱歉，你还不能投票。");
    }
} catch (NumberFormatException ex) {
    System.err.println("无效输入。" + age_input + " 不是有效的整数。");
}
```

这将打印：

```
无效输入。'abc' 不是有效的整数。
```

可以在catch之后添加一个finally子句。无论是否抛出异常，finally子句都会被执行。

```
try { ... } catch ( ... ) { ... } finally { ... }
```

```
字符串 age_input = "abc";
try {
    整数 age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("你可以投票！");
    } else {
        System.out.println("抱歉，你还不能投票。");
    }
} catch (NumberFormatException ex) {
    System.err.println("无效输入。" + age_input + " 不是有效的整数。");
} finally {
    System.out.println("即使抛出异常，这段代码也会被执行");
}
```

这将打印：

```
无效输入。'abc' 不是有效的整数。
即使抛出异常，此代码也会始终执行
```

第74.9节：break语句

break语句结束一个循环（如for、while）或switch语句的执行。

循环：

```
String name = A > B ? "Billy" : "Jimmy";
```

The value of the variable is set to the value immediately after the condition, if the condition is true. If the condition is false, the second value will be given to the variable.

Section 74.8: Try ... Catch ... Finally

The **try { ... } catch (...) { ... }** control structure is used for handling Exceptions.

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input. " + age_input + " is not a valid integer.");
}
```

This would print:

```
Invalid input. 'abc' is not a valid integer.
```

A **finally** clause can be added after the **catch**. The **finally** clause would always be executed, regardless of whether an exception was thrown.

```
try { ... } catch ( ... ) { ... } finally { ... }
```

```
String age_input = "abc";
try {
    int age = Integer.parseInt(age_input);
    if (age >= 18) {
        System.out.println("You can vote!");
    } else {
        System.out.println("Sorry, you can't vote yet.");
    }
} catch (NumberFormatException ex) {
    System.err.println("Invalid input. " + age_input + " is not a valid integer.");
} finally {
    System.out.println("This code will always be run, even if an exception is thrown");
}
```

This would print:

```
Invalid input. 'abc' is not a valid integer.
This code will always be run, even if an exception is thrown
```

Section 74.9: Break

The **break** statement ends a loop (like **for, while**) or the evaluation of a switch statement.

Loop:

```
while(true) {  
    if(someCondition == 5) {  
        break;  
    }  
}
```

示例中的循环将无限运行。但当 someCondition在某次执行时等于5，循环就会结束。

如果多个循环嵌套，使用break只会结束最内层的循环。

第74.10节：while循环

```
int i = 0;  
while (i < 100) { // 条件在循环体执行前被检查  
    System.out.println(i);  
    i++;  
}
```

当括号内的条件为真时，while循环会一直运行。这也被称为“前测试循环”结构，因为每次执行主循环体之前必须满足条件语句。

如果循环只包含一条语句，大括号是可选的，但一些编码风格规范更倾向于无论如何都使用大括号。

第74.11节：If / Else

```
int i = 2;  
if (i < 2) {  
    System.out.println("i 小于 2");  
} else {  
    System.out.println("i 大于 2");  
}
```

if语句根据括号中条件的结果有条件地执行代码。当括号中的条件为真时，它将进入由大括号{和}定义的if语句块。

从开括号到闭括号是if语句的作用域。

else块是可选的，可以省略。如果if语句为false，则执行else块；如果if语句为真，则不执行else块，因为此时if语句已经执行。

另见：三元if

第74.12节：嵌套的break / continue

可以通过使用标签语句来跳出 / 继续外层循环：

```
outerloop :  
for(...) {  
    innerloop :  
    for(...) {  
        if(condition1)  
            跳出 outerloop;  
  
        if(condition2)  
            继续 innerloop; // 等同于 :continue;
```

```
while(true) {  
    if(someCondition == 5) {  
        break;  
    }  
}
```

The loop in the example would run forever. But when someCondition equals 5 at some point of execution, then the loop ends.

If multiple loops are cascaded, only the most inner loop ends using **break**.

Section 74.10: While Loops

```
int i = 0;  
while (i < 100) { // condition gets checked BEFORE the loop body executes  
    System.out.println(i);  
    i++;  
}
```

A **while** loop runs as long as the condition inside the parentheses is **true**. This is also called the "pre-test loop" structure because the conditional statement must be met before the main loop body is performed every time.

The curly braces are optional if the loop contains just one statement, but some coding style conventions prefers having the braces regardless.

Section 74.11: If / Else

```
int i = 2;  
if (i < 2) {  
    System.out.println("i is less than 2");  
} else {  
    System.out.println("i is greater than 2");  
}
```

An **if** statement executes code conditionally depending on the result of the condition in parentheses. When condition in parentheses is true it will enter to the block of if statement which is defined by curly braces like { and }. opening bracket till the closing bracket is the scope of the if statement.

The **else** block is optional and can be omitted. It runs if the **if** statement is **false** and does not run if the **if** statement is true. Because in that case **if** statement executes.

See also: Ternary If

Section 74.12: Nested break / continue

It's possible to **break** / **continue** to an outer loop by using label statements:

```
outerloop:  
for(...) {  
    innerloop:  
    for(...) {  
        if(condition1)  
            break outerloop;  
  
        if(condition2)  
            continue innerloop; // equivalent to :continue;
```

```
    }  
}
```

标签在Java中没有其他用途。

```
    }  
}
```

There is no other use for labels in Java.

第75章：BufferedWriter

第75.1节：向文件写入一行文本

这段代码将字符串写入文件。关闭写入器非常重要，因此这一步放在了finally块中完成。

```
public void writeLineToFile(String str) throws IOException {
    File file = new File("file.txt");
    BufferedWriter bw = null;
    try {
        bw = new BufferedWriter(new FileWriter(file));
        bw.write(str);
    } 最终 {
        if (bw != null) {
            bw.close();
        }
    }
}
```

还要注意，`write(String s)`方法写入字符串后不会自动添加换行符。要添加换行符，请使用`newLine()`方法。

版本 ≥ Java SE 7

Java 7 引入了`java.nio.file`包，以及带资源的try语句：

```
public void writeLineToFile(String str) throws IOException {
    Path path = Paths.get("file.txt");
    try (BufferedWriter bw = Files.newBufferedWriter(path)) {
        bw.write(str);
    }
}
```

Chapter 75: BufferedWriter

Section 75.1: Write a line of text to File

This code writes the string to a file. It is important to close the writer, so this is done in a `finally` block.

```
public void writeLineToFile(String str) throws IOException {
    File file = new File("file.txt");
    BufferedWriter bw = null;
    try {
        bw = new BufferedWriter(new FileWriter(file));
        bw.write(str);
    } finally {
        if (bw != null) {
            bw.close();
        }
    }
}
```

Also note that `write(String s)` does not place newline character after string has been written. To put it use `newLine()` method.

Version ≥ Java SE 7

Java 7 adds the `java.nio.file` package, and try-with-resources:

```
public void writeLineToFile(String str) throws IOException {
    Path path = Paths.get("file.txt");
    try (BufferedWriter bw = Files.newBufferedWriter(path)) {
        bw.write(str);
    }
}
```

第76章：新的文件I/O

第76.1节：创建路径

Path类用于以编程方式表示文件系统中的路径（因此可以指向文件以及目录，甚至是不存在的路径）

路径可以使用辅助类Paths获取：

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
Path p3 = Paths.get("C:\\\\Users\\\\DentAr\\\\Documents\\\\HGTG.DOT");
Path p4 = Paths.get("/home", "arthur", "files", "diary.tex");
```

第76.2节：操作路径

连接两个路径

路径可以使用resolve()方法连接。传入的路径必须是部分路径，即不包含根元素的路径。

```
Path p5 = Paths.get("/home/");
Path p6 = Paths.get("arthur/files");
Path joined = p5.resolve(p6);
Path otherJoined = p5.resolve("ford/files");

joined.toString() == "/home/arthur/files"
otherJoined.toString() == "/home/ford/files"
```

规范化路径

路径可能包含元素.（表示当前目录）和..（表示父目录）。

当路径中使用.时，可以随时删除而不改变路径的目标，..可以与前面的元素一起删除。

使用Paths API，可以通过normalize()方法实现：

```
Path p7 = Paths.get("/home/./arthur/..//ford/files");
Path p8 = Paths.get("C:\\\\Users\\\\.\\\\..\\\\Program Files");

p7.normalize().toString() == "/home/ford/files"
p8.normalize().toString() == "C:\\Program Files"
```

第76.3节：获取路径信息

可以使用Path对象的方法获取路径信息：

- `toString()`返回路径的字符串表示

```
Path p1 = Paths.get("/var/www"); // p1.toString() 返回 "/var/www"
```

- `getFileName()`返回文件名（或更具体地说，路径的最后一个元素）

```
Path p1 = Paths.get("/var/www"); // p1.getFileName() 返回 "www"
```

Chapter 76: New File I/O

Section 76.1: Creating paths

The Path class is used to programmatically represent a path in the file system (and can therefore point to files as well as directories, even to non-existent ones)

A path can be obtained using the helper class Paths:

```
Path p1 = Paths.get("/var/www");
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));
Path p3 = Paths.get("C:\\\\Users\\\\DentAr\\\\Documents\\\\HGTG.DOT");
Path p4 = Paths.get("/home", "arthur", "files", "diary.tex");
```

Section 76.2: Manipulating paths

Joining Two Paths

Paths can be joined using the `resolve()` method. The path passed has to be a partial path, which is a path that doesn't include the root element.

```
Path p5 = Paths.get("/home/");
Path p6 = Paths.get("arthur/files");
Path joined = p5.resolve(p6);
Path otherJoined = p5.resolve("ford/files");

joined.toString() == "/home/arthur/files"
otherJoined.toString() == "/home/ford/files"
```

Normalizing a path

Paths may contain the elements . (which points to the directory you're currently in) and .. (which points to the parent directory).

When used in a path, . can be removed at any time without changing the path's destination, and .. can be removed together with the preceding element.

With the Paths API, this is done using the `.normalize()` method:

```
Path p7 = Paths.get("/home/./arthur/..//ford/files");
Path p8 = Paths.get("C:\\\\Users\\\\.\\\\..\\\\Program Files");

p7.normalize().toString() == "/home/ford/files"
p8.normalize().toString() == "C:\\Program Files"
```

Section 76.3: Retrieving information about a path

Information about a path can be get using the methods of a Path object:

- `toString()` returns the string representation of the path

```
Path p1 = Paths.get("/var/www"); // p1.toString() returns "/var/www"
```

- `getFileName()` returns the file name (or, more specifically, the last element of the path)

```
Path p1 = Paths.get("/var/www"); // p1.getFileName() returns "www"
```

```
Path p3 = Paths.get("C:\\\\Users\\\\DentAr\\\\Documents\\\\HHGTDG.odt"); // p3.getFileName() 返回  
"HHGTDG.odt"
```

- getNameCount() 返回构成路径的元素数量

```
Path p1 = Paths.get("/var/www"); // p1.getNameCount() 返回 2
```

- getName(int index) 返回给定索引处的元素

```
Path p1 = Paths.get("/var/www"); // p1.getName(0) 返回 "var", p1.getName(1) 返回 "www"
```

- getParent() 返回父目录的路径

```
Path p1 = Paths.get("/var/www"); // p1.getParent().toString() 返回 "/var"
```

- getRoot() 返回路径的根目录

```
Path p1 = Paths.get("/var/www"); // p1.getRoot().toString() 返回 "/"  
Path p3 = Paths.get("C:\\\\Users\\\\DentAr\\\\Documents\\\\HHGTDG.odt"); // p3.getRoot().toString()  
返回 "C:\\\"
```

```
Path p3 = Paths.get("C:\\\\Users\\\\DentAr\\\\Documents\\\\HHGTDG.odt"); // p3.getFileName() returns  
"HHGTDG.odt"
```

- getNameCount() returns the number of elements that form the path

```
Path p1 = Paths.get("/var/www"); // p1.getNameCount() returns 2
```

- getName(int index) returns the element at the given index

```
Path p1 = Paths.get("/var/www"); // p1.getName(0) returns "var", p1.getName(1) returns "www"
```

- getParent() returns the path of the parent directory

```
Path p1 = Paths.get("/var/www"); // p1.getParent().toString() returns "/var"
```

- getRoot() returns the root of the path

```
Path p1 = Paths.get("/var/www"); // p1.getRoot().toString() returns "/"  
Path p3 = Paths.get("C:\\\\Users\\\\DentAr\\\\Documents\\\\HHGTDG.odt"); // p3.getRoot().toString()  
returns "C:\\\"
```

第76.4节：使用文件系统检索信息

要与文件系统交互，您需要使用类Files的方法。

检查存在性

要检查路径指向的文件或目录是否存在，您可以使用以下方法：

```
Files.exists(Path path)
```

和

```
Files.notExists(Path path)
```

`!Files.exists(path)` 不一定等于 `Files.notExists(path)`，因为存在三种可能的情况：

- 文件或目录存在被确认（此时 `exists` 返回 `true`, `notExists` 返回 `false`）
- 文件或目录不存在被确认（此时 `exists` 返回 `false`, `notExists` 返回 `true`）
- 文件或目录的存在性和不存在性均无法确认（例如由于访问限制）：`exists` 和 `notExists` 都返回 `false`。

检查路径指向的是文件还是目录

这可以通过 `Files.isDirectory(Path path)` 和 `Files.isRegularFile(Path path)` 来完成

```
Path p1 = Paths.get("/var/www");  
Path p2 = Paths.get("/home/testuser/File.txt");  
  
Files.isDirectory(p1) == true  
Files.isRegularFile(p1) == false  
  
Files.isDirectory(p2) == false
```

Section 76.4: Retrieving information using the filesystem

To interact with the filesystem you use the methods of the class `Files`.

Checking existence

To check the existence of the file or directory a path points to, you use the following methods:

```
Files.exists(Path path)
```

and

```
Files.notExists(Path path)
```

`!Files.exists(path)` does not necessarily have to be equal to `Files.notExists(path)`, because there are three possible scenarios:

- A file's or directory's existence is verified (`exists` returns `true` and `notExists` returns `false` in this case)
- A file's or directory's nonexistence is verified (`exists` returns `false` and `notExists` returns `true`)
- Neither the existence nor the nonexistence of a file or a directory can be verified (for example due to access restrictions): Both `exists` and `nonExists` return `false`.

Checking whether a path points to a file or a directory

This is done using `Files.isDirectory(Path path)` and `Files.isRegularFile(Path path)`

```
Path p1 = Paths.get("/var/www");  
Path p2 = Paths.get("/home/testuser/File.txt");  
  
Files.isDirectory(p1) == true  
Files.isRegularFile(p1) == false  
  
Files.isDirectory(p2) == false
```

```
Files.isRegularFile(p2) == true
```

获取属性

这可以通过以下方法完成：

```
Files.isReadable(Path path)  
Files.isWritable(Path path)  
Files.isExecutable(Path path)
```

```
Files.isHidden(Path path)  
Files.isSymbolicLink(Path path)
```

获取 MIME 类型

```
Files.probeContentType(Path path)
```

这尝试获取文件的 MIME 类型。它返回一个 MIME 类型字符串，如下所示：

- text/plain 表示文本文件
- text/html 表示 HTML 页面
- application/pdf 表示 PDF 文件
- image/png 表示 PNG 文件

第76.5节：读取文件

可以使用Files类按字节和按行读取文件。

```
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));  
byte[] content = Files.readAllBytes(p2);  
List<String> linesOfContent = Files.readAllLines(p2);
```

Files.readAllLines() 可选地接受一个字符集作为参数（默认是StandardCharsets.UTF_8）：

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

第76.6节：写文件

可以使用Files类按字节和按行写文件

```
Path p2 = Paths.get("/home/testuser/File.txt");  
List<String> lines = Arrays.asList(  
    new String[]{"第一行", "第二行", "第三行"});  
  
Files.write(p2, lines);  
  
Files.write(Path path, byte[] bytes)
```

现有文件将被覆盖，不存在的文件将被创建。

```
Files.isRegularFile(p2) == true
```

Getting properties

This can be done using the following methods:

```
Files.isReadable(Path path)  
Files.isWritable(Path path)  
Files.isExecutable(Path path)
```

```
Files.isHidden(Path path)  
Files.isSymbolicLink(Path path)
```

Getting MIME type

```
Files.probeContentType(Path path)
```

This tries to get the MIME type of a file. It returns a MIME type String, like this:

- text/plain for text files
- text/html for HTML pages
- application/pdf for PDF files
- image/png for PNG files

Section 76.5: Reading files

Files can be read byte- and line-wise using the Files class.

```
Path p2 = Paths.get(URI.create("file:///home/testuser/File.txt"));  
byte[] content = Files.readAllBytes(p2);  
List<String> linesOfContent = Files.readAllLines(p2);
```

Files.readAllLines() optionally takes a charset as parameter (default is StandardCharsets.UTF_8):

```
List<String> linesOfContent = Files.readAllLines(p2, StandardCharsets.ISO_8859_1);
```

Section 76.6: Writing files

Files can be written bite- and line-wise using the Files class

```
Path p2 = Paths.get("/home/testuser/File.txt");  
List<String> lines = Arrays.asList(  
    new String[]{"First line", "Second line", "Third line"});  
  
Files.write(p2, lines);  
  
Files.write(Path path, byte[] bytes)
```

Existing files will be overridden, non-existing files will be created.

第77章：文件输入输出

[Java输入输出](#) (Input and Output) 用于处理输入并生成输出。Java使用流的概念来加快输入输出操作。java.io包包含所有进行输入输出操作所需的类。文件处理也通过Java输入输出API在Java中完成。

第77.1节：从java.io.File迁移到Java 7 NIO (java.nio.file.Path)

这些示例假设你已经大致了解Java 7的NIO，并且习惯使用java.io.File编写代码。使用这些示例作为快速查找更多以NIO为中心的迁移文档的手段。

[Java 7的NIO还有更多内容，例如内存映射文件或使用FileSystem打开ZIP或JAR文件。](#)

这些示例只涵盖有限数量的基本用例。

作为基本规则，如果你习惯使用[java.io.File](#)实例方法执行文件系统读写操作，你会发现它在[java.nio.file.Files](#)中作为静态方法存在。

指向路径

```
// -> IO  
File file = new File("io.txt");  
  
// -> NIO  
Path path = Paths.get("nio.txt");
```

相对于另一路径的路径

```
// 即使在Windows操作系统上，也可以使用正斜杠代替反斜杠  
// -> IO  
File folder = new File("C:/");  
File fileInFolder = new File(folder, "io.txt");  
  
// -> NIO  
Path directory = Paths.get("C:/");  
Path pathInDirectory = directory.resolve("nio.txt");
```

将File转换为Path或从Path转换为File以供库使用

```
// -> IO到NIO  
Path pathFromFile = new File("io.txt").toPath();  
  
// -> NIO 转 IO  
File fileFromPath = Paths.get("nio.txt").toFile();
```

检查文件是否存在，若存在则删除

```
// -> IO  
if (file.exists()) {  
    boolean deleted = file.delete();  
    if (!deleted) {  
        throw new IOException("无法删除文件");  
    }  
}  
  
// -> NIO  
Files.deleteIfExists(path);
```

通过 OutputStream 写入文件

使用 NIO 写入和读取文件有多种方式，适用于不同的性能和内存限制、可读性及使用场景，例如 [FileChannel](#)、[Files.write\(Path path, byte\\[\\] bytes, OpenOption...\)](#)

Chapter 77: File I/O

[Java I/O](#) (Input and Output) is used to process the input and produce the output. Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations. [Handling files](#) is also done in java by Java I/O API.

Section 77.1: Migrating from java.io.File to Java 7 NIO (java.nio.file.Path)

These examples assume that you already know what Java 7's NIO is in general, and you are used to writing code using [java.io.File](#). Use these examples as a means to quickly find more NIO-centric documentation for migrating.

There is much more to Java 7's NIO such as [memory-mapped files](#) or [opening a ZIP or JAR file using FileSystem](#). These examples will only cover a limited number of basic use cases.

As a basic rule, if you are used to perform a file system read/write operation using a [java.io.File](#) instance method, you will find it as a static method within [java.nio.file.Files](#).

Point to a path

```
// -> IO  
File file = new File("io.txt");  
  
// -> NIO  
Path path = Paths.get("nio.txt");
```

Paths relative to another path

```
// Forward slashes can be used in place of backslashes even on a Windows operating system  
// -> IO  
File folder = new File("C:/");  
File fileInFolder = new File(folder, "io.txt");  
  
// -> NIO  
Path directory = Paths.get("C:/");  
Path pathInDirectory = directory.resolve("nio.txt");
```

Converting File from/to Path for use with libraries

```
// -> IO to NIO  
Path pathFromFile = new File("io.txt").toPath();  
  
// -> NIO to IO  
File fileFromPath = Paths.get("nio.txt").toFile();
```

Check if the file exists and delete it if it does

```
// -> IO  
if (file.exists()) {  
    boolean deleted = file.delete();  
    if (!deleted) {  
        throw new IOException("Unable to delete file");  
    }  
}  
  
// -> NIO  
Files.deleteIfExists(path);
```

Write to a file via an OutputStream

There are several ways to write and read from a file using NIO for different performance and memory constraints, readability and use cases, such as [FileChannel](#), [Files.write\(Path path, byte\\[\\] bytes, OpenOption...\)](#)

options)... 本例仅涉及 OutputStream，但强烈建议学习内存映射文件以及 java.nio.file.Files 中提供的各种静态方法。

```
List<String> lines = Arrays.asList(
    String.valueOf(Calendar.getInstance().getTimeInMillis()),
    "line one",
    "line two");

// -> IO
if (file.exists()) {
    // 注意：非原子操作
    throw new IOException("文件已存在");
}

try (FileOutputStream outputStream = new FileOutputStream(file)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}

// -> NIO
try (OutputStream outputStream = Files.newOutputStream(path, StandardOpenOption.CREATE_NEW)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}
```

遍历文件夹中的每个文件

```
// -> IO
for (File selectedFile : folder.listFiles()) {
    // 注意：根据目录中文件的数量，folder.listFiles() 可能需要较长时间才能返回

    System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
}

// -> NIO
Files.walkFileTree(directory, EnumSet.noneOf(FileVisitOption.class), 1, new
SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});
```

递归文件夹遍历

```
// -> IO
recurseFolder(folder);

// -> NIO
// 注意：除非明确作为参数传递给 Files.walkFileTree，否则不会跟随符号链接

Files.walkFileTree(目录, new SimpleFileVisitor<Path>() {
    @Override
```

options)... In this example, only `OutputStream` is covered, but you are strongly encouraged to learn about memory-mapped files and the various static methods available in `java.nio.file.Files`.

```
List<String> lines = Arrays.asList(
    String.valueOf(Calendar.getInstance().getTimeInMillis()),
    "line one",
    "line two");

// -> IO
if (file.exists()) {
    // Note: Not atomic
    throw new IOException("File already exists");
}

try (FileOutputStream outputStream = new FileOutputStream(file)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}

// -> NIO
try (OutputStream outputStream = Files.newOutputStream(path, StandardOpenOption.CREATE_NEW)) {
    for (String line : lines) {
        outputStream.write((line + System.lineSeparator()).getBytes(StandardCharsets.UTF_8));
    }
}
```

Iterating on each file within a folder

```
// -> IO
for (File selectedFile : folder.listFiles()) {
    // Note: Depending on the number of files in the directory folder.listFiles() may take a long
time to return
    System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
}
```

```
// -> NIO
Files.walkFileTree(directory, EnumSet.noneOf(FileVisitOption.class), 1, new
SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult preVisitDirectory(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("d " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
        System.out.println("f " + selectedPath.toAbsolutePath());
        return FileVisitResult.CONTINUE;
    }
});
```

```
@Override
public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
    System.out.println("f " + selectedPath.toAbsolutePath());
    return FileVisitResult.CONTINUE;
});
```

Recursive folder iteration

```
// -> IO
recurseFolder(folder);

// -> NIO
// Note: Symbolic links are NOT followed unless explicitly passed as an argument to
Files.walkFileTree
Files.walkFileTree(directory, new SimpleFileVisitor<Path>() {
    @Override
```

```

    public FileVisitResult preVisitDirectory(Path 目录, BasicFileAttributes 属性) throws
IOException {
    System.out.println("d " + selectedPath.toAbsolutePath());
    return FileVisitResult.CONTINUE;
}

@Override
public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
    System.out.println("f " + selectedPath.toAbsolutePath());
    return FileVisitResult.CONTINUE;
}
});

```

```

private static void 递归文件夹(File 文件夹) {
    for (File 选中文件 : 文件夹.listFiles()) {
        System.out.println((选中文件.isDirectory() ? "d" : "f") + " " +
选中文件.getAbsolutePath());
        if (选中文件.isDirectory()) {
            // 注意：符号链接会被跟随
            递归文件夹(选中文件);
        }
    }
}

```

第77.2节：从文件读取图像

```

import java.awt.Image;
import javax.imageio.ImageIO;

...

try {
    Image img = ImageIO.read(new File("~/Desktop/cat.png"));
} catch (IOException e) {
    e.printStackTrace();
}

```

第77.3节：使用 FileInputStream/FileOutputStream进行文件读写

写入文件 test.txt：

```

String filepath ="C:\\\\test.txt";
FileOutputStream fos = null;
try {
fos = new FileOutputStream(filepath);
byte[] buffer = "这将被写入 test.txt".getBytes();
fos.write(buffer, 0, buffer.length);
fos.close();
} catch (FileNotFoundException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
} finally{
    if(fos != null)
        fos.close();
}

```

```

public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws
IOException {
    System.out.println("d " + selectedPath.toAbsolutePath());
    return FileVisitResult.CONTINUE;
}

@Override
public FileVisitResult visitFile(Path selectedPath, BasicFileAttributes attrs) throws
IOException {
    System.out.println("f " + selectedPath.toAbsolutePath());
    return FileVisitResult.CONTINUE;
}
});

```

```

private static void recurseFolder(File folder) {
    for (File selectedFile : folder.listFiles()) {
        System.out.println((selectedFile.isDirectory() ? "d" : "f") + " " +
selectedFile.getAbsolutePath());
        if (selectedFile.isDirectory()) {
            // Note: Symbolic links are followed
            recurseFolder(selectedFile);
        }
    }
}

```

Section 77.2: Reading an image from a file

```

import java.awt.Image;
import javax.imageio.ImageIO;

...

try {
    Image img = ImageIO.read(new File("~/Desktop/cat.png"));
} catch (IOException e) {
    e.printStackTrace();
}

```

Section 77.3: File Read/Write Using FileInputStream/FileOutputStream

Write to a file test.txt:

```

String filepath ="C:\\\\test.txt";
FileOutputStream fos = null;
try {
    fos = new FileOutputStream(filepath);
    byte[] buffer = "This will be written in test.txt".getBytes();
    fos.write(buffer, 0, buffer.length);
    fos.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally{
    if(fos != null)
        fos.close();
}

```

从文件 test.txt 读取：

```
String filepath = "C:\\test.txt";
FileInputStream fis = null;
try {
    fis = new FileInputStream(filepath);
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
    fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally{
    if(fis != null)
        fis.close();
}
```

注意，自Java 1.7起，引入了try-with-resources语句，使得读写操作的实现更加简单：

写入文件 test.txt：

```
String filepath = "C:\\test.txt";
try (FileOutputStream fos = new FileOutputStream(filepath)){
    byte[] buffer = "This will be written in test.txt".getBytes();
    fos.write(buffer, 0, buffer.length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

从文件 test.txt 读取：

```
String filepath = "C:\\test.txt";
try (FileInputStream fis = new FileInputStream(filepath)){
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
    fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

第77.4节：将所有字节读取到byte[]中

Java 7引入了非常有用的Files类

```
版本 ≥ Java SE 7
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;

Path path = Paths.get("路径/到/文件");

try {
    byte[] data = Files.readAllBytes(path);
```

Read from file test.txt:

```
String filepath = "C:\\test.txt";
FileInputStream fis = null;
try {
    fis = new FileInputStream(filepath);
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
    fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally{
    if(fis != null)
        fis.close();
}
```

Note, that since Java 1.7 the try-with-resources statement was introduced what made implementation of reading\\writing operation much simpler:

Write to a file test.txt:

```
String filepath = "C:\\test.txt";
try (FileOutputStream fos = new FileOutputStream(filepath)){
    byte[] buffer = "This will be written in test.txt".getBytes();
    fos.write(buffer, 0, buffer.length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Read from file test.txt:

```
String filepath = "C:\\test.txt";
try (FileInputStream fis = new FileInputStream(filepath)){
    int length = (int) new File(filepath).length();
    byte[] buffer = new byte[length];
    fis.read(buffer, 0, length);
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Section 77.4: Reading all bytes to a byte[]

Java 7 introduced the very useful [Files](#) class

```
Version ≥ Java SE 7
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.Path;

Path path = Paths.get("path/to/file");

try {
    byte[] data = Files.readAllBytes(path);
```

```
} catch(IOException e) {  
e.printStackTrace();  
}
```

第77.5节：使用Channel复制文件

我们可以使用Channel更快地复制文件内容。为此，我们可以使用FileChannel的transferTo()方法。

```
import java.io.File;  
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.nio.channels.FileChannel;  
  
public class FileCopier {  
  
    public static void main(String[] args) {  
        File sourceFile = new File("hello.txt");  
        File sinkFile = new File("hello2.txt");  
        copy(sourceFile, sinkFile);  
    }  
  
    public static void copy(File sourceFile, File destFile) {  
        if (!sourceFile.exists() || !destFile.exists()) {  
            System.out.println("源文件或目标文件不存在");  
            return;  
        }  
  
        try (FileChannel srcChannel = new FileInputStream(sourceFile).getChannel();  
             FileOutputStream destFile = new FileOutputStream(destFile).getChannel()) {  
  
            srcChannel.transferTo(0, srcChannel.size(), sinkFile);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
} catch(IOException e) {  
e.printStackTrace();  
}
```

Section 77.5: Copying a file using Channel

We can use Channel to copy file content faster. To do so, we can use transferTo() method of FileChannel.

```
import java.io.File;  
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.nio.channels.FileChannel;  
  
public class FileCopier {  
  
    public static void main(String[] args) {  
        File sourceFile = new File("hello.txt");  
        File sinkFile = new File("hello2.txt");  
        copy(sourceFile, sinkFile);  
    }  
  
    public static void copy(File sourceFile, File destFile) {  
        if (!sourceFile.exists() || !destFile.exists()) {  
            System.out.println("Source or destination file doesn't exist");  
            return;  
        }  
  
        try (FileChannel srcChannel = new FileInputStream(sourceFile).getChannel();  
             FileChannel destFile = new FileOutputStream(destFile).getChannel()) {  
  
            srcChannel.transferTo(0, srcChannel.size(), sinkFile);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

第77.6节：将byte[]写入文件

版本 ≥ Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };  
  
try(FileOutputStream stream = new FileOutputStream("Hello world.txt")) {  
    stream.write(bytes);  
} catch (IOException ioe) {  
    // 处理I/O异常  
    ioe.printStackTrace();  
}
```

版本 < Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };  
  
FileOutputStream stream = null;  
try {  
    stream = new FileOutputStream("Hello world.txt");  
    stream.write(bytes);  
} catch (IOException ioe) {  
    // 处理I/O异常
```

Section 77.6: Writing a byte[] to a file

Version ≥ Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };  
  
try(FileOutputStream stream = new FileOutputStream("Hello world.txt")) {  
    stream.write(bytes);  
} catch (IOException ioe) {  
    // Handle I/O Exception  
    ioe.printStackTrace();  
}
```

Version < Java SE 7

```
byte[] bytes = { 0x48, 0x65, 0x6c, 0x6c, 0x6f };  
  
FileOutputStream stream = null;  
try {  
    stream = new FileOutputStream("Hello world.txt");  
    stream.write(bytes);  
} catch (IOException ioe) {  
    // Handle I/O Exception
```

```

ioe.printStackTrace();
} finally {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException ignored) {}
    }
}

```

大多数 java.io 文件 API 接受 String 和 File 作为参数，因此你也可以使用

```

File file = new File("Hello world.txt");
FileOutputStream stream = new FileOutputStream(file);

```

第 77.7 节：流 (Stream) 与写入器/读取器 (Writer/Reader) API

流提供对二进制内容的最直接访问，因此任何 [InputStream / OutputStream](#) 实现总是操作 int 和 byte 类型的数据。

```

// 从流中读取单个字节
int b = inputStream.read();
if (b >= 0) { // 负值表示流的结束，正常值范围是 0 - 255
    // 将字节写入另一个流
    outputStream.write(b);
}

// 读取一块数据
byte[] data = new byte[1024];
int nBytesRead = inputStream.read(data);
if (nBytesRead >= 0) { // 负值表示流结束
    // 将数据块写入另一个流
    outputStream.write(data, 0, nBytesRead);
}

```

有一些例外，最显著的可能是PrintStream，它增加了“方便打印各种数据值表示”的功能。这允许将System.out既用作二进制InputStream，也用作文本输出，使用诸如System.out.println()等方法。

此外，一些流实现作为更高级内容的接口，例如Java对象（参见序列化）或原生类型，如DataOutputStream / DataInputStream。

通过Writer和Reader类，Java还提供了显式字符流的API。虽然大多数应用会基于流实现这些功能，但字符流API不提供任何处理二进制内容的方法。

```

// 该示例使用平台默认字符集，详见下文
// 以获得更好的实现。

```

```

Writer writer = new OutputStreamWriter(System.out);
writer.write("Hello world!");

Reader reader = new InputStreamReader(System.in);
char singleCharacter = reader.read();

```

每当需要将字符编码为二进制数据时（例如使用InputStreamWriter / OutputStreamWriter类时），如果不想依赖平台默认字符集，应指定字符集。

```

ioe.printStackTrace();
} finally {
    if (stream != null) {
        try {
            stream.close();
        } catch (IOException ignored) {}
    }
}

```

Most java.io file APIs accept both [Strings](#) and [Files](#) as arguments, so you could as well use

```

File file = new File("Hello world.txt");
FileOutputStream stream = new FileOutputStream(file);

```

Section 77.7: Stream vs Writer/Reader API

Streams provide the most direct access to the binary content, so any [InputStream / OutputStream](#) implementations always operate on [ints](#) and [bytes](#).

```

// Read a single byte from the stream
int b = inputStream.read();
if (b >= 0) { // A negative value represents the end of the stream, normal values are in the range 0 - 255
    // Write the byte to another stream
    outputStream.write(b);
}

// Read a chunk
byte[] data = new byte[1024];
int nBytesRead = inputStream.read(data);
if (nBytesRead >= 0) { // A negative value represents end of stream
    // Write the chunk to another stream
    outputStream.write(data, 0, nBytesRead);
}

```

There are some exceptions, probably most notably the [PrintStream](#) which adds the "ability to print representations of various data values conveniently". This allows to use [System.out](#) both as a binary [InputStream](#) and as a textual output using methods such as [System.out.println\(\)](#).

Also, some stream implementations work as an interface to higher-level contents such as Java objects (see [Serialization](#)) or native types, e.g. [DataOutputStream / DataInputStream](#).

With the [Writer](#) and [Reader](#) classes, Java also provides an API for explicit character streams. Although most applications will base these implementations on streams, the character stream API does not expose any methods for binary content.

```

// This example uses the platform's default charset, see below
// for a better implementation.

```

```

Writer writer = new OutputStreamWriter(System.out);
writer.write("Hello world!");

```

```

Reader reader = new InputStreamReader(System.in);
char singleCharacter = reader.read();

```

Whenever it is necessary to encode characters into binary data (e.g. when using the [InputStreamWriter / OutputStreamWriter](#) classes), you should specify a charset if you do not want to depend on the platform's default

如果不确定，建议使用兼容Unicode的编码，例如所有Java平台都支持的UTF-8。因此，你可能应该避免使用像FileWriter和FileReader这样的类，因为它们总是使用默认的平台字符集。使用字符流访问文件的更好方法是：

```
Charset myCharset = StandardCharsets.UTF_8;

Writer writer = new OutputStreamWriter( new FileOutputStream("test.txt"), myCharset );
writer.write('Ä');
writer.flush();
writer.close();

Reader reader = new InputStreamReader( new FileInputStream("test.txt"), myCharset );
char someUnicodeCharacter = reader.read();
reader.close();
```

最常用的Reader之一是BufferedReader，它提供了从另一个reader读取整行文本的方法，可能是逐行读取字符流的最简单方式：

```
// 从baseReader中一次读取一行
BufferedReader reader = new BufferedReader( baseReader );
String line;
while((line = reader.readLine()) != null) {
    // 记住：System.out是一个流，不是writer！
    System.out.println(line);
}
```

第77.8节：使用Scanner读取文件

逐行读取文件

```
public class Main {

    public static void main(String[] args) {
        try {
Scanner scanner = new Scanner(new File("example.txt"));
        while(scanner.hasNextLine())
        {
            String line = scanner.nextLine();
            //执行操作
        }
    } catch (FileNotFoundException e) {
e.printStackTrace();
}
    }
}
```

逐词读取

```
public class Main {

    public static void main(String[] args) {
        try {
Scanner scanner = new Scanner(new File("example.txt"));
        while(scanner.hasNext())
        {
            String line = scanner.next();
            //执行操作
        }
    }
}
```

charset. When in doubt, use a Unicode-compatible encoding, e.g. UTF-8 which is supported on all Java platforms. Therefore, you should probably stay away from classes like `FileWriter` and `FileReader` as those always use the default platform charset. A better way to access files using character streams is this:

```
Charset myCharset = StandardCharsets.UTF_8;

Writer writer = new OutputStreamWriter( new FileOutputStream("test.txt"), myCharset );
writer.write('Ä');
writer.flush();
writer.close();

Reader reader = new InputStreamReader( new FileInputStream("test.txt"), myCharset );
char someUnicodeCharacter = reader.read();
reader.close();
```

One of the most commonly used Readers is `BufferedReader` which provides a method to read whole lines of text from another reader and is presumably the simplest way to read a character stream line by line:

```
// Read from baseReader, one line at a time
BufferedReader reader = new BufferedReader( baseReader );
String line;
while((line = reader.readLine()) != null) {
    // Remember: System.out is a stream, not a writer!
    System.out.println(line);
}
```

Section 77.8: Reading a file with a Scanner

Reading a file line by line

```
public class Main {

    public static void main(String[] args) {
        try {
Scanner scanner = new Scanner(new File("example.txt"));
        while(scanner.hasNextLine())
        {
            String line = scanner.nextLine();
            //do stuff
        }
    } catch (FileNotFoundException e) {
e.printStackTrace();
}
    }
}
```

word by word

```
public class Main {

    public static void main(String[] args) {
        try {
Scanner scanner = new Scanner(new File("example.txt"));
        while(scanner.hasNext())
        {
            String line = scanner.next();
            //do stuff
        }
    }
}
```

```

        } catch (FileNotFoundException e) {
e.printStackTrace();
}
}

```

你也可以使用 `scanner.useDelimeter()` 方法来更改分隔符

第77.9节：使用 InputStream 和 OutputStream 复制文件

我们可以使用循环直接将数据从源复制到数据接收端。在这个例子中，我们从一个 `InputStream` 读取数据，同时写入到一个 `OutputStream`。读取和写入完成后，必须关闭资源。

```

public void copy(InputStream source, OutputStream destination) throws IOException {
    try {
        int c;
        while ((c = source.read()) != -1) {
            destination.write(c);
        }
    } 最终 {
        if (source != null) {
            source.close();
        }
        if (destination != null) {
            destination.close();
        }
    }
}

```

```

        } catch (FileNotFoundException e) {
e.printStackTrace();
}
}

```

and you can also change the delimeter by using `scanner.useDelimeter()` method

Section 77.9: Copying a file using InputStream and OutputStream

We can directly copy data from a source to a data sink using a loop. In this example, we are reading data from an `InputStream` and at the same time, writing to an `OutputStream`. Once we are done reading and writing, we have to close the resource.

```

public void copy(InputStream source, OutputStream destination) throws IOException {
    try {
        int c;
        while ((c = source.read()) != -1) {
            destination.write(c);
        }
    } finally {
        if (source != null) {
            source.close();
        }
        if (destination != null) {
            destination.close();
        }
    }
}

```

第77.10节：从二进制文件读取

你可以使用这段代码在所有最新版本的 Java 中读取二进制文件：

版本 ≥ Java SE 1.4

```

File file = new File("path_to_the_file");
byte[] data = new byte[(int) file.length()];
DataInputStream stream = new DataInputStream(new FileInputStream(file));
stream.readFully(data);
stream.close();

```

如果您使用的是 Java 7 或更高版本，可以使用更简单的 nio API 方法：

版本 ≥ Java SE 7

```

Path path = Paths.get("文件路径");
byte[] data = Files.readAllBytes(path);

```

第 77.11 节：使用 Channel 和缓冲区读取文件

Channel 使用 Buffer 来读写数据。缓冲区是一个固定大小的容器，我们可以一次写入一块数据。Channel 比基于流的 I/O 要快得多。

要使用 Channel 从文件读取数据，需要以下步骤——

1. 我们需要一个 `FileInputStream` 的实例。 `FileInputStream` 有一个名为 `getChannel()` 的方法，返回一个 `Channel`。

Section 77.10: Reading from a binary file

You can read an a binary file using this piece of code in all recent versions of Java:

Version ≥ Java SE 1.4

```

File file = new File("path_to_the_file");
byte[] data = new byte[(int) file.length()];
DataInputStream stream = new DataInputStream(new FileInputStream(file));
stream.readFully(data);
stream.close();

```

If you are using Java 7 or later, there is a simpler way using the nio API:

Version ≥ Java SE 7

```

Path path = Paths.get("path_to_the_file");
byte[] data = Files.readAllBytes(path);

```

Section 77.11: Reading a file using Channel and Buffer

Channel uses a Buffer to read/write data. A buffer is a fixed sized container where we can write a block of data at once. Channel is a quite faster than stream-based I/O.

To read data from a file using Channel we need to have the following steps-

1. We need an instance of `FileInputStream`. `FileInputStream` has a method named `getChannel()` which returns a `Channel`.

- 调用 `FileInputStream` 的 `getChannel()` 方法并获取 `Channel`。
- 创建一个 `ByteBuffer`。`ByteBuffer` 是一个固定大小的字节容器。
- 通道 (`Channel`) 有一个读取方法，我们必须提供一个 `ByteBuffer` 作为该读取方法的参数。
`ByteBuffer` 有两种模式——只读模式和只写模式。我们可以通过调用 `flip()` 方法来切换模式。缓冲区有位置 (`position`)、限制 (`limit`) 和容量 (`capacity`)。一旦缓冲区被创建且大小固定，其限制和容量与大小相同，位置从零开始。当缓冲区被写入数据时，位置会逐渐增加。切换模式意味着改变位置。要从缓冲区的开头读取数据，我们必须将位置设置为零。`flip()` 方法会改变位置。
- 当我们调用 `Channel` 的读取方法时，它会使用数据填充缓冲区。
- 如果我们需要从 `ByteBuffer` 读取数据，就需要翻转缓冲区，将其模式从只写切换为只读模式，然后继续从缓冲区读取数据。
- 当没有更多数据可读时，通道的 `read()` 方法会返回 0 或 -1。

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class FileChannelRead {

    public static void main(String[] args) {

        File inputFile = new File("hello.txt");

        if (!inputFile.exists()) {
            System.out.println("输入文件不存在。");
            return;
        }

        try {
            FileInputStream fis = new FileInputStream(inputFile);
            FileChannel fileChannel = fis.getChannel();
            ByteBuffer buffer = ByteBuffer.allocate(1024);

            while (fileChannel.read(buffer) > 0) {
                buffer.flip();
                while (buffer.hasRemaining()) {
                    byte b = buffer.get();
                    System.out.print((char) b);
                }
                buffer.clear();
            }

            fileChannel.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

第77.12节：添加目录

要从一个 `File` 实例创建新目录，需要使用两种方法之一：`mkdirs()` 或 `mkdir()`。

- `mkdir()` - 创建由此抽象路径名指定的目录。[\(source\)](#)
- `mkdirs()` - 创建由此抽象路径名指定的目录，包括任何必要但不存在的父目录。注意，如果此操作失败，可能已经成功创建了部分必要的目录

- Call the `getChannel()` method of `FileInputStream` and acquire `Channel`.
- Create a `ByteBuffer`. `ByteBuffer` is a fixed size container of bytes.
- Channel has a read method and we have to provide a `ByteBuffer` as an argument to this read method.
`ByteBuffer` has two modes - read-only mood and write-only mood. We can change the mode using `flip()` method call. Buffer has a position, limit, and capacity. Once a buffer is created with a fixed size, its limit and capacity are the same as the size and the position starts from zero. While a buffer is written with data, its position gradually increases. Changing mode means, changing the position. To read data from the beginning of a buffer, we have to set the position to zero. `flip()` method change the position
- When we call the read method of the `Channel`, it fills up the buffer using data.
- If we need to read the data from the `ByteBuffer`, we need to flip the buffer to change its mode to write-only to read-only mode and then keep reading data from the buffer.
- When there is no longer data to read, the `read()` method of channel returns 0 or -1.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public class FileChannelRead {

    public static void main(String[] args) {

        File inputFile = new File("hello.txt");

        if (!inputFile.exists()) {
            System.out.println("The input file doesn't exist.");
            return;
        }

        try {
            FileInputStream fis = new FileInputStream(inputFile);
            FileChannel fileChannel = fis.getChannel();
            ByteBuffer buffer = ByteBuffer.allocate(1024);

            while (fileChannel.read(buffer) > 0) {
                buffer.flip();
                while (buffer.hasRemaining()) {
                    byte b = buffer.get();
                    System.out.print((char) b);
                }
                buffer.clear();
            }

            fileChannel.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Section 77.12: Adding Directories

To make a new directory from a `File` instance you would need to use one of two methods: `mkdirs()` or `mkdir()`.

- `mkdir()` - Creates the directory named by this abstract pathname. [\(source\)](#)
- `mkdirs()` - Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary

父目录。 (source) [_____](#)

注意：createNewFile()只会创建新文件，不会创建新目录。

```
File singleDir = new File("C:/Users/SomeUser/Desktop/新建文件夹/");  
File multiDir = new File("C:/Users/SomeUser/Desktop/新建文件夹 2/另一个文件夹/");  
  
// 假设“新建文件夹”和“新建文件夹 2”都不存在  
  
singleDir.createNewFile(); // 会创建一个名为“新建文件夹.file”的新文件  
singleDir.mkdir(); // 会创建该目录  
singleDir.mkdirs(); // 会创建该目录  
  
multiDir.createNewFile(); // 会抛出 IOException  
multiDir.mkdir(); // 不会生效  
multiDir.mkdirs(); // 会创建该目录
```

第77.13节：阻塞或重定向标准输出/错误

有时设计不佳的第三方库会向System.out或System.err流写入不需要的诊断信息。推荐的解决方案是要么找到更好的库，要么（在开源情况下）修复问题并向开发者提交补丁。

如果上述解决方案不可行，则应考虑重定向这些流。

命令行重定向

在 UNIX、Linux 或 MacOSX 系统上，可以通过 shell 使用 > 重定向来实现。例如：

```
$ java -jar app.jar arg1 arg2 > /dev/null 2>&1  
$ java -jar app.jar arg1 arg2 > out.log 2> error.log
```

第一个命令将标准输出和标准错误重定向到“/dev/null”，这会丢弃写入这些流的所有内容。第二个命令将标准输出重定向到“out.log”，标准错误重定向到“error.log”。

（有关重定向的更多信息，请参阅您所使用的命令 shell 的文档。类似的建议也适用于 Windows。）

或者，您也可以在启动 Java 应用程序的包装脚本或批处理文件中实现重定向。

Java 应用程序内的重定向

也可以在 Java 应用程序内部使用 System.setOut() 和 System.setErr() 来重定向流。例如，以下代码片段将标准输出和标准错误重定向到两个日志文件：

```
System.setOut(new PrintStream(new FileOutputStream(new File("out.log"))));  
System.setErr(new PrintStream(new FileOutputStream(new File("err.log"))));
```

如果您想完全丢弃输出，可以创建一个“写入”无效文件描述符的输出流。这在功能上等同于 UNIX 上写入“/dev/null”。

```
System.setOut(new PrintStream(new FileOutputStream(new FileDescriptor())));  
System.setErr(new PrintStream(new FileOutputStream(new FileDescriptor())));
```

parent directories. (source)

Note: createNewFile() will not create a new directory only a file.

```
File singleDir = new File("C:/Users/SomeUser/Desktop/A New Folder/");  
File multiDir = new File("C:/Users/SomeUser/Desktop/A New Folder 2/Another Folder/");  
  
// assume that neither "A New Folder" or "A New Folder 2" exist  
  
singleDir.createNewFile(); // will make a new file called "A New Folder.file"  
singleDir.mkdir(); // will make the directory  
singleDir.mkdirs(); // will make the directory  
  
multiDir.createNewFile(); // will throw a IOException  
multiDir.mkdir(); // will not work  
multiDir.mkdirs(); // will make the directory
```

Section 77.13: Blocking or redirecting standard output / error

Sometimes a poorly designed 3rd-party library will write unwanted diagnostics to `System.out` or `System.err` streams. The recommended solutions to this would be to either find a better library or (in the case of open source) fix the problem and contribute a patch to the developers.

If the above solutions are not feasible, then you should consider redirecting the streams.

Redirection on the command line

On a UNIX, Linux or MacOSX system can be done from the shell using > redirection. For example:

```
$ java -jar app.jar arg1 arg2 > /dev/null 2>&1  
$ java -jar app.jar arg1 arg2 > out.log 2> error.log
```

The first one redirects standard output and standard error to “/dev/null”，which throws away anything written to those streams. The second of redirects standard output to “out.log” and standard error to “error.log”。

（For more information on redirection, refer to the documentation of the command shell you are using. Similar advice applies to Windows.）

Alternatively, you could implement the redirection in a wrapper script or batch file that launches the Java application.

Redirection within a Java application

It is also possible to redirect the streams *within* a Java application using `System.setOut()` and `System.setErr()`. For example, the following snippet redirects standard output and standard error to 2 log files:

```
System.setOut(new PrintStream(new FileOutputStream(new File("out.log"))));  
System.setErr(new PrintStream(new FileOutputStream(new File("err.log"))));
```

If you want to throw away the output entirely, you can create an output stream that “writes” to an invalid file descriptor. This is functionally equivalent to writing to “/dev/null” on UNIX.

```
System.setOut(new PrintStream(new FileOutputStream(new FileDescriptor())));  
System.setErr(new PrintStream(new FileOutputStream(new FileDescriptor())));
```

注意：使用setOut和setErr时要小心：

1. 重定向将影响整个JVM。
2. 这样做会剥夺用户从命令行重定向流的能力。

第77.14节：一次性读取整个文件

```
File f = new File(path);
String content = new Scanner(f).useDelimiter("\\Z").next();
```

\Z 是EOF（文件结束）符号。设置为分隔符时，Scanner会读取直到EOF标志为止的全部内容。

第77.15节：文件锁定

可以使用FileChannel API对文件进行锁定，该API可以从输入输出streams和readers中获取

使用streams的示例

```
// 打开文件流
FileInputStream ios = new FileInputStream(filename);
// 获取底层通道
FileChannel channel = ios.getChannel();

/*
 * 尝试锁定文件。true 表示锁是否为共享锁，即多个进程可以
 * 获取一个
 * 共享锁（仅用于读取）。对可读通道使用 false 会产生
 * 异常。你应该
 * 在使用 false 时使用可写通道（从 FileOutputStream 获得）。tryLock 总是
 * 立即返回
 */
FileLock lock = channel.tryLock(0, Long.MAX_VALUE, true);

if (lock == null) {
    System.out.println("无法获取锁");
} else {
    System.out.println("锁定成功");
}

// 你也可以使用阻塞调用，该调用会阻塞直到获取到锁。
channel.lock();

// 完成文件的所需操作后，释放锁
if (lock != null) {
    lock.release();
}

// 之后关闭文件流
// 使用读取器的示例
RandomAccessFile randomAccessFile = new RandomAccessFile(filename, "rw");
FileChannel channel = randomAccessFile.getChannel();
// 重复上述相同步骤，但现在由于通道处于读写模式，可以将shared设置为true或false
```

Caution: be careful how you use setOut and setErr:

1. The redirection will affect the entire JVM.
2. By doing this, you are taking away the user's ability to redirect the streams from the command line.

Section 77.14: Reading a whole file at once

```
File f = new File(path);
String content = new Scanner(f).useDelimiter("\\Z").next();
```

\Z is the EOF (End of File) Symbol. When set as delimiter the Scanner will read the file until the EOF Flag is reached.

Section 77.15: Locking

A file can be locked using the FileChannel API that can be acquired from Input Output streams and readers

Example with streams

```
// Open a file stream
FileInputStream ios = new FileInputStream(filename);
// get underlying channel
FileChannel channel = ios.getChannel();

/*
 * try to lock the file. true means whether the lock is shared or not i.e. multiple processes can
 * acquire a
 * shared lock (for reading only) Using false with readable channel only will generate an
 * exception. You should
 * use a writable channel (taken from FileOutputStream) when using false. tryLock will always
 * return immediately
 */
FileLock lock = channel.tryLock(0, Long.MAX_VALUE, true);

if (lock == null) {
    System.out.println("Unable to acquire lock");
} else {
    System.out.println("Lock acquired successfully");
}

// you can also use blocking call which will block until a lock is acquired.
channel.lock();

// Once you have completed desired operations of file. release the lock
if (lock != null) {
    lock.release();
}

// close the file stream afterwards
// Example with reader
RandomAccessFile randomAccessFile = new RandomAccessFile(filename, "rw");
FileChannel channel = randomAccessFile.getChannel();
//repeat the same steps as above but now you can use shared as true or false as the channel is in
read write mode
```

第77.16节：使用BufferedInputStream读取文件

使用BufferedInputStream读取文件通常比FileInputStream更快，因为它维护了一个内部

Section 77.16: Reading a file using BufferedInputStream

Reading file using a BufferedInputStream generally faster than FileInputStream because it maintains an internal

缓冲区来存储从底层输入流读取的字节。

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadingDemo {

    public static void main(String[] args) {
        String source = "hello.txt";

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(source))) {
            byte data;
            while ((data = (byte) bis.read()) != -1) {
                System.out.println((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

第77.17节：遍历目录并打印其中的子目录

```
public void iterate(final String dirPath) throws IOException {
    final DirectoryStream<Path> paths = Files.newDirectoryStream(Paths.get(dirPath));
    for (final Path path : paths) {
        if (Files.isDirectory(path)) {
            System.out.println(path.getFileName());
        }
    }
}
```

第77.18节：使用通道（Channel）和缓冲区（Buffer）写文件

要使用Channel写数据到文件，我们需要以下步骤：

- 首先，我们需要获取一个FileOutputStream对象
- 通过FileOutputStream调用getChannel()方法获取FileChannel
- 创建一个ByteBuffer，然后用数据填充它
- 然后我们必须调用ByteBuffer的flip()方法，并将其作为write()方法的参数传入FileChannel的方法
- 写入完成后，我们必须关闭资源

```
import java.io.*;
import java.nio.*;
public class FileChannelWrite {

    public static void main(String[] args) {
        File outputFile = new File("hello.txt");
        String text = "我爱孟加拉国。";

        try {
            FileOutputStream fos = new FileOutputStream(outputFile);

```

buffer to store bytes read from the underlying input stream.

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadingDemo {

    public static void main(String[] args) {
        String source = "hello.txt";

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(source))) {
            byte data;
            while ((data = (byte) bis.read()) != -1) {
                System.out.println((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Section 77.17: Iterate over a directory printing subdirectories in it

```
public void iterate(final String dirPath) throws IOException {
    final DirectoryStream<Path> paths = Files.newDirectoryStream(Paths.get(dirPath));
    for (final Path path : paths) {
        if (Files.isDirectory(path)) {
            System.out.println(path.getFileName());
        }
    }
}
```

Section 77.18: Writing a file using Channel and Buffer

To write data to a file using Channel we need to have the following steps:

- First, we need to get an object of `FileOutputStream`
- Acquire `FileChannel` calling the `getChannel()` method from the `FileOutputStream`
- Create a `ByteBuffer` and then fill it with data
- Then we have to call the `flip()` method of the `ByteBuffer` and pass it as an argument of the `write()` method of the `FileChannel`
- Once we are done writing, we have to close the resource

```
import java.io.*;
import java.nio.*;
public class FileChannelWrite {

    public static void main(String[] args) {
        File outputFile = new File("hello.txt");
        String text = "I love Bangladesh.";

        try {
            FileOutputStream fos = new FileOutputStream(outputFile);

```

```

FileChannel fileChannel = fos.getChannel();
byte[] bytes = text.getBytes();
ByteBuffer buffer = ByteBuffer.wrap(bytes);
fileChannel.write(buffer);
fileChannel.close();
} catch (java.io.IOException e) {
e.printStackTrace();
}
}
}

```

第77.19节：使用PrintStream写文件

我们可以使用PrintStream类来写文件。它有多个方法允许你打印任何数据类型的值。
println()方法会追加一个新行。打印完成后，我们必须刷新PrintStream。

```

import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.time.LocalDate;

public class FileWritingDemo {
    public static void main(String[] args) {
        String destination = "file1.txt";

        try(PrintStream ps = new PrintStream(destination)){
            ps.println("Stackoverflow 文档看起来很有趣。");
            ps.println();
            ps.println("我爱 Java !");
            ps.printf("今天是: %1$tm/%1$td/%1$tY", LocalDate.now());

            ps.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

第 77.20 节：遍历目录并按文件扩展名过滤

```

public void iterateAndFilter() throws IOException {
    Path dir = Paths.get("C:/foo/bar");
    PathMatcher imageFileMatcher =
    FileSystems.getDefault().getPathMatcher(
        "regex:.*(?i:jpg|jpeg|png|gif|bmp|jpe|jfif)");

    try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir,
        entry -> imageFileMatcher.matches(entry.getFileName()))) {

        for (Path path : stream) {
            System.out.println(path.getFileName());
        }
    }
}

```

```

FileChannel fileChannel = fos.getChannel();
byte[] bytes = text.getBytes();
ByteBuffer buffer = ByteBuffer.wrap(bytes);
fileChannel.write(buffer);
fileChannel.close();
} catch (java.io.IOException e) {
e.printStackTrace();
}
}
}

```

Section 77.19: Writing a file using PrintStream

We can use [PrintStream](#) class to write a file. It has several methods that let you print any data type values.
println() method appends a new line. Once we are done printing, we have to flush the [PrintStream](#).

```

import java.io.FileNotFoundException;
import java.io.PrintStream;
import java.time.LocalDate;

public class FileWritingDemo {
    public static void main(String[] args) {
        String destination = "file1.txt";

        try(PrintStream ps = new PrintStream(destination)){
            ps.println("Stackoverflow documentation seems fun.");
            ps.println();
            ps.println("I love Java!");
            ps.printf("Today is: %1$tm/%1$td/%1$tY", LocalDate.now());

            ps.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

Section 77.20: Iterating over a directory and filter by file extension

```

public void iterateAndFilter() throws IOException {
    Path dir = Paths.get("C:/foo/bar");
    PathMatcher imageFileMatcher =
    FileSystems.getDefault().getPathMatcher(
        "regex:.*(?i:jpg|jpeg|png|gif|bmp|jpe|jfif)");

    try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir,
        entry -> imageFileMatcher.matches(entry.getFileName()))) {

        for (Path path : stream) {
            System.out.println(path.getFileName());
        }
    }
}

```

第77.21节：访问ZIP文件的内容

Java 7的FileSystem API允许使用Java NIO文件API以与操作其他文件系统相同的方式读取和添加Zip文件中的条目。

FileSystem是一个资源，使用后应正确关闭，因此应使用try-with-resources代码块。

从现有文件读取

```
Path pathToZip = Paths.get("path/to/file.zip");
try(FileSystem zipFs = FileSystems.newFileSystem(pathToZip, null)) {
    Path root = zipFs.getPath("/");
    ... //以与普通文件相同的方式访问zip文件的内容
} catch(IOException ex) {
    ex.printStackTrace();
}
```

创建新文件

```
Map<String, String> env = new HashMap<>();
env.put("create", "true"); //创建新zip文件时必需
env.put("encoding", "UTF-8"); //可选：默认是UTF-8
URI uri = URI.create("jar:file:/path/to/file.zip");
try (FileSystem zipfs = FileSystems.newFileSystem(uri, env)) {
    Path newFile = zipFs.getPath("/newFile.txt");
    //写入文件
    Files.write(newFile, "Hello world".getBytes());
} catch(IOException ex) {
    ex.printStackTrace();
}
```

Section 77.21: Accessing the contents of a ZIP file

The FileSystem API of Java 7 allows to read and add entries from or to a Zip file using the Java NIO file API in the same way as operating on any other filesystem.

The FileSystem is a resource that should be properly closed after use, therefore the try-with-resources block should be used.

Reading from an existing file

```
Path pathToZip = Paths.get("path/to/file.zip");
try(FileSystem zipFs = FileSystems.newFileSystem(pathToZip, null)) {
    Path root = zipFs.getPath("/");
    ... //access the content of the zip file same as ordinary files
} catch(IOException ex) {
    ex.printStackTrace();
}
```

Creating a new file

```
Map<String, String> env = new HashMap<>();
env.put("create", "true"); //required for creating a new zip file
env.put("encoding", "UTF-8"); //optional: default is UTF-8
URI uri = URI.create("jar:file:/path/to/file.zip");
try (FileSystem zipfs = FileSystems.newFileSystem(uri, env)) {
    Path newFile = zipFs.getPath("/newFile.txt");
    //writing to file
    Files.write(newFile, "Hello world".getBytes());
} catch(IOException ex) {
    ex.printStackTrace();
}
```

第78章：Scanner (扫描器)

参数	详情
来源	来源可以是字符串、文件或任何类型的输入流

第78.1节：最常见任务的一般模式

下面是如何正确使用java.util.Scanner类从System.in交互式读取用户输入的示例（有时称为stdin，尤其是在C、C++及其他语言以及Unix和Linux中）。它惯用地演示了最常见的请求操作。

```
package com.stackoverflow.scanner;

import javax.annotation.NonNull;
import java.math.BigInteger;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.*;
import java.util.regex.Pattern;

import static java.lang.String.format;

public class ScannerExample
{
    private static final Set<String> EXIT_COMMANDS;
    private static final Set<String> HELP_COMMANDS;
    private static final Pattern DATE_PATTERN;
    private static final String HELP_MESSAGE;

    static
    {
        final SortedSet<String> ecmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        ecmds.addAll(Arrays.asList("exit", "done", "quit", "end", "fino"));
        EXIT_COMMANDS = Collections.unmodifiableSortedSet(ecmds);
        final SortedSet<String> hcmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        hcmds.addAll(Arrays.asList("help", "helpi", "?"));
        HELP_COMMANDS = Collections.unmodifiableSet(hcmds);
        DATE_PATTERN = Pattern.compile("\\d{4}([-\\/])\\d{2}\\1\\d{2}"); //
        http://regex101.com/r/xB8dR3/1
        HELP_MESSAGE = format("Please enter some data or enter one of the following commands to
exit %s", EXIT_COMMANDS);
    }

    /**
     * 使用异常来控制执行流程总是很糟糕。
     * 这就是为什么将其封装在一个方法中，特意这样做是为了不引入任何外部库，      * 使这个示例完全自包含。
     *
     * @param s 可能的URL
     * @return 如果s表示一个有效的URL则返回true，否则返回false
     */
    private static boolean isValidURL(@NonNull final String s)
    {
        try { new URL(s); return true; }
        catch (final MalformedURLException e) { return false; }
    }

    private static void output(@NonNull final String format, @NonNull final Object... args)
    
```

Chapter 78: Scanner

Parameter	Details
Source	Source could be either one of String, File or any kind of InputStream

Section 78.1: General Pattern that does most commonly asked about tasks

The following is how to properly use the `java.util.Scanner` class to interactively read user input from `System.in` correctly(sometimes referred to as `stdin`, especially in C, C++ and other languages as well as in Unix and Linux). It idiomatically demonstrates the most common things that are requested to be done.

```
package com.stackoverflow.scanner;

import javax.annotation.NonNull;
import java.math.BigInteger;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.*;
import java.util.regex.Pattern;

import static java.lang.String.format;

public class ScannerExample
{
    private static final Set<String> EXIT_COMMANDS;
    private static final Set<String> HELP_COMMANDS;
    private static final Pattern DATE_PATTERN;
    private static final String HELP_MESSAGE;

    static
    {
        final SortedSet<String> ecmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        ecmds.addAll(Arrays.asList("exit", "done", "quit", "end", "fino"));
        EXIT_COMMANDS = Collections.unmodifiableSortedSet(ecmds);
        final SortedSet<String> hcmds = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        hcmds.addAll(Arrays.asList("help", "helpi", "?"));
        HELP_COMMANDS = Collections.unmodifiableSet(hcmds);
        DATE_PATTERN = Pattern.compile("\\d{4}([-\\/])\\d{2}\\1\\d{2}"); //
        http://regex101.com/r/xB8dR3/1
        HELP_MESSAGE = format("Please enter some data or enter one of the following commands to
exit %s", EXIT_COMMANDS);
    }

    /**
     * Using exceptions to control execution flow is always bad.
     * That is why this is encapsulated in a method, this is done this
     * way specifically so as not to introduce any external libraries
     * so that this is a completely self contained example.
     * @param s possible url
     * @return true if s represents a valid url, false otherwise
     */
    private static boolean isValidURL(@NonNull final String s)
    {
        try { new URL(s); return true; }
        catch (final MalformedURLException e) { return false; }
    }

    private static void output(@NonNull final String format, @NonNull final Object... args)
    
```

```

{
    System.out.println(format(format, args));
}

public static void main(final String[] args)
{
    final Scanner sis = new Scanner(System.in);
    output(HELP_MESSAGE);
    while (sis.hasNext())
    {
        if (sis.hasNextInt())
        {
            final int next = sis.nextInt();
            output("你输入了一个整数 = %d", next);
        }
        else if (sis.hasNextLong())
        {
            final long next = sis.nextLong();
            output("你输入了一个长整数 = %d", next);
        }
        else if (sis.hasNextDouble())
        {
            final double next = sis.nextDouble();
            output("你输入了一个双精度浮点数 = %f", next);
        }
        else if (sis.hasNext("\\\\d+"))
        {
            final BigInteger next = sis.nextBigInteger();
            output("你输入了一个大整数 = %s", next);
        }
        else if (sis.hasNextBoolean())
        {
            final boolean next = sis.nextBoolean();
            output("你输入了一个布尔值表示 = %s", next);
        }
        else if (sis.hasNext(DATE_PATTERN))
        {
            final String next = sis.next(DATE_PATTERN);
            output("你输入了一个日期表示 = %s", next);
        }
        else // 未分类

            final String next = sis.next();
            if (isValidURL(next))
            {
                output("你输入了一个有效的URL = %s", next);
            }
            else
            {
                if (EXIT_COMMANDS.contains(next))
                {
                    output("退出命令 %s 已发出, 正在退出!", next);
                    break;
                }
                else if (HELP_COMMANDS.contains(next)) { output(HELP_MESSAGE); }
                else { output("你输入了一个未分类的字符串 = %s", next); }
            }
        }
    }
/*

```

这将关闭底层的可读流，在本例中是 `System.in`，并释放这些资源。
调用 `.close()` 后，您将无法再从 `System.in` 读取数据。

```

{
    System.out.println(format(format, args));
}

public static void main(final String[] args)
{
    final Scanner sis = new Scanner(System.in);
    output(HELP_MESSAGE);
    while (sis.hasNext())
    {
        if (sis.hasNextInt())
        {
            final int next = sis.nextInt();
            output("You entered an Integer = %d", next);
        }
        else if (sis.hasNextLong())
        {
            final long next = sis.nextLong();
            output("You entered a Long = %d", next);
        }
        else if (sis.hasNextDouble())
        {
            final double next = sis.nextDouble();
            output("You entered a Double = %f", next);
        }
        else if (sis.hasNext("\\\\d+"))
        {
            final BigInteger next = sis.nextBigInteger();
            output("You entered a BigInteger = %s", next);
        }
        else if (sis.hasNextBoolean())
        {
            final boolean next = sis.nextBoolean();
            output("You entered a Boolean representation = %s", next);
        }
        else if (sis.hasNext(DATE_PATTERN))
        {
            final String next = sis.next(DATE_PATTERN);
            output("You entered a Date representation = %s", next);
        }
        else // unclassified

            final String next = sis.next();
            if (isValidURL(next))
            {
                output("You entered a valid URL = %s", next);
            }
            else
            {
                if (EXIT_COMMANDS.contains(next))
                {
                    output("Exit command %s issued, exiting!", next);
                    break;
                }
                else if (HELP_COMMANDS.contains(next)) { output(HELP_MESSAGE); }
                else { output("You entered an unclassified String = %s", next); }
            }
        }
    }
/*

```

*This will close the underlying Readable, in this case `System.in`, and free those resources.
You will not be to read from `System.in` anymore after this you call `.close()`.*

如果您想将 `System.in` 用于其他用途, 请不要关闭 `Scanner`。

```
/*
sis.close();
    System.exit(0);
}
}
```

第78.2节：使用自定义分隔符

您可以使用自定义分隔符（正则表达式）与 `Scanner` 一起使用，方法是 `.useDelimiter(",")`，以确定输入的读取方式。这与 `String.split(...)` 的工作原理类似。例如，您可以使用 `Scanner` 从一个逗号分隔的字符串列表中读取数据：

```
Scanner scanner = null;
try{
scanner = new Scanner("i,like,unicorns").useDelimiter(",");
while(scanner.hasNext()){
    System.out.println(scanner.next());
}
}catch(Exception e){
e.printStackTrace();
}finally{
    if (scanner != null)
        scanner.close();
}
```

这将允许你逐个读取输入中的每个元素。请注意，不应使用此方法解析CSV数据，应该使用合适的CSV解析库，详见CSV parser for Java了解其他可能性。

第78.3节：使用Scanner读取系统输入

```
Scanner scanner = new Scanner(System.in); // 用于读取系统输入的Scanner对象
String inputTaken = new String();
while (true) {
    String input = scanner.nextLine(); // 读取一行输入
    if (input.matches("\\"s+"))          // 如果匹配空格/制表符, 停止读取
        break;
inputTaken += input + " ";
}
System.out.println(inputTaken);
```

扫描器对象被初始化为从键盘读取输入。因此，对于下面从键盘输入的内容，它将产生输出为 从键盘读取

从键盘读
取//
空格

第78.4节：使用Scanner读取文件输入

```
Scanner scanner = null;
try {
scanner = new Scanner(new File("Names.txt"));
while (scanner.hasNext()) {
    System.out.println(scanner.nextLine());
}
}
```

If you wanted to use `System.in` for something else, then don't close the `Scanner`.

```
/*
sis.close();
    System.exit(0);
}
}
```

Section 78.2: Using custom delimiters

You can use custom delimiters (regular expressions) with `Scanner`, with `.useDelimiter(",")`, to determine how the input is read. This works similarly to `String.split(...)`. For example, you can use `Scanner` to read from a list of comma separated values in a String:

```
Scanner scanner = null;
try{
scanner = new Scanner("i,like,unicorns").useDelimiter(",");
while(scanner.hasNext()){
    System.out.println(scanner.next());
}
}catch(Exception e){
e.printStackTrace();
}finally{
    if (scanner != null)
        scanner.close();
}
```

This will allow you to read every element in the input individually. Note that you should **not** use this to parse CSV data, instead, use a proper CSV parser library, see [CSV parser for Java](#) for other possibilities.

Section 78.3: Reading system input using Scanner

```
Scanner scanner = new Scanner(System.in); //Scanner obj to read System input
String inputTaken = new String();
while (true) {
    String input = scanner.nextLine(); // reading one line of input
    if (input.matches("\\"s+"))          // if it matches spaces/tabs, stop reading
        break;
inputTaken += input + " ";
}
System.out.println(inputTaken);
```

The scanner object is initialized to read input from keyboard. So for the below input from keyboard, it'll produce the output as Reading from keyboard

Reading
from
keyboard
//space

Section 78.4: Reading file input using Scanner

```
Scanner scanner = null;
try {
scanner = new Scanner(new File("Names.txt"));
while (scanner.hasNext()) {
    System.out.println(scanner.nextLine());
}
}
```

```

    }
} catch (Exception e) {
    System.err.println("发生异常！");
} finally {
    if (scanner != null)
        scanner.close();
}

```

这里通过传入一个包含文本文件名的File对象来创建一个Scanner对象。该文本文件将由File对象打开，并在接下来的代码中由Scanner对象读取。scanner.hasNext()方法将检查文本文件中是否有下一行数据。结合while循环，可以遍历Names.txt文件中的每一行数据。要获取数据本身，可以使用诸如

nextLine()、nextInt()、nextBoolean()等方法。在上述示例中，使用了scanner.nextLine()。nextLine()指的是文本文件中的下一行，结合Scanner对象可以打印该行的内容。

关闭Scanner对象时，可以使用close()方法。

使用带资源的try语句（Java 7及以后版本），上述代码可以优雅地写成如下形式。

```

try (Scanner scanner = new Scanner(new File("Names.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
}

```

第78.5节：使用Scanner将整个输入读取为字符串

你可以使用Scanner通过使用\Z（整个输入）作为分隔符，将输入中的所有文本作为字符串读取。例如，这可以用来在一行中读取文本文件中的所有文本：

```

String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);

```

请记住，你必须关闭Scanner，并且捕获它可能抛出的IOException，如示例“使用Scanner读取文件输入”中所述。

第78.6节：仔细关闭Scanner

如果你使用System.in作为构造函数的参数创建Scanner，需要注意关闭Scanner时也会关闭InputStream，因此之后任何尝试读取该输入（或任何其他Scanner对象）的操作都会抛出java.util.NoSuchElementException或

java.lang.IllegalStateException

示例：

```

Scanner sc1 = new Scanner(System.in);
Scanner sc2 = new Scanner(System.in);
int x1 = sc1.nextInt();
sc1.close();
// java.util.NoSuchElementException
int x2 = sc2.nextInt();
// java.lang.IllegalStateException
x2 = sc1.nextInt();

```

```

    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
} finally {
    if (scanner != null)
        scanner.close();
}

```

Here a Scanner object is created by passing a File object containing the name of a text file as input. This text file will be opened by the File object and read in by the scanner object in the following lines. scanner.hasNext() will check to see if there is a next line of data in the text file. Combining that with a while loop will allow you to iterate through every line of data in the Names.txt file. To retrieve the data itself, we can use methods such as nextLine(),nextInt(),nextBoolean(), etc. In the example above, scanner.nextLine() is used. nextLine() refers to the following line in a text file, and combining it with a scanner object allows you to print the contents of the line. To close a scanner object, you would use .close().

Using try with resources (from Java 7 onwards), the above mentioned code can be written elegantly as below.

```

try (Scanner scanner = new Scanner(new File("Names.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (Exception e) {
    System.err.println("Exception occurred!");
}

```

Section 78.5: Read the entire input as a String using Scanner

You can use Scanner to read all of the text in the input as a String, by using \Z (entire input) as the delimiter. For example, this can be used to read all text in a text file in one line:

```

String content = new Scanner(new File("filename")).useDelimiter("\\Z").next();
System.out.println(content);

```

Remember that you'll have to close the Scanner, as well as catch the IOException this may throw, as described in the example Reading file input using Scanner.

Section 78.6: Carefully Closing a Scanner

it can happen that you use a scanner with the System.in as parameter for the constructor, then you need to be aware that closing the scanner will close the InputStream too giving as next that every try to read the input on that (Or any other scanner object) will throw an java.util.NoSuchElementException or an java.lang.IllegalStateException

example:

```

Scanner sc1 = new Scanner(System.in);
Scanner sc2 = new Scanner(System.in);
int x1 = sc1.nextInt();
sc1.close();
// java.util.NoSuchElementException
int x2 = sc2.nextInt();
// java.lang.IllegalStateException
x2 = sc1.nextInt();

```

第78.7节：从命令行读取整数

```
import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();
```

如果你想从命令行读取一个整数，只需使用这段代码。首先，你必须创建一个监听 System.in 的 Scanner 对象，System.in 默认是命令行输入，当你从命令行启动程序时。之后，借助 Scanner 对象，你读取用户传入命令行的第一个整数，并将其存储在变量 number 中。现在你可以对该存储的整数进行任何操作。

Section 78.7: Read an int from the command line

```
import java.util.Scanner;

Scanner s = new Scanner(System.in);
int number = s.nextInt();
```

If you want to read an int from the command line, just use this snippet. First of all, you have to create a Scanner object, that listens to System.in, which is by default the Command Line, when you start the program from the command line. After that, with the help of the Scanner object, you read the first int that the user passes into the command line and store it in the variable number. Now you can do whatever you want with that stored int.

第79章：接口

接口 (interface) 是一种引用类型，类似于类，可以使用 `interface` 关键字声明。接口只能包含常量、方法签名、默认方法、静态方法和嵌套类型。方法体仅存在于默认方法和静态方法中。像抽象类一样，接口不能被实例化——它们只能被类实现或被其他接口继承。接口是 Java 中实现完全抽象的常用方式。

第79.1节：实现多个接口

Java 类可以实现多个接口。

```
public interface NoiseMaker {  
    String noise = "Making Noise"; // 接口变量默认是 public static final  
  
    String makeNoise(); // 接口方法默认是 public abstract  
}  
  
public interface 食物食用者 {  
    void 吃(食物 food);  
}  
  
public class 猫 implements 噪声制造者, 食物食用者 {  
    @Override  
    public String makeNoise() {  
        return "喵";  
    }  
  
    @Override  
    public void 吃(食物 food) {  
        System.out.println("感激地喵喵叫");  
    }  
}
```

注意猫类必须实现两个接口中继承的抽象方法。此外，注意一个类实际上可以实现任意数量的接口（由于JVM

限制，最大数量为65,535）。

```
噪声制造者 noiseMaker = new 猫(); // 有效  
食物食用者 foodEater = new 猫(); // 有效  
猫 cat = new 猫(); // 有效  
  
Cat invalid1 = new NoiseMaker(); // 无效  
Cat invalid2 = new FoodEater(); // 无效
```

注意：

1. 接口中声明的所有变量都是 `public static final`
2. 接口中声明的所有方法都是 `public abstract` (此说法仅适用于Java 7及以前版本)
3. 从Java 8开始，接口中允许包含不必是抽象的方法；此类方法称为默认方法 (default methods)
4. 如果多个接口声明了签名相同的方法，则实际上只视为一个方法，且无法区分该方法是从哪个接口实现的
5. 每个接口在编译时都会生成一个对应的`InterfaceName.class`文件

Chapter 79: Interfaces

An *interface* is a reference type, similar to a class, which can be declared by using `interface` keyword. Interfaces can contain only constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Like abstract classes, Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces. Interface is a common way to achieve full abstraction in Java.

Section 79.1: Implementing multiple interfaces

A Java class can implement multiple interfaces.

```
public interface NoiseMaker {  
    String noise = "Making Noise"; // interface variables are public static final by default  
  
    String makeNoise(); //interface methods are public abstract by default  
}  
  
public interface FoodEater {  
    void eat(Food food);  
}  
  
public class Cat implements NoiseMaker, FoodEater {  
    @Override  
    public String makeNoise() {  
        return "meow";  
    }  
  
    @Override  
    public void eat(Food food) {  
        System.out.println("meows appreciatively");  
    }  
}
```

Notice how the Cat class **must** implement the inherited `abstract` methods in both the interfaces. Furthermore, notice how a class can practically implement as many interfaces as needed (there is a limit of **65,535** due to [JVM Limitation](#)).

```
NoiseMaker noiseMaker = new Cat(); // Valid  
FoodEater foodEater = new Cat(); // Valid  
Cat cat = new Cat(); // valid  
  
Cat invalid1 = new NoiseMaker(); // Invalid  
Cat invalid2 = new FoodEater(); // Invalid
```

Note:

1. All variables declared in an interface are `public static final`
2. All methods declared in an interface methods are `public abstract` (This statement is valid only through Java 7. From Java 8, you are allowed to have methods in an interface, which need not be abstract; such methods are known as default methods)
3. Interfaces cannot be declared as `final`
4. If more than one interface declares a method that has identical signature, then effectively it is treated as only one method and you cannot distinguish from which interface method is implemented
5. A corresponding `InterfaceName.class` file would be generated for each interface, upon compilation

第79.2节：声明和实现接口

使用interface关键字声明接口：

```
public interface Animal {  
    String getSound(); // 接口方法默认是public的  
}
```

重写注解

```
@Override  
public String getSound() {  
    // 代码写在这里...  
}
```

这会强制编译器检查我们是否在重写，防止程序定义新方法或搞乱方法签名。

接口是使用implements关键字实现的。

```
public class 猫 implements 动物 {  
  
    @Override  
    public String 获得声音() {  
        return "喵";  
    }  
  
}  
  
public class 狗 implements 动物 {  
  
    @Override  
    public String 获得声音() {  
        return "汪";  
    }  
}
```

在示例中，类猫和狗必须定义获得声音()方法，因为接口的方法本质上是抽象的（默认方法除外）。

使用接口

```
动物 猫 = new 猫();  
动物 狗 = new 狗();  
  
System.out.println(猫.获得声音()); // 输出 "喵"  
System.out.println(狗.获得声音()); // 输出 "汪"
```

第79.3节：扩展接口

接口可以通过 extends 关键字继承另一个接口。

```
public interface BasicResourceService {  
    Resource getResource();  
}  
  
public interface ExtendedResourceService extends BasicResourceService {  
    void updateResource(Resource resource);
```

Section 79.2: Declaring and Implementing an Interface

Declaration of an interface using the **interface** keyword:

```
public interface Animal {  
    String getSound(); // Interface methods are public by default  
}
```

Override Annotation

```
@Override  
public String getSound() {  
    // Code goes here...  
}
```

This forces the compiler to check that we are overriding and prevents the program from defining a new method or messing up the method signature.

Interfaces are implemented using the **implements** keyword.

```
public class Cat implements Animal {  
  
    @Override  
    public String getSound() {  
        return "meow";  
    }  
  
}  
  
public class Dog implements Animal {  
  
    @Override  
    public String getSound() {  
        return "woof";  
    }  
}
```

In the example, classes Cat and Dog **must** define the getSound() method as methods of an interface are inherently abstract (with the exception of default methods).

Using the interfaces

```
Animal cat = new Cat();  
Animal dog = new Dog();  
  
System.out.println(cat.getSound()); // prints "meow"  
System.out.println(dog.getSound()); // prints "woof"
```

Section 79.3: Extending an interface

An interface can extend another interface via the **extends** keyword.

```
public interface BasicResourceService {  
    Resource getResource();  
}  
  
public interface ExtendedResourceService extends BasicResourceService {  
    void updateResource(Resource resource);
```

```
}
```

现在，实现 ExtendedResourceService 的类需要同时实现 getResource() 和 updateResource() 方法。

继承多个接口

与类不同，extends 关键字可以用于继承多个接口（用逗号分隔），从而将多个接口组合成一个新接口

```
public interface BasicResourceService {  
    Resource getResource();  
}  
  
public interface AlternateResourceService {  
    Resource getAlternateResource();  
}  
  
public interface ExtendedResourceService extends BasicResourceService, AlternateResourceService {  
    Resource updateResource(Resource resource);  
}
```

在这种情况下，实现 ExtendedResourceService 的类需要实现 getResource()、getAlternateResource() 和 updateResource() 方法。

第79.4节：接口的实用性

接口在许多情况下非常有用。例如，假设你有一个动物列表，想要遍历该列表，打印每个动物发出的声音。

```
{猫, 狗, 鸟}
```

一种方法是使用接口。这将允许对所有类调用相同的方法

```
public interface 动物 {  
    public String 获取声音();  
}
```

任何实现 动物 接口的类都必须有一个 获取声音() 方法，但它们可以有不同的实现

```
public class 狗 implements 动物 {  
    public String 获取声音() {  
        return "汪";  
    }  
}  
  
public class 猫 implements 动物 {  
    public String 获取声音() {  
        return "喵";  
    }  
}  
  
public class 鸟 implements 动物 {  
    public String 获取声音() {  
        return "Chirp";  
    }  
}
```

```
}
```

Now a class implementing ExtendedResourceService will need to implement both getResource() and updateResource().

Extending multiple interfaces

Unlike classes, the **extends** keyword can be used to extend multiple interfaces (Separated by commas) allowing for combinations of interfaces into a new interface

```
public interface BasicResourceService {  
    Resource getResource();  
}  
  
public interface AlternateResourceService {  
    Resource getAlternateResource();  
}  
  
public interface ExtendedResourceService extends BasicResourceService, AlternateResourceService {  
    Resource updateResource(Resource resource);  
}
```

In this case a class implementing ExtendedResourceService will need to implement getResource(), getAlternateResource(), and updateResource().

Section 79.4: Usefulness of interfaces

Interfaces can be extremely helpful in many cases. For example, say you had a list of animals and you wanted to loop through the list, each printing the sound they make.

```
{cat, dog, bird}
```

One way to do this would be to use interfaces. This would allow for the same method to be called on all of the classes

```
public interface Animal {  
    public String getSound();  
}
```

Any class that **implements** Animal also must have a getSound() method in them, yet they can all have different implementations

```
public class Dog implements Animal {  
    public String getSound() {  
        return "Woof";  
    }  
}  
  
public class Cat implements Animal {  
    public String getSound() {  
        return "Meow";  
    }  
}  
  
public class Bird implements Animal {  
    public String getSound() {  
        return "Chirp";  
    }  
}
```

```
}
```

我们现在有三个不同的类，每个类都有一个getSound()方法。因为所有这些类都实现了声明了getSound()方法的Animal接口，所以任何Animal的实例都可以调用getSound()

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();

dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

因为它们每一个都是Animal，我们甚至可以把这些动物放到一个列表中，循环遍历它们，并打印出它们的声音

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

因为数组的顺序是Dog、Cat，然后是Bird，控制台将打印出"Woof Meow Chirp"

接口也可以用作函数的返回值。例如，如果输入是"dog"则返回Dog，如果输入是"cat"则返回Cat，如果是"bird"则返回Bird，然后打印该动物的声音，可以使用

```
public Animal getAnimalByName(String name) {
    switch(name.toLowerCase()) {
        case "dog":
            return new 狗();
        case "猫":
            return new 猫();
        case "鸟":
            return new 鸟();
        default:
            return null;
    }
}

public String 根据名称获取动物叫声(String 名称){
    Animal 动物 = 根据名称获取动物(名称);
    if (动物 == null) {
        return null;
    } else {
        return 动物.获取叫声();
    }
}

String 狗叫声 = 根据名称获取动物叫声("狗"); // "汪汪"
String 猫叫声 = 根据名称获取动物叫声("猫"); // "喵喵"
String 鸟叫声 = 根据名称获取动物叫声("鸟"); // "啾啾"
String 灯泡叫声 = 根据名称获取动物叫声("灯泡"); // null
```

接口对于扩展性也很有用，因为如果你想添加一种新的动物类型，你不需要更改对它们执行的任何操作。

```
}
```

We now have three different classes, each of which has a getSound() method. Because all of these classes implement the Animal interface, which declares the getSound() method, any instance of an Animal can have getSound() called on it

```
Animal dog = new Dog();
Animal cat = new Cat();
Animal bird = new Bird();

dog.getSound(); // "Woof"
cat.getSound(); // "Meow"
bird.getSound(); // "Chirp"
```

Because each of these is an Animal, we could even put the animals in a list, loop through them, and print out their sounds

```
Animal[] animals = { new Dog(), new Cat(), new Bird() };
for (Animal animal : animals) {
    System.out.println(animal.getSound());
}
```

Because the order of the array is Dog, Cat, and then Bird, "Woof Meow Chirp" will be printed to the console.

Interfaces can also be used as the return value for functions. For example, returning a Dog if the input is "dog", Cat if the input is "cat", and Bird if it is "bird", and then printing the sound of that animal could be done using

```
public Animal getAnimalByName(String name) {
    switch(name.toLowerCase()) {
        case "dog":
            return new Dog();
        case "cat":
            return new Cat();
        case "bird":
            return new Bird();
        default:
            return null;
    }
}

public String getAnimalSoundByName(String name){
    Animal animal = getAnimalByName(name);
    if (animal == null) {
        return null;
    } else {
        return animal.getSound();
    }
}

String dogSound = getAnimalSoundByName("dog"); // "Woof"
String catSound = getAnimalSoundByName("cat"); // "Meow"
String birdSound = getAnimalSoundByName("bird"); // "Chirp"
String lightbulbSound = getAnimalSoundByName("lightbulb"); // null
```

Interfaces are also useful for extensibility, because if you want to add a new type of Animal, you wouldn't need to change anything with the operations you perform on them.

第79.5节：默认方法

默认方法是在Java 8中引入的一种在接口内部指定实现的方法。这可以用来避免典型的“基类”或“抽象”类，通过提供接口的部分实现，并限制子类层次结构。

观察者模式实现

例如，可以直接在接口中实现观察者-监听者模式，为实现类提供更多灵活性。

```
接口观察者{
    void onAction(String a);
}

接口可观察{
    public abstract List<观察者> getObservers();

    public default void addObserver(观察者 o){
        getObservers().add(o);
    }

    public default void notify(String something ){
        for( Observer l : getObservers() ){
            l.onAction(something);
        }
    }
}
```

现在，任何类只需实现Observable接口，就可以被设为“可观察”的，同时仍然可以属于不同的类层次结构。

```
抽象类 Worker{
    公共抽象 void work();
}

公共类 MyWorker extends Worker 实现 Observable {

    私有 List<Observer> myObservers = new ArrayList<Observer>();

    @Override
    公共 List<Observer> getObservers() {
        返回 myObservers;
    }

    @Override
    公共 void work(){
        notify("开始工作");
        // 代码写在这里...
    }

    notify("完成工作");
}

public static void main(String[] args) {
    MyWorker w = new MyWorker();
}
```

```
w.addListener(new Observer() {
    @Override
```

Section 79.5: Default methods

Introduced in Java 8, default methods are a way of specifying an implementation inside an interface. This could be used to avoid the typical "Base" or "Abstract" class by providing a partial implementation of an interface, and restricting the subclasses hierarchy.

Observer pattern implementation

For example, it's possible to implement the Observer-Listener pattern directly into the interface, providing more flexibility to the implementing classes.

```
interface Observer {
    void onAction(String a);
}

interface Observable{
    public abstract List<Observer> getObservers();

    public default void addObserver(Observer o){
        getObservers().add(o);
    }

    public default void notify(String something ){
        for( Observer l : getObservers() ){
            l.onAction(something);
        }
    }
}
```

Now, any class can be made "Observable" just by implementing the Observable interface, while being free to be part of a different class hierarchy.

```
abstract class Worker{
    public abstract void work();
}

public class MyWorker extends Worker implements Observable {

    private List<Observer> myObservers = new ArrayList<Observer>();

    @Override
    public List<Observer> getObservers() {
        return myObservers;
    }

    @Override
    public void work(){
        notify("Started work");

        // Code goes here...

        notify("Completed work");
    }

    public static void main(String[] args) {
        MyWorker w = new MyWorker();

        w.addListener(new Observer() {
            @Override
```

```

        public void onAction(String a) {
            System.out.println(a + " (" + new Date() + ")");
        });

w.work();
}
}

```

钻石问题

Java 8 的编译器能够识别由类实现包含相同签名方法的接口时引起的钻石问题。

为了解决该问题，实现类必须重写共享的方法并提供自己的实现。

```

interface InterfaceA {
    public default String getName(){
        return "a";
    }
}

interface InterfaceB {
    public default String getName(){
        return "b";
    }
}

public class ImpClass implements InterfaceA, InterfaceB {

    @Override
    public String getName() {
        //必须提供自己的实现
        return InterfaceA.super.getName() + InterfaceB.super.getName();
    }

    public static void main(String[] args) {
        ImpClass c = new ImpClass();

        System.out.println( c.getName() );           // 输出 "ab"
        System.out.println( ((InterfaceA)c).getName() ); // 输出 "ab"
        System.out.println( ((InterfaceB)c).getName() ); // 输出 "ab"
    }
}

```

仍然存在具有相同名称和参数但返回类型不同的方法的问题，这将导致无法编译。

使用默认方法解决兼容性问题

如果在现有系统的接口中添加方法，而该接口被多个类使用，默认方法的实现非常有用。

为了避免整个系统崩溃，当你向接口添加方法时，可以提供默认方法实现。这样，系统仍然可以编译，实际的实现可以逐步完成。

更多信息，请参见默认方法主题。

```

        public void onAction(String a) {
            System.out.println(a + " (" + new Date() + ")");
        });

        w.work();
    }
}

```

Diamond problem

The compiler in Java 8 is aware of the [diamond problem](#) which is caused when a class is implementing interfaces containing a method with the same signature.

In order to solve it, an implementing class must override the shared method and provide its own implementation.

```

interface InterfaceA {
    public default String getName(){
        return "a";
    }
}

interface InterfaceB {
    public default String getName(){
        return "b";
    }
}

public class ImpClass implements InterfaceA, InterfaceB {

    @Override
    public String getName() {
        //Must provide its own implementation
        return InterfaceA.super.getName() + InterfaceB.super.getName();
    }

    public static void main(String[] args) {
        ImpClass c = new ImpClass();

        System.out.println( c.getName() );           // Prints "ab"
        System.out.println( ((InterfaceA)c).getName() ); // Prints "ab"
        System.out.println( ((InterfaceB)c).getName() ); // Prints "ab"
    }
}

```

There's still the issue of having methods with the same name and parameters with different return types, which will not compile.

Use default methods to resolve compatibility issues

The default method implementations come in very handy if a method is added to an interface in an existing system where the interfaces is used by several classes.

To avoid breaking up the entire system, you can provide a default method implementation when you add a method to an interface. This way, the system will still compile and the actual implementations can be done step by step.

For more information, see the Default Methods topic.

第79.6节：接口中的修饰符

Oracle Java 风格指南指出：

当修饰符是隐式时，不应写出修饰符。

(参见 Oracle 官方代码标准中的修饰符部分，了解上下文及实际 Oracle 文档链接。)

该风格指导特别适用于接口。让我们来看以下代码片段：

```
interface I {  
    public static final int VARIABLE = 0;  
  
    public abstract void method();  
  
    public static void staticMethod() { ... }  
    public default void defaultMethod() { ... }  
}
```

变量

所有接口变量隐式为常量，且隐式带有public（所有人可访问）、static（可通过接口名访问）和final（声明时必须初始化）修饰符：

```
public static final int VARIABLE = 0;
```

方法

- 所有没有提供实现的方法都是隐式的公共（public）和抽象（abstract）方法。

公共抽象（public abstract）无返回值（void）方法（method）();

版本 ≥ Java SE 8

- 所有带有静态（static）或默认（default）修饰符的方法必须提供实现，并且隐式为公共（public）。

公共静态（public static）无返回值（void）静态方法（staticMethod）() { ... }

应用上述所有更改后，我们将得到以下内容：

```
接口 (interface) I {  
    整数 (int) 变量 (VARIABLE) = 0;  
  
    无返回值 (void) 方法 (method)();  
  
    静态 (static) 无返回值 (void) 静态方法 (staticMethod) () { ... }  
    默认 (default) 无返回值 (void) 默认方法 (defaultMethod) () { ... }  
}
```

第79.7节：使用带泛型的接口

假设你想定义一个接口，允许向不同类型的通道（例如 AMQP、JMS 等）发布/消费数据，但你希望能够切换实现细节.....

让我们定义一个可以在多个实现中重用的基本 IO 接口：

Section 79.6: Modifiers in Interfaces

The Oracle Java Style Guide states:

Modifiers should not be written out when they are implicit.

(See Modifiers in Oracle Official Code Standard for the context and a link to the actual Oracle document.)

This style guidance applies particularly to interfaces. Let's consider the following code snippet:

```
interface I {  
    public static final int VARIABLE = 0;  
  
    public abstract void method();  
  
    public static void staticMethod() { ... }  
    public default void defaultMethod() { ... }  
}
```

Variables

All interface variables are implicitly *constants* with implicit **public** (accessible for all), **static** (are accessible by interface name) and **final** (must be initialized during declaration) modifiers:

```
public static final int VARIABLE = 0;
```

Methods

- All methods which *don't provide implementation* are implicitly **public** and **abstract**.

public abstract void method();

Version ≥ Java SE 8

- All methods with **static** or **default** modifier *must provide implementation* and are implicitly **public**.

```
public static void staticMethod() { ... }
```

After all of the above changes have been applied, we will get the following:

```
interface I {  
    int VARIABLE = 0;  
  
    void method();  
  
    static void staticMethod() { ... }  
    default void defaultMethod() { ... }  
}
```

Section 79.7: Using Interfaces with Generics

Let's say you want to define an interface that allows publishing / consuming data to and from different types of channels (e.g. AMQP, JMS, etc), but you want to be able to switch out the implementation details ...

Let's define a basic IO interface that can be re-used across multiple implementations:

```

public interface IO<IncomingType, OutgoingType> {

    void publish(OutgoingType data);
    IncomingType consume();
    IncomingType RPCSubmit(OutgoingType data);

}

```

现在我可以实例化该接口，但由于这些方法没有默认实现，实例化时需要提供实现：

```

IO<String, String> mockIO = new IO<String, String>() {

    private String channel = "somechannel";

    @Override
    public void publish(String data) {
        System.out.println("Publishing " + data + " to " + channel);
    }

    @Override
    public String consume() {
        System.out.println("从 " + channel 消费中");
        return "一些有用的数据";
    }

    @Override
    public String RPCSubmit(String data) {
        return "刚刚收到 " + data + "";
    }

};

mockIO.consume(); // 打印：从 somechannel 消费中
mockIO.publish("TestData"); // 向 somechannel 发布 TestData
System.out.println(mockIO.RPCSubmit("TestData")); // 刚刚收到 TestData

```

我们也可以用该接口做一些更有用的事情，比如说我们想用它来封装一些基本的 RabbitMQ 功能：

```

public class RabbitMQ implements IO<String, String> {

    private String exchange;
    private String queue;

    public RabbitMQ(String exchange, String queue){
        this.exchange = exchange;
        this.queue = queue;
    }

    @Override
    public void publish(String data) {
        rabbit.basicPublish(exchange, queue, data.getBytes());
    }

    @Override
    public String consume() {
        return rabbit.basicConsume(exchange, queue);
    }
}

```

```

public interface IO<IncomingType, OutgoingType> {

    void publish(OutgoingType data);
    IncomingType consume();
    IncomingType RPCSubmit(OutgoingType data);

}

```

Now I can instantiate that interface, but since we don't have default implementations for those methods, it'll need an implementation when we instantiate it:

```

IO<String, String> mockIO = new IO<String, String>() {

    private String channel = "somechannel";

    @Override
    public void publish(String data) {
        System.out.println("Publishing " + data + " to " + channel);
    }

    @Override
    public String consume() {
        System.out.println("Consuming from " + channel);
        return "some useful data";
    }

    @Override
    public String RPCSubmit(String data) {
        return "received " + data + " just now ";
    }

};

mockIO.consume(); // prints: Consuming from somechannel
mockIO.publish("TestData"); // Publishing TestData to somechannel
System.out.println(mockIO.RPCSubmit("TestData")); // received TestData just now

```

We can also do something more useful with that interface, let's say we want to use it to wrap some basic RabbitMQ functions:

```

public class RabbitMQ implements IO<String, String> {

    private String exchange;
    private String queue;

    public RabbitMQ(String exchange, String queue){
        this.exchange = exchange;
        this.queue = queue;
    }

    @Override
    public void publish(String data) {
        rabbit.basicPublish(exchange, queue, data.getBytes());
    }

    @Override
    public String consume() {
        return rabbit.basicConsume(exchange, queue);
    }
}

```

```

@Override
public String RPCSubmit(String data) {
    return rabbit.rpcPublish(exchange, queue, data);
}

}

```

假设我现在想用这个IO接口作为统计自上次系统重启以来访问我网站的次数，并且能够显示访问总数——你可以这样做：

```

import java.util.concurrent.atomic.AtomicLong;

public class VisitCounter implements IO<Long, Integer> {

    private static AtomicLong websiteCounter = new AtomicLong(0);

    @Override
    public void publish(Integer count) {
        websiteCounter.addAndGet(count);
    }

    @Override
    public Long consume() {
        return websiteCounter.get();
    }

    @Override
    public Long RPCSubmit(Integer count) {
        return websiteCounter.addAndGet(count);
    }

}

```

现在让我们使用访问计数器：

```

VisitCounter counter = new VisitCounter();

// 刚刚有4次访问，耶
counter.publish(4);
// 又有一次访问，耶
counter.publish(1);

// 获取统计计数器的数据
System.out.println(counter.consume()); // 输出5

// 显示统计计数器页面的数据，但将其计入页面浏览量
System.out.println(counter.RPCSubmit(1)); // 输出6

```

实现多个接口时，不能实现同一个接口两次。这同样适用于泛型接口。因此，以下代码是无效的，会导致编译错误：

```

interface Printer<T> {
    void print(T value);
}

// 无效！
类 SystemPrinter 实现 Printer<Double>, Printer<Integer> {
    @Override 公共 无返回值 打印(Double d){ 系统输出.打印行("十进制: " + d); }
    @Override 公共 无返回值 打印(Integer i){ 系统输出.打印行("离散: " + i); }
}

```

```

@Override
public String RPCSubmit(String data) {
    return rabbit.rpcPublish(exchange, queue, data);
}

}

```

Let's say I want to use this IO interface now as a way to count visits to my website since my last system restart and then be able to display the total number of visits - you can do something like this:

```

import java.util.concurrent.atomic.AtomicLong;

public class VisitCounter implements IO<Long, Integer> {

    private static AtomicLong websiteCounter = new AtomicLong(0);

    @Override
    public void publish(Integer count) {
        websiteCounter.addAndGet(count);
    }

    @Override
    public Long consume() {
        return websiteCounter.get();
    }

    @Override
    public Long RPCSubmit(Integer count) {
        return websiteCounter.addAndGet(count);
    }

}

```

Now let's use the VisitCounter:

```

VisitCounter counter = new VisitCounter();

// just had 4 visits, yay
counter.publish(4);
// just had another visit, yay
counter.publish(1);

// get data for stats counter
System.out.println(counter.consume()); // prints 5

// show data for stats counter page, but include that as a page view
System.out.println(counter.RPCSubmit(1)); // prints 6

```

When implementing multiple interfaces, you can't implement the same interface twice. That also applies to generic interfaces. Thus, the following code is invalid, and will result in a compile error:

```

interface Printer<T> {
    void print(T value);
}

// Invalid!
class SystemPrinter implements Printer<Double>, Printer<Integer> {
    @Override public void print(Double d){ System.out.println("Decimal: " + d); }
    @Override public void print(Integer i){ System.out.println("Discrete: " + i); }
}

```

}

第79.8节：加强有界类型参数

有界类型参数 允许你对泛型类型参数设置限制：

```
类 SomeClass {  
}  
  
类 Demo<T extends SomeClass> {  
}
```

但类型参数只能绑定到单一的类类型。

接口类型可以绑定到已经有绑定的类型。这是通过使用 & 符号实现的：

```
接口 SomeInterface {  
}  
  
类 GenericClass<T extends SomeClass & SomeInterface> {  
}
```

这加强了绑定，可能需要类型参数继承自多个类型。

多个接口类型可以绑定到一个类型参数：

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {  
}
```

但应谨慎使用。多个接口绑定通常是代码异味的标志，表明应该创建一个新类型，作为其他类型的适配器：

```
interface NewInterface extends FirstInterface, SecondInterface {  
}  
  
class Demo<T extends SomeClass & NewInterface> {  
}
```

第79.9节：在抽象类中实现接口

接口中定义的方法默认是公共抽象的。当抽象类实现接口时，接口中定义的任何方法不必由抽象类实现。这是因为声明为抽象的类可以包含抽象方法声明。因此，继承自接口和/或抽象类的任何抽象方法的实现责任，落在第一个具体子类上。

```
public interface NoiseMaker {  
    void makeNoise();  
}
```

}

Section 79.8: Strengthen bounded type parameters

[Bounded type parameters](#) allow you to set restrictions on generic type arguments:

```
class SomeClass {  
}  
  
class Demo<T extends SomeClass> {  
}
```

But a type parameter can only bind to a single class type.

An interface type can be bound to a type that already had a binding. This is achieved using the & symbol:

```
interface SomeInterface {  
}  
  
class GenericClass<T extends SomeClass & SomeInterface> {  
}
```

This strengthens the bind, potentially requiring type arguments to derive from multiple types.

Multiple interface types can be bound to a type parameter:

```
class Demo<T extends SomeClass & FirstInterface & SecondInterface> {  
}
```

But should be used with caution. Multiple interface bindings is usually a sign of a [code smell](#), suggesting that a new type should be created which acts as an adapter for the other types:

```
interface NewInterface extends FirstInterface, SecondInterface {  
}  
  
class Demo<T extends SomeClass & NewInterface> {  
}
```

Section 79.9: Implementing interfaces in an abstract class

A method defined in an **interface** is by default **public abstract**. When an **abstract class** implements an **interface**, any methods which are defined in the **interface** do not have to be implemented by the **abstract class**. This is because a **class** that is declared **abstract** can contain abstract method declarations. It is therefore the responsibility of the first concrete sub-class to implement any **abstract** methods inherited from any interfaces and/or the **abstract class**.

```
public interface NoiseMaker {  
    void makeNoise();  
}
```

```

public abstract class Animal implements NoiseMaker {
    //不需要声明或实现 makeNoise()
    public abstract void eat();
}

//因为Dog是具体类，必须定义makeNoise()和eat()
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("汪汪");
    }

    @Override
    public void eat() {
        System.out.println("狗吃一些狗粮。");
    }
}

```

从Java 8开始，接口可以声明方法的default默认实现，这意味着该方法不会是abstract抽象的，因此任何具体子类都不必强制实现该方法，而是继承default默认实现，除非被重写。

```

public abstract class Animal implements NoiseMaker {
    //Does not need to declare or implement makeNoise()
    public abstract void eat();
}

//Because Dog is concrete, it must define both makeNoise() and eat()
public class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Borf borf");
    }

    @Override
    public void eat() {
        System.out.println("Dog eats some kibble.");
    }
}

```

From Java 8 onward it is possible for an **interface** to declare **default** implementations of methods which means the method won't be **abstract**, therefore any concrete sub-classes will not be forced to implement the method but will inherit the **default** implementation unless overridden.

第80章：正则表达式

正则表达式是一种特殊的字符序列，帮助匹配或查找其他字符串或字符串集合，使用一种特殊的语法模式。Java通过[java.util.regex](#)包支持正则表达式的使用。本主题旨在通过示例介绍并帮助开发者更好地理解Java中正则表达式的使用方法。

第80.1节：使用捕获组

如果你需要从输入字符串中提取一部分字符串，我们可以使用正则表达式的捕获组。

在这个例子中，我们将从一个简单的电话号码正则表达式开始：

```
\d{3}-\d{3}-\d{4}
```

如果在正则表达式中添加括号，每一对括号都被视为一个捕获组。在这种情况下，我们使用的是所谓的编号捕获组：

```
(\d{3})-(\d{3})-(\d{4})
```

^-----^ ^-----^ ^-----^

第1组 第2组 第3组

在我们可以在Java中使用它之前，必须遵守字符串的规则，转义反斜杠，得到以下模式：

```
"(\\"d{3}\")-(\\"d{3}\")-(\\"d{4}\")"
```

我们首先需要编译正则表达式模式以创建一个Pattern，然后需要一个Matcher来将输入字符串与该模式匹配：

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

接下来，Matcher需要找到第一个匹配正则表达式的子序列：

```
phoneMatcher.find();
```

现在，使用group方法，我们可以从字符串中提取数据：

```
String number = phoneMatcher.group(0); // "800-555-1234" (组0是正则表达式匹配的全部内容)
String aCode = phoneMatcher.group(1); // "800"
String threeDigit = phoneMatcher.group(2); // "555"
String fourDigit = phoneMatcher.group(3); // "1234"
```

注意：Matcher.group()可以替代Matcher.group(0)使用。

版本 ≥ Java SE 7

Java 7 引入了命名捕获组。命名捕获组的功能与数字捕获组相同（只是用名称代替数字），虽然语法上有些许变化。使用命名捕获组可以提高代码的可读性。

我们可以修改上述代码以使用命名组：

Chapter 80: Regular Expressions

A regular expression is a special sequence of characters that helps in matching or finding other strings or sets of strings, using a specialized syntax held in a pattern. Java has support for regular expression usage through the [java.util.regex](#) package. This topic is to introduce and help developers understand more with examples on how Regular Expressions must be used in Java.

Section 80.1: Using capture groups

If you need to extract a part of string from the input string, we can use **capture groups** of regex.

For this example, we'll start with a simple phone number regex:

```
\d{3}-\d{3}-\d{4}
```

If parentheses are added to the regex, each set of parentheses is considered a *capturing group*. In this case, we are using what are called numbered capture groups:

```
(\d{3})-(\d{3})-(\d{4})
```

^-----^ ^-----^ ^-----^

Group 1 Group 2 Group 3

Before we can use it in Java, we must not forget to follow the rules of Strings, escaping the backslashes, resulting in the following pattern:

```
"(\\"d{3}\")-(\\"d{3}\")-(\\"d{4}\")"
```

We first need to compile the regex pattern to make a Pattern and then we need a Matcher to match our input string with the pattern:

```
Pattern phonePattern = Pattern.compile("(\\d{3})-(\\d{3})-(\\d{4})");
Matcher phoneMatcher = phonePattern.matcher("abcd800-555-1234wxyz");
```

Next, the Matcher needs to find the first subsequence that matches the regex:

```
phoneMatcher.find();
```

Now, using the group method, we can extract the data from the string:

```
String number = phoneMatcher.group(0); // "800-555-1234" (Group 0 is everything the regex matched)
String aCode = phoneMatcher.group(1); // "800"
String threeDigit = phoneMatcher.group(2); // "555"
String fourDigit = phoneMatcher.group(3); // "1234"
```

Note: Matcher.group() can be used in place of Matcher.group(0).

Version ≥ Java SE 7

Java 7 introduced named capture groups. Named capture groups function the same as numbered capture groups (but with a name instead of a number), although there are slight syntax changes. Using named capture groups improves readability.

We can alter the above code to use named groups:

```
(?\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
区号      组2  组3
```

要获取“AreaCode”的内容，我们可以改用：

```
String aCode = phoneMatcher.group("AreaCode"); // "800"
```

第80.2节：通过带标志的编译Pattern使用带自定义行为的正则表达式

一个Pattern可以带标志编译，如果正则表达式作为字面量String使用，则使用内联修饰符：

```
Pattern pattern = Pattern.compile("foo.", Pattern.CASE_INSENSITIVE | Pattern.DOTALL); pattern.matcher("FOO").matches(); // 返回true.

/* 如果正则表达式没有以不区分大小写和单行模式编译， * 它会失败，因为 FOO 不匹配 /foo/，且 (换行符) * 不匹配 /. */


Pattern anotherPattern = Pattern.compile("(?si)foo"); anotherPattern.matcher("FOO").matches(); // 返回true.

"foOt".replaceAll("(?si)foo", "ca"); // 返回 "cat"。
```

```
(?\d{3})-(\d{3})-(\d{4})
^-----^ ^-----^ ^-----^
AreaCode      Group 2 Group 3
```

To get the contents of "AreaCode", we can instead use:

```
String aCode = phoneMatcher.group("AreaCode"); // "800"
```

Section 80.2: Using regex with custom behaviour by compiling the Pattern with flags

A Pattern can be compiled with flags, if the regex is used as a literal `String`, use inline modifiers:

```
Pattern pattern = Pattern.compile("foo.", Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
pattern.matcher("FOO\n").matches(); // Is true.

/* Had the regex not been compiled case insensitively and singlelined,
 * it would fail because FOO does not match /foo/ and \n (newline)
 * does not match ../../.
 */

Pattern anotherPattern = Pattern.compile("(?si)foo");
anotherPattern.matcher("FOO\n").matches(); // Is true.

"foOt".replaceAll("(?si)foo", "ca"); // Returns "cat".
```

第80.3节：转义字符

一般情况

要将正则表达式中特定字符（如 `?+|` 等）按字面含义使用，需要对它们进行转义。在常见的正则表达式中，这是通过反斜杠 `\` 来完成的。然而，由于它在 Java 字符串中有特殊含义，因此必须使用双反斜杠 `\\"`。

这两个示例将无法工作：

```
"???.replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???.replaceAll ("|?", "!"); //无效的转义序列
```

这个示例可以正常工作

```
"???.replaceAll ("\\\?", "\\!"); //!!!"
```

拆分以管道符分隔的字符串

这不会返回预期结果：

```
"a|b".split ("|"); // [a, |, b]
```

这会返回预期结果：

```
"a|b".split ("\\\\|"); // [a, b]
```

转义反斜杠 \

Section 80.3: Escape Characters

Generally

To use regular expression specific characters (`?+|` etc.) in their literal meaning they need to be escaped. In common regular expression this is done by a backslash `\`. However, as it has a special meaning in Java Strings, you have to use a double backslash `\\"`.

These two examples will not work:

```
"???.replaceAll ("?", "!"); //java.util.regex.PatternSyntaxException
"???.replaceAll ("\?", "!"); //Invalid escape sequence
```

This example works

```
"???.replaceAll ("\\\?", "\\!"); //!!!"
```

Splitting a Pipe Delimited String

This does not return the expected result:

```
"a|b".split ("|"); // [a, |, b]
```

This returns the expected result:

```
"a|b".split ("\\\\|"); // [a, b]
```

Escaping backslash \

这将导致错误：

```
"\\\".matches("\\\\"); // PatternSyntaxException  
\"\\\".matches("\\\\\""); // 语法错误
```

这样可以正常工作：

```
"\\\".matches("\\\\\""); // true
```

第80.4节：不匹配给定字符串

要匹配不包含给定字符串的内容，可以使用负向前瞻：

正则表达式语法：(?![string-to-not-match])

示例：

```
//不匹配 "popcorn"  
String regexString = "^(?![popcorn]).*$";  
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "匹配成功!" : "不匹配!"));  
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "匹配成功!" : "不匹配!"));
```

输出：

```
[popcorn] 不匹配!  
[unicorn] 匹配成功!
```

第80.5节：使用正则表达式字面量匹配

如果你需要匹配属于正则表达式语法的一部分的字符，可以将全部或部分模式标记为正则表达式字面量。

\Q 标记正则表达式字面量的开始。 \E 标记正则表达式字面量的结束。

```
// 以下代码会抛出PatternSyntaxException异常，因为括号未闭合  
"[123".matches("[123");  
  
// 使用 |Q 和 |E 包裹括号可以使模式按预期匹配。  
"[123".matches("\Q[\E123"); // 返回 true
```

一种更简单的方法是不必记住\Q和\E转义序列，而是使用Pattern.quote()

```
"[123".matches(Pattern.quote("[") + "123"); // 返回true
```

第80.6节：匹配反斜杠

如果你想在正则表达式中匹配反斜杠，你必须对它进行转义。

反斜杠是正则表达式中的转义字符。你可以使用'\\'来表示正则表达式中的单个反斜杠。

然而，反斜杠在Java字符串字面量中也是转义字符。要从字符串字面量创建正则表达式，你必须对每个反斜杠进行转义。在字符串字面量中，'\\\\\\'可以用来创建一个正则表达式

This will give an error:

```
"\\\".matches("\\\\"); // PatternSyntaxException  
\"\\\".matches("\\\\\""); // Syntax Error
```

This works:

```
"\\\".matches("\\\\\""); // true
```

Section 80.4: Not matching a given string

To match something that does *not* contain a given string, one can use negative lookahead:

Regex syntax: (?![string-to-not-match])

Example:

```
//not matching "popcorn"  
String regexString = "^(?!popcorn).*$";  
System.out.println("[popcorn] " + ("popcorn".matches(regexString) ? "matched!" : "nope!"));  
System.out.println("[unicorn] " + ("unicorn".matches(regexString) ? "matched!" : "nope!"));
```

Output:

```
[popcorn] nope!  
[unicorn] matched!
```

Section 80.5: Matching with a regex literal

If you need to match characters that are a part of the regular expression syntax you can mark all or part of the pattern as a regex literal.

\Q marks the beginning of the regex literal. \E marks the end of the regex literal.

```
// the following throws a PatternSyntaxException because of the un-closed bracket  
"[123".matches("[123");  
  
// wrapping the bracket in \Q and \E allows the pattern to match as you would expect.  
"[123".matches("\Q[\E123"); // returns true
```

An easier way of doing it without having to remember the \Q and \E escape sequences is to use Pattern.quote()

```
"[123".matches(Pattern.quote("[") + "123"); // returns true
```

Section 80.6: Matching a backslash

If you want to match a backslash in your regular expression, you'll have to escape it.

Backslash is an escape character in regular expressions. You can use '\\\' to refer to a single backslash in a regular expression.

However, backslash is *also* an escape character in Java literal strings. To make a regular expression from a string literal, you have to escape each of its backslashes. In a string literal '\\\\\' can be used to create a regular expression

包含'\\'，它又可以匹配'\\'。

例如，考虑匹配像"C:\dir\myfile.txt"这样的字符串。正则表达式([A-Za-z]):\\(.*)将匹配，并提供驱动器字母作为捕获组。注意双反斜杠。

要在Java字符串字面量中表示该模式，正则表达式中的每个反斜杠都需要被转义。

```
String path = "C:\\dir\\myfile.txt";
System.out.println( "本地路径: " + path ); // "C:\\dir\\myfile.txt"

String regex = "([A-Za-z]):\\\\\\(.*)"; // 四个匹配一个
System.out.println("正则表达式: " + regex); // "[A-Za-z]):\\(.*)"

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );
if ( matcher.matches() ) {
    System.out.println( "此路径位于驱动器 " + matcher.group( 1 ) + ":" );
    // 此路径位于C盘。
}
```

如果你想匹配两个反斜杠，你会发现自己在字面字符串中使用八个，以表示正则表达式中的四个，从而匹配两个。

```
String path = "\\\\myhost\\share\\myfile.txt";
System.out.println( "UNC 路径: " + path ); // \\myhost\\share\\myfile.txt

String regex = "\\\\\\\\\\\\\\\\(.+?)\\\\\\(.*)"; // 八个反斜杠匹配两个
System.out.println("正则表达式: " + regex); // \\\\\\\(.+?)\\(.*)

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );

if ( matcher.matches() ) {
    System.out.println( "该路径位于主机 " + matcher.group( 1 ) + "。" );
    // 该路径位于主机 'myhost' 上。
}
```

with '\\', which in turn can match '\\'.

For example, consider matching strings like "C:\\dir\\myfile.txt". A regular expression ([A-Za-z]):\\(.*) will match, and provide the drive letter as a capturing group. Note the doubled backslash.

To express that pattern in a Java string literal, each of the backslashes in the regular expression needs to be escaped.

```
String path = "C:\\\\dir\\\\myfile.txt";
System.out.println( "Local path: " + path ); // "C:\\dir\\myfile.txt"

String regex = "([A-Za-z]):\\\\\\(.*)"; // Four to match one
System.out.println("Regex: " + regex); // "[A-Za-z]):\\(.*)"

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );
if ( matcher.matches() ) {
    System.out.println( "This path is on drive " + matcher.group( 1 ) + ":" );
    // This path is on drive C:.
}
```

If you want to match two backslashes, you'll find yourself using eight in a literal string, to represent four in the regular expression, to match two.

```
String path = "\\\\\\\\\\myhost\\\\share\\\\myfile.txt";
System.out.println( "UNC path: " + path ); // \\myhost\\share\\myfile.txt

String regex = "\\\\\\\\\\\\\\\\(.+?)\\\\\\(.*)"; // Eight to match two
System.out.println("Regex: " + regex); // \\\\\\\(.+?)\\(.*)

Pattern pattern = Pattern.compile( regex );
Matcher matcher = pattern.matcher( path );

if ( matcher.matches() ) {
    System.out.println( "This path is on host " + matcher.group( 1 ) + "。" );
    // This path is on host 'myhost'.
}
```

第81章：Comparable和Comparator

第81.1节：使用Comparable<T>或Comparator<T>对列表进行排序

假设我们正在编写一个表示人的类，通过他们的名字和姓氏。我们已经创建了一个基本类来实现这个功能，并且实现了合适的equals和hashCode方法。

```
public class Person {  
  
    private final String lastName; //不变式 - 非空  
    private final String firstName; //不变式 - 非空  
  
    public Person(String firstName, String lastName){  
        this.firstName = firstName != null ? firstName : "";  
        this.lastName = lastName != null ? lastName : "";  
    }  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public String toString() {  
        return lastName + ", " + firstName;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (! (o instanceof Person)) return false;  
        Person p = (Person)o;  
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(firstName, lastName);  
    }  
}
```

现在我们想按姓名对一组 Person 对象进行排序，例如在以下场景中：

```
public static void main(String[] args) {  
    List<Person> people = Arrays.asList(new Person("John", "Doe"),  
                                         new Person("Bob", "Dole"),  
                                         new Person("Ronald", "McDonald"),  
                                         new Person("Alice", "McDonald"),  
                                         new Person("Jill", "Doe"));  
  
    Collections.sort(people); //这目前无法工作。  
}
```

不幸的是，如标记所示，上述代码目前无法编译。Collections.sort(..) 只知道如何对列表进行排序且仅当该列表中的元素是可比较的，或者提供了自定义的比较方法。

如果有人让你对以下列表进行排序：1,3,5,4,2，你会毫无问题地说答案是1,2,3,4,5。这

Chapter 81: Comparable and Comparator

Section 81.1: Sorting a List using Comparable<T> or a Comparator<T>

Say we are working on a class representing a Person by their first and last names. We have created a basic class to do this and implemented proper equals and hashCode methods.

```
public class Person {  
  
    private final String lastName; //invariant - nonnull  
    private final String firstName; //invariant - nonnull  
  
    public Person(String firstName, String lastName){  
        this.firstName = firstName != null ? firstName : "";  
        this.lastName = lastName != null ? lastName : "";  
    }  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public String toString() {  
        return lastName + ", " + firstName;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (! (o instanceof Person)) return false;  
        Person p = (Person)o;  
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(firstName, lastName);  
    }  
}
```

Now we would like to sort a list of Person objects by their name, such as in the following scenario:

```
public static void main(String[] args) {  
    List<Person> people = Arrays.asList(new Person("John", "Doe"),  
                                         new Person("Bob", "Dole"),  
                                         new Person("Ronald", "McDonald"),  
                                         new Person("Alice", "McDonald"),  
                                         new Person("Jill", "Doe"));  
  
    Collections.sort(people); //This currently won't work.  
}
```

Unfortunately, as marked, the above currently won't compile. `Collections.sort(..)` only knows how to sort a list if the elements in that list are comparable, or a custom method of comparison is given.

If you were asked to sort the following list : 1, 3, 5, 4, 2, you'd have no problem saying the answer is 1, 2, 3, 4, 5. This

这是因为整数（无论是在Java中还是数学上）都有一个自然排序，即一个标准的、默认的比较基准排序。为了给我们的Person类一个自然排序，我们实现了Comparable<Person>接口，这要求实现方法compareTo(Person p)：

```
public class Person implements Comparable<Person> {

    private final String lastName; //不变式 - 非空
    private final String firstName; //不变式 - 非空

    public Person(String firstName, String lastName) {
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }

    @Override
    public int compareTo(Person other) {
        // 如果此对象的lastName与另一个对象的lastName不可比较相等,
        // 通过比较他们的lastName来比较此对象和另一个对象。
        // 否则, 通过比较他们的firstName来比较此对象和另一个对象
        int lastNameCompare = lastName.compareTo(other.lastName);
        if (lastNameCompare != 0) {
            return lastNameCompare;
        } else {
            return firstName.compareTo(other.firstName);
        }
    }
}
```

现在，给定的主方法将正常运行

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                         new Person("Bob", "Dole"),
                                         new Person("Ronald", "McDonald"),
                                         new Person("Alice", "McDonald"),
                                         new Person("Jill", "Doe"));
    Collections.sort(people); //现在正常运行
}
```

is because Integers (both in Java and mathematically) have a *natural ordering*, a standard, default comparison base ordering. To give our Person class a natural ordering, we implement Comparable<Person>, which requires implementing the method compareTo(Person p) :

```
public class Person implements Comparable<Person> {

    private final String lastName; //invariant - nonnull
    private final String firstName; //invariant - nonnull

    public Person(String firstName, String lastName) {
        this.firstName = firstName != null ? firstName : "";
        this.lastName = lastName != null ? lastName : "";
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String toString() {
        return lastName + ", " + firstName;
    }

    @Override
    public boolean equals(Object o) {
        if (! (o instanceof Person)) return false;
        Person p = (Person)o;
        return firstName.equals(p.firstName) && lastName.equals(p.lastName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(firstName, lastName);
    }

    @Override
    public int compareTo(Person other) {
        // If this' lastName and other's lastName are not comparably equivalent,
        // Compare this to other by comparing their last names.
        // Otherwise, compare this to other by comparing their first names
        int lastNameCompare = lastName.compareTo(other.lastName);
        if (lastNameCompare != 0) {
            return lastNameCompare;
        } else {
            return firstName.compareTo(other.firstName);
        }
    }
}
```

Now, the main method given will function correctly

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                         new Person("Bob", "Dole"),
                                         new Person("Ronald", "McDonald"),
                                         new Person("Alice", "McDonald"),
                                         new Person("Jill", "Doe"));
    Collections.sort(people); //Now functions correctly
}
```

```
//people 现在按姓氏，然后按名字排序：
// --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

但是，如果你不想或无法修改类Person，你可以提供一个自定义的Comparator<T>来处理任意两个Person对象的比较。如果你被要求对以下列表进行排序：circle,square, rectangle, triangle, hexagon你无法直接排序，但如果你被要求根据角的数量排序，你就可以。类似地，提供一个比较器告诉Java如何比较两个通常不可比较的对象。

```
public class PersonComparator implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        // 如果 p1 的 lastName 和 p2 的 lastName 不相等,
        // 通过比较他们的姓氏来比较 p1 和 p2
        // 否则，通过比较他们的名字来比较 p1 和 p2
        if (p1.getLastName().compareTo(p2.getLastName()) != 0) {
            return p1.getLastName().compareTo(p2.getLastName());
        } else {
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
}
```

```
//假设这里使用的是第一个版本的Person (未实现Comparable接口)
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                         new Person("Bob", "Dole"),
                                         new Person("Ronald", "McDonald"),
                                         new Person("Alice", "McDonald"),
                                         new Person("Jill", "Doe"));
    Collections.sort(people); //非法, Person未实现Comparable接口。
    Collections.sort(people, new PersonComparator()); //合法

    //people现在按姓氏，然后按名字排序：
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

比较器也可以作为匿名内部类创建/使用

```
//假设这里使用的是第一个版本的Person (未实现Comparable接口)
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                         new Person("Bob", "Dole"),
                                         new Person("Ronald", "McDonald"),
                                         new Person("Alice", "McDonald"),
                                         new Person("Jill", "Doe"));
    Collections.sort(people); //非法, Person未实现Comparable接口。

    Collections.sort(people, new PersonComparator()); //合法

    //people现在按姓氏，然后按名字排序：
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald

    //匿名类
    Collections.sort(people, new Comparator<Person>() { //合法的
        public int compare(Person p1, Person p2) {
            //方法代码...
        }
    });
}
```

```
//people is now sorted by last name, then first name:
// --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

If, however, you either do not want or are unable to modify class Person, you can provide a custom Comparator<T> that handles the comparison of any two Person objects. If you were asked to sort the following list: circle, square, rectangle, triangle, hexagon you could not, but if you were asked to sort that list *based on the number of corners*, you could. Just so, providing a comparator instructs Java how to compare two normally not comparable objects.

```
public class PersonComparator implements Comparator<Person> {

    public int compare(Person p1, Person p2) {
        // If p1's lastName and p2's lastName are not comparably equivalent,
        // Compare p1 to p2 by comparing their last names.
        // Otherwise, compare p1 to p2 by comparing their first names
        if (p1.getLastName().compareTo(p2.getLastName()) != 0) {
            return p1.getLastName().compareTo(p2.getLastName());
        } else {
            return p1.getFirstName().compareTo(p2.getFirstName());
        }
    }
}

//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                         new Person("Bob", "Dole"),
                                         new Person("Ronald", "McDonald"),
                                         new Person("Alice", "McDonald"),
                                         new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.
    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald
}
```

Comparators can also be created/used as an anonymous inner class

```
//Assume the first version of Person (that does not implement Comparable) is used here
public static void main(String[] args) {
    List<Person> people = Arrays.asList(new Person("John", "Doe"),
                                         new Person("Bob", "Dole"),
                                         new Person("Ronald", "McDonald"),
                                         new Person("Alice", "McDonald"),
                                         new Person("Jill", "Doe"));
    Collections.sort(people); //Illegal, Person doesn't implement Comparable.

    Collections.sort(people, new PersonComparator()); //Legal

    //people is now sorted by last name, then first name:
    // --> Jill Doe, John Doe, Bob Dole, Alice McDonald, Ronald McDonald

    //Anonymous Class
    Collections.sort(people, new Comparator<Person>() { //Legal
        public int compare(Person p1, Person p2) {
            //Method code...
        }
    });
}
```

```
}
```

版本 ≥ Java SE 8

基于Lambda表达式的比较器

从Java 8开始，比较器也可以用lambda表达式来表示

```
//Lambda  
Collections.sort(people, (p1, p2) -> { //合法的  
    //方法代码....  
});
```

Comparator默认方法

此外，Comparator接口上有一些有趣的默认方法用于构建比较器：

下面的代码构建了一个先按lastName再按firstName进行比较的比较器。

```
Collections.sort(people, Comparator.comparing(Person::getLastName)  
    .thenComparing(Person::getFirstName));
```

反转比较器的顺序

任何比较器也可以通过 reversedMethod轻松反转，该方法会将升序变为降序。

第81.2节：compareTo和compare方法

Comparable<T>接口要求一个方法：

```
public interface Comparable<T> {  
  
    public int compareTo(T other);  
  
}
```

而Comparator<T>接口要求一个方法：

```
public interface Comparator<T> {  
  
    public int compare(T t1, T t2);  
  
}
```

这两个方法本质上做的事情相同，只有一个小区别：compareTo比较this和其他，而compare比较t1和t2，完全不关心this。

除此之外，这两种方法有类似的要求。具体来说（对于 compareTo），比较此对象与指定对象的顺序。返回一个负整数、零或正整数，表示此对象小于、等于或大于指定对象。小于、等于或大于指定对象。因此，对于 a 和 b 的比较：

- 如果 a < b, a.compareTo(b) 和 compare(a,b) 应返回负整数，b.compareTo(a) 和 compare(b,a) 应返回正整数
- 如果 a > b, a.compareTo(b) 和 compare(a,b) 应返回正整数，b.compareTo(a) 和 compare(b,a) 应返回负整数
- 如果 a 与 b 比较相等，所有比较应返回 0。

```
}  
Version ≥ Java SE 8
```

Lambda expression based comparators

As of Java 8, comparators can also be expressed as lambda expressions

```
//Lambda  
Collections.sort(people, (p1, p2) -> { //Legal  
    //Method code....  
});
```

Comparator default methods

Furthermore, there are interesting default methods on the Comparator interface for building comparators : the following builds a comparator comparing by lastName and then firstName.

```
Collections.sort(people, Comparator.comparing(Person::getLastName)  
    .thenComparing(Person::getFirstName));
```

Inversing the order of a comparator

Any comparator can also easily be reversed using the reversedMethod which will change ascending order to descending.

Section 81.2: The compareTo and compare Methods

The Comparable<T> interface requires one method:

```
public interface Comparable<T> {  
  
    public int compareTo(T other);  
  
}
```

And the Comparator<T> interface requires one method:

```
public interface Comparator<T> {  
  
    public int compare(T t1, T t2);  
  
}
```

These two methods do essentially the same thing, with one minor difference: compareTo compares this to other, whereas compare compares t1 to t2, not caring at all about this.

Aside from that difference, the two methods have similar requirements. Specifically (for compareTo), Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Thus, for the comparison of a and b:

- If a < b, a.compareTo(b) and compare(a,b) should return a negative integer, and b.compareTo(a) and compare(b,a) should return a positive integer
- If a > b, a.compareTo(b) and compare(a,b) should return a positive integer, and b.compareTo(a) and compare(b,a) should return a negative integer
- If a equals b for comparison, all comparisons should return 0.

第81.3节：自然（可比较）排序与显式（比较器）排序

有两种 Collections.sort() 方法：

- 一种接受 List<T> 作为参数，其中 T 必须实现 Comparable 并重写 compareTo() 方法以确定排序顺序。
- 另一种接受 List 和 Comparator 作为参数，由 Comparator 决定排序顺序。

首先，这里有一个实现了 Comparable 接口的 Person 类：

```
public class Person implements Comparable<Person> {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    @Override  
    public int compareTo(Person o) {  
        return this.getAge() - o.getAge();  
    }  
    @Override  
    public String toString() {  
        return this.getAge() + "-" + this.getName();  
    }  
}
```

以下是如何使用上述类根据元素的自然顺序（由 compareTo() 方法重写定义）对列表进行排序：

```
//-- 使用方法  
List<Person> pList = new ArrayList<Person>();  
Person p = new Person();  
p.setName("A");  
p.setAge(10);  
pList.add(p);  
p = new Person();  
p.setName("Z");  
p.setAge(20);  
pList.add(p);  
p = new Person();  
p.setName("D");  
p.setAge(30);  
pList.add(p);  
  
//-- 自然排序，即通过对象实现，按年龄排序  
Collections.sort(pList);
```

Section 81.3: Natural (comparable) vs explicit (comparator) sorting

There are two `Collections.sort()` methods:

- One that takes a List<T> as a parameter where T must implement Comparable and override the `compareTo()` method that determines sort order.
- One that takes a List and a Comparator as the arguments, where the Comparator determines the sort order.

First, here is a Person class that implements Comparable:

```
public class Person implements Comparable<Person> {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    @Override  
    public int compareTo(Person o) {  
        return this.getAge() - o.getAge();  
    }  
    @Override  
    public String toString() {  
        return this.getAge() + "-" + this.getName();  
    }  
}
```

Here is how you would use the above class to sort a List in the natural ordering of its elements, defined by the `compareTo()` method override:

```
//-- usage  
List<Person> pList = new ArrayList<Person>();  
Person p = new Person();  
p.setName("A");  
p.setAge(10);  
pList.add(p);  
p = new Person();  
p.setName("Z");  
p.setAge(20);  
pList.add(p);  
p = new Person();  
p.setName("D");  
p.setAge(30);  
pList.add(p);  
  
//-- natural sorting i.e comes with object implementation, by age  
Collections.sort(pList);
```

```
System.out.println(pList);
```

下面是如何使用匿名内联比较器（Comparator）对一个未实现Comparable接口的列表进行排序，或者在本例中，如何以非自然顺序对列表进行排序：

```
//-- 显式排序，在这里定义基于另一个属性的排序，比如按姓名排序
Collections.sort(pList, new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getName().compareTo(o2.getName());
    }
});
System.out.println(pList);
```

第81.4节：使用comparing方法创建Comparator

```
Comparator.comparing(Person::getName)
```

这将为Person类创建一个比较器，使用该人的姓名作为比较依据。同时，也可以使用方法版本来比较long、int和double类型。例如：

```
Comparator.comparingInt(Person::getAge)
```

反转顺序

要创建一个施加反向排序的比较器，请使用 reversed()方法：

```
Comparator.comparing(Person::getName).reversed()
```

比较器链

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

这将创建一个先按姓氏比较再按名字比较的比较器。你可以链式连接任意数量的比较器。

第81.5节：对Map条目进行排序

从Java 8开始，Map.Entry接口上有默认方法，允许对Map的迭代进行排序。

版本 ≥ Java SE 8

```
Map<String, Integer> numberOfEmployees = new HashMap<>();
numberOfEmployees.put("executives", 10);
numberOfEmployees.put("human resources", 32);
numberOfEmployees.put("accounting", 12);
numberOfEmployees.put("IT", 100);

// 输出员工人数最少的部门
numberOfEmployees.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .limit(1)
.forEach(System.out::println); // 输出 : executives=10
```

当然，这些也可以在流（stream）API之外使用：

```
System.out.println(pList);
```

Here is how you would use an anonymous inline Comparator to sort a List that does not implement Comparable, or in this case, to sort a List in an order other than the natural ordering:

```
//-- explicit sorting, define sort on another property here goes with name
Collections.sort(pList, new Comparator<Person>() {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.getName().compareTo(o2.getName());
    }
});
System.out.println(pList);
```

Section 81.4: Creating a Comparator using comparing method

```
Comparator.comparing(Person::getName)
```

This creates a comparator for the class Person that uses this person name as the comparison source. Also it is possible to use method version to compare long, int and double. For example:

```
Comparator.comparingInt(Person::getAge)
```

Reversed order

To create a comparator that imposes the reverse ordering use reversed() method:

```
Comparator.comparing(Person::getName).reversed()
```

Chain of comparators

```
Comparator.comparing(Person::getLastName).thenComparing(Person::getFirstName)
```

This will create a comparator that first compares with last name then compares with first name. You can chain as many comparators as you want.

Section 81.5: Sorting Map entries

As of Java 8, there are default methods on the `Map.Entry` interface to allow sorting of map iterations.

Version ≥ Java SE 8

```
Map<String, Integer> numberOfEmployees = new HashMap<>();
numberOfEmployees.put("executives", 10);
numberOfEmployees.put("human resources", 32);
numberOfEmployees.put("accounting", 12);
numberOfEmployees.put("IT", 100);

// Output the smallest department in terms of number of employees
numberOfEmployees.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .limit(1)
.forEach(System.out::println); // outputs : executives=10
```

Of course, these can also be used outside of the stream api :

版本 ≥ Java SE 8

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());  
Collections.sort(entries, Map.Entry.comparingByValue());
```

Version ≥ Java SE 8

```
List<Map.Entry<String, Integer>> entries = new ArrayList<>(numberOfEmployees.entrySet());  
Collections.sort(entries, Map.Entry.comparingByValue());
```

第82章：Java浮点数运算

浮点数是带有小数部分的数字（通常用小数点表示）。在Java中，有两种浮点数的基本类型，分别是float（使用4字节）和double（使用8字节）。

本说明文档页面详细介绍了Java中浮点数可执行的操作示例。

第82.1节：比较浮点数值

在使用关系运算符比较浮点数值（float 或 double）时应当小心：==、!=、<等。这些运算符根据浮点数的二进制表示来给出结果。例如：

```
public class CompareTest {  
    public static void main(String[] args) {  
        double oneThird = 1.0 / 3.0;  
        double one = oneThird * 3;  
        System.out.println(one == 1.0); // 输出 "false"  
    }  
}
```

计算 oneThird 引入了微小的舍入误差，当我们用 oneThird 乘以 3 时，得到的结果与 1.0 略有不同。

当我们尝试在计算中混合使用 double 和 float 时，这种不精确表示的问题更加明显。

例如：

```
public class CompareTest2 {  
    public static void main(String[] args) {  
        float floatVal = 0.1f;  
        double doubleVal = 0.1;  
        double doubleValCopy = floatVal;  
  
        System.out.println(floatVal); // 0.1  
        System.out.println(doubleVal); // 0.1  
        System.out.println(doubleValCopy); // 0.1000000149011612  
  
        System.out.println(floatVal == doubleVal); // false  
        System.out.println(doubleVal == doubleValCopy); // false  
    }  
}
```

Java中用于float和double类型的浮点数表示具有有限的精度位数。对于float类型，精度为23个二进制位，约等于8个十进制位。对于double类型，精度为52位，约等于15个十进制位。此外，某些算术运算会引入舍入误差。

因此，当程序比较浮点数值时，通常的做法是为比较定义一个可接受的误差范围（acceptable delta）。如果两个数的差值小于该误差范围，则认为它们相等。例如

```
if (Math.abs(v1 - v2) < delta)
```

误差比较示例：

```
public class DeltaCompareExample {
```

Chapter 82: Java Floating Point Operations

Floating-point numbers are numbers that have fractional parts (usually expressed with a decimal point). In Java, there are two primitive types for floating-point numbers which are **float** (uses 4 bytes), and **double** (uses 8 bytes). This documentation page is for detailing with examples operations that can be done on floating points in Java.

Section 82.1: Comparing floating point values

You should be careful when comparing floating-point values (**float** or **double**) using relational operators: ==, !=, < and so on. These operators give results according to the binary representations of the floating point values. For example:

```
public class CompareTest {  
    public static void main(String[] args) {  
        double oneThird = 1.0 / 3.0;  
        double one = oneThird * 3;  
        System.out.println(one == 1.0); // prints "false"  
    }  
}
```

The calculation oneThird has introduced a tiny rounding error, and when we multiply oneThird by 3 we get a result that is slightly different to 1.0.

This problem of inexact representations is more stark when we attempt to mix **double** and **float** in calculations. For example:

```
public class CompareTest2 {  
    public static void main(String[] args) {  
        float floatVal = 0.1f;  
        double doubleVal = 0.1;  
        double doubleValCopy = floatVal;  
  
        System.out.println(floatVal); // 0.1  
        System.out.println(doubleVal); // 0.1  
        System.out.println(doubleValCopy); // 0.1000000149011612  
  
        System.out.println(floatVal == doubleVal); // false  
        System.out.println(doubleVal == doubleValCopy); // false  
    }  
}
```

The floating point representations used in Java for the **float** and **double** types have limited number of digits of precision. For the **float** type, the precision is 23 binary digits or about 8 decimal digits. For the **double** type, it is 52 bits or about 15 decimal digits. On top of that, some arithmetical operations will introduce rounding errors. Therefore, when a program compares floating point values, it standard practice to define an **acceptable delta** for the comparison. If the difference between the two numbers is less than the delta, they are deemed to be equal. For example

```
if (Math.abs(v1 - v2) < delta)
```

Delta compare example:

```
public class DeltaCompareExample {
```

```

private static boolean deltaCompare(double v1, double v2, double delta) {
    // 当v1和v2的差值小于delta时返回true
    return Math.abs(v1 - v2) < delta;
}

public static void main(String[] args) {
    double[] doubles = {1.0, 1.0001, 1.000001, 1.00000001, 1.00000000001};
    double[] deltas = {0.01, 0.0001, 0.000001, 0.00000001, 0};

    // 遍历上面初始化的所有误差范围
    for (int j = 0; j < deltas.length; j++) {
        double delta = deltas[j];
        System.out.println("delta: " + delta);

        // 遍历上面初始化的所有双精度数
        for (int i = 0; i < doubles.length - 1; i++) {
            double d1 = doubles[i];
            double d2 = doubles[i + 1];
            boolean result = deltaCompare(d1, d2, delta);

            System.out.println(" " + d1 + " == " + d2 + " ? " + result);
        }
        System.out.println();
    }
}

```

结果：

```

delta: 0.01
1.0 == 1.0001 ? true
1.0001 == 1.000001 ? true
1.000001 == 1.00000001 ? true
1.00000001 == 1.00000000001 ? true

delta: 1.0E-5
1.0 == 1.0001 ? false
1.0001 == 1.000001 ? false
1.000001 == 1.00000001 ? true
1.00000001 == 1.00000000001 ? true

delta: 1.0E-7
1.0 == 1.0001 ? false
1.0001 == 1.000001 ? false
1.000001 == 1.00000001 ? true
1.00000001 == 1.00000000001 ? true

delta: 1.0E-10
1.0 == 1.0001 ? false
1.0001 == 1.000001 ? false
1.000001 == 1.00000001 ? false
1.00000001 == 1.00000000001 ? false

delta: 0.0
1.0 == 1.0001 ? false
1.0001 == 1.000001 ? false
1.000001 == 1.00000001 ? false
1.00000001 == 1.00000000001 ? false

```

```

private static boolean deltaCompare(double v1, double v2, double delta) {
    // return true iff the difference between v1 and v2 is less than delta
    return Math.abs(v1 - v2) < delta;
}

public static void main(String[] args) {
    double[] doubles = {1.0, 1.0001, 1.000001, 1.00000001, 1.00000000001};
    double[] deltas = {0.01, 0.0001, 0.000001, 0.00000001, 0};

    // loop through all of deltas initialized above
    for (int j = 0; j < deltas.length; j++) {
        double delta = deltas[j];
        System.out.println("delta: " + delta);

        // loop through all of the doubles initialized above
        for (int i = 0; i < doubles.length - 1; i++) {
            double d1 = doubles[i];
            double d2 = doubles[i + 1];
            boolean result = deltaCompare(d1, d2, delta);

            System.out.println(" " + d1 + " == " + d2 + " ? " + result);
        }
        System.out.println();
    }
}

```

Result:

```

delta: 0.01
1.0 == 1.0001 ? true
1.0001 == 1.000001 ? true
1.000001 == 1.00000001 ? true
1.00000001 == 1.00000000001 ? true

delta: 1.0E-5
1.0 == 1.0001 ? false
1.0001 == 1.000001 ? false
1.000001 == 1.00000001 ? true
1.00000001 == 1.00000000001 ? true

delta: 1.0E-7
1.0 == 1.0001 ? false
1.0001 == 1.000001 ? false
1.000001 == 1.00000001 ? true
1.00000001 == 1.00000000001 ? true

delta: 1.0E-10
1.0 == 1.0001 ? false
1.0001 == 1.000001 ? false
1.000001 == 1.00000001 ? false
1.00000001 == 1.00000000001 ? false

delta: 0.0
1.0 == 1.0001 ? false
1.0001 == 1.000001 ? false
1.000001 == 1.00000001 ? false
1.00000001 == 1.00000000001 ? false

```

对于比较 double 和 float 原始类型，也可以使用对应装箱类型的静态 compare 方法。例如：

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b)); // -1
System.out.println(Double.compare(b, a)); // 1
```

最后，确定哪些差值最适合比较可能很棘手。一种常用的方法是选择我们直觉认为合适的差值。然而，如果你知道输入值的量级和（真实）精度，以及所执行的计算，可能有办法得出结果精度的数学合理界限，从而确定差值。（有一个正式的数学分支称为

数值分析，过去曾教授给计算科学家，涵盖这类分析。）

第82.2节：溢出和下溢

浮点数数据类型

float数据类型是单精度32位IEEE 754浮点数。

浮点数溢出

最大可能值是3.4028235e+38，超过此值时会产生无穷大（Infinity）

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); // Infinity
```

浮点数下溢

最小值是1.4e-45f，当低于此值时会产生0.0

```
float f = 1e-45f;
float result = f/1000;
System.out.println(result);
```

double 数据类型

double数据类型是一个双精度64位IEEE 754浮点数。

Double 溢出

最大可能值是1.7976931348623157e+308，当超过此值时会产生Infinity

```
double d = 1e308;
double result=d*2;
System.out.println(result); // Infinity
```

Double 下溢

最小值是4.9e-324，当低于此值时会产生0.0

```
double d = 4.8e-323;
double result = d/1000;
```

Also for comparison of **double** and **float** primitive types static compare method of corresponding boxing type can be used. For example:

```
double a = 1.0;
double b = 1.0001;

System.out.println(Double.compare(a, b)); // -1
System.out.println(Double.compare(b, a)); // 1
```

Finally, determining what deltas are most appropriate for a comparison can be tricky. A commonly used approach is to pick delta values that are our intuition says are about right. However, if you know scale and (true) accuracy of the input values, and the calculations performed, it may be possible to come up with mathematically sound bounds on the accuracy of the results, and hence for the deltas. (There is a formal branch of Mathematics known as Numerical Analysis that used to be taught to computational scientists that covered this kind of analysis.)

Section 82.2: OverFlow and UnderFlow

Float data type

The float data type is a single-precision 32-bit IEEE 754 floating point.

Float overflow

Maximum possible value is 3.4028235e+38 , When it exceeds this value it produces Infinity

```
float f = 3.4e38f;
float result = f*2;
System.out.println(result); // Infinity
```

Float UnderFlow

Minimum value is 1.4e-45f, when is goes below this value it produces 0.0

```
float f = 1e-45f;
float result = f/1000;
System.out.println(result);
```

double data type

The double data type is a double-precision 64-bit IEEE 754 floating point.

Double OverFlow

Maximum possible value is 1.7976931348623157e+308 , When it exceeds this value it produces Infinity

```
double d = 1e308;
double result=d*2;
System.out.println(result); // Infinity
```

Double UnderFlow

Minimum value is 4.9e-324, when is goes below this value it produces 0.0

```
double d = 4.8e-323;
double result = d/1000;
```

```
System.out.println(result); //0.0
```

第82.3节：格式化浮点数值

浮点数可以使用String.format和'f'标志格式化为十进制数

```
//小数部分保留两位并四舍五入
String format1 = String.format("%.2f", 1.2399);
System.out.println(format1); // "1.24"

//小数部分保留三位并四舍五入
String format2 = String.format("%.3f", 1.2399);
System.out.println(format2); // "1.240"

//四舍五入到两位，小数部分不足补零
String format3 = String.format("%.2f", 1.2);
System.out.println(format3); // 返回 "1.20"

// 四舍五入到两位小数
String format4 = String.format("%.2f", 3.19999);
System.out.println(format4); // "3.20"
```

浮点数可以使用DecimalFormat格式化为十进制数字

```
// 四舍五入保留一位小数
String format = new DecimalFormat("0.#").format(4.3200);
System.out.println(format); // 4.3

// 四舍五入保留两位小数
String format = new DecimalFormat("0.##").format(1.2323000);
System.out.println(format); //1.23

// 将浮点数格式化为十进制数字
double dv = 123456789;
System.out.println(dv); // 1.23456789E8
String format = new DecimalFormat("0").format(dv);
System.out.println(format); //123456789
```

第82.4节：严格遵守IEEE规范

默认情况下，float 和 double 类型的浮点运算并不严格遵守 IEEE 754 规范的规则。表达式允许使用实现特定的扩展来扩大这些值的范围；本质上允许它们比规范要求的更精确。

strictfp 禁用此行为。它可以应用于类、接口或方法，并作用于其中包含的所有内容，例如类、接口、方法、构造函数、变量初始化器等。使用 strictfp 时，浮点表达式的中间值必须位于 float 值集或 double 值集内。这使得此类表达式的结果完全符合 IEEE 754 规范的预测。

所有常量表达式隐式为严格，即使它们不在 strictfp 范围内。

因此，strictfp 的净效果有时会使某些边缘情况的计算不那么精确，并且可能使浮点运算更慢（因为 CPU 现在需要做更多工作以确保任何本地额外精度不会影响结果）。然而，它也使得结果在所有平台上完全相同。因此，它在科学程序等需要可重复性胜过速度的场景中非常有用。

```
public class StrictFP { // 无 strictfp -> 默认宽松
```

```
System.out.println(result); //0.0
```

Section 82.3: Formatting the floating point values

Floating point Numbers can be formatted as a decimal number using `String.format` with 'f' flag

```
//Two digits in fractional part are rounded
String format1 = String.format("%.2f", 1.2399);
System.out.println(format1); // "1.24"

// three digits in fractional part are rounded
String format2 = String.format("%.3f", 1.2399);
System.out.println(format2); // "1.240"

//rounded to two digits, filled with zero
String format3 = String.format("%.2f", 1.2);
System.out.println(format3); // returns "1.20"

//rounder to two digits
String format4 = String.format("%.2f", 3.19999);
System.out.println(format4); // "3.20"
```

Floating point Numbers can be formatted as a decimal number using `DecimalFormat`

```
// rounded with one digit fractional part
String format = new DecimalFormat("0.#").format(4.3200);
System.out.println(format); // 4.3

// rounded with two digit fractional part
String format = new DecimalFormat("0.##").format(1.2323000);
System.out.println(format); //1.23

// formatting floating numbers to decimal number
double dv = 123456789;
System.out.println(dv); // 1.23456789E8
String format = new DecimalFormat("0").format(dv);
System.out.println(format); //123456789
```

Section 82.4: Strict Adherence to the IEEE Specification

By default, floating point operations on `float` and `double` do not strictly adhere to the rules of the IEEE 754 specification. An expression is allowed to use implementation-specific extensions to the range of these values; essentially allowing them to be *more* accurate than required.

strictfp disables this behavior. It is applied to a class, interface, or method, and applies to everything contained in it, such as classes, interfaces, methods, constructors, variable initializers, etc. With **strictfp**, the intermediate values of a floating-point expression *must* be within the float value set or the double value set. This causes the results of such expressions to be exactly those that the IEEE 754 specification predicts.

All constant expressions are implicitly strict, even if they aren't inside a **strictfp** scope.

Therefore, **strictfp** has the net effect of sometimes making certain corner case computations *less* accurate, and can also make floating point operations *slower* (as the CPU is now doing more work to ensure any native extra precision does not affect the result). However, it also causes the results to be exactly the same on all platforms. It is therefore useful in things like scientific programs, where reproducibility is more important than speed.

```
public class StrictFP { // No strictfp -> default lenient
```

```
public strictfp float strict(float input) {
    return input * input / 3.4f; // 严格遵守规范。
        // 可能不那么精确且可能更慢。
}
```

```
public float lenient(float input) {
    return input * input / 3.4f; // 有时可能更精确且更快,
        // 但结果可能不可复现。
}
```

```
public static final strictfp class Ops { // strictfp 影响所有包含的实体
    private StrictOps() {}

    public static div(double dividend, double divisor) { // 隐式 strictfp
        return dividend / divisor;
    }
}
```

```
public strictfp float strict(float input) {
    return input * input / 3.4f; // Strictly adheres to the spec.
        // May be less accurate and may be slower.
}
```

```
public float lenient(float input) {
    return input * input / 3.4f; // Can sometimes be more accurate and faster,
        // but results may not be reproducible.
}
```

```
public static final strictfp class Ops { // strictfp affects all enclosed entities
    private StrictOps() {}

    public static div(double dividend, double divisor) { // implicitly strictfp
        return dividend / divisor;
    }
}
```

第83章：货币与金钱

第83.1节：添加自定义货币

类路径中需要的JAR包：

- javax.money:money-api:1.0 (JSR354 货币和货币API)
- org.javamoney:moneta:1.0 (参考实现)
- javax.annotation-api:1.2. (参考实现使用的通用注解)

```
// 让我们创建非ISO货币，比如比特币

// 一开始，这会抛出 UnknownCurrencyException
MonetaryAmount moneys = Money.of(new BigDecimal("0.1"), "BTC");

// 这是因为比特币对默认货币
// 提供者来说是未知的
System.out.println(Monetary.isCurrencyAvailable("BTC")); // false

// 我们将使用 org.javamoney.moneta 提供的 CurrencyUnitBuilder 构建新货币
CurrencyUnit bitcoin = CurrencyUnitBuilder
.of("BTC", "BtcCurrencyProvider") // 设置货币代码和货币提供者名称
.setDefaultFractionDigits(2) // 设置默认小数位数
.build(true); // 构建新的货币单位。这里 'true' 表示
// 货币单位将被注册并
// 在默认货币环境中可用

// 现在 BTC 可用
System.out.println(Monetary.isCurrencyAvailable("BTC")); // 返回 true
```

Chapter 83: Currency and Money

Section 83.1: Add custom currency

Required JARs on classpath:

- javax.money:money-api:1.0 (JSR354 money and currency api)
- org.javamoney:moneta:1.0 (Reference implementation)
- javax.annotation-api:1.2. (Common annotations used by reference implementation)

```
// Let's create non-ISO currency, such as bitcoin

// At first, this will throw UnknownCurrencyException
MonetaryAmount moneys = Money.of(new BigDecimal("0.1"), "BTC");

// This happens because bitcoin is unknown to default currency
// providers
System.out.println(Monetary.isCurrencyAvailable("BTC")); // false

// We will build new currency using CurrencyUnitBuilder provided by org.javamoney.moneta
CurrencyUnit bitcoin = CurrencyUnitBuilder
.of("BTC", "BtcCurrencyProvider") // Set currency code and currency provider name
.setDefaultFractionDigits(2) // Set default fraction digits
.build(true); // Build new currency unit. Here 'true' means
// currency unit is to be registered and
// accessible within default monetary context

// Now BTC is available
System.out.println(Monetary.isCurrencyAvailable("BTC")); // True
```

第84章：对象克隆

第84.1节：执行深拷贝的克隆

要复制嵌套对象，必须执行深拷贝，如本例所示。

```
import java.util.ArrayList;
import java.util.List;

public class 羊 implements Cloneable {

    private String 名字;
    private int 体重;
    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        Sheep clone = (Sheep) super.clone();
        if (children != null) {
            // 对子对象进行深拷贝
            List<Sheep> cloneChildren = new ArrayList<>(children.size());
            for (Sheep child : children) {
                cloneChildren.add((Sheep) child.clone());
            }
            clone.setChildren(cloneChildren);
        }
        return clone;
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }
}

import java.util.Arrays;
import java.util.List;

// 创建一只羊
Sheep sheep = new Sheep("Dolly", 20);

// 创建子对象
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// 克隆羊对象
Sheep dolly = (Sheep) sheep.clone();
```

Chapter 84: Object Cloning

Section 84.1: Cloning performing a deep copy

To copy nested objects, a [deep copy](#) must be performed, as shown in this example.

```
import java.util.ArrayList;
import java.util.List;

public class Sheep implements Cloneable {

    private String name;
    private int weight;
    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        Sheep clone = (Sheep) super.clone();
        if (children != null) {
            // make a deep copy of the children
            List<Sheep> cloneChildren = new ArrayList<>(children.size());
            for (Sheep child : children) {
                cloneChildren.add((Sheep) child.clone());
            }
            clone.setChildren(cloneChildren);
        }
        return clone;
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }
}

import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly = (Sheep) sheep.clone();
```

```

List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // 输出 false, 两个数组都包含对象的副本
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}

```

第84.2节：使用复制工厂进行克隆

```

public class Sheep {

    private String 名字;

    private int 体重;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public static Sheep newInstance(Sheep other);
        return new Sheep(other.name, other.weight)
    }

}

```

第84.3节：使用复制构造函数进行克隆

克隆对象的一种简单方法是实现复制构造函数。

```

public class Sheep {

    private String 名字;

    private int 体重;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    // 复制构造函数
    // 将other的字段复制到新对象中
    public Sheep(Sheep other) {
        this.name = other.name;
        this.weight = other.weight;
    }

}

// 创建一只羊
Sheep sheep = new Sheep("Dolly", 20);
// 克隆羊对象
Sheep dolly = new Sheep(sheep); // dolly.name 是 "Dolly", dolly.weight 是 20

```

第84.4节：通过实现Clonable接口进行克隆

通过实现Cloneable接口克隆对象。

```

List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints false, both arrays contain copies of the objects inside
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}

```

Section 84.2: Cloning using a copy factory

```

public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public static Sheep newInstance(Sheep other);
        return new Sheep(other.name, other.weight)
    }

}

```

Section 84.3: Cloning using a copy constructor

An easy way to clone an object is by implementing a copy constructor.

```

public class Sheep {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    // copy constructor
    // copies the fields of other into the new object
    public Sheep(Sheep other) {
        this.name = other.name;
        this.weight = other.weight;
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly = new Sheep(sheep); // dolly.name is "Dolly" and dolly.weight is 20

```

Section 84.4: Cloning by implementing Clonable interface

Cloning an object by implementing the [Cloneable](#) interface.

```

public class 羊 implements Cloneable {

    private String 名字;

    private int 体重;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}

// 创建一只羊
Sheep sheep = new Sheep("Dolly", 20);
// 克隆羊对象
羊 dolly = (羊)sheep.clone(); // dolly.name 是 "Dolly", dolly.weight 是 20

```

第84.5节：执行浅拷贝的克隆

克隆对象时的默认行为是对对象的字段执行浅拷贝。在这种情况下，原始对象和克隆对象都持有对相同对象的引用。

此示例展示了该行为。

```

import java.util.List;

public class 羊 implements Cloneable {

    private String 名字;

    private int 体重;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }

}

```

```

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

}

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);
// clone the sheep
Sheep dolly = (Sheep)sheep.clone(); // dolly.name is "Dolly" and dolly.weight is 20

```

Section 84.5: Cloning performing a shallow copy

Default behavior when cloning an object is to perform a [shallow copy](#) of the object's fields. In that case, both the original object and the cloned object, hold references to the same objects.

This example shows that behavior.

```

import java.util.List;

public class Sheep implements Cloneable {

    private String name;

    private int weight;

    private List<Sheep> children;

    public Sheep(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public List<Sheep> getChildren() {
        return children;
    }

    public void setChildren(List<Sheep> children) {
        this.children = children;
    }

}

```

```
import java.util.Arrays;
import java.util.List;

// 创建一只羊
Sheep sheep = new Sheep("Dolly", 20);

// 创建子对象
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// 克隆羊对象
Sheep dolly = (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // 输出 true, 两个数组包含相同的对象
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}
```

```
import java.util.Arrays;
import java.util.List;

// create a sheep
Sheep sheep = new Sheep("Dolly", 20);

// create children
Sheep child1 = new Sheep("Child1", 4);
Sheep child2 = new Sheep("Child2", 5);

sheep.setChildren(Arrays.asList(child1, child2));

// clone the sheep
Sheep dolly = (Sheep) sheep.clone();
List<Sheep> sheepChildren = sheep.getChildren();
List<Sheep> dollysChildren = dolly.getChildren();
for (int i = 0; i < sheepChildren.size(); i++) {
    // prints true, both arrays contain the same objects
    System.out.println(sheepChildren.get(i) == dollysChildren.get(i));
}
```

第85章：递归

递归是指一个方法调用自身。这样的一个方法称为**递归方法**。递归方法可能比等效的非递归方法更简洁。然而，对于深度递归，有时迭代解决方案可以消耗更少的线程有限的栈空间。

本主题包括Java中递归的示例。

第85.1节：递归的基本思想

什么是递归：

一般来说，递归是指函数直接或间接调用自身。例如：

```
// 该方法“无限”调用自身
public void useless() {
    useless(); // 方法直接调用自身
}
```

应用递归解决问题的条件：

使用递归函数解决特定问题有两个前提条件：

1. 问题必须有一个基本条件，作为递归的终点。当递归函数达到基本条件时，不会再进行更深层的递归调用。
2. 递归的每一层都应尝试解决一个更小的问题。递归函数因此将问题划分为越来越小的部分。假设问题是有限的，这将确保递归终止。

在Java中还有第三个前提条件：解决问题时不应需要过深的递归；参见“Java中深度递归的问题”。

示例

下面的函数使用递归计算阶乘。注意方法factorial如何在函数内部调用自身。每次调用自身时，参数 n 减少1。当 n 达到1（基本条件）时，函数将不再进行更深的递归。

```
public int factorial(int n) {
    if (n <= 1) { // 基本条件
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

这不是在Java中计算阶乘的实用方法，因为它没有考虑整数溢出，或者对于较大的 n值会导致调用栈溢出（即StackOverflowError异常）。

第85.2节：Java中的深度递归问题

考虑以下使用递归添加两个正数的简单方法：

Chapter 85: Recursion

Recursion occurs when a method calls itself. Such a method is called **recursive**. A recursive method may be more concise than an equivalent non-recursive approach. However, for deep recursion, sometimes an iterative solution can consume less of a thread's finite stack space.

This topic includes examples of recursion in Java.

Section 85.1: The basic idea of recursion

What is recursion:

In general, recursion is when a function invokes itself, either directly or indirectly. For example:

```
// This method calls itself "infinitely"
public void useless() {
    useless(); // method calls itself (directly)
}
```

Conditions for applying recursion to a problem:

There are two preconditions for using recursive functions to solving a specific problem:

1. There must be a base condition for the problem, which will be the endpoint for the recursion. When a recursive function reaches the base condition, it makes no further (deeper) recursive calls.
2. Each level of recursion should be attempting a smaller problem. The recursive function thus divides the problem into smaller and smaller parts. Assuming that the problem is finite, this will ensure that the recursion terminates.

In Java there is a third precondition: it should not be necessary to recurse too deeply to solve the problem; see Deep recursion is problematic in Java

Example

The following function calculates factorials using recursion. Notice how the method factorial calls itself within the function. Each time it calls itself, it reduces the parameter n by 1. When n reaches 1 (the base condition) the function will recurse no deeper.

```
public int factorial(int n) {
    if (n <= 1) { // the base condition
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

This is not a practical way of computing factorials in Java, since it does not take account of integer overflow, or call stack overflow (i.e. `StackOverflowError` exceptions) for large values of n.

Section 85.2: Deep recursion is problematic in Java

Consider the following naive method for adding two positive numbers using recursion:

```

public static int add(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1); // 尾调用
    }
}

```

该算法在逻辑上是正确的，但存在一个重大问题。如果你用较大的 a 调用 add，它将在任何版本的 Java（至少到 Java 9）中崩溃并抛出 StackOverflowError 异常。

在典型的函数式编程语言（以及许多其他语言）中，编译器会优化尾递归。编译器会注意到对 add 的调用（在标记行处）是一个尾调用，并会有效地将递归重写为循环。这种转换称为尾调用消除。

然而，当前一代的 Java 编译器并不执行尾调用消除。（这并非简单的疏忽。这背后有实质性的技术原因；详见下文。）相反，每次对 add 的递归调用都会在线程的栈上分配一个新的栈帧。例如，如果你调用 add(1000, 1)，计算结果 1001 需要 1000 次递归调用。

问题在于 Java 线程栈的大小在创建线程时是固定的。（这包括单线程程序中的“主”线程。）如果分配了过多的栈帧，栈就会溢出。JVM 会检测到这一点并抛出 StackOverflowError。

解决这个问题的一种方法是简单地使用更大的栈。JVM 有选项可以控制栈的默认大小，你也可以在 Thread 构造函数参数中指定栈大小。不幸的是，这只是“推迟”了栈溢出。如果你需要进行更大栈空间的计算，

StackOverflowError 仍然会出现。

真正的解决方案是识别那些可能出现深度递归的递归算法，并在源代码层面手动执行尾调用优化。例如，我们的 add 方法可以重写如下：

```

public static int add(int a, int b) {
    while (a != 0) {
        a = a - 1;
        b = b + 1;
    }
    return b;
}

```

（显然，有更好的方法来相加两个整数。上述仅用于说明手动尾调用消除的效果。）

为什么 Java 尚未实现尾调用消除

添加尾调用消除到 Java 中并不容易，原因有很多。例如：

- 有些代码可能依赖于 StackOverflowError 来（例如）限制计算问题的规模。
- 沙箱安全管理器通常依赖于分析调用栈来决定是否允许非特权代码执行特权操作。

正如约翰·罗斯在“[虚拟机中的尾调用](#)”中解释的：

“移除调用者”的栈帧的效果对某些 API 是可见的，特别是访问控制检查和

```

public static int add(int a, int b) {
    if (a == 0) {
        return b;
    } else {
        return add(a - 1, b + 1); // TAIL CALL
    }
}

```

This is algorithmically correct, but it has a major problem. If you call add with a large a, it will crash with a StackOverflowError, on any version of Java up to (at least) Java 9.

In a typical functional programming language (and many other languages) the compiler optimizes tail recursion. The compiler would notice that the call to add (at the tagged line) is a [tail call](#), and would effectively rewrite the recursion as a loop. This transformation is called tail-call elimination.

However, current generation Java compilers do not perform tail call elimination. (This is not a simple oversight. There are substantial technical reasons for this; see below.) Instead, each recursive call of add causes a new frame to be allocated on the thread's stack. For example, if you call add(1000, 1), it will take 1000 recursive calls to arrive at the answer 1001.

The problem is that the size of Java thread stack is fixed when the thread is created. (This includes the “main” thread in a single-threaded program.) If too many stack frames are allocated the stack will overflow. The JVM will detect this and throw a StackOverflowError.

One approach to dealing with this is to simply use a bigger stack. There are JVM options that control the default size of a stack, and you can also specify the stack size as a [Thread](#) constructor parameter. Unfortunately, this only “puts off” the stack overflow. If you need to do a computation that requires an even larger stack, then the StackOverflowError comes back.

The real solution is to identify recursive algorithms where deep recursion is likely, and *manually* perform the tail-call optimization at the source code level. For example, our add method can be rewritten as follows:

```

public static int add(int a, int b) {
    while (a != 0) {
        a = a - 1;
        b = b + 1;
    }
    return b;
}

```

（Obviously, there are better ways to add two integers. The above is simply to illustrate the effect of manual tail-call elimination.）

Why tail-call elimination is not implemented in Java (yet)

There are a number of reasons why adding tail call elimination to Java is not easy. For example:

- Some code could rely on StackOverflowError to (for example) place a bound on the size of a computational problem.
- Sandbox security managers often rely on analyzing the call stack when deciding whether to allow non-privileged code to perform a privileged action.

As John Rose explains in [“Tail calls in the VM”](#):

“The effects of removing the caller's stack frame are visible to some APIs, notably access control checks and

栈跟踪。就好像调用者的调用者直接调用了被调用者。调用者拥有的任何权限在控制权转移给被调用者后都会被丢弃。然而，被调用方法的链接和可访问性是在控制权转移之前计算的，并且考虑了尾调用的调用者。”

stack tracing. It is as if the caller's caller had directly called the callee. Any privileges possessed by the caller are discarded after control is transferred to the callee. However, the linkage and accessibility of the callee method are computed before the transfer of control, and take into account the tail-calling caller."

换句话说，尾调用消除可能导致访问控制方法错误地认为安全敏感的API是由受信任的代码调用的。

第85.3节：递归类型

递归可以根据递归方法调用的位置分为头递归或尾递归。

在头递归中，递归调用发生在函数中的其他处理之前（可以理解为发生在函数的顶部或头部）。

在尾递归中，情况正好相反——处理发生在递归调用之前。选择这两种递归风格看似随意，但选择可能带来完全不同的效果。

路径上在开始处只有一个递归调用的函数使用的是所谓的头递归。

前面示例中的阶乘函数使用的是头递归。它一旦确定需要递归，首先做的事情就是用递减的参数调用自身。路径末尾只有一个递归调用的函数使用的是尾递归。

```
public void tail(int n)           public void head(int n)
{
    if(n == 1)                   {
        return;
    } else
        System.out.println(n);
    tail(n-1);                  System.out.println(n);
}
```

如果递归调用发生在方法的末尾，则称为尾递归。尾递归类似于一个循环。该方法在跳转到下一个递归调用之前执行所有语句。

如果递归调用发生在方法的开头，则称为头递归。该方法在跳转到下一个递归调用之前保存状态。

参考：[头递归与尾递归的区别](#)

第85.4节：计算第N个斐波那契数

下面的方法使用递归计算第N个斐波那契数。

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

该方法实现了一个基准情况 ($n \leq 2$) 和一个递归情况 ($n > 2$)。这说明了使用递归来计算递归关系。

In other words, tail-call elimination could cause an access control method to mistakenly think that a security sensitive API was being called by trusted code.

Section 85.3: Types of Recursion

Recursion can be categorized as either **Head Recursion** or **Tail Recursion**, depending on where the recursive method call is placed.

In **head recursion**, the recursive call, when it happens, comes before other processing in the function (think of it happening at the top, or head, of the function).

In **tail recursion**, it's the opposite—the processing occurs before the recursive call. Choosing between the two recursive styles may seem arbitrary, but the choice can make all the difference.

A function with a path with a single recursive call at the beginning of the path uses what is called head recursion. The factorial function of a previous exhibit uses head recursion. The first thing it does once it determines that recursion is needed is to call itself with the decremented parameter. A function with a single recursive call at the end of a path is using tail recursion.

```
public void tail(int n)           public void head(int n)
{
    if(n == 1)                   {
        return;
    } else
        System.out.println(n);
    tail(n-1);                  System.out.println(n);
}
```

If the recursive call occurs at the end of a method, it is called a tail recursion. The tail recursion is similar to a loop. The method executes all the statements before jumping into the next recursive call.

If the recursive call occurs at the beginning of a method, it is called a head recursion. The method saves the state before jumping into the next recursive call.

Reference: [The difference between head & tail recursion](#)

Section 85.4: Computing the Nth Fibonacci Number

The following method computes the Nth Fibonacci number using recursion.

```
public int fib(final int n) {
    if (n > 2) {
        return fib(n - 2) + fib(n - 1);
    }
    return 1;
}
```

The method implements a base case ($n \leq 2$) and a recursive case ($n > 2$). This illustrates the use of recursion to compute a recursive relation.

然而，虽然这个例子具有说明性，但效率低下：方法的每个实例都会调用函数自身两次，导致随着N的增加，函数调用次数呈指数增长。
上述函数的时间复杂度是 $O(2N)$ ，但等效的迭代解法的复杂度为 $O(N)$ 。此外，还有一个“闭式”表达式，可以在 $O(N)$ 次浮点乘法运算内计算。

第85.5节：StackOverflowError 与递归转循环

如果递归调用“过深”，会导致StackOverflowError。Java 为其线程栈上的每个方法调用分配一个新的栈帧。然而，每个线程栈的空间是有限的。栈上帧数过多会导致栈溢出（SO）。

示例

```
public static void recursion(int depth) {  
    if (depth > 0) {  
        recursion(depth-1);  
    }  
}
```

用较大参数调用此方法（例如 `recursion(50000)`）很可能会导致栈溢出。具体数值取决于线程栈大小，而线程栈大小又取决于线程构造、命令行参数如 `-Xss`，或 JVM 的默认大小。

解决方法

递归可以通过将每次递归调用的数据存储在数据结构中转换为循环。该数据结构可以存储在堆上，而不是线程栈上。

通常，恢复方法调用状态所需的数据可以存储在栈中，并且可以使用 `while` 循环来“模拟”递归调用。可能需要的数据包括：

- 调用该方法的对象（仅限实例方法）
- 方法参数
- 局部变量
- 执行或方法中的当前位置

示例

以下类允许递归打印树结构，直到指定深度。

```
public class Node {  
  
    public int data;  
    public Node left;  
    public Node right;  
  
    public Node(int data) {  
        this(data, null, null);  
    }  
  
    public Node(int data, Node left, Node right) {  
        this.data = data;  
        this.left = left;  
        this.right = right;  
    }  
  
    public void print(final int maxDepth) {  
        if (maxDepth <= 0) {  
            return;  
        }  
        System.out.println(data);  
        if (left != null) {  
            left.print(maxDepth - 1);  
        }  
        if (right != null) {  
            right.print(maxDepth - 1);  
        }  
    }  
}
```

However, while this example is illustrative, it is also inefficient: each single instance of the method will call the function itself twice, leading to an exponential growth in the number of times the function is called as N increases. The above function is $O(2N)$, but an equivalent iterative solution has complexity $O(N)$. In addition, there is a "closed form" expression that can be evaluated in $O(N)$ floating-point multiplications.

Section 85.5: StackOverflowError & recursion to loop

If a recursive call goes "too deep", this results in a [StackOverflowError](#). Java allocates a new frame for every method call on its thread's stack. However, the space of each thread's stack is limited. Too many frames on the stack leads to the Stack Overflow (SO).

Example

```
public static void recursion(int depth) {  
    if (depth > 0) {  
        recursion(depth-1);  
    }  
}
```

Calling this method with large parameters (e.g. `recursion(50000)`) probably will result in a stack overflow. The exact value depends on the thread stack size, which in turn depends on the thread construction, command-line parameters such as `-Xss`, or the default size for the JVM.

Workaround

A recursion can be converted to a loop by storing the data for each recursive call in a data structure. This data structure can be stored on the heap rather than on the thread stack.

In general the data required to restore the state of a method invocation can be stored in a stack and a while loop can be used to "simulate" the recursive calls. Data that may be required include:

- the object the method was called for (instance methods only)
- the method parameters
- local variables
- the current position in the execution of the method

Example

The following class allows recursive of a tree structure printing up to a specified depth.

```
public class Node {  
  
    public int data;  
    public Node left;  
    public Node right;  
  
    public Node(int data) {  
        this(data, null, null);  
    }  
  
    public Node(int data, Node left, Node right) {  
        this.data = data;  
        this.left = left;  
        this.right = right;  
    }  
  
    public void print(final int maxDepth) {  
        if (maxDepth <= 0) {  
            return;  
        }  
        System.out.println(data);  
        if (left != null) {  
            left.print(maxDepth - 1);  
        }  
        if (right != null) {  
            right.print(maxDepth - 1);  
        }  
    }  
}
```

```

        System.out.print("(...)");
    } else {
        System.out.print("(");
        if (left != null) {
            left.print(maxDepth-1);
        }
        System.out.print(data);
        if (right != null) {
            right.print(maxDepth-1);
        }
        System.out.print(")");
    }
}

```

例如

```

节点 n = 新建 节点(10, 新建 节点(20, 新建 节点(50), 新建 节点(1)), 新建 节点(30, 新建 节点(42), 空));
n.print(2);
System.out.println();

```

打印

```
((...))20(...))10((...))30))
```

这可以转换为以下循环：

```

public class Frame {

    public final Node node;

    // 0: 在打印任何内容之前
    // 1: 在打印数据之前
    // 2: 在打印 ")" 之前
    public int state = 0;
    public final int maxDepth;

    public Frame(Node node, int maxDepth) {
        this.node = node;
        this.maxDepth = maxDepth;
    }

    List<Frame> stack = new ArrayList<>();
    stack.add(new Frame(n, 2)); // 第一个帧 = 初始调用

    while (!stack.isEmpty()) {
        // 获取栈顶元素
        int index = stack.size() - 1;
        Frame frame = stack.get(index); // 获取栈顶帧
        if (frame.maxDepth <= 0) {
            // 终止情况 (递归过深)
            System.out.print("(...)");
            stack.remove(index); // 弹出栈帧
        } else {
            switch (frame.state) {
                case 0:
                    frame.state++;

```

```

        System.out.print("(...)");
    } else {
        System.out.print("(");
        if (left != null) {
            left.print(maxDepth-1);
        }
        System.out.print(data);
        if (right != null) {
            right.print(maxDepth-1);
        }
        System.out.print(")");
    }
}

```

e.g.

```

Node n = new Node(10, new Node(20, new Node(50), new Node(1)), new Node(30, new Node(42), null));
n.print(2);
System.out.println();

```

Prints

```
((...))20(...))10((...))30))
```

This could be converted to the following loop:

```

public class Frame {

    public final Node node;

    // 0: before printing anything
    // 1: before printing data
    // 2: before printing ")"
    public int state = 0;
    public final int maxDepth;

    public Frame(Node node, int maxDepth) {
        this.node = node;
        this.maxDepth = maxDepth;
    }

    List<Frame> stack = new ArrayList<>();
    stack.add(new Frame(n, 2)); // first frame = initial call

    while (!stack.isEmpty()) {
        // get topmost stack element
        int index = stack.size() - 1;
        Frame frame = stack.get(index); // get topmost frame
        if (frame.maxDepth <= 0) {
            // terminal case (too deep)
            System.out.print("(...)");
            stack.remove(index); // drop frame
        } else {
            switch (frame.state) {
                case 0:
                    frame.state++;

```

```

// 在第一次递归调用之前完成所有操作
System.out.print("(");
if (frame.node.left != null) {
    // 添加新帧 (递归调用左子节点并停止)
    stack.add(new Frame(frame.node.left, frame.maxDepth - 1));
    break;
}
case 1:
frame.state++;

// 在第二次递归调用之前完成所有操作
System.out.print(frame.node.data);
if (frame.node.right != null) {
    // 添加新帧 (递归调用右子节点并停止)
    stack.add(new Frame(frame.node.right, frame.maxDepth - 1));
    break;
}
case 2:
// 在第二次递归调用之后执行所有操作并移除帧
System.out.print(")");
stack.remove(index);
}
}
System.out.println();

```

注意：这只是通用方法的一个示例。通常你可以想出更好的方式来表示帧和/或存储帧数据。

第85.6节：计算数字的N次方

以下方法使用递归计算 num 的 exp 次幂的值：

```

public long power(final int num, final int exp) {
    if (exp == 0) {
        return 1;
    }
    if (exp == 1) {
        return num;
    }
    return num * power(num, exp - 1);
}

```

这说明了上述原则：递归方法实现了一个基例（两个情况， $n = 0$ 和 $n = 1$ ）来终止递归，以及一个递归情况来再次调用该方法。该方法的时间复杂度为 $O(N)$ ，并且可以通过尾递归优化简化为一个简单的循环。

第85.7节：使用递归遍历树数据结构

考虑如下具有3个成员的 Node 类：数据、左子节点指针和右子节点指针。

```

public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
}

```

```

// do everything done before the first recursive call
System.out.print("(");
if (frame.node.left != null) {
    // add new frame (recursive call to left and stop)
    stack.add(new Frame(frame.node.left, frame.maxDepth - 1));
    break;
}
case 1:
frame.state++;

// do everything done before the second recursive call
System.out.print(frame.node.data);
if (frame.node.right != null) {
    // add new frame (recursive call to right and stop)
    stack.add(new Frame(frame.node.right, frame.maxDepth - 1));
    break;
}
case 2:
// do everything after the second recursive call & drop frame
System.out.print(")");
stack.remove(index);
}
}
System.out.println();

```

Note: This is just an example of the general approach. Often you can come up with a much better way to represent a frame and/or store the frame data.

Section 85.6: Computing the Nth power of a number

The following method computes the value of num raised to the power of exp using recursion:

```

public long power(final int num, final int exp) {
    if (exp == 0) {
        return 1;
    }
    if (exp == 1) {
        return num;
    }
    return num * power(num, exp - 1);
}

```

This illustrates the principles mentioned above: the recursive method implements a base case ($n = 0$ and $n = 1$) that terminates the recursion, and a recursive case that calls the method again. This method is $O(N)$ and can be reduced to a simple loop using tail-call optimization.

Section 85.7: Traversing a Tree data structure with recursion

Consider the Node class having 3 members data, left child pointer and right child pointer like below.

```

public class Node {
    public int data;
    public Node left;
    public Node right;

    public Node(int data){
        this.data = data;
    }
}

```

```
}
```

我们可以像下面这样遍历通过连接多个 Node 类对象构建的树，这种遍历称为树的中序遍历。

```
public static void inOrderTraversal(Node root) {
    if (root != null) {
        inOrderTraversal(root.left); // 遍历左子树
        System.out.print(root.data + " ");
        inOrderTraversal(root.right); // 遍历右子树
    }
}
```

如上所示，使用递归我们可以遍历树数据结构而不使用任何其他数据结构，这在迭代方法中是不可能的。

第85.8节：使用递归反转字符串

下面是一个递归反转字符串的代码

```
/**
 * 仅是一个解释递归思想的代码片段
 */
public class Reverse {
    public static void main (String args[]) {
        String string = "hello world";
        System.out.println(reverse(string)); //打印 dlrow olleh
    }

    public static String reverse(String s) {
        if (s.length() == 1) {
            return s;
        }

        return reverse(s.substring(1)) + s.charAt(0);
    }
}
```

第85.9节：计算从1到N的整数和

下面的方法使用递归计算从0到N的整数和。

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

该方法的时间复杂度为O(N)，可以通过尾递归优化简化为一个简单的循环。实际上，有一个闭式表达式可以在O(1)操作内计算该和。

```
}
```

We can traverse the tree constructed by connecting multiple Node class's object like below, the traversal is called in-order traversal of tree.

```
public static void inOrderTraversal(Node root) {
    if (root != null) {
        inOrderTraversal(root.left); // traverse left sub tree
        System.out.print(root.data + " ");
        inOrderTraversal(root.right); // traverse right sub tree
    }
}
```

As demonstrated above, using **recursion** we can traverse the **tree data structure** without using any other data structure which is not possible with the **iterative** approach.

Section 85.8: Reverse a string using Recursion

Below is a recursive code to reverse a string

```
/**
 * Just a snippet to explain the idea of recursion
 */
public class Reverse {
    public static void main (String args[]) {
        String string = "hello world";
        System.out.println(reverse(string)); //prints dlrow olleh
    }

    public static String reverse(String s) {
        if (s.length() == 1) {
            return s;
        }

        return reverse(s.substring(1)) + s.charAt(0);
    }
}
```

Section 85.9: Computing the sum of integers from 1 to N

The following method computes the sum of integers from 0 to N using recursion.

```
public int sum(final int n) {
    if (n > 0) {
        return n + sum(n - 1);
    } else {
        return n;
    }
}
```

This method is O(N) and can be reduced to a simple loop using tail-call optimization. In fact there is a *closed form* expression that computes the sum in O(1) operations.

第86章：字符串的转换

第86.1节：字符串转换为其他数据类型

您可以将数字字符串转换为各种Java数字类型，方法如下：

字符串转int：

```
String number = "12";
int num = Integer.parseInt(number);
```

字符串转float：

```
String number = "12.0";
float num = Float.parseFloat(number);
```

字符串转double：

```
String double = "1.47";
double num = Double.parseDouble(double);
```

字符串转boolean：

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString); // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString); // trueBool = true
```

字符串转长整型：

```
String number = "47";
long num = Long.parseLong(number);
```

字符串转大整数：

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

字符串转大十进制数：

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

转换异常：

上述数值转换如果尝试解析一个格式不正确的字符串，或者超出目标类型的范围，都会抛出（未检查的）NumberFormatException异常。异常主题讨论了如何处理此类异常。

如果你想测试是否能解析一个字符串，可以实现一个类似如下的tryParse...方法：

Chapter 86: Converting to and from Strings

Section 86.1: Converting String to other datatypes

You can convert a **numeric** string to various Java numeric types as follows:

String to int:

```
String number = "12";
int num = Integer.parseInt(number);
```

String to float:

```
String number = "12.0";
float num = Float.parseFloat(number);
```

String to double:

```
String double = "1.47";
double num = Double.parseDouble(double);
```

String to boolean:

```
String falseString = "False";
boolean falseBool = Boolean.parseBoolean(falseString); // falseBool = false

String trueString = "True";
boolean trueBool = Boolean.parseBoolean(trueString); // trueBool = true
```

String to long:

```
String number = "47";
long num = Long.parseLong(number);
```

String to BigInteger:

```
String bigNumber = "21";
BigInteger reallyBig = new BigInteger(bigNumber);
```

String to BigDecimal:

```
String bigFraction = "17.21455";
BigDecimal reallyBig = new BigDecimal(bigFraction);
```

Conversion Exceptions:

The numeric conversions above will all throw an (unchecked) **NumberFormatException** if you attempt to parse a string that is not a suitably formatted number, or is out of range for the target type. The Exceptions topic discusses how to deal with such exceptions.

If you wanted to test that you can parse a string, you could implement a `tryParse...` method like this:

```
boolean tryParseInt (String value) {  
    try {  
        String somechar = Integer.parseInt(value);  
        return true;  
    } catch (NumberFormatException e) {  
        return false;  
    }  
}
```

然而，在解析之前立即调用这个tryParse...方法（可以说）是不好的做法。更好的方式是直接调用parse...方法并处理异常。

第86.2节：与字节的转换

要将字符串编码为字节数组，可以简单地使用String#getBytes()方法，配合任何Java运行时环境中可用的标准字符集之一：

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

解码：

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

你可以通过使用静态导入来进一步简化调用：

```
import static java.nio.charset.StandardCharsets.UTF_8;  
...  
byte[] bytes = "test".getBytes(UTF_8);
```

对于不常用的字符集，你可以用字符串来指定字符集：

```
byte[] bytes = "test".getBytes("UTF-8");
```

反向操作：

```
String testString = new String (bytes, "UTF-8");
```

这意味着您必须处理已检查的UnsupportedCharsetException异常。

以下调用将使用默认字符集。默认字符集是平台特定的，通常在Windows、Mac和Linux平台之间有所不同。

```
byte[] bytes = "test".getBytes();
```

反向操作：

```
String testString = new String(bytes);
```

请注意，这些方法可能会替换或跳过无效字符和字节。若需更精细的控制——例如验证输入——建议使用CharsetEncoder和CharsetDecoder类。

第86.3节：Base64编码/解码

有时你需要将二进制数据编码为base64编码的字符串。

```
boolean tryParseInt (String value) {  
    try {  
        String somechar = Integer.parseInt(value);  
        return true;  
    } catch (NumberFormatException e) {  
        return false;  
    }  
}
```

However, calling this tryParse... method immediately before parsing is (arguably) poor practice. It would be better to just call the parse... method and deal with the exception.

Section 86.2: Conversion to / from bytes

To encode a string into a byte array, you can simply use the [String#getBytes\(\)](#) method, with one of the standard character sets available on any Java runtime:

```
byte[] bytes = "test".getBytes(StandardCharsets.UTF_8);
```

and to decode:

```
String testString = new String(bytes, StandardCharsets.UTF_8);
```

you can further simplify the call by using a static import:

```
import static java.nio.charset.StandardCharsets.UTF_8;  
...  
byte[] bytes = "test".getBytes(UTF_8);
```

For less common character sets you can indicate the character set with a string:

```
byte[] bytes = "test".getBytes("UTF-8");
```

and the reverse:

```
String testString = new String (bytes, "UTF-8");
```

this does however mean that you have to handle the checked [UnsupportedCharsetException](#).

The following call will use the default character set. The default character set is platform specific and generally differs between Windows, Mac and Linux platforms.

```
byte[] bytes = "test".getBytes();
```

and the reverse:

```
String testString = new String(bytes);
```

Note that invalid characters and bytes may be replaced or skipped by these methods. For more control - for instance for validating input - you're encouraged to use the CharsetEncoder and CharsetDecoder classes.

Section 86.3: Base64 Encoding / Decoding

Occasionally you will find the need to encode binary data as a [base64](#)-encoded string.

为此，我们可以使用来自`javax.xml.bind`包的`DatatypeConverter`类：

```
import javax.xml.bind.DatatypeConverter;
import java.util.Arrays;

// 以字节数组形式指定的任意二进制数据
byte[] binaryData = "some arbitrary data".getBytes("UTF-8");

// 将二进制数据转换为Base64编码的字符串
String encodedData = DatatypeConverter.printBase64Binary(binaryData);
// encodedData 现在是 "c29tZSBhcmJpdHJhcNkgZGF0YQ=="

// 将Base64编码的字符串转换回字节数组
byte[] decodedData = DatatypeConverter.parseBase64Binary(encodedData);

// 断言原始数据和解码后的数据相等
assert Arrays.equals(binaryData, decodedData);
```

Apache commons-codec

另外，我们可以使用Apache commons-codec中的Base64。

```
import org.apache.commons.codec.binary.Base64;

// 你的二进制数据块，作为字节数组
byte[] blob = "someBinaryData".getBytes();

// 使用 Base64 类进行编码
String binaryAsAString = Base64.encodeBase64String(blob);

// 使用 Base64 类进行解码
byte[] blob2 = Base64.decodeBase64(binaryAsAString);

// 断言两个 blob 相等
System.out.println("Equal : " + Boolean.toString(Arrays.equals(blob, blob2)));
```

如果你在运行时检查此程序，你会看到 `someBinaryData` 编码为 `c29tZUJpbmFyeURhdGE=`，这是一个非常易于管理的 `UTF-8` 字符串对象。

版本 ≥ Java SE 8

相关详情可在 [Base64](#) 找到

```
// 带填充的编码
String encoded = Base64.getEncoder().encodeToString(someByteArray);

// 无填充编码
String encoded = Base64.getEncoder().withoutPadding().encodeToString(someByteArray);

// 解码字符串
byte[] barr = Base64.getDecoder().decode(encoded);
```

参考

第86.4节：将其他数据类型转换为字符串

For this we can use the `DatatypeConverter` class from the `javax.xml.bind` package:

```
import javax.xml.bind.DatatypeConverter;
import java.util.Arrays;

// arbitrary binary data specified as a byte array
byte[] binaryData = "some arbitrary data".getBytes("UTF-8");

// convert the binary data to the base64-encoded string
String encodedData = DatatypeConverter.printBase64Binary(binaryData);
// encodedData is now "c29tZSBhcmJpdHJhcNkgZGF0YQ=="

// convert the base64-encoded string back to a byte array
byte[] decodedData = DatatypeConverter.parseBase64Binary(encodedData);

// assert that the original data and the decoded data are equal
assert Arrays.equals(binaryData, decodedData);
```

Apache commons-codec

Alternatively, we can use Base64 from [Apache commons-codec](#).

```
import org.apache.commons.codec.binary.Base64;

// your blob of binary as a byte array
byte[] blob = "someBinaryData".getBytes();

// use the Base64 class to encode
String binaryAsAString = Base64.encodeBase64String(blob);

// use the Base64 class to decode
byte[] blob2 = Base64.decodeBase64(binaryAsAString);

// assert that the two blobs are equal
System.out.println("Equal : " + Boolean.toString(Arrays.equals(blob, blob2)));
```

If you inspect this program while running, you will see that `someBinaryData` encodes to `c29tZUJpbmFyeURhdGE=`，a very manageable `UTF-8` String object.

Version ≥ Java SE 8

Details for the same can be found at [Base64](#)

```
// encode with padding
String encoded = Base64.getEncoder().encodeToString(someByteArray);

// encode without padding
String encoded = Base64.getEncoder().withoutPadding().encodeToString(someByteArray);

// decode a String
byte[] barr = Base64.getDecoder().decode(encoded);
```

Reference

Section 86.4: Converting other datatypes to String

- 您可以使用String类的valueOf方法之一，将其他原始数据类型的值作为字符串获取。

例如：

```
int i = 42;
String string = String.valueOf(i);
//字符串现在等于 "42"。
```

此方法也针对其他数据类型进行了重载，例如float、double、boolean，甚至Object。

- 你也可以通过调用.toString方法将任何其他对象（任何类的实例）转换为字符串。为了使其输出有意义，类必须重写toString()方法。大多数标准Java库类都重写了该方法，例如Date等。

例如：

```
Foo foo = new Foo(); //任何类。
String stringifiedFoo = foo.toString();
```

这里stringifiedFoo包含了foo作为字符串的表示。

你也可以用下面这种简短的方式将任何数字类型转换为字符串。

```
int i = 10;
String str = i + "";
```

或者更简单的方式是

```
String str = 10 + "";
```

第86.5节：从输入流获取字符串

可以使用字节数组构造函数从输入流读取字符串。

```
import java.io.*;

public String readString(InputStream input) throws IOException {
    byte[] bytes = new byte[50]; // 在此处提供字符串的字节长度
    input.read(bytes);
    return new String(bytes);
}
```

这使用系统默认字符集，尽管可以指定其他字符集：

```
return new String(bytes, Charset.forName("UTF-8"));
```

- You can get the value of other primitive data types as a String using one the String class's valueOf methods.

For example:

```
int i = 42;
String string = String.valueOf(i);
//string now equals "42".
```

This method is also overloaded for other datatypes, such as **float**, **double**, **boolean**, and even **Object**.

- You can also get any other Object (any instance of any class) as a String by calling `.toString` on it. For this to give useful output, the class must override `toString()`. Most of the standard Java library classes do, such as `Date` and others.

For example:

```
Foo foo = new Foo(); //Any class.
String stringifiedFoo = foo.toString();
```

Here `stringifiedFoo` contains a representation of `foo` as a String.

You can also convert any number type to String with short notation like below.

```
int i = 10;
String str = i + "";
```

Or just simple way is

```
String str = 10 + "";
```

Section 86.5: Getting a `String` from an `InputStream`

A `String` can be read from an `InputStream` using the byte array constructor.

```
import java.io.*;

public String readString(InputStream input) throws IOException {
    byte[] bytes = new byte[50]; // supply the length of the string in bytes here
    input.read(bytes);
    return new String(bytes);
}
```

This uses the system default charset, although an alternate charset may be specified:

```
return new String(bytes, Charset.forName("UTF-8"));
```

第87章：随机数生成

第87.1节：伪随机数

Java作为utils包的一部分，提供了一个基本的伪随机数生成器，恰如其名随机。该对象可用于生成伪随机值，类型可为任何内置数值数据类型（int、float等）。你也可以用它生成随机布尔值，或随机字节数组。示例用法如下：

```
import java.util.Random;  
...  
  
Random random = new Random();  
int randInt = random.nextInt();  
long randLong = random.nextLong();  
  
double randDouble = random.nextDouble(); //返回0.0到1.0之间的值  
float randFloat = random.nextFloat(); //与nextDouble相同  
  
byte[] randBytes = new byte[16];  
random.nextBytes(randBytes); //nextBytes接收用户提供的字节数组，并用随机字节填充它。无返回值。
```

注意：该类仅生成质量较低的伪随机数，绝不应用于生成密码学操作或其他对高质量随机性要求严格的场景中的随机数（针对这些情况，应使用下文提到的SecureRandom类）。关于“安全”与“不安全”随机性的区别解释超出本示例范围。

第87.2节：特定范围内的伪随机数

方法 nextInt(int bound) 属于 Random，接受一个上限（不包含），即返回的随机值必须小于该数字。然而，只有 nextInt 方法接受边界；nextLong、nextDouble 等方法不接受。

```
Random random = new Random();  
random.nextInt(1000); // 0 - 999  
  
int number = 10 + random.nextInt(100); // number 的范围是 10 到 109
```

从 Java 1.7 开始，你也可以使用 ThreadLocalRandom ([source](#))。该类提供了线程安全的伪随机数生成器（PRNG）。注意，该类的 nextInt 方法接受上下限。

```
import java.util.concurrent.ThreadLocalRandom;  
  
// nextInt 通常不包含上限值，  
// 所以加 1 使其包含  
ThreadLocalRandom.current().nextInt(min, max + 1);
```

注意，官方文档指出当 bound 接近 $2^{31}+1$ 时，nextInt(int bound) 可能会出现异常情况（加重强调）：

该算法稍显复杂。它会拒绝导致分布不均匀的值（因为 2^{31} 不能被 n 整除）。被拒绝的概率取决于 n。
最坏情况

Chapter 87: Random Number Generation

Section 87.1: Pseudo Random Numbers

Java provides, as part of the utils package, a basic pseudo-random number generator, appropriately named `Random`. This object can be used to generate a pseudo-random value as any of the built-in numerical datatypes (`int`, `float`, etc). You can also use it to generate a random Boolean value, or a random array of bytes. An example usage is as follows:

```
import java.util.Random;  
...  
  
Random random = new Random();  
int randInt = random.nextInt();  
long randLong = random.nextLong();  
  
double randDouble = random.nextDouble(); //This returns a value between 0.0 and 1.0  
float randFloat = random.nextFloat(); //Same as nextDouble  
  
byte[] randBytes = new byte[16];  
random.nextBytes(randBytes); //nextBytes takes a user-supplied byte array, and fills it with random  
bytes. It returns nothing.
```

NOTE: This class only produces fairly low-quality pseudo-random numbers, and should never be used to generate random numbers for cryptographic operations or other situations where higher-quality randomness is critical (For that, you would want to use the `SecureRandom` class, as noted below). An explanation for the distinction between "secure" and "insecure" randomness is beyond the scope of this example.

Section 87.2: Pseudo Random Numbers in Specific Range

The method `nextInt(int bound)` of `Random` accepts an upper exclusive boundary, i.e. a number that the returned random value must be less than. However, only the `nextInt` method accepts a bound; `nextLong`, `nextDouble` etc. do not.

```
Random random = new Random();  
random.nextInt(1000); // 0 - 999  
  
int number = 10 + random.nextInt(100); // number is in the range of 10 to 109
```

Starting in Java 1.7, you may also use `ThreadLocalRandom (source)`. This class provides a thread-safe PRNG (pseudo-random number generator). Note that the `nextInt` method of this class accepts both an upper and lower bound.

```
import java.util.concurrent.ThreadLocalRandom;  
  
// nextInt is normally exclusive of the top value,  
// so add 1 to make it inclusive  
ThreadLocalRandom.current().nextInt(min, max + 1);
```

Note that [the official documentation](#) states that `nextInt(int bound)` can do weird things when bound is near $2^{31}+1$ (emphasis added):

The algorithm is slightly tricky. **It rejects values that would result in an uneven distribution** (due to the fact that 2^{31} is not divisible by n). The probability of a value being rejected depends on n. **The worst**

案例是 $n=2^{30}+1$, 拒绝的概率为 $1/2$, 循环终止前的期望迭代次数为 2。

换句话说, 指定一个边界会(稍微)降低 `nextInt` 方法的性能, 且当 `bound` 接近最大整数值的一半时, 这种性能下降会更加明显。

第87.3节：生成密码学安全的伪随机数

`Random` 和 `ThreadLocalRandom` 对日常使用来说已经足够好, 但它们有一个大问题: 它们基于一个线性同余生成器, 这是一种输出很容易被预测的算法。因此, 这两个类不适合用于密码学用途(例如密钥生成)。

在需要输出难以预测的伪随机数生成器(PRNG)的情况下, 可以使用 `java.security.SecureRandom`。预测该类实例生成的随机数非常困难, 因此该类被称为密码学安全的。

```
import java.security.SecureRandom;
import java.util.Arrays;

public class Foo {
    public static void main(String[] args) {
        SecureRandom rng = new SecureRandom();
        byte[] randomBytes = new byte[64];
        rng.nextBytes(randomBytes); // 用随机字节填充randomBytes (显而易见)
        System.out.println(Arrays.toString(randomBytes));
    }
}
```

除了密码学安全之外, `SecureRandom` 的周期长达 2^{160} 次方, 而 `Random` 的周期为 2^{48} 次方。然而, 它的一个缺点是比 `Random` 和其他线性伪随机数生成器(如 `MersenneTwister` 和 `Xorshift`)慢得多。

请注意, `SecureRandom` 的实现依赖于平台和提供者。默认的 `SecureRandom` (由 SUN 提供者在 `sun.security.provider.SecureRandom` 中给出) :

- 在类 Unix 系统上, 使用来自 `/dev/random` 和/或 `/dev/urandom` 的数据作为种子。
- 在 Windows 上, 使用 CryptoAPI 中的 `CryptGenRandom()` 调用作为种子。

第 87.4 节：使用指定种子生成随机数

```
// 创建一个种子为 12345 的 Random 实例
Random random = new Random(12345L);

// 获取一个 ThreadLocalRandom 实例
ThreadLocalRandom tlr = ThreadLocalRandom.current();

// 设置该实例的种子。
tlr.setSeed(12345L);
```

使用相同的种子生成随机数每次都会返回相同的数字, 因此如果不想得到重复的数字, 为每个 `Random` 实例设置不同的种子是个好主意。

case is $n=2^{30}+1$, for which the probability of a reject is $1/2$, and the expected number of iterations before the loop terminates is 2.

In other words, specifying a bound will (slightly) decrease the performance of the `nextInt` method, and this performance decrease will become more pronounced as the bound approaches half the max int value.

Section 87.3: Generating cryptographically secure pseudorandom numbers

`Random` 和 `ThreadLocalRandom` 对日常使用来说已经足够好, 但它们有一个大问题: 它们基于一个线性同余生成器, 这是一种输出很容易被预测的算法。因此, 这两个类不适合用于密码学用途(例如密钥生成)。

One can use `java.security.SecureRandom` in situations where a PRNG with an output that is very hard to predict is required. Predicting the random numbers created by instances of this class is hard enough to label the class as **cryptographically secure**.

```
import java.security.SecureRandom;
import java.util.Arrays;

public class Foo {
    public static void main(String[] args) {
        SecureRandom rng = new SecureRandom();
        byte[] randomBytes = new byte[64];
        rng.nextBytes(randomBytes); // Fills randomBytes with random bytes (duh)
        System.out.println(Arrays.toString(randomBytes));
    }
}
```

Besides being cryptographically secure, `SecureRandom` has a gigantic period of 2^{160} , compared to `Random`'s period of 2^{48} . It has one drawback of being considerably slower than `Random` and other linear PRNGs such as `MersenneTwister` and `Xorshift`, however.

Note that `SecureRandom` implementation is both platform and provider dependent. The default `SecureRandom` (given by SUN provider in `sun.security.provider.SecureRandom`):

- on Unix-like systems, seeded with data from `/dev/random` and/or `/dev/urandom`.
- on Windows, seeded with calls to `CryptGenRandom()` in `CryptoAPI`.

Section 87.4: Generating Random Numbers with a Specified Seed

```
//Creates a Random instance with a seed of 12345.
Random random = new Random(12345L);

//Gets a ThreadLocalRandom instance
ThreadLocalRandom tlr = ThreadLocalRandom.current();

//Set the instance's seed.
tlr.setSeed(12345L);
```

Using the same seed to generate random numbers will return the same numbers every time, so setting a different seed for every `Random` instance is a good idea if you don't want to end up with duplicate numbers.

获取每次调用都不同的Long类型值的一个好方法是System.currentTimeMillis()：

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

第87.5节：选择无重复的随机数

```
/*
 * 返回一个无重复随机数的数组
 * @param range 可能的数字范围，例如如果是100，则数字范围为1-100
 * @param length 随机数数组的长度
 * @return 无重复随机数的数组。
 */
public static int[] getRandomNumbersWithNoDuplicates(int range, int length){
    if (length < range) {
        // 这里是所有随机数的存放处
        int[] randomNumbers = new int[length];

        // 遍历所有随机数进行赋值
        for (int q = 0; q < randomNumbers.length; q++) {

            // 获取剩余的可能数字
            int remainingNumbers = range - q;

            // 从剩余数字中获取一个新的随机数
            int newRandSpot = (int) (Math.random() * remainingNumbers);

            newRandSpot++;

            // 遍历所有可能的数字
            for (int t = 1; t < range + 1; t++) {

                // 检查该数字是否已被占用
                boolean taken = false;
                for (int number : randomNumbers) {
                    if (t == number) {
                        taken = true;
                        break;
                    }
                }

                // 如果未被占用，则从位置中减去一个
                if (!taken) {
                    newRandSpot--;
                }
            }

            // 如果我们已经遍历了所有位置，则设置该值
            if (newRandSpot == 0) {
                randomNumbers[q] = t;
            }
        }
    }
    return randomNumbers;
} else {
    // 无效，长度不能大于可能数字的范围
}
return null;
}
```

该方法通过循环一个大小为请求长度的数组，找到剩余的

A good method to get a Long that is different for every call is [System.currentTimeMillis\(\)](#):

```
Random random = new Random(System.currentTimeMillis());
ThreadLocalRandom.current().setSeed(System.currentTimeMillis());
```

Section 87.5: Select random numbers without duplicates

```
/*
 * returns a array of random numbers with no duplicates
 * @param range the range of possible numbers for ex. if 100 then it can be anywhere from 1-100
 * @param length the length of the array of random numbers
 * @return array of random numbers with no duplicates.
 */
public static int[] getRandomNumbersWithNoDuplicates(int range, int length){
    if (length < range) {
        // this is where all the random numbers
        int[] randomNumbers = new int[length];

        // loop through all the random numbers to set them
        for (int q = 0; q < randomNumbers.length; q++) {

            // get the remaining possible numbers
            int remainingNumbers = range - q;

            // get a new random number from the remainingNumbers
            int newRandSpot = (int) (Math.random() * remainingNumbers);

            newRandSpot++;

            // loop through all the possible numbers
            for (int t = 1; t < range + 1; t++) {

                // check to see if this number has already been taken
                boolean taken = false;
                for (int number : randomNumbers) {
                    if (t == number) {
                        taken = true;
                        break;
                    }
                }

                // if it hasn't been taken then remove one from the spots
                if (!taken) {
                    newRandSpot--;
                }
            }

            // if we have gone though all the spots then set the value
            if (newRandSpot == 0) {
                randomNumbers[q] = t;
            }
        }
    }
    return randomNumbers;
} else {
    // invalid can't have a length larger than the range of possible numbers
}
return null;
}
```

The method works by looping though an array that has the size of the requested length and finds the remaining

可能数字的长度。它设置这些可能数字中的一个随机数newRandSpot，并在剩余未被占用的数字中找到该数字。它通过循环范围并检查该数字是否已被占用来实现。

例如，如果范围是5，长度是3，且我们已经选择了数字2。那么剩余数字有4个，我们在1到4之间获取一个随机数，并循环范围(5)，跳过任何已使用的数字(2)。

现在假设下一个在1到4之间选择的数字是3。在第一次循环中我们得到1，1还没有被取走所以我们可以从3中减去1，变成2。现在在第二次循环中我们得到2，2已经被取走，所以我们什么都不做。我们按照这个模式继续，直到到达4，一旦我们减去1，它变成0，所以我们将新的randomNumber设置为4。

第87.6节：使用apache-common lang3生成随机数

我们可以使用org.apache.commons.lang3.RandomUtils通过一行代码生成随机数。

```
int x = RandomUtils.nextInt(1, 1000);
```

方法nextInt(int startInclusive, int endExclusive)接受一个范围。

除了int，我们还可以使用这个类生成随机的long、double、float和bytes。

RandomUtils类包含以下方法-

```
static byte[] nextBytes(int count) //创建一个随机字节数组。  
static double nextDouble() //返回一个介于0到Double.MAX_VALUE之间的随机double值static double  
nextDouble(double startInclusive, double endInclusive) //返回一个指定范围内的随机double值。  
  
static float nextFloat() // 返回一个介于 0 到 Float.MAX_VALUE 之间的随机浮点数static float n  
extFloat(float startInclusive, float endInclusive) // 返回一个介于指定范围内的随机浮点数。  
  
static int nextInt() // 返回一个介于 0 到 Integer.MAX_VALUE 之间的随机整数static int ne  
xtInt(int startInclusive, int endExclusive) // 返回一个介于指定范围内的随机整数。  
  
static long nextLong() // 返回一个介于 0 到 Long.MAX_VALUE 之间的随机长整数static lo  
ng nextLong(long startInclusive, long endExclusive) // 返回一个介于指定范围内的随机长整数。
```

length of possible numbers. It sets a random number of those possible numbers newRandSpot and finds that number within the non taken number left. It does this by looping through the range and checking to see if that number has already been taken.

For example if the range is 5 and the length is 3 and we have already chosen the number 2. Then we have 4 remaining numbers so we get a random number between 1 and 4 and we loop through the range(5) skipping over any numbers that we have already used(2).

Now let's say the next number chosen between 1 & 4 is 3. On the first loop we get 1 which has not yet been taken so we can remove 1 from 3 making it 2. Now on the second loop we get 2 which has been taken so we do nothing. We follow this pattern until we get to 4 where once we remove 1 it becomes 0 so we set the new randomNumber to 4.

Section 87.6: Generating Random number using apache-common lang3

We can use org.apache.commons.lang3.RandomUtils to generate random numbers using a single line.

```
int x = RandomUtils.nextInt(1, 1000);
```

The method nextInt(int startInclusive, int endExclusive) takes a range.

Apart from int, we can generate random long, double, float and bytes using this class.

RandomUtils class contains the following methods-

```
static byte[] nextBytes(int count) //Creates an array of random bytes.  
static double nextDouble() //Returns a random double within 0 - Double.MAX_VALUE  
static double nextDouble(double startInclusive, double endInclusive) //Returns a random double  
within the specified range.  
static float nextFloat() //Returns a random float within 0 - Float.MAX_VALUE  
static float nextFloat(float startInclusive, float endInclusive) //Returns a random float within  
the specified range.  
static int nextInt() //Returns a random int within 0 - Integer.MAX_VALUE  
static int nextInt(int startInclusive, int endExclusive) //Returns a random integer within the  
specified range.  
static long nextLong() //Returns a random long within 0 - Long.MAX_VALUE  
static long nextLong(long startInclusive, long endExclusive) //Returns a random long within the  
specified range.
```

第88章：单例模式

单例是指一个类只有一个实例。有关单例设计模式的更多信息，请参阅设计模式标签中的单例主题。

第88.1节：枚举单例

版本 ≥ Java SE 5

```
public enum Singleton {  
    INSTANCE;  
  
    public void execute (String arg) {  
        // 在此执行操作  
    }  
}
```

枚举具有私有构造函数，是最终类，并提供了适当的序列化机制。它们也非常简洁，并且以线程安全的方式延迟初始化。

JVM 保证枚举值不会被实例化多次，这为枚举单例模式提供了对反射攻击的强有力防护。

枚举模式无法防止的是其他开发者在源代码中实际添加更多元素。因此，如果你选择这种实现风格来实现单例，务必明确文档说明不应向这些枚举中添加新值。

这是实现单例模式的推荐方式，正如 Joshua Bloch 在《Effective Java》中所解释的那样。

第 88.2 节：不使用枚举的单例（饿汉式初始化）

```
public class Singleton {  
  
    private static final Singleton INSTANCE = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

可以说这个例子实际上是延迟初始化。Java 语言规范第 12.4.1 节指出：

类或接口类型 T 将在以下任一情况首次出现之前立即初始化：

- T 是一个类，并且创建了 T 的一个实例
- T 是一个类，并且调用了 T 声明的静态方法
- 给 T 声明的静态字段赋值
- 由 T 声明的静态字段被使用，且该字段不是常量变量
- T 是顶层类，并且在 T 中嵌套的 assert 语句被执行。

Chapter 88: Singletons

A singleton is a class that only ever has one single instance. For more information on the Singleton *design pattern*, please refer to the Singleton topic in the Design Patterns tag.

Section 88.1: Enum Singleton

Version ≥ Java SE 5

```
public enum Singleton {  
    INSTANCE;  
  
    public void execute (String arg) {  
        // Perform operation here  
    }  
}
```

Enums have private constructors, are final and provide proper serialization machinery. They are also very concise and lazily initialized in a thread safe manner.

The JVM provides a guarantee that enum values will not be instantiated more than once each, which gives the enum singleton pattern a very strong defense against reflection attacks.

What the enum pattern *doesn't* protect against is other developers physically adding more elements to the source code. Consequently, if you choose this implementation style for your singletons it is imperative that you very clearly document that no new values should be added to those enums.

This is the recommended way of implementing the singleton pattern, as [explained](#) by Joshua Bloch in Effective Java.

Section 88.2: Singleton without use of Enum (eager initialization)

```
public class Singleton {  
  
    private static final Singleton INSTANCE = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

It can be argued that this example is *effectively lazy* initialization. [Section 12.4.1 of the Java Language Specification](#) states:

A class or interface type T will be initialized immediately before the first occurrence of any one of the following:

- T 是一个类，并且创建了 T 的一个实例
- T 是一个类，并且一个静态方法声明由 T 定义并被调用
- 由 T 声明的静态字段被赋值
- 由 T 声明的静态字段被使用，且该字段不是常量变量
- T 是顶层类，并且在 T 中嵌套的 assert 语句被执行。

因此，只要类中没有其他静态字段或静态方法，Singleton实例将不会被初始化，直到第一次调用方法getInstance()。

第88.3节：使用持有者类的线程安全延迟初始化 | Bill Pugh单例实现

```
public class Singleton {  
    private static class InstanceHolder {  
        static final Singleton INSTANCE = new Singleton();  
    }  
  
    public static Singleton getInstance() {  
        return InstanceHolder.INSTANCE;  
    }  
  
    private Singleton() {}  
}
```

这会在第一次调用Singleton.getInstance()时初始化INSTANCE变量，利用语言对静态初始化的线程安全保证，无需额外同步。

该实现也被称为Bill Pugh单例模式。 [\[Wiki\]](#)

第88.4节：使用双重检查锁定的线程安全单例模式

这种单例模式是线程安全的，并且在单例实例创建后防止不必要的锁定。

版本 ≥ Java SE 5

```
public class MySingleton {  
  
    // 类的实例  
    private static volatile MySingleton instance = null;  
  
    // 私有构造函数  
    private MySingleton() {  
        // 构造对象的一些代码  
    }  
  
    public static MySingleton getInstance() {  
        MySingleton result = instance;  
  
        // 如果实例已经存在，则不需要锁定  
        if(result == null) {  
            // 单例实例不存在，锁定并再次检查  
            synchronized(MySingleton.class) {  
                结果 = 实例;  
                if(result == null) {  
                    instance = result = new MySingleton();  
                }  
            }  
        }  
        return result;  
    }  
}
```

必须强调的是——在 Java SE 5 之前的版本，上述实现是不正确的，应该避免使用。

Therefore, as long as there are no other static fields or static methods in the class, the Singleton instance will not be initialized until the method getInstance() is invoked the first time.

Section 88.3: Thread-safe lazy initialization using holder class | Bill Pugh Singleton implementation

```
public class Singleton {  
    private static class InstanceHolder {  
        static final Singleton INSTANCE = new Singleton();  
    }  
  
    public static Singleton getInstance() {  
        return InstanceHolder.INSTANCE;  
    }  
  
    private Singleton() {}  
}
```

This initializes the INSTANCE variable on the first call to Singleton.getInstance(), taking advantage of the language's thread safety guarantees for static initialization without requiring additional synchronization.

This implementation is also known as Bill Pugh singleton pattern. [\[Wiki\]](#)

Section 88.4: Thread safe Singleton with double checked locking

This type of Singleton is thread safe, and prevents unnecessary locking after the Singleton instance has been created.

Version ≥ Java SE 5

```
public class MySingleton {  
  
    // instance of class  
    private static volatile MySingleton instance = null;  
  
    // Private constructor  
    private MySingleton() {  
        // Some code for constructing object  
    }  
  
    public static MySingleton getInstance() {  
        MySingleton result = instance;  
  
        // If the instance already exists, no locking is necessary  
        if(result == null) {  
            // The singleton instance doesn't exist, lock and check again  
            synchronized(MySingleton.class) {  
                result = instance;  
                if(result == null) {  
                    instance = result = new MySingleton();  
                }  
            }  
        }  
        return result;  
    }  
}
```

It must be emphasized -- in versions prior to Java SE 5, the implementation above is [incorrect](#) and should be

在 Java 5 之前的版本中，无法正确实现双重检查锁定。

第 88.5 节：扩展单例（单例继承）

在此示例中，基类Singleton提供了getMessage()方法，返回"Hello world!"消息。

它的子类UppercaseSingleton和LowercaseSingleton重写了 getMessage() 方法，以提供消息的适当表示形式。

```
//是的，我们需要使用反射来实现这一点。
import java.lang.reflect.*;

/*
表示单例实例可能类别的枚举。
如果未知，我们将使用基类 - Singleton。
*/
enum SingletonKind {
    UNKNOWN,
    LOWERCASE,
    UPPERCASE
}

//基类
class Singleton{

    /*
扩展类必须是私有内部类，以防止以不受控的方式扩展它们。
*/
private class UppercaseSingleton extends Singleton {

    private UppercaseSingleton(){
        super();
    }

    @Override
    public String getMessage() {
        return super.getMessage().toUpperCase();
    }
}

//另一个扩展类。
private class LowercaseSingleton extends Singleton
{
    private LowercaseSingleton(){
        super();
    }

    @Override
    public String getMessage() {
        return super.getMessage().toLowerCase();
    }
}

//应用单例模式
private static SingletonKind kind = SingletonKind.UNKNOWN;

private static Singleton instance;

/*
通过在调用 getInstance() 方法之前使用此方法，您可以有效地更改

```

avoided. It is not possible to implement double-checked locking correctly in Java prior to Java 5.

Section 88.5: Extending singleton (singleton inheritance)

In this example, base class Singleton provides getMessage() method that returns "Hello world!" message.

Its subclasses UppercaseSingleton and LowercaseSingleton override getMessage() method to provide appropriate representation of the message.

```
//Yeah, we'll need reflection to pull this off.
import java.lang.reflect.*;

/*
Enumeration that represents possible classes of singleton instance.
If unknown, we'll go with base class - Singleton.
*/
enum SingletonKind {
    UNKNOWN,
    LOWERCASE,
    UPPERCASE
}

//Base class
class Singleton{

    /*
Extended classes has to be private inner classes, to prevent extending them in
uncontrolled manner.
*/
private class UppercaseSingleton extends Singleton {

    private UppercaseSingleton(){
        super();
    }

    @Override
    public String getMessage() {
        return super.getMessage().toUpperCase();
    }
}

//Another extended class.
private class LowercaseSingleton extends Singleton
{
    private LowercaseSingleton(){
        super();
    }

    @Override
    public String getMessage() {
        return super.getMessage().toLowerCase();
    }
}

//Applying Singleton pattern
private static SingletonKind kind = SingletonKind.UNKNOWN;

private static Singleton instance;

/*
By using this method prior to getInstance() method, you effectively change the

```

要创建的单例实例的类型。

```
/*
public static void setKind(SingletonKind kind) {
    Singleton.kind = kind;
}

/*
如果需要, getInstance() 会根据 singletonKind 字段的值创建相应类的实例。
```

```
/*
public static Singleton getInstance()
    throws NoSuchMethodException,
        IllegalAccessException,
        InvocationTargetException,
        InstantiationException {
```

```
    if(instance==null){
        synchronized (Singleton.class){
            if(instance==null){
                Singleton singleton = new Singleton();
                switch (kind){
                    case UNKNOWN:

instance = singleton;
                    break;
```

case LOWERCASE:

/*

我不能直接使用简单的

```
    instance = new LowercaseSingleton();
```

因为 Java 编译器不允许我在静态上下文中使用
内部类的构造函数,
所以我改用反射 API。

为了能够通过反射 API 访问内部类,
我必须先创建外部类的实例。

因此, 在此实现中, 单例类不能是抽象类。

*/

//获取内部类的构造函数。

```
Constructor<LowercaseSingleton> lcConstructor =
LowercaseSingleton.class.getDeclaredConstructor(Singleton.class);

//构造函数是私有的, 所以我必须将其设置为可访问。
lcConstructor.setAccessible(true);

//使用构造函数创建实例。
instance = lcConstructor.newInstance(singleton);

break;
```

case UPPERCASE:

//这里也是一样, 只是类型不同

```
Constructor<UppercaseSingleton> ucConstructor =
UppercaseSingleton.class.getDeclaredConstructor(Singleton.class);
```

type of singleton instance to be created.

```
/*
public static void setKind(SingletonKind kind) {
    Singleton.kind = kind;
}
```

```
/*
If needed, getInstance() creates instance appropriate class, based on value of
singletonKind field.
```

```
/*
public static Singleton getInstance()
    throws NoSuchMethodException,
        IllegalAccessException,
        InvocationTargetException,
        InstantiationException {
```

```
    if(instance==null){
        synchronized (Singleton.class){
            if(instance==null){
                Singleton singleton = new Singleton();
                switch (kind){
                    case UNKNOWN:
```

```
                instance = singleton;
                break;
```

case LOWERCASE:

```
/*  
I can't use simple
```

```
    instance = new LowercaseSingleton();
```

because java compiler won't allow me to use
constructor of inner class in static context,
so I use reflection API instead.

To be able to access inner class by reflection API,
I have to create instance of outer class first.
Therefore, in this implementation, Singleton cannot be
abstract class.

*/

//Get the constructor of inner class.

```
Constructor<LowercaseSingleton> lcConstructor =
LowercaseSingleton.class.getDeclaredConstructor(Singleton.class);

//The constructor is private, so I have to make it accessible.
lcConstructor.setAccessible(true);

// Use the constructor to create instance.
instance = lcConstructor.newInstance(singleton);

break;
```

case UPPERCASE:

//Same goes here, just with different type
Constructor<UppercaseSingleton> ucConstructor =

```
UppercaseSingleton.class.getDeclaredConstructor(Singleton.class);
```

```

ucConstructor.setAccessible(true);
    instance = ucConstructor.newInstance(singleton);
}
}
}
return instance;
}

//单例状态，供子类使用
protected String message;

//私有构造函数，防止外部实例化。
private Singleton()
{
message = "Hello world!";

//单例的API。实现可以被子类覆盖。
public String getMessage() {
    return message;
}

//只是一个小测试程序
public class ExtendingSingletonExample {

public static void main(String args[]){
    //只需取消注释以下任意一行即可更改单例类

    //Singleton.setKind(SingletonKind.UPPERCASE);
    //Singleton.setKind(SingletonKind.LOWERCASE);

Singleton singleton = null;
try {
singleton = Singleton.getInstance();
} catch (NoSuchMethodException e) {
e.printStackTrace();
} catch (IllegalAccessException e) {
e.printStackTrace();
} catch (InvocationTargetException e) {
e.printStackTrace();
} catch (InstantiationException e) {
e.printStackTrace();
}
System.out.println(singleton.getMessage());
}
}

```

```

ucConstructor.setAccessible(true);
instance = ucConstructor.newInstance(singleton);
}
}
}
return instance;
}

//Singleton's state that is to be used by subclasses
protected String message;

//Private constructor prevents external instantiation.
private Singleton()
{
message = "Hello world!";

//Singleton's API. Implementation can be overwritten by subclasses.
public String getMessage() {
    return message;
}

//Just a small test program
public class ExtendingSingletonExample {

public static void main(String args[]){
    //just uncomment one of following lines to change singleton class

    //Singleton.setKind(SingletonKind.UPPERCASE);
    //Singleton.setKind(SingletonKind.LOWERCASE);

Singleton singleton = null;
try {
singleton = Singleton.getInstance();
} catch (NoSuchMethodException e) {
e.printStackTrace();
} catch (IllegalAccessException e) {
e.printStackTrace();
} catch (InvocationTargetException e) {
e.printStackTrace();
} catch (InstantiationException e) {
e.printStackTrace();
}
System.out.println(singleton.getMessage());
}
}

```

第89章：自动装箱

自动装箱是Java编译器在基本类型与其对应的对象包装类之间进行的自动转换。例如，将int转换为Integer，double转换为Double.....如果转换方向相反，则称为拆箱。通常，这用于只能存放对象的集合中，在将基本类型放入集合之前需要进行装箱。

第89.1节：交替使用int和Integer

当你在实用类中使用泛型类型时，你可能会发现数字类型在指定为对象类型时并不太有用，因为它们与其基本类型对应物不相等。

```
List<Integer> ints = new ArrayList<Integer>();  
版本 ≥ Java SE 7  
List<Integer> ints = new ArrayList<>();
```

幸运的是，求值为int的表达式可以在需要Integer的地方使用。

```
for (int i = 0; i < 10; i++)  
    ints.add(i);
```

语句ints.add(i);等价于：

```
ints.add(Integer.valueOf(i));
```

并且保留了来自Integer#valueOf的属性，例如当数值在JVM的数字缓存范围内时，具有相同的Integer对象缓存。

这同样适用于：

- `byte` 和 `Byte`
- `short` 和 `Short`
- `float` 和 `Float`
- `double` 和 `Double`
- `long` 和 `Long`
- `char` 和 `Character`
- `boolean` 和 `Boolean`

然而，在模糊的情况下必须小心。考虑以下代码：

```
List<Integer> ints = new ArrayList<Integer>();  
ints.add(1);  
ints.add(2);  
ints.add(3);  
ints.remove(1); // ints 现在是 [1, 3]
```

java.util.List接口同时包含remove(int index) (List接口方法) 和remove(Object o) (继承自java.util.Collection的方法)。在这种情况下不会发生装箱，调用的是remove(int index)方法。

另一个由自动装箱导致的奇怪Java代码行为的例子，涉及值在-128到范围内的Integer
127:

Chapter 89: Autoboxing

[Autoboxing](#) is the automatic conversion that Java compiler makes between primitive types and their corresponding object wrapper classes. Example, converting int -> Integer, double -> Double... If the conversion goes the other way, this is called unboxing. Typically, this is used in Collections that cannot hold other than Objects, where boxing primitive types is needed before setting them in the collection.

Section 89.1: Using int and Integer interchangeably

As you use generic types with utility classes, you may often find that number types aren't very helpful when specified as the object types, as they aren't equal to their primitive counterparts.

```
List<Integer> ints = new ArrayList<Integer>();  
Version ≥ Java SE 7  
List<Integer> ints = new ArrayList<>();
```

Fortunately, expressions that evaluate to `int` can be used in place of an `Integer` when it is needed.

```
for (int i = 0; i < 10; i++)  
    ints.add(i);
```

The `ints.add(i);` statement is equivalent to:

```
ints.add(Integer.valueOf(i));
```

And retains properties from `Integer#valueOf` such as having the same `Integer` objects cached by the JVM when it is within the number caching range.

This also applies to:

- `byte` and `Byte`
- `short` and `Short`
- `float` and `Float`
- `double` and `Double`
- `long` and `Long`
- `char` and `Character`
- `boolean` and `Boolean`

Care must be taken, however, in ambiguous situations. Consider the following code:

```
List<Integer> ints = new ArrayList<Integer>();  
ints.add(1);  
ints.add(2);  
ints.add(3);  
ints.remove(1); // ints is now [1, 3]
```

The `java.util.List` interface contains both a `remove(int index)` (`List` interface method) and a `remove(Object o)` (method inherited from `java.util.Collection`). In this case no boxing takes place and `remove(int index)` is called.

One more example of strange Java code behavior caused by autoboxing Integers with values in range from -128 to 127:

```
整数 a = 127;
整数 b = 127;
整数 c = 128;
整数 d = 128;
System.out.println(a == b); // true
System.out.println(c <= d); // true
System.out.println(c >= d); // true
System.out.println(c == d); // false
```

这是因为`>=`运算符隐式调用了`intValue()`，该方法返回`int`类型，而`==`比较的是引用，而不是`int`值。

默认情况下，Java 缓存范围为`[-128, 127]`内的值，因此`==`运算符有效，因为该范围内的`Integer`对象如果值相同，则引用相同的对象。缓存范围的最大值可以通过`-XX:AutoBoxCacheMax` JVM 选项定义。因此，如果你使用`-XX:AutoBoxCacheMax=1000`运行程序，以下代码将打印`true`：

```
整数 a = 1000;
Integer b = 1000;
System.out.println(a == b); // true
```

第89.2节：自动拆箱可能导致空指针异常

这段代码可以编译：

```
Integer arg = null;
int x = arg;
```

但在运行时第二行会抛出`java.lang.NullPointerException`异常。

问题在于基本类型`int`不能有`null`值。

这是一个极简示例，但在实际中通常以更复杂的形式出现。

`NullPointerException`并不直观，且通常对定位此类错误帮助不大。

使用自动装箱和自动拆箱时要小心，确保拆箱的值在运行时不会为`null`。

第89.3节：在if语句中使用Boolean

由于自动拆箱，可以在if语句中使用`Boolean`：

```
Boolean a = Boolean.TRUE;
if (a) { // a被转换为boolean类型
    System.out.println("它有效！");
}
```

这同样适用于`while`、`do while`以及`for`语句中的条件部分。

注意，如果`Boolean`为`null`，则在转换时会抛出`NullPointerException`异常。

第89.4节：Integer和int可以互换使用的不同情况

情况1：作为方法参数使用时。

```
Integer a = 127;
Integer b = 127;
Integer c = 128;
Integer d = 128;
System.out.println(a == b); // true
System.out.println(c <= d); // true
System.out.println(c >= d); // true
System.out.println(c == d); // false
```

This happens because `>=` operator implicitly calls `intValue()` which returns `int` while `==` compares `references`, not the `int` values.

By default, Java caches values in range `[-128, 127]`, so the operator `==` works because the `Integers` in this range reference to the same objects if their values are same. Maximal value of the cacheable range can be defined with `-XX:AutoBoxCacheMax` JVM option. So, if you run the program with `-XX:AutoBoxCacheMax=1000`, the following code will print `true`:

```
Integer a = 1000;
Integer b = 1000;
System.out.println(a == b); // true
```

Section 89.2: Auto-unboxing may lead to NullPointerException

This code compiles:

```
Integer arg = null;
int x = arg;
```

But it will crash at runtime with a `java.lang.NullPointerException` on the second line.

The problem is that a primitive `int` cannot have a `null` value.

This is a minimalistic example, but in practice it often manifests in more sophisticated forms. The `NullPointerException` is not very intuitive and is often little help in locating such bugs.

Rely on autoboxing and auto-unboxing with care, make sure that unboxed values will not have `null` values at runtime.

Section 89.3: Using Boolean in if statement

Due to auto unboxing, one can use a `Boolean` in an if statement:

```
Boolean a = Boolean.TRUE;
if (a) { // a gets converted to boolean
    System.out.println("It works!");
}
```

That works for `while`, `do while` and the condition in the `for` statements as well.

Note that, if the `Boolean` is `null`, a `NullPointerException` will be thrown in the conversion.

Section 89.4: Different Cases When Integer and int can be used interchangeably

Case 1: While using in the place of method arguments.

如果方法需要一个包装类对象作为参数，那么可以互换地传递相应的基本类型变量，反之亦然。

示例：

```
int i;  
整数 j;  
void ex_method(整数 i)// 是一个有效的语句  
void ex_method1(int j)// 是一个有效的语句
```

情况 2： 传递返回值时：

当一个方法返回基本类型变量时，可以互换地传递对应包装类的对象作为返回值，反之亦然。

示例：

```
int i;  
整数 j;  
int ex_method()  
{...  
return j;}// 是一个有效的语句  
整数 ex_method1()  
{...  
return i;}// 是一个有效的语句  
}
```

情况 3： 执行操作时。

在对数字执行操作时，基本类型变量和相应包装类的对象可以互换使用。

```
int i=5;  
Integer j=new Integer(7);  
int k=i+j;// 是一个有效的语句  
Integer m=i+j;// 这也是一个有效的语句
```

陷阱：记得初始化或赋值给包装类对象。

在使用包装类对象和基本变量交替时，切勿忘记或遗漏初始化或赋值给包装类对象，否则可能导致运行时出现空指针异常。

示例：

```
public class Test{  
    Integer i;  
    int j;  
    public void met()  
    {j=i;// 空指针异常  
    SOP(j);  
    SOP(i);}  
    public static void main(String[] args)  
    {Test t=new Test();  
    t.go();//空指针异常  
    }
```

在上述示例中，对象的值未被赋值且未初始化，因此在运行时程序将

If a method requires an object of wrapper class as argument. Then interchangeably the argument can be passed as a variable of the respective primitive type and vice versa.

Example:

```
int i;  
Integer j;  
void ex_method(Integer i)//Is a valid statement  
void ex_method1(int j)//Is a valid statement
```

Case 2: While passing return values:

When a method returns a primitive type variable then an object of corresponding wrapper class can be passed as the return value interchangeably and vice versa.

Example:

```
int i;  
Integer j;  
int ex_method()  
{...  
return j;}//Is a valid statement  
Integer ex_method1()  
{...  
return i;}//Is a valid statement  
}
```

Case 3: While performing operations.

Whenever performing operations on numbers the primitive type variable and object of respective wrapper class can be used interchangeably.

```
int i=5;  
Integer j=new Integer(7);  
int k=i+j;//Is a valid statement  
Integer m=i+j;//Is also a valid statement
```

Pitfall: Remember to initialize or assign a value to an object of the wrapper class.

While using wrapper class object and primitive variable interchangeably never forget or miss to initialize or assign a value to the wrapper class object else it may lead to null pointer exception at runtime.

Example:

```
public class Test{  
    Integer i;  
    int j;  
    public void met()  
    {j=i;//Null pointer exception  
    SOP(j);  
    SOP(i);}  
    public static void main(String[] args)  
    {Test t=new Test();  
    t.go();//Null pointer exception  
    }
```

In the above example, the value of the object is unassigned and uninitialized and thus at runtime the program will

遇到空指针异常。因此，从上述示例可以清楚看出，对象的值绝不应被未初始化和未赋值。

第89.5节：自动装箱的内存和计算开销

自动装箱可能带来大量的内存开销。例如：

```
Map<Integer, Integer> square = new HashMap<Integer, Integer>();
for(int i = 256; i < 1024; i++) {
    square.put(i, i * i); // 大整数的自动装箱
}
```

通常会消耗大量内存（大约6千条实际数据占用60KB）。

此外，装箱的整数通常需要额外的内存往返访问，因此使CPU缓存效率降低。在上述示例中，访问的内存分布在五个可能完全不同内存区域的位置：1. `HashMap` 对象，2. 映射的 `Entry[]` 表对象，3. `Entry` 对象，4.

条目的key对象（对原始键进行装箱），5. 条目的value对象（对原始值进行装箱）。

```
类 Example {
    int primitive; // 直接存储在类 `Example` 中
    Integer boxed; // 引用另一个内存位置
}
```

读取boxed需要两次内存访问，访问primitive只需一次。

从这个映射中获取数据时，看似无害的代码

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(i);
}
```

等同于：

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(Integer.valueOf(i)).intValue();
}
```

通常，上述代码会导致每次

`Map#get(Integer)`操作时都创建和垃圾回收一个`Integer`对象。（详见下方注释。）

为了减少这种开销，几个库提供了针对原始类型的优化集合，这些集合不需要装箱。除了避免装箱开销之外，这些集合每个条目所需的内存大约减少4倍。虽然Java Hotspot可能通过在栈上而非堆上操作对象来优化自动装箱，但无法优化内存开销及由此产生的内存间接访问。

Java 8流还针对原始数据类型优化了接口，例如`IntStream`，不需要装箱。

注意：典型的Java运行时维护一个简单的`Integer`及其他原始包装对象缓存，该缓存被`valueOf`工厂方法和自动装箱使用。对于`Integer`，该缓存的默认范围是-128到+127。一些JVM提供了用于更改缓存大小/范围的JVM命令行选项。

run into null pointer exception. So as clear from the above example the value of object should never be left uninitialized and unassigned.

Section 89.5: Memory and Computational Overhead of Autoboxing

Autoboxing can come at a substantial memory overhead. For example:

```
Map<Integer, Integer> square = new HashMap<Integer, Integer>();
for(int i = 256; i < 1024; i++) {
    square.put(i, i * i); // Autoboxing of large integers
}
```

will typically consume substantial amount of memory (about 60kb for 6k of actual data).

Furthermore, boxed integers usually require additional round-trips in the memory, and thus make CPU caches less effective. In above example, the memory accessed is spread out to five different locations that may be in entirely different regions of the memory: 1. the `HashMap` object, 2. the map's `Entry[]` table object, 3. the `Entry` object, 4. the entry's key object (boxing the primitive key), 5. the entry's value object (boxing the primitive value).

```
class Example {
    int primitive; // Stored directly in the class `Example`
    Integer boxed; // Reference to another memory location
}
```

Reading boxed requires two memory accesses, accessing primitive only one.

When getting data from this map, the seemingly innocent code

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(i);
}
```

is equivalent to:

```
int sumOfSquares = 0;
for(int i = 256; i < 1024; i++) {
    sumOfSquares += square.get(Integer.valueOf(i)).intValue();
}
```

Typically, the above code causes the creation and garbage collection of an `Integer` object for every `Map#get(Integer)` operation. (See Note below for more details.)

To reduce this overhead, several libraries offer optimized collections for primitive types that do not require boxing. In addition to avoiding the boxing overhead, these collection will require about 4x less memory per entry. While Java Hotspot may be able to optimize the autoboxing by working with objects on the stack instead of the heap, it is not possible to optimize the memory overhead and resulting memory indirection.

Java 8 streams also have optimized interfaces for primitive data types, such as `IntStream` that do not require boxing.

Note: a typical Java runtime maintains a simple cache of `Integer` and other primitive wrapper object that is used by the `valueOf` factory methods, and by autoboxing. For `Integer`, the default range of this cache is -128 to +127. Some JVMs provide a JVM command-line option for changing the cache size / range.

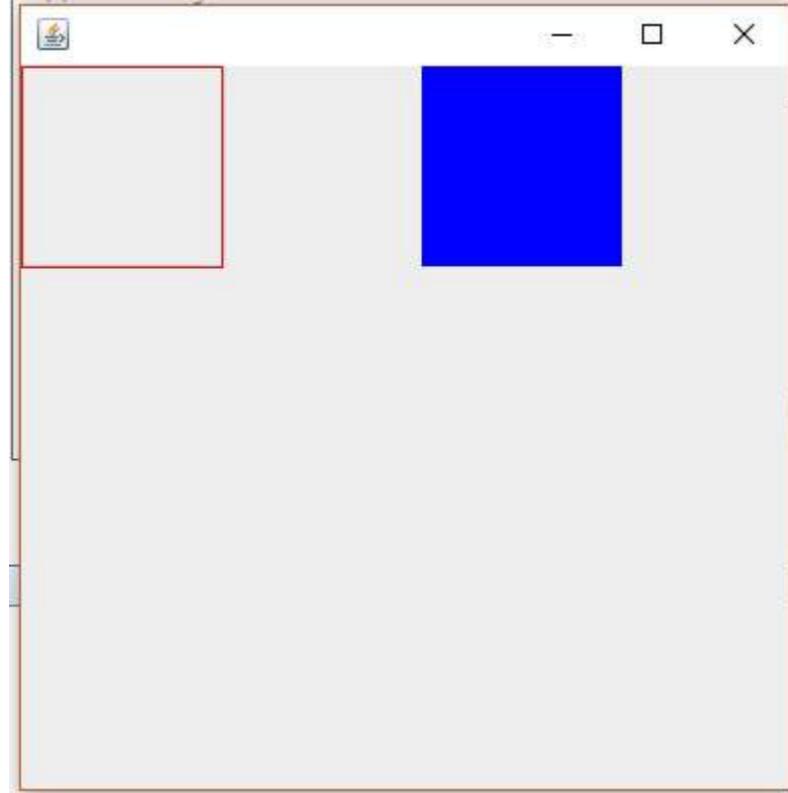
第90章：Java中的二维图形

图形是某些表面上的视觉图像或设计，如墙壁、画布、屏幕、纸张或石头，用于传达信息、说明或娱乐。它包括：数据的图示表示，如计算机辅助设计与制造、排版和平面艺术，以及教育和娱乐软件。由计算机生成的图像称为计算机图形。

Java 2D API功能强大且复杂。Java中有多种方式实现二维图形。

第90.1节：示例1：使用Java绘制和填充矩形

这是一个打印矩形并填充矩形颜色的示例。<https://i.stack.imgur.com/dlC5v.jpg>



Graphics类的大多数方法可以分为两大类：

1. 绘制和填充方法，使您能够渲染基本形状、文本和图像
2. 属性设置方法，影响绘制和填充的显示效果

代码示例：让我们从一个绘制矩形并填充颜色的小例子开始。这里我们声明了两个类，一个是 MyPanel 类，另一个是 Test 类。在 MyPanel 类中，我们使用 drawRect() 和 fillRect() 方法来绘制矩形并填充颜色。我们通过 setColor(Color.blue) 方法设置颜色。在第二个类 Test 中，我们测试图形，创建一个 Frame 并将 MyPanel 对象 p=new MyPanel() 放入其中。运行 Test 类时，我们会看到一个矩形和一个填充了蓝色的矩形。

第一个类：MyPanel

```
import javax.swing.*;
import java.awt.*;
// MyPanel 继承自 JPanel, 最终将被放置在 JFrame 中
public class MyPanel extends JPanel {
    // 自定义绘制由 paintComponent 方法执行
    @Override
    public void paintComponent(Graphics g){
```

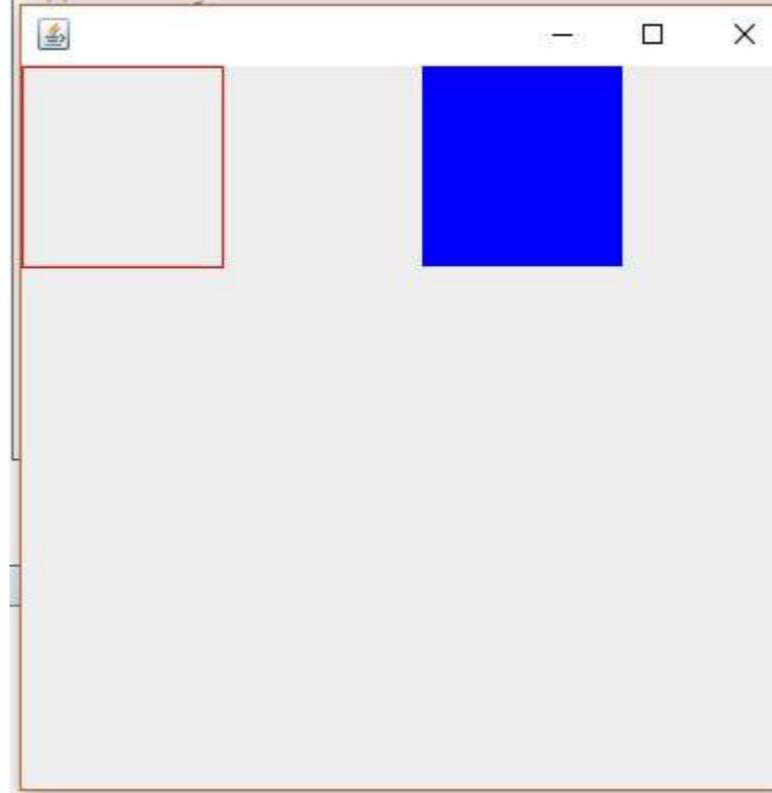
Chapter 90: 2D Graphics in Java

Graphics are visual images or designs on some surface, such as a wall, canvas, screen, paper, or stone to inform, illustrate, or entertain. It includes: pictorial representation of data, as in computer-aided design and manufacture, in typesetting and the graphic arts, and in educational and recreational software. Images that are generated by a computer are called computer graphics.

The Java 2D API is powerful and complex. There are multiple ways to do 2D graphics in Java.

Section 90.1: Example 1: Draw and Fill a Rectangle Using Java

This is an Example which print rectangle and fill color in the rectangle. <https://i.stack.imgur.com/dlC5v.jpg>



Most methods of the Graphics class can be divided into two basic groups:

1. Draw and fill methods, enabling you to render basic shapes, text, and images
2. Attributes setting methods, which affect how that drawing and filling appears

Code Example: Let us start this with a little example of drawing a rectangle and filling color in it. There we declare two classes, one class is MyPanel and other Class is Test. In class MyPanel we use drawRect() & fillRect() methods to draw rectangle and fill Color in it. We set the color by setColor(Color.blue) method. In Second Class we Test our graphic which is Test Class we make a Frame and put MyPanel with p=new MyPanel() object in it. By running Test Class we see a Rectangle and a Blue Color Filled Rectangle.

First Class: MyPanel

```
import javax.swing.*;
import java.awt.*;
// MyPanel extends JPanel, which will eventually be placed in a JFrame
public class MyPanel extends JPanel {
    // custom painting is performed by the paintComponent method
    @Override
    public void paintComponent(Graphics g){
```

```

// 清除之前的绘制内容
super.paintComponent(g);
// 将 Graphics 强制转换为 Graphics2D
Graphics2D g2 = (Graphics2D) g;
g2.setColor(Color.red); // 设置 Graphics2D 颜色
// 绘制矩形
g2.drawRect(0,0,100,100); // drawRect(x 位置, y 位置, 宽度, 高度)
g2.setColor(Color.blue);
g2.fillRect(200,0,100,100); // 用蓝色填充新矩形
}
}

```

第二类：测试

```

import javax.swing.*;
import java.awt.*;
public class Test { // 用于显示矩形的类
    JFrame f;
    MyPanel p;
    public Test(){
        f = new JFrame();
        // 获取面板的内容区域。
        Container c = f.getContentPane();
        // 设置布局管理器
        c.setLayout(new BorderLayout());
        p = new MyPanel();
        // 将 MyPanel 对象添加到容器中
        c.add(p);
        // 设置 JFrame 的大小
        f.setSize(400,400);
        // 使 JFrame 可见
        f.setVisible(true);
        // 设置关闭行为；EXIT_ON_CLOSE 在关闭 JFrame 时调用 System.exit(0)
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String args[ ]){
        Test t = new Test();
    }
}

```

关于边界布局的更多说明：<https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html>

paintComponent()

- 这是一个用于绘制的主方法
- 默认情况下，它首先绘制背景
- 之后执行自定义绘制（绘制圆形、矩形等）

Graphic2D 指的是 *Graphic2D* 类

注意：Java 2D API 使您能够轻松执行以下任务：

- 绘制线条、矩形及其他任何几何形状。
- 用纯色、渐变色和纹理填充这些形状。
- 绘制文本，并可对字体和渲染过程进行精细控制。
- 绘制图像，可选择性地应用滤镜操作。
- 在上述任何渲染操作中应用合成和变换等操作。

```

// clear the previous painting
super.paintComponent(g);
// cast Graphics to Graphics2D
Graphics2D g2 = (Graphics2D) g;
g2.setColor(Color.red); // sets Graphics2D color
// draw the rectangle
g2.drawRect(0,0,100,100); // drawRect(x-position, y-position, width, height)
g2.setColor(Color.blue);
g2.fillRect(200,0,100,100); // fill new rectangle with color blue
}
}

```

Second Class: Test

```

import javax.swing.*;
import java.awt.*;
public class Test { //the Class by which we display our rectangle
    JFrame f;
    MyPanel p;
    public Test(){
        f = new JFrame();
        // get the content area of Panel.
        Container c = f.getContentPane();
        // set the LayoutManager
        c.setLayout(new BorderLayout());
        p = new MyPanel();
        // add MyPanel object into container
        c.add(p);
        // set the size of the JFrame
        f.setSize(400,400);
        // make the JFrame visible
        f.setVisible(true);
        // sets close behavior; EXIT_ON_CLOSE invokes System.exit(0) on closing the JFrame
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public static void main(String args[ ]){
        Test t = new Test();
    }
}

```

For More Explanation about Border Layout: <https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html>

paintComponent()

- It is a main method for painting
- By default, it first paints the background
- After that, it performs custom painting (drawing circle, rectangles etc.)

Graphic2D refers *Graphic2D Class*

Note: The Java 2D API enables you to easily perform the following tasks:

- Draw lines, rectangles and any other geometric shape.
- Fill those shapes with solid colors or gradients and textures.
- Draw text with options for fine control over the font and rendering process.
- Draw images, optionally applying filtering operations.
- Apply operations such as compositing and transforming during any of the above rendering operations.

第90.2节：示例2：绘制和填充椭圆

```
import javax.swing.*;
import java.awt.*;

public class MyPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g){
        // 清除之前的绘制内容
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setColor(Color.blue);
        g2.drawOval(0, 0, 20, 20);
        g2.fillOval(50, 50, 20, 20);
    }
}
```

g2.drawOval(int x,int y,int height, int width);

此方法将在指定的x和y位置绘制一个给定高度和宽度的椭圆。

g2.fillOval(int x,int y,int height, int width); 此方法将在指定的x和y位置填充一个给定高度和宽度的椭圆。

Section 90.2: Example 2: Drawing and Filling Oval

```
import javax.swing.*;
import java.awt.*;

public class MyPanel extends JPanel {
    @Override
    public void paintComponent(Graphics g){
        // clear the previous painting
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D)g;
        g2.setColor(Color.blue);
        g2.drawOval(0, 0, 20, 20);
        g2.fillOval(50, 50, 20, 20);
    }
}
```

g2.drawOval(int x,int y,int height, int width);

This method will draw an oval at specified x and y position with given height and width.

g2.fillOval(int x,int y,int height, int width); This method will fill an oval at specified x and y position with given height and width.

第91章：JAXB

参数	详情
fileObjOfXML	XML文件的文件对象
className	带有 .class 扩展名的类名

JAXB 或者说 Java XML 绑定架构 (JAXB) 是一个软件框架，允许 Java 开发者将 Java 类映射到 XML 表示形式。本文将通过详细示例介绍 JAXB 的功能，主要用于将 Java 对象编组 (marshaling) 和解组 (un-marshaling) 为 XML 格式，反之亦然。

第 91.1 节：读取 XML 文件 (解组)

要读取名为UserDetails.xml的 XML 文件，内容如下

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

我们需要一个名为User.java的 POJO 类，如下所示

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getter 和 setter 方法
}
```

这里我们根据 XML 节点创建了变量和类名。为了映射它们，我们在类上使用了注解XmlRootElement。

```
public class XMLReader {
    public static void main(String[] args) {
        try {
            User user = JAXB.unmarshal(new File("UserDetails.xml"), User.class);
            System.out.println(user.getName()); // 输出 Jon Skeet
            System.out.println(user.getUserID()); // 输出 8884321
        } catch (Exception e) {
            System.err.println("读取 XML 时发生异常！");
        }
    }
}
```

这里使用了 unmarshal() 方法来解析 XML 文件。它接受 XML 文件名和类类型作为两个参数。然后我们可以使用对象的 getter 方法来打印数据。

第 91.2 节：写入 XML 文件 (对象的编组)

```
import javax.xml.bind.annotation.XmlRootElement;
```

Chapter 91: JAXB

Parameter	Details
fileObjOfXML	File object of an XML file
className	Name of a class with .class extension

JAXB or [Java Architecture for XML Binding](#) (JAXB) is a software framework that allows Java developers to map Java classes to XML representations. This Page will introduce readers to JAXB using detailed examples about its functions provided mainly for marshaling and un-marshaling Java Objects into xml format and vice-versa.

Section 91.1: Reading an XML file (unmarshalling)

To read an XML file named UserDetails.xml with the below content

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>
```

We need a POJO class named User.java as below

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}
```

Here we have created the variables and class name according to the XML nodes. To map them, we use the annotation XmlRootElement on the class.

```
public class XMLReader {
    public static void main(String[] args) {
        try {
            User user = JAXB.unmarshal(new File("UserDetails.xml"), User.class);
            System.out.println(user.getName()); // prints Jon Skeet
            System.out.println(user.getUserID()); // prints 8884321
        } catch (Exception e) {
            System.err.println("Exception occurred while reading the XML!");
        }
    }
}
```

Here unmarshal() method is used to parse the XML file. It takes the XML file name and the class type as two arguments. Then we can use the getter methods of the object to print the data.

Section 91.2: Writing an XML file (marshalling an object)

```
import javax.xml.bind.annotation.XmlRootElement;
```

```

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getter 和 setter 方法
}

```

通过使用注解 `XmlRootElement`, 我们可以将一个类标记为 XML 文件的根元素。

```

import java.io.File;
import javax.xml.bind.JAXB;

public class XMLCreator {
    public static void main(String[] args) {
        User user = new User();
        user.setName("Jon Skeet");
        user.setUserID(8884321);

        try {
            JAXB.marshal(user, new File("UserDetails.xml"));
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        } finally {
            System.out.println("XML created");
        }
    }
}

```

`marshal()` 用于将对象的内容写入 XML 文件。这里将 `user` 对象和一个新的 `File` 对象作为参数传递给 `marshal()`。

执行成功后, 会在类路径下创建一个名为 `UserDetails.xml` 的 XML 文件, 内容如下。

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>

```

第 91.3 节：手动字段/属性 XML 映射配置

注解 `@XmlElement`、`@XmlAttribute` 或 `@XmlTransient` 以及包 `javax.xml.bind.annotation` 中的其他注解允许程序员指定哪些字段或属性以及如何序列化它们。

```

@XmlAccessorType(XmlAccessType.NONE) // 我们不希望自动进行字段/属性的编组
public class ManualXmlElementsExample {

    @XmlElement
    private String field="field value";

    @XmlAttribute
    private String attribute="attr value";

    @XmlAttribute(name="differentAttribute")
    private String oneAttribute="other attr value";
}

```

```

@XmlRootElement
public class User {

    private long userID;
    private String name;

    // getters and setters
}

```

By using the annotation `XmlRootElement`, we can mark a class as a root element of an XML file.

```

import java.io.File;
import javax.xml.bind.JAXB;

public class XMLCreator {
    public static void main(String[] args) {
        User user = new User();
        user.setName("Jon Skeet");
        user.setUserID(8884321);

        try {
            JAXB.marshal(user, new File("UserDetails.xml"));
        } catch (Exception e) {
            System.err.println("Exception occurred while writing in XML!");
        } finally {
            System.out.println("XML created");
        }
    }
}

```

`marshal()` is used to write the object's content into an XML file. Here `userobject` and a new `File` object are passed as arguments to the `marshal()`.

On successful execution, this creates an XML file named `UserDetails.xml` in the class-path with the below content.

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
    <name>Jon Skeet</name>
    <userID>8884321</userID>
</user>

```

Section 91.3: Manual field/property XML mapping configuration

Annotations `@XmlElement`, `@XmlAttribute` or `@XmlTransient` and other in package `javax.xml.bind.annotation` allow the programmer to specify which and how marked fields or properties should be serialized.

```

@XmlAccessorType(XmlAccessType.NONE) // we want no automatic field/property marshalling
public class ManualXmlElementsExample {

    @XmlElement
    private String field="field value";

    @XmlAttribute
    private String attribute="attr value";

    @XmlAttribute(name="differentAttribute")
    private String oneAttribute="other attr value";
}

```

```

@XmlElement(name="different name")
private String oneName="different name value";

@XmlTransient
private String transientField = "will not get serialized ever";

XmlElement
public String getModifiedTransientValue() {
    return transientField.replace(" ever", " ", unless in a getter");
}

public void setModifiedTransientValue(String val) {} // empty on purpose

public static void main(String[] args) {
    try {
        JAXB.marshal(new ManualXmlElementsExample(), System.out);
    } catch (Exception e) {
        System.err.println("Exception occurred while writing in XML!");
    }
}

```

第91.4节：将XML命名空间绑定到可序列化的Java类

这是一个绑定XML命名空间到可序列化Java类的package-info.java文件示例。该文件应放置在与应使用该命名空间序列化的Java类相同的包中。

```

/**
 * 包含可序列化类的包。
 */
@XmlSchema
(
xmlns =
{
@XmlNs(prefix = MySerializableClass.NAMESPACE_PREFIX, namespaceURI =
MySerializableClass.NAMESPACE)
},
namespace = MySerializableClass.NAMESPACE,
elementFormDefault = XmlNsForm.QUALIFIED
)
package com.test.jaxb;

import javax.xml.bind.annotation.XmlNs;
import javax.xml.bind.annotation.XmlNsForm;
import javax.xml.bind.annotation.XmlSchema;

```

第91.5节：使用XmlAdapter生成所需的XML格式

当所需的XML格式与Java对象模型不同时，可以使用XmlAdapter实现将模型对象转换为XML格式对象，反之亦然。此示例演示如何将字段的值放入具有字段名称的元素属性中。

```
public class XmlAdapterExample {
```

```

@XmlElement(name="different name")
private String oneName="different name value";

@XmlTransient
private String transientField = "will not get serialized ever";

XmlElement
public String getModifiedTransientValue() {
    return transientField.replace(" ever", " ", unless in a getter");
}

public void setModifiedTransientValue(String val) {} // empty on purpose

public static void main(String[] args) {
    try {
        JAXB.marshal(new ManualXmlElementsExample(), System.out);
    } catch (Exception e) {
        System.err.println("Exception occurred while writing in XML!");
    }
}

```

Section 91.4: Binding an XML namespace to a serializable Java class

This is an example of a package-info.java file that binds an XML namespace to a serializable Java class. This should be placed in the same package as the Java classes that should be serialized using the namespace.

```

/**
 * A package containing serializable classes.
 */
@XmlSchema
(
xmlns =
{
@XmlNs(prefix = MySerializableClass.NAMESPACE_PREFIX, namespaceURI =
MySerializableClass.NAMESPACE)
},
namespace = MySerializableClass.NAMESPACE,
elementFormDefault = XmlNsForm.QUALIFIED
)
package com.test.jaxb;

import javax.xml.bind.annotation.XmlNs;
import javax.xml.bind.annotation.XmlNsForm;
import javax.xml.bind.annotation.XmlSchema;

```

Section 91.5: Using XmlAdapter to generate desired xml format

When desired XML format differs from Java object model, an XmlAdapter implementation can be used to transform model object into xml-format object and vice versa. This example demonstrates how to put a field's value into an attribute of an element with field's name.

```
public class XmlAdapterExample {
```

```

@XmlAccessorType(XmlAccessType.FIELD)
public static class NodeValueElement {

    @XmlAttribute(name="attrValue")
    String value;

    public NodeValueElement() {
    }

    public NodeValueElement(String value) {
        super();
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}

public static class ValueAsAttrXmlAdapter extends XmlAdapter<NodeValueElement, String> {

    @Override
    public NodeValueElement marshal(String v) throws Exception {
        return new NodeValueElement(v);
    }

    @Override
    public String unmarshal(NodeValueElement v) throws Exception {
        if (v==null) return "";
        return v.getValue();
    }
}

@XmlRootElement(name="DataObject")
@XmlAccessorType(XmlAccessType.FIELD)
public static class DataObject {

    String elementWithValue;

    @XmlJavaTypeAdapter(value=ValueAsAttrXmlAdapter.class)
    String elementWithAttribute;
}

public static void main(String[] args) {
    DataObject data = new DataObject();
    data.elementWithValue="value1";
    data.elementWithAttribute ="value2";

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    JAXB.marshal(data, baos);

    String xmlString = new String(baos.toByteArray(), StandardCharsets.UTF_8);

    System.out.println(xmlString);
}
}

@XmlAccessorType(XmlAccessType.FIELD)
public static class NodeValueElement {

    @XmlAttribute(name="attrValue")
    String value;

    public NodeValueElement() {
    }

    public NodeValueElement(String value) {
        super();
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}

public static class ValueAsAttrXmlAdapter extends XmlAdapter<NodeValueElement, String> {

    @Override
    public NodeValueElement marshal(String v) throws Exception {
        return new NodeValueElement(v);
    }

    @Override
    public String unmarshal(NodeValueElement v) throws Exception {
        if (v==null) return "";
        return v.getValue();
    }
}

@XmlRootElement(name="DataObject")
@XmlAccessorType(XmlAccessType.FIELD)
public static class DataObject {

    String elementWithValue;

    @XmlJavaTypeAdapter(value=ValueAsAttrXmlAdapter.class)
    String elementWithAttribute;
}

public static void main(String[] args) {
    DataObject data = new DataObject();
    data.elementWithValue="value1";
    data.elementWithAttribute ="value2";

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    JAXB.marshal(data, baos);

    String xmlString = new String(baos.toByteArray(), StandardCharsets.UTF_8);

    System.out.println(xmlString);
}
}

```

第91.6节：使用XmlAdapter修剪字符串

```
package com.example.xml.adapters;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class StringTrimAdapter extends XmlAdapter<String, String> {
    @Override
    public String unmarshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }

    @Override
    public String marshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }
}
```

并且在 package-info.java 中添加以下声明。

```
@XmlJavaTypeAdapter(value = com.example.xml.adapters.StringTrimAdapter.class, type = String.class)
package com.example.xml.jaxb.bindings;// 你打算应用修剪过滤器的包

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
```

第 91.7 节：自动字段/属性 XML 映射配置 (@XmlAccessorType)

注解 `@XmlAccessorType` 决定字段/属性是否会自动序列化为 XML。注意，字段和方法上的注解 `@XmlElement`、`@XmlAttribute` 或 `@XmlTransient` 优先于默认设置。

```
public class XmlAccessTypeExample {

@XmlAccessorType(XmlAccessType.FIELD)
static class AccessorExampleField {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.NONE)
static class AccessorExampleNone {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}
```

Section 91.6: Using XmlAdapter to trim string

```
package com.example.xml.adapters;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class StringTrimAdapter extends XmlAdapter<String, String> {
    @Override
    public String unmarshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }

    @Override
    public String marshal(String v) throws Exception {
        if (v == null)
            return null;
        return v.trim();
    }
}
```

And in package-info.java add following declaration.

```
@XmlJavaTypeAdapter(value = com.example.xml.adapters.StringTrimAdapter.class, type = String.class)
package com.example.xml.jaxb.bindings;// Packge where you intend to apply trimming filter

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
```

Section 91.7: Automatic field/property XML mapping configuration (@XmlAccessorType)

Annotation `@XmlAccessorType` determines whether fields/properties will be automatically serialized to XML. Note, that field and method annotations `@XmlElement`, `@XmlAttribute` or `@XmlTransient` take precedence over the default settings.

```
public class XmlAccessTypeExample {

@XmlAccessorType(XmlAccessType.FIELD)
static class AccessorExampleField {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.NONE)
static class AccessorExampleNone {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}
```

```

@XmlAccessorType(XmlAccessType.PROPERTY)
static class AccessorExampleProperty {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)
static class AccessorExamplePublic {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

public static void main(String[] args) {
    try {
        System.out.println("Field:");
        JAXB.marshal(new AccessorExampleField(), System.out);System.out.println("None:");
        JAXB.marshal(new AccessorExampleNone(), System.out);System.out.println("Property:");
        JAXB.marshal(new AccessorExampleProperty(), System.out);System.out.println("Public:");
        JAXB.marshal(new AccessorExamplePublic(), System.out);
    } catch (Exception e) {
        System.err.println("Exception occurred while writing in XML!");
    }
}

} // 外部类结束

```

输出

字段：
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

value1

无：
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

属性：
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

getter

公共：
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

value1

```

@XmlAccessorType(XmlAccessType.PROPERTY)
static class AccessorExampleProperty {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)
static class AccessorExamplePublic {
    public String field="value1";

    public String getGetter() {
        return "getter";
    }

    public void setGetter(String value) {}
}

public static void main(String[] args) {
    try {
        System.out.println("\nField:");
        JAXB.marshal(new AccessorExampleField(), System.out);
        System.out.println("\nNone:");
        JAXB.marshal(new AccessorExampleNone(), System.out);
        System.out.println("\nProperty:");
        JAXB.marshal(new AccessorExampleProperty(), System.out);
        System.out.println("\nPublic:");
        JAXB.marshal(new AccessorExamplePublic(), System.out);
    } catch (Exception e) {
        System.err.println("Exception occurred while writing in XML!");
    }
}

} // outer class end

```

Output

Field：
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

value1

None：
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

Property：
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

getter

Public：
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

value1

第91.8节：指定XmlAdapter实例以（重新）使用现有数据

有时应使用特定的数据实例。不希望重新创建，并且引用静态数据会有代码异味。

可以指定Unmarshaller应使用的XmlAdapter实例，这允许用户使用无零参数构造函数的XmlAdapter和/或向适配器传递数据。

示例

用户类

以下类包含一个名称和用户的图像。

```
import java.awt.image.BufferedImage;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
public class User {

    private String name;
    private BufferedImage image;

    @XmlAttribute
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
    @XmlAttribute
    public BufferedImage getImage() {
        return image;
    }

    public void setImage(BufferedImage image) {
        this.image = image;
    }

    public User(String name, BufferedImage image) {
        this.name = name;
        this.image = image;
    }

    public User() {
        this("", null);
    }
}
```

适配器

Section 91.8: Specifying a XmlAdapter instance to (re)use existing data

Sometimes specific instances of data should be used. Recreation is not desired and referencing **static** data would have a code smell.

It is possible to specify a XmlAdapter instance the Unmarshaller should use, which allows the user to use XmlAdapters with no zero-arg constructor and/or pass data to the adapter.

Example

User class

The following class contains a name and a user's image.

```
import java.awt.image.BufferedImage;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
public class User {

    private String name;
    private BufferedImage image;

    @XmlAttribute
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
    @XmlAttribute
    public BufferedImage getImage() {
        return image;
    }

    public void setImage(BufferedImage image) {
        this.image = image;
    }

    public User(String name, BufferedImage image) {
        this.name = name;
        this.image = image;
    }

    public User() {
        this("", null);
    }
}
```

为了避免在内存中创建相同的图像两次（以及重复下载数据），适配器将图像存储在一个映射中。

版本 ≤ Java SE 7

对于有效的 Java 7 代码，将getImage方法替换为

```
public BufferedImage getImage(URL url) {
    BufferedImage image = imageCache.get(url);
    if (image == null) {
        try {
            image = ImageIO.read(url);
        } catch (IOException ex) {
            Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
        }
        return null;
    }
    imageCache.put(url, image);
    reverseIndex.put(image, url);
}
return image;

import java.awt.image.BufferedImage;
import java.io.IOException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
import javax.xml.bind.annotation.adapters.XmlAdapter;

public class ImageCacheAdapter extends XmlAdapter<String, BufferedImage> {

    private final Map<URL, BufferedImage> imageCache = new HashMap<>();
    private final Map<BufferedImage, URL> reverseIndex = new HashMap<>();

    public BufferedImage getImage(URL url) {
        // 使用 Java 8 方法进行单次查找
        return imageCache.computeIfAbsent(url, s -> {
            try {
                BufferedImage img = ImageIO.read(s);
                reverseIndex.put(img, s);
                return img;
            } catch (IOException ex) {
                Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
            }
            return null;
        });
    }

    @Override
    public BufferedImage unmarshal(String v) throws Exception {
        return getImage(new URL(v));
    }

    @Override
    public String marshal(BufferedImage v) throws Exception {
        return reverseIndex.get(v).toExternalForm();
    }
}
```

To avoid creating the same image in memory twice (as well as downloading the data again), the adapter stores the images in a map.

Version ≤ Java SE 7

For valid Java 7 code replace the getImage method with

```
public BufferedImage getImage(URL url) {
    BufferedImage image = imageCache.get(url);
    if (image == null) {
        try {
            image = ImageIO.read(url);
        } catch (IOException ex) {
            Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
        }
        return null;
    }
    imageCache.put(url, image);
    reverseIndex.put(image, url);
}
return image;

import java.awt.image.BufferedImage;
import java.io.IOException;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.imageio.ImageIO;
import javax.xml.bind.annotation.adapters.XmlAdapter;

public class ImageCacheAdapter extends XmlAdapter<String, BufferedImage> {

    private final Map<URL, BufferedImage> imageCache = new HashMap<>();
    private final Map<BufferedImage, URL> reverseIndex = new HashMap<>();

    public BufferedImage getImage(URL url) {
        // using a single lookup using Java 8 methods
        return imageCache.computeIfAbsent(url, s -> {
            try {
                BufferedImage img = ImageIO.read(s);
                reverseIndex.put(img, s);
                return img;
            } catch (IOException ex) {
                Logger.getLogger(ImageCacheAdapter.class.getName()).log(Level.SEVERE, null, ex);
            }
            return null;
        });
    }

    @Override
    public BufferedImage unmarshal(String v) throws Exception {
        return getImage(new URL(v));
    }

    @Override
    public String marshal(BufferedImage v) throws Exception {
        return reverseIndex.get(v).toExternalForm();
    }
}
```

```
}
```

示例 XML

以下两个 XML 是为乔恩·斯基特及其地球2对应者准备的，两者看起来完全相同，因此使用相同的头像。

```
<?xml version="1.0" encoding="UTF-8"?>

<user name="乔恩·斯基特"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=32&#038;d=identicon&#038;r=PG"/>

<?xml version="1.0" encoding="UTF-8"?>

<user name="乔恩·斯基特 (地球2)"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=32&#038;d=identicon&#038;r=PG"/>
```

使用适配器

```
ImageCacheAdapter adapter = new ImageCacheAdapter();

JAXBContext context = JAXBContext.newInstance(User.class);

Unmarshaller unmarshaller = context.createUnmarshaller();

// 指定每个
// @XmlJavaTypeAdapter(value=ImageCacheAdapter.class) 使用的适配器实例
unmarshaller.setAdapter(ImageCacheAdapter.class, adapter);

User result1 = (User) unmarshaller.unmarshal(Main.class.getResource("user.xml"));

// 使用相同的适配器实例反序列化第二个 XML
Unmarshaller unmarshaller2 = context.createUnmarshaller();
unmarshaller2.setAdapter(ImageCacheAdapter.class, adapter);
User result2 = (User) unmarshaller2.unmarshal(Main.class.getResource("user2.xml"));

System.out.println(result1.getName());
System.out.println(result2.getName());

// 返回true, 因为图像被重用
System.out.println(result1.getImage() == result2.getImage());
```

```
}
```

Example XMLs

The following 2 xmls are for Jon Skeet and his earth 2 counterpart, which both look exactly the same and therefore use the same avatar.

```
<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=32&#038;d=identicon&#038;r=PG"/>

<?xml version="1.0" encoding="UTF-8"?>

<user name="Jon Skeet (Earth 2)"
image="https://www.gravatar.com/avatar/6d8ebb117e8d83d74ea95fbdd0f87e13?s=32&#038;d=identicon&#038;r=PG"/>
```

Using the adapter

```
ImageCacheAdapter adapter = new ImageCacheAdapter();

JAXBContext context = JAXBContext.newInstance(User.class);

Unmarshaller unmarshaller = context.createUnmarshaller();

// specify the adapter instance to use for every
// @XmlJavaTypeAdapter(value=ImageCacheAdapter.class)
unmarshaller.setAdapter(ImageCacheAdapter.class, adapter);

User result1 = (User) unmarshaller.unmarshal(Main.class.getResource("user.xml"));

// unmarshal second xml using the same adapter instance
Unmarshaller unmarshaller2 = context.createUnmarshaller();
unmarshaller2.setAdapter(ImageCacheAdapter.class, adapter);
User result2 = (User) unmarshaller2.unmarshal(Main.class.getResource("user2.xml"));

System.out.println(result1.getName());
System.out.println(result2.getName());

// yields true, since image is reused
System.out.println(result1.getImage() == result2.getImage());
```

第92章：类 - Java反射

java.lang.Class类提供了许多方法，可用于获取元数据、检查和更改类的运行时行为。

java.lang和java.lang.reflect包提供了Java反射的相关类。

使用场景

反射API主要用于：

集成开发环境（IDE），例如Eclipse、MyEclipse、NetBeans等。调试器 测试工具等。

第92.1节：Object类的getClass()方法

```
类 Simple { }

类 Test {
    void printName(Object obj){
        Class c = obj.getClass();
        System.out.println(c.getName());
    }
    public static void main(String args[]){
        Simple s = new Simple();

        Test t = new Test();
        t.printName(s);
    }
}
```

Chapter 92: Class - Java Reflection

The java.lang.Class class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

The java.lang and java.lang.reflect packages provide classes for java reflection.

Where it is used

The Reflection API is mainly used in:

IDE (Integrated Development Environment) e.g. Eclipse, MyEclipse, NetBeans etc. Debugger Test Tools etc.

Section 92.1: getClass() method of Object class

```
class Simple { }

class Test {
    void printName(Object obj){
        Class c = obj.getClass();
        System.out.println(c.getName());
    }
    public static void main(String args[]){
        Simple s = new Simple();

        Test t = new Test();
        t.printName(s);
    }
}
```

第93章：网络

第93.1节：使用

Socket的基本客户端和服务器通信

服务器：启动，并等待传入连接

```
//在端口1234上打开一个监听的"ServerSocket"。
ServerSocket serverSocket = new ServerSocket(1234);

while (true) {
    // 等待客户端连接。
    // 一旦客户端连接，我们将获得一个"Socket"对象// 可用于向新连接的客户端发送和接收消息

    Socket clientSocket = serverSocket.accept();

    // 这里我们将添加处理特定客户端的代码。
}
```

服务器：处理客户端

我们将为每个客户端创建一个单独的线程，以便多个客户端可以同时与服务器交互。只要客户端数量较少（远小于1000个客户端，具体取决于操作系统架构和每个线程的预期负载），这种技术运行良好。

```
new Thread(() -> {
    // 获取socket的输入流，从socket读取字节
    InputStream in = clientSocket.getInputStream();
    // 将输入流包装成读取器，这样可以读取字符串而不是字节
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(in, StandardCharsets.UTF_8));
    // 从套接字读取文本并逐行打印
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}).start();
```

客户端：连接服务器并发送消息

```
// 127.0.0.1 是服务器的地址（这是本地主机地址；即
// 我们自己机器的地址）
// 1234 是服务器监听的端口
Socket socket = new Socket("127.0.0.1", 1234);

// 向套接字写入字符串，并刷新缓冲区
OutputStream outStream = socket.getOutputStream();
PrintWriter writer = new PrintWriter(
    new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
writer.println("Hello world!");
writer.flush();
```

关闭套接字和处理异常

以上示例省略了一些内容以便于阅读。

- 就像文件和其他外部资源一样，告诉操作系统我们已经完成使用它们非常重要。当

Chapter 93: Networking

Section 93.1: Basic Client and Server Communication using a Socket

Server: Start, and wait for incoming connections

```
//Open a listening "ServerSocket" on port 1234.
ServerSocket serverSocket = new ServerSocket(1234);

while (true) {
    // Wait for a client connection.
    // Once a client connected, we get a "Socket" object
    // that can be used to send and receive messages to/from the newly
    // connected client
    Socket clientSocket = serverSocket.accept();

    // Here we'll add the code to handle one specific client.
}
```

Server: Handling clients

We'll handle each client in a separate thread so multiple clients could interact with the server at the same time. This technique works fine as long as the number of clients is low (<< 1000 clients, depending on the OS architecture and the expected load of each thread).

```
new Thread(() -> {
    // Get the socket's InputStream, to read bytes from the socket
    InputStream in = clientSocket.getInputStream();
    // wrap the InputStream in a reader so you can read a String instead of bytes
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(in, StandardCharsets.UTF_8));
    // Read text from the socket and print line by line
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
}).start();
```

Client: Connect to the server and send a message

```
// 127.0.0.1 is the address of the server (this is the localhost address; i.e.
// the address of our own machine)
// 1234 is the port that the server will be listening on
Socket socket = new Socket("127.0.0.1", 1234);

// Write a string into the socket, and flush the buffer
OutputStream outStream = socket.getOutputStream();
PrintWriter writer = new PrintWriter(
    new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
writer.println("Hello world!");
writer.flush();
```

Closing Sockets and Handling Exceptions

The above examples left out some things to make them easier to read.

- Just like files and other external resources, it's important we tell the OS when we're done with them. When

我们完成使用套接字时，调用 `socket.close()` 来正确关闭它。

2. 套接字处理依赖多种外部因素的输入/输出 (I/O) 操作。例如，如果对方突然断开连接怎么办？如果出现网络错误怎么办？这些都是我们无法控制的。这就是为什么许多套接字操作可能会抛出异常，尤其是 `IOException`。

因此，更完整的客户端代码可能如下所示：

```
// "try-with-resources" 会在离开其作用域时关闭套接字
try (Socket socket = new Socket("127.0.0.1", 1234)) {
    OutputStream outStream = socket.getOutputStream();
    PrintWriter writer = new PrintWriter(
        new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
    writer.println("Hello world!");
    writer.flush();
} catch (IOException e) {
    //处理错误
}
```

we're done with a socket, call `socket.close()` to properly close it.

2. Sockets handle I/O (Input/Output) operations that depend on a variety of external factors. For example what if the other side suddenly disconnects? What if there are network error? These things are beyond our control. This is why many socket operations might throw exceptions, especially `IOException`.

A more complete code for the client would therefore be something like this:

```
// "try-with-resources" will close the socket once we leave its scope
try (Socket socket = new Socket("127.0.0.1", 1234)) {
    OutputStream outStream = socket.getOutputStream();
    PrintWriter writer = new PrintWriter(
        new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
    writer.println("Hello world!");
    writer.flush();
} catch (IOException e) {
    //Handle the error
}
```

基础服务器和客户端 - 完整示例

服务器：

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Server {
    public static void main(String args[]) {
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            while (true) {
                // 等待客户端连接。
                Socket clientSocket = serverSocket.accept();

                // 创建并启动一个线程来处理新客户端
                new Thread(() -> {
                    try {
                        // 获得套接字的输入流，以便从套接字读取字节
                        InputStream in = clientSocket.getInputStream();
                        // 将输入流包装成读取器，这样可以读取字符串而不是字节
                        BufferedReader reader = new BufferedReader(
                            new InputStreamReader(in, StandardCharsets.UTF_8));
                        // 从套接字读取并逐行打印
                        String line;
                        while ((line = reader.readLine()) != null) {
                            System.out.println(line);
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    } finally {
                        // 这个finally块确保套接字被关闭。
                        // 不能使用try-with-resources块，因为套接字被传入线程，所以不能
                    }
                });
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // 这个finally块确保套接字被关闭。
            // 不能使用try-with-resources块，因为套接字被传入线程，所以不能
        }
    }
}
```

Basic Server and Client - complete examples

Server:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Server {
    public static void main(String args[]) {
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            while (true) {
                // Wait for a client connection.
                Socket clientSocket = serverSocket.accept();

                // Create and start a thread to handle the new client
                new Thread(() -> {
                    try {
                        // Get the socket's InputStream, to read bytes
                        // from the socket
                        InputStream in = clientSocket.getInputStream();
                        // wrap the InputStream in a reader so you can
                        // read a String instead of bytes
                        BufferedReader reader = new BufferedReader(
                            new InputStreamReader(in, StandardCharsets.UTF_8));
                        // Read from the socket and print line by line
                        String line;
                        while ((line = reader.readLine()) != null) {
                            System.out.println(line);
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    } finally {
                        // This finally block ensures the socket is closed.
                        // A try-with-resources block cannot be used because
                        // the socket is passed into a thread, so it isn't
                    }
                });
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            // This finally block ensures the socket is closed.
            // A try-with-resources block cannot be used because
            // the socket is passed into a thread, so it isn't
        }
    }
}
```

```

        // 在同一代码块中创建并关闭
        try {
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }).start();
}
catch (IOException e) {
e.printStackTrace();
}
}

```

客户端：

```

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Client {
    public static void main(String args[]) {
        try (Socket socket = new Socket("127.0.0.1", 1234)) {
            // 连接到服务器后将执行此代码

            // 向套接字写入字符串，并刷新缓冲区
            OutputStream outStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(
                new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
            writer.println("Hello world!");
            writer.flush();
        } catch (IOException e) {
            // 应该处理异常。
        }
        e.printStackTrace();
    }
}

```

```

        // created and closed in the same block
        try {
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }).start();
}
catch (IOException e) {
e.printStackTrace();
}
}

```

Client:

```

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class Client {
    public static void main(String args[]) {
        try (Socket socket = new Socket("127.0.0.1", 1234)) {
            // We'll reach this code once we've connected to the server

            // Write a string into the socket, and flush the buffer
            OutputStream outStream = socket.getOutputStream();
            PrintWriter writer = new PrintWriter(
                new OutputStreamWriter(outStream, StandardCharsets.UTF_8));
            writer.println("Hello world!");
            writer.flush();
        } catch (IOException e) {
            // Exception should be handled.
            e.printStackTrace();
        }
    }
}

```

第93.2节：使用UDP（数据报）的基本客户端/服务器通信

Client.java

```

import java.io.*;
import java.net.*;

public class Client{
    public static void main(String [] args) throws IOException{
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);

        String ex = "Hello, World!";
        byte[] buf = ex.getBytes();

```

Section 93.2: Basic Client/Server Communication using UDP (Datagram)

Client.java

```

import java.io.*;
import java.net.*;

public class Client{
    public static void main(String [] args) throws IOException{
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress address = InetAddress.getByName(args[0]);

        String ex = "Hello, World!";
        byte[] buf = ex.getBytes();

```

```

        DatagramPacket packet = new DatagramPacket(buf,buf.length, address, 4160);
        clientSocket.send(packet);
    }
}

```

在这种情况下，我们通过参数 (args[0]) 传入服务器的地址。我们使用的端口是4160。

Server.java

```

import java.io.*;
import java.net.*;

public class Server{
    public static void main(String [] args) throws IOException{
        DatagramSocket serverSocket = new DatagramSocket(4160);

        byte[] rbuf = new byte[256];
        DatagramPacket packet = new DatagramPacket(rbuf, rbuf.length);
        serverSocket.receive(packet);
        String response = new String(packet.getData());
        System.out.println("Response: " + response);
    }
}

```

在服务器端，声明一个 DatagramSocket，端口与我们发送消息的端口相同（4160），并等待响应。

第 93.3 节：从

InputStream 加载 TrustStore 和 KeyStore

```

public class TrustLoader {

    public static void main(String args[]) {
        try {
            //获取 ssl/rpgrenadesClient.jks 下信任库文件的输入流//该路径指的是 jar 文件中的 ssl 文件夹,
            //位于与该 jar 文件相同目录下的一个 jar 文件中, 或者位于与该 jar 文件相同目录的不同目录中
            //与该 jar 文件相同目录中的另一个目录
            InputStream stream =
TrustLoader.class.getResourceAsStream("/ssl/rpgrenadesClient.jks");
            // 信任库 (trustStores) 和密钥库 (keyStores) 都由 KeyStore 对象表示
            KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
            // trustStore 的密码
            char[] trustStorePassword = "password".toCharArray();
            // 将信任库加载到对象中
            trustStore.load(stream, trustStorePassword);

            // 定义 SSLContext, 使其使用信任库
            // 获取默认的 SSLContext 以进行修改。
SSLContext context = SSLContext.getInstance("SSL");
            // TrustManagers 持有信任库, 可以添加多个
TrustManagerFactory 工厂 =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
            //将信任库添加到工厂中
factory.init(trustStore);
            //这将传递给 SSLContext 的 init 方法
TrustManager[] 管理器 = factory.getTrustManagers();
            context.init(null, 管理器, null);
            //设置我们新的 SSLContext 以供使用。
SSLContext.setDefault(context);

```

```

        DatagramPacket packet = new DatagramPacket(buf,buf.length, address, 4160);
        clientSocket.send(packet);
    }
}

```

In this case, we pass in the address of the server, via an argument (args[0]). The port we are using is 4160.

Server.java

```

import java.io.*;
import java.net.*;

public class Server{
    public static void main(String [] args) throws IOException{
        DatagramSocket serverSocket = new DatagramSocket(4160);

        byte[] rbuf = new byte[256];
        DatagramPacket packet = new DatagramPacket(rbuf, rbuf.length);
        serverSocket.receive(packet);
        String response = new String(packet.getData());
        System.out.println("Response: " + response);
    }
}

```

On the server-side, declare a DatagramSocket on the same port which we sent our message to (4160) and wait for a response.

Section 93.3: Loading TrustStore and KeyStore from InputStream

```

public class TrustLoader {

    public static void main(String args[]) {
        try {
            //Gets the inputstream of a a trust store file under ssl/rpgrenadesClient.jks
            //This path refers to the ssl folder in the jar file, in a jar file in the same
            directory
            //as this jar file, or a different directory in the same directory as the jar file
            InputStream stream =
TrustLoader.class.getResourceAsStream("/ssl/rpgrenadesClient.jks");
            //Both trustStores and keyStores are represented by the KeyStore object
            KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
            //The password for the trustStore
            char[] trustStorePassword = "password".toCharArray();
            //This loads the trust store into the object
            trustStore.load(stream, trustStorePassword);

            //This is defining the SSLContext so the trust store will be used
            //Getting default SSLContext to edit.
SSLContext context = SSLContext.getInstance("SSL");
            //TrustMangers hold trust stores, more than one can be added
            TrustManagerFactory factory =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
            //Adds the truststore to the factory
            factory.init(trustStore);
            //This is passed to the SSLContext init method
            TrustManager[] managers = factory.getTrustManagers();
            context.init(null, managers, null);
            //Sets our new SSLContext to be used.
            SSLContext.setDefault(context);

```

```

        } catch (KeyStoreException | IOException | NoSuchAlgorithmException
            | CertificateException | KeyManagementException ex) {
            //处理错误
            ex.printStackTrace();
        }
    }
}

```

初始化 KeyStore 的方法相同，只需将对象名称中的任何单词Trust替换为Key。此外，KeyManager[]数组必须作为SSLContext.init的第一个参数传入。即
SSLContext.init(keyManagers, trustManagers, null)

第 93.4 节：Socket 示例 - 使用简单的 socket 读取网页

```

import java.io.*;
import java.net.Socket;

public class Main {

    public static void main(String[] args) throws IOException {//本示例中不处理异常

        //打开一个到 stackoverflow.com, 端口 80 的 socket 连接
        Socket socket = new Socket("stackoverflow.com",80);

        //在发送请求前准备输入、输出流
        OutputStream outStream = socket.getOutputStream();
        InputStream inStream = socket.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inStream));
        PrintWriter writer = new PrintWriter(new BufferedOutputStream(outStream));

        //发送一个基本的 HTTP 头
        writer.print("GET / HTTP/1.1 Host:stackoverflow.com");
        writer.flush();

        //读取响应
        System.out.println(readFully(reader));

        //关闭套接字
        socket.close();
    }

    private static String readFully(Reader in) {
        StringBuilder sb = new StringBuilder();
        int BUFFER_SIZE=1024;
        char[] buffer = new char[BUFFER_SIZE]; // 其他大小
        int charsRead = 0;
        while ( (charsRead = rd.read(buffer, 0, BUFFER_SIZE)) != -1) {
            sb.append(buffer, 0, charsRead);
        }
    }
}

```

你应该会收到一个以 HTTP/1.1 200 OK 开头的响应，这表示正常的HTTP响应，后面跟着HTTP头的其余部分，再后面是以HTML形式的网页原始内容。

注意 readFully() 方法对于防止过早的EOF异常非常重要。网页的最后一行可能缺少换行符以标示行尾，readLine() 会报错，因此必须手动读取或

```

        } catch (KeyStoreException | IOException | NoSuchAlgorithmException
            | CertificateException | KeyManagementException ex) {
            //Handle error
            ex.printStackTrace();
        }
    }
}

```

Initiating a KeyStore works the same, except replace any word Trust in a object name with Key. Additionally, the KeyManager[] array must be passed to the first argument of SSLContext.init. That is
SSLContext.init(keyMangers, trustMangers, null)

Section 93.4: Socket example - reading a web page using a simple socket

```

import java.io.*;
import java.net.Socket;

public class Main {

    public static void main(String[] args) throws IOException {//We don't handle Exceptions in this
example
        //Open a socket to stackoverflow.com, port 80
        Socket socket = new Socket("stackoverflow.com",80);

        //Prepare input, output stream before sending request
        OutputStream outStream = socket.getOutputStream();
        InputStream inStream = socket.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(inStream));
        PrintWriter writer = new PrintWriter(new BufferedOutputStream(outStream));

        //Send a basic HTTP header
        writer.print("GET / HTTP/1.1\nHost:stackoverflow.com\n\n");
        writer.flush();

        //Read the response
        System.out.println(readFully(reader));

        //Close the socket
        socket.close();
    }

    private static String readFully(Reader in) {
        StringBuilder sb = new StringBuilder();
        int BUFFER_SIZE=1024;
        char[] buffer = new char[BUFFER_SIZE]; // or some other size,
        int charsRead = 0;
        while ( (charsRead = rd.read(buffer, 0, BUFFER_SIZE)) != -1) {
            sb.append(buffer, 0, charsRead);
        }
    }
}

```

You should get a response that starts with HTTP/1.1 200 OK, which indicates a normal HTTP response, followed by the rest of the HTTP header, followed by the raw web page in HTML form.

Note the readFully() method is important to prevent a premature EOF exception. The last line of the web page may be missing a return, to signal the end of line, then readLine() will complain, so one must read it by hand or

使用Apache commons-io IOUtils的实用方法

此示例旨在简单演示如何使用套接字连接到现有资源，这并不是访问网页的实用方法。如果需要使用Java访问网页，最好使用现有的HTTP客户端库，如Apache的HTTP Client或Google的HTTP Client

第93.5节：临时禁用SSL验证（用于测试目的）

有时在开发或测试环境中，SSL证书链可能尚未完全建立

(尚未)。

为了继续开发和测试，可以通过安装一个“全信任”的

信任管理器来以编程方式关闭SSL验证：

```
try {
    // 创建一个不验证证书链的信任管理器
    TrustManager[] trustAllCerts = new TrustManager[] {
        new X509TrustManager() {
            public X509Certificate[] getAcceptedIssuers() {
                return null;
            }
            public void checkClientTrusted(X509Certificate[] certs, String authType) {
            }
            public void checkServerTrusted(X509Certificate[] certs, String authType) {
            }
        }
    };

    // 安装全信任的信任管理器
    SSLContext sc = SSLContext.getInstance("SSL");
    sc.init(null, trustAllCerts, new java.security.SecureRandom());
    HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

    // 创建全信任的主机名验证器
    HostnameVerifier allHostsValid = new HostnameVerifier() {
        public boolean verify(String hostname, SSLSession session) {
            return true;
        }
    };

    // 安装全信任的主机验证器
    HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
} catch (NoSuchAlgorithmException | KeyManagementException e) {
    e.printStackTrace();
}
```

第93.6节：使用通道下载文件

如果文件已存在，将被覆盖！

```
String fileName      = "file.zip";           // 文件名
String urlToGetFrom = "http://www.mywebsite.com/"; // 获取文件的URL
String pathToSaveTo = "C:\\\\Users\\\\user\\\\";       // 保存路径

//如果文件已存在，将被覆盖！

//打开目标文件的输出流
```

use utility methods from [Apache commons-io IOUtils](#)

This example is meant as a simple demonstration of connecting to an existing resource using a socket, it's not a practical way of accessing web pages. If you need to access a web page using Java, it's best to use an existing HTTP client library such as [Apache's HTTP Client](#) or [Google's HTTP Client](#)

Section 93.5: Temporarily disable SSL verification (for testing purposes)

Sometimes in a development or testing environment, the SSL certificate chain might not have been fully established (yet).

To continue developing and testing, you can turn off SSL verification programmatically by installing an "all-trusting" trust manager:

```
try {
    // Create a trust manager that does not validate certificate chains
    TrustManager[] trustAllCerts = new TrustManager[] {
        new X509TrustManager() {
            public X509Certificate[] getAcceptedIssuers() {
                return null;
            }
            public void checkClientTrusted(X509Certificate[] certs, String authType) {
            }
            public void checkServerTrusted(X509Certificate[] certs, String authType) {
            }
        }
    };

    // Install the all-trusting trust manager
    SSLContext sc = SSLContext.getInstance("SSL");
    sc.init(null, trustAllCerts, new java.security.SecureRandom());
    HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

    // Create all-trusting host name verifier
    HostnameVerifier allHostsValid = new HostnameVerifier() {
        public boolean verify(String hostname, SSLSession session) {
            return true;
        }
    };

    // Install the all-trusting host verifier
    HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
} catch (NoSuchAlgorithmException | KeyManagementException e) {
    e.printStackTrace();
}
```

Section 93.6: Downloading a file using Channel

If the file already exists, it will be overwritten!

```
String fileName      = "file.zip";           // name of the file
String urlToGetFrom = "http://www.mywebsite.com/"; // URL to get it from
String pathToSaveTo = "C:\\\\Users\\\\user\\\\";       // where to put it

//If the file already exists, it will be overwritten!

//Opening OutputStream to the destination file
```

```

try (ReadableByteChannel rbc =
Channels.newChannel(new URL(urlToGetFrom + fileName).openStream()) ) {
    try ( FileChannel channel =
        new FileOutputStream(pathToSaveTo + fileName).getChannel(); ) {
        channel.transferFrom(rbc, 0, Long.MAX_VALUE);
    }
    catch (FileNotFoundException e) { /* 输出目录未找到 */ }
    catch (IOException e) { /* 文件IO错误 */ }
}
catch (MalformedURLException e) { /* URL格式错误 */ }
catch (IOException e) { /* 连接网站时IO错误 */ }

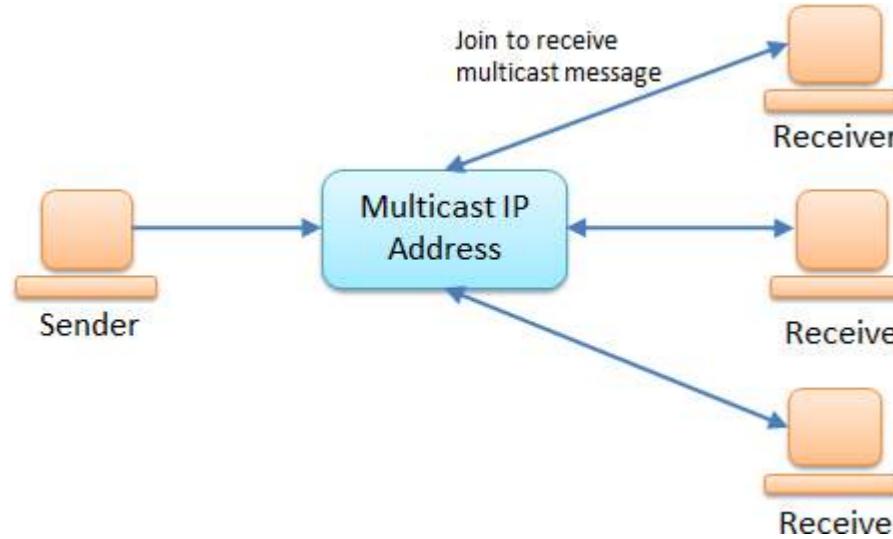
```

注意事项

- 不要让捕获块为空！
- 发生错误时，检查远程文件是否存在
- 这是一个阻塞操作，大文件时可能耗时较长

第93.7节：多播

多播是一种数据报套接字。与普通数据报不同，多播不单独处理每个客户端而是发送到一个IP地址，所有订阅的客户端都会收到该消息。



服务器端示例代码：

```

public class Server {

    private DatagramSocket serverSocket;
    private String ip;
    private int port;

    public Server(String ip, int port) throws SocketException, IOException{
        this.ip = ip;
        this.port = port;
        // 用于发送的套接字
        serverSocket = new DatagramSocket();
    }

    public void send() throws IOException{
        // 创建数据报包
        byte[] message = ("Multicasting...").getBytes();
    }
}

```

```

try (ReadableByteChannel rbc =
Channels.newChannel(new URL(urlToGetFrom + fileName).openStream()) ) {
    try ( FileChannel channel =
        new FileOutputStream(pathToSaveTo + fileName).getChannel(); ) {
        channel.transferFrom(rbc, 0, Long.MAX_VALUE);
    }
    catch (FileNotFoundException e) { /* Output directory not found */ }
    catch (IOException e) { /* File I/O error */ }
}
catch (MalformedURLException e) { /* URL is malformed */ }
catch (IOException e) { /* I/O error connecting to website */ }

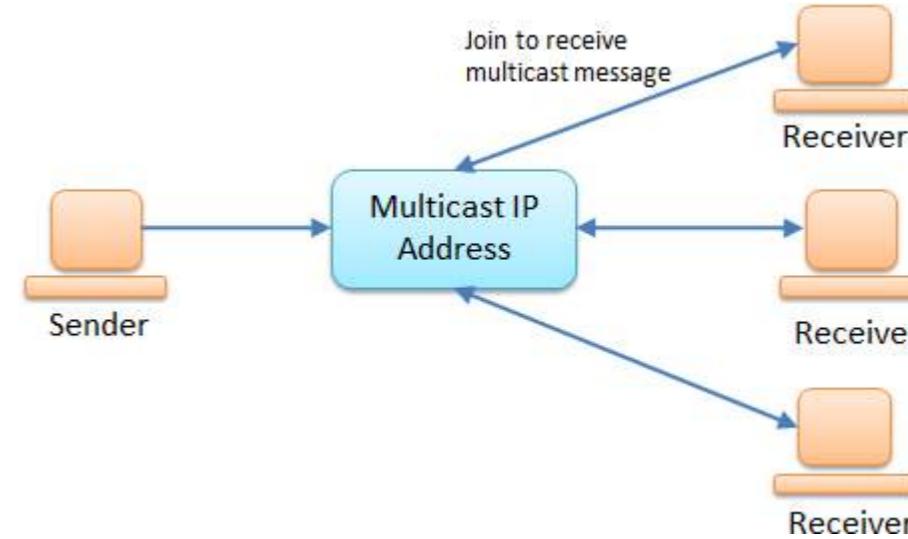
```

Notes

- Don't leave the catch blocks empty!
- In case of error, check if the remote file exists
- This is a blocking operation, can take long time with large files

Section 93.7: Multicasting

Multicasting is a type of Datagram Socket. Unlike regular Datagrams, Multicasting doesn't handle each client individually instead it sends it out to one IP Address and all subscribed clients will get the message.



Example code for a server side:

```

public class Server {

    private DatagramSocket serverSocket;
    private String ip;
    private int port;

    public Server(String ip, int port) throws SocketException, IOException{
        this.ip = ip;
        this.port = port;
        // socket used to send
        serverSocket = new DatagramSocket();
    }

    public void send() throws IOException{
        // make datagram packet
        byte[] message = ("Multicasting...").getBytes();
    }
}

```

```

        DatagramPacket packet = new DatagramPacket(message, message.length,
                InetAddress.getByName(ip), port);
        // 发送数据包
serverSocket.send(packet);
    }

    public void close(){
        serverSocket.close();
    }
}

```

客户端示例代码：

```

public class Client {

    private MulticastSocket socket;

    public Client(String ip, int port) throws IOException {
        // 重要的是这是一个多播套接字
socket = new MulticastSocket(port);

        // 通过ip加入
socket.joinGroup(InetAddress.getByName(ip));
    }

    public void printMessage() throws IOException{
        // 创建数据报包以接收
        byte[] message = new byte[256];
        DatagramPacket packet = new DatagramPacket(message, message.length);

        // 接收数据包
socket.receive(packet);
        System.out.println(new String(packet.getData()));
    }

    public void close(){
        socket.close();
    }
}

```

运行服务器的代码：

```

public static void main(String[] args) {
    try {
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Server server = new Server(ip, port);
        server.send();
server.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

```

运行客户端的代码：

```

public static void main(String[] args) {
    try {
        final String ip = args[0];

```

```

        DatagramPacket packet = new DatagramPacket(message, message.length,
                InetAddress.getByName(ip), port);
        // send packet
serverSocket.send(packet);
    }

    public void close(){
        serverSocket.close();
    }
}

```

Example code for a client side:

```

public class Client {

    private MulticastSocket socket;

    public Client(String ip, int port) throws IOException {
        // important that this is a multicast socket
socket = new MulticastSocket(port);

        // join by ip
socket.joinGroup(InetAddress.getByName(ip));
    }

    public void printMessage() throws IOException{
        // make datagram packet to receive
        byte[] message = new byte[256];
        DatagramPacket packet = new DatagramPacket(message, message.length);

        // receive the packet
socket.receive(packet);
        System.out.println(new String(packet.getData()));
    }

    public void close(){
        socket.close();
    }
}

```

Code for running the Server:

```

public static void main(String[] args) {
    try {
        final String ip = args[0];
        final int port = Integer.parseInt(args[1]);
        Server server = new Server(ip, port);
        server.send();
        server.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

```

Code for running a Client:

```

public static void main(String[] args) {
    try {
        final String ip = args[0];

```

```
final int port = Integer.parseInt(args[1]);
Client client = new Client(ip, port);
client.printMessage();
client.close();
} catch (IOException ex) {
    ex.printStackTrace();
}
}
```

先运行客户端：客户端必须先订阅IP，才能开始接收任何数据包。如果你先启动服务器并调用 `send()`方法，然后再创建客户端（并调用 `printMessage()`），将不会有任何反应，因为客户端是在消息发送之后连接的。

```
final int port = Integer.parseInt(args[1]);
Client client = new Client(ip, port);
client.printMessage();
client.close();
} catch (IOException ex) {
    ex.printStackTrace();
}
}
```

Run the Client First: The Client must subscribe to the IP before it can start receiving any packets. If you start the server and call the `send()` method, and then make a client (& call `printMessage()`). Nothing will happen because the client connected after the message was sent.

第94章：NIO - 网络编程

第94.1节：使用Selector等待事件（以OP_CONNECT为例）

NIO出现在Java 1.4中，引入了“通道（Channels）”的概念，通道被认为比常规的I/O更快。在网络方面，SelectableChannel是最有趣的，因为它允许监控通道的不同状态。它的工作方式类似于C语言中的SELECT()系统调用：当某些类型的事件发生时，我们会被唤醒：

- 连接接收 (OP_ACCEPT)
- 连接建立 (OP_CONNECT)
- 读取FIFO中有数据可用 (OP_READ)
- 数据可以写入FIFO (OP_WRITE)

它允许将检测套接字I/O（是否有数据可读/可写/...）与执行I/O操作（读/写/...）分离。特别是，所有I/O检测可以在单个线程中针对多个套接字（客户端）完成，而执行I/O操作可以在线程池或其他地方处理。这使得应用程序能够轻松扩展以支持更多的连接客户端。

下面的示例展示了基本用法：

1. 创建一个Selector
2. 创建一个SocketChannel
3. 将SocketChannel注册到Selector
4. 使用Selector循环检测事件

```
Selector sel = Selector.open(); // 创建 Selector
SocketChannel sc = SocketChannel open(); // 创建 SocketChannel
sc.configureBlocking(FALSE); // ... 非阻塞
sc.setOption(StandardSocketOptions SO_KEEPALIVE, TRUE); // ... 设置一些选项

// 将通道注册到选择器以便在连接事件上唤醒并使用一些描述作为附件
sc.register(sel, SelectionKey.OP_CONNECT, "连接到 google.com"); // 返回一个 SelectionKey：
SocketChannel 与选择器之间的关联
System.out.println("开始连接");
如果 (sc.connect(NEW InetSocketAddress("www.google.com", 80)))
    System.out.println("已连接"); // 立即连接：无其他操作
否则 {
    BOOLEAN exit = FALSE;
while (!exit) {
    如果 (sel.select(100) == 0) // 在过去的 100毫秒内注册的通道上有事件发生吗？
        continue; // 没有，继续等待
    // 有事件发生...
    Set<SelectionKey> KEYS = sel.selectedKeys(); // 触发了某些已注册操作的 SelectionKeys 列表
    FOR (SelectionKey k : KEYS) {
        System.out.println("正在检查 "+k.attachment());
        IF (k.isConnectable()) { // 连接事件
            System.out.print("通过 select() 连接到 "+k.channel()+" -> ");
            IF (sc.finishConnect()) { // 完成连接过程
                m.out.println("完成！");
                k.interestOps(k.i)
            }
            interestOps() & ~SelectionKey.OP_CONNECT); // 我们已经连接：移除对连接事件的关注
        }
    }
}
System.out.println("所有事件处理完毕");
```

Chapter 94: NIO - Networking

Section 94.1: Using Selector to wait for events (example with OP_CONNECT)

NIO appeared in Java 1.4 and introduced the concept of "Channels", which are supposed to be faster than regular I/O. Network-wise, the [SelectableChannel](#) is the most interesting as it allows to monitor different states of the Channel. It works in a similar manner as the C [SELECT\(\)](#) system call: we get woken-up when certain types of events occur:

- connection received (OP_ACCEPT)
- connection realized (OP_CONNECT)
- data available in read FIFO (OP_READ)
- data can be pushed to write FIFO (OP_WRITE)

It allows for separation between *detecting* socket I/O (something can be read/written/...) and *performing* the I/O (read/write/...). Especially, all I/O detection can be done in a single thread for multiple sockets (clients), while performing I/O can be handled in a thread pool or anywhere else. That allows for an application to scale easily to the number of connected clients.

The following example shows the basics:

1. Create a [Selector](#)
2. Create a [SocketChannel](#)
3. Register the [SocketChannel](#) to the [Selector](#)
4. Loop with the [Selector](#) to detect events

```
Selector sel = Selector.open(); // CREATE the Selector
SocketChannel sc = SocketChannel open(); // CREATE a SocketChannel
sc.configureBlocking(FALSE); // ... non blocking
sc.setOption(StandardSocketOptions SO_KEEPALIVE, TRUE); // ... SET SOME options

// Register the Channel TO the Selector FOR wake-up ON CONNECT event AND USE SOME description AS an attachment
sc.register(sel, SelectionKey.OP_CONNECT, "Connection to google.com"); // RETURNS a SelectionKey: the association BETWEEN the SocketChannel AND the Selector
System.out.println("Initiating connection");
IF (sc.connect(NEW InetSocketAddress("www.google.com", 80)))
    System.out.println("Connected"); // Connected right away: nothing ELSE TO do
ELSE {
    BOOLEAN exit = FALSE;
    while (!exit) {
        IF (sel.select(100) == 0) // Did something happen ON SOME registered Channels during the LAST 100ms?
            continue; // No, wait SOME more

        // Something happened...
        Set<SelectionKey> KEYS = sel.selectedKeys(); // List OF SelectionKeys ON which SOME registered operation was triggered
        FOR (SelectionKey k : KEYS) {
            System.out.println("Checking "+k.attachment());
            IF (k.isConnectable()) { // CONNECT event
                System.out.print("Connected through select() on "+k.channel()+" -> ");
                IF (sc.finishConnect()) { // Finish connection process
                    System.out.println("done!");
                    k.interestOps(k.interestOps() & ~SelectionKey.OP_CONNECT); // We are already connected: remove interest IN CONNECT event
                }
            }
        }
    }
}
```

```

exit = TRUE;
    } ELSE
System.out.println("未完成...");
}
// TODO: ELSE IF (k.isReadable()) { ...
KEYS.clear(); // 必须在处理后清除所选的KEYS SET !
}
System.out.print("断开连接中 ... ");
sc.shutdownOutput(); // 发起优雅断开连接
// TODO: 清空接收缓冲区
sc.close();
System.out.println("完成");

```

将输出如下内容：

```

开始连接
正在检查与 google.com 的连接
通过 'select()' 连接成功, java.nio.channels.SocketChannel[connection-pending
remote=www.google.com/216.58.208.228:80] -> 完成 !
断开连接中 ... 完成

```

```

exit = TRUE;
    } ELSE
System.out.println("unfinished...");
}
// TODO: ELSE IF (k.isReadable()) { ...
KEYS.clear(); // Have TO clear the selected KEYS SET once processed!
}
System.out.print("Disconnecting ... ");
sc.shutdownOutput(); // Initiate graceful disconnection
// TODO: empty receive buffer
sc.close();
System.out.println("done");

```

Would give the following output:

```

Initiating connection
Checking Connection to google.com
Connected through 'select()' on java.nio.channels.SocketChannel[connection-pending
remote=www.google.com/216.58.208.228:80] -> done!
Disconnecting ... done

```

第95章：HttpURLConnection

第95.1节：从URL获取响应体作为字符串

```
String getText(String url) throws IOException {
    HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
    //向连接添加头部，或根据需要检查状态.

    // 处理发生的错误响应码
    int responseCode = conn.getResponseCode();
    InputStream inputStream;
    if (200 <= responseCode && responseCode <= 299) {
        inputStream = connection.getInputStream();
    } else {
        inputStream = connection.getErrorStream();
    }

    BufferedReader in = new BufferedReader(
        new InputStreamReader(
            inputStream));
    String response = new StringBuilder();
    String currentLine;

    while ((currentLine = in.readLine()) != null)
        response.append(currentLine);

    in.close();

    return response.toString();
}
```

这将从指定的URL下载文本数据，并以字符串形式返回。

工作原理：

- 首先，我们通过new URL(url).openConnection()从URL创建一个HttpURLConnection。我们将返回的URLConnection强制转换为HttpURLConnection，这样就可以访问添加请求头（例如User Agent）或检查响应码等功能。（本示例未实现这些，但很容易添加。）
- 然后，根据响应码创建InputStream（用于错误处理）
- 接着，创建一个BufferedReader，用于从连接获取的InputStream中读取文本。
- 现在，我们逐行将文本追加到StringBuilder中。
- 关闭InputStream，并返回我们得到的字符串。

注意事项：

- 如果失败（例如网络错误或无网络连接），此方法将抛出IOException，如果给定的URL无效，还会抛出unchecked MalformedURLException。
- 它可用于读取任何返回文本的URL，例如网页（HTML）、返回JSON或XML的REST API等。
- 另见：[用几行Java代码读取URL到字符串](#)。

Chapter 95: HttpURLConnection

Section 95.1: Get response body from a URL as a String

```
String getText(String url) throws IOException {
    HttpURLConnection connection = (HttpURLConnection) new URL(url).openConnection();
    //add headers to the connection, or check the status if desired.

    // handle error response code it occurs
    int responseCode = conn.getResponseCode();
    InputStream inputStream;
    if (200 <= responseCode && responseCode <= 299) {
        inputStream = connection.getInputStream();
    } else {
        inputStream = connection.getErrorStream();
    }

    BufferedReader in = new BufferedReader(
        new InputStreamReader(
            inputStream));
    String response = new StringBuilder();
    String currentLine;

    while ((currentLine = in.readLine()) != null)
        response.append(currentLine);

    in.close();

    return response.toString();
}
```

This will download text data from the specified URL, and return it as a String.

How this works:

- First, we create a HttpURLConnection from our URL, with new URL(url).openConnection(). We cast the HttpURLConnection this returns to a HttpURLConnection, so we have access to things like adding headers (such as User Agent), or checking the response code. (This example does not do that, but it's easy to add.)
- Then, create InputStream basing on the response code (for error handling)
- Then, create a BufferedReader which allows us to read text from InputStream we get from the connection.
- Now, we append the text to a StringBuilder, line by line.
- Close the InputStream, and return the String we now have.

Notes:

- This method will throw an IOException in case of failure (such as a network error, or no internet connection), and it will also throw an unchecked MalformedURLException if the given URL is not valid.
- It can be used for reading from any URL which returns text, such as webpages (HTML), REST APIs which return JSON or XML, etc.
- See also: [Read URL to String in few lines of Java code](#).

用法：

非常简单：

```
String text = getText("http://example.com");
//对 example.com 的文本进行操作，这里是 HTML.
```

第95.2节：POST数据

```
public static void post(String url, byte [] data, String contentType) throws IOException {
    HttpURLConnection connection = null;
    OutputStream out = null;
    InputStream in = null;

    try {
        connection = (HttpURLConnection) new URL(url).openConnection();
        connection.setRequestProperty("Content-Type", contentType);
        connection.setDoOutput(true);

        out = connection.getOutputStream();
        out.write(data);
        out.close();

        in = connection.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        String line = null;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        in.close();

    } finally {
        if (connection != null) connection.disconnect();
        if (out != null) out.close();
        if (in != null) in.close();
    }
}
```

这将向指定的URL发送POST数据，然后逐行读取响应。

工作原理

- 像往常一样，我们从一个URL获取`HttpURLConnection`。
- 使用`setRequestProperty`设置内容类型，默认是`application/x-www-form-urlencoded`
- `setDoOutput(true)`告诉连接我们将发送数据。
- 然后通过调用`getOutputStream()`获取`OutputStream`并写入数据。完成后别忘了关闭它。
- 最后我们读取服务器响应。

第95.3节：删除资源

```
public static void delete (String urlString, String contentType) throws IOException {
    HttpURLConnection connection = null;

    try {
        URL url = new URL(urlString);
        connection = (HttpURLConnection) url.openConnection();
        connection.setDoInput(true);
```

Usage:

Is very simple:

```
String text = getText("http://example.com");
//Do something with the text from example.com, in this case the HTML.
```

Section 95.2: POST data

```
public static void post(String url, byte [] data, String contentType) throws IOException {
    HttpURLConnection connection = null;
    OutputStream out = null;
    InputStream in = null;

    try {
        connection = (HttpURLConnection) new URL(url).openConnection();
        connection.setRequestProperty("Content-Type", contentType);
        connection.setDoOutput(true);

        out = connection.getOutputStream();
        out.write(data);
        out.close();

        in = connection.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        String line = null;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        in.close();

    } finally {
        if (connection != null) connection.disconnect();
        if (out != null) out.close();
        if (in != null) in.close();
    }
}
```

This will POST data to the specified URL, then read the response line-by-line.

How it works

- As usual we obtain the `HttpURLConnection` from a `URL`.
- Set the content type using `setRequestProperty`, by default it's `application/x-www-form-urlencoded`
- `setDoOutput(true)` tells the connection that we will send data.
- Then we obtain the `OutputStream` by calling `getOutputStream()` and write data to it. Don't forget to close it after you are done.
- At last we read the server response.

Section 95.3: Delete resource

```
public static void delete (String urlString, String contentType) throws IOException {
    HttpURLConnection connection = null;

    try {
        URL url = new URL(urlString);
        connection = (HttpURLConnection) url.openConnection();
        connection.setDoInput(true);
```

```

connection.setRequestMethod("DELETE");
connection.setRequestProperty("Content-Type", contentType);

Map<String, List<String>> map = connection.getHeaderFields();
StringBuilder sb = new StringBuilder();
Iterator<Map.Entry<String, String>> iterator = responseHeader.entrySet().iterator();
while(iterator.hasNext())
{
    Map.Entry<String, String> entry = iterator.next();
    sb.append(entry.getKey());
    sb.append('=').append('\"');
    sb.append(entry.getValue());
    sb.append('\"');
    if(iterator.hasNext())
    {
        sb.append(',').append(' ');
    }
}
System.out.println(sb.toString());

} catch (Exception e) {
e.printStackTrace();
} finally {
    if (connection != null) connection.disconnect();
}
}

```

这将删除指定URL中的资源，然后打印响应头。

工作原理

- 我们从一个URL获取HttpURLConnection。
- 使用setRequestProperty设置内容类型，默认是application/x-www-form-urlencoded
- setDoInput(true)表示连接将用于输入。
- 使用setRequestMethod("DELETE")执行HTTP DELETE操作

最后我们打印服务器响应头。

第95.4节：检查资源是否存在

```

/**
 * 通过发送HEAD请求检查资源是否存在。
 * @param url 需要检查的资源的 URL。
 * @return 如果响应码是 200 OK，则返回 true。
 */
public static final boolean checkIfResourceExists(URL url) throws IOException {
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("HEAD");
    int code = conn.getResponseCode();
    conn.disconnect();
    return code == 200;
}

```

说明：

如果您只是检查资源是否存在，使用 HEAD 请求比 GET 更好。这可以避免传输资源的开销。

请注意，该方法仅在响应码为 200 时返回true。如果您预期会有重定向（即 3XX）响应，那么该方法可能需要增强以支持这些响应。

```

connection.setRequestMethod("DELETE");
connection.setRequestProperty("Content-Type", contentType);

Map<String, List<String>> map = connection.getHeaderFields();
StringBuilder sb = new StringBuilder();
Iterator<Map.Entry<String, String>> iterator = responseHeader.entrySet().iterator();
while(iterator.hasNext())
{
    Map.Entry<String, String> entry = iterator.next();
    sb.append(entry.getKey());
    sb.append('=').append('\"');
    sb.append(entry.getValue());
    sb.append('\"');
    if(iterator.hasNext())
    {
        sb.append(',').append(' ');
    }
}
System.out.println(sb.toString());

} catch (Exception e) {
e.printStackTrace();
} finally {
    if (connection != null) connection.disconnect();
}
}

```

This will DELETE the resource in the specified URL, then print the response header.

How it works

- we obtain the `HttpURLConnection` from a `URL`.
- Set the content type using `setRequestProperty`, by default it's `application/x-www-form-urlencoded`
- `setDoInput(true)` tells the connection that we intend to use the URL connection for input.
- `setRequestMethod("DELETE")` to perform HTTP DELETE

At last we print the server response header.

Section 95.4: Check if resource exists

```

/**
 * Checks if a resource exists by sending a HEAD-Request.
 * @param url The url of a resource which has to be checked.
 * @return true if the response code is 200 OK.
 */
public static final boolean checkIfResourceExists(URL url) throws IOException {
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("HEAD");
    int code = conn.getResponseCode();
    conn.disconnect();
    return code == 200;
}

```

Explanation:

If you are just checking if a resource exists, it better to use a HEAD request than a GET. This avoids the overhead of transferring the resource.

Note that the method only returns `true` if the response code is `200`. If you anticipate redirect (i.e. 3XX) responses, then the method may need to be enhanced to honor them.

示例：

```
checkIfResourceExists(new URL("http://images.google.com/")); // true  
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

Example:

```
checkIfResourceExists(new URL("http://images.google.com/")); // true  
checkIfResourceExists(new URL("http://pictures.google.com/")); // false
```

第96章：JAX-WS

第96.1节：基本认证

使用基本认证进行JAX-WS调用的方法有些不太明显。

下面是一个示例，其中Service是服务类表示，Port是你想要访问的服务端口。

```
Service s = new Service();
Port port = s.getPort();

BindingProvider prov = (BindingProvider)port;
prov.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "myusername");
prov.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "mypassword");

port.call();
```

Chapter 96: JAX-WS

Section 96.1: Basic Authentication

The way to do a JAX-WS call with basic authentication is a little unobvious.

Here is an example where Service is the service class representation and Port is the service port you want to access.

```
Service s = new Service();
Port port = s.getPort();

BindingProvider prov = (BindingProvider)port;
prov.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "myusername");
prov.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "mypassword");

port.call();
```

第97章：Nashorn JavaScript引擎

Nashorn 是由甲骨文用 Java 开发的 JavaScript 引擎，随 Java 8 一起发布。Nashorn 允许通过 JSR-223 在 Java 应用程序中嵌入 JavaScript，并支持开发独立的 JavaScript 应用程序，同时提供更好的运行时性能和更符合 ECMA 标准化的 JavaScript 规范。

第 97.1 节：执行 JavaScript 文件

```
// 必要的导入
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;
import java.io.FileReader;
import java.io.FileNotFoundException;

// 获取 JavaScript 引擎实例
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// 从文件 'demo.js' 加载并执行脚本
try {
    engine.eval(new FileReader("demo.js"));
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (ScriptException ex) {
    // 这是脚本 API 的通用异常子类
    ex.printStackTrace();
}

// 结果：
// "来自文件的脚本！" 打印到标准输出
```

demo.js:

```
print('来自文件的脚本！');
```

第97.2节：拦截脚本输出

```
// 获取JavaScript引擎实例
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// 设置自定义写入器
StringWriter stringWriter = new StringWriter();
// 修改引擎上下文，使自定义写入器成为引擎的默认
// 输出写入器
engine.getContext().setWriter(stringWriter);

// 执行一些脚本
try {
    engine.eval("print('重定向的文本！');");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// 结果：
// 标准输出没有任何打印，但
```

Chapter 97: Nashorn JavaScript engine

[Nashorn](#) is a JavaScript engine developed in Java by Oracle, and has been released with Java 8. Nashorn allows embedding Javascript in Java applications via JSR-223 and allows to develop standalone Javascript applications, and it provides better runtime performance and better compliance with the ECMA normalized Javascript specification.

Section 97.1: Execute JavaScript file

```
// Required imports
import javax.script.ScriptEngineManager;
import javax.script.ScriptEngine;
import javax.script.ScriptException;
import java.io.FileReader;
import java.io.FileNotFoundException;

// Obtain an instance of the JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Load and execute a script from the file 'demo.js'
try {
    engine.eval(new FileReader("demo.js"));
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Script from file!' printed on standard output
```

demo.js:

```
print('Script from file!');
```

Section 97.2: Intercept script output

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Setup a custom writer
StringWriter stringWriter = new StringWriter();
// Modify the engine context so that the custom writer is now the default
// output writer of the engine
engine.getContext().setWriter(stringWriter);

// Execute some script
try {
    engine.eval("print('Redirected text!');");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// Nothing printed on standard output, but
```

```
// stringWriter.toString() 包含 '重定向的文本！'
```

第97.3节：你好，Nashorn

```
// 获取JavaScript引擎实例
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// 执行硬编码脚本
try {
    engine.eval("print('你好，Nashorn！');");
} catch (ScriptException ex) {
    // 这是脚本 API 的通用异常子类
    ex.printStackTrace();
}

// 结果：
// 标准输出打印了 '你好，Nashorn！'
```

```
// stringWriter.toString() contains 'Redirected text!'
```

Section 97.3: Hello Nashorn

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Execute an hardcoded script
try {
    engine.eval("print('Hello Nashorn!');");
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

第97.4节：计算算术字符串

```
// 获取JavaScript引擎实例
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");

// 待计算的字符串
String str = "3+2*4+5";
// 执行带有运算符优先级的算术运算后的值将是16

// 打印该值
try {
    System.out.println(engine.eval(str));
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// 结果：
// 算术计算后字符串的值将打印到标准输出。
// 在此情况下，标准输出将打印“16.0”。
```

Section 97.4: Evaluate Arithmetic Strings

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");

// String to be evaluated
String str = "3+2*4+5";
// Value after doing Arithmetic operation with operator precedence will be 16

// Printing the value
try {
    System.out.println(engine.eval(str));
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// Value of the string after arithmetic evaluation is printed on standard output.
// In this case '16.0' will be printed on standard output.
```

第97.5节：设置全局变量

```
// 获取JavaScript引擎实例
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// 定义一个全局变量
engine.put("textToPrint", "Data defined in Java.");

// 打印全局变量
try {
    engine.eval("print(textToPrint);");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// 结果：
// 在标准输出上打印“Java中定义的数据。”
```

Section 97.5: Set global variables

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

// Define a global variable
engine.put("textToPrint", "Data defined in Java.");

// Print the global variable
try {
    engine.eval("print(textToPrint);");
} catch (ScriptException ex) {
    ex.printStackTrace();
}

// Outcome:
// 'Data defined in Java.' printed on standard output
```

第97.6节：设置和获取全局变量

```
// 获取JavaScript引擎实例
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

try {
    // 在引擎的全局命名空间中设置值
    engine.put("name", "Nashorn");
    // 执行硬编码脚本
    engine.eval("var value='Hello ' + name + '!'");
    // 获取值
    String value=(String)engine.get("value");
    System.out.println(value);
} catch (ScriptException ex) {
    // 这是脚本API的通用异常子类
    ex.printStackTrace();
}

// 结果：
// 标准输出打印了 '你好, Nashorn !'
```

Section 97.6: Set and get global variables

```
// Obtain an instance of JavaScript engine
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("nashorn");

try {
    // Set value in the global name space of the engine
    engine.put("name", "Nashorn");
    // Execute an hardcoded script
    engine.eval("var value='Hello ' + name + '!'");
    // Get value
    String value=(String)engine.get("value");
    System.out.println(value);
} catch (ScriptException ex) {
    // This is the generic Exception subclass for the Scripting API
    ex.printStackTrace();
}

// Outcome:
// 'Hello Nashorn!' printed on standard output
```

第97.7节：Nashorn中JavaScript使用Java对象

可以将Java对象传递给Nashorn引擎以在Java代码中处理。同时，存在一些JavaScript（和Nashorn）特有的结构，且它们与Java对象的交互方式并不总是清晰。

下面的表格描述了JavaScript结构中原生Java对象的行为。

测试的结构：

1. if语句中的表达式。在JS中，if语句中的表达式不必是布尔类型，这与Java不同。对于所谓的假值（null、undefined、0、空字符串等），其结果被评估为false
2. for each语句。Nashorn有一种特殊的循环——for each——可以遍历不同的JS和Java对象。
3. 获取对象大小。在JS中，对象有一个length属性，返回数组或字符串的大小。

结果：

类型	如果	对于每个	.长度
Java 空值	假	无迭代	异常
Java 空字符串	假	无迭代	0
Java 字符串	true	遍历字符串字符	字符串长度
Java 整数/长整数值 != 0	无迭代		undefined
Java ArrayList	true	遍历元素	列表长度
Java HashMap	true	遍历值	null
Java HashSet	true	遍历项	undefined

建议：

- 建议使用if(some_string)来检查字符串是否非空且非null for each可以安全地用于遍历
- 任何集合，且如果集合不可迭代、为null或undefined，也不会抛出异常
- 在获取对象长度之前，必须检查其是否为null或undefined（对于调用Java对象的方法或获取其属性的任何尝试也是如此）

Section 97.7: Usage of Java objects in JavaScript in Nashorn

It's possible to pass Java objects to Nashorn engine to be processed in Java code. At the same time, there are some JavaScript (and Nashorn) specific constructions, and it's not always clear how they work with java objects.

Below there is a table which describes behaviour of native Java objects inside JavaScript constructions.

Tested constructions:

1. Expression in if clause. In JS expression in if clause doesn't have to be boolean unlike Java. It's evaluated as false for so called falsy values (null, undefined, 0, empty strings etc)
2. for each statement Nashorn has a special kind of loop - for each - which can iterate over different JS and Java object.
3. Getting object size. In JS objects have a property length, which returns size of an array or a string.

Results:

Type	If	for each	.length
Java null	false	No iterations	Exception
Java empty string	false	No iterations	0
Java string	true	Iterates over string characters	Length of the string
Java Integer/Long value != 0	No iterations		undefined
Java ArrayList	true	Iterates over elements	Length of the list
Java HashMap	true	Iterates over values	null
Java HashSet	true	Iterates over items	undefined

Recommendations:

- It's advisable to use if (some_string) to check if a string is not null and not empty
- for each can be safely used to iterate over any collection, and it doesn't raise exceptions if the collection is not iterable, null or undefined
- Before getting length of an object it must be checked for null or undefined (the same is true for any attempt of calling a method or getting a property of Java object)

第97.8节：从脚本实现接口

```
import java.io.FileReader;
import java.io.IOException;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class InterfaceImplementationExample {
    public static interface Pet {
        public void eat();
    }

    public static void main(String[] args) throws IOException {
        // 获取JavaScript引擎实例
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        try {
            // 执行脚本
            /* pet.js */
            /*
            var Pet = Java.type("InterfaceImplementationExample.Pet");
            new Pet()
            eat: function() { print("eat"); }
            */

            Pet pet = (Pet) engine.eval(new FileReader("pet.js"));

            pet.eat();
        } catch (ScriptException ex) {
            ex.printStackTrace();
        }

        // 结果：
        // 'eat' printed on standard output
    }
}
```

Section 97.8: Implementing an interface from script

```
import java.io.FileReader;
import java.io.IOException;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class InterfaceImplementationExample {
    public static interface Pet {
        public void eat();
    }

    public static void main(String[] args) throws IOException {
        // Obtain an instance of JavaScript engine
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("nashorn");

        try {
            //evaluate a script
            /* pet.js */
            /*
            var Pet = Java.type("InterfaceImplementationExample.Pet");
            new Pet()
            eat: function() { print("eat"); }
            */

            Pet pet = (Pet) engine.eval(new FileReader("pet.js"));

            pet.eat();
        } catch (ScriptException ex) {
            ex.printStackTrace();
        }

        // Outcome:
        // 'eat' printed on standard output
    }
}
```

第98章：Java本地接口

参数	详情
JNIEnv	指向JNI环境的指针
jobject	调用非静态本地方法的对象
jclass	调用static native方法的类

第98.1节：从Java调用C++方法

Java中的静态方法和成员方法可以标记为native，表示它们的实现位于共享库文件中。执行本地方方法时，JVM会在已加载的库中查找对应的函数（参见加载本地库），使用简单的名称修饰方案，执行参数转换和栈设置，然后将控制权交给本地代码。

Java代码

```
/** com/example/jni/JNIJava.java **/ 

包com.example.jni;

公共类JNIJava{
    静态{
        System.loadLibrary("libJNI_CPP");
    }

    // 显然，本地方法在Java中不能定义方法体
    公共本地void printString(String name);
    公共静态本地double average(int[] nums);

    公共静态void main(final String[] args) {
        JNIJava jniJava = new JNIJava();
        jniJava.printString("从Java调用C++的'printString'");

        double d = average(new int[]{1, 2, 3, 4, 7});
        System.out.println("从C++的'average'获得结果: " + d);
    }
}
```

C++ 代码

应使用javah工具针对目标类生成包含本地函数声明的头文件。
在构建目录下运行以下命令：

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

... 生成以下头文件（为简洁起见，已去除注释）：

```
// com_example_jni_JNIJava.hpp

/* 请勿编辑此文件 - 它是机器生成的 */
#include <jni.h> // JNI API 声明

#ifndef _Included_com_example_jni_JNIJava
#define _Included_com_example_jni_JNIJava
#ifndef __cplusplus
extern "C" { // 如果使用C++编译器，这一点绝对必要
#endif

```

Chapter 98: Java Native Interface

Parameter	Details
JNIEnv	Pointer to the JNI environment
jobject	The object which invoked the non-static native method
jclass	The class which invoked the static native method

Section 98.1: Calling C++ methods from Java

Static and member methods in Java can be marked as *native* to indicate that their implementation is to be found in a shared library file. Upon execution of a native method, the JVM looks for a corresponding function in loaded libraries (see Loading native libraries), using a simple name mangling scheme, performs argument conversion and stack setup, then hands over control to native code.

Java code

```
/** com/example/jni/JNIJava.java **/

package com.example.jni;

public class JNIJava {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    // Obviously, native methods may not have a body defined in Java
    public native void printString(String name);
    public static native double average(int[] nums);

    public static void main(final String[] args) {
        JNIJava jniJava = new JNIJava();
        jniJava.printString("Invoked C++ 'printString' from Java");

        double d = average(new int[]{1, 2, 3, 4, 7});
        System.out.println("Got result from C++ 'average': " + d);
    }
}
```

C++ code

Header files containing native function declarations should be generated using the javah tool on target classes.
Running the following command at the build directory :

```
javah -o com_example_jni_JNIJava.hpp com.example.jni.JNIJava
```

... produces the following header file (comments stripped for brevity) :

```
// com_example_jni_JNIJava.hpp

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h> // The JNI API declarations

#ifndef _Included_com_example_jni_JNIJava
#define _Included_com_example_jni_JNIJava
#ifndef __cplusplus
extern "C" { // This is absolutely required if using a C++ compiler
#endif

```

```

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString
(JNIEnv *, jobject, jstring);

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average
(JNIEnv *, jclass, jintArray);

#ifndef __cplusplus
}
#endif
#endif

```

这是一个示例实现：

```

// com_example_jni_JNIJava.cpp

#include <iostream>
#include "com_example_jni_JNIJava.hpp"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString(JNIEnv *env, jobject jthis, jstring string) {
    const char *stringInC = env->GetStringUTFChars(string, NULL);
    if (NULL == stringInC)
        return;
    cout << stringInC << endl;
    env->ReleaseStringUTFChars(string, stringInC);
}

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average(JNIEnv *env, jclass jthis, jintArray intArray) {
    jint *intArrayInC = env->GetIntArrayElements(intArray, NULL);
    if (NULL == intArrayInC)
        return -1;
    jsize length = env->GetArrayLength(intArray);
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += intArrayInC[i];
    }
    env->ReleaseIntArrayElements(intArray, intArrayInC, 0);
    return (double) sum / length;
}

```

输出

运行上述示例类将产生以下输出：

从Java调用了C++的'printString'
从C++的'average'获得结果：3.4

第98.2节：从C++调用Java方法（回调）

从本地代码调用Java方法是一个两步过程：

1. 使用方法名和描述符，通过GetMethodID JNI函数获取方法指针；
2. 调用列出的Call*Method函数之一。

```

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString
(JNIEnv *, jobject, jstring);

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average
(JNIEnv *, jclass, jintArray);

#ifndef __cplusplus
}
#endif
#endif

```

Here is an example implementation :

```

// com_example_jni_JNIJava.cpp

#include <iostream>
#include "com_example_jni_JNIJava.hpp"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJava_printString(JNIEnv *env, jobject jthis, jstring string) {
    const char *stringInC = env->GetStringUTFChars(string, NULL);
    if (NULL == stringInC)
        return;
    cout << stringInC << endl;
    env->ReleaseStringUTFChars(string, stringInC);
}

JNIEXPORT jdouble JNICALL Java_com_example_jni_JNIJava_average(JNIEnv *env, jclass jthis, jintArray intArray) {
    jint *intArrayInC = env->GetIntArrayElements(intArray, NULL);
    if (NULL == intArrayInC)
        return -1;
    jsize length = env->GetArrayLength(intArray);
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += intArrayInC[i];
    }
    env->ReleaseIntArrayElements(intArray, intArrayInC, 0);
    return (double) sum / length;
}

```

Output

Running the example class above yields the following output :

Invoked C++ 'printString' from Java
Got result from C++ 'average': 3.4

Section 98.2: Calling Java methods from C++ (callback)

Calling a Java method from native code is a two-step process :

1. obtain a method pointer with the GetMethodID JNI function, using the method name and descriptor ;
2. call one of the Call*Method functions listed [here](#).

Java代码

```
/** com.example.jni.JNIJavaCallback.java **/ 

包com.example.jni;

公共类 JNIJavaCallback {
    静态 {
        System.loadLibrary("libJNI_CPP");
    }

    公共静态 无返回值 main(字符串[] 参数) {
        新建 JNIJavaCallback().回调();
    }

    公共本地 无返回值 回调();

    公共静态 无返回值 打印数字(整数 i) {
        系统输出.打印行("从C++获得的整数: " + i);
    }

    公共 无返回值 打印浮点数(浮点数 i) {
        系统输出.打印行("从C++获得的浮点数: " + i);
    }
}
```

C++ 代码

```
// com_example_jni_JNICppCallback.cpp

#include <iostream>
#include "com_example_jni_JNIJavaCallback.h"

使用命名空间 std;

JNIEXPORT 无返回值 JNICALL Java_com_example_jni_JNIJavaCallback_callback(JNIEnv *env, jobject jthis) {
    jclass thisClass = env->获取对象类(jthis);

    jmethodID printFloat = env->GetMethodID(thisClass, "printFloat", "(F)V");
    if (NULL == printFloat)
        return;
    env->CallVoidMethod(jthis, printFloat, 5.221);

    jmethodID staticPrintInt = env->GetStaticMethodID(thisClass, "printNum", "(I)V");
    if (NULL == staticPrintInt)
        return;
    env->CallVoidMethod(jthis, staticPrintInt, 17);
}
```

输出

```
从C++获取的浮点数: 5.221
从C++获取的整数: 17
```

正在获取描述符

描述符（或内部类型签名）是通过对编译后的.class文件使用javap程序获得的。以下是javap -p -s com.example.jni.JNIJavaCallback的输出：

Java code

```
/** com.example.jni.JNIJavaCallback.java **/ 

package com.example.jni;

public class JNIJavaCallback {
    static {
        System.loadLibrary("libJNI_CPP");
    }

    public static void main(String[] args) {
        new JNIJavaCallback().callback();
    }

    public native void callback();

    public static void printNum(int i) {
        System.out.println("Got int from C++: " + i);
    }

    public void printFloat(float i) {
        System.out.println("Got float from C++: " + i);
    }
}
```

C++ code

```
// com_example_jni_JNICppCallback.cpp

#include <iostream>
#include "com_example_jni_JNIJavaCallback.h"

using namespace std;

JNIEXPORT void JNICALL Java_com_example_jni_JNIJavaCallback_callback(JNIEnv *env, jobject jthis) {
    jclass thisClass = env->GetObjectClass(jthis);

    jmethodID printFloat = env->GetMethodID(thisClass, "printFloat", "(F)V");
    if (NULL == printFloat)
        return;
    env->CallVoidMethod(jthis, printFloat, 5.221);

    jmethodID staticPrintInt = env->GetStaticMethodID(thisClass, "printNum", "(I)V");
    if (NULL == staticPrintInt)
        return;
    env->CallVoidMethod(jthis, staticPrintInt, 17);
}
```

Output

```
Got float from C++: 5.221
Got int from C++: 17
```

Getting the descriptor

Descriptors (or *internal type signatures*) are obtained using the **javap** program on the compiled **.class** file. Here is the output of **javap -p -s com.example.jni.JNIJavaCallback**:

```

编译自 "JNIJavaCallback.java"
公共类 com.example.jni.JNIJavaCallback {
    静态 {}
    描述符: ()V

    公共 com.example.jni.JNIJavaCallback();
    描述符: ()V

    公共静态void main(java.lang.String[]);
    描述符: ([Ljava/lang/String;)V

    public native void callback();
    descriptor: ()V

    public static void printNum(int);
    descriptor: (I)V // <---- 需要的

    public void printFloat(float);
    descriptor: (F)V // <---- 需要的
}

```

第98.3节：加载本地库

在Java中加载共享库文件的常用惯用法如下：

```

public class ClassWithNativeMethods {
    static {
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
}

```

对System.loadLibrary的调用几乎总是静态的，以便在类加载期间发生，确保在共享库加载之前没有本地方法可以执行。然而，以下情况是可能的：

```

public class ClassWithNativeMethods {
    // 在使用任何本地方法之前调用此方法
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
}

```

这允许延迟加载共享库直到必要时，但需要额外注意以避免java.lang.UnsatisfiedLinkErrors。

目标文件查找

共享库文件会在由 java.library.path 系统属性定义的路径中搜索，该属性可以通过运行时的 -Djava.library.path= JVM 参数覆盖：

```
java -Djava.library.path=path/to/lib:/path/to/other/lib MainClassWithNativeMethods
```

注意系统路径分隔符：例如，Windows 使用 ; 而不是 :。

请注意，System.loadLibrary 以平台相关的方式解析库文件名：上面的代码片段期望在 Linux 上有一个名为 libExample.so 的文件，在 Windows 上则是 Example.dll。

```

Compiled from "JNIJavaCallback.java"
public class com.example.jni.JNIJavaCallback {
    static {}
    descriptor: ()V

    public com.example.jni.JNIJavaCallback();
    descriptor: ()V

    public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V

    public native void callback();
    descriptor: ()V

    public static void printNum(int);
    descriptor: (I)V // <---- Needed

    public void printFloat(float);
    descriptor: (F)V // <---- Needed
}

```

Section 98.3: Loading native libraries

The common idiom for loading shared library files in Java is the following :

```

public class ClassWithNativeMethods {
    static {
        System.loadLibrary("Example");
    }

    public native void someNativeMethod(String arg);
    ...
}

```

Calls to System.loadLibrary are almost always static so as to occur during class loading, ensuring that no native method can execute before the shared library has been loaded. However the following is possible :

```

public class ClassWithNativeMethods {
    // Call this before using any native method
    public static void prepareNativeMethods() {
        System.loadLibrary("Example");
    }

    ...
}

```

This allows to defer shared library loading until necessary, but requires extra care to avoid java.lang.UnsatisfiedLinkErrors.

Target file lookup

Shared library files are searched for in the paths defined by the java.library.path system property, which can be overridden using the -Djava.library.path= JVM argument at runtime :

```
java -Djava.library.path=path/to/lib:/path/to/other/lib MainClassWithNativeMethods
```

Watch out for system path separators : for example, Windows uses ; instead of :.

Note that System.loadLibrary resolves library filenames in a platform-dependent manner : the code snippet above expects a file named libExample.so on Linux, and Example.dll on Windows.

`System.loadLibrary` 的替代方法是 `System.load(String)`，它接受共享库文件的完整路径，绕过 `java.library.path` 的查找：

```
public class ClassWithNativeMethods {  
    static {  
        System.load("/path/to/lib/libExample.so");  
    }  
  
    ...
```

An alternative to `System.loadLibrary` is `System.load(String)`, which takes the full path to a shared library file, circumventing the `java.library.path` lookup :

```
public class ClassWithNativeMethods {  
    static {  
        System.load("/path/to/lib/libExample.so");  
    }  
  
    ...
```

第99章：函数式接口

在Java 8及以上版本中，**函数式接口**是指仅有一个抽象方法的接口（不包括Object的方法）。参见JLS §9.8. 函数式接口。

第99.1节：按签名分类的标准Java运行时库函数式接口列表

参数类型	返回类型	接口
()	void	Runnable
()	T	Supplier
()	boolean	BooleanSupplier
()	int	IntSupplier
()	long	LongSupplier
()	double	DoubleSupplier
(T)	void	Consumer<T>
(T)	T	UnaryOperator<T>
(T)	R	Function<T,R>
(T)	boolean	Predicate<T>
(T)	int	ToIntFunction<T>
(T)	long	ToLongFunction<T>
(T)	double	ToDoubleFunction<T>
(T, T)	T	BinaryOperator<T>
(T, U)	void	BiConsumer<T,U>
(T, U)	R	BiFunction<T,U,R>
(T, U)	boolean	BiPredicate<T,U>
(T, U)	int	ToIntBiFunction<T,U>
(T, U)	long	ToLongBiFunction<T,U>
(T, U)	double	ToDoubleBiFunction<T,U>
(T, int)	void	ObjIntConsumer<T>
(T, long)	void	ObjLongConsumer<T>
(T, double)	void	ObjDoubleConsumer<T>
(int)	void	IntConsumer
(int)	R	IntFunction<R>
(int)	boolean	IntPredicate
(int)	int	IntUnaryOperator
(int)	long	IntToLongFunction
(int)	double	InttoDoubleFunction
(int, int)	int	IntBinaryOperator
(long)	void	LongConsumer
(long)	R	LongFunction<R>
(long)	boolean	LongPredicate
(long)	int	LongToIntFunction
(long)	long	LongUnaryOperator
(long)	double	LongtoDoubleFunction
(long, long)	long	LongBinaryOperator

Chapter 99: Functional Interfaces

In Java 8+, a *functional interface* is an interface that has just one abstract method (aside from the methods of Object). See JLS §9.8. Functional Interfaces.

Section 99.1: List of standard Java Runtime Library functional interfaces by signature

Parameter Types	Return Type	Interface
()	void	Runnable
()	T	Supplier
()	boolean	BooleanSupplier
()	int	IntSupplier
()	long	LongSupplier
()	double	DoubleSupplier
(T)	void	Consumer<T>
(T)	T	UnaryOperator<T>
(T)	R	Function<T,R>
(T)	boolean	Predicate<T>
(T)	int	ToIntFunction<T>
(T)	long	ToLongFunction<T>
(T)	double	ToDoubleFunction<T>
(T, T)	T	BinaryOperator<T>
(T, U)	void	BiConsumer<T,U>
(T, U)	R	BiFunction<T,U,R>
(T, U)	boolean	BiPredicate<T,U>
(T, U)	int	ToIntBiFunction<T,U>
(T, U)	long	ToLongBiFunction<T,U>
(T, U)	double	ToDoubleBiFunction<T,U>
(T, int)	void	ObjIntConsumer<T>
(T, long)	void	ObjLongConsumer<T>
(T, double)	void	ObjDoubleConsumer<T>
(int)	void	IntConsumer
(int)	R	IntFunction<R>
(int)	boolean	IntPredicate
(int)	int	IntUnaryOperator
(int)	long	IntToLongFunction
(int)	double	InttoDoubleFunction
(int, int)	int	IntBinaryOperator
(long)	void	LongConsumer
(long)	R	LongFunction<R>
(long)	boolean	LongPredicate
(long)	int	LongToIntFunction
(long)	long	LongUnaryOperator
(long)	double	LongtoDoubleFunction
(long, long)	long	LongBinaryOperator

(double)	void	DoubleConsumer
(double)	R	DoubleFunction<R>
(double)	boolean	DoublePredicate
(double)	int	DoubleToIntFunction
(double)	long	DoubleToLongFunction
(double)	double	DoubleUnaryOperator
(double, double)	double	DoubleBinaryOperator

(double)	void	DoubleConsumer
(double)	R	DoubleFunction<R>
(double)	boolean	DoublePredicate
(double)	int	DoubleToIntFunction
(double)	long	DoubleToLongFunction
(double)	double	DoubleUnaryOperator
(double, double)	double	DoubleBinaryOperator

第100章：流畅接口

第100.1节：流畅的编程风格

在流畅的编程风格中，你从流畅（setter）方法返回this，而在非流畅编程风格中这些方法不会返回任何内容。

这允许你链式调用不同的方法，使代码更简短且更易于开发者处理。

考虑以下非流畅代码：

```
public class Person {  
    private String firstName;  
    private String lastName;  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public String whoAreYou() {  
        return "我是 " + firstName + " " + lastName;  
    }  
  
    public static void main(String[] args) {  
        Person person = new Person();  
        person.setFirstName("约翰");  
        person.setLastName("多伊");  
        System.out.println(person.whoAreYou());  
    }  
}
```

由于设置方法不返回任何内容，我们需要在main方法中使用4条指令来实例化一个带有
一些数据的Person对象并打印它。使用流畅风格，这段代码可以改为：

```
public class Person {  
    private String firstName;  
    private String lastName;  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public Person withFirstName(String firstName) {  
        this.firstName = firstName;  
        return this;  
    }  
}
```

Chapter 100: Fluent Interface

Section 100.1: Fluent programming style

In fluent programming style you return **this** from fluent (setter) methods that would return nothing in non-fluent programming style.

This allows you to chain the different method calls which makes your code shorter and easier to handle for the developers.

Consider this non-fluent code:

```
public class Person {  
    private String firstName;  
    private String lastName;  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public String whoAreYou() {  
        return "I am " + firstName + " " + lastName;  
    }  
  
    public static void main(String[] args) {  
        Person person = new Person();  
        person.setFirstName("John");  
        person.setLastName("Doe");  
        System.out.println(person.whoAreYou());  
    }  
}
```

As the setter methods don't return anything, we need 4 instructions in the `main` method to instantiate a `Person` with some data and print it. With a fluent style this code can be changed to:

```
public class Person {  
    private String firstName;  
    private String lastName;  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public Person withFirstName(String firstName) {  
        this.firstName = firstName;  
        return this;  
    }  
}
```

```

}

public String getLastName() {
    return lastName;
}

public Person withLastName(String lastName) {
    this.lastName = lastName;
    return this;
}

public String whoAreYou() {
    return "我是 " + firstName + " " + lastName;
}

public static void main(String[] args) {
    System.out.println(new Person().withFirstName("John")
        .withLastName("Doe").whoAreYou());
}

```

这个想法是始终返回某个对象，以便构建方法调用链，并使用反映自然语言的方法名称。这种流畅的风格使代码更易读。

第100.2节：Truth - 流畅测试框架

摘自“如何使用Truth” <http://google.github.io/truth/>

```

String string = "awesome";
assertThat(string).startsWith("awe");
assertWithMessage("Without me, it's just aweso").that(string).contains("me");

Iterable<Color> googleColors = googleLogo.getColors();
assertThat(googleColors)
    .containsExactly(BLUE, RED, YELLOW, BLUE, GREEN, RED)
    .inOrder();

```

```

}

public String getLastName() {
    return lastName;
}

public Person withLastName(String lastName) {
    this.lastName = lastName;
    return this;
}

public String whoAreYou() {
    return "I am " + firstName + " " + lastName;
}

public static void main(String[] args) {
    System.out.println(new Person().withFirstName("John")
        .withLastName("Doe").whoAreYou());
}

```

The idea is to always return some object to enable building of a method call chain and to use method names which reflect natural speaking. This fluent style makes the code more readable.

Section 100.2: Truth - Fluent Testing Framework

From "How to use Truth" <http://google.github.io/truth/>

```

String string = "awesome";
assertThat(string).startsWith("awe");
assertWithMessage("Without me, it's just aweso").that(string).contains("me");

Iterable<Color> googleColors = googleLogo.getColors();
assertThat(googleColors)
    .containsExactly(BLUE, RED, YELLOW, BLUE, GREEN, RED)
    .inOrder();

```

第101章：远程方法调用 (RMI)

第101.1节：回调：在“客户端”上调用方法

概述

在此示例中，两个客户端通过服务器相互发送信息。一个客户端向服务器发送一个数字，该数字被转发给第二个客户端。第二个客户端将数字减半后通过服务器发送回第一个客户端。第一个客户端也执行相同操作。当服务器从任一客户端收到的返回数字小于10时，通信停止。服务器返回给客户端的值（数字转换为字符串表示）随后回溯该过程。

1. 登录服务器将自身绑定到注册表。
2. 客户端查找登录服务器并使用其信息调用login方法。然后：
 - 登录服务器存储客户端信息。它包括带有回调方法的客户端存根。
 - 登录服务器创建并返回一个服务器存根（“连接”或“会话”）给客户端以供存储。它包括带有其方法的服务器存根，其中包含一个logout方法（本例中未使用）。
3. 客户端调用服务器的passInt方法，传入接收客户端的名称和一个int值。
4. 服务器调用接收客户端的half方法，传入该int值。这启动了一个来回的（调用和回调）通信，直到服务器停止。

共享的远程接口

登录服务器：

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteLogin extends Remote {
    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}
```

服务器端：

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteConnection extends Remote {
    void logout() throws RemoteException;
    String passInt(String name, int i) throws RemoteException;
}
```

客户端：

```
package callbackRemote;

import java.rmi.Remote;
```

Chapter 101: Remote Method Invocation (RMI)

Section 101.1: Callback: invoking methods on a "client"

Overview

In this example 2 clients send information to each other through a server. One client sends the server a number which is relayed to the second client. The second client halves the number and sends it back to the first client through the server. The first client does the same. The server stops the communication when the number returned to it by any of the clients is less than 10. The return value from the server to the clients (the number it got converted to string representation) then backtracks the process.

1. A login server binds itself to a registry.
2. A client looks up the login server and calls the login method with its information. Then:
 - The login server stores the client information. It includes the client's stub with the callback methods.
 - The login server creates and returns a server stub ("connection" or "session") to the client to store. It includes the server's stub with its methods including a logout method (unused in this example).
3. A client calls the server's passInt with the name of the recipient client and an int.
4. The server calls the half on the recipient client with that int. This initiates a back-and-forth (calls and callbacks) communication until stopped by the server.

The shared remote interfaces

The login server:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteLogin extends Remote {
    RemoteConnection login(String name, RemoteClient client) throws RemoteException;
}
```

The server:

```
package callbackRemote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteConnection extends Remote {
    void logout() throws RemoteException;
    String passInt(String name, int i) throws RemoteException;
}
```

The client:

```
package callbackRemote;

import java.rmi.Remote;
```

```

import java.rmi.RemoteException;

public interface RemoteClient extends Remote {
    void half(int i) throws RemoteException;
}

```

实现类

登录服务器：

```

package callbackServer;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class LoginServer implements RemoteLogin {
    static Map<String, RemoteClient> clients = new HashMap<>();

    @Override
    public RemoteConnection login(String name, RemoteClient client) {
        Connection connection = new Connection(name, client);
        clients.put(name, client);
        System.out.println(name + " logged in");
        return connection;
    }

    public static void main(String[] args) {
        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            LoginServer server = new LoginServer();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("LoginServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

```

服务器端：

```

package callbackServer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.Unreferenced;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;

```

```

import java.rmi.RemoteException;

public interface RemoteClient extends Remote {
    void half(int i) throws RemoteException;
}

```

The implementations

The login server:

```

package callbackServer;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class LoginServer implements RemoteLogin {
    static Map<String, RemoteClient> clients = new HashMap<>();

    @Override
    public RemoteConnection login(String name, RemoteClient client) {
        Connection connection = new Connection(name, client);
        clients.put(name, client);
        System.out.println(name + " logged in");
        return connection;
    }

    public static void main(String[] args) {
        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            LoginServer server = new LoginServer();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("LoginServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

```

The server:

```

package callbackServer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.Unreferenced;

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;

```

```

public class Connection implements RemoteConnection, Unreferenced {

    RemoteClient client;
    String name;

    public Connection(String name, RemoteClient client) {

        this.client = client;
        this.name = name;
        try {
            UnicastRemoteObject.exportObject(this, Registry.REGISTRY_PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void unreferenced() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void logout() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String passInt(String recipient, int i) {

        System.out.println("服务器收到来自 " + name + ":" + i);
        if (i < 10)
            return String.valueOf(i);
        RemoteClient client = LoginServer.clients.get(recipient);
        try {
            client.half(i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return String.valueOf(i);
    }
}

```

客户端：

```

package callbackClient;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

```

```

public class Connection implements RemoteConnection, Unreferenced {

    RemoteClient client;
    String name;

    public Connection(String name, RemoteClient client) {

        this.client = client;
        this.name = name;
        try {
            UnicastRemoteObject.exportObject(this, Registry.REGISTRY_PORT);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void unreferenced() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void logout() {

        try {
            UnicastRemoteObject.unexportObject(this, true);
        } catch (NoSuchObjectException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String passInt(String recipient, int i) {

        System.out.println("Server received from " + name + ":" + i);
        if (i < 10)
            return String.valueOf(i);
        RemoteClient client = LoginServer.clients.get(recipient);
        try {
            client.half(i);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
        return String.valueOf(i);
    }
}

```

The client:

```

package callbackClient;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

```

```

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class Client implements RemoteClient {

    RemoteConnection connection;
    String name, target;

    Client(String name, String target) {

        this.name = name;
        this.target = target;
    }

    public static void main(String[] args) {

        Client client = new Client(args[0], args[1]);
        try {
            Registry reg = LocateRegistry.getRegistry();
            RemoteLogin login = (RemoteLogin) reg.lookup("LoginServerName");
            UnicastRemoteObject.exportObject(client, Integer.parseInt(args[2]));
            client.connection = login.login(client.name, client);
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }
    }

    if ("Client1".equals(client.name)) {
        try {
            client.connection.passInt(client.target, 120);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

@Override
public void half(int i) throws RemoteException {

    String result = connection.passInt(target, i / 2);
    System.out.println(name + " received: " + result + "\n");
}
}

```

运行示例：

1. 运行登录服务器。
2. 使用参数Client2 Client1 1097运行客户端。
3. 使用参数Client1 Client2 1098运行客户端。

由于有3个JVM，输出将显示在3个控制台中。这里将它们合并显示：

```

Client2 已登录
Client1 已登录
服务器收到来自 Client1 的数据 : 120
服务器收到来自 Client2 的数据 : 60
服务器收到来自 Client1 的数据 : 30
服务器收到来自 Client2 的数据 : 15
服务器收到来自 Client1 的数据 : 7

```

```

import callbackRemote.RemoteClient;
import callbackRemote.RemoteConnection;
import callbackRemote.RemoteLogin;

public class Client implements RemoteClient {

    RemoteConnection connection;
    String name, target;

    Client(String name, String target) {

        this.name = name;
        this.target = target;
    }

    public static void main(String[] args) {

        Client client = new Client(args[0], args[1]);
        try {
            Registry reg = LocateRegistry.getRegistry();
            RemoteLogin login = (RemoteLogin) reg.lookup("LoginServerName");
            UnicastRemoteObject.exportObject(client, Integer.parseInt(args[2]));
            client.connection = login.login(client.name, client);
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }
    }

    if ("Client1".equals(client.name)) {
        try {
            client.connection.passInt(client.target, 120);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

@Override
public void half(int i) throws RemoteException {

    String result = connection.passInt(target, i / 2);
    System.out.println(name + " received: " + result + "\n");
}
}

```

Running the example:

1. Run the login server.
2. Run a client with the arguments Client2 Client1 1097.
3. Run a client with the arguments Client1 Client2 1098.

The outputs will appear in 3 consoles since there are 3 JVMs. here they are lumped together:

```

Client2 logged in
Client1 logged in
Server received from Client1:120
Server received from Client2:60
Server received from Client1:30
Server received from Client2:15
Server received from Client1:7

```

```
Client1 收到："7"  
Client2 收到："15"  
Client1 收到："30"  
Client2 收到："60"
```

第101.2节：带有客户端和服务器实现的简单RMI示例

这是一个简单的RMI示例，包含五个Java类和两个包，server和client。

服务器包

PersonListInterface.java

```
public interface PersonListInterface extends Remote  
{  
    /**  
     * 此接口由客户端和服务器端共同使用  
     * @return 人员列表  
     * @throws RemoteException  
     */  
    ArrayList<String> getPersonList() throws RemoteException;  
}
```

PersonListImplementation.java

```
public class PersonListImplementation  
extends UnicastRemoteObject  
implements PersonListInterface  
{  
  
    private static final long serialVersionUID = 1L;  
  
    // 需要提供标准构造函数  
    public PersonListImplementation() throws RemoteException  
    {}  
  
    /**  
     * "PersonListInterface"的实现  
     * @throws RemoteException  
     */  
    @Override  
    public ArrayList<String> getPersonList() throws RemoteException  
    {  
        ArrayList<String> personList = new ArrayList<String>();  
  
        personList.add("Peter Pan");  
        personList.add("皮皮·长袜");  
        // 在此添加您的姓名 :)  
  
        return personList;  
    }  
}
```

Server.java

```
public class Server {
```

```
Client1 received: "7"  
Client2 received: "15"  
Client1 received: "30"  
Client2 received: "60"
```

Section 101.2: Simple RMI example with Client and Server implementation

This is a simple RMI example with five Java classes and two packages, *server* and *client*.

Server Package

PersonListInterface.java

```
public interface PersonListInterface extends Remote  
{  
    /**  
     * This interface is used by both client and server  
     * @return List of Persons  
     * @throws RemoteException  
     */  
    ArrayList<String> getPersonList() throws RemoteException;  
}
```

PersonListImplementation.java

```
public class PersonListImplementation  
extends UnicastRemoteObject  
implements PersonListInterface  
{  
  
    private static final long serialVersionUID = 1L;  
  
    // standard constructor needs to be available  
    public PersonListImplementation() throws RemoteException  
    {}  
  
    /**  
     * Implementation of "PersonListInterface"  
     * @throws RemoteException  
     */  
    @Override  
    public ArrayList<String> getPersonList() throws RemoteException  
    {  
        ArrayList<String> personList = new ArrayList<String>();  
  
        personList.add("Peter Pan");  
        personList.add("Pippi Langstrumpf");  
        // add your name here :)  
  
        return personList;  
    }  
}
```

Server.java

```
public class Server {
```

```

    /**
 * 将服务注册到已知的公共方法中
 */
private static void createServer() {
    try {
        // 使用标准端口1099注册注册表
        LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
        System.out.println("服务器：注册表已创建。");

        // 将PersonList注册到注册表
        Naming.rebind("PersonList", new PersonListImplementation());
        System.out.println("服务器：PersonList已注册");

    } catch (final IOException e) {
        e.printStackTrace();
    }
}

public static void main(final String[] args) {
    createServer();
}
}

```

客户端包

PersonListLocal.java

```

public class PersonListLocal {
    private static PersonListLocal instance;
    private PersonListInterface personList;

    /**
     * 创建单例实例
     */
    private PersonListLocal() {
        try {
            // 查找本地运行的端口为1099的服务器
            final Registry registry = LocateRegistry.getRegistry("localhost",
                Registry.REGISTRY_PORT);

            // 查找已注册的"PersonList"
            personList = (PersonListInterface) registry.lookup("PersonList");
        } catch (final RemoteException e) {
            e.printStackTrace();
        } catch (final NotBoundException e) {
            e.printStackTrace();
        }
    }

    public static PersonListLocal getInstance() {
        if (instance == null) {
            instance = new PersonListLocal();
        }
        return instance;
    }

    /**
     * 返回服务器的 PersonList
     */
    public ArrayList<String> getPersonList() {

```

```

    /**
     * Register servicer to the known public methods
     */
private static void createServer() {
    try {
        // Register registry with standard port 1099
        LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
        System.out.println("Server : Registry created.");

        // Register PersonList to registry
        Naming.rebind("PersonList", new PersonListImplementation());
        System.out.println("Server : PersonList registered");

    } catch (final IOException e) {
        e.printStackTrace();
    }
}

public static void main(final String[] args) {
    createServer();
}
}

```

Client package

PersonListLocal.java

```

public class PersonListLocal {
    private static PersonListLocal instance;
    private PersonListInterface personList;

    /**
     * Create a singleton instance
     */
    private PersonListLocal() {
        try {
            // Lookup to the local running server with port 1099
            final Registry registry = LocateRegistry.getRegistry("localhost",
                Registry.REGISTRY_PORT);

            // Lookup to the registered "PersonList"
            personList = (PersonListInterface) registry.lookup("PersonList");
        } catch (final RemoteException e) {
            e.printStackTrace();
        } catch (final NotBoundException e) {
            e.printStackTrace();
        }
    }

    public static PersonListLocal getInstance() {
        if (instance == null) {
            instance = new PersonListLocal();
        }
        return instance;
    }

    /**
     * Returns the servers PersonList
     */
    public ArrayList<String> getPersonList() {

```

```

    if (instance != null) {
        try {
            return personList.getPersonList();
        } catch (final RemoteException e) {
            e.printStackTrace();
        }
    }

    return new ArrayList<>();
}
}

```

PersonTest.java

```

public class PersonTest
{
    public static void main(String[] args)
    {
        // 获取(本地) PersonList
        ArrayList<String> personList = PersonListLocal.getInstance().getPersonList();

        // 打印所有人员
        for(String person : personList)
        {
            System.out.println(person);
        }
    }
}

```

测试你的应用程序

- 启动 Server.java 的主方法。输出：

服务器：注册表已创建。
服务器：PersonList 已注册

- 启动 PersonTest.java 的主方法。输出：

彼得·潘
皮皮·长袜

第101.3节：客户端-服务器：在一个JVM中调用另一个JVM的方法

共享远程接口：

```

package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {
    int stringToInt(String string) throws RemoteException;
}

```

```

    if (instance != null) {
        try {
            return personList.getPersonList();
        } catch (final RemoteException e) {
            e.printStackTrace();
        }
    }

    return new ArrayList<>();
}
}

```

PersonTest.java

```

public class PersonTest
{
    public static void main(String[] args)
    {
        // get (local) PersonList
        ArrayList<String> personList = PersonListLocal.getInstance().getPersonList();

        // print all persons
        for(String person : personList)
        {
            System.out.println(person);
        }
    }
}

```

Test your application

- Start main method of Server.java. Output:

Server : Registry created.
Server : PersonList registered

- Start main method of PersonTest.java. Output:

Peter Pan
Pippi Langstrumpf

Section 101.3: Client-Server: invoking methods in one JVM from another

The shared remote interface:

```

package remote;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteServer extends Remote {
    int stringToInt(String string) throws RemoteException;
}

```

实现共享远程接口的服务器：

```
package server;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import remote.RemoteServer;

public class 服务器 implements 远程服务器 {

    @Override
    public int 字符串转整数(String 字符串) throws RemoteException {
        System.out.println("服务器接收到: " + 字符串 + " ");
        return Integer.parseInt(字符串);
    }

    public static void main(String[] args) {
        try {
            Registry 注册表 = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            服务器 server = new 服务器();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            注册表.rebind("服务器名称", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

客户端远程调用服务器上的方法：

```
package 客户端;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import remote.远程服务器;

public class 客户端 {

    static 远程服务器 服务器;

    public static void main(String[] 参数) {
        try {
            Registry 注册表 = LocateRegistry.getRegistry();
            服务器 = (远程服务器) 注册表.lookup("服务器名称");
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }
        Client client = new Client();
        client.callServer();
    }
}
```

The server implementing the shared remote interface:

```
package server;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import remote.RemoteServer;

public class Server implements RemoteServer {

    @Override
    public int stringToInt(String string) throws RemoteException {
        System.out.println("Server received: " + string + " ");
        return Integer.parseInt(string);
    }

    public static void main(String[] args) {
        try {
            Registry reg = LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
            Server server = new Server();
            UnicastRemoteObject.exportObject(server, Registry.REGISTRY_PORT);
            reg.rebind("ServerName", server);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

The client invoking a method on the server (remotely):

```
package client;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import remote.RemoteServer;

public class Client {

    static RemoteServer server;

    public static void main(String[] args) {
        try {
            Registry reg = LocateRegistry.getRegistry();
            server = (RemoteServer) reg.lookup("ServerName");
        } catch (RemoteException | NotBoundException e) {
            e.printStackTrace();
        }
        Client client = new Client();
        client.callServer();
    }
}
```

```
void callServer() {  
  
    try {  
        int i = server.stringToInt("120");  
        System.out.println("客户端接收: " + i);  
    } catch (RemoteException e) {  
        e.printStackTrace();  
    }  
}
```

输出：

```
服务器接收: "120"  
客户端接收: 120
```

```
void callServer() {  
  
    try {  
        int i = server.stringToInt("120");  
        System.out.println("Client received: " + i);  
    } catch (RemoteException e) {  
        e.printStackTrace();  
    }  
}
```

Output:

```
Server received: "120"  
Client received: 120
```

第102章：迭代器和可迭代对象

java.util.Iterator 是实现迭代器设计模式的标准Java SE接口。
java.lang.Iterable 接口用于能够提供迭代器的对象。

第102.1节：使用迭代器删除元素

Iterator.remove() 方法是一个可选方法，用于删除上一次调用Iterator.next() 返回的元素。例如，以下代码填充了一个字符串列表，然后删除所有空字符串。

```
List<String> names = new ArrayList<>();
names.add("名称 1");
names.add("名称 2");
names.add("");
names.add("名称 3");
names.add("");
System.out.println("旧大小 : " + names.size());
Iterator<String> it = names.iterator();
while (it.hasNext()) {
    String el = it.next();
    if (el.equals("")) {
        it.remove();
    }
}
System.out.println("新大小 : " + names.size());
```

输出：

```
旧大小 : 5
新大小 : 3
```

请注意，上述代码是在迭代典型集合时安全移除元素的方式。如果你尝试像下面这样从集合中移除元素：

```
for (String el: names) {
    if (el.equals(""))
        names.remove(el); // 错误 !
}
```

一个典型的集合（例如ArrayList），它为迭代器提供快速失败 (*fail fast*) 的迭代器语义，将抛出 ConcurrentModificationException 异常。

remove() 方法只能在调用 next() 之后调用（一次）。如果在调用 next() 之前调用，或者在调用 next() 之后连续调用两次 remove()，则 remove() 调用将抛出 IllegalStateException。

remove 操作被描述为一个可选操作；即并非所有迭代器都支持它。不支持的例子包括不可变集合的迭代器、集合的只读视图或固定大小的集合。如果在迭代器不支持删除时调用 remove()，将抛出 UnsupportedOperationException。

第 102.2 节：创建你自己的 Iterable

要创建你自己的 Iterable，和实现任何接口一样，只需实现接口中的抽象方法即可。对于

Chapter 102: Iterator and Iterable

The java.util.Iterator is the standard Java SE interface for object that implement the Iterator design pattern.
The java.lang.Iterable interface is for objects that can provide an iterator.

Section 102.1: Removing elements using an iterator

The Iterator.remove() method is an optional method that removes the element returned by the previous call to Iterator.next(). For example, the following code populates a list of strings and then removes all of the empty strings.

```
List<String> names = new ArrayList<>();
names.add("name 1");
names.add("name 2");
names.add("");
names.add("name 3");
names.add("");
System.out.println("Old Size : " + names.size());
Iterator<String> it = names.iterator();
while (it.hasNext()) {
    String el = it.next();
    if (el.equals(""))
        it.remove();
}
System.out.println("New Size : " + names.size());
```

Output :

```
Old Size : 5
New Size : 3
```

Note that is the code above is the safe way to remove elements while iterating a typical collection. If instead, you attempt to do remove elements from a collection like this:

```
for (String el: names) {
    if (el.equals(""))
        names.remove(el); // WRONG!
}
```

a typical collection (such as ArrayList) which provides iterators with fail fast iterator semantics will throw a ConcurrentModificationException.

The remove() method can only called (once) following a next() call. If it is called before calling next() or if it is called twice following a next() call, then the remove() call will throw an IllegalStateException.

The remove operation is described as an optional operation; i.e. not all iterators will allow it. Examples where it is not supported include iterators for immutable collections, read-only views of collections, or fixed sized collections. If remove() is called when the iterator does not support removal, it will throw an UnsupportedOperationException.

Section 102.2: Creating your own Iterable

To create your own Iterable as with any interface you just implement the abstract methods in the interface. For

Iterable，只有一个抽象方法，称为 iterator()。但它的返回类型 Iterator 本身是一个接口，包含三个抽象方法。你可以返回与某个集合关联的迭代器，或者创建你自己的自定义实现：

```
public static class Alphabet 实现 Iterable<Character> {  
  
    @Override  
    public Iterator<Character> iterator() {  
        return new Iterator<Character>() {  
            char letter = 'a';  
  
            @Override  
            public boolean hasNext() {  
                return letter <= 'z';  
            }  
  
            @Override  
            public Character next() {  
                return letter++;  
            }  
  
            @Override  
            public void remove() {  
                throw new UnsupportedOperationException("删除字母没有意义");  
            }  
        };  
    }  
}
```

使用方法：

```
public static void main(String[] args) {  
    for(char c : new Alphabet()) {  
        System.out.println("c = " + c);  
    }  
}
```

新的Iterator应带有指向第一个元素的状态，每次调用 next 都会更新状态指向下一个元素。 hasNext()方法用于检查迭代器是否已到达末尾。如果迭代器连接到一个可修改的集合，那么迭代器的可选方法remove()可能会被实现，用于从底层集合中移除当前指向的元素。

第102.3节：在for循环中使用Iterable

实现了Iterable<>接口的类可以在for循环中使用。这实际上只是获取对象的迭代器并顺序获取所有元素的语法糖；它使代码更清晰、编写更快捷且不易出错。

```
public class UsingIterable {  
  
    public static void main(String[] args) {  
        List<Integer> intList = Arrays.asList(1,2,3,4,5,6,7);  
  
        // List 继承自 Collection, Collection 继承自 Iterable  
        Iterable<Integer> iterable = intList;  
  
        // 类似 foreach 的循环  
        for (Integer i: iterable) {  
    }
```

Iterable there is only one which is called iterator(). But its return type **Iterator** is itself an interface with three abstract methods. You can return an iterator associated with some collection or create your own custom implementation:

```
public static class Alphabet implements Iterable<Character> {  
  
    @Override  
    public Iterator<Character> iterator() {  
        return new Iterator<Character>() {  
            char letter = 'a';  
  
            @Override  
            public boolean hasNext() {  
                return letter <= 'z';  
            }  
  
            @Override  
            public Character next() {  
                return letter++;  
            }  
  
            @Override  
            public void remove() {  
                throw new UnsupportedOperationException("Doesn't make sense to remove a letter");  
            }  
        };  
    }  
}
```

To use:

```
public static void main(String[] args) {  
    for(char c : new Alphabet()) {  
        System.out.println("c = " + c);  
    }  
}
```

The new **Iterator** should come with a state pointing to the first item, each call to next updates its state to point to the next one. The hasNext() checks to see if the iterator is at the end. If the iterator were connected to a modifiable collection then the iterator's optional remove() method might be implemented to remove the item currently pointed to from the underlying collection.

Section 102.3: Using Iterable in for loop

Classes implementing Iterable<> interface can be used in **for** loops. This is actually only [syntactic sugar](#) for getting an iterator from the object and using it to get all elements sequentially; it makes code clearer, faster to write and less error-prone.

```
public class UsingIterable {  
  
    public static void main(String[] args) {  
        List<Integer> intList = Arrays.asList(1,2,3,4,5,6,7);  
  
        // List extends Collection, Collection extends Iterable  
        Iterable<Integer> iterable = intList;  
  
        // foreach-like loop  
        for (Integer i: iterable) {  
    }
```

```

        System.out.println(i);
    }

    // Java 5 之前的循环遍历方式
    for(Iterator<Integer> i = iterable.iterator(); i.hasNext(); ) {
        Integer item = i.next();
        System.out.println(item);
    }
}

```

第102.4节：使用原始迭代器

虽然使用foreach循环（或“扩展for循环”）很简单，但有时直接使用迭代器更有优势。例如，如果你想输出一串以逗号分隔的值，但不希望最后一个元素后面有逗号：

```

List<String> yourData = //...
Iterator<String> iterator = yourData.iterator();
while (iterator.hasNext()){
    // next() 会“移动”迭代器到下一个条目并返回其值。
    String entry = iterator.next();
    System.out.print(entry);
    if (iterator.hasNext()){
        // 如果迭代器在当前元素之后还有另一个元素：
        System.out.print(",");
    }
}

```

这比使用isLastEntry变量或用循环索引进行计算要简单明了得多。

```

        System.out.println(i);
    }

    // pre java 5 way of iterating loops
    for(Iterator<Integer> i = iterable.iterator(); i.hasNext(); ) {
        Integer item = i.next();
        System.out.println(item);
    }
}

```

Section 102.4: Using the raw iterator

While using the foreach loop (or "extended for loop") is simple, it's sometimes beneficial to use the iterator directly. For example, if you want to output a bunch of comma-separated values, but don't want the last item to have a comma:

```

List<String> yourData = //...
Iterator<String> iterator = yourData.iterator();
while (iterator.hasNext()){
    // next() "moves" the iterator to the next entry and returns its value.
    String entry = iterator.next();
    System.out.print(entry);
    if (iterator.hasNext()){
        // If the iterator has another element after the current one:
        System.out.print(",");
    }
}

```

This is much easier and clearer than having a isLastEntry variable or doing calculations with the loop index.

第103章：反射API

反射通常被需要检查或修改JVM中运行的应用程序运行时行为的程序使用。Java反射API用于此目的，它使得在运行时检查类、接口、字段和方法成为可能，而无需在编译时知道它们的名称。它还使得通过反射实例化新对象和调用方法成为可能。

第103.1节：动态代理

动态代理实际上与反射关系不大，但它们是该API的一部分。它基本上是一种创建接口动态实现的方式。这在创建模拟服务时可能很有用。

动态代理是接口的一个实例，它是通过所谓的调用处理器创建的，该处理器拦截所有方法调用并允许手动处理它们的调用。

```
public class DynamicProxyTest {  
  
    public interface MyInterface1{  
        public void someMethod1();  
        public int someMethod2(String s);  
    }  
  
    public interface MyInterface2{  
        public void anotherMethod();  
    }  
  
    public static void main(String args[]) throws Exception {  
        // 动态代理类  
        Class<?> proxyClass = Proxy.getProxyClass(  
            ClassLoader.getSystemClassLoader(),  
            new Class[] {MyInterface1.class, MyInterface2.class});  
        // 动态代理类构造函数  
        Constructor<?> proxyConstructor =  
            proxyClass.getConstructor(InvocationHandler.class);  
  
        // 调用处理器  
        InvocationHandler handler = new InvocationHandler(){  
            // 每次代理方法调用时都会调用此方法  
            // method 是被调用的方法，args 保存方法参数  
            // 必须返回方法结果  
            @Override  
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
                String methodName = method.getName();  
  
                if(methodName.equals("someMethod1")){  
                    System.out.println("someMethod1 被调用了！");  
                    return null;  
                }  
                if(methodName.equals("someMethod2")){  
                    System.out.println("someMethod2 被调用了！");  
                    System.out.println("参数: " + args[0]);  
                    return 42;  
                }  
                if(methodName.equals("anotherMethod")){  
                    System.out.println("anotherMethod 被调用了！");  
                    return null;  
                }  
                System.out.println("未知方法！");  
                return null;  
            }  
        };  
    }  
}
```

Chapter 103: Reflection API

Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the JVM. [Java Reflection API](#) is used for that purpose where it makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing their names at compile time. And It also makes it possible to instantiate new objects, and to invoke methods using reflection.

Section 103.1: Dynamic Proxies

Dynamic Proxies do not really have much to do with Reflection but they are part of the API. It's basically a way to create a dynamic implementation of an interface. This could be helpful when creating mockup services.

A Dynamic Proxy is an instance of an interface that is created with a so-called invocation handler that intercepts all method calls and allows the handling of their invocation manually.

```
public class DynamicProxyTest {  
  
    public interface MyInterface1{  
        public void someMethod1();  
        public int someMethod2(String s);  
    }  
  
    public interface MyInterface2{  
        public void anotherMethod();  
    }  
  
    public static void main(String args[]) throws Exception {  
        // the dynamic proxy class  
        Class<?> proxyClass = Proxy.getProxyClass(  
            ClassLoader.getSystemClassLoader(),  
            new Class[] {MyInterface1.class, MyInterface2.class});  
        // the dynamic proxy class constructor  
        Constructor<?> proxyConstructor =  
            proxyClass.getConstructor(InvocationHandler.class);  
  
        // the invocation handler  
        InvocationHandler handler = new InvocationHandler(){  
            // this method is invoked for every proxy method call  
            // method is the invoked method, args holds the method parameters  
            // it must return the method result  
            @Override  
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
                String methodName = method.getName();  
  
                if(methodName.equals("someMethod1")){  
                    System.out.println("someMethod1 was invoked!");  
                    return null;  
                }  
                if(methodName.equals("someMethod2")){  
                    System.out.println("someMethod2 was invoked!");  
                    System.out.println("Parameter: " + args[0]);  
                    return 42;  
                }  
                if(methodName.equals("anotherMethod")){  
                    System.out.println("anotherMethod was invoked!");  
                    return null;  
                }  
                System.out.println("Unknown method!");  
                return null;  
            }  
        };  
    }  
}
```

```

    }

    // 创建动态代理实例
MyInterface1 i1 = (MyInterface1) proxyConstructor.newInstance(handler);
MyInterface2 i2 = (MyInterface2) proxyConstructor.newInstance(handler);

    // 并调用一些方法
i1.someMethod1();
    i1.someMethod2("stackoverflow");
    i2.anotherMethod();
}

}

```

这段代码的结果是：

```

someMethod1 被调用了！
someMethod2 被调用了！
参数: stackoverflow
anotherMethod 被调用了！

```

第103.2节：介绍

基础知识

反射API允许在运行时检查代码的类结构并动态调用代码。这非常强大，但也很危险，因为编译器无法静态确定动态调用是否有效。

一个简单的例子是获取给定类的公共构造函数和方法：

```

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

// 这是一个表示String类的对象（不是String的实例！）
Class<String> clazz = String.class;

Constructor<?>[] constructors = clazz.getConstructors(); // 返回String的所有公共构造函数
Method[] methods = clazz.getMethods(); // 返回String及其父类的所有公共方法

```

通过这些信息，可以实例化对象并动态调用不同的方法。

反射与泛型类型

泛型类型信息可用于：

- 方法参数，使用getGenericParameterTypes()。
- 方法返回类型，使用getGenericReturnType()。
- 公共字段，使用getGenericType。

下面的示例展示了如何在这三种情况下提取泛型类型信息：

```

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;

```

```

    }

    // create the dynamic proxy instances
MyInterface1 i1 = (MyInterface1) proxyConstructor.newInstance(handler);
MyInterface2 i2 = (MyInterface2) proxyConstructor.newInstance(handler);

    // and invoke some methods
i1.someMethod1();
i1.someMethod2("stackoverflow");
i2.anotherMethod();
}

}

```

The result of this code is this:

```

someMethod1 was invoked!
someMethod2 was invoked!
Parameter: stackoverflow
anotherMethod was invoked!

```

Section 103.2: Introduction

Basics

The Reflection API allows one to check the class structure of the code at runtime and invoke code dynamically. This is very powerful, but it is also dangerous since the compiler is not able to statically determine whether dynamic invocations are valid.

A simple example would be to get the public constructors and methods of a given class:

```

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

// This is a object representing the String class (not an instance of String!)
Class<String> clazz = String.class;

Constructor<?>[] constructors = clazz.getConstructors(); // returns all public constructors of
String
Method[] methods = clazz.getMethods(); // returns all public methods from String and parents

```

With this information it is possible to instance the object and call different methods dynamically.

Reflection and Generic Types

Generic type information is available for:

- method parameters, using getGenericParameterTypes().
- method return types, using getGenericReturnType().
- **public** fields, using getGenericType.

The following example shows how to extract the generic type information in all three cases:

```

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.List;

```

```

import java.util.Map;

public class GenericTest {

    public static void main(final String[] args) throws Exception {
        final Method method = GenericTest.class.getMethod("testMethod", Map.class);
        final Field field = GenericTest.class.getField("testField");

        System.out.println("方法参数:");
        final Type parameterType = method.getGenericParameterTypes()[0];
        displayGenericType(parameterType, "");

        System.out.println("方法返回类型:");
        final Type returnType = method.getGenericReturnType();
        displayGenericType(returnType, "");

        System.out.println("字段类型:");
        final Type fieldType = field.getGenericType();
        displayGenericType(fieldType, "");

    }

    private static void displayGenericType(final Type type, final String prefix) {
        System.out.println(prefix + type.getTypeName());
        if (type instanceof ParameterizedType) {
            for (final Type subtype : ((ParameterizedType) type).getActualTypeArguments()) {
                displayGenericType(subtype, prefix + "");
            }
        }
    }

    public Map<String, Map<Integer, List<String>>> testField;

    public List<Number> testMethod(final Map<String, Double> arg) {
        return null;
    }
}

```

这将产生以下输出：

```

方法参数:
java.util.Map
java.lang.String
java.lang.Double
方法返回类型:
java.util.List
java.lang.Number
字段类型 :
java.util.Map
java.lang.String
java.util.Map
java.lang.Integer
java.util.List
java.lang.String

```

第103.3节：使用反射的恶意Java技巧

反射API可以用来更改JDK默认库中私有和最终字段的值。这

```

import java.util.Map;

public class GenericTest {

    public static void main(final String[] args) throws Exception {
        final Method method = GenericTest.class.getMethod("testMethod", Map.class);
        final Field field = GenericTest.class.getField("testField");

        System.out.println("Method parameter:");
        final Type parameterType = method.getGenericParameterTypes()[0];
        displayGenericType(parameterType, "\t");

        System.out.println("Method return type:");
        final Type returnType = method.getGenericReturnType();
        displayGenericType(returnType, "\t");

        System.out.println("Field type:");
        final Type fieldType = field.getGenericType();
        displayGenericType(fieldType, "\t");

    }

    private static void displayGenericType(final Type type, final String prefix) {
        System.out.println(prefix + type.getTypeName());
        if (type instanceof ParameterizedType) {
            for (final Type subtype : ((ParameterizedType) type).getActualTypeArguments()) {
                displayGenericType(subtype, prefix + "\t");
            }
        }
    }

    public Map<String, Map<Integer, List<String>>> testField;

    public List<Number> testMethod(final Map<String, Double> arg) {
        return null;
    }
}

```

This results in the following output:

```

Method parameter:
java.util.Map
java.lang.String
java.lang.Double
Method return type:
java.util.List
java.lang.Number
Field type:
java.util.Map
java.lang.String
java.util.Map
java.lang.Integer
java.util.List
java.lang.String

```

Section 103.3: Evil Java hacks with Reflection

The Reflection API could be used to change values of private and final fields even in the JDK default library. This

可以用来操控一些知名类的行为，正如我们将看到的那样。

什么是不可能的

我们先从唯一的限制开始，也就是唯一一个我们不能通过反射更改的字段。那就是Java SecurityManager。它在java.lang.System中声明为

```
private static volatile SecurityManager security = null;
```

但是如果我们运行这段代码，它不会在System类中被列出

```
for(Field f : System.class.getDeclaredFields())
    System.out.println(f);
```

这是因为 fieldFilterMap 在 sun.reflect.Reflection 中持有映射本身和 System.class 中的安全字段，并保护它们不被反射访问。所以我们无法禁用 SecurityManager。

疯狂字符串

每个 Java 字符串在 JVM 中表示为 String 类的一个实例。然而，在某些情况下，JVM 通过对相同的字符串使用同一个实例来节省堆空间。这发生在字符串字面量，以及通过调用 String.intern() 进行“驻留”的字符串上。因此，如果你的代码中多次出现 "hello"，它们实际上都是同一个对象实例。

字符串应该是不可变的，但可以使用“恶意”反射来修改它们。下面的示例展示了如何通过替换字符串的 value 字段来改变字符串中的字符。

```
public class CrazyStrings {
    static {
        try {
            Field f = String.class.getDeclaredField("value");
            f.setAccessible(true);
            f.set("hello", "you stink!".toCharArray());
        } catch (Exception e) {
        }
    }
    public static void main(String args[]) {
        System.out.println("hello");
    }
}
```

所以这段代码将打印“你很臭！”

1 = 42

同样的思路也可以用于 Integer 类

```
public class CrazyMath {
    static {
        try {
            Field value = Integer.class.getDeclaredField("value");
            value.setAccessible(true);
            value.setInt(Integer.valueOf(1), 42);
        } catch (Exception e) {
        }
    }
}
```

could be used to manipulate the behaviour of some well known classes as we will see.

What is not possible

Lets start first with the only limitation means the only field we can't change with Reflection. That is the Java SecurityManager. It is declared in [java.lang.System](#) as

```
private static volatile SecurityManager security = null;
```

But it won't be listed in the System class if we run this code

```
for(Field f : System.class.getDeclaredFields())
    System.out.println(f);
```

Thats because of the fieldFilterMap in [sun.reflect.Reflection](#) that holds the map itself and the security field in the System.class and protects them against any access with Reflection. So we could not deactivate the SecurityManager.

Crazy Strings

Each Java String is represented by the JVM as an instance of the String class. However, in some situations the JVM saves heap space by using the same instance for Strings that are. This happens for string literals, and also for strings that have been "interned" by calling String.intern(). So if you have "hello" in your code multiple times it is always the same object instance.

Strings are supposed to be immutable, but it is possible to use "evil" reflection to change them. The example below show how we can change the characters in a String by replacing its value field.

```
public class CrazyStrings {
    static {
        try {
            Field f = String.class.getDeclaredField("value");
            f.setAccessible(true);
            f.set("hello", "you stink!".toCharArray());
        } catch (Exception e) {
        }
    }
    public static void main(String args[]) {
        System.out.println("hello");
    }
}
```

So this code will print "you stink!"

1 = 42

The same idea could be used with the Integer Class

```
public class CrazyMath {
    static {
        try {
            Field value = Integer.class.getDeclaredField("value");
            value.setAccessible(true);
            value.setInt(Integer.valueOf(1), 42);
        } catch (Exception e) {
        }
    }
}
```

```

public static void main(String args[]) {
    System.out.println(Integer.valueOf(1));
}

```

一切都是真的

根据这个[stackoverflow帖子](#)，我们可以使用反射做一些非常恶意的事情。

```

public class Evil {
    static {
        try {
            Field field = Boolean.class.getField("FALSE");
            field.setAccessible(true);
            Field modifiersField = Field.class.getDeclaredField("modifiers");
            modifiersField.setAccessible(true);
            modifiersField.setInt(field, field.getModifiers() & ~Modifier.FINAL);
            field.set(null, true);
        } catch (Exception e) {
        }
    }
    public static void main(String args[]){
        System.out.format("Everything is %s", false);
    }
}

```

请注意，我们在这里所做的操作将导致JVM表现出难以解释的行为。这是非常危险的。

第103.4节：滥用反射API修改私有和final变量

反射在正确使用且目的正当时非常有用。通过使用反射，你可以访问私有变量并重新初始化final变量。

下面是代码片段，不推荐使用。

```

import java.lang.reflect.*;

public class ReflectionDemo{
    public static void main(String args[]){
        try{
            Field[] fields = A.class.getDeclaredFields();
            A a = new A();
            for ( Field field:fields ) {
                if(field.getName().equalsIgnoreCase("name")){
                    field.setAccessible(true);
                    field.set(a, "StackOverFlow");
                    System.out.println("A.name="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("age")){
                    field.set(a, 20);
                    System.out.println("A.age="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("rep")){
                    field.setAccessible(true);
                    field.set(a, "New Reputation");
                    System.out.println("A.rep="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("count")){

```

```

public static void main(String args[]) {
    System.out.println(Integer.valueOf(1));
}

```

Everything is true

And according to [this stackoverflow post](#) we can use reflection to do something really evil.

```

public class Evil {
    static {
        try {
            Field field = Boolean.class.getField("FALSE");
            field.setAccessible(true);
            Field modifiersField = Field.class.getDeclaredField("modifiers");
            modifiersField.setAccessible(true);
            modifiersField.setInt(field, field.getModifiers() & ~Modifier.FINAL);
            field.set(null, true);
        } catch (Exception e) {
        }
    }
    public static void main(String args[]){
        System.out.format("Everything is %s", false);
    }
}

```

Note that what we are doing here is going to cause the JVM to behave in inexplicable ways. This is very dangerous.

Section 103.4: Misuse of Reflection API to change private and final variables

Reflection is useful when it is properly used for right purpose. By using reflection, you can access private variables and re-initialize final variables.

Below is the code snippet, which is **not** recommended.

```

import java.lang.reflect.*;

public class ReflectionDemo{
    public static void main(String args[]){
        try{
            Field[] fields = A.class.getDeclaredFields();
            A a = new A();
            for ( Field field:fields ) {
                if(field.getName().equalsIgnoreCase("name")){
                    field.setAccessible(true);
                    field.set(a, "StackOverFlow");
                    System.out.println("A.name="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("age")){
                    field.set(a, 20);
                    System.out.println("A.age="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("rep")){
                    field.setAccessible(true);
                    field.set(a, "New Reputation");
                    System.out.println("A.rep="+field.get(a));
                }
                if(field.getName().equalsIgnoreCase("count")){

```

```

field.set(a, 25);
    System.out.println("A.count="+field.get(a));
}
} catch(Exception err){
err.printStackTrace();
}
}

class A {
private String name;
public int age;
public final String rep;
public static int count=0;

public A(){
name = "Unset";
age = 0;
rep = "Reputation";
count++;
}
}

```

输出：

```

A.name=StackOverFlow
A.age=20
A.rep>New Reputation
A.count=25

```

说明：

在正常情况下，`private` 变量不能在声明类之外访问（没有 `getter` 和 `setter` 方法）。`final` 变量初始化后不能重新赋值。

反射打破了这两个限制，如上所述，可以被滥用来更改 `private` 和 `final` 变量。

`field.setAccessible(true)` 是实现所需功能的关键。

第 103.5 节：获取和设置字段

使用反射API，可以在运行时更改或获取字段的值。例如，您可以在API中根据某个因素（如操作系统）检索不同的字段。您还可以移除诸如`final`之类的修饰符，以允许修改被声明为`final`的字段。

为此，您需要以如下方式使用方法`Class#getField()`：

```

// 获取SomeClass类中名为"NAME"的字段。
Field nameField = SomeClass.class.getDeclaredField("NAME");

// 获取Field类中名为"modifiers"的字段。注意它不需要是静态的
// Field modifiersField = Field.class.getDeclaredField("modifiers");

// 允许任何人访问，即使它被声明为私有的
modifiersField.setAccessible(true);

```

```

field.set(a, 25);
    System.out.println("A.count="+field.get(a));
}
} catch(Exception err){
err.printStackTrace();
}
}

class A {
private String name;
public int age;
public final String rep;
public static int count=0;

public A(){
name = "Unset";
age = 0;
rep = "Reputation";
count++;
}
}

```

Output:

```

A.name=Stack0verFlow
A.age=20
A.rep>New Reputation
A.count=25

```

Explanation:

In normal scenario, `private` variables can't be accessed outside of declared class (without getter and setter methods). `final` variables can't be re-assigned after initialization.

Reflection breaks both barriers can be abused to change both private and final variables as explained above.

`field.setAccessible(true)` is the key to achieve desired functionality.

Section 103.5: Getting and Setting fields

Using the Reflection API, it is possible to change or get the value of a field at runtime. For example, you could use it in an API to retrieve different fields based on a factor, like the OS. You can also remove modifiers like `final` to allow modifying fields that are `final`.

To do so, you will need to use the method `Class#getField()` in a way such as the one shown below:

```

// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Get the field in class Field "modifiers". Note that it does not
// need to be static
Field modifiersField = Field.class.getDeclaredField("modifiers");

// Allow access from anyone even if it's declared private
modifiersField.setAccessible(true);

```

```

// 以int类型获取"NAME"字段上的修饰符。
int existingModifiersOnNameField = nameField.getModifiers();

// 对现有修饰符执行按位与非操作 Modifier.FINAL (16)
// 详细说明见 https://en.wikipedia.org/wiki/Bitwise\_operations\_in\_C
// 如果你不确定什么是按位操作。
int newModifiersOnNameField = existingModifiersOnNameField & ~Modifier.FINAL;

// 为非静态字段设置对象下修饰符字段的值
modifiersField.setInt(nameField, newModifiersOnNameField);

// 设置为可访问。这会覆盖正常的Java
// private/protected/package等访问控制检查。
nameField.setAccessible(true);

// 在这里设置"NAME"的值。注意传入null参数。
// 修改静态字段时传入null，因为没有实例对象
nameField.set(null, "通过反射破解..." );

// 这里我可以直接访问它。如有需要，使用反射获取它。（见下文）
System.out.println(SomeClass.NAME);

```

获取字段要容易得多。我们可以使用Field#get()及其变体来获取其值：

```

// 获取SomeClass类中名为"NAME"的字段。
Field nameField = SomeClass.class.getDeclaredField("NAME");

// 设置私有字段的可访问性
nameField.setAccessible(true);

// 传入null，因为没有实例，记住吗？
String name = (String) nameField.get(null);

```

请注意这一点：

使用Class#getDeclaredField时，用它来获取类自身的字段：

```

class HackMe extends Hacked {
    public String iAmDeclared;
}

class Hacked {
    public String someState;
}

```

这里，HackMe#iAmDeclared 是声明的字段。然而，HackMe#someState 不是声明的字段，因为它是从其超类 Hacked 继承的。

第103.6节：调用构造函数

获取构造函数对象

你可以通过如下方式从Class对象获取Constructor类：

```

Class myClass = ... // 获取一个类对象
Constructor[] constructors = myClass.getConstructors();

```

其中constructors变量将包含每个在该类中声明的公共构造函数对应的一个Constructor实例

```

// Get the modifiers on the "NAME" field as an int.
int existingModifiersOnNameField = nameField.getModifiers();

// Bitwise AND NOT Modifier.FINAL (16) on the existing modifiers
// Readup here https://en.wikipedia.org/wiki/Bitwise\_operations\_in\_C
// if you're unsure what bitwise operations are.
int newModifiersOnNameField = existingModifiersOnNameField & ~Modifier.FINAL;

// Set the value of the modifiers field under an object for non-static fields
modifiersField.setInt(nameField, newModifiersOnNameField);

// Set it to be accessible. This overrides normal Java
// private/protected/package/etc access control checks.
nameField.setAccessible(true);

// Set the value of "NAME" here. Note the null argument.
// Pass null when modifying static fields, as there is no instance object
nameField.set(null, "Hacked by reflection..." );

// Here I can directly access it. If needed, use reflection to get it. (Below)
System.out.println(SomeClass.NAME);

```

Getting fields is much easier. We can use [Field#get\(\)](#) and its variants to get its value:

```

// Get the field in class SomeClass "NAME".
Field nameField = SomeClass.class.getDeclaredField("NAME");

// Set accessible for private fields
nameField.setAccessible(true);

// Pass null as there is no instance, remember?
String name = (String) nameField.get(null);

```

Do note this:

When using [Class#getDeclaredField](#), use it to get a field in the class itself:

```

class HackMe extends Hacked {
    public String iAmDeclared;
}

class Hacked {
    public String someState;
}

```

Here, HackMe#iAmDeclared is declared field. However, HackMe#someState is not a declared field as it is inherited from its superclass, Hacked.

Section 103.6: Call constructor

Getting the Constructor Object

You can obtain [Constructor](#) class from the [Class](#) object like this:

```

Class myClass = ... // get a class object
Constructor[] constructors = myClass.getConstructors();

```

Where the constructors variable will have one [Constructor](#) instance for each public constructor declared in the

类。

如果你知道想要访问的构造函数的精确参数类型，可以筛选特定的构造函数。下面的例子返回给定类中接受一个Integer作为参数的公共构造函数：

```
Class myClass = ... // 获取一个类对象  
Constructor constructor = myClass.getConstructor(new Class[]{Integer.class});
```

如果没有构造函数匹配给定的构造函数参数，将抛出NoSuchMethodException异常。

使用构造函数对象创建新实例

```
Class myClass = MyObj.class // 获取一个类对象  
Constructor constructor = myClass.getConstructor(Integer.class);  
MyObj myObj = (MyObj) constructor.newInstance(Integer.valueOf(123));
```

第103.7节：调用嵌套类的构造函数

如果你想创建一个内部嵌套类的实例，需要为Class#getDeclaredConstructor提供一个外部类的类对象作为额外参数。

```
public class Enclosing{  
    public class Nested{  
        public Nested(String a){  
            System.out.println("构造函数 :String => "+a);  
        }  
        public static void main(String args[]) throws Exception {  
            Class<?> clazzEnclosing = Class.forName("Enclosing");  
            Class<?> clazzNested = Class.forName("Enclosing$Nested");  
            Enclosing objEnclosing = (Enclosing)clazzEnclosing.newInstance();  
            Constructor<?> constructor = clazzNested.getDeclaredConstructor(new  
Class[]{}{Enclosing.class, String.class});  
            Nested objInner = (Nested)constructor.newInstance(new Object[]{}{objEnclosing,  
"StackOverFlow"});  
        }  
    }  
}
```

如果嵌套类是静态的，则不需要这个外部实例。

第103.8节：调用方法

使用反射，可以在运行时调用对象的方法。

```
import java.lang.reflect.InvocationTargetException;  
import java.lang.reflect.Method;  
  
String s = "Hello World!";  
  
// 无参数方法  
// 调用 s.length()  
Method method1 = String.class.getMethod("length");  
int length = (int) method1.invoke(s); // 变量 length 包含"12"  
  
// 有参数方法
```

class.

If you know the precise parameter types of the constructor you want to access, you can filter the specific constructor. The next example returns the public constructor of the given class which takes a Integer as parameter:

```
Class myClass = ... // get a class object  
Constructor constructor = myClass.getConstructor(new Class[]{Integer.class});
```

If no constructor matches the given constructor arguments a NoSuchMethodException is thrown.

New Instance using Constructor Object

```
Class myClass = MyObj.class // get a class object  
Constructor constructor = myClass.getConstructor(Integer.class);  
MyObj myObj = (MyObj) constructor.newInstance(Integer.valueOf(123));
```

Section 103.7: Call constructor of nested class

If you want to create an instance of an inner nested class you need to provide a class object of the enclosing class as an extra parameter with [Class#getDeclaredConstructor](#).

```
public class Enclosing{  
    public class Nested{  
        public Nested(String a){  
            System.out.println("Constructor :String => "+a);  
        }  
        public static void main(String args[]) throws Exception {  
            Class<?> clazzEnclosing = Class.forName("Enclosing");  
            Class<?> clazzNested = Class.forName("Enclosing$Nested");  
            Enclosing objEnclosing = (Enclosing)clazzEnclosing.newInstance();  
            Constructor<?> constructor = clazzNested.getDeclaredConstructor(new  
Class[]{}{Enclosing.class, String.class});  
            Nested objInner = (Nested)constructor.newInstance(new Object[]{}{objEnclosing,  
"StackOverFlow"});  
        }  
    }  
}
```

If the nested class is static you will not need this enclosing instance.

Section 103.8: Invoking a method

Using reflection, a method of an object can be invoked during runtime.

The example shows how to invoke the methods of a String object.

```
import java.lang.reflect.InvocationTargetException;  
import java.lang.reflect.Method;  
  
String s = "Hello World!";  
  
// method without parameters  
// invoke s.length()  
Method method1 = String.class.getMethod("length");  
int length = (int) method1.invoke(s); // variable length contains "12"  
  
// method with parameters
```

```
// 调用 s.substring(6)
Method method2 = String.class.getMethod("substring", int.class);
String substring = (String) method2.invoke(s, 6); // 变量 substring 包含 "World!"
```

第103.9节：根据（完全限定的）名称获取类

给定一个包含类名的String，可以使用Class.forName访问其Class对象：

```
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer");
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

版本 ≥ Java SE 1.2

可以指定是否初始化该类（forName的第二个参数）以及使用哪个ClassLoader（第三个参数）：

```
ClassLoader classLoader = ...
boolean initialize = ...
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer", initialize, classLoader);
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

第103.10节：获取枚举的常量

以此枚举为例：

```
enum 指南针 {
    北(0),
    东(90),
    南(180),
    西(270);
    private int 度数;
    指南针(int deg){
        度数 = deg;
    }
    public int 获取度数(){
        return 度数;
    }
}
```

在Java中，枚举类就像其他类一样，但为枚举值定义了一些常量。此外，它还有一个字段是一个数组，保存所有的值，并且有两个静态方法，分别是values()和valueOf(String)。

如果我们使用反射打印该类中的所有字段，就可以看到这一点

```
for(Field f : 指南针.class.getDeclaredFields())
    System.out.println(f.getName());
```

输出将是：

北

```
// invoke s.substring(6)
Method method2 = String.class.getMethod("substring", int.class);
String substring = (String) method2.invoke(s, 6); // variable substring contains "World!"
```

Section 103.9: Get Class given its (fully qualified) name

Given a `String` containing the name of a class, its `Class` object can be accessed using `Class.forName`:

```
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer");
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Version ≥ Java SE 1.2

It can be specified, if the class should be initialized (second parameter of `forName`) and which `ClassLoader` should be used (third parameter):

```
ClassLoader classLoader = ...
boolean initialize = ...
Class clazz = null;
try {
    clazz = Class.forName("java.lang.Integer", initialize, classLoader);
} catch (ClassNotFoundException ex) {
    throw new IllegalStateException(ex);
}
```

Section 103.10: Getting the Constants of an Enumeration

Giving this enumeration as Example:

```
enum Compass {
    NORTH(0),
    EAST(90),
    SOUTH(180),
    WEST(270);
    private int degree;
    Compass(int deg){
        degree = deg;
    }
    public int getDegree(){
        return degree;
    }
}
```

In Java an enum class is like any other class but has some defined constants for the enum values. Additionally it has a field that is an array that holds all the values and two static methods with name `values()` and `valueOf(String)`. We can see this if we use Reflection to print all fields in this class

```
for(Field f : Compass.class.getDeclaredFields())
    System.out.println(f.getName());
```

the output will be:

NORTH

东
南
西
度
ENUM\$VALUES

因此我们可以像检查其他类一样使用反射检查枚举类。但反射API提供了三个特定于枚举的方法。

枚举检查

```
Compass.class.isEnum();
```

对于表示枚举类型的类返回true。

获取值

```
Object[] values = Compass.class.getEnumConstants();
```

返回所有枚举值的数组，类似于 `Compass.values()`，但不需要实例。

枚举常量检查

```
for(Field f : Compass.class.getDeclaredFields()){
    if(f.isEnumConstant())
        System.out.println(f.getName());
}
```

列出所有作为枚举值的类字段。

第103.11节：使用反射调用重载构造函数

示例：通过传递相关参数调用不同的构造函数

```
import java.lang.reflect.*;

class NewInstanceWithReflection{
    public NewInstanceWithReflection(){
        System.out.println("默认构造函数");
    }
    public NewInstanceWithReflection( String a){
        System.out.println("构造函数 :String => "+a);
    }
    public static void main(String args[]) throws Exception {
        NewInstanceWithReflection object =
        (NewInstanceWithReflection)Class.forName("NewInstanceWithReflection").newInstance();
        Constructor constructor = NewInstanceWithReflection.class.getDeclaredConstructor( new
        Class[] {String.class});
        NewInstanceWithReflection object1 = (NewInstanceWithReflection)constructor.newInstance(new
        Object[]{"StackOverFlow"});
    }
}
```

输出：

EAST
SOUTH
WEST
degree
ENUM\$VALUES

So we could examine enum classes with Reflection like any other class. But the Reflection API offers three enum-specific methods.

enum check

```
Compass.class.isEnum();
```

Returns true for classes that represents an enum type.

retrieving values

```
Object[] values = Compass.class.getEnumConstants();
```

Returns an array of all enum values like `Compass.values()` but without the need of an instance.

enum constant check

```
for(Field f : Compass.class.getDeclaredFields()){
    if(f.isEnumConstant())
        System.out.println(f.getName());
}
```

Lists all the class fields that are enum values.

Section 103.11: Call overloaded constructors using reflection

Example: Invoke different constructors by passing relevant parameters

```
import java.lang.reflect.*;

class NewInstanceWithReflection{
    public NewInstanceWithReflection(){
        System.out.println("Default constructor");
    }
    public NewInstanceWithReflection( String a){
        System.out.println("Constructor :String => "+a);
    }
    public static void main(String args[]) throws Exception {
        NewInstanceWithReflection object =
        (NewInstanceWithReflection)Class.forName("NewInstanceWithReflection").newInstance();
        Constructor constructor = NewInstanceWithReflection.class.getDeclaredConstructor( new
        Class[] {String.class});
        NewInstanceWithReflection object1 = (NewInstanceWithReflection)constructor.newInstance(new
        Object[]{"StackOverFlow"});
    }
}
```

output:

默认构造函数

构造函数 :String => StackOverFlow

说明：

1. 使用Class.forName创建类的实例：它调用默认构造函数
2. 通过传递参数类型作为Class数组，调用类的getDeclaredConstructor方法
3. 获取构造函数后，通过传递参数值作为Object数组，创建newInstance

Default constructor

Constructor :String => StackOverFlow

Explanation:

1. Create instance of class using `Class.forName` : It calls default constructor
2. Invoke `getDeclaredConstructor` of the class by passing type of parameters as `Class` array
3. After getting the constructor, create `newInstance` by passing parameter value as `Object` array

第104章：ByteBuffer

ByteBuffer类在Java 1.4中引入，以简化对二进制数据的操作。它特别适合用于原始类型数据。它不仅允许创建byte[]，还允许在更高的抽象层次上对其进行后续操作

第104.1节：基本用法 - 使用DirectByteBuffer

DirectByteBuffer是ByteBuffer的特殊实现，它没有底层的byte[]

我们可以通过调用以下方法分配这样的ByteBuffer：

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(16);
```

此操作将分配16字节的内存。直接缓冲区的内容可能位于普通垃圾回收堆之外。

我们可以通过调用以下方法来验证 ByteBuffer 是否为直接缓冲区：

```
directBuffer.isDirect(); // true
```

DirectByteBuffer的主要特点是JVM会尝试直接操作分配的内存，而不进行任何额外的缓冲，因此对其执行的操作可能比基于数组的ByteBuffer更快。

建议在依赖执行速度的重度IO操作中使用DirectByteBuffer，比如实时通信。

我们需要注意，如果尝试使用array()方法，会抛出UnsupportedOperationException异常。因此，最好先检查我们的ByteBuffer是否有（字节数组）再访问它：

```
byte[] arrayOfBytes;
if(buffer.hasArray()) {
    arrayOfBytes = buffer.array();
}
```

直接字节缓冲区的另一个用途是通过JNI进行互操作。由于直接字节缓冲区不使用byte[]，而是使用实际的内存块，因此可以通过本地代码中的指针直接访问该内存。这可以节省在Java和本地数据表示之间进行封送处理时的一些麻烦和开销。

JNI接口定义了几个函数来处理直接字节缓冲区：NIO支持。

第104.2节：基本用法 - 创建ByteBuffer

创建ByteBuffer有两种方式，其中一种还可以细分。

如果你已经有一个现成的byte[]，可以将其“包装”成ByteBuffer以简化处理：

```
byte[] reqBuffer = new byte[BUFFER_SIZE];
int readBytes = socketInputStream.read(reqBuffer);
final ByteBuffer reqBufferWrapper = ByteBuffer.wrap(reqBuffer);
```

这可能是处理低级网络交互的代码示例

Chapter 104: ByteBuffer

The ByteBuffer class was introduced in java 1.4 to ease working on binary data. It's especially suited to use with primitive type data. It allows the creation, but also subsequent manipulation of a byte[]s on a higher abstraction level

Section 104.1: Basic Usage - Using DirectByteBuffer

DirectByteBuffer is special implementation of ByteBuffer that has no byte[] laying underneath.

We can allocate such ByteBuffer by calling:

```
ByteBuffer directBuffer = ByteBuffer.allocateDirect(16);
```

This operation will allocate 16 bytes of memory. The contents of direct buffers *may* reside outside of the normal garbage-collected heap.

We can verify whether ByteBuffer is direct by calling:

```
directBuffer.isDirect(); // true
```

The main characteristics of DirectByteBuffer is that JVM will try to natively work on allocated memory without any additional buffering so operations performed on it may be faster then those performed on ByteBuffers with arrays lying underneath.

It is recommended to use DirectByteBuffer with heavy IO operations that rely on speed of execution, like real time communication.

We have to be aware that if we try using array() method we will get `UnsupportedOperationException`. So it is a good practice to check whether our ByteBuffer has it (byte array) before we try to access it:

```
byte[] arrayOfBytes;
if(buffer.hasArray()) {
    arrayOfBytes = buffer.array();
}
```

Another use of direct byte buffer is interop through JNI. Since a direct byte buffer does not use a byte[], but an actual block of memory, it is possible to access that memory directly through a pointer in native code. This can save a bit of trouble and overhead on marshalling between the Java and native representation of data.

The JNI interface defines several functions to handle direct byte buffers: [NIO Support](#).

Section 104.2: Basic Usage - Creating a ByteBuffer

There's two ways to create a ByteBuffer, where one can be subdivided again.

If you have an already existing byte[], you can "wrap" it into a ByteBuffer to simplify processing:

```
byte[] reqBuffer = new byte[BUFFER_SIZE];
int readBytes = socketInputStream.read(reqBuffer);
final ByteBuffer reqBufferWrapper = ByteBuffer.wrap(reqBuffer);
```

This would be a possibility for code that handles low-level networking interactions

如果你还没有现成的byte[], 可以像这样在专门分配的数组上创建一个ByteBuffer作为缓冲区：

```
final ByteBuffer respBuffer = ByteBuffer.allocate(RESPONSE_BUFFER_SIZE);
putResponseData(respBuffer);
socketOutputStream.write(respBuffer.array());
```

如果代码路径对性能极其关键，并且你需要直接系统内存访问，ByteBuffer甚至可以使用#allocateDirect()分配direct缓冲区

第104.3节：基本用法 - 向缓冲区写入数据

给定一个ByteBuffer实例，可以使用relative和absoluteput方法写入原始类型数据。显著的区别是，使用relative方法写入数据时，会自动跟踪数据插入的索引，而绝对方法总是需要指定索引来put数据。

这两种方法都支持“链式”调用。只要缓冲区足够大，就可以按如下方式操作：

```
buffer.putInt(0xCAFEBAE).putChar('c').putFloat(0.25).putLong(0xDEADBEEFCAFEBAE);
```

等价于：

```
buffer.putInt(0xCAFEBAE);
buffer.putChar('c');
buffer.putFloat(0.25);
buffer.putLong(0xDEADBEEFCAFEBAE);
```

请注意，针对byte操作的方法没有特别命名。另外请注意，传入ByteBuffer和byte[]给put也是有效的。除此之外，所有基本类型都有专门的put方法。

另一个注意事项：使用绝对位置的put*时，给定的索引总是以byte为单位计数。

If you do not have an already existing `byte[]`, you can create a ByteBuffer over an array that's specifically allocated for the buffer like this:

```
final ByteBuffer respBuffer = ByteBuffer.allocate(RESPONSE_BUFFER_SIZE);
putResponseData(respBuffer);
socketOutputStream.write(respBuffer.array());
```

If the code-path is extremely performance critical and you need **direct system memory access**, the ByteBuffer can even allocate *direct* buffers using `#allocateDirect()`

Section 104.3: Basic Usage - Write Data to the Buffer

Given a ByteBuffer instance one can write primitive-type data to it using *relative* and *absolute* put. The striking difference is that putting data using the *relative* method keeps track of the index the data is inserted at for you, while the absolute method always requires giving an index to put the data at.

Both methods allow "chaining" calls. Given a sufficiently sized buffer one can accordingly do the following:

```
buffer.putInt(0xCAFEBAE).putChar('c').putFloat(0.25).putLong(0xDEADBEEFCAFEBAE);
```

which is equivalent to:

```
buffer.putInt(0xCAFEBAE);
buffer.putChar('c');
buffer.putFloat(0.25);
buffer.putLong(0xDEADBEEFCAFEBAE);
```

Do note that the method operating on `bytes` is not named specially. Additionally note that it's also valid to pass both a ByteBuffer and a `byte[]` to put. Other than that, all primitive types have specialized put-methods.

An additional note: The index given when using absolute put* is always counted in `bytes`.

第105章：小程序 (Applets)

小程序自Java正式发布以来一直存在，并且多年来一直被用于教授Java和编程。

近年来，推动摆脱小程序和其他浏览器插件的趋势日益活跃，一些浏览器阻止或主动不支持它们。

2016年，Oracle宣布计划弃用该插件，[迈向无插件的网页](#)

现有更新更好的API可用

第105.1节：最小Applet

一个非常简单的Applet绘制一个矩形并在屏幕上打印一段字符串。

```
public class MyApplet extends JApplet{  
  
    private String str = "StackOverflow";  
  
    @Override  
    public void init() {  
        setBackground(Color.gray);  
    }  
    @Override  
    public void destroy() {}  
    @Override  
    public void start() {}  
    @Override  
    public void stop() {}  
    @Override  
    public void paint(Graphics g) {  
        g.setColor(Color.yellow);  
        g.fillRect(1,1,300,150);  
        g.setColor(Color.red);  
        g.setFont(new Font("TimesRoman", Font.PLAIN, 48));  
        g.drawString(str, 10, 80);  
    }  
}
```

小程序的主类继承自javax.swing.JApplet。

版本 ≤ Java SE 1.2

在 Java 1.2 之前以及 swing API 引入之前，小程序继承自java.applet.Applet。

小程序不需要 main 方法。入口点由生命周期控制。要使用它们，需要将其嵌入到 HTML 文档中。这也是定义其大小的地方。

```
<html>  
    <head></head>  
    <body>  
        <applet code="MyApplet.class" width="400" height="200"></applet>  
    </body>  
</html>
```

Chapter 105: Applets

Applets have been part of Java since its official release and have been used to teach Java and programming for a number of years.

Recent years have seen an active push to move away from Applets and other browser plugins, with some browsers blocking them or actively not supporting them.

In 2016, Oracle announced their plans to deprecate the plugin, [Moving to a Plugin-Free Web](#)

Newer and better APIs are now available

Section 105.1: Minimal Applet

A very simple applet draws a rectangle and prints a string something on the screen.

```
public class MyApplet extends JApplet{  
  
    private String str = "StackOverflow";  
  
    @Override  
    public void init() {  
        setBackground(Color.gray);  
    }  
    @Override  
    public void destroy() {}  
    @Override  
    public void start() {}  
    @Override  
    public void stop() {}  
    @Override  
    public void paint(Graphics g) {  
        g.setColor(Color.yellow);  
        g.fillRect(1,1,300,150);  
        g.setColor(Color.red);  
        g.setFont(new Font("TimesRoman", Font.PLAIN, 48));  
        g.drawString(str, 10, 80);  
    }  
}
```

The main class of an applet extends from javax.swing.JApplet.

Version ≤ Java SE 1.2

Before Java 1.2 and the introduction of the swing API applets had extended from java.applet.Applet.

Applets don't require a main method. The entry point is controlled by the life cycle. To use them, they need to be embedded in a HTML document. This is also the point where their size is defined.

```
<html>  
    <head></head>  
    <body>  
        <applet code="MyApplet.class" width="400" height="200"></applet>  
    </body>  
</html>
```

第105.2节：创建图形用户界面（GUI）

Applet可以轻松用于创建GUI。它们像一个容器，并且有一个add()方法，可以接受任何awt或swing组件。

```
public class MyGUIApplet extends JApplet{  
  
    private JPanel panel;  
    private JButton button;  
    private JComboBox<String> comboBox;  
    private JTextField textField;  
  
    @Override  
    public void init(){  
        panel = new JPanel();  
        button = new JButton("ClickMe!");  
        button.addActionListener(new ActionListener(){  
            @Override  
            public void actionPerformed(ActionEvent ae) {  
                if(((String)comboBox.getSelectedItem()).equals("greet")) {  
                    JOptionPane.showMessageDialog(null, "Hello " + textField.getText());  
                } else {  
                    JOptionPane.showMessageDialog(null, textField.getText() + " stinks!");  
                }  
            }  
        });  
        comboBox = new JComboBox<>(new String[]{"greet", "offend"});  
        textField = new JTextField("John Doe");  
        panel.add(comboBox);  
        panel.add(textField);  
        panel.add(button);  
        add(panel);  
    }  
}
```

Section 105.2: Creating a GUI

Applets could easily be used to create a GUI. They act like a Container and have an add() method that takes any awt or swing component.

```
public class MyGUIApplet extends JApplet{  
  
    private JPanel panel;  
    private JButton button;  
    private JComboBox<String> comboBox;  
    private JTextField textField;  
  
    @Override  
    public void init(){  
        panel = new JPanel();  
        button = new JButton("ClickMe!");  
        button.addActionListener(new ActionListener(){  
            @Override  
            public void actionPerformed(ActionEvent ae) {  
                if(((String)comboBox.getSelectedItem()).equals("greet")) {  
                    JOptionPane.showMessageDialog(null, "Hello " + textField.getText());  
                } else {  
                    JOptionPane.showMessageDialog(null, textField.getText() + " stinks!");  
                }  
            }  
        });  
        comboBox = new JComboBox<>(new String[]{"greet", "offend"});  
        textField = new JTextField("John Doe");  
        panel.add(comboBox);  
        panel.add(textField);  
        panel.add(button);  
        add(panel);  
    }  
}
```

第105.3节：从applet内部打开链接

您可以使用方法getAppletContext()来获取一个AppletContext对象，该对象允许您请求浏览器打开链接。为此，您使用方法showDocument()。它的第二个参数告诉浏览器使用新窗口_blank还是显示applet的窗口_self。

```
public class MyLinkApplet extends JApplet{  
    @Override  
    public void init(){  
        JButton button = new JButton("点击我！");  
        button.addActionListener(new ActionListener(){  
            @Override  
            public void actionPerformed(ActionEvent ae) {  
                AppletContext a = getAppletContext();  
                try {  
                    URL url = new URL("http://stackoverflow.com/");  
a.showDocument(url, "_blank");  
                } catch (Exception e) { /* 省略部分代码 */ }  
            }  
        });  
        add(button);  
    }  
}
```

Section 105.3: Open links from within the applet

You can use the method getAppletContext() to get an AppletContext object that allows you to request the browser to open a link. For this you use the method showDocument(). Its second parameter tells the browser to use a new window _blank or the one that shows the applet _self.

```
public class MyLinkApplet extends JApplet{  
    @Override  
    public void init(){  
        JButton button = new JButton("ClickMe!");  
        button.addActionListener(new ActionListener(){  
            @Override  
            public void actionPerformed(ActionEvent ae) {  
                AppletContext a = getAppletContext();  
                try {  
                    URL url = new URL("http://stackoverflow.com/");  
a.showDocument(url, "_blank");  
                } catch (Exception e) { /* omitted for brevity */ }  
            }  
        });  
        add(button);  
    }  
}
```

第105.4节：加载图像、音频和其他资源

Java小程序能够加载不同的资源。但由于它们运行在客户端的网页浏览器中，你需要确保这些资源是可访问的。小程序无法访问客户端资源，如本地文件系统。

如果你想从存储小程序的相同URL加载资源，可以使用方法`getCodeBase()`来获取基础URL。为了加载资源，小程序提供了方法`getImage()`和`getAudioClip()`来加载图像或音频文件。

加载并显示图像

```
public class MyImgApplet extends JApplet{  
  
    private Image img;  
  
    @Override  
    public void init(){  
        try {  
            img = getImage(new URL("http://cdn.sstatic.net/stackexchange/img/logos/so/logo.png"));  
        } catch (MalformedURLException e) { /* 省略细节 */ }  
    }  
    @Override  
    public void paint(Graphics g) {  
        g.drawImage(img, 0, 0, this);  
    }  
}
```

加载并播放音频文件

```
public class MyAudioApplet extends JApplet{  
  
    private AudioClip audioClip;  
  
    @Override  
    public void init(){  
        try {  
            audioClip = getAudioClip(new URL("URL/TO/AN/AUDIO/FILE.WAV"));  
        } catch (MalformedURLException e) { /* omitted for brevity */ }  
    }  
    @Override  
    public void start() {  
        audioClip.play();  
    }  
    @Override  
    public void stop(){  
        audioClip.stop();  
    }  
}
```

加载并显示文本文件

```
public class MyTextApplet extends JApplet{  
    @Override  
    public void init(){  
        JTextArea textArea = new JTextArea();  
        JScrollPane sp = new JScrollPane(textArea);  
        add(sp);  
    }  
}
```

Section 105.4: Loading images, audio and other resources

Java applets are able to load different resources. But since they are running in the web browser of the client you need to make sure that these resources are accessible. Applets are not able to access client resources as the local file system.

If you want to load resources from the same URL the Applet is stored you can use the method `getCodeBase()` to retrieve the base URL. To load resources, applets offer the methods `getImage()` and `getAudioClip()` to load images or audio files.

Load and show an image

```
public class MyImgApplet extends JApplet{  
  
    private Image img;  
  
    @Override  
    public void init(){  
        try {  
            img = getImage(new URL("http://cdn.sstatic.net/stackexchange/img/logos/so/logo.png"));  
        } catch (MalformedURLException e) { /* omitted for brevity */ }  
    }  
    @Override  
    public void paint(Graphics g) {  
        g.drawImage(img, 0, 0, this);  
    }  
}
```

Load and play an audio file

```
public class MyAudioApplet extends JApplet{  
  
    private AudioClip audioClip;  
  
    @Override  
    public void init(){  
        try {  
            audioClip = getAudioClip(new URL("URL/TO/AN/AUDIO/FILE.WAV"));  
        } catch (MalformedURLException e) { /* omitted for brevity */ }  
    }  
    @Override  
    public void start() {  
        audioClip.play();  
    }  
    @Override  
    public void stop(){  
        audioClip.stop();  
    }  
}
```

Load and display a text file

```
public class MyTextApplet extends JApplet{  
    @Override  
    public void init(){  
        JTextArea textArea = new JTextArea();  
        JScrollPane sp = new JScrollPane(textArea);  
        add(sp);  
    }  
}
```

```
// 加载文本
try {
    URL url = new URL("http://www.textfiles.com/fun/quotes.txt");
    InputStream in = url.openStream();
    BufferedReader bf = new BufferedReader(new InputStreamReader(in));
    String line = "";
    while((line = bf.readLine()) != null) {
textArea.append(line + "\n");
    }
} catch(Exception e) { /* 省略部分代码 */ }
}
```

```
// load text
try {
    URL url = new URL("http://www.textfiles.com/fun/quotes.txt");
    InputStream in = url.openStream();
    BufferedReader bf = new BufferedReader(new InputStreamReader(in));
    String line = "";
    while((line = bf.readLine()) != null) {
        textArea.append(line + "\n");
    }
} catch(Exception e) { /* omitted for brevity */ }
}
```

第106章：表达式

Java中的表达式是进行计算的主要构造。

第106.1节：运算符优先级

当一个表达式包含多个运算符时，可能会有不同的解析方式。例如，数学表达式 $1 + 2 \times 3$ 可以有两种读法：

- 将1和2相加，然后将结果乘以3。这样得到的答案是9。如果我们加上括号，表达式会是像 $(1 + 2) \times 3$ 。
- 将1加到2和3相乘的结果中。这得到答案7。如果我们加上括号，看起来会是像 $1 + (2 \times 3)$ 。

在数学中，惯例是按照第二种方式来读表达式。一般规则是先进行乘法和除法，再进行加法和减法。当使用更高级的数学符号时，含义要么对受过训练的数学家来说是“显而易见”的，要么加上括号以消除歧义。无论哪种情况，符号传达意义的有效性都取决于数学家的智力和共同知识。

Java也有同样明确的规则来读取表达式，基于所使用运算符的优先级。

一般来说，每个运算符都有一个优先级值；见下表。

例如：

`1 + 2 * 3`

`+`的优先级低于`*`的优先级，所以表达式的结果是7，而不是9。

描述	运算符 / 结构 (主要)	优先级	结合性
限定符	名称.名称		
括号	(表达式)		
实例创建	<code>new</code> (类)		
字段访问	主表达式.名称	15	从左到右
数组访问	主表达式[表达式]		
方法调用	主表达式(表达式, ...)		
方法引用	主表达式::名称		
后置递增	表达式++, 表达式--	14	-
前置递增	<code>++expr, --expr,</code>		
一元	<code>+expr, -expr, ~expr, !expr,</code> <code>(type)expr</code>	13	从右到左 从右到左
类型转换 ¹			
乘法	<code>* / %</code>	12	从左到右
加法	<code>+ -</code>	11	从左到右
移位	<code><< >> >>></code>	10	从左到右
关系	<code>< > <= >= instanceof</code>	9	从左到右
平等	<code>== !=</code>	8	从左到右
按位与	<code>&</code>	7	从左到右
按位异或 ^		6	从左到右
按位或		5	从左到右
逻辑与	<code>&&</code>	4	从左到右

Chapter 106: Expressions

Expressions in Java are the primary construct for doing calculations.

Section 106.1: Operator Precedence

When an expression contains multiple operators, it can potentially be read in different ways. For example, the mathematical expression `1 + 2 × 3` could be read in two ways:

- Add 1 and 2 and multiply the result by 3. This gives the answer 9. If we added parentheses, this would look like `(1 + 2) × 3`.
- Add 1 to the result of multiplying 2 and 3. This gives the answer 7. If we added parentheses, this would look like `1 + (2 × 3)`.

In mathematics, the convention is to read the expression the second way. The general rule is that multiplication and division are done before addition and subtraction. When more advanced mathematical notation is used, either the meaning is either "self-evident" (to a trained mathematician!), or parentheses are added to disambiguate. In either case, the effectiveness of the notation to convey meaning depends on the intelligence and shared knowledge of the mathematicians.

Java has the same clear rules on how to read an expression, based on the *precedence* of the operators that are used.

In general, each operator is ascribed a *precedence* value; see the table below.

For example:

`1 + 2 * 3`

The precedence of `+` is lower than the precedence of `*`, so the result of the expression is 7, not 9.

Description	Operators / constructs (primary)	Precedence	Associativity
Qualifier	<code>name.name</code>		
Parentheses	<code>(expr)</code>		
Instance creation	<code>new</code>		
Field access	<code>primary.name</code>	15	Left to right
Array access	<code>primary[expr]</code>		
Method invocation	<code>primary(expr, ...)</code>		
Method reference	<code>primary::name</code>		
Post increment	<code>expr++, expr--</code>	14	-
Pre increment	<code>++expr, --expr,</code>		
Unary	<code>+expr, -expr, ~expr, !expr,</code>	13	Right to left
Cast1	<code>(type)expr</code>		Right to left
Multiplicative	<code>* / %</code>	12	Left to right
Additive	<code>+ -</code>	11	Left to right
Shift	<code><< >> >>></code>	10	Left to right
Relational	<code>< > <= >= instanceof</code>	9	Left to right
Equality	<code>== !=</code>	8	Left to right
Bitwise AND	<code>&</code>	7	Left to right
Bitwise exclusive OR ^		6	Left to right
Bitwise inclusive OR		5	Left to right
Logical AND	<code>&&</code>	4	Left to right

逻辑或		3	从左到右
条件1	?:	2	从右到左
赋值	= *= /= %= += -= <<= >>= >>>= &= ^= =	1	从右到左
Lambda1	->		

1 Lambda 表达式的优先级较为复杂，因为它也可以出现在类型转换之后，或者作为条件三元运算符的第三部分。

第106.2节：表达式基础

Java中的表达式是进行计算的主要构造。以下是一些示例：

```

1          // 一个简单的字面量是一个表达式
1 + 2      // 一个简单的表达式，计算两个数字的和
(i + j) / k // 一个包含多个运算的表达式
(flag) ? c : d // 使用"条件"运算符的表达式
(String) s    // 类型转换是一个表达式
obj.test()    // 方法调用是一个表达式
new Object()  // 创建对象是一个表达式
new int[]     // 创建对象是一个表达式
  
```

一般来说，表达式由以下形式组成：

- 表达式名称包括：
 - 简单标识符；例如 `someIdentifier`
 - 限定标识符；例如 `MyClass.someField`
- 主表达式包括：
 - 字面量；例如 `1`、`1.0`、`'X'`、`"hello"`、`false` 和 `null`
 - 类字面量表达式；例如 `MyClass.class`
 - `this` 和 `<TypeName>.this`
 - 带括号的表达式；例如 `(a + b)`
 - 类实例创建表达式；例如 `new MyClass(1, 2, 3)`
 - 数组实例创建表达式；例如 `new int[3]`
 - 字段访问表达式；例如 `obj.someField` 或 `this.someField`
 - 数组访问表达式；例如 `vector[21]`
 - 方法调用；例如 `obj.doIt(1, 2, 3)`
 - 方法引用 (Java 8及以后)；例如 `MyClass::doIt`
- 一元运算符表达式；例如 `!a` 或 `i++`
- 二元运算符表达式；例如 `a + b` 或 `obj == null`
- 三元运算符表达式；例如 `(obj == null) ? 1 : obj.getCount()`
- Lambda表达式 (Java 8及以后)；例如 `obj -> obj.getCount()`

不同形式表达式的详细内容可在其他主题中找到。

- 运算符主题涵盖一元、二元和三元运算符表达式。
- Lambda表达式主题涵盖Lambda表达式和方法引用表达式。
- 类与对象主题涵盖类实例创建表达式。
- 数组主题涵盖数组访问表达式和数组实例创建表达式。
- 字面量主题涵盖了不同种类的字面量表达式。

表达式的类型

在大多数情况下，表达式具有可以通过检查其子表达式在编译时确定的静态类型。这些被称为独立表达式。

Logical OR		3	Left to right
Conditional1	?:	2	Right to left
Assignment	= *= /= %= += -= <<= >>= >>>= &= ^= =	1	Right to left
Lambda1	->		

1 Lambda 表达式的优先级较为复杂，因为它也可以出现在类型转换之后，或者作为条件三元运算符的第三部分。

Section 106.2: Expression Basics

Expressions in Java are the primary construct for doing calculations. Here are some examples:

```

1          // A simple literal is an expression
1 + 2      // A simple expression that adds two numbers
(i + j) / k // An expression with multiple operations
(flag) ? c : d // An expression using the "conditional" operator
(String) s    // A type-cast is an expression
obj.test()    // A method call is an expression
new Object()  // Creation of an object is an expression
new int[]     // Creation of an object is an expression
  
```

In general, an expression consists of the following forms:

- Expression names which consist of:
 - Simple identifiers; e.g. `someIdentifier`
 - Qualified identifiers; e.g. `MyClass.someField`
- Primaries which consist of:
 - Literals; e.g. `1`, `1.0`, `'X'`, `"hello"`, `false` and `null`
 - Class literal expressions; e.g. `MyClass.class`
 - `this` and `<TypeName>.this`
 - Parenthesized expressions; e.g. `(a + b)`
 - Class instance creation expressions; e.g. `new MyClass(1, 2, 3)`
 - Array instance creation expressions; e.g. `new int[3]`
 - Field access expressions; e.g. `obj.someField` or `this.someField`
 - Array access expressions; e.g. `vector[21]`
 - Method invocations; e.g. `obj.doIt(1, 2, 3)`
 - Method references (Java 8 and later); e.g. `MyClass::doIt`
- Unary operator expressions; e.g. `!a` or `i++`
- Binary operator expressions; e.g. `a + b` or `obj == null`
- Ternary operator expressions; e.g. `(obj == null) ? 1 : obj.getCount()`
- Lambda expressions (Java 8 and later); e.g. `obj -> obj.getCount()`

The details of the different forms of expressions may be found in other Topics.

- The Operators topic covers unary, binary and ternary operator expressions.
- The Lambda expressions topic covers lambda expressions and method reference expressions.
- The Classes and Objects topic covers class instance creation expressions.
- The Arrays topic covers array access expressions and array instance creation expressions.
- The Literals topic covers the different kinds of literals expressions.

The Type of an Expression

In most cases, an expression has a static type that can be determined at compile time by examining and its subexpressions. These are referred to as *stand-alone* expressions.

然而，（在Java 8及以后版本中）以下类型的表达式可能是多态表达式：

- 带括号的表达式
- 类实例创建表达式
- 方法调用表达式
- 方法引用表达式
- 条件表达式
- Lambda表达式

当表达式是多态表达式时，其类型可能会受到表达式的目标类型的影响；即它被用于什么目的。

表达式的值

表达式的值与其类型在赋值上是兼容的。例外情况是发生了堆污染；例如因为“非安全转换”警告被（不适当）抑制或忽略。

表达式语句

与许多其他语言不同，Java 通常不允许将表达式用作语句。例如：

```
public void compute(int i, int j) {  
    i + j; // 错误  
}
```

由于像这样的表达式的计算结果不能被使用，且它不能以其他方式影响程序的执行，Java 设计者认为这种用法要么是错误的，要么是误导性的。

然而，这并不适用于所有表达式。一部分表达式实际上是合法的语句。该集合包括：

- 赋值表达式，包括操作并赋值的赋值。
- 前置和后置的自增与自减表达式。
- 方法调用（void或非void）。
- 类实例创建表达式。

第106.3节：表达式求值顺序

Java 表达式的求值遵循以下规则：

- 操作数从左到右进行求值。
- 操作符的操作数在操作符之前被求值。
- 操作符根据优先级进行求值
- 参数列表从左到右进行求值。

简单示例

在以下示例中：

```
int i = method1() + method2();
```

求值顺序是：

1. = 操作符的左操作数被求值为 i 的地址。
2. + 操作符的左操作数（method1()）被求值。

However, (in Java 8 and later) the following kinds of expressions may be *poly expressions*:

- Parenthesized expressions
- Class instance creation expressions
- Method invocation expressions
- Method reference expressions
- Conditional expressions
- Lambda expressions

When an expression is a poly expression, its type may be influenced by the expression's *target type*; i.e. what it is being used for.

The value of an Expression

The value of an expression is assignment compatible with its type. The exception to this is when *heap pollution* has occurred; e.g. because "unsafe conversion" warnings have been (inappropriately) suppressed or ignored.

Expression Statements

Unlike many other languages, Java does not generally allow expressions to be used as statements. For example:

```
public void compute(int i, int j) {  
    i + j; // ERROR  
}
```

Since the result of evaluating an expression like cannot be used, and since it cannot affect the execution of the program in any other way, the Java designers took the position that such usage is either a mistake, or misguided.

However, this does not apply to all expressions. A subset of expressions are (in fact) legal as statements. The set comprises:

- Assignment expression, including *operation-and-becomes* assignments.
- Pre and post increment and decrement expressions.
- Method calls (**void** or non-**void**).
- Class instance creation expressions.

Section 106.3: Expression evaluation order

Java expressions are evaluated following the following rules:

- Operands are evaluated from left to right.
- The operands of an operator are evaluated before the operator.
- Operators are evaluated according to operator precedence
- Argument lists are evaluated from left to right.

Simple Example

In the following example:

```
int i = method1() + method2();
```

the order of evaluation is:

1. The left operand of = operator is evaluated to the address of i.
2. The left operand of the + operator (method1()) is evaluated.

3. 右操作数+运算符 (method2()) 被求值。
4. +运算被求值。
5. =运算被求值，将加法结果赋值给 i。

注意，如果调用的效果是可观察的，你将能够观察到对method1的调用发生在对method2的调用之前。

带有副作用的运算符示例

在以下示例中：

```
int i = 1;
intArray[i] = ++i + 1;
```

求值顺序是：

1. =运算符的左操作数被求值。这给出了intArray[1]的地址。
2. 前置递增被求值。这会给 i 加上1，并求值为2。
3. +的右操作数被求值。
4. 加法操作计算结果为：2 + 1 -> 3。
5. 该=操作被计算，将3赋值给intArray[1]。

注意，由于=的左操作数先被计算，它不会受到++i 子表达式副作用的影响。

参考：

- [JLS 15.7 - 计算顺序](#)

第106.4节：常量表达式

常量表达式是产生原始类型或字符串的表达式，其值可以在编译时计算为字面量。该表达式必须在不抛出异常的情况下求值，并且必须仅由以下内容组成：

- 原始类型和字符串字面量。
- 向原始类型或String的类型转换。
- 以下一元运算符：+、-、~和!。
- 以下二元运算符：*、/、%、+、-、<<、>>、<、<=、>、>=、==、!=、&、^、|、&&和||。
- 三元条件运算符？:。
- 带括号的常量表达式。
- 引用常量变量的简单名称。（常量变量是指声明为final且初始化表达式本身是常量表达式的变量。）
- 形式为<TypeName>.<Identifier>的限定名称，引用常量变量。

注意，上述列表不包括++和--，赋值运算符，class和instanceof，方法调用以及对一般变量或字段的引用。

类型为String的常量表达式会产生一个“驻留”的String，常量表达式中的浮点运算采用FP严格语义进行计算。

3. The right operand of the + operator (method2()) is evaluated.
4. The + operation is evaluated.
5. The = operation is evaluated, assigning the result of the addition to i.

Note that if the effects of the calls are observable, you will be able to observe that the call to method1 occurs before the call to method2.

Example with an operator which has a side-effect

In the following example:

```
int i = 1;
intArray[i] = ++i + 1;
```

the order of evaluation is:

1. The left operand of = operator is evaluated. This gives the address of intArray[1].
2. The pre-increment is evaluated. This adds 1 to i, and evaluates to 2.
3. The right hand operand of the + is evaluated.
4. The + operation is evaluated to: 2 + 1 -> 3.
5. The = operation is evaluated, assigning 3 to intArray[1].

Note that since the left-hand operand of the = is evaluated first, it is not influenced by the side-effect of the ++i subexpression.

Reference:

- [JLS 15.7 - Evaluation Order](#)

Section 106.4: Constant Expressions

A constant expression is an expression that yields a primitive type or a String, and whose value can be evaluated at compile time to a literal. The expression must evaluate without throwing an exception, and it must be composed of only the following:

- Primitive and String literals.
- Type casts to primitive types or [String](#).
- The following unary operators: +, -, ~ and !.
- The following binary operators: *, /, %, +, -, <<, >>, >>>, <, <=, >, >=, ==, !=, &, ^, |, && and ||.
- The ternary conditional operator ? :.
- Parenthesized constant expressions.
- Simple names that refer to constant variables. (A constant variable is a variable declared as [final](#) where the initializer expression is itself a constant expression.)
- Qualified names of the form <TypeName> . <Identifier> that refer to constant variables.

Note that the above list excludes ++ and --, the assignment operators, [class](#) and [instanceof](#), method calls and references to general variables or fields.

Constant expressions of type [String](#) result in an "interned" [String](#), and floating point operations in constant expressions are evaluated with FP-strict semantics.

常量表达式的用途

常量表达式几乎可以在普通表达式可用的任何地方使用。但它们在以下上下文中具有特殊意义。

switch语句中的case标签需要常量表达式。例如：

```
switch (someValue) {  
    case 1 + 1: // 正确  
    case Math.min(2, 3): // 错误 - 不是常量表达式  
        doSomething();  
}
```

当赋值右侧的表达式是常量表达式时，赋值可以执行原始类型的缩小转换。只要常量表达式的值在赋值左侧类型的范围内，这种转换是允许的。（参见JLS 5.1.3和5.2）例如：

```
byte b1 = 1 + 1; // 正确 - 原始类型缩小转换。  
byte b2 = 127 + 1; // 错误 - 超出范围  
byte b3 = b1 + 1; // 错误 - 不是常量表达式  
byte b4 = (byte) (b1 + 1); // 正确
```

当常量表达式用作 do、while 或 for 的条件时，会影响可读性分析。
例如：

```
while (false) {  
    doSomething(); // 错误 - 语句不可达  
}  
  
boolean flag = false;  
while (flag) {  
    doSomething(); // 正确  
}
```

（请注意，这不适用于if语句。Java编译器允许if语句的then或else代码块不可达。这是C和C++中条件编译的Java对应实现。）

最后，带有常量表达式初始化器的类或接口中的static final字段会被立即初始化。因此，即使类初始化依赖图中存在循环，也能保证这些常量在初始化状态下被观察到。

更多信息请参阅[JLS 15.28. 常量表达式](#)。

Uses for Constant Expressions

Constant expressions can be used (just about) anywhere that a normal expression can be used. However, they have a special significance in the following contexts.

Constant expressions are required for case labels in switch statements. For example:

```
switch (someValue) {  
    case 1 + 1: // OK  
    case Math.min(2, 3): // Error - not a constant expression  
        doSomething();  
}
```

When the expression on the right hand side of an assignment is a constant expression, then the assignment can perform a primitive narrowing conversion. This is allowed provided that the value of the constant expression is within the range of the type on the left hand side. (See [JLS 5.1.3](#) and [5.2](#)) For example:

```
byte b1 = 1 + 1; // OK - primitive narrowing conversion.  
byte b2 = 127 + 1; // Error - out of range  
byte b3 = b1 + 1; // Error - not a constant expression  
byte b4 = (byte) (b1 + 1); // OK
```

When a constant expression is used as the condition in a do, **while** or **for**, then it affects the readability analysis.
For example:

```
while (false) {  
    doSomething(); // Error - statement not reachable  
}  
  
boolean flag = false;  
while (flag) {  
    doSomething(); // OK  
}
```

(Note that this does not apply if statements. The Java compiler allows the then or else block of an if statement to be unreachable. This is the Java analog of conditional compilation in C and C++.)

Finally, static final fields in a class or interface with constant expression initializers are initialized eagerly. Thus, it is guaranteed that these constants will be observed in the initialized state, even when there is a cycle in the class initialization dependency graph.

For more information, refer to [JLS 15.28. Constant Expressions](#).

第107章：Java中的JSON

JSON (JavaScript对象表示法) 是一种轻量级、基于文本、与语言无关的数据交换格式，便于人类和机器读取与编写。JSON可以表示两种结构化类型：对象和数组。JSON常用于Ajax应用、配置、数据库和RESTful Web服务。Java JSON处理API提供了可移植的API，用于解析、生成、转换和查询JSON。

第107.1节：使用Jackson对象映射器

Pojo模型

```
public class Model {  
    private String firstName;  
    private String lastName;  
    private int age;  
    /* 为简洁起见，未显示 getter 和 setter */  
}
```

示例：字符串转对象

```
Model outputObject = objectMapper.readValue(  
    "{\"firstName\":\"John\", \"lastName\":\"Doe\", \"age\":23}",  
    Model.class);  
System.out.println(outputObject.getFirstName());  
//结果：John
```

示例：对象转字符串

```
String jsonString = objectMapper.writeValueAsString(inputObject);  
//结果：{"firstName":"John", "lastName":"Doe", "age":23}
```

详情

需要的导入语句：

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

[Maven 依赖：jackson-databind](#)

ObjectMapper 实例

```
//创建一个  
ObjectMapper objectMapper = new ObjectMapper();
```

- ObjectMapper 是线程安全的
- 推荐：使用共享的静态实例

反序列化：

```
<T> T readValue(String content, Class<T> valueType)
```

- 需要指定 valueType —— 返回值将是该类型
- 抛出异常
 - IOException - 发生底层 I/O 问题时
 - JsonParseException - 如果底层输入包含无效内容
 - JsonMappingException - 如果输入的 JSON 结构与对象结构不匹配

Chapter 107: JSON in Java

JSON (JavaScript Object Notation) is a lightweight, text-based, language-independent data exchange format that is easy for humans and machines to read and write. JSON can represent two structured types: objects and arrays. JSON is often used in Ajax applications, configurations, databases, and RESTful web services. [The Java API for JSON Processing](#) provides portable APIs to parse, generate, transform, and query JSON.

Section 107.1: Using Jackson Object Mapper

Pojo Model

```
public class Model {  
    private String firstName;  
    private String lastName;  
    private int age;  
    /* Getters and setters not shown for brevity */  
}
```

Example: String to Object

```
Model outputObject = objectMapper.readValue(  
    "{\"firstName\":\"John\", \"lastName\":\"Doe\", \"age\":23}",  
    Model.class);  
System.out.println(outputObject.getFirstName());  
//result: John
```

Example: Object to String

```
String jsonString = objectMapper.writeValueAsString(inputObject);  
//result: {"firstName":"John", "lastName":"Doe", "age":23}
```

Details

Import statement needed:

```
import com.fasterxml.jackson.databind.ObjectMapper;
```

[Maven dependency: jackson-databind](#)

ObjectMapper instance

```
//creating one  
ObjectMapper objectMapper = new ObjectMapper();
```

- ObjectMapper is threadsafe
- recommended: have a shared, static instance

Deserialization:

```
<T> T readValue(String content, Class<T> valueType)
```

- valueType needs to be specified -- the return will be of this type
- Throws
 - IOException - in case of a low-level I/O problem
 - JsonParseException - if underlying input contains invalid content
 - JsonMappingException - if the input JSON structure does not match object structure

使用示例 (jsonString 是输入字符串) :

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

序列化方法：

```
String writeValueAsString(Object value)
```

- 抛出

- JsonProcessingException 出错时抛出
- 注意：在2.1版本之前，throws子句包含IOException；2.1版本移除了它。

第107.2节：JSON转对象 (Gson库)

假设你有一个名为Person的类，只有name

```
private class Person {  
    public String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

代码：

```
Gson gson = new Gson();  
String json = "{\"name\": \"John\"}";  
  
Person person = gson.fromJson(json, Person.class);  
System.out.println(person.name); //John
```

你必须在类路径中包含gson库。

第107.3节：JSONObject.NULL

如果你需要添加一个值为null的属性，应该使用预定义的静态常量JSONObject.NULL，而不是标准的Java null引用。

JSONObject.NULL是一个哨兵值，用于显式定义一个空值属性。

```
JSONObject obj = new JSONObject();  
obj.put("some", JSONObject.NULL); //创建: {"some":null}  
System.out.println(obj.get("some")); //输出: null
```

注意

```
JSONObject.NULL.equals(null); //返回 true
```

这明显违反了Java.equals()的契约：

对于任何非空引用值x，x.equals(null)应该返回false

Usage example (jsonString is the input string):

```
Model fromJson = objectMapper.readValue(jsonString, Model.class);
```

Method for serialization:

```
String writeValueAsString(Object value)
```

- Throws

- JsonProcessingException in case of an error
- Note: prior to version 2.1, throws clause included IOException; 2.1 removed it.

Section 107.2: JSON To Object (Gson Library)

Lets assume you have a class called Person with just name

```
private class Person {  
    public String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

Code:

```
Gson gson = new Gson();  
String json = "{\"name\": \"John\"}";  
  
Person person = gson.fromJson(json, Person.class);  
System.out.println(person.name); //John
```

You must have [gson library](#) in your classpath.

Section 107.3: JSONObject.NULL

If you need to add a property with a **null** value, you should use the predefined static final `JSONObject.NULL` and not the standard Java **null** reference.

`JSONObject.NULL` is a sentinel value used to explicitly define a property with an empty value.

```
JSONObject obj = new JSONObject();  
obj.put("some", JSONObject.NULL); //Creates: {"some":null}  
System.out.println(obj.get("some")); //prints: null
```

Note

```
JSONObject.NULL.equals(null); //returns true
```

Which is a **clear violation** of [Java.equals\(\)](#) contract:

For any non-null reference value x, `x.equals(null)` should return false

第107.4节：JSON构建器 - 链式方法

在使用`JSONObject`和`JSONArray`时，可以使用方法链。

JSONObject 示例

```
JSONObject obj = new JSONObject(); //初始化一个空的 JSON 对象  
//之前: {}  
obj.put("name", "Nikita").put("age", "30").put("isMarried", "true");  
//之后: {"name": "Nikita", "age": "30", "isMarried": "true"}
```

JSONArray

```
JSONArray arr = new JSONArray(); //初始化一个空数组  
//之前: []  
arr.put("Stack").put("Over").put("Flow");  
//之后: ["Stack", "Over", "Flow"]
```

第107.5节：对象转 JSON (Gson 库)

假设你有一个名为Person的类，只有name

```
private class Person {  
    public String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

代码：

```
Gson g = new Gson();  
  
Person person = new Person("John");  
System.out.println(g.toJson(person)); // {"name": "John"}
```

当然，Gson jar必须在类路径中。

第107.6节：JSON迭代

遍历JSONObject属性

```
JSONObject obj = new JSONObject("{\"isMarried\": \"true\", \"name\": \"Nikita\", \"age\": \"30\"}");  
Iterator<String> keys = obj.keys(); //所有键：isMarried、name和age  
while (keys.hasNext()) {  
    String key = keys.next(); //只要还有下一个键  
    Object value = obj.get(key); //获取下一个键  
    System.out.println(key + " : " + value); //打印键：值  
}
```

遍历JSONArray值

```
JSONArray arr = new JSONArray(); //初始化一个空数组  
//推入(追加)一些值：  
arr.put("Stack");
```

Section 107.4: JSON Builder - chaining methods

You can use [method chaining](#) while working with `JSONObject` and `JSONArray`.

JSONObject example

```
JSONObject obj = new JSONObject(); //Initialize an empty JSON object  
//Before: {}  
obj.put("name", "Nikita").put("age", "30").put("isMarried", "true");  
//After: {"name": "Nikita", "age": "30", "isMarried": "true"}
```

JSONArray

```
JSONArray arr = new JSONArray(); //Initialize an empty array  
//Before: []  
arr.put("Stack").put("Over").put("Flow");  
//After: ["Stack", "Over", "Flow"]
```

Section 107.5: Object To JSON (Gson Library)

Lets assume you have a class called Person with just name

```
private class Person {  
    public String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
}
```

Code:

```
Gson g = new Gson();  
  
Person person = new Person("John");  
System.out.println(g.toJson(person)); // {"name": "John"}
```

Of course the [Gson](#) jar must be on the classpath.

Section 107.6: JSON Iteration

Iterate over JSONObject properties

```
JSONObject obj = new JSONObject("{\"isMarried\": \"true\", \"name\": \"Nikita\", \"age\": \"30\"}");  
Iterator<String> keys = obj.keys(); //all keys: isMarried, name & age  
while (keys.hasNext()) {  
    String key = keys.next(); //as long as there is another key  
    Object value = obj.get(key); //get next key  
    System.out.println(key + " : " + value); //get next value by key  
}
```

Iterate over JSONArray values

```
JSONArray arr = new JSONArray(); //Initialize an empty array  
//push (append) some values in:  
arr.put("Stack");
```

```

arr.put("Over");
arr.put("Flow");
for (int i = 0; i < arr.length(); i++) {//遍历所有值
    Object value = arr.get(i); //获取值
    System.out.println(value); //打印每个值
}

```

第107.7节：optXXX 与 getXXX 方法

JSONObject 和 JSONArray 有一些方法，在处理你尝试获取的值不存在或类型不符的可能性时非常有用。

```

JSONObject obj = new JSONObject();
obj.putString("foo", "bar");

// 对于存在且类型正确的属性，没有区别
obj.getString("foo"); // 返回 "bar"
obj.optString("foo"); // 返回 "bar"
obj.optString("foo", "tux"); // 返回 "bar"

// 但是，如果值无法强制转换为所需类型，行为会不同
obj.getInt("foo"); // 抛出 JSONException
obj.optInt("foo"); // 返回 0
obj.optInt("foo", 123); // 返回 123

// 如果属性不存在，情况相同
obj.getString("undefined"); // 抛出 JSONException
obj.optString("undefined"); // 返回 ""
obj.optString("undefined", "tux"); // 返回 "tux"

```

相同规则适用于 JSONArray 的 getXXX / optXXX 方法。

第107.8节：从JSON中提取单个元素

```

String json = "{\"name\": \"John\", \"age\":21}";

JsonObject jsonObject = new JsonParser().parse(json).getAsJsonObject();

System.out.println(jsonObject.get("name").getAsString()); //John
System.out.println(jsonObject.get("age").getAsInt()); //21

```

第107.9节：JsonArray转Java列表 (Gson库)

这里有一个简单的JSONArray，你想将其转换为Java的ArrayList：

```
{
    "list": [
        "Test_String_1",
        "Test_String_2"
    ]
}
```

现在将JSONArray 'list' 传递给以下方法，该方法返回对应的JavaArrayList：

```

public ArrayList<String> getListString(String jsonList){
    Type listType = new TypeToken<List<String>>() {}.getType();
    //确保名称 'list' 与你的 'Json' 中的 'JSONArray' 名称匹配。
    ArrayList<String> list = new Gson().fromJson(jsonList, listType);
}

```

```

arr.put("Over");
arr.put("Flow");
for (int i = 0; i < arr.length(); i++) //iterate over all values
    Object value = arr.get(i); //get value
    System.out.println(value); //print each value
}

```

Section 107.7: optXXX vs getXXX methods

JSONObject and JSONArray have a few methods that are very useful while dealing with a possibility that a value you are trying to get does not exist or is of another type.

```

JSONObject obj = new JSONObject();
obj.putString("foo", "bar");

// For existing properties of the correct type, there is no difference
obj.getString("foo"); // returns "bar"
obj.optString("foo"); // returns "bar"
obj.optString("foo", "tux"); // returns "bar"

// However, if a value cannot be coerced to the required type, the behavior differs
obj.getInt("foo"); // throws JSONException
obj.optInt("foo"); // returns 0
obj.optInt("foo", 123); // returns 123

// Same if a property does not exist
obj.getString("undefined"); // throws JSONException
obj.optString("undefined"); // returns ""
obj.optString("undefined", "tux"); // returns "tux"

```

The same rules apply to the getXXX / optXXX methods of JSONArray.

Section 107.8: Extract single element from JSON

```

String json = "{\"name\": \"John\", \"age\":21}";

JsonObject jsonObject = new JsonParser().parse(json).getAsJsonObject();

System.out.println(jsonObject.get("name").getAsString()); //John
System.out.println(jsonObject.get("age").getAsInt()); //21

```

Section 107.9: JSONArray to Java List (Gson Library)

Here is a simple JSONArray which you would like to convert to a Java [ArrayList](#):

```
{
    "list": [
        "Test_String_1",
        "Test_String_2"
    ]
}
```

Now pass the JSONArray 'list' to the following method which returns a corresponding Java [ArrayList](#):

```

public ArrayList<String> getListString(String jsonList){
    Type listType = new TypeToken<List<String>>() {}.getType();
    //make sure the name 'list' matches the name of 'JSONArray' in your 'Json'.
    ArrayList<String> list = new Gson().fromJson(jsonList, listType);
}

```

```
    return list;
}
```

你应该将以下 maven 依赖添加到你的 POM.xml 文件中：

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.7</version>
</dependency>
```

或者你应该在你的类路径中包含 jar com.google.code.gson:gson:jar:<version>。

第 107.10 节：将数据编码为 JSON

如果你需要创建一个JSONObject并向其中放入数据，请参考以下示例：

```
// 创建一个新的 javax.json.JSONObject 实例。
JSONObject first = new JSONObject();

first.put("foo", "bar");
first.put("temperature", 21.5);
first.put("year", 2016);

// 添加第二个对象。
JSONObject second = new JSONObject();
second.put("Hello", "world");
first.put("message", second);

// 创建一个包含一些值的新 JSONArray
JSONArray someMonths = new JSONArray(new String[] { "January", "February" });
someMonths.put("March");
// 作为第五个元素添加另一个月份，第四个元素保持未设置状态。
someMonths.put(4, "May");

// 将数组添加到我们的对象中
object.put("months", someMonths);

// 编码
String json = object.toString();

// 给读者的练习：添加美化打印功能！
/* {
    "foo": "bar",
    "temperature": 21.5,
    "year": 2016,
    "message": {"Hello": "world"},
    "months": ["January", "February", "March", null, "May"]
}
```

```
    return list;
}
```

You should add the following maven dependency to your POM.xml file:

```
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.7</version>
</dependency>
```

Or you should have the jar com.google.code.gson:gson:jar:<version> in your classpath.

Section 107.10: Encoding data as JSON

If you need to create a JSONObject and put data in it, consider the following example:

```
// Create a new javax.json.JSONObject instance.
JSONObject first = new JSONObject();

first.put("foo", "bar");
first.put("temperature", 21.5);
first.put("year", 2016);

// Add a second object.
JSONObject second = new JSONObject();
second.put("Hello", "world");
first.put("message", second);

// Create a new JSONArray with some values
JSONArray someMonths = new JSONArray(new String[] { "January", "February" });
someMonths.put("March");
// Add another month as the fifth element, leaving the 4th element unset.
someMonths.put(4, "May");

// Add the array to our object
object.put("months", someMonths);

// Encode
String json = object.toString();

// An exercise for the reader: Add pretty-printing!
/* {
    "foo": "bar",
    "temperature": 21.5,
    "year": 2016,
    "message": {"Hello": "world"},
    "months": ["January", "February", "March", null, "May"]
}
```

第107.11节：解码JSON数据

如果你需要从JSONObject获取数据，可以参考以下示例：

```
String json =
"{"foo": "bar", "temperature": 21.5, "year": 2016, "message": {"Hello": "world"}, "months": [
    "January", "February", "March", null, "May"]}"
```

Section 107.11: Decoding JSON data

If you need to get data from a JSONObject, consider the following example:

```
String json =
"{"foo": "bar", "temperature": 21.5, "year": 2016, "message": {"Hello": "world"}, "months": [
    "January", "February", "March", null, "May"]}"
```

```
// 解码 JSON 编码的字符串
JSONObject object = new JSONObject(json);

// 获取一些值
String foo = object.getString("foo");
double temperature = object.getDouble("temperature");
int year = object.getInt("year");

// 获取另一个对象
JSONObject secondary = object.getJSONObject("message");
String world = secondary.getString("Hello");

// 获取一个数组
JSONArray someMonths = object.getJSONArray("months");
// 从数组中获取一些值
int nMonths = someMonths.length();
String february = someMonths.getString(1);
```

```
// Decode the JSON-encoded string
JSONObject object = new JSONObject(json);

// Retrieve some values
String foo = object.getString("foo");
double temperature = object.getDouble("temperature");
int year = object.getInt("year");

// Retrieve another object
JSONObject secondary = object.getJSONObject("message");
String world = secondary.getString("Hello");

// Retrieve an array
JSONArray someMonths = object.getJSONArray("months");
// Get some values from the array
int nMonths = someMonths.length();
String february = someMonths.getString(1);
```

第108章：使用JAXP API进行XML解析

第108.1节：使用StAX API解析文档

考虑以下文档：

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
  <book id='1'>Effective Java</book>
  <book id='2'>Java Concurrency In Practice</book>
  <notABook id='3'>This is not a book element</notABook>
</library>
```

可以使用以下代码解析它，并根据书籍ID构建书名映射。

```
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;

public class StaxDemo {

    public static void main(String[] args) throws Exception {
        String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>" +
            + "<library>" +
            + "<book id='1'>Effective Java</book>" +
            + "<book id='2'>Java Concurrency In Practice</book>" +
            + "<notABook id='3'>This is not a book element </notABook>" +
            + "</library>";

        XMLInputFactory xmlInputFactory = XMLInputFactory.newInstance();
        // 各种方式都是可能的，例如来自 InputStream、Source 等
        XMLStreamReader xmlStreamReader = xmlInputFactory.createXMLStreamReader(new
StringReader(xmlDocument));

        Map<Integer, String> bookTitlesById = new HashMap<>();

        // 我们通过循环遍历每个事件
        while (xmlStreamReader.hasNext()) {
            switch (xmlStreamReader.getEventType()) {
                case XMLStreamConstants.START_ELEMENT:
                    System.out.println("发现元素开始: " + xmlStreamReader.getLocalName());
                    // 检查是否处于 <book> 元素的开始位置
                    if ("book".equals(xmlStreamReader.getLocalName())) {
                        int bookId = Integer.parseInt(xmlStreamReader.getAttributeValue("", "id"));
                        String bookTitle = xmlStreamReader.getElementText();
                        bookTitlesById.put(bookId, bookTitle);
                    }
                    break;
                // 还有许多其他可能的情况：注释、处理指令、
                // 空白字符...
                default:
                    break;
            }
            xmlStreamReader.next();
        }
    }
}
```

Chapter 108: XML Parsing using the JAXP APIs

Section 108.1: Parsing a document using the StAX API

Considering the following document :

```
<?xml version='1.0' encoding='UTF-8' ?>
<library>
  <book id='1'>Effective Java</book>
  <book id='2'>Java Concurrency In Practice</book>
  <notABook id='3'>This is not a book element</notABook>
</library>
```

One can use the following code to parse it and build a map of book titles by book id.

```
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;

public class StaxDemo {

    public static void main(String[] args) throws Exception {
        String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>" +
            + "<library>" +
            + "<book id='1'>Effective Java</book>" +
            + "<book id='2'>Java Concurrency In Practice</book>" +
            + "<notABook id='3'>This is not a book element </notABook>" +
            + "</library>";

        XMLInputFactory xmlInputFactory = XMLInputFactory.newInstance();
        // Various flavors are possible, e.g. from an InputStream, a Source, ...
        XMLStreamReader xmlStreamReader = xmlInputFactory.createXMLStreamReader(new
StringReader(xmlDocument));

        Map<Integer, String> bookTitlesById = new HashMap<>();

        // We go through each event using a loop
        while (xmlStreamReader.hasNext()) {
            switch (xmlStreamReader.getEventType()) {
                case XMLStreamConstants.START_ELEMENT:
                    System.out.println("Found start of element: " + xmlStreamReader.getLocalName());
                    // Check if we are at the start of a <book> element
                    if ("book".equals(xmlStreamReader.getLocalName())) {
                        int bookId = Integer.parseInt(xmlStreamReader.getAttributeValue("", "id"));
                        String bookTitle = xmlStreamReader.getElementText();
                        bookTitlesById.put(bookId, bookTitle);
                    }
                    break;
                // A bunch of other things are possible : comments, processing instructions,
                // Whitespace...
                default:
                    break;
            }
            xmlStreamReader.next();
        }
    }
}
```

```

        }
        System.out.println(bookTitlesById);
    }
}

```

输出结果为：

```

发现元素开始：library
发现元素开始：book
发现元素开始：book
发现元素开始：notABook
{1=Effective Java, 2=Java Concurrency In Practice}

```

在此示例中，需要注意以下几点：

1. 使用xmlStreamReader.getAttributeValue是可行的，因为我们首先检查了解析器处于 START_ELEMENT状态。在其他所有状态（除ATTRIBUTES外），解析器必须抛出 `IllegalStateException`，因为属性只能出现在元素的开始处。
2. 同样适用于xmlStreamReader.getTextContent()，它之所以有效，是因为我们处于START_ELEMENT状态，且知道在此文档中，`<book>`元素没有非文本子节点。

对于更复杂的文档解析（更深层次的嵌套元素等），将解析器“委托”给子方法或其他对象是一种良好做法，例如创建一个BookParser类或方法，让它处理从书籍XML标签的START_ELEMENT到END_ELEMENT的所有元素。

也可以使用Stack对象在树的上下部分保存重要数据。

第108.2节：使用DOM API解析和导航文档

考虑以下文档：

```

<?xml version='1.0' encoding='UTF-8' ?>
<library>
    <book id='1'>Effective Java</book>
    <book id='2'>Java Concurrency In Practice</book>
</library>

```

可以使用以下代码从一个String构建DOM树：

```

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.StringReader;

public class DOMDemo {

    public static void main(String[] args) throws Exception {
        String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>" +
            + "<library>" +
            + "<book id='1'>Effective Java</book>"
```

```

        }
        System.out.println(bookTitlesById);
    }
}

```

This outputs :

```

Found start of element: library
Found start of element: book
Found start of element: book
Found start of element: notABook
{1=Effective Java, 2=Java Concurrency In Practice}

```

In this sample, one must be careful of a few things :

1. The use of `xmlStreamReader.getAttributeValue` works because we have checked first that the parser is in the START_ELEMENT state. In every other states (except ATTRIBUTES), the parser is mandated to throw `IllegalStateException`, because attributes can only appear at the beginning of elements.
2. same goes for `xmlStreamReader.getTextContent()`, it works because we are at a START_ELEMENT and we know in this document that the `<book>` element has no non-text child nodes.

For more complex documents parsing (deeper, nested elements, ...), it is a good practice to "delegate" the parser to sub-methods or other objects, e.g. have a BookParser class or method, and have it deal with every element from the START_ELEMENT to the END_ELEMENT of the book XML tag.

One can also use a `Stack` object to keep around important data up and down the tree.

Section 108.2: Parsing and navigating a document using the DOM API

Considering the following document :

```

<?xml version='1.0' encoding='UTF-8' ?>
<library>
    <book id='1'>Effective Java</book>
    <book id='2'>Java Concurrency In Practice</book>
</library>

```

One can use the following code to build a DOM tree out of a `String` :

```

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.StringReader;

public class DOMDemo {

    public static void main(String[] args) throws Exception {
        String xmlDocument = "<?xml version='1.0' encoding='UTF-8' ?>" +
            + "<library>" +
            + "<book id='1'>Effective Java</book>"
```

```

+ "<book id='2'>Java Concurrency In Practice</book>"  

+ "</library>";  
  

DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();  

// 这里没有用处, 因为XML没有命名空间, 但这个选项在某些情况下是有用的  
  

documentBuilderFactory.setNamespaceAware(true);  

DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();  

// 这里有多种选项, 可以从InputStream、文件等读取...
Document document = documentBuilder.parse(new InputSource(new StringReader(xmlDocument)));  
  

// 文档的根节点  

System.out.println("XML文档的根节点是: " +  

document.getDocumentElement().getLocalName());  
  

// 遍历内容  

NodeList firstLevelChildren = document.getDocumentElement().getChildNodes();  

for (int i = 0; i < firstLevelChildren.getLength(); i++) {  

    Node item = firstLevelChildren.item(i);  

    System.out.println("找到一级子节点, XML标签名是: " + item.getLocalName());  

    System.out.println("该标签的id属性是 : " +  

item.getAttributes().getNamedItem("id").getTextContent());  

}  
  

// 另一种方式是  

NodeList allBooks = document.getDocumentElement().getElementsByTagName("book");  

}
}

```

代码产生以下结果：

```

XML文档的根节点： library
发现一级子节点, XML标签名为： book
该标签的id属性为：1
发现一级子节点, XML标签名为： book
该标签的id属性为：2

```

```

+ "<book id='2'>Java Concurrency In Practice</book>"  

+ "</library>";  
  

DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();  

// This is useless here, because the XML does not have namespaces, but this option is useful to  

know in case  

documentBuilderFactory.setNamespaceAware(true);  

DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder();  

// There are various options here, to read from an InputStream, from a file, ...
Document document = documentBuilder.parse(new InputSource(new StringReader(xmlDocument)));  
  

// Root of the document  

System.out.println("Root of the XML Document: " +  

document.getDocumentElement().getLocalName());  
  

// Iterate the contents  

NodeList firstLevelChildren = document.getDocumentElement().getChildNodes();  

for (int i = 0; i < firstLevelChildren.getLength(); i++) {  

    Node item = firstLevelChildren.item(i);  

    System.out.println("First level child found, XML tag name is: " + item.getLocalName());  

    System.out.println("\tid attribute of this tag is : " +  

item.getAttributes().getNamedItem("id").getTextContent());  

}  
  

// Another way would have been
NodeList allBooks = document.getDocumentElement().getElementsByTagName("book");
}
}

```

The code yields the following :

```

Root of the XML Document: library
First level child found, XML tag name is: book
id attribute of this tag is : 1
First level child found, XML tag name is: book
id attribute of this tag is : 2

```

第109章：XML XPath评估

第109.1节：解析单个

使用与在XML文档中评估NodeList相同的示例，以下是如何高效地进行多个XPath调用：

给定以下XML文档：

```
<documentation>
  <tags>
    <tag name="Java">
      <topic name="正则表达式">
        <example>匹配分组</example>
        <example>转义元字符</example>
      </topic>
      <topic name="数组">
        <example>遍历数组</example>
        <example>将数组转换为列表</example>
      </topic>
    </tag>
    <tag name="安卓">
      <topic name="构建安卓项目">
        <example>使用Gradle构建安卓应用</example>
        <example>使用Maven构建安卓应用</example>
      </topic>
      <topic name="布局资源">
        <example>包含布局资源</example>
        <example>支持多种设备屏幕</example>
      </topic>
    </tag>
  </tags></tags>
```

这就是你如何使用XPath在一个文档中评估多个表达式的方法：

```
XPath xPath = XPathFactory.newInstance().newXPath(); //创建新的 XPath
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //指定XML文件路径

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", doc, XPathConstants.NODESET);
//评估XPath
xPath.reset(); // 重置 xPath 以便再次使用
NodeList androidExampleNodes = (NodeList)
  xPath.evaluate("/documentation/tags/tag[@name='Android']//example", doc, XPathConstants.NODESET);
//评估XPath

...
```

第109.2节：在XML中多次解析单个XPath表达式

在这种情况下，您希望在评估之前先编译表达式，这样每次调用 evaluate 时就不会编译相同的表达式。简单的语法如下：

Chapter 109: XML XPath Evaluation

Section 109.1: Parsing multiple XPath Expressions in a single XML

Using the same example as [Evaluating a NodeList in an XML document](#), here is how you would make multiple XPath calls efficiently:

Given the following XML document:

```
<documentation>
  <tags>
    <tag name="Java">
      <topic name="Regular expressions">
        <example>Matching groups</example>
        <example>Escaping metacharacters</example>
      </topic>
      <topic name="Arrays">
        <example>Looping over arrays</example>
        <example>Converting an array to a list</example>
      </topic>
    </tag>
    <tag name="Android">
      <topic name="Building Android projects">
        <example>Building an Android application using Gradle</example>
        <example>Building an Android application using Maven</example>
      </topic>
      <topic name="Layout resources">
        <example>Including layout resources</example>
        <example>Supporting multiple device screens</example>
      </topic>
    </tag>
  </tags>
</documentation>
```

This is how you would use XPath to evaluate multiple expressions in one document:

```
XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path

NodeList javaExampleNodes = (NodeList)
xPath.evaluate("/documentation/tags/tag[@name='Java']//example", doc, XPathConstants.NODESET);
//Evaluate the XPath
xPath.reset(); //Resets the xPath so it can be used again
NodeList androidExampleNodes = (NodeList)
  xPath.evaluate("/documentation/tags/tag[@name='Android']//example", doc, XPathConstants.NODESET);
//Evaluate the XPath

...
```

Section 109.2: Parsing single XPath Expression multiple times in an XML

In this case, you want to have the expression compiled before the evaluations, so that each call to evaluate does not compile the same expression. The simple syntax would be:

```

XPath xPath = XPathFactory.newInstance().newXPath(); //创建新的XPath
XPathExpression exp = xPath.compile("/documentation/tags/tag[@name='Java']//example");
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //指定XML文件路径
NodeList javaExampleNodes = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //从已编译的表达式中评估XPath
NodeList javaExampleNodes2 = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //再次执行

```

总体来说，两次调用 `XPathExpression.evaluate()` 比两次调用 `XPath.evaluate()` 效率要高得多。

第109.3节：在XML文档中评估NodeList

给定以下XML文档：

```

<documentation>
  <tags>
    <tag name="Java">
      <topic name="正则表达式">
        <example>匹配分组</example>
        <example>转义元字符</example>
      </topic>
      <topic name="数组">
        <example>遍历数组</example>
        <example>将数组转换为列表</example>
      </topic>
    </tag>
    <tag name="安卓">
      <topic name="构建安卓项目">
        <example>使用Gradle构建安卓应用</example>
        <example>使用Maven构建安卓应用</example>
      </topic>
      <topic name="布局资源">
        <example>包含布局资源</example>
        <example>支持多种设备屏幕</example>
      </topic>
    </tag>
  </tags></tags>

```

下面的代码检索Java标签的所有 `example` 节点（如果只在XML中评估一次XPath，请使用此方法。有关在同一XML文件中多次评估XPath调用的其他示例，请参见其他示例：）

```

XPathFactory xPathFactory = XPathFactory.newInstance();
XPath xPath = xPathFactory.newXPath(); //创建新的 XPath
InputSource inputSource = new InputSource("path/to/xml.xml"); //指定 XML 文件路径
NodeList javaExampleNodes = (NodeList) xPath.evaluate("/documentation/tags/tag[@name='Java']//example", inputSource, XPathConstants.NODESET); // 评估 XPath
...

```

```

XPath xPath = XPathFactory.newInstance().newXPath(); //Make new XPath
XPathExpression exp = xPath.compile("/documentation/tags/tag[@name='Java']//example");
DocumentBuilder builder = DocumentBuilderFactory.newInstance();
Document doc = builder.parse(new File("path/to/xml.xml")); //Specify XML file path
NodeList javaExampleNodes = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Evaluate the XPath from the already-compiled expression
NodeList javaExampleNodes2 = (NodeList) exp.evaluate(doc, XPathConstants.NODESET); //Do it again

```

Overall, two calls to `XPathExpression.evaluate()` will be much more efficient than two calls to `XPath.evaluate()`.

Section 109.3: Evaluating a NodeList in an XML document

Given the following XML document:

```

<documentation>
  <tags>
    <tag name="Java">
      <topic name="Regular expressions">
        <example>Matching groups</example>
        <example>Escaping metacharacters</example>
      </topic>
      <topic name="Arrays">
        <example>Looping over arrays</example>
        <example>Converting an array to a list</example>
      </topic>
    </tag>
    <tag name="Android">
      <topic name="Building Android projects">
        <example>Building an Android application using Gradle</example>
        <example>Building an Android application using Maven</example>
      </topic>
      <topic name="Layout resources">
        <example>Including layout resources</example>
        <example>Supporting multiple device screens</example>
      </topic>
    </tag>
  </tags>
</documentation>

```

The following retrieves all `example` nodes for the Java tag (Use this method if only evaluating XPath in the XML once. See other example for when multiple XPath calls are evaluated in the same XML file.):

```

XPathFactory xPathFactory = XPathFactory.newInstance();
XPath xPath = xPathFactory.newXPath(); //Make new XPath
InputSource inputSource = new InputSource("path/to/xml.xml"); //Specify XML file path
NodeList javaExampleNodes = (NodeList) xPath.evaluate("/documentation/tags/tag[@name='Java']//example", inputSource, XPathConstants.NODESET); //Evaluate the XPath
...

```

第110章：XOM - XML对象模型

第110.1节：读取XML文件

为了使用XOM加载XML数据，你需要创建一个Builder，从中可以构建成一个Document。

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

要获取根元素，即XML文件中最高层的父元素，需要在Document

实例上使用getRootElement()方法。

```
Element root = doc.getRootElement();
```

现在，Element类有许多方便的方法，使读取XML变得非常简单。以下列出了一些最有用的方法：

- `getChildElements(String name)` - 返回一个Elements实例，该实例作为元素数组使用。
- `getFirstChildElement(String name)` - 返回具有该标签的第一个子元素。
- `getValue()` - 返回元素内部的值。
- `getAttributeValue(String name)` - 返回指定名称属性的值。

当你调用`getChildElements()`时，你会得到一个Elements实例。通过它可以循环调用`get(int index)`方法来获取所有内部元素。

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

示例：下面是读取XML文件的示例：

XML文件：

```
1 <example>
2   <person>
3     <name>
4       <first>Dan</first>
5       <last>Smith</last>
6     </name>
7     <age unit="years">23</age>
8     <fav_color>green</fav_color>
9   </person>
10  <person>
11    <name>
12      <first>Bob</first>
13      <last>Autry</last>
14    </name>
15    <age unit="months">3</age>
16    <fav_color>N/A</fav_color>
17  </person>
18 </example>
```

读取并打印的代码：

Chapter 110: XOM - XML Object Model

Section 110.1: Reading a XML file

In order to load the XML data with [XOM](#) you will need to make a Builder from which you can build it into a Document.

```
Builder builder = new Builder();
Document doc = builder.build(file);
```

To get the root element, the highest parent in the xml file, you need to use the `getRootElement()` on the [Document](#) instance.

```
Element root = doc.getRootElement();
```

Now the Element class has a lot of handy methods that make reading xml really easy. Some of the most useful are listed below:

- `getChildElements(String name)` - returns an Elements instance that acts as an array of elements
- `getFirstChildElement(String name)` - returns the first child element with that tag.
- `getValue()` - returns the value inside the element.
- `getAttributeValue(String name)` - returns the value of an attribute with the specified name.

When you call the `getChildElements()` you get a Elements instance. From this you can loop through and call the `get(int index)` method on it to retrieve all the elements inside.

```
Elements colors = root.getChildElements("color");
for (int q = 0; q < colors.size(); q++){
    Element color = colors.get(q);
}
```

Example: Here is an example of reading an XML File:

XML File:

```
1 <example>
2   <person>
3     <name>
4       <first>Dan</first>
5       <last>Smith</last>
6     </name>
7     <age unit="years">23</age>
8     <fav_color>green</fav_color>
9   </person>
10  <person>
11    <name>
12      <first>Bob</first>
13      <last>Autry</last>
14    </name>
15    <age unit="months">3</age>
16    <fav_color>N/A</fav_color>
17  </person>
18 </example>
```

Code for reading and printing it:

```

import java.io.File;
import java.io.IOException;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;

public class XMLReader {

    public static void main(String[] args) throws ParsingException, IOException{
        File file = new File("insert path here");
        // 构建器构建xml数据
        Builder builder = new Builder();
        Document doc = builder.build(file);

        // 获取根元素 <example>
        Element root = doc.getRootElement();

        // 获取所有标签为 <person> 的元素
        Elements people = root.getChildElements("person");

        for (int q = 0; q < people.size(); q++){
            // 获取当前的person元素
            Element person = people.get(q);

            // 获取name元素及其子元素：first和last
            Element nameElement = person.getFirstChildElement("name");
            Element firstNameElement = nameElement.getFirstChildElement("first");
            Element lastNameElement = nameElement.getFirstChildElement("last");

            // 获取age元素
            Element ageElement = person.getFirstChildElement("age");

            // 获取最喜欢的颜色元素
            Element favColorElement = person.getFirstChildElement("fav_color");

            String fName, lName, ageUnit, favColor;
            int age;

            try {
                fName = firstNameElement.getValue();
                lName = lastNameElement.getValue();
                age = Integer.parseInt(ageElement.getValue());
                ageUnit = ageElement.getAttributeValue("unit");
                favColor = favColorElement.getValue();

                System.out.println("姓名: " + lName + ", " + fName);
                System.out.println("年龄: " + age + " (" + ageUnit + ")");
                System.out.println("最喜欢的颜色: " + favColor);
                System.out.println("-----");
            } catch (NullPointerException ex){
                ex.printStackTrace();
            } catch (NumberFormatException ex){
                ex.printStackTrace();
            }
        }
    }
}

```

```

import java.io.File;
import java.io.IOException;
import nu.xom.Builder;
import nu.xom.Document;
import nu.xom.Element;
import nu.xom.Elements;
import nu.xom.ParsingException;

public class XMLReader {

    public static void main(String[] args) throws ParsingException, IOException{
        File file = new File("insert path here");
        // builder builds xml data
        Builder builder = new Builder();
        Document doc = builder.build(file);

        // get the root element <example>
        Element root = doc.getRootElement();

        // gets all element with tag <person>
        Elements people = root.getChildElements("person");

        for (int q = 0; q < people.size(); q++){
            // get the current person element
            Element person = people.get(q);

            // get the name element and its children: first and last
            Element nameElement = person.getFirstChildElement("name");
            Element firstNameElement = nameElement.getFirstChildElement("first");
            Element lastNameElement = nameElement.getFirstChildElement("last");

            // get the age element
            Element ageElement = person.getFirstChildElement("age");

            // get the favorite color element
            Element favColorElement = person.getFirstChildElement("fav_color");

            String fName, lName, ageUnit, favColor;
            int age;

            try {
                fName = firstNameElement.getValue();
                lName = lastNameElement.getValue();
                age = Integer.parseInt(ageElement.getValue());
                ageUnit = ageElement.getAttributeValue("unit");
                favColor = favColorElement.getValue();

                System.out.println("Name: " + lName + ", " + fName);
                System.out.println("Age: " + age + " (" + ageUnit + ")");
                System.out.println("Favorite Color: " + favColor);
                System.out.println("-----");
            } catch (NullPointerException ex){
                ex.printStackTrace();
            } catch (NumberFormatException ex){
                ex.printStackTrace();
            }
        }
    }
}

```

这将在控制台打印出：

```
姓名: 史密斯, 丹  
年龄: 23 (岁)  
喜欢的颜色: 绿色
```

```
姓名: 奥特里, 鲍勃  
年龄: 3 (月)  
喜欢的颜色: 无/无
```

第110.2节：写入XML文件

使用XOM写入XML文件与读取非常相似，只不过在这种情况下我们是创建实例，而不是从根节点获取它们。

要创建一个新的元素，请使用构造函数Element(String name)。你需要创建一个根元素，以便可以轻松地将其添加到Document中。

```
Element root = new Element("root");
```

Element类有一些方便的编辑元素的方法。它们列举如下：

- appendChild(String name) - 这基本上会将元素的值设置为name。
- appendChild(Node node) - 这会使node成为元素的父节点。（元素是节点，因此你可以解析元素）。
- addAttribute(Attribute attribute) - 会向元素添加一个属性。

Attribute类有几种不同的构造函数。最简单的是Attribute(String name, String value)。

一旦你将所有元素添加到根元素后，就可以将其转换为Document。构造函数中Document会接受一个Element作为参数。

你可以使用Serializer将你的XML写入文件。你需要创建一个新的输出流，并在Serializer的构造函数中传入该流。

```
FileOutputStream fileOutputStream = new FileOutputStream(file);  
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");  
serializer.setIndent(4);  
serializer.write(doc);
```

示例

代码：

```
import java.io.File;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.UnsupportedEncodingException;  
import nu.xom.Attribute;  
import nu.xom.Builder;  
import nu.xom.Document;  
import nu.xom.Element;  
import nu.xom.Elements;  
import nu.xom.ParsingException;  
import nu.xom.Serializer;
```

This will print out in the console:

```
Name: Smith, Dan  
Age: 23 (years)  
Favorite Color: green  
-----  
Name: Autry, Bob  
Age: 3 (months)  
Favorite Color: N/A
```

Section 110.2: Writing to a XML File

Writing to a XML File using [XOM](#) is very similar to reading it except in this case we are making the instances instead of retrieving them off the root.

To make a new Element use the constructor `Element(String name)`. You will want to make a root element so that you can easily add it to a `Document`.

```
Element root = new Element("root");
```

The `Element` class has some handy methods for editing elements. They are listed below:

- `appendChild(String name)` - this will basically set the value of the element to name.
- `appendChild(Node node)` - this will make node the elements parent. (Elements are nodes so you can parse elements).
- `addAttribute(Attribute attribute)` - will add an attribute to the element.

The `Attribute` class has a couple of different constructors. The simplest one is `Attribute(String name, String value)`.

Once you have all of your elements add to your root element you can turn it into a `Document`. `Document` will take a `Element` as an argument in its constructor.

You can use a `Serializer` to write your XML to a file. You will need to make a new output stream to parse in the constructor of `Serializer`.

```
FileOutputStream fileOutputStream = new FileOutputStream(file);  
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");  
serializer.setIndent(4);  
serializer.write(doc);
```

Example

Code:

```
import java.io.File;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.UnsupportedEncodingException;  
import nu.xom.Attribute;  
import nu.xom.Builder;  
import nu.xom.Document;  
import nu.xom.Element;  
import nu.xom.Elements;  
import nu.xom.ParsingException;  
import nu.xom.Serializer;
```

```

public class XMLWriter{

    public static void main(String[] args) throws UnsupportedEncodingException,
        IOException{
        // 根元素 <example>
        Element root = new Element("example");

        // 创建一个用于存储人员的数组
        Person[] people = {new Person("史密斯", "丹", "年", "绿色", 23),
            new Person("奥特里", "鲍勃", "月", "N/A", 3)};

        // 添加所有人
        for (Person person : people){

            // 创建主person元素 <person>
            Element personElement = new Element("person");

            // 创建name元素及其子元素：first和last
            Element nameElement = new Element("name");
            Element firstNameElement = new Element("first");
            Element lastNameElement = new Element("last");

            // 创建age元素
            Element ageElement = new Element("age");

            // 创建喜欢的颜色元素
            Element favColorElement = new Element("fav_color");

            // 添加值到名字
            firstNameElement.appendChild(person.getFirstName());
            lastNameElement.appendChild(person.getLastName());

            // 将名字添加到name元素
            nameElement.appendChild(firstNameElement);
            nameElement.appendChild(lastNameElement);

            // 添加值到年龄
            ageElement.appendChild(String.valueOf(person.getAge()));

            // 给年龄添加单位属性
            ageElement.addAttribute(new Attribute("unit", person.getAgeUnit()));

            // 给 favColor 添加值
            favColorElement.appendChild(person.getFavoriteColor());

            // 将所有内容添加到 person
            personElement.appendChild(nameElement);
            personElement.appendChild(ageElement);
            personElement.appendChild(favColorElement);

            // 将 person 添加到根节点
            root.appendChild(personElement);
        }

        // 基于根节点创建文档
        Document doc = new Document(root);

        // 文件将被存储的位置
        File file = new File("out.xml");
        if (!file.exists()){
            file.createNewFile();
        }
    }
}

```

```

public class XMLWriter{

    public static void main(String[] args) throws UnsupportedEncodingException,
        IOException{
        // root element <example>
        Element root = new Element("example");

        // make a array of people to store
        Person[] people = {new Person("Smith", "Dan", "years", "green", 23),
            new Person("Autry", "Bob", "months", "N/A", 3)};

        // add all the people
        for (Person person : people){

            // make the main person element <person>
            Element personElement = new Element("person");

            // make the name element and its children: first and last
            Element nameElement = new Element("name");
            Element firstNameElement = new Element("first");
            Element lastNameElement = new Element("last");

            // make age element
            Element ageElement = new Element("age");

            // make favorite color element
            Element favColorElement = new Element("fav_color");

            // add value to names
            firstNameElement.appendChild(person.getFirstName());
            lastNameElement.appendChild(person.getLastName());

            // add names to name
            nameElement.appendChild(firstNameElement);
            nameElement.appendChild(lastNameElement);

            // add value to age
            ageElement.appendChild(String.valueOf(person.getAge()));

            // add unit attribute to age
            ageElement.addAttribute(new Attribute("unit", person.getAgeUnit()));

            // add value to favColor
            favColorElement.appendChild(person.getFavoriteColor());

            // add all contents to person
            personElement.appendChild(nameElement);
            personElement.appendChild(ageElement);
            personElement.appendChild(favColorElement);

            // add person to root
            root.appendChild(personElement);
        }

        // create doc off of root
        Document doc = new Document(root);

        // the file it will be stored in
        File file = new File("out.xml");
        if (!file.exists()){
            file.createNewFile();
        }
    }
}

```

```

// 准备一个文件输出流
FileOutputStream fileOutputStream = new FileOutputStream(file);

// 使用序列化器类写入所有内容
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
}

private static class Person {

    private String lName, fName, ageUnit, favColor;
    private int age;

    public Person(String lName, String fName, String ageUnit, String favColor, int age) {
        this.lName = lName;
        this.fName = fName;
        this.age = age;
        this.ageUnit = ageUnit;
        this.favColor = favColor;
    }

    public String getLastName() { return lName; }
    public String getFirstName() { return fName; }
    public String getAgeUnit() { return ageUnit; }
    public String getFavoriteColor() { return favColor; }
    public int getAge() { return age; }
}
}

```

这将是“out.xml”的内容：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <example>
3   <person>
4     <name>
5       <first>Dan</first>
6       <last>Smith</last>
7     </name>
8     <age unit="years">23</age>
9     <fav_color>green</fav_color>
10    </person>
11    <person>
12      <name>
13        <first>Bob</first>
14        <last>Aultry</last>
15      </name>
16      <age unit="months">3</age>
17      <fav_color>N/A</fav_color>
18    </person>
19  </example>
20

```

```

// get a file output stream ready
FileOutputStream fileOutputStream = new FileOutputStream(file);

// use the serializer class to write it all
Serializer serializer = new Serializer(fileOutputStream, "UTF-8");
serializer.setIndent(4);
serializer.write(doc);
}

private static class Person {

    private String lName, fName, ageUnit, favColor;
    private int age;

    public Person(String lName, String fName, String ageUnit, String favColor, int age) {
        this.lName = lName;
        this.fName = fName;
        this.age = age;
        this.ageUnit = ageUnit;
        this.favColor = favColor;
    }

    public String getLastName() { return lName; }
    public String getFirstName() { return fName; }
    public String getAgeUnit() { return ageUnit; }
    public String getFavoriteColor() { return favColor; }
    public int getAge() { return age; }
}
}

```

This will be the contents of "out.xml":

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <example>
3   <person>
4     <name>
5       <first>Dan</first>
6       <last>Smith</last>
7     </name>
8     <age unit="years">23</age>
9     <fav_color>green</fav_color>
10    </person>
11    <person>
12      <name>
13        <first>Bob</first>
14        <last>Aultry</last>
15      </name>
16      <age unit="months">3</age>
17      <fav_color>N/A</fav_color>
18    </person>
19  </example>
20

```

第111章：多态性

多态性是面向对象编程（OOP）中的主要概念之一。多态性一词来源于希腊语“poly”和“morphs”。Poly意为“多”，morphs意为“形式”（多种形式）。

实现多态性有两种方式。方法重载和方法重写。

第111.1节：方法重写

方法重写是子类型重新定义（重写）其超类型行为的能力。

在Java中，这意味着子类重写超类中定义的方法。在Java中，所有非原始类型变量实际上都是引用，类似于指向内存中实际对象位置的指针。引用只有一种类型，即声明时的类型。然而，它们可以指向声明类型或其任何子类型的对象。

当在引用上调用方法时，实际被指向对象的对应方法被调用。

```
class SuperType {  
    public void sayHello(){  
        System.out.println("Hello from SuperType");  
    }  
  
    public void sayBye(){  
        System.out.println("Bye from SuperType");  
    }  
}  
  
class SubType extends SuperType {  
    // override the superclass method  
    public void sayHello(){  
        System.out.println("Hello from SubType");  
    }  
}  
  
class Test {  
    public static void main(String... args){  
        SuperType superType = new SuperType();  
        superType.sayHello(); // -> Hello from SuperType  
  
        // make the reference point to an object of the subclass  
        superType = new SubType();  
        // 行为由对象控制，而非由引用控制  
        superType.sayHello(); // -> 来自子类的问候  
  
        // 未重写的方法将被直接继承  
        superType.sayBye(); // -> 来自超类的再见  
    }  
}
```

需要牢记的规则

要在子类中重写方法，重写的方法（即子类中的方法）必须具备：

- 相同的方法名
- 对于基本类型相同的返回类型（对于类允许使用子类，这也称为协变返回类型）

Chapter 111: Polymorphism

Polymorphism is one of main OOP(object oriented programming) concepts. Polymorphism word was derived from the greek words "poly" and "morphs". Poly means "many" and morphs means "forms" (many forms).

There are two ways to perform polymorphism. **Method Overloading** and **Method Overriding**.

Section 111.1: Method Overriding

Method overriding is the ability of subtypes to redefine (override) the behavior of their supertypes.

In Java, this translates to subclasses overriding the methods defined in the super class. In Java, all non-primitive variables are actually references, which are akin to pointers to the location of the actual object in memory. The references only have one type, which is the type they were declared with. However, they can point to an object of either their declared type or any of its subtypes.

When a method is called on a reference, the corresponding **method of the actual object being pointed to is invoked**.

```
class SuperType {  
    public void sayHello(){  
        System.out.println("Hello from SuperType");  
    }  
  
    public void sayBye(){  
        System.out.println("Bye from SuperType");  
    }  
}  
  
class SubType extends SuperType {  
    // override the superclass method  
    public void sayHello(){  
        System.out.println("Hello from SubType");  
    }  
}  
  
class Test {  
    public static void main(String... args){  
        SuperType superType = new SuperType();  
        superType.sayHello(); // -> Hello from SuperType  
  
        // make the reference point to an object of the subclass  
        superType = new SubType();  
        // behaviour is governed by the object, not by the reference  
        superType.sayHello(); // -> Hello from SubType  
  
        // non-overridden method is simply inherited  
        superType.sayBye(); // -> Bye from SuperType  
    }  
}
```

Rules to keep in mind

To override a method in the subclass, the overriding method (i.e. the one in the subclass) **MUST HAVE**:

- same name
- same return type in case of primitives (a subclass is allowed for classes, this is also known as covariant return

类型)。

- 相同类型的参数
- 它可能只抛出在超类方法的throws子句中声明的异常，或者是声明异常的子类异常。它也可以选择不抛出任何异常。
参数类型的名称无关紧要。例如，void methodX(int i) 与 void methodX(int k) 是相同的。
- 我们无法重写final或static方法。我们唯一能做的就是更改方法体。

第111.2节：方法重载

方法重载，也称为函数重载，是指一个类能够拥有多个同名方法，前提是它们在参数的数量或类型上有所不同。

编译器检查方法签名以实现方法重载。

方法签名由三部分组成——

- 1.方法名
- 2.参数数量
- 3.参数类型

如果一个类中有任意两个方法的这三项都相同，则编译器会抛出重复方法错误。

这种多态性称为静态或编译时多态性，因为调用哪个方法由编译器在编译时根据参数列表决定。

```
class Polymorph {  
  
    public int add(int a, int b){  
        return a + b;  
    }  
  
    public int add(int a, int b, int c){  
        return a + b + c;  
    }  
  
    public float add(float a, float b){  
        return a + b;  
    }  
  
    public static void main(String... args){  
        Polymorph poly = new Polymorph();  
        int a = 1, b = 2, c = 3;  
        float d = 1.5, e = 2.5;  
  
        System.out.println(poly.add(a, b));  
        System.out.println(poly.add(a, b, c));  
        System.out.println(poly.add(d, e));  
    }  
}
```

这将导致：

types).

- same type and order of parameters
- it may throw only those exceptions that are declared in the throws clause of the superclass's method or exceptions that are subclasses of the declared exceptions. It may also choose NOT to throw any exception.
- The names of the parameter types do not matter. For example, void methodX(int i) is same as void methodX(int k)
- We are unable to Override final or Static methods. Only thing that we can do change only method body.

Section 111.2: Method Overloading

Method overloading, also known as **function overloading**, is the ability of a class to have multiple methods with the same name, granted that they differ in either number or type of arguments.

Compiler checks **method signature** for method overloading.

Method signature consists of three things -

1. Method name
2. Number of parameters
3. Types of parameters

If these three are same for any two methods in a class, then compiler throws **duplicate method error**.

This type of polymorphism is called *static* or *compile time* polymorphism because the appropriate method to be called is decided by the compiler during the compile time based on the argument list.

```
class Polymorph {  
  
    public int add(int a, int b){  
        return a + b;  
    }  
  
    public int add(int a, int b, int c){  
        return a + b + c;  
    }  
  
    public float add(float a, float b){  
        return a + b;  
    }  
  
    public static void main(String... args){  
        Polymorph poly = new Polymorph();  
        int a = 1, b = 2, c = 3;  
        float d = 1.5, e = 2.5;  
  
        System.out.println(poly.add(a, b));  
        System.out.println(poly.add(a, b, c));  
        System.out.println(poly.add(d, e));  
    }  
}
```

This will result in:

```
2  
6  
4.000000
```

重载方法可以是静态的也可以是非静态的。这也不会影响方法重载。

```
public class Polymorph {  
  
    private static void methodOverloaded()  
    {  
        //无参数，私有静态方法  
    }  
  
    private int methodOverloaded(int i)  
    {  
        //一个参数的私有非静态方法  
        return i;  
    }  
  
    static int methodOverloaded(double d)  
    {  
        //静态方法  
        return 0;  
    }  
  
    public void methodOverloaded(int i, double d)  
    {  
        //公共非静态方法  
    }  
}
```

另外，如果您更改方法的返回类型，我们将无法将其视为方法重载。

```
public class 多态 {  
  
    void 方法重载(){  
        //无参数且无返回类型  
    }  
  
    int 方法重载(){  
        //无参数且返回int类型  
        return 0;  
    }  
}
```

第111.3节：多态及不同类型的重写

来自Java教程

多态的词典定义指的是生物学中的一个原理，即一个生物体或物种可以有多种不同的形态或阶段。这个原理也可以应用于面向对象编程和像Java语言这样的语言。类的子类可以定义它们自己独特的行为，同时共享父类的一些相同功能。

看这个例子以理解不同类型的重写。

1. 基类不提供实现，子类必须重写完整方法 - (抽象)
2. 基类提供默认实现，子类可以改变行为
3. 子类通过调用 super.方法名() 作为第一条语句来扩展基类实现
4. 基类定义算法结构（模板方法），子类将重写算法的一部分

Overloaded methods may be static or non-static. This also does not effect method overloading.

```
public class Polymorph {  
  
    private static void methodOverloaded()  
    {  
        //No argument, private static method  
    }  
  
    private int methodOverloaded(int i)  
    {  
        //One argument private non-static method  
        return i;  
    }  
  
    static int methodOverloaded(double d)  
    {  
        //static Method  
        return 0;  
    }  
  
    public void methodOverloaded(int i, double d)  
    {  
        //Public non-static Method  
    }  
}
```

Also if you change the return type of method, we are unable to get it as method overloading.

```
public class Polymorph {  
  
    void methodOverloaded(){  
        //No argument and No return type  
    }  
  
    int methodOverloaded(){  
        //No argument and int return type  
        return 0;  
    }  
}
```

Section 111.3: Polymorphism and different types of overriding

From java tutorial

The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. **Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.**

Have a look at this example to understand different types of overriding.

1. Base class provides no implementation and sub-class has to override complete method - (abstract)
2. Base class provides default implementation and sub-class can change the behaviour
3. Sub-class adds extension to base class implementation by calling **super .methodName()** as first statement
4. Base class defines structure of the algorithm (Template method) and sub-class will override a part of algorithm

代码片段：

```
import java.util.HashMap;

abstract class Game implements Runnable{

    protected boolean 运行游戏 = true;
    protected Player 玩家1 = null;
    protected Player 玩家2 = null;
    protected Player 当前玩家 = null;

    public Game(){
        玩家1 = new Player("玩家 1");
        玩家2 = new Player("玩家 2");
        当前玩家 = 玩家1;
        初始化游戏();
    }

    /* 类型1：让子类定义自己的实现。基类定义抽象方法以强制
     * 子类定义实现
     */

    protected abstract void 初始化游戏();

    /* 类型2：子类可以改变行为。如果不改变，则适用基类行为 */
    protected void 记录移动间隔时间(Player player){
        System.out.println("基类：移动持续时间：player.PlayerActTime - player.MoveShownTime");
    }

    /* 类型3：基类提供实现。子类可以通过调用来自增强基类的实现
     * 在子类方法的第一行调用 super.methodName(), 具体实现随后进行
     */
    protected void logGameStatistics(){
        System.out.println("基类：logGameStatistics()");
    }

    /* 类型4：模板方法：基类结构不能改变，但子类可以改变部分行为 */
    protected void runGame() throws Exception{
        System.out.println("基类：定义游戏流程：" );
        while (运行游戏) {
            /*
            1. 设置当前玩家
            2. 获取玩家移动
            */
            validatePlayerMove(currentPlayer);
            logTimeBetweenMoves(currentPlayer);
            Thread.sleep(500);
            setNextPlayer();
        }
        logGameStatistics();
    }

    /* 模板方法的子部分，定义子类行为 */
    protected abstract void validatePlayerMove(Player p);

    protected void setRunGame(boolean status){
        this.runGame = status;
    }

    public void setCurrentPlayer(Player p){
        this.currentPlayer = p;
    }
}
```

code snippet:

```
import java.util.HashMap;

abstract class Game implements Runnable{

    protected boolean runGame = true;
    protected Player player1 = null;
    protected Player player2 = null;
    protected Player currentPlayer = null;

    public Game(){
        player1 = new Player("Player 1");
        player2 = new Player("Player 2");
        currentPlayer = player1;
        initializeGame();
    }

    /* Type 1: Let subclass define own implementation. Base class defines abstract method to force
     * sub-classes to define implementation
     */

    protected abstract void initializeGame();

    /* Type 2: Sub-class can change the behaviour. If not, base class behaviour is applicable */
    protected void logTimeBetweenMoves(Player player){
        System.out.println("Base class: Move Duration: player.PlayerActTime - player.MoveShownTime");
    }

    /* Type 3: Base class provides implementation. Sub-class can enhance base class implementation by
     * calling
     *      super.methodName() in first line of the child class method and specific implementation later
     */
    protected void logGameStatistics(){
        System.out.println("Base class: logGameStatistics()");
    }

    /* Type 4: Template method: Structure of base class can't be changed but sub-class can some part
     * of behaviour */
    protected void runGame() throws Exception{
        System.out.println("Base class: Defining the flow for Game:" );
        while (runGame) {
            /*
            1. Set current player
            2. Get Player Move
            */
            validatePlayerMove(currentPlayer);
            logTimeBetweenMoves(currentPlayer);
            Thread.sleep(500);
            setNextPlayer();
        }
        logGameStatistics();
    }

    /* sub-part of the template method, which define child class behaviour */
    protected abstract void validatePlayerMove(Player p);

    protected void setRunGame(boolean status){
        this.runGame = status;
    }

    public void setCurrentPlayer(Player p){
        this.currentPlayer = p;
    }
}
```

```

public void setNextPlayer(){
    if (currentPlayer == player1) {
        currentPlayer = player2;
    } else{
        currentPlayer = player1;
    }
}

public void run(){
    try{
        runGame();
    }catch(Exception err){
        err.printStackTrace();
    }
}

class Player{
    String name;
    Player(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}

/* 具体的游戏实现 */
class 国际象棋 extends 游戏{
    public 国际象棋(){
        super();
    }

    public void 初始化游戏(){
        System.out.println("子类：初始化国际象棋游戏");
    }

    protected void 验证玩家移动(玩家 p){
        System.out.println("子类：验证国际象棋移动：" + p.getName());
    }

    protected void logGameStatistics(){
        super.记录游戏统计();
        System.out.println("子类：添加国际象棋特定的记录游戏统计：" );
    }
}

class 井字棋 extends 游戏{
    public 井字棋(){
        super();
    }

    public void 初始化游戏(){
        System.out.println("子类：初始化井字棋游戏");
    }

    protected void 验证玩家移动(玩家 p){
        System.out.println("子类：验证井字棋移动：" + p.getName());
    }
}

public class 多态{
    public static void main(String args[]){
        try{

            游戏 game = new 国际象棋();
            线程 t1 = new 线程(game);
            t1.start();
            线程.sleep(1000);
        }
    }
}

```

```

public void setNextPlayer(){
    if (currentPlayer == player1) {
        currentPlayer = player2;
    } else{
        currentPlayer = player1;
    }
}

public void run(){
    try{
        runGame();
    }catch(Exception err){
        err.printStackTrace();
    }
}

class Player{
    String name;
    Player(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}

/* Concrete Game implementation */
class Chess extends Game{
    public Chess(){
        super();
    }

    public void initializeGame(){
        System.out.println("Child class: Initialized Chess game");
    }

    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate Chess move：" + p.getName());
    }

    protected void logGameStatistics(){
        super.logGameStatistics();
        System.out.println("Child class: Add Chess specific logGameStatistics:");
    }
}

class TicTacToe extends Game{
    public TicTacToe(){
        super();
    }

    public void initializeGame(){
        System.out.println("Child class: Initialized TicTacToe game");
    }

    protected void validatePlayerMove(Player p){
        System.out.println("Child class: Validate TicTacToe move：" + p.getName());
    }
}

public class Polymorphism{
    public static void main(String args[]){
        try{

            Game game = new Chess();
            Thread t1 = new Thread(game);
            t1.start();
            Thread.sleep(1000);
        }
    }
}

```

```

game.setRunGame(false);
    线程.sleep(1000);

game = new 井字棋();
    线程 t2 = new 线程(game);
    t2.start();
    线程.sleep(1000);
game.setRunGame(false);

}catch(异常 err){
    err.printStackTrace();
}
}
}

```

输出：

```

子类：初始化国际象棋游戏
基类：定义游戏流程：
子类：验证国际象棋移动：玩家1
基类：移动持续时间：player.PlayerActTime - player.MoveShownTime
子类：验证国际象棋移动：玩家2
基类：移动持续时间：player.PlayerActTime - player.MoveShownTime
基类：记录游戏统计信息：
子类：添加国际象棋特定的记录游戏统计信息：

子类：初始化井字棋游戏
基类：定义游戏流程：
子类：验证井字棋移动：玩家1
基类：移动持续时间：player.PlayerActTime - player.MoveShownTime
子类：验证井字棋移动：玩家2
基类：移动持续时间：player.PlayerActTime - player.MoveShownTime
基类：记录游戏统计信息：

```

第111.4节：虚函数

虚方法是Java中非静态且前面没有final关键字的方法。Java中所有方法默认都是虚方法。虚方法在多态中起着重要作用，因为Java中的子类可以重写父类的方法，前提是被重写的方法是非静态的且具有相同的方法签名。

然而，也有一些方法不是虚方法。例如，如果方法被声明为private或带有final关键字，则该方法不是虚方法。

请考虑以下来自StackOverflow帖子“虚函数在C#和Java中如何工作？”的继承与虚方法的修改示例：

```

public class A{
    public void hello(){
        System.out.println("Hello");
    }

    public void boo(){
        System.out.println("Say boo");
    }
}

```

```

game.setRunGame(false);
Thread.sleep(1000);

game = new TicTacToe();
Thread t2 = new Thread(game);
t2.start();
Thread.sleep(1000);
game.setRunGame(false);

}catch(Exception err){
    err.printStackTrace();
}
}
}

```

Output:

```

Child class: Initialized Chess game
Base class: Defining the flow for Game:
Child class: Validate Chess move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate Chess move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:
Child class: Add Chess specific logGameStatistics:

Child class: Initialized TicTacToe game
Base class: Defining the flow for Game:
Child class: Validate TicTacToe move:Player 1
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Child class: Validate TicTacToe move:Player 2
Base class: Move Duration: player.PlayerActTime - player.MoveShownTime
Base class: logGameStatistics:

```

Section 111.4: Virtual functions

Virtual Methods are methods in Java that are non-static and without the keyword Final in front. All methods by default are virtual in Java. Virtual Methods play important roles in Polymorphism because children classes in Java can override their parent classes' methods if the function being overridden is non-static and has the same method signature.

There are, however, some methods that are not virtual. For example, if the method is declared private or with the keyword final, then the method is not Virtual.

Consider the following modified example of inheritance with Virtual Methods from this StackOverflow post [How do virtual functions work in C# and Java?](#):

```

public class A{
    public void hello(){
        System.out.println("Hello");
    }

    public void boo(){
        System.out.println("Say boo");
    }
}

```

```

public class B extends A{
    public void hello(){
        System.out.println("No");
    }

    public void boo(){
        System.out.println("Say haha");
    }
}

```

如果我们调用类B并调用hello()和boo()，输出结果将是“No”和“Say haha”，因为B重写了A中的相同方法。尽管上面的例子几乎与方法重写完全相同，但理解类A中的方法默认都是虚方法（Virtual）是很重要的。

此外，我们可以使用abstract关键字来实现虚方法。用“abstract”关键字声明的方法没有方法定义，意味着方法体尚未实现。再次考虑上面的例子，只是将boo()方法声明为抽象方法：

```

public class A{
    public void hello(){
        System.out.println("Hello");
    }

    abstract void boo();
}

public class B extends A{
    public void hello(){
        System.out.println("No");
    }

    public void boo(){
        System.out.println("Say haha");
    }
}

```

如果我们从B调用boo()，输出仍然是“Say haha”，因为B继承了抽象方法boo()并使boo()输出“Say haha”。

使用的资料来源及进一步阅读：

[C# 和 Java 中的虚函数是如何工作的？](#)

看看这个很棒的答案，它提供了关于虚函数的更完整的信息：

[你能在Java中编写虚函数/方法吗？](#)

第111.5节：通过添加类而不修改现有代码来添加行为

```

import java.util.ArrayList;
import java.util.List;

import static java.lang.System.out;

public class 多态演示 {

```

```

public class B extends A{
    public void hello(){
        System.out.println("No");
    }

    public void boo(){
        System.out.println("Say haha");
    }
}

```

If we invoke class B and call hello() and boo(), we would get "No" and "Say haha" as the resulting output because B overrides the same methods from A. Even though the example above is almost exactly the same as method overriding, it is important to understand that the methods in class A are all, by default, Virtual.

Additionally, we can implement Virtual methods using the abstract keyword. Methods declared with the keyword "abstract" does not have a method definition, meaning the method's body is not yet implemented. Consider the example from above again, except the boo() method is declared abstract:

```

public class A{
    public void hello(){
        System.out.println("Hello");
    }

    abstract void boo();
}

public class B extends A{
    public void hello(){
        System.out.println("No");
    }

    public void boo(){
        System.out.println("Say haha");
    }
}

```

If we invoke boo() from B, the output will still be "Say haha" since B inherits the abstract method boo() and makes boo () output "Say haha".

Sources used and further readings:

[How do virtual functions work in C# and Java?](#)

Check out this great answer that gives a much more complete information about Virtual functions:

[Can you write virtual functions / methods in Java?](#)

Section 111.5: Adding behaviour by adding classes without touching existing code

```

import java.util.ArrayList;
import java.util.List;

import static java.lang.System.out;

public class PolymorphismDemo {

```

```

public static void main(String[] args) {
    List<飞行器> machines = new ArrayList<飞行器>();
    machines.add(new 飞行器());
    machines.add(new 喷气机());
    machines.add(new 直升机());
    machines.add(new 喷气机());

    new 让东西飞().让机器飞(machines);
}

class 让东西飞 {
    public void 让机器飞(List<飞行器> flyingMachines) {
        for (飞行器 flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}

class 飞行器 {
    public void fly() {
        out.println("无实现");
    }
}

class 喷气式飞机 extends 飞行器 {
    @Override
    public void fly() {
        out.println("起飞, 滑行, 飞行");
    }

    public void 轰炸() {
        out.println("发射导弹");
    }
}

class 直升机 extends 飞行器 {
    @Override
    public void fly() {
        out.println("垂直起飞, 悬停, 飞行");
    }
}

```

说明

- a) MakeThingsFly 类可以处理所有类型为 FlyingMachine 的对象。
- b) 方法 letTheMachinesFly 在添加新类时，例如

PropellerPlane，也无需任何更改 () 即可正常工作。

```

public void 让机器飞(List<飞行器> flyingMachines) {
    for (飞行器 flyingMachine : flyingMachines) {
        flyingMachine.fly();
    }
}

```

这就是多态的力量。你可以用它来实现开闭原则。

```

public static void main(String[] args) {
    List<FlyingMachine> machines = new ArrayList<FlyingMachine>();
    machines.add(new FlyingMachine());
    machines.add(new Jet());
    machines.add(new Helicopter());
    machines.add(new Jet());

    new MakeThingsFly().letTheMachinesFly(machines);
}

class MakeThingsFly {
    public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
        for (FlyingMachine flyingMachine : flyingMachines) {
            flyingMachine.fly();
        }
    }
}

class FlyingMachine {
    public void fly() {
        out.println("No implementation");
    }
}

class Jet extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start, taxi, fly");
    }

    public void bombardment() {
        out.println("Fire missile");
    }
}

class Helicopter extends FlyingMachine {
    @Override
    public void fly() {
        out.println("Start vertically, hover, fly");
    }
}

```

Explanation

- a) The MakeThingsFly class can work with everything that is of type FlyingMachine.
- b) The method letTheMachinesFly also works without any change (!) when you add a new class, for example PropellerPlane:

```

public void letTheMachinesFly(List<FlyingMachine> flyingMachines) {
    for (FlyingMachine flyingMachine : flyingMachines) {
        flyingMachine.fly();
    }
}

```

That's the power of polymorphism. You can implement the [open-closed-principle](#) with it.

第112章：封装

想象一下，你有一个类，其中包含一些非常重要的变量，而这些变量被其他程序员通过他们的代码设置为不可接受的值。他们的代码导致了你的代码出现错误。作为解决方案，在面向对象编程中，你只允许通过方法来修改对象的状态（存储在其变量中）。隐藏对象的状态，并通过对象的方法提供所有交互，这被称为数据封装。

第112.1节：通过封装维护不变量

一个类有两个部分：接口和实现。

接口是类公开的功能。它的公共方法和变量是接口的一部分。

实现是类的内部工作机制。其他类不需要了解一个类的实现细节。

封装指的是将类的实现对该类的使用者隐藏的做法。这使得类可以对其内部状态做出假设。

例如，来看这个表示角度的类：

```
public class Angle {  
  
    private double angleInDegrees;  
    private double angleInRadians;  
  
    public static Angle angleFromDegrees(double degrees){  
        Angle a = new Angle();  
        a.angleInDegrees = degrees;  
        a.angleInRadians = Math.PI*degrees/180;  
        return a;  
    }  
  
    public static Angle angleFromRadians(double radians){  
        Angle a = new Angle();  
        a.angleInRadians = radians;  
        a.angleInDegrees = radians*180/Math.PI;  
        return a;  
    }  
  
    public double getDegrees(){  
        return angleInDegrees;  
    }  
  
    public double getRadians(){  
        return angleInRadians;  
    }  
  
    public void setDegrees(double degrees){  
        this.angleInDegrees = degrees;  
        this.angleInRadians = Math.PI*degrees/180;  
    }  
  
    public void setRadians(double radians){  
        this.angleInRadians = radians;  
        this.angleInDegrees = radians*180/Math.PI;  
    }  
    private Angle(){}
}
```

Chapter 112: Encapsulation

Imagine you had a class with some pretty important variables and they were set (by other programmers from their code) to unacceptable values. Their code brought errors in your code. As a solution, In OOP, you allow the state of an object (stored in its variables) to be modified only through methods. Hiding the state of an object and providing all interaction through an objects methods is known as Data Encapsulation.

Section 112.1: Encapsulation to maintain invariants

There are two parts of a class: the interface and the implementation.

The interface is the exposed functionality of the class. Its public methods and variables are part of the interface.

The implementation is the internal workings of a class. Other classes shouldn't need to know about the implementation of a class.

Encapsulation refers to the practice of hiding the implementation of a class from any users of that class. This allows the class to make assumptions about its internal state.

For example, take this class representing an Angle:

```
public class Angle {  
  
    private double angleInDegrees;  
    private double angleInRadians;  
  
    public static Angle angleFromDegrees(double degrees){  
        Angle a = new Angle();  
        a.angleInDegrees = degrees;  
        a.angleInRadians = Math.PI*degrees/180;  
        return a;  
    }  
  
    public static Angle angleFromRadians(double radians){  
        Angle a = new Angle();  
        a.angleInRadians = radians;  
        a.angleInDegrees = radians*180/Math.PI;  
        return a;  
    }  
  
    public double getDegrees(){  
        return angleInDegrees;  
    }  
  
    public double getRadians(){  
        return angleInRadians;  
    }  
  
    public void setDegrees(double degrees){  
        this.angleInDegrees = degrees;  
        this.angleInRadians = Math.PI*degrees/180;  
    }  
  
    public void setRadians(double radians){  
        this.angleInRadians = radians;  
        this.angleInDegrees = radians*180/Math.PI;  
    }  
    private Angle(){}
}
```

}

该类依赖于一个基本假设（或不变量）：angleInDegrees 和 angleInRadians 始终保持同步。如果类成员是公共的，则无法保证这两种角度表示是相关联的。

第112.2节：封装以减少耦合

封装允许你对类进行内部更改，而不会影响调用该类的任何代码。这减少了耦合，即某个类对另一个类实现的依赖程度。

例如，我们来更改前面示例中 Angle 类的实现：

```
public class Angle {
    private double angleInDegrees;

    public static Angle angleFromDegrees(double degrees) {
        Angle a = new Angle();
        a.angleInDegrees = degrees;
        return a;
    }

    public static Angle angleFromRadians(double radians) {
        Angle a = new Angle();
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees() {
        return angleInDegrees;
    }

    public double getRadians() {
        return angleInDegrees*Math.PI / 180;
    }

    public void setDegrees(double degrees) {
        this.angleInDegrees = degrees;
    }

    public void setRadians(double radians) {
        this.angleInDegrees = radians*180/Math.PI;
    }

    private Angle(){}
}
```

该类的实现已更改为仅存储角度的一种表示形式，并在需要时计算另一种角度。

然而，实现发生了变化，但接口没有改变。如果调用类依赖于访问angleInRadians 方法，则需要更改为使用新的Angle版本。调用类不应关心类的内部表示。

}

This class relies on a basic assumption (or *invariant*): **angleInDegrees and angleInRadians are always in sync**. If the class members were public, there would be no guarantees that the two representations of angles are correlated.

Section 112.2: Encapsulation to reduce coupling

Encapsulation allows you to make internal changes to a class without affecting any code that calls the class. This reduces *coupling*, or how much any given class relies on the implementation of another class.

For example, let's change the implementation of the Angle class from the previous example:

```
public class Angle {
    private double angleInDegrees;

    public static Angle angleFromDegrees(double degrees) {
        Angle a = new Angle();
        a.angleInDegrees = degrees;
        return a;
    }

    public static Angle angleFromRadians(double radians) {
        Angle a = new Angle();
        a.angleInDegrees = radians*180/Math.PI;
        return a;
    }

    public double getDegrees() {
        return angleInDegrees;
    }

    public double getRadians() {
        return angleInDegrees*Math.PI / 180;
    }

    public void setDegrees(double degrees) {
        this.angleInDegrees = degrees;
    }

    public void setRadians(double radians) {
        this.angleInDegrees = radians*180/Math.PI;
    }

    private Angle(){}
}
```

The implementation of this class has changed so that it only stores one representation of the angle and calculates the other angle when needed.

However, **the implementation changed, but the interface didn't**. If a calling class relied on accessing the angleInRadians method, it would need to be changed to use the new version of Angle. Calling classes shouldn't care about the internal representation of a class.

第113章：Java代理

第113.1节：使用代理修改类

首先，确保所使用的代理在 Manifest.mf 中具有以下属性：

```
Can-Redefine-Classes: true  
Can-Retransform-Classes: true
```

启动Java代理将允许代理访问Instrumentation类。通过Instrumentation，您可以调用addTransformer(ClassFileTransformer transformer)。ClassFileTransformer允许您重写类的字节。该类只有一个方法，提供加载该类的ClassLoader、类名、java.lang.Class实例、其ProtectionDomain，最后是类本身的字节。

它看起来像这样：

```
byte[] transform(ClassLoader 加载器, String 类名, Class<?> 正在重新定义的类,  
ProtectionDomain 保护域, byte[] 类文件缓冲区)
```

仅通过字节修改类可能需要很长时间。为了解决这个问题，有一些库可以将类字节转换成更易用的形式。

在这个例子中，我将使用 ASM，但其他替代品如 Javassist 和 BCEL 也具有类似的功能。

```
ClassNode 获取节点(byte[] 字节) {  
    // 创建一个 ClassReader，将字节数组解析为 ClassNode  
    ClassReader cr = new ClassReader(字节);  
    ClassNode cn = new ClassNode();  
    try {  
        // 这会填充 ClassNode  
        cr.accept(cn, ClassReader.EXPAND_FRAMES);  
        cr = null;  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return cn;  
}
```

从这里可以对 ClassNode 对象进行修改。这使得更改字段/方法访问权限变得非常简单。此外，使用 ASM 的 Tree API 修改方法的字节码非常轻松。

编辑完成后，您可以使用以下方法将 ClassNode 转换回字节，并在 transform 方法中返回它们：

```
public static byte[] getNodeBytes(ClassNode cn, boolean useMaxs) {  
    ClassWriter cw = new ClassWriter(useMaxs ? ClassWriter.COMPUTE_MAXS :  
    ClassWriter.COMPUTE_FRAMES);  
    cn.accept(cw);  
    byte[] b = cw.toByteArray();  

```

第113.2节：在运行时添加代理

代理可以在 JVM 运行时添加。要加载代理，您需要使用 Attach API 的

Chapter 113: Java Agents

Section 113.1: Modifying classes with agents

Firstly, make sure that the agent being used has the following attributes in the Manifest.mf:

```
Can-Redefine-Classes: true  
Can-Retransform-Classes: true
```

Starting a java agent will let the agent access the class Instrumentation. With Instrumentation you can call addTransformer(ClassFileTransformer transformer). ClassFileTransformers will let you rewrite the bytes of classes. The class has only a single method which supplies the ClassLoader that loads the class, the class's name, a java.lang.Class instance of it, its ProtectionDomain, and lastly the bytes of the class itself.

It looks like this:

```
byte[] transform(ClassLoader loader, String className, Class<?> classBeingRedefined,  
ProtectionDomain protectionDomain, byte[] classfileBuffer)
```

Modifying a class purely from bytes can take ages. To remedy this there are libraries that can be used to convert the class bytes into something more usable.

In this example I'll be using ASM, but other alternatives like Javassist and BCEL have similar features.

```
ClassNode getNode(byte[] bytes) {  
    // Create a ClassReader that will parse the byte array into a ClassNode  
    ClassReader cr = new ClassReader(bytes);  
    ClassNode cn = new ClassNode();  
    try {  
        // This populates the ClassNode  
        cr.accept(cn, ClassReader.EXPAND_FRAMES);  
        cr = null;  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return cn;  
}
```

From here changes can be made to the ClassNode object. This makes changing field/method access incredibly easy. Plus with ASM's Tree API modifying the bytecode of methods is a breeze.

Once the edits are finished you can convert the ClassNode back into bytes with the following method and return them in the transform method:

```
public static byte[] getNodeBytes(ClassNode cn, boolean useMaxs) {  
    ClassWriter cw = new ClassWriter(useMaxs ? ClassWriter.COMPUTE_MAXS :  
    ClassWriter.COMPUTE_FRAMES);  
    cn.accept(cw);  
    byte[] b = cw.toByteArray();  

```

Section 113.2: Adding an agent at runtime

Agents can be added to a JVM at runtime. To load an agent you will need to use the Attach API's

`VirtualMachine.attach(String id)`。然后，您可以使用以下方法加载已编译的代理 jar：

```
public static void loadAgent(String agentPath) {
    String vmName = ManagementFactory.getRuntimeMXBean().getName();
    int index = vmName.indexOf('@');
    String pid = vmName.substring(0, index);
    try {
        File agentFile = new File(agentPath);
        VirtualMachine vm = VirtualMachine.attach(pid);
        vm.loadAgent(agentFile.getAbsolutePath(), "");
        VirtualMachine.attach(vm.id());
    } 捕获 (异常 e) {
        抛出新的 运行时异常(e);
    }
}
```

这不会调用已加载代理中的 `premain(String agentArgs, Instrumentation inst)` 方法，而是会调用 `agentmain(String agentArgs, Instrumentation inst)` 方法。这要求在代理的 Manifest.mf 中设置 `Agent-Class` 属性。

第113.3节：设置一个基本代理

Premain 类将包含方法 "`premain(String agentArgs, Instrumentation inst)`"

示例如下：

```
导入 java.lang.instrument.Instrumentation;

公共类 PremainExample {
    公共静态 无返回值 premain(字符串 agentArgs, Instrumentation inst) {
        系统.输出.打印(agentArgs);
    }
}
```

编译成 jar 文件后，打开 Manifest 并确保其中包含 `Premain-Class` 属性。

示例如下：

Premain-类: PremainExample

要在另一个 Java 程序 "myProgram" 中使用该代理，必须在 JVM 参数中定义该代理：

```
java -javaagent:PremainAgent.jar -jar myProgram.jar
```

`VirtualMachine.attatch(String id)`。您可以在以下方法中加载一个编译好的 agent jar：

```
public static void loadAgent(String agentPath) {
    String vmName = ManagementFactory.getRuntimeMXBean().getName();
    int index = vmName.indexOf('@');
    String pid = vmName.substring(0, index);
    try {
        File agentFile = new File(agentPath);
        VirtualMachine vm = VirtualMachine.attatch(pid);
        vm.loadAgent(agentFile.getAbsolutePath(), "");
        VirtualMachine.attatch(vm.id());
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

This will not call `premain(String agentArgs, Instrumentation inst)` in the loaded agent, but instead will call `agentmain(String agentArgs, Instrumentation inst)`. This requires `Agent-Class` to be set in the agent Manifest.mf.

Section 113.3: Setting up a basic agent

The Premain class will contain the method "`premain(String agentArgs Instrumentation inst)`"

Here is an example:

```
import java.lang.instrument.Instrumentation;

public class PremainExample {
    public static void premain(String agentArgs, Instrumentation inst) {
        System.out.println(agentArgs);
    }
}
```

When compiled into a jar file open the Manifest and ensure that it has the `Premain-Class` attribute.

Here is an example:

Premain-类: PremainExample

To use the agent with another java program "myProgram" you must define the agent in the JVM arguments:

```
java -javaagent:PremainAgent.jar -jar myProgram.jar
```

第114章：可变参数（Varargs）

第114.1节：使用可变参数

将可变参数用作方法定义的参数时，可以传入数组或一系列参数。如果传入一系列参数，它们会自动转换为数组。

此示例展示了如何将数组和一系列参数传入printVarArgArray()方法，以及它们在方法内部代码中如何被相同处理：

```
public class VarArgs {  
  
    // 此方法将打印传入参数的全部内容  
  
    void printVarArgArray(int... x) {  
        for (int i = 0; i < x.length; i++) {  
            System.out.print(x[i] + ",");  
        }  
    }  
  
    public static void main(String args[]) {  
        VarArgs obj = new VarArgs();  
  
        // 使用数组：  
        int[] testArray = new int[]{10, 20};  
        obj.printVarArgArray(testArray);  
  
        System.out.println(" ");  
  
        // 使用一系列参数  
        obj.printVarArgArray(5, 6, 5, 8, 6, 31);  
    }  
}
```

输出：

```
10,20,  
5,6,5,8,6,31
```

如果你这样定义方法，会导致编译时错误。

```
void method(String... a, int... b, int c){} //编译时错误 (多个可变参数)  
void method(int... a, String b){} //编译时错误 (可变参数必须是最后一个参数)
```

第114.2节：指定可变参数

```
void doSomething(String... strings) {  
    for (String s : strings) {  
        System.out.println(s);  
    }  
}
```

最后一个参数类型后面的三个点表示最后一个参数可以作为数组传递，也可以作为一系列参数传递。可变参数只能用于最后一个参数位置。

Chapter 114: Varargs (Variable Argument)

Section 114.1: Working with Varargs parameters

Using varargs as a parameter for a method definition, it is possible to pass either an array or a sequence of arguments. If a sequence of arguments are passed, they are converted into an array automatically.

This example shows both an array and a sequence of arguments being passed into the printVarArgArray() method, and how they are treated identically in the code inside the method:

```
public class VarArgs {  
  
    // this method will print the entire contents of the parameter passed in  
  
    void printVarArgArray(int... x) {  
        for (int i = 0; i < x.length; i++) {  
            System.out.print(x[i] + ",");  
        }  
    }  
  
    public static void main(String args[]) {  
        VarArgs obj = new VarArgs();  
  
        // Using an array:  
        int[] testArray = new int[]{10, 20};  
        obj.printVarArgArray(testArray);  
  
        System.out.println(" ");  
  
        // Using a sequence of arguments  
        obj.printVarArgArray(5, 6, 5, 8, 6, 31);  
    }  
}
```

Output:

```
10,20,  
5,6,5,8,6,31
```

If you define the method like this, it will give compile-time errors.

```
void method(String... a, int... b, int c){} //Compile time error (multiple varargs )  
void method(int... a, String b){} //Compile time error (varargs must be the last argument)
```

Section 114.2: Specifying a varargs parameter

```
void doSomething(String... strings) {  
    for (String s : strings) {  
        System.out.println(s);  
    }  
}
```

The three periods after the final parameter's type indicate that the final argument may be passed as an array or as a sequence of arguments. Varargs can be used only in the final argument position.

第115章：日志记录 (java.util.logging)

第115.1节：记录复杂消息（高效地）

让我们来看一个你在许多程序中都能看到的日志记录示例：

```
public class LoggingComplex {  
  
    private static final Logger logger =  
        Logger.getLogger(LoggingComplex.class.getName());  
  
    private int total = 50, orders = 20;  
    private String username = "Bob";  
  
    public void takeOrder() {  
        // (...) 做一些操作  
        logger.fine(String.format("用户 %s 订购了 %d 件商品 (总共 %d 件) ",  
            username, orders, total));  
        // (...) 其他内容  
    }  
  
    // 一些其他方法和计算  
}
```

上面的例子看起来完全没问题，但许多程序员忘记了Java虚拟机是栈机器。这意味着所有方法的参数都在执行方法之前**计算完成**。

这一事实对于Java中的日志记录至关重要，尤其是在记录像FINE、FINER、FINEST这样默认被禁用的低级别日志时。让我们来看一下takeOrder()方法的Java字节码。

使用javap -c LoggingComplex.class得到的结果大致如下：

```
public void takeOrder();  
    代码:  
    0: getstatic      #27 // 字段 logger:Ljava/util/logging/Logger;  
    3: ldc           #45 // 字符串 User %s ordered %d things (%d in total)  
    5: iconst_3  
    6: anewarray     #3 // 类 java/lang/Object  
    9: dup  
   10: iconst_0  
   11: aload_0  
   12: getfield     #40 // 字段 username:Ljava/lang/String;  
   15: aastore  
   16: dup  
   17: iconst_1  
   18: aload_0  
   19: getfield     #36 // 字段 orders:I  
   22: invokestatic #47 // 方法 java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
   25: aastore  
   26: 复制  
   27: iconst_2  
   28: aload_0  
   29: getfield     #34 // 字段 total:I  
   32: invokestatic #47 // 方法 java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
   35: aastore  
   36: invokevirtual #53 // 方法  
   java/lang/String.format:(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;  
   39: invokevirtual #59 // 方法 java/util/logging/Logger.fine:(Ljava/lang/String;)V
```

Chapter 115: Logging (java.util.logging)

Section 115.1: Logging complex messages (efficiently)

Let's look at a sample of logging which you can see in many programs:

```
public class LoggingComplex {  
  
    private static final Logger logger =  
        Logger.getLogger(LoggingComplex.class.getName());  
  
    private int total = 50, orders = 20;  
    private String username = "Bob";  
  
    public void takeOrder() {  
        // (...) making some stuff  
        logger.fine(String.format("User %s ordered %d things (%d in total)",  
            username, orders, total));  
        // (...) some other stuff  
    }  
  
    // some other methods and calculations  
}
```

The above example looks perfectly fine, but many programmers forgets that Java VM is stack machine. This means that all method's parameters are calculated **before** executing the method.

This fact is crucial for logging in Java, especially for logging something in low levels like FINE, FINER, FINEST which are disabled by default. Let's look at Java bytecode for the takeOrder() method.

The result for javap -c LoggingComplex.class is something like this:

```
public void takeOrder();  
    Code:  
    0: getstatic      #27 // Field logger:Ljava/util/logging/Logger;  
    3: ldc           #45 // String User %s ordered %d things (%d in total)  
    5: iconst_3  
    6: anewarray     #3 // class java/lang/Object  
    9: dup  
   10: iconst_0  
   11: aload_0  
   12: getfield     #40 // Field username:Ljava/lang/String;  
   15: aastore  
   16: dup  
   17: iconst_1  
   18: aload_0  
   19: getfield     #36 // Field orders:I  
   22: invokestatic #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
   25: aastore  
   26: dup  
   27: iconst_2  
   28: aload_0  
   29: getfield     #34 // Field total:I  
   32: invokestatic #47 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
   35: aastore  
   36: invokevirtual #53 // Method  
   java/lang/String.format:(Ljava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;  
   39: invokevirtual #59 // Method java/util/logging/Logger.fine:(Ljava/lang/String;)V
```

42: return

第39行执行实际的日志记录。之前的所有工作（加载变量、创建新对象、使用format方法连接字符串）如果日志级别设置高于FINE（默认就是这样），都可能白费。这种日志记录非常低效，会消耗不必要的内存和处理器资源。

这就是为什么你应该先检查你想使用的日志级别是否启用。

正确的做法应该是：

```
public void takeOrder() {  
    // 执行一些操作  
    if (logger.isLoggable(Level.FINE)) {  
        // 当不需要时不执行任何操作  
        logger.fine(String.format("用户 %s 订购了 %d 件商品 (共 %d 件) ",  
            username, orders, total));  
    }  
    // 其他操作  
}
```

42: return

Line 39 runs the actual logging. All of the previous work (loading variables, creating new objects, concatenating Strings in format method) can be for nothing if logging level is set higher than FINE (and by default it is). Such logging can be very inefficient, consuming unnecessary memory and processor resources.

That's why you should ask if the level you want to use is enabled.

The right way should be:

```
public void takeOrder() {  
    // making some stuff  
    if (logger.isLoggable(Level.FINE)) {  
        // no action taken when there's no need for it  
        logger.fine(String.format("User %s ordered %d things (%d in total)",  
            username, orders, total));  
    }  
    // some other stuff  
}
```

自Java 8起：

Logger类新增了接受Supplier<String>作为参数的方法，可以直接通过lambda表达式提供：

```
public void takeOrder() {  
    // 执行一些操作  
    logger.fine(() -> String.format("用户 %s 订购了 %d 件商品 (共 %d 件) ",  
        username, orders, total));  
    // 其他操作  
}
```

供应商get()方法——在这种情况下是lambda——仅在相应级别启用时调用，因此不再需要if结构。

第115.2节：使用默认日志记录器

此示例展示了如何使用默认的日志记录API。

```
import java.util.logging.Level;  
import java.util.logging.Logger;  
  
public class MyClass {  
  
    // 获取当前类的日志记录器  
    private static final Logger LOG = Logger.getLogger(MyClass.class.getName());  
  
    public void foo() {  
        LOG.info("一条日志消息");  
        LOG.log(Level.INFO, "另一条日志消息");  
  
        LOG.fine("一条详细消息");  
  
        // 记录异常  
        try {  
            // 代码可能抛出异常  
        } catch (SomeException ex) {  
            // 记录警告，打印“出现了问题”  
        }  
    }  
}
```

Since Java 8:

The Logger class has additional methods that take a Supplier<String> as parameter, which can simply be provided by a lambda:

```
public void takeOrder() {  
    // making some stuff  
    logger.fine(() -> String.format("User %s ordered %d things (%d in total)",  
        username, orders, total));  
    // some other stuff  
}
```

The Suppliers get() method - in this case the lambda - is only called when the corresponding level is enabled and so the ifconstruction is not needed anymore.

Section 115.2: Using the default logger

This example shows how to use the default logging api.

```
import java.util.logging.Level;  
import java.util.logging.Logger;  
  
public class MyClass {  
  
    // retrieve the logger for the current class  
    private static final Logger LOG = Logger.getLogger(MyClass.class.getName());  
  
    public void foo() {  
        LOG.info("A log message");  
        LOG.log(Level.INFO, "Another log message");  
  
        LOG.fine("A fine message");  
  
        // logging an exception  
        try {  
            // code might throw an exception  
        } catch (SomeException ex) {  
            // log a warning printing "Something went wrong"  
        }  
    }  
}
```

```

    // 以及异常信息和堆栈跟踪
LOG.log(Level.WARNING, "出现了问题", ex);
}

String s = "Hello World!";

// 记录一个对象
LOG.log(Level.FINER, "字符串 s: {0}", s);

// 记录多个对象
LOG.log(Level.FINEST, "字符串 s: {0} 的长度为 {1}", new Object[]{s, s.length()});
}
}

```

第115.3节：日志级别

Java 日志 API 有 7 个级别。级别按降序排列如下：

- SEVERE (最高值)
- WARNING
- INFO
- 配置
- 细
- 更细
- 最细 (最低值)

默认级别是INFO（但这取决于系统和所使用的虚拟机）。

注意：还有级别OFF（可用于关闭日志记录）和ALL（与OFF相反）。

示例代码：

```

import java.util.logging.Logger;

public class Levels {
    private static final Logger logger = Logger.getLogger(Levels.class.getName());

    public static void main(String[] args) {

logger.severe("由SEVERE记录的消息");
        logger.warning("由WARNING记录的消息");
        logger.info("由INFO记录的消息");
logger.config("由CONFIG记录的消息");
        logger.fine("由FINE记录的消息");
        logger.finer("由FINER记录的消息");
        logger.finest("由FINEST记录的消息");

        // 上述所有方法实际上只是
        // public void log(Level level, String msg): 的快捷方式
logger.log(Level.FINEST, "由 FINEST 记录的消息");
    }
}

```

默认情况下，运行此类只会输出级别高于CONFIG的消息：

```

2016年7月23日 21:16:11 LevelsExample main
SEVERE: 由 SEVERE 记录的消息

```

```

    // together with the exception message and stacktrace
LOG.log(Level.WARNING, "Something went wrong", ex);
}

String s = "Hello World!";

// logging an object
LOG.log(Level.FINER, "String s: {0}", s);

// logging several objects
LOG.log(Level.FINEST, "String s: {0} has length {1}", new Object[]{s, s.length()});
}
}

```

Section 115.3: Logging levels

Java Logging Api has 7 [levels](#). The levels in descending order are:

- SEVERE (highest value)
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST (lowest value)

The default level is INFO (but this depends on the system and used a virtual machine).

Note: There are also levels OFF (can be used to turn logging off) and ALL (the opposite of OFF).

Code example for this:

```

import java.util.logging.Logger;

public class Levels {
    private static final Logger logger = Logger.getLogger(Levels.class.getName());

    public static void main(String[] args) {

logger.severe("Message logged by SEVERE");
        logger.warning("Message logged by WARNING");
        logger.info("Message logged by INFO");
logger.config("Message logged by CONFIG");
        logger.fine("Message logged by FINE");
        logger.finer("Message logged by FINER");
        logger.finest("Message logged by FINEST");

        // All of above methods are really just shortcut for
        // public void log(Level level, String msg):
logger.log(Level.FINEST, "Message logged by FINEST");
    }
}

```

By default running this class will output only messages with level higher than CONFIG:

```

Jul 23, 2016 9:16:11 PM LevelsExample main
SEVERE: Message logged by SEVERE

```

```
2016年7月23日 21:16:11 LevelsExample main  
WARNING: 由 WARNING 记录的消息  
2016年7月23日 21:16:11 LevelsExample main  
INFO: 由 INFO 记录的消息
```

```
Jul 23, 2016 9:16:11 PM LevelsExample main  
WARNING: Message logged by WARNING  
Jul 23, 2016 9:16:11 PM LevelsExample main  
INFO: Message logged by INFO
```

第116章 : log4j / log4j2

Apache Log4j 是一个基于Java的日志工具，它是多个Java日志框架之一。本章节旨在展示如何在Java中设置和配置Log4j，并通过详细示例介绍其所有可能的使用方面。

第116.1节：属性文件用于记录到数据库

要使此示例正常工作，您需要一个与数据库运行系统兼容的JDBC驱动程序。一个开源的驱动程序，允许您连接到IBM System i上的DB2数据库，可以在这里找到：JT400尽管此示例针对DB2，但如果更换驱动程序并调整JDBC URL，它几乎适用于所有其他系统。

```
# 根日志记录器选项
log4j.rootLogger= ERROR, DB

# 将日志消息重定向到DB2
# 定义DB appender
log4j.appenders.DB=org.apache.log4j.jdbc.JDBCAppender

# 设置 JDBC URL (!! 请根据目标系统调整 !!!)
log4j.appenders.DB.URL=jdbc:as400://10.10.10.1:446/DATABASENAME;naming=system;errors=full;

# 设置 数据库 驱动程序 (!! 请根据目标系统调整 !!!)
log4j.appenders.DB.driver=com.ibm.as400.access.AS400JDBCDriver

# 设置 数据库用户名和密码
log4j.appenders.DB.user=USER
log4j.appenders.DB.password=PASSWORD

# 设置 要执行的SQL语句。
log4j.appenders.DB.sql=INSERT INTO DB.TABLENAME VALUES( '%d{yyyy-MM-
dd}', '%d{HH:mm:ss}', '%C', '%p', '%m' )

# 定义文件appender的布局
log4j.appenders.DB.layout=org.apache.log4j.PatternLayout
```

第116.2节：如何获取Log4j

当前版本 (log4j2)

使用Maven：

将以下依赖项添加到您的POM.xml文件中：

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.6.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

Chapter 116: log4j / log4j2

Apache Log4j is a Java-based logging utility, it is one of several Java logging frameworks. This topic is to show how to setup and configure Log4j in Java with detailed examples on all of its possible aspects of usage.

Section 116.1: Properties-File to log to DB

For this example to work you'll need a JDBC driver compatible to the system the database is running on. An opensource one that allows you to connect to DB2 databases on an IBM System i can be found here: [JT400](#)

Even though this example is DB2 specific, it works for almost every other system if you exchange the driver and adapt the JDBC URL.

```
# Root logger option
log4j.rootLogger= ERROR, DB

# Redirect log messages to a DB2
# Define the DB appender
log4j.appenders.DB=org.apache.log4j.jdbc.JDBCAppender

# Set JDBC URL (!! adapt to your target system !!!)
log4j.appenders.DB.URL=jdbc:as400://10.10.10.1:446/DATABASENAME;naming=system;errors=full;

# Set Database Driver (!! adapt to your target system !!!)
log4j.appenders.DB.driver=com.ibm.as400.access.AS400JDBCDriver

# Set database user name and password
log4j.appenders.DB.user=USER
log4j.appenders.DB.password=PASSWORD

# Set the SQL statement to be executed.
log4j.appenders.DB.sql=INSERT INTO DB.TABLENAME VALUES( '%d{yyyy-MM-
dd}', '%d{HH:mm:ss}', '%C', '%p', '%m' )

# Define the layout for file appender
log4j.appenders.DB.layout=org.apache.log4j.PatternLayout
```

Section 116.2: How to get Log4j

Current version (log4j2)

Using Maven:

Add the following dependency to your POM.xml file:

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.6.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>
```

使用 Ivy :

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-api" rev="2.6.2" />
  <dependency org="org.apache.logging.log4j" name="log4j-core" rev="2.6.2" />
</dependencies>
```

使用 Gradle :

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.6.2'
  compile group: 'org.apache.logging.log4j', name: 'log4j-core', version: '2.6.2'
}
```

获取 log4j 1.x

注意： Log4j 1.x 已达到生命周期终止 (EOL) (参见备注)。

使用 Maven :

在 POM.xml 文件中声明此依赖：

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

使用 Ivy :

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

使用 Gradle :

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

使用 Buildr :

```
'log4j:log4j:jar:1.2.17'
```

手动添加到构建路径中：

从 Log4j 官网项目下载 _____

第116.3节：设置属性文件

Log4j 允许你同时将数据记录到控制台和文件。创建一个log4j.properties文件，并在其中放入以下基本配置：

```
# 根日志记录器选项
log4j.rootLogger=DEBUG, stdout, file

# 将日志消息重定向到控制台
log4j.appenders.stdout=org.apache.log4j.ConsoleAppender
log4j.appenders.stdout.Target=System.out
log4j.appenders.stdout.layout=org.apache.log4j.PatternLayout
```

Using Ivy:

```
<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-api" rev="2.6.2" />
  <dependency org="org.apache.logging.log4j" name="log4j-core" rev="2.6.2" />
</dependencies>
```

Using Gradle:

```
dependencies {
  compile group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.6.2'
  compile group: 'org.apache.logging.log4j', name: 'log4j-core', version: '2.6.2'
}
```

Getting log4j 1.x

Note: Log4j 1.x has reached End-of-Life (EOL) (see Remarks).

Using Maven:

Declare this dependency in the POM.xml file:

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

Using Ivy:

```
<dependency org="log4j" name="log4j" rev="1.2.17"/>
```

Using Gradle:

```
compile group: 'log4j', name: 'log4j', version: '1.2.17'
```

Using Buildr:

```
'log4j:log4j:jar:1.2.17'
```

Adding manually in path build:

Download from Log4j [website project](#)

Section 116.3: Setting up property file

Log4j gives you possibility to log data into console and file at same time. Create a log4j.properties file and put inside this basic configuration:

```
# Root logger option
log4j.rootLogger=DEBUG, stdout, file

# Redirect log messages to console
log4j.appenders.stdout=org.apache.log4j.ConsoleAppender
log4j.appenders.stdout.Target=System.out
log4j.appenders.stdout.layout=org.apache.log4j.PatternLayout
```

```

log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# 将日志消息重定向到日志文件，支持文件滚动。
log4j.appenders.file=org.apache.log4j.RollingFileAppender
log4j.appenders.file.File=C:\\\\log4j-application.log
log4j.appenders.file.MaxFileSize=5MB
log4j.appenders.file.MaxBackupIndex=10
log4j.appenders.file.layout=org.apache.log4j.PatternLayout
log4j.appenders.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

```

如果您使用的是 maven，请将此属性文件放在路径：

```
/ProjectFolder/src/java/resources
```

第116.4节：基本的 log4j2.xml 配置文件

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%t] %C{2} %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>

```

这是一个基本的 log4j2.xml 配置，包含一个控制台输出器和一个根记录器。模式布局指定了用于记录语句的模式。

为了调试 log4j2.xml 的加载，可以在 log4j2.xml 的配置标签中添加属性 status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO>。

你也可以添加一个监视间隔，使其在指定的时间间隔后重新加载配置。监视间隔可以添加到配置标签中，格式如下：monitorInterval = 30。表示配置每 30 秒加载一次。

```

log4j.appenders.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

# Redirect log messages to a log file, support file rolling.
log4j.appenders.file=org.apache.log4j.RollingFileAppender
log4j.appenders.file.File=C:\\\\log4j-application.log
log4j.appenders.file.MaxFileSize=5MB
log4j.appenders.file.MaxBackupIndex=10
log4j.appenders.file.layout=org.apache.log4j.PatternLayout
log4j.appenders.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p %c{1}:%L - %m%n

```

If you are using maven, put this propertie file in path:

```
/ProjectFolder/src/java/resources
```

Section 116.4: Basic log4j2.xml configuration file

```

<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <Console name="STDOUT" target="SYSTEM_OUT">
      <PatternLayout pattern="%d %-5p [%t] %C{2} %m%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="STDOUT"/>
    </Root>
  </Loggers>
</Configuration>

```

This is a basic log4j2.xml configuration which has a console appender and a root logger. The pattern layout specifies which pattern should be used for logging the statements.

In order to debug the loading of log4j2.xml you can add the attribute status = <WARN | DEBUG | ERROR | FATAL | TRACE | INFO> in the configuration tag of your log4j2.xml.

You can also add a monitor interval so that it loads the configuration again after the specified interval period. The monitor interval can be added to the configuration tag as follows: monitorInterval = 30. It means that the config will be loaded every 30 seconds.

第 116.5 节：如何在 Java 代码中使用 Log4j

首先需要创建一个 final static logger 对象：

```
final static Logger logger = Logger.getLogger(classname.class);
```

然后，调用日志记录方法：

```

//记录一条错误信息
logger.info("关于某个参数的信息: " + parameter); // 注意这行代码可能会抛出
NullPointerException !

//为了提升性能，建议使用 `isXXXEnabled()` 方法
if( logger.isInfoEnabled() ){
  logger.info("关于某个参数的信息: " + parameter);
}

// 在 log4j2 中，参数替换因可读性和性能更佳而被推荐
// 参数替换仅在 info 级别激活时进行，这使得使用

```

First need to create a **final static** logger object:

```
final static Logger logger = Logger.getLogger(classname.class);
```

Then, call logging methods:

```

//logs an error message
logger.info("Information about some param: " + parameter); // Note that this line could throw a
NullPointerException !

//in order to improve performance, it is advised to use the `isXXXEnabled()` Methods
if( logger.isInfoEnabled() ){
  logger.info("Information about some param: " + parameter);
}

// In log4j2 parameter substitution is preferable due to readability and performance
// The parameter substitution only takes place if info level is active which obsoletes the use of

```

```

isXXXEnabled().
logger.info("关于某个参数的信息: {}", parameter);

//记录异常
logger.error("关于某个错误的信息: ", exception);

```

第116.6节：从 log4j 1.x 迁移到 2.x

如果您想将项目中现有的 log4j 1.x 迁移到 log4j 2.x，则需移除所有现有的 log4j 1.x 依赖，并添加以下依赖：

Log4j 1.x API 桥接

Maven 构建

```

<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>

```

Ivy 构建

```

<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-1.2-api" rev="2.6.2" />
</dependencies>

```

Gradle 构建

```

dependencies {
compile group: 'org.apache.logging.log4j', name: 'log4j-1.2-api', version: '2.6.2'
}

```

Apache Commons Logging 桥接如果您的项目使用 Apache Commons Logging 并且使用的是 log4j 1.x，且您想将其迁移到 log4j 2.x，则添加以下依赖：

Maven 构建

```

<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>

```

Ivy 构建

```

<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-jcl" rev="2.6.2" />
</dependencies>

```

Gradle 构建

```

isXXXEnabled().
logger.info("Information about some param: {}", parameter);

//logs an exception
logger.error("Information about some error: ", exception);

```

Section 116.6: Migrating from log4j 1.x to 2.x

If you want to migrate from existing log4j 1.x in your project to log4j 2.x then remove all existing log4j 1.x dependencies and add the following dependency:

Log4j 1.x API Bridge

Maven Build

```

<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-1.2-api</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>

```

Ivy Build

```

<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-1.2-api" rev="2.6.2" />
</dependencies>

```

Gradle Build

```

dependencies {
compile group: 'org.apache.logging.log4j', name: 'log4j-1.2-api', version: '2.6.2'
}

```

Apache Commons Logging Bridge If your project is using Apache Commons Logging which use log4j 1.x and you want to migrate it to log4j 2.x then add the following dependencies:

Maven Build

```

<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-jcl</artifactId>
    <version>2.6.2</version>
  </dependency>
</dependencies>

```

Ivy Build

```

<dependencies>
  <dependency org="org.apache.logging.log4j" name="log4j-jcl" rev="2.6.2" />
</dependencies>

```

Gradle Build

```
dependencies {
compile group: 'org.apache.logging.log4j', name: 'log4j-jcl', version: '2.6.2'
}
```

注意：不要删除任何现有的 Apache commons logging 依赖

参考：<https://logging.apache.org/log4j/2.x/maven-artifacts.html>

第116.7节：按级别过滤日志输出 (log4j 1.x)

您可以使用过滤器仅记录“低于”例如ERROR级别的消息。但PropertyConfigurator不支持该过滤器。因此，您必须切换到XML配置才能使用它。参见log4j-Wiki关于过滤器的说明。

示例“特定级别”

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
    <param name="File" value="info.log"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%m%n"/>
    </layout>
    <filter class="org.apache.log4j.varia.LevelMatchFilter">
        <param name="LevelToMatch" value="info" />
        <param name="AcceptOnMatch" value="true"/>
    </filter>
    <filter class="org.apache.log4j.varia.DenyAllFilter" />
</appender>
```

或“级别范围”

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
    <param name="File" value="info.log"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%m%n"/>
    </layout>
    <filter class="org.apache.log4j.varia.LevelRangeFilter">
        <param name="LevelMax" value="info"/>
        <param name="LevelMin" value="info"/>
        <param name="AcceptOnMatch" value="true"/>
    </filter>
</appender>
```

```
dependencies {
compile group: 'org.apache.logging.log4j', name: 'log4j-jcl', version: '2.6.2'
}
```

Note: Do not remove any existing dependencies of Apache commons logging

Reference: <https://logging.apache.org/log4j/2.x/maven-artifacts.html>

Section 116.7: Filter Logoutput by level (log4j 1.x)

You can use a filter to log only messages "lower" than e.g. ERROR level. **But the filter is not supported by PropertyConfigurator. So you must change to XML config to use it.** See [log4j-Wiki about filters](#).

Example "specific level"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
    <param name="File" value="info.log"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%m%n"/>
    </layout>
    <filter class="org.apache.log4j.varia.LevelMatchFilter">
        <param name="LevelToMatch" value="info" />
        <param name="AcceptOnMatch" value="true"/>
    </filter>
    <filter class="org.apache.log4j.varia.DenyAllFilter" />
</appender>
```

Or "Level range"

```
<appender name="info-out" class="org.apache.log4j.FileAppender">
    <param name="File" value="info.log"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%m%n"/>
    </layout>
    <filter class="org.apache.log4j.varia.LevelRangeFilter">
        <param name="LevelMax" value="info"/>
        <param name="LevelMin" value="info"/>
        <param name="AcceptOnMatch" value="true"/>
    </filter>
</appender>
```

第117章：Oracle官方代码标准

Oracle官方Java编程语言风格指南是Oracle开发人员遵循的标准，并建议其他Java开发人员也遵循。它涵盖了文件名、文件组织、缩进、注释、声明、语句、空白、命名约定、编程实践，并包含代码示例。

第117.1节：命名约定

包名

- 包名应全部小写，不使用下划线或其他特殊字符。
- 包名以开发者公司网站地址的反向权限部分开头。
该部分后面可以跟随项目/程序结构依赖的包子结构。
- 不要使用复数形式。遵循标准API的惯例，例如
`java.lang.annotation` 而不是 `java.lang.annotations`。
- **示例：**`com.yourcompany.widget.button`, `com.yourcompany.core.api`

类、接口和枚举名称

- 类和枚举名称通常应为名词。
- 接口名称通常应为名词或以 `...able` 结尾的形容词。
- 使用混合大小写，每个单词首字母大写（即CamelCase）。
- 匹配正则表达式 `^[A-Z][a-zA-Z0-9]*$`。
- 使用完整单词，避免使用缩写，除非该缩写比完整形式更为广泛使用。
- 如果缩写是较长类名的一部分，则将其格式化为单词。
- **示例：**`ArrayList`, `BigInteger`, `ArrayIndexOutOfBoundsException`, `Iterable`.

方法名

方法名通常应为动词或其他描述动作的词语

- 它们应符合正则表达式`^[a-z][a-zA-Z0-9]*$`。
- 使用混合大小写，首字母小写。
- **示例：**`toString`, `hashCode`

变量

变量名应使用混合大小写，首字母小写

- 符合正则表达式`^[a-z][a-zA-Z0-9]*$`
- **进一步建议：**[变量](#)
- **示例：**`elements`, `currentIndex`

类型变量

对于涉及类型变量较少的简单情况，使用单个大写字母。

- 匹配正则表达式 `^[A-Z][0-9]?$`
- 如果某个字母比另一个字母更具描述性（例如用于映射中的键和值的 `K` 和 `V`，或函数返回类型的 `R`），则使用该字母，否则使用 `T`。
- 对于单个字母类型变量变得混淆的复杂情况，使用全部大写字母的较长名称，并用下划线（`_`）分隔单词。

Chapter 117: Oracle Official Code Standard

[Oracle official style guide](#) for the Java Programming Language is a standard followed by developers at Oracle and recommended to be followed by any other Java developer. It covers filenames, file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices and includes a code example.

Section 117.1: Naming Conventions

Package names

- Package names should be all lower case without underscores or other special characters.
- Package names begin with the reversed authority part of the web address of the company of the developer.
This part can be followed by project/program structure dependent package substructure.
- Don't use plural form. Follow the convention of the standard API which uses for instance
`java.lang.annotation` and not `java.lang.annotations`.
- **Examples:** `com.yourcompany.widget.button`, `com.yourcompany.core.api`

Class, Interface and Enum Names

- Class and enum names should typically be nouns.
- Interface names should typically be nouns or adjectives ending with `...able`.
- Use mixed case with the first letter in each word in upper case (i.e. [CamelCase](#)).
- Match the regular expression `^[A-Z][a-zA-Z0-9]*$`.
- Use whole words and avoid using abbreviations unless the abbreviation is more widely used than the long form.
- Format an abbreviation as a word if it is part of a longer class name.
- **Examples:** `ArrayList`, `BigInteger`, `ArrayIndexOutOfBoundsException`, `Iterable`.

Method Names

Method names should typically be verbs or other descriptions of actions

- They should match the regular expression `^[a-z][a-zA-Z0-9]*$`.
- Use mixed case with the first letter in lower case.
- **Examples:** `toString`, `hashCode`

Variables

Variable names should be in mixed case with the first letter in lower case

- Match the regular expression `^[a-z][a-zA-Z0-9]*$`
- Further recommendation: [Variables](#)
- **Examples:** `elements`, `currentIndex`

Type Variables

For simple cases where there are few type variables involved use a single upper case letter.

- Match the regular expression `^[A-Z][0-9]?$`
- If one letter is more descriptive than another (such as `K` and `V` for keys and values in maps or `R` for a function return type) use that, otherwise use `T`.
- For complex cases where single letter type variables become confusing, use longer names written in all capital letters and use underscore (`_`) to separate words.

- 示例 : T, V, SRC_VERTEX

常量

常量（语言规则或约定中内容不可变的 static final 字段）应使用全部大写字母命名，并用下划线（_）分隔单词。

- 匹配正则表达式 `^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$`
- 示例 : BUFFER_SIZE, MAX_LEVEL

其他命名指南

- 避免在外部作用域中隐藏/遮蔽方法、变量和类型变量。
- 名称的冗长程度应与作用域的大小相关。（例如，对大型类的字段使用描述性名称，对局部短生命周期变量使用简短名称。）
- 为公共静态成员命名时，如果你认为它们会被静态导入，则让标识符具有自描述性。
- 进一步阅读：命名部分（[官方Java风格指南](#)中）

来源：[Oracle的Java风格指南](#)

第117.2节：类结构

类成员的顺序

类成员应按以下顺序排列：

1. 字段（按public、protected和private顺序）
2. 构造函数
3. 工厂方法
4. 其他方法（按 public、protected 和 private 顺序）

不要求主要按访问修饰符或标识符对字段和方法进行排序。

以下是这种顺序的示例：

```
类 Example {
    private int i;

    Example(int i) {
        this.i = i;
    }

    static Example getExample(int i) {
        return new Example(i);
    }

    @Override
    public String toString() {
        return "An example [" + i + "]";
    }
}
```

类成员的分组

- Examples: T, V, SRC_VERTEX

Constants

Constants (static final fields whose content is immutable, by language rules or by convention) should be named with all capital letters and underscore (_) to separate words.

- Match the regular expression `^[A-Z][A-Z0-9]*(_[A-Z0-9]+)*$`
- Examples: BUFFER_SIZE, MAX_LEVEL

Other guidelines on naming

- Avoid hiding/shadowing methods, variables and type variables in outer scopes.
- Let the verbosity of the name correlate to the size of the scope. (For instance, use descriptive names for fields of large classes and brief names for local short-lived variables.)
- When naming public static members, let the identifier be self descriptive if you believe they will be statically imported.
- Further reading: [Naming Section](#) (in the official Java Style Guide)

Source: [Java Style Guidelines](#) from Oracle

Section 117.2: Class Structure

Order of class members

Class members should be ordered as follows:

1. Fields (in order of public, protected and private)
2. Constructors
3. Factory methods
4. Other Methods (in order of public, protected and private)

Ordering fields and methods primarily by their access modifiers or identifier is not required.

Here is an example of this order:

```
class Example {

    private int i;

    Example(int i) {
        this.i = i;
    }

    static Example getExample(int i) {
        return new Example(i);
    }

    @Override
    public String toString() {
        return "An example [" + i + "]";
    }
}
```

Grouping of class members

- 相关字段应当分组放置。
- 嵌套类型可以在首次使用之前声明；否则应在字段之前声明。
- 构造函数和重载方法应按功能分组，并按参数个数递增排序。这意味着这些结构之间的委托在代码中是向下流动的。
- 构造函数应集中分组，中间不应有其他成员。
- 方法的重载版本应集中分组，中间不应有其他成员。

第117.3节：注解

声明注解应单独占一行，与被注解的声明分开。

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

但是，如果注解较少或较短，且注解的是单行方法，为了提高可读性，可以将注解写在方法同一行。例如，可以写成：

```
@Nullable String getName() { return name; }
```

为了保持一致性和可读性，所有注释要么都放在同一行，要么每个注释都放在单独的一行。

```
// 不好。
@Deprecated @SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

```
// 更糟。
@Deprecated @SafeVarargs
@CustomAnnotation public final Tuple<T> extend(T... elements) {
    ...
}
```

```
// 好。
@Deprecated
@SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

```
// 好。
@Deprecated @SafeVarargs @CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

- Related fields should be grouped together.
- A nested type may be declared right before its first use; otherwise it should be declared before the fields.
- Constructors and overloaded methods should be grouped together by functionality and ordered with increasing arity. This implies that delegation among these constructs flow downward in the code.
- Constructors should be grouped together without other members between.
- Overloaded variants of a method should be grouped together without other members between.

Section 117.3: Annotations

Declaration annotations should be put on a separate line from the declaration being annotated.

```
@SuppressWarnings("unchecked")
public T[] toArray(T[] typeHolder) {
    ...
}
```

However, few or short annotations annotating a single-line method may be put on the same line as the method if it improves readability. For example, one may write:

```
@Nullable String getName() { return name; }
```

For a matter of consistency and readability, either all annotations should be put on the same line or each annotation should be put on a separate line.

```
// Bad.
@Deprecated @SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

```
// Even worse.
@Deprecated @SafeVarargs
@CustomAnnotation public final Tuple<T> extend(T... elements) {
    ...
}
```

```
// Good.
@Deprecated
@SafeVarargs
@CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

```
// Good.
@Deprecated @SafeVarargs @CustomAnnotation
public final Tuple<T> extend(T... elements) {
    ...
}
```

第117.4节：导入语句

```
// 首先是 java/javax 包
import java.util.ArrayList;
import javax.tools.JavaCompiler;
```

Section 117.4: Import statements

```
// First java/javax packages
import java.util.ArrayList;
import javax.tools.JavaCompiler;
```

```
// 然后是第三方库
import com.fasterxml.jackson.annotation.JsonProperty;

// 然后是项目导入
import com.example.my.package.ClassA;
import com.example.my.package.ClassB;

// 然后是静态导入 (顺序与上述相同)
import static java.util.stream.Collectors.toList;
```

- 导入语句应排序...

- ...主要按非静态 / 静态排序，非静态导入优先。
- ...其次按包来源排序，顺序如下
 - java 包
 - javax 包
 - 外部包（例如 org.xml）
 - 内部包（例如 com.sun）
- ...按包和类标识符的字典顺序进行三级排序

- 导入语句不应换行，无论是否超过推荐的最大行长度。
- 不应存在未使用的导入。

通配符导入

- 通常不应使用通配符导入。
- 当导入大量密切相关的类时（例如实现一个遍历树的访问者，树中有数十个不同的“节点”类），可以使用通配符导入。
- 无论如何，每个文件中不应使用超过一个通配符导入。

第117.5节：大括号

```
类 Example {
    void method(boolean error) {
        if (error) {
            Log.error("发生错误！");
            System.out.println("错误！");
        } else { // 使用大括号，因为另一个代码块使用了大括号。
            System.out.println("无错误");
        }
    }
}
```

- 左大括号应放在当前行的末尾，而不是单独占一行。
- 闭合大括号前应有换行，除非代码块为空（见下文简写形式）建议即使语言允许省略大括号的情况，如单行if 和循环体，也应使用大括号。
 - 如果代码块跨过多行（包括注释），必须使用大括号。
 - 如果if / else语句中的一个代码块使用了大括号，另一个代码块也必须使用。
 - 如果代码块是包围块中的最后一个，必须使用大括号。
- else、catch以及do...while循环中的while关键字应与前一个代码块的闭合大括号在同一行。

```
// Then third party libraries
import com.fasterxml.jackson.annotation.JsonProperty;

// Then project imports
import com.example.my.package.ClassA;
import com.example.my.package.ClassB;

// Then static imports (in the same order as above)
import static java.util.stream.Collectors.toList;
```

- Import statements should be sorted...

- ...primarily by non-static / static with non-static imports first.
- ...secondarily by package origin according to the following order
 - java packages
 - javax packages
 - external packages (e.g. org.xml)
 - internal packages (e.g. com.sun)
- ...tertiary by package and class identifier in lexicographical order

- Import statements should not be line wrapped, regardless of whether it exceeds the recommended maximum length of a line.
- No unused imports should be present.

Wildcard imports

- Wildcard imports should in general not be used.
- When importing a large number of closely-related classes (such as implementing a visitor over a tree with dozens of distinct “node” classes), a wildcard import may be used.
- In any case, no more than one wildcard import per file should be used.

Section 117.5: Braces

```
class Example {
    void method(boolean error) {
        if (error) {
            Log.error("Error occurred!");
            System.out.println("Error!");
        } else { // Use braces since the other block uses braces.
            System.out.println("No error");
        }
    }
}
```

- Opening braces should be put on the end of the current line rather than on a line by its own.
- There should be a new line in front of a closing brace unless the block is empty (see Short Forms below)
- Braces are recommended even where the language makes them optional, such as single-line if and loop bodies.
 - If a block spans more than one line (including comments) it must have braces.
 - If one of the blocks in a if / else statement has braces, the other block must too.
 - If the block comes last in an enclosing block, it must have braces.
- The else, catch and the while keyword in do...while loops go on the same line as the closing brace of the

简写形式

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

上述建议旨在提高统一性（从而增加熟悉度/可读性）。在某些情况下，偏离上述指南的“简写形式”同样具有良好的可读性，也可以使用。

这些情况包括例如简单的枚举声明、简单的方法和 lambda 表达式。

第117.6节：多余的括号

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";

// 不要这样做
return (flag ? "yes" : "no");
```

- 如果多余的分组括号（即不影响计算的括号）能提高可读性，则可以使用。
- 在涉及常见运算符的较短表达式中，通常应省略多余的分组括号，但在较长的表达式或涉及运算符优先级和结合性不明确的表达式中应包含括号。带有非平凡条件的三元表达式属于后者。
- 紧跟在return关键字后的整个表达式不得被括号包围。

第117.7节：修饰符

```
class 示例类 {
    // 访问修饰符优先 (不要写成"static public")
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

interface 示例接口 {
    // 避免使用'public' 和 'abstract'，因为它们是隐含的
    void sayHello();
}
```

- 修饰符应按以下顺序排列

- 访问修饰符 (**public / private / protected**)
- **abstract**
- **static**
- **final**
- **transient**
- **volatile**
- **default**
- **synchronized**
- **native**
- **strictfp**

preceding block.

Short forms

```
enum Response { YES, NO, MAYBE }
public boolean isReference() { return true; }
```

The above recommendations are intended to improve uniformity (and thus increase familiarity / readability). In some cases “short forms” that deviate from the above guidelines are just as readable and may be used instead. These cases include for instance simple enum declarations and trivial methods and lambda expressions.

Section 117.6: Redundant Parentheses

```
return flag ? "yes" : "no";

String cmp = (flag1 != flag2) ? "not equal" : "equal";

// Don't do this
return (flag ? "yes" : "no");
```

- Redundant grouping parentheses (i.e. parentheses that does not affect evaluation) may be used if they improve readability.
- Redundant grouping parentheses should typically be left out in shorter expressions involving common operators but included in longer expressions or expressions involving operators whose precedence and associativity is unclear without parentheses. Ternary expressions with non-trivial conditions belong to the latter.
- The entire expression following a **return** keyword must not be surrounded by parentheses.

Section 117.7: Modifiers

```
class ExampleClass {
    // Access modifiers first (don't do for instance "static public")
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}

interface ExampleInterface {
    // Avoid 'public' and 'abstract' since they are implicit
    void sayHello();
}
```

- Modifiers should go in the following order

- Access modifier (**public / private / protected**)
- **abstract**
- **static**
- **final**
- **transient**
- **volatile**
- **default**
- **synchronized**
- **native**
- **strictfp**

- 当修饰符是隐含时，不应写出。例如，接口方法既不应声明为public，也不应声明为abstract，嵌套的枚举和接口也不应声明为static。
- 除非能提高可读性或记录实际的设计决策，否则方法参数和局部变量不应声明为final。
- 字段应声明为final，除非有充分理由使其可变。

第117.8节：缩进

- 缩进级别为四个空格。
- 缩进只能使用空格字符。禁止使用制表符 (Tab)。
- 空行不得缩进。（这由禁止行尾空白规则隐含规定。）
- case行应缩进四个空格，case内的语句应再缩进四个空格。

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

有关如何缩进续行的指导，请参阅包装语句 (Wrapping statements)。

第117.9节：字面量

```
long l = 5432L;
int i = 0x123 + 0xABCD;
byte b = 0b1010;
float f1 = 1 / 5432f;
float f2 = 0.123e4f;
double d1 = 1 / 5432d; // 或 1 / 5432.0
double d2 = 0x1.3p2;
```

- long 字面量应使用大写字母 L 后缀。
- 十六进制字面量应使用大写字母 A-F。
- 所有其他数字前缀、中缀和后缀应使用小写字母。

第117.10节：包声明

```
package com.example.my.package;
```

包声明不应换行，无论是否超过推荐的行最大长度。

第117.11节：Lambda表达式

```
Runnable r = () -> System.out.println("Hello World");
Supplier<String> c = () -> "Hello World";
```

- Modifiers should not be written out when they are implicit. For example, interface methods should neither be declared **public** nor **abstract**, and nested enums and interfaces should not be declared static.
- Method parameters and local variables should not be declared **final** unless it improves readability or documents an actual design decision.
- Fields should be declared **final** unless there is a compelling reason to make them mutable.

Section 117.8: Indentation

- Indentation level is **four spaces**.
- Only space characters may be used for indentation. **No tabs**.
- Empty lines must not be indented. (This is implied by the no trailing white space rule.)
- case** lines should be indented with four spaces, and statements within the case should be indented with another four spaces.

```
switch (var) {
    case TWO:
        setChoice("two");
        break;
    case THREE:
        setChoice("three");
        break;
    default:
        throw new IllegalArgumentException();
}
```

Refer to Wrapping statements for guidelines on how to indent continuation lines.

Section 117.9: Literals

```
long l = 5432L;
int i = 0x123 + 0xABCD;
byte b = 0b1010;
float f1 = 1 / 5432f;
float f2 = 0.123e4f;
double d1 = 1 / 5432d; // or 1 / 5432.0
double d2 = 0x1.3p2;
```

- long** literals should use the upper case letter L suffix.
- Hexadecimal literals should use upper case letters A-F.
- All other numerical prefixes, infixes, and suffixes should use lowercase letters.

Section 117.10: Package declaration

```
package com.example.my.package;
```

The package declaration should not be line wrapped, regardless of whether it exceeds the recommended maximum length of a line.

Section 117.11: Lambda Expressions

```
Runnable r = () -> System.out.println("Hello World");
Supplier<String> c = () -> "Hello World";
```

```
// Collection::contains 是一个简单的一元方法，其行为  
// 从上下文中可以清楚地看出。这里更推荐使用方法引用。  
appendFilter(goodStrings::contains);
```

```
// 在这种情况下，Lambda表达式比单纯的 tempMap::put 更容易理解  
trackTemperature((time, temp) -> tempMap.put(time, temp));
```

- 表达式 lambda 优先于单行块 lambda。
- 方法引用通常应优先于 lambda 表达式。
- 对于绑定的实例方法引用，或参数个数大于一的方法，lambda 表达式可能更易理解，因此更受推荐。尤其当方法的行为从上下文中不明确时。
- 除非能提高可读性，否则应省略参数类型。
- 如果 lambda 表达式超过几行，考虑创建一个方法。

第 117.12 节：Java 源文件

- 所有行必须以换行符 (LF, ASCII 值 10) 结尾，而不能是 CR 或 CR+LF。
- 行尾不得有多余的空白字符。
- 源文件名必须等于其包含的类名，后跟 .java 扩展名，即使文件只包含包私有类也如此。此规则不适用于不包含任何类声明的文件，例如 package-info.java。

第 117.13 节：语句换行

- 源代码和注释每行通常不应超过80个字符，几乎不应超过100个字符，包括缩进。

字符限制必须根据具体情况判断。真正重要的是语义“密度”和行的可读性。无故将行写得过长会使其难以阅读；同样，“英勇的尝试”将其限制在80列内也会使其难以阅读。这里所述的灵活性旨在帮助开发者避免这两种极端，而不是最大化利用显示器空间。

- URL或示例命令不应换行。

```
// 好的，即使缩进后可能超过最大行宽。  
Error e = isTypeParam  
    ? Errors.InvalidRepeatableAnnotationNotApplicable(targetContainerType, on)  
    : Errors.InvalidRepeatableAnnotationNotApplicableInContext(targetContainerType));
```

```
// 建议换行  
String pretty = Stream.of(args)  
    .map(Argument::prettyPrint)  
    .collectors(joining(", "));
```

```
// 对最大行宽的解释过于严格。可读性受损。  
Error e = isTypeParam  
    ? Errors.InvalidRepeatableAnnotationNotApplicable(  
        targetContainerType, on)  
    : Errors.InvalidRepeatableAnnotationNotApplicableInContext(  
        targetContainerType);
```

```
// 即使在字符限制内，也应换行
```

```
// Collection::contains is a simple unary method and its behavior is  
// clear from the context. A method reference is preferred here.  
appendFilter(goodStrings::contains);
```

```
// A lambda expression is easier to understand than just tempMap::put in this case  
trackTemperature((time, temp) -> tempMap.put(time, temp));
```

- Expression lambdas are preferred over single-line block lambdas.
- Method references should generally be preferred over lambda expressions.
- For bound instance method references, or methods with arity greater than one, a lambda expression may be easier to understand and therefore preferred. Especially if the behavior of the method is not clear from the context.
- The parameter types should be omitted unless they improve readability.
- If a lambda expression stretches over more than a few lines, consider creating a method.

Section 117.12: Java Source Files

- All lines must be terminated with a line feed character (LF, ASCII value 10) and not for instance CR or CR+LF.
- There may be no trailing white space at the end of a line.
- The name of a source file must equal the name of the class it contains followed by the .java extension, even for files that only contain a package private class. This does not apply to files that do not contain any class declarations, such as package-info.java.

Section 117.13: Wrapping statements

- Source code and comments should generally not exceed 80 characters per line and rarely if ever exceed 100 characters per line, including indentation.

The character limit must be judged on a case by case basis. What really matters is the semantical “density” and readability of the line. Making lines gratuitously long makes them hard to read; similarly, making “heroic attempts” to fit them into 80 columns can also make them hard to read. The flexibility outlined here aims to enable developers to avoid these extremes, not to maximize use of monitor real-estate.

- URLs or example commands should not be wrapped.

```
// Ok even though it might exceed max line width when indented.  
Error e = isTypeParam  
    ? Errors.InvalidRepeatableAnnotationNotApplicable(targetContainerType, on)  
    : Errors.InvalidRepeatableAnnotationNotApplicableInContext(targetContainerType));
```

```
// Wrapping preferable  
String pretty = Stream.of(args)  
    .map(Argument::prettyPrint)  
    .collectors(joining(", "));
```

```
// Too strict interpretation of max line width. Readability suffers.  
Error e = isTypeParam  
    ? Errors.InvalidRepeatableAnnotationNotApplicable(  
        targetContainerType, on)  
    : Errors.InvalidRepeatableAnnotationNotApplicableInContext(  
        targetContainerType);
```

```
// Should be wrapped even though it fits within the character limit
```

```
String pretty = Stream.of(args).map(Argument::prettyPrint).collectors(joining(", "));
```

- 在较高的语法层次进行换行优于在较低的语法层次进行换行。
- 每行最多应包含一条语句。
- 续行应采用以下四种缩进方式之一
 - 变体1：相对于上一行缩进多8个空格。
 - 变体2：相对于换行表达式的起始列多8个空格。
 - 变体3：与前一个同级表达式对齐（只要能明确这是续行）
 - 变体4：与链式表达式中前一个方法调用对齐。

第117.14节：方法声明的换行

```
int someMethod(String aString,
               List<Integer> aList,
               Map<String, String> aMap,
               int anInt,
               long aLong,
               Set<Number> aSet,
               double aDouble) {
}

int someMethod(String aString, List<Integer> aList,
               Map<String, String> aMap, int anInt, long aLong,
               double aDouble, long aLong) {
}

int someMethod(String aString,
               List<Map<Integer, StringBuffer>> aListOfMaps,
               Map<String, String> aMap)
               throws IllegalArgumentException {
}

int someMethod(String aString, List<Integer> aList,
               Map<String, String> aMap, int anInt)
               throws IllegalArgumentException {
}
```

- 方法声明可以通过垂直列出参数，或者换行并加8个额外空格来格式化
- 如果需要换行包装 throws 子句，请将换行放在 throws 子句前面，并确保它从参数列表中突出显示，可以相对于函数声明缩进 +8，或者相对于上一行缩进 +8。

第117.15节：表达式换行

- 如果一行接近最大字符限制，始终考虑将其拆分为多条语句 / 表达式，而不是换行该行。
- 在运算符前换行。
- 在链式方法调用中的 . 前换行。

```
String pretty = Stream.of(args).map(Argument::prettyPrint).collectors(joining(" ", " "));
```

- Wrapping at a higher syntactical level is preferred over wrapping at a lower syntactical level.
- There should be at most 1 statement per line.
- A continuation line should be indented in one of the following four ways
 - **Variant 1:** With 8 extra spaces relative to the indentation of the previous line.
 - **Variant 2:** With 8 extra spaces relative to the starting column of the wrapped expression.
 - **Variant 3:** Aligned with previous sibling expression (as long as it is clear that it's a continuation line)
 - **Variant 4:** Aligned with previous method call in a chained expression.

Section 117.14: Wrapping Method Declarations

```
int someMethod(String aString,
               List<Integer> aList,
               Map<String, String> aMap,
               int anInt,
               long aLong,
               Set<Number> aSet,
               double aDouble) {
}

int someMethod(String aString, List<Integer> aList,
               Map<String, String> aMap, int anInt, long aLong,
               double aDouble, long aLong) {
}

int someMethod(String aString,
               List<Map<Integer, StringBuffer>> aListOfMaps,
               Map<String, String> aMap)
               throws IllegalArgumentException {
}

int someMethod(String aString, List<Integer> aList,
               Map<String, String> aMap, int anInt)
               throws IllegalArgumentException {
}
```

- Method declarations can be formatted by listing the arguments vertically, or by a new line and +8 extra spaces
- If a throws clause needs to be wrapped, put the line break in front of the throws clause and make sure it stands out from the argument list, either by indenting +8 relative to the function declaration, or +8 relative to the previous line.

Section 117.15: Wrapping Expressions

- If a line approaches the maximum character limit, always consider breaking it down into multiple statements / expressions instead of wrapping the line.
- Break before operators.
- Break before the . in chained method calls.

```

popupMsg("收件箱通知：您有 "
+newMsgs + " 条新消息");

// 不要这样！看起来像是两个参数
popupMsg("收件箱通知：您有 " +
newMsgs + " 条新消息");

```

第117.16节：空白字符

垂直空白

- 应使用单个空白行来分隔...

- 包声明
- 类声明
- 构造函数
- 方法
- 静态初始化块
- 实例初始化块

- ...并且可用于分隔逻辑上的分组

- 导入语句
- 字段
- 语句

- 多个连续的空白行应仅用于分隔相关成员组，而不应作为标准的成员间距。

水平空白

- 应使用单个空格...

- 用于将关键字与相邻的开括号或闭括号及大括号分开
- 在所有二元运算符及类似箭头的运算符符号（如lambda表达式中的箭头）前后以及增强型for循环中的冒号后（但标签的冒号前不加空格）
- 在以//开始的注释后。
- 在用逗号分隔的参数和用分号分隔的for循环部分后。
- 在强制类型转换的右括号后。

- 在变量声明中，不建议对齐类型和变量名。

第117.17节：特殊字符

- 除换行符 (LF) 外，唯一允许的空白字符是空格 (ASCII值32)。注意，这意味着其他空白字符（例如字符串和字符字面量中）必须以转义形式书写。
- 应优先使用'\', '\"', '\\', '\b', '\r', '\f' 和 '\n'，而非对应的八进制（例如 \047）或Unicode（例如 \u0027）转义字符。
- 如果为了测试需要违反上述规则，测试应以程序方式生成所需的输入。

```

popupMsg( "Inbox notification: You have "
+ newMsgs + " new messages");

// Don't! Looks like two arguments
popupMsg( "Inbox notification: You have " +
newMsgs + " new messages");

```

Section 117.16: Whitespace

Vertical Whitespace

- A single blank line should be used to separate...

- Package declaration
- Class declarations
- Constructors
- Methods
- Static initializers
- Instance initializers

- ...and may be used to separate logical groups of

- import statements
- fields
- statements

- Multiple consecutive blank lines should only be used to separate groups of related members and not as the standard inter-member spacing.

Horizontal Whitespace

- A single space should be used...

- To separate keywords from neighboring opening or closing brackets and braces
- Before and after all binary operators and operator like symbols such as arrows in lambda expressions and the colon in enhanced for loops (but not before the colon of a label)
- After // that starts a comment.
- After commas separating arguments and semicolons separating the parts of a for loop.
- After the closing parenthesis of a cast.

- In variable declarations it is not recommended to align types and variables.

Section 117.17: Special Characters

- Apart from LF the only allowed white space character is Space (ASCII value 32). Note that this implies that other white space characters (in, for instance, string and character literals) must be written in escaped form.
- '\', '\"', '\\', '\t', '\b', '\r', '\f', and '\n' should be preferred over corresponding octal (e.g. \047) or Unicode (e.g. \u0027) escaped characters.
- Should there be a need to go against the above rules for the sake of testing, the test should generate the required input programmatically.

第117.18节：变量声明

- 每次声明一个变量（每行最多声明一个变量）
- 数组的方括号应放在类型上 (`String[] args`)，而不是变量上 (`String args[]`)。
- 在首次使用前声明局部变量，并尽可能靠近声明处初始化。

Section 117.18: Variable Declarations

- One variable per declaration (and at most one declaration per line)
- Square brackets of arrays should be at the type (`String[] args`) and not on the variable (`String args[]`).
- Declare a local variable right before it is first used, and initialize it as close to the declaration as possible.

第118章：字符编码

第118.1节：从UTF-8编码的文件读取文本

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadingUTF8TextFile {

    public static void main(String[] args) throws IOException {
        //StandardCharsets 自 Java 1.7 起可用
        //对于早期版本使用 Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
wr.write("奇怪的西里尔字母符号 bl");
    }
    /* 第一种方法。适用于大文件 */
    try (BufferedReader reader = Files.newBufferedReader(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
String line;
while ((line = reader.readLine()) != null) {
    System.out.print(line);
}
System.out.println(); //仅用于分隔输出
    /* 第二种方法。适用于小文件 */
    String s = new String(Files.readAllBytes(Paths.get("test.txt")), StandardCharsets.UTF_8);
System.out.print(s);
    }
}
}
```

第118.2节：以UTF-8编码将文本写入文件

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class WritingUTF8TextFile {
    public static void main(String[] args) throws IOException {
        //StandardCharsets 自 Java 1.7 起可用
        //对于早期版本使用 Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test2.txt"),
StandardCharsets.UTF_8)) {
wr.write("西里尔字母 bl");
    }
}
}
```

Chapter 118: Character encoding

Section 118.1: Reading text from a file encoded in UTF-8

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class ReadingUTF8TextFile {

    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for earlier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
wr.write("Strange cyrillic symbol bl");
    }
    /* First Way. For big files */
    try (BufferedReader reader = Files.newBufferedReader(Paths.get("test.txt"),
StandardCharsets.UTF_8)) {
String line;
while ((line = reader.readLine()) != null) {
    System.out.print(line);
}
System.out.println(); //just separating output
    /* Second way. For small files */
    String s = new String(Files.readAllBytes(Paths.get("test.txt")), StandardCharsets.UTF_8);
System.out.print(s);
    }
}
}
```

Section 118.2: Writing text to a file in UTF-8

```
import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;

public class WritingUTF8TextFile {
    public static void main(String[] args) throws IOException {
        //StandardCharsets is available since Java 1.7
        //for earlier version use Charset.forName("UTF-8");
        try (BufferedWriter wr = Files.newBufferedWriter(Paths.get("test2.txt"),
StandardCharsets.UTF_8)) {
wr.write("Cyrillic symbol bl");
    }
}
}
```

第118.3节：获取字符串的UTF-8字节表示

```
import java.nio.charset.StandardCharsets;
import java.util.Arrays;

public class GetUtf8BytesFromString {

    public static void main(String[] args) {
        String str = "西里尔字母 бі";
        //StandardCharsets 自 Java 1.7 起可用
        //较早版本使用 Charset.forName("UTF-8");
        byte[] textInUtf8 = str.getBytes(StandardCharsets.UTF_8);

        System.out.println(Arrays.toString(textInUtf8));
    }
}
```

Section 118.3: Getting byte representation of a string in UTF-8

```
import java.nio.charset.StandardCharsets;
import java.util.Arrays;

public class GetUtf8BytesFromString {

    public static void main(String[] args) {
        String str = "Cyrillic symbol бі";
        //StandardCharsets is available since Java 1.7
        //for earlier version use Charset.forName("UTF-8");
        byte[] textInUtf8 = str.getBytes(StandardCharsets.UTF_8);

        System.out.println(Arrays.toString(textInUtf8));
    }
}
```

第119章 : Apache Commons Lang

第119.1节 : 实现equals()方法

要轻松实现对象的equals方法，可以使用EqualsBuilder类。

选择字段：

```
@Override  
public boolean equals(Object obj) {  
  
    if(!(obj instanceof MyClass)) {  
        return false;  
    }  
    MyClass theOther = (MyClass) obj;  
  
    EqualsBuilder builder = new EqualsBuilder();  
    builder.append(field1, theOther.field1);  
    builder.append(field2, theOther.field2);  
    builder.append(field3, theOther.field3);  
  
    return builder.isEquals();  
}
```

使用反射：

```
@Override  
public boolean equals(Object obj) {  
    return EqualsBuilder.reflectionEquals(this, obj, false);  
}
```

布尔参数用于指示 equals 方法是否应检查瞬态字段。

使用反射时避免某些字段：

```
@Override  
public boolean equals(Object obj) {  
    return EqualsBuilder.reflectionEquals(this, obj, "field1", "field2");  
}
```

第119.2节 : 实现hashCode()方法

为了轻松实现对象的 hashCode方法，你可以使用 HashCodeBuilder类。

选择字段：

```
@Override  
public int hashCode() {  
  
    HashCodeBuilder builder = new HashCodeBuilder();  
    builder.append(field1);  
    builder.append(field2);  
    builder.append(field3);  
  
    return builder.hashCode();  
}
```

Chapter 119: Apache Commons Lang

Section 119.1: Implement equals() method

To implement the equals method of an object easily you could use the EqualsBuilder class.

Selecting the fields:

```
@Override  
public boolean equals(Object obj) {  
  
    if(!(obj instanceof MyClass)) {  
        return false;  
    }  
    MyClass theOther = (MyClass) obj;  
  
    EqualsBuilder builder = new EqualsBuilder();  
    builder.append(field1, theOther.field1);  
    builder.append(field2, theOther.field2);  
    builder.append(field3, theOther.field3);  
  
    return builder.isEquals();  
}
```

Using reflection:

```
@Override  
public boolean equals(Object obj) {  
    return EqualsBuilder.reflectionEquals(this, obj, false);  
}
```

the boolean parameter is to indicates if the equals should check transient fields.

Using reflection avoiding some fields:

```
@Override  
public boolean equals(Object obj) {  
    return EqualsBuilder.reflectionEquals(this, obj, "field1", "field2");  
}
```

Section 119.2: Implement hashCode() method

To implement the hashCode method of an object easily you could use the HashCodeBuilder class.

Selecting the fields:

```
@Override  
public int hashCode() {  
  
    HashCodeBuilder builder = new HashCodeBuilder();  
    builder.append(field1);  
    builder.append(field2);  
    builder.append(field3);  
  
    return builder.hashCode();  
}
```

使用反射：

```
@Override  
public int hashCode() {  
    return HashCodeBuilder.reflectionHashCode(this, false);  
}
```

布尔参数表示是否应使用瞬态字段。

使用反射时避免某些字段：

```
@Override  
public int hashCode() {  
    return HashCodeBuilder.reflectionHashCode(this, "field1", "field2");  
}
```

第119.3节：实现toString()方法

要轻松实现对象的toString方法，可以使用ToStringBuilder类。

选择字段：

```
@Override  
public String toString() {  
  
    ToStringBuilder builder = new ToStringBuilder(this);  
    builder.append(field1);  
    builder.append(field2);  
    builder.append(field3);  
  
    return builder.toString();  
}
```

示例结果：

```
ar.com.jonat.lang.MyClass@dd7123[<null>,0,false]
```

显式指定字段名称：

```
@Override  
public String toString() {  
  
    ToStringBuilder builder = new ToStringBuilder(this);  
    builder.append("field1", field1);  
    builder.append("field2", field2);  
    builder.append("field3", field3);  
  
    return builder.toString();  
}
```

示例结果：

```
ar.com.jonat.lang.MyClass@dd7404[field1=<null>,field2=0,field3=false]
```

你可以通过参数更改样式：

```
@Override
```

Using reflection:

```
@Override  
public int hashCode() {  
    return HashCodeBuilder.reflectionHashCode(this, false);  
}
```

the boolean parameter indicates if it should use transient fields.

Using reflection avoiding some fields:

```
@Override  
public int hashCode() {  
    return HashCodeBuilder.reflectionHashCode(this, "field1", "field2");  
}
```

Section 119.3: Implement toString() method

To implement the `toString` method of an object easily you could use the `ToStringBuilder` class.

Selecting the fields:

```
@Override  
public String toString() {  
  
    ToStringBuilder builder = new ToStringBuilder(this);  
    builder.append(field1);  
    builder.append(field2);  
    builder.append(field3);  
  
    return builder.toString();  
}
```

Example result:

```
ar.com.jonat.lang.MyClass@dd7123[<null>,0,false]
```

Explicitly giving names to the fields:

```
@Override  
public String toString() {  
  
    ToStringBuilder builder = new ToStringBuilder(this);  
    builder.append("field1", field1);  
    builder.append("field2", field2);  
    builder.append("field3", field3);  
  
    return builder.toString();  
}
```

Example result:

```
ar.com.jonat.lang.MyClass@dd7404[field1=<null>,field2=0,field3=false]
```

You could change the style via parameter:

```
@Override
```

```
public String toString() {  
  
    ToStringBuilder builder = new ToStringBuilder(this,  
        ToStringStyle.MULTI_LINE_STYLE);  
    builder.append("field1", field1);  
    builder.append("field2", field2);  
    builder.append("field3", field3);  
  
    return builder.toString();  
}
```

示例结果：

```
ar.com.bna.lang.MyClass@ebbf5c[  
    field1=<null>  
    field2=0  
    field3=false  
]
```

有一些样式，例如 JSON、无类名、简短等.....

通过反射：

```
@Override  
public String toString() {  
    return ToStringBuilder.reflectionToString(this);  
}
```

你也可以指定样式：

```
@Override  
public String toString() {  
    return ToStringBuilder.reflectionToString(this, ToStringStyle.JSON_STYLE);  
}
```

```
public String toString() {
```

```
    ToStringBuilder builder = new ToStringBuilder(this,  
        ToStringStyle.MULTI_LINE_STYLE);  
    builder.append("field1", field1);  
    builder.append("field2", field2);  
    builder.append("field3", field3);  
  
    return builder.toString();  
}
```

Example result:

```
ar.com.bna.lang.MyClass@ebbf5c[  
    field1=<null>  
    field2=0  
    field3=false  
]
```

There are some styles, for example JSON, no Classname, short, etc ...

Via reflection:

```
@Override  
public String toString() {  
    return ToStringBuilder.reflectionToString(this);  
}
```

You could also indicate the style:

```
@Override  
public String toString() {  
    return ToStringBuilder.reflectionToString(this, ToStringStyle.JSON_STYLE);  
}
```

第120章：本地化与国际化

第120.1节：区域设置

`java.util.Locale` 类用于表示“地理、政治或文化”区域，以便对给定的文本、数字、日期或操作进行本地化。Locale对象因此可能包含国家、地区、语言，以及语言的变体，例如某个国家特定地区使用的方言，或在语言起源国以外的国家使用的方言。

Locale实例会传递给需要本地化其操作的组件，无论是转换输入、输出，还是仅用于内部操作。Locale类本身无法执行任何国际化或本地化操作。

语言

语言必须是ISO 639的2或3字符语言代码，或最多8个字符的注册语言子标签。如果某语言同时有2字符和3字符代码，应使用2字符代码。完整的语言代码列表可在IANA语言子标签注册表中找到。

语言代码不区分大小写，但Locale类始终使用小写形式的语言代码

创建区域设置

创建一个`java.util.Locale`实例可以通过四种不同的方式完成：

Locale常量
Locale构造函数
Locale.Builder类
Locale.forLanguageTag工厂方法

Java资源包（ResourceBundle）

你可以这样创建一个ResourceBundle实例：

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

假设我有一个属性文件i18n.properties：

message=这是本地化内容

输出：

这是本地化内容

设置Locale

如果你想使用其他语言重现该状态，可以使用`setDefault()`方法。用法如下：

```
setDefault(Locale.JAPANESE); //设置为日语
```

Chapter 120: Localization and Internationalization

Section 120.1: Locale

The `java.util.Locale` class is used to represent a "geographical, political or cultural" region to localize a given text, number, date or operation to. A Locale object may thus contain a country, region, language, and also a variant of a language, for instance a dialect spoken in a certain region of a country, or spoken in a different country than the country from which the language originates.

The Locale instance is handed to components that need to localize their actions, whether it is converting the input, output, or just need it for internal operations. The Locale class cannot do any internationalization or localization by itself

Language

The language must be an ISO 639 2 or 3 character language code, or a registered language subtag of up to 8 characters. In case a language has both a 2 and 3 character language code, use the 2 character code. A full list of language codes can be found in the IANA Language Subtag Registry.

Language codes are case insensitive, but the Locale class always use lowercase versions of the language codes

Creating a Locale

Creating a `java.util.Locale` instance can be done in four different ways:

Locale constants
Locale constructors
Locale.Builder class
Locale.forLanguageTag factory method

Java ResourceBundle

You create a ResourceBundle instance like this:

```
Locale locale = new Locale("en", "US");
ResourceBundle labels = ResourceBundle.getBundle("i18n.properties");
System.out.println(labels.getString("message"));
```

Consider I have a property file i18n.properties:

message=This is locale

Output:

This is locale

Setting Locale

If you want to reproduce the state using other languages, you can use `setDefault()` method. Its usage:

```
setDefault(Locale.JAPANESE); //Set Japanese
```

第120.2节：使用“locale”自动格式化日期

SimpleDateFormat在紧急情况下很有用，但正如其名称所示，它不适合大规模使用。

如果你在应用程序中到处硬编码"MM/dd/yyyy"，国际用户会不满意的。

让Java帮你完成这项工作

使用**DateFormat**中的**static**方法来获取适合用户的格式。对于桌面应用程序

(依赖于**默认区域设置**)，只需调用：

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

其中style是DateFormat中指定的格式化常量之一 (FULL、LONG、MEDIUM、SHORT等)。

对于服务器端应用程序，如果用户在请求中指定了他们的区域设置，则应显式地将其传递给getDateInstance()，具体如下：

```
字符串 localizedDate =
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

第120.3节：字符串比较

忽略大小写比较两个字符串：

```
"School".equalsIgnoreCase("school"); // true
```

不要使用

```
text1.toLowerCase().equals(text2.toLowerCase());
```

不同语言对大小写转换有不同规则。英文中 'I' 会转换为 'i'。但在土耳其语中，'I' 会变成 'İ'。如果必须使用 `toLowerCase()`，请使用带有 `Locale` 参数的重载版本：
`String.toLowerCase(Locale)`。

忽略细微差异比较两个字符串：

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.PRIMARY);
collator.equals("Gärten", "gaerten"); // 返回 true
```

按照自然语言顺序排序字符串，忽略大小写（使用排序键以：

```
String[] texts = new String[] {"Birne", "äther", "Apfel"};
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.SECONDARY); // 忽略大小写
Arrays.sort(texts, collator::compare); // 返回 {"Apfel", "äther", "Birne"}
```

Section 120.2: Automatically formatted Dates using "locale"

SimpleDateFormat is great in a pinch, but like the name suggests it doesn't scale well.

If you hard-code "MM/dd/yyyy" all over your application your international users won't be happy.

Let Java do the work for you

Use the **static** methods in **DateFormat** to retrieve the right formatting for your user. For a desktop application (where you'll rely on the [default locale](#)), simply call:

```
String localizedDate = DateFormat.getDateInstance(style).format(date);
```

Where style is one of the formatting constants (FULL, LONG, MEDIUM, SHORT, etc.) specified in **DateFormat**.

For a server-side application where the user specifies their locale as part of the request, you should pass it explicitly to `getDateInstance()` instead:

```
String localizedDate =
    DateFormat.getDateInstance(style, request.getLocale()).format(date);
```

Section 120.3: String Comparison

Compare two Strings ignoring case:

```
"School".equalsIgnoreCase("school"); // true
```

Don't use

```
text1.toLowerCase().equals(text2.toLowerCase());
```

Languages have different rules for converting upper and lower case. A 'I' would be converted to 'i' in English. But in Turkish a 'I' becomes a 'İ'. If you have to use `toLowerCase()` use the overload which expects a `Locale`:
`String.toLowerCase(Locale)`.

Comparing two Strings ignoring minor differences:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.PRIMARY);
collator.equals("Gärten", "gaerten"); // returns true
```

Sort Strings respecting natural language order, ignoring case (use collation key to:

```
String[] texts = new String[] {"Birne", "äther", "Apfel"};
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.SECONDARY); // ignore case
Arrays.sort(texts, collator::compare); // will return {"Apfel", "äther", "Birne"}
```

第121章：使用 Fork/Join 框架的并行编程

第121.1节：Java中的 Fork/Join 任务

Java中的 fork/join 框架非常适合可以拆分成更小部分并行解决的问题。
fork/join 问题的基本步骤是：

- 将问题拆分成多个部分
- 并行解决每个部分
- 将每个子解决方案合并成一个整体解决方案

`ForkjoinTask` 是定义此类问题的接口。通常期望您继承其抽象实现之一（通常是`RecursiveTask`），而不是直接实现该接口。

在这个例子中，我们将对一组整数求和，直到批处理大小不超过十为止。

```
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class SummingTask extends RecursiveTask<Integer> {
    private static final int MAX_BATCH_SIZE = 10;

    private final List<Integer> numbers;
    private final int minInclusive, maxExclusive;

    public SummingTask(List<Integer> numbers) {
        this(numbers, 0, numbers.size());
    }

    // 该构造函数仅在分割过程中内部使用
    private SummingTask(List<Integer> numbers, int minInclusive, int maxExclusive) {
        this.numbers = numbers;
        this.minInclusive = minInclusive;
        this.maxExclusive = maxExclusive;
    }

    @Override
    public Integer compute() {
        if (maxExclusive - minInclusive > MAX_BATCH_SIZE) {
            // 这对于单个批次来说太大了，所以我们将分成两个任务
            int mid = (minInclusive + maxExclusive) / 2;
            SummingTask leftTask = new SummingTask(numbers, minInclusive, mid);
            SummingTask rightTask = new SummingTask(numbers, mid, maxExclusive);

            // 将左侧任务作为新任务提交到同一个 ForkJoinPool
            leftTask.fork();

            // 在同一线程上运行右侧任务并获取结果
            int rightResult = rightTask.compute();

            // 等待左侧任务完成并获取其结果
            int leftResult = leftTask.join();

            // 并合并结果
            return leftResult + rightResult;
        } else {
    
```

Chapter 121: Parallel programming with Fork/Join framework

Section 121.1: Fork/Join Tasks in Java

The fork/join framework in Java is ideal for a problem that can be divided into smaller pieces and solved in parallel.
The fundamental steps of a fork/join problem are:

- Divide the problem into multiple pieces
- Solve each of the pieces in parallel to each other
- Combine each of the sub-solutions into one overall solution

A `ForkJoinTask` is the interface that defines such a problem. It is generally expected that you will subclass one of its abstract implementations (usually the `RecursiveTask`) rather than implement the interface directly.

In this example, we are going to sum a collection of integers, dividing until we get to batch sizes of no more than ten.

```
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class SummingTask extends RecursiveTask<Integer> {
    private static final int MAX_BATCH_SIZE = 10;

    private final List<Integer> numbers;
    private final int minInclusive, maxExclusive;

    public SummingTask(List<Integer> numbers) {
        this(numbers, 0, numbers.size());
    }

    // This constructor is only used internally as part of the dividing process
    private SummingTask(List<Integer> numbers, int minInclusive, int maxExclusive) {
        this.numbers = numbers;
        this.minInclusive = minInclusive;
        this.maxExclusive = maxExclusive;
    }

    @Override
    public Integer compute() {
        if (maxExclusive - minInclusive > MAX_BATCH_SIZE) {
            // This is too big for a single batch, so we shall divide into two tasks
            int mid = (minInclusive + maxExclusive) / 2;
            SummingTask leftTask = new SummingTask(numbers, minInclusive, mid);
            SummingTask rightTask = new SummingTask(numbers, mid, maxExclusive);

            // Submit the left hand task as a new task to the same ForkJoinPool
            leftTask.fork();

            // Run the right hand task on the same thread and get the result
            int rightResult = rightTask.compute();

            // Wait for the left hand task to complete and get its result
            int leftResult = leftTask.join();

            // And combine the result
            return leftResult + rightResult;
        } else {
    
```

```
// 这对于单个批次来说是可以的，所以我们将在这里立即运行它
int sum = 0;
for (int i = minInclusive; i < maxExclusive; i++) {
    sum += numbers.get(i);
}
return sum;
}
```

现在可以将此任务的实例传递给ForkJoinPool的实例。

```
// 因为我没有指定线程数
// 它将为每个可用处理器创建一个线程
ForkJoinPool pool = new ForkJoinPool();

// 将任务提交到线程池，并获取实际上是 Future 的对象
ForkJoinTask<Integer> task = pool.submit(new SummingTask(numbers));

// 等待结果
int result = task.join();
```

```
// This is fine for a single batch, so we will run it here and now
int sum = 0;
for (int i = minInclusive; i < maxExclusive; i++) {
    sum += numbers.get(i);
}
return sum;
}
```

An instance of this task can now be passed to an instance of [ForkJoinPool](#).

```
// Because I am not specifying the number of threads
// it will create a thread for each available processor
ForkJoinPool pool = new ForkJoinPool();

// Submit the task to the pool, and get what is effectively the Future
ForkJoinTask<Integer> task = pool.submit(new SummingTask(numbers));

// Wait for the result
int result = task.join();
```

第122章：非访问修饰符

非访问修饰符不会改变变量和方法的可访问性，但它们确实赋予它们特殊属性。

第122.1节：final

Java中的final可以指变量、方法和类。有三个简单规则：

- final变量不能被重新赋值
- final方法不能被重写
- final类不能被继承

用法

良好的编程实践

一些开发者认为在可能的情况下将变量标记为final是一种良好实践。如果你有一个不应该被更改的变量，你应该将其标记为final。

final关键字的一个重要用途是方法参数。如果你想强调一个方法不会改变其输入参数，就将这些属性标记为final。

```
public int sumup(final List<Integer> ints);
```

这强调了sumup方法不会改变ints。

内部类访问

如果你的匿名内部类想访问一个变量，该变量应被标记为final

```
public IPrintName printName(){  
    String name;  
    return new IPrintName(){  
        @Override  
        public void printName(){  
            System.out.println(name);  
        }  
    };  
}
```

该类无法编译，因为变量name不是final。

版本 ≥ Java SE 8

有效final变量是一个例外。这些是只被赋值一次的局部变量，因此可以被视为final。有效final变量也可以被匿名类访问。

final static 变量

尽管下面的代码在final变量foo不是static时完全合法，但如果是static则无法编译：

```
class TestFinal {  
    private final static List foo;
```

Chapter 122: Non-Access Modifiers

Non-Access Modifiers **do not change the accessibility of variables** and methods, but they do provide them **special properties**.

Section 122.1: final

final in Java can refer to variables, methods and classes. There are three simple rules:

- final variable cannot be reassigned
- final method cannot be overridden
- final class cannot be extended

Usages

Good Programming Practice

Some developer consider it good practice to mark a variable final when you can. If you have a variable that should not be changed, you should mark it final.

An important use of final keyword if for method parameters. If you want to emphasize that a method doesn't change its input parameters, mark the properties as final.

```
public int sumup(final List<Integer> ints);
```

This emphasizes that the sumup method is not going to change the ints.

Inner class Access

If your anonymous inner class wants to access a variable, the variable should be marked final

```
public IPrintName printName(){  
    String name;  
    return new IPrintName(){  
        @Override  
        public void printName(){  
            System.out.println(name);  
        }  
    };  
}
```

This class doesn't compile, as the variable name, is not final.

Version ≥ Java SE 8

Effectively final variables are an exception. These are local variables that are written to only once and could therefore be made final. Effectively final variables can be accessed from anonymous classes too.

final static variable

Even though the code below is completely legal when final variable foo is not static, in case of static it will not compile:

```
class TestFinal {  
    private final static List foo;
```

```
public Test() {
    foo = new ArrayList();
}
```

原因是，我们再重复一遍，final 变量不能被重新赋值。由于foo是静态的，它被所有TestFinal类的实例共享。当创建一个新的TestFinal类实例时，其构造函数被调用，因此foo被重新赋值，而编译器不允许这样做。在这种情况下初始化变量foo的正确方法是：

```
class TestFinal {
    private static final List foo = new ArrayList();
    //..
}
```

或者使用静态初始化块：

```
class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}
```

final方法在基类实现了一些派生类不应更改的重要功能时非常有用。它们也比非final方法更快，因为不涉及虚拟表的概念。

Java中的所有包装类都是final的，例如Integer、Long等。这些类的创建者不希望任何人例如将Integer扩展成自己的类并改变Integer类的基本行为。使类不可变的一个要求是子类不能重写方法。实现这一点最简单的方法是将类声明为final。

第122.2节：静态

static关键字用于类、方法或字段，使它们独立于类的任何实例而工作。

- 静态字段是类的所有实例共有的。它们不需要实例即可访问。
- 静态方法可以在没有类实例的情况下运行。但是，它们只能访问该类的静态字段。
- 静态类可以声明在其他类内部。它们不需要所在类的实例即可被实例化。

```
public class TestStatic
{
    static int staticVariable;

    static {
        // 这段代码在类首次加载时运行
        staticVariable = 11;
    }

    int nonStaticVariable = 5;

    static void doSomething() {

```

```
public Test() {
    foo = new ArrayList();
}
}
```

The reason is, let's repeat again, *final variable cannot be reassigned*. Since foo is static, it is shared among all instances of class TestFinal. When a new instance of a class TestFinal is created, its constructor is invoked and therefore foo gets reassigned which compiler does not allow. A correct way to initialize variable foo in this case is either:

```
class TestFinal {
    private static final List foo = new ArrayList();
    //..
}
```

or by using a static initializer:

```
class TestFinal {
    private static final List foo;
    static {
        foo = new ArrayList();
    }
    //..
}
```

final methods are useful when base class implements some important functionality that derived class is not supposed to change it. They are also faster than non-final methods, because there is no concept of virtual table involved.

All wrapper classes in Java are final, such as [Integer](#), [Long](#) etc. Creators of these classes didn't want that anyone can e.g. extend Integer into his own class and change the basic behavior of Integer class. One of the requirements to make a class immutable is that subclasses may not override methods. The simplest way to do this is to declare the class as **final**.

Section 122.2: static

The **static** keyword is used on a class, method, or field to make them work independently of any instance of the class.

- Static fields are common to all instances of a class. They do not need an instance to access them.
- Static methods can be run without an instance of the class they are in. However, they can only access static fields of that class.
- Static classes can be declared inside of other classes. They do not need an instance of the class they are in to be instantiated.

```
public class TestStatic
{
    static int staticVariable;

    static {
        // This block of code is run when the class first loads
        staticVariable = 11;
    }

    int nonStaticVariable = 5;

    static void doSomething() {

```

```

// 我们可以从静态方法访问静态变量
staticVariable = 10;
}

void add() {
    // 我们可以从非静态方法访问静态变量和非静态变量
    nonStaticVariable += staticVariable;
}

static class StaticInnerClass {
    int number;
    public StaticInnerClass(int _number) {
        number = _number;
    }

    void doSomething() {
        // 我们可以访问 number 和 staticVariable, 但不能访问 nonStaticVariable
        number += staticVariable;
    }

    int getNumber() {
        return number;
    }
}
}

// 静态字段和方法
TestStatic object1 = new TestStatic();

System.out.println(object1.staticVariable); // 11
System.out.println(TestStatic.staticVariable); // 11

TestStatic.doSomething();

TestStatic object2 = new TestStatic();

System.out.println(object1.staticVariable); // 10
System.out.println(object2.staticVariable); // 10
System.out.println(TestStatic.staticVariable); // 10

object1.add();

System.out.println(object1.nonStaticVariable); // 15
System.out.println(object2.nonStaticVariable); // 10

// 静态内部类
StaticInnerClass object3 = new TestStatic.StaticInnerClass(100);
StaticInnerClass object4 = new TestStatic.StaticInnerClass(200);

System.out.println(object3.getNumber()); // 100
System.out.println(object4.getNumber()); // 200

object3.doSomething();

System.out.println(object3.getNumber()); // 110
System.out.println(object4.getNumber()); // 200

```

第122.3节：抽象

抽象是隐藏实现细节，仅向用户展示功能的过程。一个抽象

```

// We can access static variables from static methods
staticVariable = 10;
}

void add() {
    // We can access both static and non-static variables from non-static methods
    nonStaticVariable += staticVariable;
}

static class StaticInnerClass {
    int number;
    public StaticInnerClass(int _number) {
        number = _number;
    }

    void doSomething() {
        // We can access number and staticVariable, but not nonStaticVariable
        number += staticVariable;
    }

    int getNumber() {
        return number;
    }
}
}

// Static fields and methods
TestStatic object1 = new TestStatic();

System.out.println(object1.staticVariable); // 11
System.out.println(TestStatic.staticVariable); // 11

TestStatic.doSomething();

TestStatic object2 = new TestStatic();

System.out.println(object1.staticVariable); // 10
System.out.println(object2.staticVariable); // 10
System.out.println(TestStatic.staticVariable); // 10

object1.add();

System.out.println(object1.nonStaticVariable); // 15
System.out.println(object2.nonStaticVariable); // 10

// Static inner classes
StaticInnerClass object3 = new TestStatic.StaticInnerClass(100);
StaticInnerClass object4 = new TestStatic.StaticInnerClass(200);

System.out.println(object3.getNumber()); // 100
System.out.println(object4.getNumber()); // 200

object3.doSomething();

System.out.println(object3.getNumber()); // 110
System.out.println(object4.getNumber()); // 200

```

Section 122.3: abstract

Abstraction is a process of hiding the implementation details and showing only functionality to the user. An abstract

类永远不能被实例化。如果一个类被声明为抽象类，那么其唯一目的就是供该类被继承。

```
抽象类 车辆
{
    抽象 无返回值 标语();
}

类 本田 继承 车辆
{
    无返回值 标语()
    {
        系统输出.打印行("开始特别的事情");
    }
}

类 丰田 继承 车辆
{
    无返回值 标语()
    {
        系统输出.打印行("驾驶你的梦想");
    }
}
```

第122.4节：strictfp

版本 ≥ Java SE 1.2

strictfp 修饰符用于浮点计算。该修饰符使浮点变量在多个平台上更加一致，并确保所有浮点计算均按照 IEEE 754 标准进行，以避免计算误差（舍入误差）、溢出和下溢，适用于32位和64位架构。该修饰符不能应用于抽象方法、变量或构造函数。

```
// strictfp 关键字可以应用于方法、类和接口。
strictfp 类 A{}

strictfp 接口 M{}

类 A{
    strictfp void m(){}
}
```

第122.5节：volatile

volatile 修饰符用于多线程编程。如果将字段声明为 volatile，则向线程发出信号，要求它们必须读取最新的值，而不是本地缓存的值。此外，volatile 的读写操作保证是原子的（对非 volatile 的 long 或 double 类型访问不是原子的），从而避免多个线程之间的某些读/写错误。

```
public class MyRunnable implements Runnable
{
    private volatile boolean active;

    public void run(){ // run 在一个线程中调用
        active = true;
        while (active){
            // 这里是一些代码
        }
    }
}
```

class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

```
abstract class Car
{
    abstract void tagLine();
}

class Honda extends Car
{
    void tagLine()
    {
        System.out.println("Start Something Special");
    }
}

class Toyota extends Car
{
    void tagLine()
    {
        System.out.println("Drive Your Dreams");
    }
}
```

Section 122.4: strictfp

Version ≥ Java SE 1.2

strictfp modifier is used for floating-point calculations. This modifier makes floating point variable more consistent across multiple platforms and ensure all the floating point calculations are done according to IEEE 754 standards to avoid errors of calculation (round-off errors), overflows and underflows on both 32bit and 64bit architecture. This cannot be applied on abstract methods, variables or constructors.

```
// strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}

strictfp interface M{}

class A{
    strictfp void m(){}
}
```

Section 122.5: volatile

The volatile modifier is used in multi threaded programming. If you declare a field as volatile it is a signal to threads that they must read the most recent value, not a locally cached one. Furthermore, volatile reads and writes are guaranteed to be atomic (access to a non-volatile long or double is not atomic), thus avoiding certain read/write errors between multiple threads.

```
public class MyRunnable implements Runnable
{
    private volatile boolean active;

    public void run(){ // run is called in one thread
        active = true;
        while (active){
            // some code here
        }
    }
}
```

```

    }

    public void stop(){ // stop() 从另一个线程调用
        active = false;
    }
}

```

第122.6节 : synchronized

同步修饰符用于控制多个线程对特定方法或代码块的访问。只有一个线程可以进入声明为同步的方法或代码块。synchronized关键字作用于对象的内置锁，对于同步方法，使用当前对象的锁，静态方法则使用类对象的锁。任何试图执行同步代码块的线程必须先获取对象锁。

```

类 Shared
{
    int i;

    synchronized void SharedMethod()
    {
        线程 t = 线程.currentThread();

        for(int i = 0; i <= 1000; i++)
        {
            系统.out.println(t.getName()+" : "+i);
        }
    }

    void SharedMethod2()
    {
        synchronized (this)
        {
            系统.out.println("对当前对象的访问是同步的 "+this);
        }
    }

    ...

    public static void main(String[] args)
    {
        final Shared s1 = new Shared();

        线程 t1 = new 线程("线程 - 1")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };

        线程 t2 = new 线程("线程 - 2")
        {
            @Override
            public void run()
            {
                s1.SharedMethod();
            }
        };
    }
}

```

```

    }

    public void stop(){ // stop() is called from another thread
        active = false;
    }
}

```

Section 122.6: synchronized

Synchronized modifier is used to control the access of a particular method or a block by multiple threads. Only one thread can enter into a method or a block which is declared as synchronized. synchronized keyword works on intrinsic lock of an object, in case of a synchronized method current objects lock and static method uses class object. Any thread trying to execute a synchronized block must acquire the object lock first.

```

class Shared
{
    int i;

    synchronized void SharedMethod()
    {
        Thread t = Thread.currentThread();

        for(int i = 0; i <= 1000; i++)
        {
            System.out.println(t.getName()+" : "+i);
        }
    }

    void SharedMethod2()
    {
        synchronized (this)
        {
            System.out.println("Thais access to correct object is synchronize "+this);
        }
    }

    public class ThreadsInJava
    {
        public static void main(String[] args)
        {
            final Shared s1 = new Shared();

            Thread t1 = new Thread("Thread - 1")
            {
                @Override
                public void run()
                {
                    s1.SharedMethod();
                }
            };

            Thread t2 = new Thread("Thread - 2")
            {
                @Override
                public void run()
                {
                    s1.SharedMethod();
                }
            };
        }
    }
}

```

```
};

t1.start();

    t2.start();
}

}
```

第122.7节 : transient (瞬态)

声明为transient的变量在对象序列化过程中不会被序列化。

```
public transient int limit = 55; // 不会被持久化
public int b; // 会被持久化
```

```
};

t1.start();

    t2.start();
}

}
```

Section 122.7: transient

A variable which is declared as transient will not be serialized during object serialization.

```
public transient int limit = 55; // will not persist
public int b; // will persist
```

第123章：进程

第123.1节：陷阱：Runtime.exec、Process和ProcessBuilder不理解shell语法

Runtime.exec(String ...) 和 Runtime.exec(String) 方法允许你作为外部进程执行命令1。在第一个版本中，你将命令名称和命令参数作为字符串数组的独立元素提供，Java运行时请求操作系统运行时系统启动外部命令。

第二个版本看似使用简单，但存在一些陷阱。

首先，下面是一个安全使用 exec(String) 的示例：

```
Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("创建了目录");
}
```

路径名中的空格

假设我们将上面的示例泛化，以便可以创建任意目录：

```
Process p = Runtime.exec("mkdir " + dirPath);
// ...
```

这通常可行，但如果 dirPath 是（例如）"/home/user/My Documents"，则会失败。问题在于 exec(String) 通过简单地查找空白字符将字符串拆分为命令和参数。命令字符串：

```
"mkdir /home/user/My Documents"
```

将被拆分为：

```
"mkdir", "/home/user/My", "Documents"
```

这将导致"mkdir"命令失败，因为它期望一个参数，而不是两个。

面对这种情况，一些程序员尝试在路径名周围添加引号。但这也不起作用：

```
"mkdir \" /home/user/My Documents\""
```

将被拆分为：

```
"mkdir", "\", "/home/user/My", "Documents\""
```

为“引用”空格而添加的额外双引号字符被视为其他非空白字符。实际上，任何我们尝试引用或转义空格的做法都会失败。

解决此特定问题的方法是使用 exec(String ...) 重载。

```
Process p = Runtime.exec("mkdir", dirPath);
// ...
```

Chapter 123: Process

Section 123.1: Pitfall: Runtime.exec, Process and ProcessBuilder don't understand shell syntax

The `Runtime.exec(String ...)` and `Runtime.exec(String)` methods allow you to execute a command as an external process¹. In the first version, you supply the command name and the command arguments as separate elements of the string array, and the Java runtime requests the OS runtime system to start the external command. The second version is deceptively easy to use, but it has some pitfalls.

First of all, here is an example of using `exec(String)` being used safely:

```
Process p = Runtime.exec("mkdir /tmp/testDir");
p.waitFor();
if (p.exitValue() == 0) {
    System.out.println("created the directory");
}
```

Spaces in pathnames

Suppose that we generalize the example above so that we can create an arbitrary directory:

```
Process p = Runtime.exec("mkdir " + dirPath);
// ...
```

This will typically work, but it will fail if `dirPath` is (for example) "/home/user/My Documents". The problem is that `exec(String)` splits the string into a command and arguments by simply looking for whitespace. The command string:

```
"mkdir /home/user/My Documents"
```

will be split into:

```
"mkdir", "/home/user/My", "Documents"
```

and this will cause the "mkdir" command to fail because it expects one argument, not two.

Faced with this, some programmers try to add quotes around the pathname. This doesn't work either:

```
"mkdir \" /home/user/My Documents\""
```

will be split into:

```
"mkdir", "\", "/home/user/My", "Documents\""
```

The extra double-quote characters that were added in attempt to "quote" the spaces are treated like any other non-whitespace characters. Indeed, anything we do quote or escape the spaces is going to fail.

The way to deal with this particular problems is to use the `exec(String ...)` overload.

```
Process p = Runtime.exec("mkdir", dirPath);
// ...
```

如果 `dirpath` 包含空白字符，这将有效，因为此重载的 `exec` 不会尝试拆分参数。字符串将按原样传递给操作系统的 `exec` 系统调用。

重定向、管道和其他 shell 语法

假设我们想重定向外部命令的输入或输出，或者运行一个管道。例如：

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

或者

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

(第一个示例列出文件系统中所有 Java 文件的名称，第二个示例打印“source”目录树中 Java 文件中的 `package` 语句2。)

这些不会按预期工作。在第一种情况下，“`find`”命令将以“`2>/dev/null`”作为命令参数运行。它不会被解释为重定向。在第二个示例中，管道符号（“`|`”）及其后面的内容将被传递给“`find`”命令。

这里的问题是 `exec` 方法和 `ProcessBuilder` 不理解任何 shell 语法。这包括重定向、管道、变量扩展、通配符等。

在少数情况下（例如，简单的重定向），您可以轻松地使用 `ProcessBuilder` 实现所需的效果。
然而，情况并非总是如此。另一种方法是在 shell 中运行命令行；例如：

```
Process p = Runtime.exec("bash", "-c",
    "find / -name *.java -print 2>/dev/null");
```

或者

```
Process p = Runtime.exec("bash", "-c",
    "find source -name \\*.java | xargs grep package");
```

但请注意，在第二个例子中，我们需要转义通配符字符（“`*`”），因为我们希望通配符由 “`find`” 解释，而不是由 shell 解释。

Shell 内置命令不起作用

假设以下示例在带有类 UNIX shell 的系统上不起作用：

```
Process p = Runtime.exec("cd", "/tmp"); // 更改 java 应用的主目录
```

或者

```
Process p = Runtime.exec("export", "NAME=value"); // 将 NAME 导出到 java 应用的环境中
```

这不起作用有几个原因：

1. “`cd`” 和 “`export`” 命令是 shell 内置命令。它们不存在作为独立的可执行文件。
2. 对于 shell 内置命令来说，要完成它们应做的事情（例如更改工作目录、更新环境），它们需要更改状态所在的位置。对于普通应用程序（包括 Java 应用程序），状态是与应用程序进程相关联的。例如，运行“`cd`”命令的子进程无法更改其父进程“`java`”的工作目录。同样，

This will work if `dirpath` includes whitespace characters because this overload of `exec` does not attempt to split the arguments. The strings are passed through to the OS `exec` system call as-is.

Redirection, pipelines and other shell syntax

Suppose that we want to redirect an external command's input or output, or run a pipeline. For example:

```
Process p = Runtime.exec("find / -name *.java -print 2>/dev/null");
```

or

```
Process p = Runtime.exec("find source -name *.java | xargs grep package");
```

(The first example lists the names of all Java files in the file system, and the second one prints the `package` statements2 in the Java files in the “source” tree.)

These are not going to work as expected. In the first case, the “`find`” command will be run with “`2>/dev/null`” as a command argument. It will not be interpreted as a redirection. In the second example, the pipe character (“`|`”）及其后面的内容将被传递给“`find`”命令。

The problem here is that the `exec` methods and `ProcessBuilder` do not understand any shell syntax. This includes redirections, pipelines, variable expansion, globbing, and so on.

In a few cases (for example, simple redirection) you can easily achieve the desired effect using `ProcessBuilder`. However, this is not true in general. An alternative approach is to run the command line in a shell; for example:

```
Process p = Runtime.exec("bash", "-c",
    "find / -name *.java -print 2>/dev/null");
```

or

```
Process p = Runtime.exec("bash", "-c",
    "find source -name \\*.java | xargs grep package");
```

But note that in the second example, we needed to escape the wildcard character (“`*`”） because we want the wildcard to be interpreted by “`find`” rather than the shell.

Shell builtin commands don't work

Suppose the following examples won't work on a system with a UNIX-like shell:

```
Process p = Runtime.exec("cd", "/tmp"); // Change java app's home directory
```

or

```
Process p = Runtime.exec("export", "NAME=value"); // Export NAME to the java app's environment
```

There are a couple of reasons why this won't work:

1. On “`cd`” and “`export`” commands are shell builtin commands. They don't exist as distinct executables.
2. For shell builtins to do what they are supposed to do (e.g. change the working directory, update the environment), they need to change the place where that state resides. For a normal application (including a Java application) the state is associated with the application process. So for example, the child process that would run the “`cd`” command could not change the working directory of its parent “`java`” process. Similarly,

一个exec的进程不能更改其后续进程的工作目录。

这个推理适用于所有 shell 内置命令。

1 - 你也可以使用ProcessBuilder，但这与本例的重点无关。

2 - 这有点粗糙.....但同样，这种方法的缺陷与本例无关。

第123.2节：简单示例（Java 版本 < 1.5）

此示例将调用 Windows 计算器。需要注意的是，退出代码会根据被调用的程序/脚本而变化。

```
package process.example;

import java.io.IOException;

public class App {

    public static void main(String[] args) {
        try {
            // 执行 Windows 计算器
            Process p = Runtime.getRuntime().exec("calc.exe");

            // 等待进程直到其终止
            int exitCode = p.waitFor();

            System.out.println(exitCode);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

one exec'd process cannot change the working directory for a process that follows it.

This reasoning applies to all shell builtin commands.

1 - You can use ProcessBuilder as well, but that is not relevant to the point of this example.

2 - This is a bit rough and ready ... but once again, the failings of this approach are not relevant to the example.

Section 123.2: Simple example (Java version < 1.5)

This example will call the windows calculator. It's important to notice that the exit code will vary accordingly to the program/script that is being called.

```
package process.example;

import java.io.IOException;

public class App {

    public static void main(String[] args) {
        try {
            // Executes windows calculator
            Process p = Runtime.getRuntime().exec("calc.exe");

            // Wait for process until it terminates
            int exitCode = p.waitFor();

            System.out.println(exitCode);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

第124章：Java本地访问

第124.1节：JNA简介

什么是JNA？

Java本地访问（JNA）是一个社区开发的库，为Java程序提供了方便访问本地共享库（Windows上的.dll文件，Unix上的.so文件……）

我该如何使用它？

- 首先，下载JNA的最新版本，并在项目的CLASSPATH中引用其jna.jar。
- 其次，复制、编译并运行下面的Java代码

为了本介绍的目的，我们假设使用的本地平台是Windows。如果你在其他平台运行，只需将代码中“msvcrt”字符串替换为“c”字符串即可。

下面这个小的Java程序将通过调用C语言的printf函数在控制台打印一条消息。

CRuntimeLibrary.java

```
包 jna.introduction;

导入 com.sun.jna.Library;
导入 com.sun.jna.Native;

// 我们声明所需的printf函数及包含它的库 (msvcrt) ...
公共接口 CRuntimeLibrary extends Library {

    CRuntimeLibrary INSTANCE =
        (CRuntimeLibrary) Native.loadLibrary("msvcrt", CRuntimeLibrary.class);

    void printf(String format, Object... args);
}
```

MyFirstJNAProgram.java

```
package jna.introduction;

// 现在我们调用 printf 函数...
public class MyFirstJNAProgram {
    public static void main(String args[]) {
        CRuntimeLibrary.INSTANCE.printf("Hello World from JNA !");
    }
}
```

接下来去哪？

在这里跳转到另一个主题或跳转到官方网站。_____

Chapter 124: Java Native Access

Section 124.1: Introduction to JNA

What is JNA?

Java Native Access (JNA) is a community-developed library providing Java programs an easy access to native shared libraries (.dll files on windows, .so files on Unix ...)

How can I use it?

- Firstly, download the [latest release of JNA](#) and reference its jna.jar in your project's CLASSPATH.
- Secondly, copy, compile and run the Java code below

For the purpose of this introduction, we suppose the native platform in use is Windows. If you're running on another platform simply replace the string "msvcrt" with the string "c" in the code below.

The small Java program below will print a message on the console by calling the C printf function.

CRuntimeLibrary.java

```
package jna.introduction;

import com.sun.jna.Library;
import com.sun.jna.Native;

// We declare the printf function we need and the library containing it (msvcrt)...
public interface CRuntimeLibrary extends Library {

    CRuntimeLibrary INSTANCE =
        (CRuntimeLibrary) Native.loadLibrary("msvcrt", CRuntimeLibrary.class);

    void printf(String format, Object... args);
}
```

MyFirstJNAProgram.java

```
package jna.introduction;

// Now we call the printf function...
public class MyFirstJNAProgram {
    public static void main(String args[]) {
        CRuntimeLibrary.INSTANCE.printf("Hello World from JNA !");
    }
}
```

Where to go now?

Jump into another topic here or jump to the [official site](#).

第125章：模块

第125.1节：定义基本模块

模块定义在名为`module-info.java`的文件中，称为模块描述符。它应放置在源代码根目录：

```
|-- module-info.java  
|-- com  
|-- example  
|-- foo  
|-- Foo.java  
|-- bar  
|-- Bar.java
```

这是一个简单的模块描述符：

```
module com.example {  
    requires java.httpclient;  
    exports com.example.foo;  
}
```

模块名称应当唯一，建议使用与包相同的反向域名命名法，以帮助确保唯一性。

模块`java.base`，包含Java的基础类，对任何模块都是隐式可见的，无需显式包含。

`requires` 声明允许我们使用其他模块，示例中导入了模块`java.httpclient`。

模块还可以指定其`exports`的包，从而使这些包对其他模块可见。

在`exports`子句中声明的包`com.example.foo`将对其他模块可见。`com.example.foo`的任何子包将不会被导出，它们需要自己的`export`声明。

相反，`com.example.bar`如果未在`exports`条款中列出，将不会对其他模块可见。

Chapter 125: Modules

Section 125.1: Defining a basic module

Modules are defined in a file named `module-info.java`, named a module descriptor. It should be placed in the source-code root:

```
|-- module-info.java  
|-- com  
|-- example  
|-- foo  
|-- Foo.java  
|-- bar  
|-- Bar.java
```

Here is a simple module descriptor:

```
module com.example {  
    requires java.httpclient;  
    exports com.example.foo;  
}
```

The module name should be unique and it is recommended that you use the same [Reverse-DNS naming notation](#) as used by packages to help ensure this.

The module `java.base`, which contains Java's basic classes, is implicitly visible to any module and does not need to be included.

The `requires` declaration allows us to use other modules, in the example the module `java.httpclient` is imported.

A module can also specify which packages it exports and therefore makes it visible to other modules.

The package `com.example.foo` declared in the `exports` clause will be visible to other modules. Any sub-packages of `com.example.foo` will not be exported, they need their own `export` declarations.

Conversely, `com.example.bar` which is not listed in `exports` clauses will not be visible to other modules.

第126章：并发编程 (线程)

并发计算是一种计算形式，其中多个计算同时执行，而不是顺序执行。Java语言设计支持通过线程使用并发编程。对象和资源可以被多个线程访问；每个线程都可能访问程序中的任何对象，程序员必须确保线程之间对对象的读写访问得到正确同步。

第126.1节：Callable和Future

虽然Runnable提供了一种将代码包装以在不同线程中执行的方法，但它有一个限制，即无法从执行中返回结果。获取Runnable执行返回值的唯一方法是将结果赋值给Runnable外部作用域中可访问的变量。

Callable在Java 5中作为Runnable的同级引入。Callable本质上相同，只是它有一个call方法代替run。call方法具有返回结果的额外能力，并且允许抛出受检异常。

通过Callable任务提交的结果可以通过Future获取

Future可以被视为一种容器，存放Callable计算的结果。Callable的计算可以在另一个线程中继续，任何尝试获取Future结果的操作都会阻塞，直到结果可用时才返回。

Callable接口

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

Future

```
interface Future<V> {  
    V get();  
    V get(long timeout, TimeUnit unit);  
    boolean cancel(boolean mayInterruptIfRunning);  
    boolean isCancelled();  
    boolean isDone();  
}
```

使用 Callable 和 Future 示例：

```
public static void main(String[] args) throws Exception {  
    ExecutorService es = Executors.newSingleThreadExecutor();  
  
    System.out.println("任务提交时间：" + new Date());  
    Future<String> result = es.submit(new ComplexCalculator());  
    // 调用 Future.get() 会阻塞，直到结果可用。所以我们现在大约要等待10秒  
  
    System.out.println("复杂计算结果是：" + result.get());  
    System.out.println("打印结果时的时间：" + new Date());  
}
```

我们的可调用对象执行一个耗时的计算

Chapter 126: Concurrent Programming (Threads)

Concurrent computing is a form of computing in which several computations are executed concurrently instead of sequentially. Java language is designed to support [concurrent programming](#) through the usage of threads. Objects and resources can be accessed by multiple threads; each thread can potentially access any object in the program and the programmer must ensure read and write access to objects is properly synchronized between threads.

Section 126.1: Callable and Future

While [Runnable](#) provides a means to wrap code to be executed in a different thread, it has a limitation in that it cannot return a result from the execution. The only way to get some return value from the execution of a [Runnable](#) is to assign the result to a variable accessible in a scope outside of the [Runnable](#).

Callable was introduced in Java 5 as a peer to [Runnable](#). Callable is essentially the same except it has a call method instead of run. The call method has the additional capability to return a result and is also allowed to throw checked exceptions.

The result from a Callable task submission is available to be tapped via a Future

Future can be considered a container of sorts that houses the result of the Callable computation. Computation of the callable can carry on in another thread, and any attempt to tap the result of a Future will block and will only return the result once it is available.

Callable Interface

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

Future

```
interface Future<V> {  
    V get();  
    V get(long timeout, TimeUnit unit);  
    boolean cancel(boolean mayInterruptIfRunning);  
    boolean isCancelled();  
    boolean isDone();  
}
```

Using Callable and Future example:

```
public static void main(String[] args) throws Exception {  
    ExecutorService es = Executors.newSingleThreadExecutor();  
  
    System.out.println("Time At Task Submission : " + new Date());  
    Future<String> result = es.submit(new ComplexCalculator());  
    // the call to Future.get() blocks until the result is available. So we are in for about a 10 sec  
    wait now  
    System.out.println("Result of Complex Calculation is : " + result.get());  
    System.out.println("Time At the Point of Printing the Result : " + new Date());  
}
```

Our Callable that does a lengthy computation

```

public class ComplexCalculator implements Callable<String> {

    @Override
    public String call() throws Exception {
        // 仅休眠10秒以模拟耗时计算
        Thread.sleep(10000);
        System.out.println("经过漫长的10秒计算后的结果");
        return "复杂结果"; // 结果
    }
}

```

输出

任务提交时间 : 2016年8月4日 星期四 15:05:15 EDT
 经过漫长的10秒计算后的结果
 复杂计算的结果是 : 复杂结果
 打印结果时的时间 : 2016年8月4日 星期四 15:05:25 EDT

Future允许的其他操作

虽然 get() 是提取实际结果的方法，Future还提供了

- get(long timeout, TimeUnit unit) 定义当前线程等待结果的最长时间；
- 要取消任务，调用 cancel(mayInterruptIfRunning)。标志 mayInterrupt 表示如果任务已经启动并正在运行，则应中断任务；
- 通过调用isDone()检查任务是否完成/结束；
- 通过调用isCancelled()检查耗时任务是否被取消。

第126.2节 : CountDownLatch (倒计时锁存器)

[CountDownLatch \(倒计时锁存器\)](#)

一种同步辅助工具，允许一个或多个线程等待，直到其他线程中执行的一组操作完成。

1. 一个CountDownLatch通过给定的计数进行初始化。
2. await方法会阻塞，直到当前计数因调用countDown()方法而达到零，之后所有等待的线程被释放，任何后续对await的调用都会立即返回。
3. 这是一次性现象—计数不能重置。如果需要可重置计数的版本，可以考虑使用CyclicBarrier (循环屏障)。

主要方法：

public void await() throws InterruptedException

使当前线程等待，直到计数器减到零，除非线程被中断。

public void countDown()

```

public class ComplexCalculator implements Callable<String> {

    @Override
    public String call() throws Exception {
        // just sleep for 10 secs to simulate a lengthy computation
        Thread.sleep(10000);
        System.out.println("Result after a lengthy 10sec calculation");
        return "Complex Result"; // the result
    }
}

```

Output

Time At Task Submission : Thu Aug 04 15:05:15 EDT 2016
 Result after a lengthy 10sec calculation
 Result of Complex Calculation is : Complex Result
 Time At the Point of Printing the Result : Thu Aug 04 15:05:25 EDT 2016

Other operations permitted on Future

While get() is the method to extract the actual result Future has provision

- get(**long** timeout, TimeUnit unit) defines maximum time period during current thread will wait for a result;
- To cancel the task call cancel(mayInterruptIfRunning). The flag mayInterrupt indicates that task should be interrupted if it was started and is running right now;
- To check if task is completed/finished by calling isDone();
- To check if the lengthy task were cancelled isCancelled().

Section 126.2: CountDownLatch

[CountDownLatch](#)

A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

1. A CountDownLatch is initialized with a given count.
2. The await methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of await return immediately.
3. This is a one-shot phenomenon—the count cannot be reset. If you need a version that resets the count, consider using a CyclicBarrier.

Key Methods:

public void await() throws InterruptedException

Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.

public void countDown()

递减计数器的计数，如果计数达到零，则释放所有等待的线程。

示例：

```
import java.util.concurrent.*;

class DoSomethingInAThread implements Runnable {
    CountDownLatch latch;
    public DoSomethingInAThread(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            System.out.println("做一些事情");
            latch.countDown();
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}

public class CountDownLatchDemo {
    public static void main(String[] args) {
        try {
            int 线程数量 = 5;
            if (args.length < 1) {
                System.out.println("用法：java CountDownLatchDemo 线程数");
                return;
            }
            try {
                numberOfThreads = Integer.parseInt(args[0]);
            } catch(NumberFormatException ne) {
                System.out.println(ne);
            }
            CountDownLatch latch = new CountDownLatch(numberOfThreads);
            for (int n = 0; n < numberOfThreads; n++) {
                Thread t = new Thread(new DoSomethingInAThread(latch));
                t.start();
            }
            latch.await();
            System.out.println("主线程在完成 " + numberOfThreads + " 个线程后");
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}
```

输出：

```
java CountDownLatchDemo 5
做一些事情
做某事
做某事
做某事
做某事
做某事
在主线程中，5个线程完成后
```

Decrements the count of the latch, releasing all waiting threads if the count reaches zero.

Example:

```
import java.util.concurrent.*;

class DoSomethingInAThread implements Runnable {
    CountDownLatch latch;
    public DoSomethingInAThread(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            System.out.println("Do some thing");
            latch.countDown();
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}

public class CountDownLatchDemo {
    public static void main(String[] args) {
        try {
            int number_of_threads = 5;
            if (args.length < 1) {
                System.out.println("Usage: java CountDownLatchDemo number_of_threads");
                return;
            }
            try {
                number_of_threads = Integer.parseInt(args[0]);
            } catch(NumberFormatException ne) {
                System.out.println(ne);
            }
            CountDownLatch latch = new CountDownLatch(number_of_threads);
            for (int n = 0; n < number_of_threads; n++) {
                Thread t = new Thread(new DoSomethingInAThread(latch));
                t.start();
            }
            latch.await();
            System.out.println("In Main thread after completion of " + number_of_threads + " threads");
        } catch(Exception err) {
            err.printStackTrace();
        }
    }
}
```

output:

```
java CountDownLatchDemo 5
Do some thing
Do some thing
Do some thing
Do some thing
Do some thing
In Main thread after completion of 5 threads
```

说明：

Explanation:

1. 主线程中，CountDownLatch 初始化计数器为5
2. 主线程使用await()方法等待。
3. 已创建五个DoSomethingInAThread实例。每个实例都通过countDown()方法递减计数器。
4. 一旦计数器变为零，主线程将继续执行

第126.3节：基本多线程

如果你有许多任务需要执行，且所有这些任务都不依赖于前一个任务的结果，你可以使用**多线程**让计算机同时执行所有这些任务，前提是你的计算机支持使用多个处理器。如果你有一些大型独立任务，这可以使程序执行得更快。

```
class CountAndPrint implements Runnable {
    private final String name;

    CountAndPrint(String name) {
        this.name = name;
    }

    /** This is CountAndPrint will do */
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.println(this.name + ":" + i);
        }
    }

    public static void main(String[] args) {
        // 启动4个并行线程
        for (int i = 1; i <= 4; i++) {
            // `start` 方法将在另一个线程中调用 CountAndPrint 的
            // `run` 方法
            new Thread(new CountAndPrint("Instance " + i)).start();
        }

        // 在主线程中执行其他任务
        for (int i = 0; i < 10000; i++) {
            System.out.println("Main: " + i);
        }
    }
}
```

各种CountAndPrint实例的run方法代码将以不可预测的顺序执行。一个示例执行片段可能如下所示：

```
实例4: 1
实例2: 1
实例4: 2
实例1: 1
实例1: 2
主线程: 1
实例4: 3
主线程: 2
实例3: 1
实例4: 4
...
```

1. CountDownLatch is initialized with a counter of 5 in Main thread
2. Main thread is waiting by using await() method.
3. Five instances of DoSomethingInAThread have been created. Each instance decremented the counter with countDown() method.
4. Once the counter becomes zero, Main thread will resume

Section 126.3: Basic Multithreading

If you have many tasks to execute, and all these tasks are not dependent of the result of the precedent ones, you can use **Multithreading** for your computer to do all this tasks at the same time using more processors if your computer can. This can make your program execution **faster** if you have some big independent tasks.

```
class CountAndPrint implements Runnable {
    private final String name;

    CountAndPrint(String name) {
        this.name = name;
    }

    /** This is what a CountAndPrint will do */
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            System.out.println(this.name + ":" + i);
        }
    }

    public static void main(String[] args) {
        // Launching 4 parallel threads
        for (int i = 1; i <= 4; i++) {
            // `start` method will call the `run` method
            // of CountAndPrint in another thread
            new Thread(new CountAndPrint("Instance " + i)).start();
        }

        // Doing some others tasks in the main Thread
        for (int i = 0; i < 10000; i++) {
            System.out.println("Main: " + i);
        }
    }
}
```

The code of the run method of the various CountAndPrint instances will execute in non predictable order. A snippet of a sample execution might look like this:

```
Instance 4: 1
Instance 2: 1
Instance 4: 2
Instance 1: 1
Instance 1: 2
Main: 1
Instance 4: 3
Main: 2
Instance 3: 1
Instance 4: 4
...
```

第126.4节：作为同步辅助的锁

在Java 5的concurrent包引入之前，线程处理更偏底层。该包的引入提供了若干更高级的并发编程辅助工具/构造。

锁是线程同步机制，本质上与synchronized代码块或关键字的作用相同。

内置锁

```
int count = 0; // 多线程共享

public void doSomething() {
    synchronized (this) {
        ++count; // 非原子操作
    }
}
```

使用锁进行同步

```
int count = 0; // 多线程共享

Lock lockObj = new ReentrantLock();
public void doSomething() {
    try {
        lockObj.lock();
        ++count; // 非原子操作
    } finally {
        lockObj.unlock(); // 确保无论如何都释放锁
    }
}
```

锁还提供了内置锁不具备的功能，例如锁定时仍能响应中断，或者尝试加锁时无法加锁而不阻塞。

响应中断的锁定

```
class Locky {
    int count = 0; // 多个线程共享

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        try {
            try {
                lockObj.lockInterruptibly();
                ++count; // 一个非原子操作
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // 停止
            }
        } finally {
            if (!Thread.currentThread().isInterrupted()) {
                lockObj.unlock(); // 确保无误释放锁
            }
        }
    }
}
```

Section 126.4: Locks as Synchronisation aids

Prior to Java 5's concurrent package introduction threading was more low level. The introduction of this package provided several higher level concurrent programming aids/constructs.

Locks are thread synchronisation mechanisms that essentially serve the same purpose as synchronized blocks or key words.

Intrinsic Locking

```
int count = 0; // shared among multiple threads

public void doSomething() {
    synchronized(this) {
        ++count; // a non-atomic operation
    }
}
```

Synchronisation using Locks

```
int count = 0; // shared among multiple threads

Lock lockObj = new ReentrantLock();
public void doSomething() {
    try {
        lockObj.lock();
        ++count; // a non-atomic operation
    } finally {
        lockObj.unlock(); // sure to release the lock without fail
    }
}
```

Locks also have functionality available that intrinsic locking does not offer, such as locking but remaining responsive to interruption, or trying to lock, and not block when unable to.

Locking, responsive to interruption

```
class Locky {
    int count = 0; // shared among multiple threads

    Lock lockObj = new ReentrantLock();

    public void doSomething() {
        try {
            try {
                lockObj.lockInterruptibly();
                ++count; // a non-atomic operation
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt(); // stopping
            }
        } finally {
            if (!Thread.currentThread().isInterrupted()) {
                lockObj.unlock(); // sure to release the lock without fail
            }
        }
    }
}
```

只有在能够获取锁时才执行操作

```
public class Locky2 {  
    int count = 0; // 多个线程共享  
  
    Lock lockObj = new ReentrantLock();  
  
    public void doSomething() {  
        boolean locked = lockObj.tryLock(); // 成功获取锁时返回true  
        if (locked) {  
            try {  
                ++count; // 一个非原子操作  
            } finally {  
                lockObj.unlock(); // 确保锁一定被释放  
            }  
        }  
    }  
}
```

有几种不同的锁可用。更多详情请参阅API文档 [here](#)

第126.5节：信号量

信号量是一种高级同步器，维护一组许可，线程可以获取和释放这些许可。信号量可以被想象成一个许可计数器，当线程获取许可时计数器递减，线程释放许可时计数器递增。如果线程尝试获取许可时许可数量为0，则该线程将阻塞，直到有许可可用（或线程被中断）。

信号量的初始化方式如下：

```
Semaphore semaphore = new Semaphore(1); // 整数值表示许可数量
```

Semaphore构造函数接受一个额外的布尔参数用于公平性。当设置为false时，该类不保证线程获取许可的顺序。当公平性设置为true时，信号量保证调用任何acquire方法的线程按照调用顺序获得许可。声明方式如下：

```
Semaphore semaphore = new Semaphore(1, true);
```

现在让我们看一个来自javadocs的示例，其中Semaphore用于控制对一组资源池的访问。该示例中使用Semaphore提供阻塞功能，以确保在调用getItem()时总有资源可用。

```
class Pool {  
    /*  
     * 请注意，这并不限制可能释放的数量！  
     * 这只是信号量的起始值，除非你在  
     * getNextAvailableItem() 和 markAsUnused() 方法中强制执行，否则没有其他重要意义  
     */  
    private static final int MAX_AVAILABLE = 100;  
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);  
  
    /**  
     * 获取下一个可用项，并将许可数量减少1。  
     * 如果没有可用项，则阻塞。  
     */  
    public Object getItem() throws InterruptedException {
```

Only do something when able to lock

```
public class Locky2 {  
    int count = 0; // shared among multiple threads  
  
    Lock lockObj = new ReentrantLock();  
  
    public void doSomething() {  
        boolean locked = lockObj.tryLock(); // returns true upon successful lock  
        if (locked) {  
            try {  
                ++count; // a non-atomic operation  
            } finally {  
                lockObj.unlock(); // sure to release the lock without fail  
            }  
        }  
    }  
}
```

There are several variants of lock available. For more details refer the api docs [here](#)

Section 126.5: Semaphore

A Semaphore is a high-level synchronizer that maintains a set of *permits* that can be acquired and released by threads. A Semaphore can be imagined as a counter of *permits* that will be decremented when a thread acquires, and incremented when a thread releases. If the amount of *permits* is 0 when a thread attempts to acquire, then the thread will block until a permit is made available (or until the thread is interrupted).

A semaphore is initialized as:

```
Semaphore semaphore = new Semaphore(1); // The int value being the number of permits
```

The Semaphore constructor accepts an additional boolean parameter for fairness. When set false, this class makes no guarantees about the order in which threads acquire permits. When fairness is set true, the semaphore guarantees that threads invoking any of the acquire methods are selected to obtain permits in the order in which their invocation of those methods was processed. It is declared in the following manner:

```
Semaphore semaphore = new Semaphore(1, true);
```

Now let's look at an example from javadocs, where Semaphore is used to control access to a pool of items. A Semaphore is used in this example to provide blocking functionality in order to ensure that there are always items to be obtained when getItem() is called.

```
class Pool {  
    /*  
     * Note that this DOES NOT bound the amount that may be released!  
     * This is only a starting value for the Semaphore and has no other  
     * significant meaning UNLESS you enforce this inside of the  
     * getNextAvailableItem() and markAsUnused() methods  
     */  
    private static final int MAX_AVAILABLE = 100;  
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);  
  
    /**  
     * Obtains the next available item and reduces the permit count by 1.  
     * If there are no items available, block.  
     */  
    public Object getItem() throws InterruptedException {
```

```

available.acquire();
    return getNextAvailableItem();
}

/**
 * 将该项放回池中，并增加1个许可。
 */
public void putItem(Object x) {
    if (markAsUnused(x))
available.release();
}

private Object getNextAvailableItem() {
    // Implementation
}

private boolean markAsUnused(Object o) {
    // Implementation
}
}

```

第126.6节：同步

在Java中，有一个内置的语言级锁机制：**synchronized**块，它可以使用任何Java对象作为内置锁（即每个Java对象都可能有一个与之关联的监视器）。

内置锁为一组语句提供原子性。为了理解这对我们意味着什么，让我们看一个 **synchronized**有用的示例：

```

private static int t = 0;
private static Object mutex = new Object();

public static void main(String[] args) {
ExecutorService executorService = Executors.newFixedThreadPool(400); // 高线程数
是为了演示目的。
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            synchronized (mutex) {
t++;
                System.out.println(MessageFormat.format("t: {0}", t));
            }
        });
    }
executorService.shutdown();
}

```

在这种情况下，如果没有 **synchronized**代码块，将会出现多个并发问题。

第一个问题是后置递增运算符（它本身不是原子操作），第二个问题是可能会在任意数量的其他线程修改t之后观察到t的值。

然而，由于我们获得了一个内置锁，这里不会有竞态条件，输出将包含从1到100的数字，且顺序正常。

Java中的内置锁是互斥锁（即互斥执行锁）。互斥执行意味着如果一个线程已经获得了锁，第二个线程将被迫等待第一个线程释放锁后才能获得锁。

注意：可能使线程进入等待（休眠）状态的操作称为阻塞操作。因此，获取锁是一种阻塞操作。

Java中的内置锁是可重入的。这意味着如果一个线程尝试获取它已经拥有的锁，它不会

```

available.acquire();
    return getNextAvailableItem();
}

/**
 * Puts the item into the pool and add 1 permit.
 */
public void putItem(Object x) {
    if (markAsUnused(x))
        available.release();
}

private Object getNextAvailableItem() {
    // Implementation
}

private boolean markAsUnused(Object o) {
    // Implementation
}
}

```

Section 126.6: Synchronization

In Java, there is a built-in language-level locking mechanism: the **synchronized** block, which can use any Java object as an intrinsic lock (i.e. every Java object may have a monitor associated with it).

Intrinsic locks provide atomicity to groups of statements. To understand what that means for us, let's have a look at an example where **synchronized** is useful:

```

private static int t = 0;
private static Object mutex = new Object();

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread count
is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            synchronized (mutex) {
t++;
                System.out.println(MessageFormat.format("t: {0}", t));
            }
        });
    }
executorService.shutdown();
}

```

In this case, if it weren't for the **synchronized** block, there would have been multiple concurrency issues involved. The first one would be with the post increment operator (it isn't atomic in itself), and the second would be that we would be observing the value of t after an arbitrary amount of other threads has had the chance to modify it. However, since we acquired an intrinsic lock, there will be no race conditions here and the output will contain numbers from 1 to 100 in their normal order.

Intrinsic locks in Java are *mutexes* (i.e. mutual execution locks). Mutual execution means that if one thread has acquired the lock, the second will be forced to wait for the first one to release it before it can acquire the lock for itself. Note: An operation that may put the thread into the wait (sleep) state is called a *blocking operation*. Thus, acquiring a lock is a blocking operation.

Intrinsic locks in Java are *reentrant*. This means that if a thread attempts to acquire a lock it already owns, it will not

代码块并且它将成功获取它。例如，以下代码在调用时不会阻塞：

```
public void bar(){  
    synchronized(this){  
        ...  
    }  
}  
public void foo(){  
    synchronized(this){  
        bar();  
    }  
}
```

除了 `synchronized`代码块，还有 `synchronized`方法。

以下代码块实际上是等价的（尽管字节码看起来不同）：

1. 对 `this`的 `synchronized`代码块：

```
public void foo() {  
    synchronized(this) {  
        doStuff();  
    }  
}
```

2. 同步方法：

```
public synchronized void foo() {  
    doStuff();  
}
```

对于 `static`方法，同样是：

```
class MyClass {  
    ...  
    public static void bar() {  
        同步(MyClass.class) {  
            doSomeOtherStuff();  
        }  
    }  
}
```

具有与此相同的效果：

```
class MyClass {  
    ...  
    public static synchronized void bar() {  
        doSomeOtherStuff();  
    }  
}
```

第126.7节：Runnable对象

`Runnable`接口定义了一个单一方法，`run()`，用于包含线程中执行的代码。

`Runnable`对象被传递给`Thread`构造函数。然后调用`Thread`的`start()`方法。

block and it will successfully acquire it. For instance, the following code will *not* block when called:

```
public void bar(){  
    synchronized(this){  
        ...  
    }  
}  
public void foo(){  
    synchronized(this){  
        bar();  
    }  
}
```

Beside `synchronized` blocks, there are also `synchronized` methods.

The following blocks of code are practically equivalent (even though the bytecode seems to be different):

1. `synchronized` block on `this`:

```
public void foo() {  
    synchronized(this) {  
        doStuff();  
    }  
}
```

2. `synchronized` method:

```
public synchronized void foo() {  
    doStuff();  
}
```

Likewise for `static` methods, this:

```
class MyClass {  
    ...  
    public static void bar() {  
        synchronized(MyClass.class) {  
            doSomeOtherStuff();  
        }  
    }  
}
```

has the same effect as this:

```
class MyClass {  
    ...  
    public static synchronized void bar() {  
        doSomeOtherStuff();  
    }  
}
```

Section 126.7: Runnable Object

The `Runnable` interface defines a single method, `run()`, meant to contain the code executed in the thread.

The `Runnable` object is passed to the `Thread` constructor. And Thread's `start()` method is called.

示例

```
public class HelloRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("来自线程的问候");  
    }  
  
    public static void main(String[] args) {  
        new Thread(new HelloRunnable()).start();  
    }  
}
```

Java8 示例：

```
public static void main(String[] args) {  
    Runnable r = () -> System.out.println("Hello world");  
    new Thread(r).start();  
}
```

Runnable 与 Thread 子类

使用 `Runnable` 对象更为通用，因为 `Runnable` 对象可以继承除

`Thread` 以外的其他类。

`Thread` 子类在简单应用中更易使用，但受限于任务类必须是 `Thread` 的子类这一事实。

`Runnable` 对象适用于高级线程管理 API。

第126.8节：创建基本死锁系统

当两个竞争操作相互等待对方完成时，就会发生死锁，因此两者都无法完成。在 Java 中，每个对象都关联一个锁。为了避免多个线程对单个对象的并发修改，我们可以使用 `synchronized` 关键字，但一切都有代价。错误使用 `synchronized` 关键字可能导致系统卡死，称为死锁系统。

假设有两个线程操作同一个实例，称这两个线程为 First 和 Second，且有两个资源 R1 和 R2。First 获取了 R1，同时需要 R2 才能完成，而 Second 获取了 R2，需要 R1 才能完成。

所以假设在时间 t=0，

第一个持有R1，第二个持有R2。现在第一个在等待R2，而第二个在等待R1。这种等待是无限期的，这导致了死锁。

```
public class Example2 {  
  
    public static void main(String[] args) throws InterruptedException {  
        final DeadLock dl = new DeadLock();  
        Thread t1 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                // TODO 自动生成的方法存根  
                dl.methodA();  
            }  
        });  
        t1.start();  
  
        Thread t2 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                // TODO 自动生成的方法存根  
                dl.methodB();  
            }  
        });  
        t2.start();  
    }  
}
```

Example

```
public class HelloRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("Hello from a thread");  
    }  
  
    public static void main(String[] args) {  
        new Thread(new HelloRunnable()).start();  
    }  
}
```

Example in Java8:

```
public static void main(String[] args) {  
    Runnable r = () -> System.out.println("Hello world");  
    new Thread(r).start();  
}
```

Runnable vs Thread subclass

A `Runnable` object employment is more general, because the `Runnable` object can subclass a class other than `Thread`.

`Thread` subclassing is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of `Thread`.

A `Runnable` object is applicable to the high-level thread management APIs.

Section 126.8: Creating basic deadlocked system

A deadlock occurs when two competing actions wait for the other to finish, and thus neither ever does. In java there is one lock associated with each object. To avoid concurrent modification done by multiple threads on single object we can use `synchronized` keyword, but everything comes at a cost. Using `synchronized` keyword wrongly can lead to stuck systems called as deadlocked system.

Consider there are 2 threads working on 1 instance, Lets call threads as First and Second, and lets say we have 2 resources R1 and R2. First acquires R1 and also needs R2 for its completion while Second acquires R2 and needs R1 for completion.

so say at time t=0,

First has R1 and Second has R2. now First is waiting for R2 while Second is waiting for R1. this wait is indefinite and this leads to deadlock.

```
public class Example2 {  
  
    public static void main(String[] args) throws InterruptedException {  
        final DeadLock dl = new DeadLock();  
        Thread t1 = new Thread(new Runnable() {  
  
            @Override  
            public void run() {  
                // TODO Auto-generated method stub  
                dl.methodA();  
            }  
        });  
        t1.start();  
  
        Thread t2 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                // TODO Auto-generated method stub  
                dl.methodB();  
            }  
        });  
        t2.start();  
    }  
}
```

```

    }

    Thread t2 = new Thread(new Runnable() {

        @Override
        public void run() {
            // TODO 自动生成的方法存根
            try {
                dl.method2();
            } catch (InterruptedException e) {
                // TODO 自动生成的异常捕获块
            }
            e.printStackTrace();
        }
    });

    t1.setName("First");
    t2.setName("Second");
    t1.start();
    t2.start();
}

class 死锁 {

    Object mLock1 = new Object();
    Object mLock2 = new Object();

    public void 方法A() {
        System.out.println("方法A 等待 mLock1 " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("方法A 获得 mLock1 " + Thread.currentThread().getName());
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // TODO 自动生成的异常捕获块
            }
            e.printStackTrace();
        }
    }

    public void method2() throws InterruptedException {
        System.out.println("method2 等待 mLock2 " + Thread.currentThread().getName());
        synchronized (mLock2) {
            System.out.println("method2 mLock2 已获取 " + Thread.currentThread().getName());
            Thread.sleep(100);
        }
    }

    public void method3() throws InterruptedException {
        System.out.println("method3 mLock1 " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("method3 mLock1 已获取 " + Thread.currentThread().getName());
        }
    }
}

```

该程序的输出：

methodA 等待 mLock1 First

```

    }

    Thread t2 = new Thread(new Runnable() {

        @Override
        public void run() {
            // TODO Auto-generated method stub
            try {
                dl.method2();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    });

    t1.setName("First");
    t2.setName("Second");
    t1.start();
    t2.start();
}

class DeadLock {

    Object mLock1 = new Object();
    Object mLock2 = new Object();

    public void methodA() {
        System.out.println("methodA wait for mLock1 " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("methodA mLock1 acquired " + Thread.currentThread().getName());
            try {
                Thread.sleep(100);
                method2();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }

    public void method2() throws InterruptedException {
        System.out.println("method2 wait for mLock2 " + Thread.currentThread().getName());
        synchronized (mLock2) {
            System.out.println("method2 mLock2 acquired " + Thread.currentThread().getName());
            Thread.sleep(100);
            method3();
        }
    }

    public void method3() throws InterruptedException {
        System.out.println("method3 mLock1 " + Thread.currentThread().getName());
        synchronized (mLock1) {
            System.out.println("method3 mLock1 acquired " + Thread.currentThread().getName());
        }
    }
}

```

Output of this program:

methodA wait for mLock1 First

```
method2 等待 mLock2 Second  
method2 mLock2 已获取 Second  
methodA mLock1 已获取 First  
method3 mLock1 Second  
method2 等待 mLock2 First
```

第126.9节：创建一个 java.lang.Thread 实例

在Java中创建线程主要有两种方法。本质上，创建线程就像编写将在其中执行的代码一样简单。这两种方法的区别在于代码的定义位置。

在Java中，线程由一个对象表示——即java.lang.Thread类或其子类的实例。因此，第一种方法是创建孩子类并重写 run() 方法。

注意：我将使用Thread来指代java.lang.Thread类，使用thread来指代线程的逻辑概念。

```
class MyThread extends Thread {  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println("线程运行中！");  
        }  
    }  
}
```

既然我们已经定义了要执行的代码，线程就可以简单地创建为：

```
MyThread t = new MyThread();
```

Thread类还包含一个接受字符串的构造函数，该字符串将用作线程的名称。在调试多线程程序时，这尤其有用。

```
class MyThread extends Thread {  
    public MyThread(String name) {  
        super(name);  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println("线程运行中！");  
        }  
    }  
}
```

```
MyThread t = new MyThread("问候生产者");
```

第二种方法是使用java.lang.Runnable接口及其唯一方法 run() 来定义代码。然后，Thread类允许你在一个独立的线程中执行该方法。为此，使用接受Runnable接口实例的构造函数来创建线程。

```
Thread t = new Thread(aRunnable);
```

当与lambda表达式或方法引用（仅限Java 8）结合使用时，这种方法非常强大：

```
Thread t = new Thread(operator::hardWork);
```

```
method2 wait for mLock2 Second  
method2 mLock2 acquired Second  
methodA mLock1 acquired First  
method3 mLock1 Second  
method2 wait for mLock2 First
```

Section 126.9: Creating a java.lang.Thread instance

There are two main approaches to creating a thread in Java. In essence, creating a thread is as easy as writing the code that will be executed in it. The two approaches differ in where you define that code.

In Java, a thread is represented by an object - an instance of [java.lang.Thread](#) or its subclass. So the first approach is to create that subclass and override the `run()` method.

Note: I'll use `Thread` to refer to the [java.lang.Thread](#) class and `thread` to refer to the logical concept of threads.

```
class MyThread extends Thread {  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println("Thread running!");  
        }  
    }  
}
```

Now since we've already defined the code to be executed, the thread can be created simply as:

```
MyThread t = new MyThread();
```

The `Thread` class also contains a constructor accepting a string, which will be used as the thread's name. This can be particularly useful when debugging a multi thread program.

```
class MyThread extends Thread {  
    public MyThread(String name) {  
        super(name);  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println("Thread running!");  
        }  
    }  
}
```

```
MyThread t = new MyThread("Greeting Producer");
```

The second approach is to define the code using [java.lang.Runnable](#) and its only method `run()`. The `Thread` class then allows you to execute that method in a separated thread. To achieve this, create the thread using a constructor accepting an instance of the [Runnable](#) interface.

```
Thread t = new Thread(aRunnable);
```

This can be very powerful when combined with lambdas or methods references (Java 8 only):

```
Thread t = new Thread(operator::hardWork);
```

你也可以指定线程的名称。

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

实际上，你可以放心地使用这两种方法。不过，一般的建议是使用后者。

对于上述四个构造函数中的每一个，也有一个接受
[java.lang.ThreadGroup](#)实例作为第一个参数的替代构造函数。

```
ThreadGroup tg = new ThreadGroup("Operators");
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

线程组 (ThreadGroup) 表示一组线程。你只能通过线程 (Thread) 的构造函数将线程添加到线程组中。线程组
可以用来统一管理其所有线程，同时线程也可以从其线程组中获取信息。

总结一下，线程 (Thread) 可以通过以下公共构造函数之一创建：

```
Thread()
Thread(String name)
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

最后一个构造函数允许我们为新线程定义所需的栈大小。

当创建和配置许多具有相同属性或相同模式的线程时，代码的可读性往往会下降。这时可以使用 `java.util.concurrent.ThreadFactory` 接口。该接口允许你通过工厂模式封装线程的创建过程，其唯一方法是 `newThread(Runnable)`。

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

第126.10节：原子操作

原子操作是一种“全部一次性”执行的操作，在原子操作执行期间，其他线程无法观察或修改状态。

让我们考虑一个错误示例。

```
private static int t = 0;

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // 高线程数
    // 是为了演示目的。
    for (int i = 0; i < 100; i++) {
```

You can specify the thread's name, too.

```
Thread t = new Thread(operator::hardWork, "Pi operator");
```

Practically speaking, you can use both approaches without worries. However the [general wisdom](#) says to use the latter.

For every of the four mentioned constructors, there is also an alternative accepting an instance of [java.lang.ThreadGroup](#) as the first parameter.

```
ThreadGroup tg = new ThreadGroup("Operators");
Thread t = new Thread(tg, operator::hardWork, "PI operator");
```

The `ThreadGroup` represents a set of threads. You can only add a `Thread` to a `ThreadGroup` using a `Thread`'s constructor. The `ThreadGroup` can then be used to manage all its `Threads` together, as well as the `Thread` can gain information from its `ThreadGroup`.

So to summarize, the `Thread` can be created with one of these public constructors:

```
Thread()
Thread(String name)
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

The last one allows us to define desired stack size for the new thread.

Often the code readability suffers when creating and configuring many Threads with same properties or from the same pattern. That's when `java.util.concurrent.ThreadFactory` can be used. This interface allows you to encapsulate the procedure of creating the thread through the factory pattern and its only method `newThread(Runnable)`.

```
class WorkerFactory implements ThreadFactory {
    private int id = 0;

    @Override
    public Thread newThread(Runnable r) {
        return new Thread(r, "Worker " + id++);
    }
}
```

Section 126.10: Atomic operations

An atomic operation is an operation that is executed “all at once”, without any chance of other threads observing or modifying state during the atomic operation's execution.

Lets consider a **BAD EXAMPLE**.

```
private static int t = 0;

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread count
    // is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
```

```

executorService.execute(() -> {
    t++;
    System.out.println(MessageFormat.format("t: {0}", t));
});
}
executorService.shutdown();
}

```

在这种情况下，有两个问题。第一个问题是后置递增运算符不是原子的。它由多个操作组成：获取值、值加1、设置值。这就是为什么如果运行该示例，很可能不会在输出中看到 t: 100——两个线程可能会同时获取值、递增并设置值：假设t的值是10，有两个线程在递增t。两个线程都会将t的值设置为11，因为第二个线程在第一个线程完成递增之前观察到了t的值。

第二个问题是观察t的方式。当我们打印t的值时，该值可能已经被其他线程在本线程的递增操作之后更改。

为了解决这些问题，我们将使用`java.util.concurrent.atomic.AtomicInteger`，它为我们提供了许多原子操作可用。

```

private static AtomicInteger t = new AtomicInteger(0);

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // 高线程数
    // 是为了演示目的。
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            int currentT = t.incrementAndGet();
            System.out.println(MessageFormat.format("t: {0}", currentT));
        });
    }
    executorService.shutdown();
}

```

`AtomicInteger` 的 `incrementAndGet` 方法会原子性地递增并返回新值，从而消除了之前的竞态条件。请注意，在此示例中，输出行仍然可能是无序的，因为我们没有对 `println` 调用进行排序，这超出了本示例的范围，因为那需要同步，而本示例的目的是展示如何使用 `AtomicInteger` 来消除与状态相关的竞态条件。

第126.11节：独占写入 / 并发读取访问

有时需要一个进程同时对相同的“数据”进行写入和读取。

`ReadWriteLock` 接口及其 `ReentrantReadWriteLock` 实现允许如下描述的访问模式：

1. 数据可以有任意数量的并发读取者。如果至少有一个读取者获得访问权限，则不允许写入者访问。
2. 数据最多只能有一个写入者。如果写入者获得访问权限，则不允许读取者访问数据。

一个实现示例如下：

```

import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

```

```

executorService.execute(() -> {
    t++;
    System.out.println(MessageFormat.format("t: {0}", t));
});
}
executorService.shutdown();
}

```

In this case, there are two issues. The first issue is that the post increment operator is *not* atomic. It is comprised of multiple operations: get the value, add 1 to the value, set the value. That's why if we run the example, it is likely that we won't see t: 100 in the output - two threads may concurrently get the value, increment it, and set it: let's say the value of t is 10, and two threads are incrementing t. Both threads will set the value of t to 11, since the second thread observes the value of t before the first thread had finished incrementing it.

The second issue is with how we are observing t. When we are printing the value of t, the value may have already been changed by a different thread after this thread's increment operation.

To fix those issues, we'll use the `java.util.concurrent.atomic.AtomicInteger`, which has many atomic operations for us to use.

```

private static AtomicInteger t = new AtomicInteger(0);

public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(400); // The high thread count
    // is for demonstration purposes.
    for (int i = 0; i < 100; i++) {
        executorService.execute(() -> {
            int currentT = t.incrementAndGet();
            System.out.println(MessageFormat.format("t: {0}", currentT));
        });
    }
    executorService.shutdown();
}

```

The `incrementAndGet` method of `AtomicInteger` atomically increments and returns the new value, thus eliminating the previous race condition. Please note that in this example the lines will still be out of order because we make no effort to sequence the `println` calls and that this falls outside the scope of this example, since it would require synchronization and the goal of this example is to show how to use `AtomicInteger` to eliminate race conditions concerning state.

Section 126.11: Exclusive write / Concurrent read access

It is sometimes required for a process to concurrently write and read the same "data".

The `ReadWriteLock` interface, and its `ReentrantReadWriteLock` implementation allows for an access pattern that can be described as follow :

1. There can be any number of concurrent readers of the data. If there is at least one reader access granted, then no writer access is possible.
2. There can be at most one single writer to the data. If there is a writer access granted, then no reader can access the data.

An implementation could look like :

```

import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

```

```

public class Sample {

    // 我们的锁。构造函数允许设置“公平性”，保证锁分配的时间顺序。
    protected static final ReadWriteLock RW_LOCK = new ReentrantReadWriteLock();

    // 这是一个典型的数据，需要保护以防止并发访问
    protected static int data = 0;

    /** 这将以独占访问方式写入数据 */
    public static void writeToData() {
        RW_LOCK.writeLock().lock();
        try {
            data++;
        } finally {
            RW_LOCK.writeLock().unlock();
        }
    }

    public static int readData() {
        RW_LOCK.readLock().lock();
        try {
            return data;
        } finally {
            RW_LOCK.readLock().unlock();
        }
    }
}

```

注释 1：这个具体用例使用AtomicInteger有更简洁的解决方案，但这里描述的是一种访问模式，无论这里的数据是否是Atomic变体的整数，该模式都适用。

注释 2：读取部分的锁确实是必要的，尽管对普通读者来说可能看不出来。实际上，如果不锁住读取端，可能会出现多种问题，其中包括：

1. 对于所有JVM，基本类型值的写入不保证是原子的，因此如果data是64位长整型，读取端可能只看到64位写入中的32位
2. JVM只保证在写入线程和读取线程之间建立了先行发生关系（Happen Before）时，写入的可见性才得以保证。该关系在读写双方都使用各自的锁时建立，否则不成立

版本 ≥ Java SE 8

如果需要更高的性能，并且在某些使用场景下，有一种更快的锁类型可用，称为StampedLock，它实现了乐观锁模式。该锁的工作方式与ReadWriteLock截然不同，本示例不可直接套用。

第126.12节：生产者-消费者

生产者-消费者问题的一个简单示例。注意这里使用了JDK类（AtomicBoolean和BlockingQueue）进行同步，这减少了创建无效解决方案的可能性。请查阅Javadoc了解各种类型的BlockingQueue；选择不同的实现可能会极大地改变该示例的行为（如DelayQueue或PriorityQueue）。

```
public class Producer implements Runnable {
```

```

public class Sample {

    // Our lock. The constructor allows a "fairness" setting, which guarantees the chronology of lock
    // attributions.
    protected static final ReadWriteLock RW_LOCK = new ReentrantReadWriteLock();

    // This is a typical data that needs to be protected for concurrent access
    protected static int data = 0;

    /** This will write to the data, in an exclusive access */
    public static void writeToData() {
        RW_LOCK.writeLock().lock();
        try {
            data++;
        } finally {
            RW_LOCK.writeLock().unlock();
        }
    }

    public static int readData() {
        RW_LOCK.readLock().lock();
        try {
            return data;
        } finally {
            RW_LOCK.readLock().unlock();
        }
    }
}

```

NOTE 1 : This precise use case has a cleaner solution using AtomicInteger, but what is described here is an access pattern, that works regardless of the fact that data here is an integer that as an Atomic variant.

NOTE 2 : The lock on the reading part is really needed, although it might not look so to the casual reader. Indeed, if you do not lock on the reader side, any number of things can go wrong, amongst which :

1. The writes of primitive values are not guaranteed to be atomic on all JVMs, so the reader could see e.g. only 32bits of a 64bits write if data were a 64bits long type
2. The visibility of the write from a thread that did not perform it is guaranteed by the JVM only if we establish *Happen Before relationship* between the writes and the reads. This relationship is established when both readers and writers use their respective locks, but not otherwise

Version ≥ Java SE 8

In case higher performance is required, and under certain types of usage, there is a faster lock type available, called the StampedLock, that amongst other things implements an optimistic lock mode. This lock works very differently from the ReadWriteLock, and this sample is not transposable.

Section 126.12: Producer-Consumer

A simple example of producer-consumer problem solution. Notice that JDK classes (AtomicBoolean and BlockingQueue) are used for synchronization, which reduces the chance of creating an invalid solution. Consult Javadoc for various types of [BlockingQueue](#); choosing different implementation may drastically change the behavior of this example (like [DelayQueue](#) or [Priority Queue](#)).

```
public class Producer implements Runnable {
```

```

private final BlockingQueue<ProducedData> 队列;

public 生产者(BlockingQueue<ProducedData> 队列) {
    this.队列 = 队列;
}

public void run() {
    int producedCount = 0;
    try {
        while (true) {
producedCount++;
            //当线程被中断时, put会抛出InterruptedException
            queue.put(new ProducedData());
        }
    } catch (InterruptedException e) {
        //线程已被中断:清理并退出
        producedCount--;
        //如果上层需要中断标志,则重新中断线程
        Thread.currentThread().interrupt();
    }
    System.out.println("Produced " + producedCount + " objects");
}
}

public class Consumer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

    public Consumer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int consumedCount = 0;
        try {
            while (true) {
                //当线程被中断时, put会抛出InterruptedException
                ProducedData data = queue.poll(10, TimeUnit.MILLISECONDS);
                // 处理数据
consumedCount++;
            }
        } catch (InterruptedException e) {
            // 线程已被中断:清理并退出
            consumedCount--;
            //如果上层需要中断标志,则重新中断线程
            Thread.currentThread().interrupt();
        }
        System.out.println("已消费 " + consumedCount + " 个对象");
    }
}

public class 生产者消费者示例 {
    static class 生产数据 {
        // 空数据对象
    }

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<生产数据> queue = new ArrayBlockingQueue<生产数据>(1000);
        // 队列的选择决定实际行为:参见各种 BlockingQueue 实现

        Thread 生产者 = new Thread(new 生产者(queue));
    }
}

```

```

private final BlockingQueue<ProducedData> queue;

public Producer(BlockingQueue<ProducedData> queue) {
    this.queue = queue;
}

public void run() {
    int producedCount = 0;
    try {
        while (true) {
            producedCount++;
            //put throws an InterruptedException when the thread is interrupted
            queue.put(new ProducedData());
        }
    } catch (InterruptedException e) {
        // the thread has been interrupted: cleanup and exit
        producedCount--;
        //re-interrupt the thread in case the interrupt flag is needed higher up
        Thread.currentThread().interrupt();
    }
    System.out.println("Produced " + producedCount + " objects");
}
}

public class Consumer implements Runnable {

    private final BlockingQueue<ProducedData> queue;

    public Consumer(BlockingQueue<ProducedData> queue) {
        this.queue = queue;
    }

    public void run() {
        int consumedCount = 0;
        try {
            while (true) {
                //put throws an InterruptedException when the thread is interrupted
                ProducedData data = queue.poll(10, TimeUnit.MILLISECONDS);
                // process data
                consumedCount++;
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
            consumedCount--;
            //re-interrupt the thread in case the interrupt flag is needed higher up
            Thread.currentThread().interrupt();
        }
        System.out.println("Consumed " + consumedCount + " objects");
    }
}

public class ProducerConsumerExample {
    static class ProducedData {
        // empty data object
    }

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<ProducedData> queue = new ArrayBlockingQueue<ProducedData>(1000);
        // choice of queue determines the actual behavior: see various BlockingQueue implementations

        Thread producer = new Thread(new Producer(queue));
    }
}

```

```

    Thread 消费者 = new Thread(new 消费者(queue));
    生产者.start();
    消费者.start();

    Thread.sleep(1000);
    生产者.interrupt();
    Thread.sleep(10);
    消费者.interrupt();
}
}

```

第126.13节：使用同步 (synchronized) /易变 (volatile) 时可视化读/写屏障

众所周知，我们应该使用 `synchronized`关键字来使方法或代码块的执行具有排他性。但我们中很少有人可能不了解使用 `synchronized`和 `volatile`关键字的另一个重要方面：除了使一段代码成为原子操作外，它还提供了读/写屏障。什么是读/写屏障？让我们通过一个例子来讨论这个问题：

```

类 Counter {
    私有 Integer count = 10;

    公共同步 void incrementCount() {
        count++;
    }

    公共 Integer getCount() {
        返回 count;
    }
}

```

假设线程 A先调用 `incrementCount()`，然后另一个线程 B调用 `getCount()`。在这种情况下，不能保证线程B会看到更新后的 `count`值。它可能仍然看到 `count`为10，甚至有可能永远看不到 `count`的更新值。

要理解这种行为，我们需要了解Java内存模型如何与硬件架构集成。在Java中，每个线程都有自己的线程栈。该栈包含：方法调用栈和该线程中创建的局部变量。在多核系统中，很可能两个线程同时在不同的核心上运行。在这种情况下，线程栈的一部分可能位于核心的寄存器/缓存中。如果在线程内部，使用 `synchronized`（或 `volatile`）关键字访问对象，那么在 `synchronized`代码块之后，该线程会将该变量的本地副本与主内存同步。这就形成了读/写屏障，确保线程看到该对象的最新值。

但在我们的例子中，由于线程B没有使用同步访问 `count`，它可能引用存储在寄存器中的 `count`值，并且可能永远看不到线程A的更新。为了确保线程B看到 `count`的最新值，我们需要将 `getCount()`也声明为同步方法。

```

    公共同步 Integer getCount() {
        返回 count;
    }
}

```

现在，当线程A完成对 `count`的更新后，它会释放 `Counter`实例的锁，同时创建写屏障并将该代码块内的所有更改刷新到主内存。类似地，当线程B获取同一个 `Counter`实例的锁时，它进入读屏障，从主内存读取 `count`的值，并看到所有

```

    Thread consumer = new Thread(new Consumer(queue));
    producer.start();
    consumer.start();

    Thread.sleep(1000);
    producer.interrupt();
    Thread.sleep(10);
    consumer.interrupt();
}
}

```

Section 126.13: Visualizing read/write barriers while using synchronized / volatile

As we know that we should use `synchronized` keyword to make execution of a method or block exclusive. But few of us may not be aware of one more important aspect of using `synchronized` and `volatile` keyword: *apart from making a unit of code atomic, it also provides read / write barrier*. What is this read / write barrier? Let's discuss this using an example:

```

class Counter {
    private Integer count = 10;

    public synchronized void incrementCount() {
        count++;
    }

    public Integer getCount() {
        return count;
    }
}

```

Let's suppose a thread A calls `incrementCount()` first then another thread B calls `getCount()`. In this scenario there is no guarantee that B will see updated value of `count`. It may still see `count` as 10, even it is also possible that it never sees updated value of `count` ever.

To understand this behavior we need to understand how Java memory model integrates with hardware architecture. In Java, each thread has its own thread stack. This stack contains: method call stack and local variable created in that thread. In a multi core system, it is quite possible that two threads are running concurrently in separate cores. In such scenario it is possible that part of a thread's stack lies inside register / cache of a core. If inside a thread, an object is accessed using `synchronized` (or `volatile`) keyword, after `synchronized` block that thread syncs its local copy of that variable with the main memory. This creates a read / write barrier and makes sure that the thread is seeing the latest value of that object.

But in our case, since thread B has not used synchronized access to `count`, it might be referring value of `count` stored in register and may never see updates from thread A. To make sure that B sees latest value of `count` we need to make `getCount()` synchronized as well.

```

public synchronized Integer getCount() {
    return count;
}

```

Now when thread A is done with updating `count` it unlocks `Counter` instance, at the same time creates write barrier and flushes all changes done inside that block to the main memory. Similarly when thread B acquires lock on the same instance of `Counter`, it enters into read barrier and reads value of `count` from main memory and sees all

更新。

Thread A Acquire lock

Increment 'count'

Release lock

Flush everything to
main memory

Updates its local copy
with main memory

Acquire lock

Thread B

Read 'count'

Release lock

updates.

Thread A Acquire lock

Increment 'count'

Release lock

Flush everything to
main memory

Updates its local copy
with main memory

Acquire lock

Thread B

Read 'count'

Release lock

相同的可见性效果也适用于**volatile**读/写。所有在写入**volatile**之前更新的变量将被刷新到主内存，且所有在读取**volatile**变量之后的读取都将来自主内存。

第126.14节：获取由你的程序启动的所有线程状态，排除系统线程

代码片段：

```
import java.util.Set;

public class ThreadStatus {
    public static void main(String args[]) throws Exception {
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new MyThread());
            t.setName("MyThread:" + i);
            t.start();
        }
        int threadCount = 0;
        Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
        for (Thread t : threadSet) {
            if (t.getThreadGroup() == Thread.currentThread().getThreadGroup()) {
                System.out.println("线程 :" + t + ":" + "状态:" + t.getState());
                ++threadCount;
            }
        }
        System.out.println("主线程启动的线程数:" + threadCount);
    }

    class MyThread implements Runnable {
        public void run() {
            try {
                Thread.sleep(2000);
            } catch(Exception err) {

```

相同的可见性效果也适用于**volatile**读/写。所有在写入**volatile**之前更新的变量将被刷新到主内存，且所有在读取**volatile**变量之后的读取都将来自主内存。

Section 126.14: Get status of all threads started by your program excluding system threads

Code snippet:

```
import java.util.Set;

public class ThreadStatus {
    public static void main(String args[]) throws Exception {
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread(new MyThread());
            t.setName("MyThread:" + i);
            t.start();
        }
        int threadCount = 0;
        Set<Thread> threadSet = Thread.getAllStackTraces().keySet();
        for (Thread t : threadSet) {
            if (t.getThreadGroup() == Thread.currentThread().getThreadGroup()) {
                System.out.println("Thread :" + t + ":" + "state:" + t.getState());
                ++threadCount;
            }
        }
        System.out.println("Thread count started by Main thread:" + threadCount);
    }

    class MyThread implements Runnable {
        public void run() {
            try {
                Thread.sleep(2000);
            } catch(Exception err) {

```

```
err.printStackTrace();
}
}
```

输出：

```
线程 :Thread[MyThread:1,main]:状态:TIMED_WAITING
线程 :Thread[MyThread:3,main]:状态:TIMED_WAITING
线程 :Thread[main,5,main]:状态:RUNNABLE
线程 :Thread[MyThread:4,5,main]:状态:TIMED_WAITING
线程 :Thread[MyThread:0,5,main]:状态:TIMED_WAITING
线程 :Thread[MyThread:2,5,main]:状态:TIMED_WAITING
主线程启动的线程数:6
```

说明：

Thread.getAllStackTraces().keySet() 返回所有线程，包括应用线程和系统线程。如果你只关心由你的应用启动的线程状态，可以通过检查特定线程的线程组是否与主程序线程的线程组相同，来遍历该线程集合。

在没有上述线程组条件下，程序返回以下系统线程的状态：

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

第126.15节：使用ThreadLocal

Java并发中的一个有用工具是ThreadLocal——它允许你拥有一个对特定线程唯一的变量。因此，如果相同的代码在不同线程中运行，这些执行不会共享该值，而是每个线程都有自己的线程本地变量。

例如，这通常用于在servlet中建立处理请求的上下文（如授权信息）。你可能会这样做：

```
private static final ThreadLocal<MyUserContext> contexts = new ThreadLocal<>();

public static MyUserContext getContext() {
    return contexts.get(); // get返回该线程唯一的变量
}

public void doGet(...) {
    MyUserContext context = magicGetContextFromRequest(request);
    contexts.put(context); // 将该上下文保存到我们的线程本地变量——其他线程
                           // 调用此方法不会覆盖我们的变量
    try {
        // 业务逻辑
    } finally {
        contexts.remove(); // '确保' 移除线程本地变量
    }
}
```

现在，不用在每个方法中传入MyUserContext，而是在需要的地方使用MyServlet.getContext()。当然，这确实引入了一个需要文档说明的变量，但它是线程安全的，

```
    err.printStackTrace();
}
}
```

Output:

```
Thread :Thread[MyThread:1,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:3,5,main]:state:TIMED_WAITING
Thread :Thread[main,5,main]:state:RUNNABLE
Thread :Thread[MyThread:4,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:0,5,main]:state:TIMED_WAITING
Thread :Thread[MyThread:2,5,main]:state:TIMED_WAITING
Thread count started by Main thread:6
```

Explanation:

Thread.getAllStackTraces().keySet() returns all Threads including application threads and system threads. If you are interested only in status of Threads, started by your application, iterate the Thread set by checking Thread Group of a particular thread against your main program thread.

In absence of above ThreadGroup condition, the program returns status of below System Threads:

```
Reference Handler
Signal Dispatcher
Attach Listener
Finalizer
```

Section 126.15: Using ThreadLocal

A useful tool in Java Concurrency is ThreadLocal – this allows you to have a variable that will be unique to a given thread. Thus, if the same code runs in different threads, these executions will not share the value, but instead each thread has its own variable that is *local to the thread*.

For example, this is frequently used to establish the context (such as authorization information) of handling a request in a servlet. You might do something like this:

```
private static final ThreadLocal<MyUserContext> contexts = new ThreadLocal<>();

public static MyUserContext getContext() {
    return contexts.get(); // get returns the variable unique to this thread
}

public void doGet(...) {
    MyUserContext context = magicGetContextFromRequest(request);
    contexts.put(context); // save that context to our thread-local - other threads
                           // making this call don't overwrite ours
    try {
        // business logic
    } finally {
        contexts.remove(); // 'ensure' removal of thread-local variable
    }
}
```

Now, instead of passing MyUserContext into every single method, you can instead use MyServlet.getContext() where you need it. Now of course, this does introduce a variable that needs to be documented, but it's thread-safe,

这消除了使用如此高作用域变量的许多缺点。

这里的关键优势是每个线程在contexts容器中都有自己的线程局部变量。只要你从定义好的入口点使用它（比如要求每个servlet维护其上下文，或者通过添加servlet过滤器），你就可以依赖在需要时该上下文存在。

第126.16节：带有共享全局队列的多生产者/消费者示例

下面的代码展示了一个多生产者/消费者程序。生产者和消费者线程共享同一个全局队列。

```
import java.util.concurrent.*;
import java.util.Random;

public class ProducerConsumerWithES {
    public static void main(String args[]) {
        BlockingQueue<Integer> sharedQueue = new LinkedBlockingQueue<Integer>();

        ExecutorService pes = Executors.newFixedThreadPool(2);
        ExecutorService ces = Executors.newFixedThreadPool(2);

        pes.submit(new Producer(sharedQueue, 1));
        pes.submit(new Producer(sharedQueue, 2));
        ces.submit(new Consumer(sharedQueue, 1));
        ces.submit(new Consumer(sharedQueue, 2));

        pes.shutdown();
        ces.shutdown();
    }
}

/* 不同的生产者持续不断地向一个共享队列中生产整数流，该队列在所有生产者和消费者之间共享 */

class Producer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    private Random random = new Random();
    public Producer(BlockingQueue<Integer> sharedQueue, int threadNo) {
        this.threadNo = threadNo;
        this.sharedQueue = sharedQueue;
    }
    @Override
    public void run() {
        // 生产者每200毫秒产生一个连续的数字流
        while (true) {
            try {
                int number = random.nextInt(1000);
                System.out.println("Produced:" + number + ":by thread:" + threadNo);
                sharedQueue.put(number);
                Thread.sleep(200);
            } catch (Exception err) {
                err.printStackTrace();
            }
        }
    }
}

/* 不同的消费者从共享队列中消费数据，该队列由生产者和消费者线程共享 */
```

which eliminates a lot of the downsides to using such a highly scoped variable.

The key advantage here is that every thread has its own thread local variable in that contexts container. As long as you use it from a defined entry point (like demanding that each servlet maintains its context, or perhaps by adding a servlet filter) you can rely on this context being there when you need it.

Section 126.16: Multiple producer/consumer example with shared global queue

Below code showcases multiple Producer/Consumer program. Both Producer and Consumer threads share same global queue.

```
import java.util.concurrent.*;
import java.util.Random;

public class ProducerConsumerWithES {
    public static void main(String args[]) {
        BlockingQueue<Integer> sharedQueue = new LinkedBlockingQueue<Integer>();

        ExecutorService pes = Executors.newFixedThreadPool(2);
        ExecutorService ces = Executors.newFixedThreadPool(2);

        pes.submit(new Producer(sharedQueue, 1));
        pes.submit(new Producer(sharedQueue, 2));
        ces.submit(new Consumer(sharedQueue, 1));
        ces.submit(new Consumer(sharedQueue, 2));

        pes.shutdown();
        ces.shutdown();
    }
}

/* Different producers produces a stream of integers continuously to a shared queue, which is shared between all Producers and consumers */

class Producer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    private Random random = new Random();
    public Producer(BlockingQueue<Integer> sharedQueue, int threadNo) {
        this.threadNo = threadNo;
        this.sharedQueue = sharedQueue;
    }
    @Override
    public void run() {
        // Producer produces a continuous stream of numbers for every 200 milli seconds
        while (true) {
            try {
                int number = random.nextInt(1000);
                System.out.println("Produced:" + number + ":by thread:" + threadNo);
                sharedQueue.put(number);
                Thread.sleep(200);
            } catch (Exception err) {
                err.printStackTrace();
            }
        }
    }
}

/* Different consumers consume data from shared queue, which is shared by both producer and consumer threads */
```

```

类 消费者 实现 Runnable {
    私有最终 阻塞队列<整数> 共享队列;
    私有 整数 线程号;
    公共 消费者 (阻塞队列<整数> 共享队列,整数 线程号) {
        this.共享队列 = 共享队列;
        this.线程号 = 线程号;
    }
    @Override
    public void run() {
        // 消费者持续消费由生产者线程生成的数字
        while(真){
            try {
                整数 数字 = 共享队列取();
                系统.输出.打印行("已消费: "+ 数字 + ":由线程:"+线程号);
            } 捕获 (异常 err) {
                err.printStackTrace();
            }
        }
    }
}

```

输出：

```

生产：69：由线程2完成
生产：553：由线程1完成
消费：69：由线程1完成
消费：553：由线程2完成
生产：41：由线程2完成
生产：796：由线程1完成
消费：41：由线程1完成
消费：796：由线程2完成
生产：728：由线程2完成
消费：728：由线程1完成

```

等等.....

说明：

1. sharedQueue，是一个LinkedBlockingQueue，被所有生产者和消费者线程共享。
2. 生产者线程每隔200毫秒持续生成一个整数并将其追加到 sharedQueue
3. 消费者线程持续从sharedQueue中消费整数。
4. 该程序未使用显式的 synchronized或Lock结构。关键在于BlockingQueue的使用。

BlockingQueue的实现主要设计用于生产者-消费者队列。

BlockingQueue 实现是线程安全的。所有排队方法都通过使用内部锁或其他形式的并发控制以原子方式实现其效果。

第126.17节：使用线程池添加两个int数组

线程池有一个任务队列，每个任务将在这些线程中的一个上执行。

```

class Consumer implements Runnable {
    private final BlockingQueue<Integer> sharedQueue;
    private int threadNo;
    public Consumer (BlockingQueue<Integer> sharedQueue,int threadNo) {
        this.sharedQueue = sharedQueue;
        this.threadNo = threadNo;
    }
    @Override
    public void run() {
        // Consumer consumes numbers generated from Producer threads continuously
        while(true){
            try {
                int num = sharedQueue.take();
                System.out.println("Consumed: "+ num + " :by thread:" +threadNo);
            } catch (Exception err) {
                err.printStackTrace();
            }
        }
    }
}

```

output:

```

Produced:69:by thread:2
Produced:553:by thread:1
Consumed: 69:by thread:1
Consumed: 553:by thread:2
Produced:41:by thread:2
Produced:796:by thread:1
Consumed: 41:by thread:1
Consumed: 796:by thread:2
Produced:728:by thread:2
Consumed: 728:by thread:1

```

and so on

Explanation:

1. sharedQueue, which is a LinkedBlockingQueue is shared among all Producer and Consumer threads.
2. Producer threads produces one integer for every 200 milli seconds continuously and append it to sharedQueue
3. Consumer thread consumes integer from sharedQueue continuously.
4. This program is implemented with-out explicit synchronized or Lock constructs. [BlockingQueue](#) is the key to achieve it.

BlockingQueue implementations are designed to be used primarily for producer-consumer queues.

BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control.

Section 126.17: Add two `int` arrays using a Threadpool

A Threadpool has a Queue of tasks, of which each will be executed on one of these Threads.

下面的示例展示了如何使用线程池添加两个int数组。

```
版本 ≥ Java SE 8
int[] firstArray = { 2, 4, 6, 8 };
int[] secondArray = { 1, 3, 5, 7 };
int[] result = { 0, 0, 0, 0 };

ExecutorService pool = Executors.newCachedThreadPool();

// 设置线程池：
// 对数组中的每个元素，提交一个工作线程到线程池，执行元素相加操作
for (int i = 0; i < result.length; i++) {
    final int worker = i;
    pool.submit(() -> result[worker] = firstArray[worker] + secondArray[worker]);
}

// 等待所有工作线程完成：
try {
    // 执行所有提交的任务
    pool.关闭();
    // 等待直到所有工作线程完成，或超时结束
    pool.等待终止(12, TimeUnit.SECONDS);
}
捕获 (InterruptedException e) {
    pool.立即关闭(); // 终止线程
}

System.out.println(Arrays.toString(result));
```

注意事项：

1. 这个示例仅用于说明。实际上，对于这么小的任务，使用线程不会带来任何加速。反而可能变慢，因为任务创建和调度的开销会超过执行任务所需的时间。
2. 如果你使用的是 Java 7 及更早版本，你需要使用匿名类而不是 lambda 来实现任务。

第126.18节：暂停执行

Thread.sleep 会使当前线程暂停执行指定时间。这是一种有效的方式，可以让处理器时间让给应用程序的其他线程或可能正在运行的其他应用程序。Thread 类中有两个重载的 睡眠 方法。

一个指定睡眠时间到毫秒的方法

```
public static void sleep(long millis) throws InterruptedException
```

指定睡眠时间到纳秒级别的方法

```
public static void sleep(long millis, int nanos)
```

暂停执行1秒钟

```
Thread.sleep(1000);
```

需要注意的是，这只是对操作系统内核调度器的一个提示。这可能不一定

The following example shows how to add two int arrays using a Threadpool.

```
Version ≥ Java SE 8
int[] firstArray = { 2, 4, 6, 8 };
int[] secondArray = { 1, 3, 5, 7 };
int[] result = { 0, 0, 0, 0 };

ExecutorService pool = Executors.newCachedThreadPool();

// Setup the ThreadPool:
// for each element in the array, submit a worker to the pool that adds elements
for (int i = 0; i < result.length; i++) {
    final int worker = i;
    pool.submit(() -> result[worker] = firstArray[worker] + secondArray[worker]);
}

// Wait for all Workers to finish:
try {
    // execute all submitted tasks
    pool.shutdown();
    // waits until all workers finish, or the timeout ends
    pool.awaitTermination(12, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    pool.shutdownNow(); // kill thread
}

System.out.println(Arrays.toString(result));
```

Notes:

1. This example is purely illustrative. In practice, there won't be any speedup by using threads for a task this small. A slowdown is likely, since the overheads of task creation and scheduling will swamp the time taken to run a task.
2. If you were using Java 7 and earlier, you would use anonymous classes instead of lambdas to implement the tasks.

Section 126.18: Pausing Execution

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. There are two overloaded sleep methods in the Thread class.

One that specifies the sleep time to the millisecond

```
public static void sleep(long millis) throws InterruptedException
```

One that specifies the sleep time to the nanosecond

```
public static void sleep(long millis, int nanos)
```

Pausing Execution for 1 second

```
Thread.sleep(1000);
```

It is important to note that this is a hint to the operating system's kernel's scheduler. This may not necessarily be

精确，有些实现甚至不考虑纳秒参数（可能会四舍五入到最近的毫秒）。

建议将对Thread.sleep的调用放在try/catch中，并捕获InterruptedException异常。

第126.19节：线程中断 / 停止线程

每个Java线程都有一个中断标志，初始值为false。中断一个线程，本质上不过是将该标志设置为true。运行在线程上的代码可以偶尔检查该标志并据此采取行动。代码也可以完全忽略它。但为什么每个线程都要有这样一个标志呢？毕竟，在需要时，我们完全可以自己组织一个线程上的布尔标志。其实，当线程被中断时，有些方法会表现出特殊的行为。这些方法被称为阻塞方法。

这些方法会使线程进入等待（WAITING）或计时等待（TIMED_WAITING）状态。当线程处于该状态时，中断它会导致被中断的线程抛出InterruptedException异常，而不是将中断标志设置为true，线程也会重新变为可运行（RUNNABLE）状态。调用阻塞方法的代码必须处理InterruptedException，因为它是一个受检异常。因此，正如其名称所示，中断可以起到中断等待（WAIT）的作用，有效地结束等待。注意，并非所有以某种方式等待的方法（例如

阻塞IO）以这种方式响应中断，因为它们不会将线程置于等待状态。最后，一个中断标志被设置的线程，如果进入阻塞方法（即尝试进入等待状态），将立即抛出InterruptedException异常，并且中断标志将被清除。

除了这些机制之外，Java 并没有赋予中断任何特殊的语义含义。代码可以自由地以任何方式解释中断。但中断最常见的用途是向线程发出信号，告诉它应尽早停止运行。但正如上文所述，线程上的代码需要适当地响应该中断以停止运行。停止线程是一个协作过程。当线程被中断时，其运行的代码可能已经深入调用栈的多层。大多数代码不会调用阻塞方法，并且能及时完成，从而不会过度延迟线程的停止。主要需要关注响应中断的是那些在循环中处理任务的代码，直到任务处理完毕或设置了停止循环的标志。处理可能无限任务的循环（即原则上持续运行）应检查中断标志以退出循环。对于有限循环，语义可能要求在结束前完成所有任务，或者允许留下部分任务未处理。调用阻塞方法的代码将被迫处理 InterruptedException。如果语义上允许，它可以简单地传播 InterruptedException 并声明抛出它。因此，它本身也成为一个针对其调用者的阻塞方法。如果不能传播该异常，至少应设置中断标志，以便调用栈上层的调用者也知道线程已被中断。在某些情况下，方法需要无视 InterruptedException 继续等待，这时必须延迟设置中断标志，直到等待完成，这可能涉及设置一个局部变量，在方法退出前检查该变量，然后中断其线程。

示例：

中断时停止处理任务的代码示例

```
class TaskHandler implements Runnable {  
  
    private final BlockingQueue<Task> queue;  
  
    TaskHandler(BlockingQueue<Task> queue) {  
        this.queue = queue;  
    }  
  
    @Override  
    public void run() {  
        while (!Thread.currentThread().isInterrupted()) { // 检查中断标志，退出循环  
            // 处理任务逻辑  
        }  
    }  
}
```

precise, and some implementations do not even consider the nanosecond parameter (possibly rounding to the nearest millisecond).

It is recommended to enclose a call to `Thread.sleep` in try/catch and catch `InterruptedException`.

Section 126.19: Thread Interruption / Stopping Threads

Each Java Thread has an interrupt flag, which is initially false. Interrupting a thread, is essentially nothing more than setting that flag to true. The code running on that thread can check the flag on occasion and act upon it. The code can also ignore it completely. But why would each Thread have such a flag? After all, having a boolean flag on a thread is something we can just organize ourselves, if and when we need it. Well, there are methods that behave in a special way when the thread they're running on is interrupted. These methods are called blocking methods. These are methods that put the thread in the WAITING or TIMED_WAITING state. When a thread is in this state, interrupting it, will cause an InterruptedException to be thrown on the interrupted thread, rather than the interrupt flag being set to true, and the thread becomes RUNNABLE again. Code that invokes a blocking method is forced to deal with the InterruptedException, since it is a checked exception. So, and hence its name, an interrupt can have the effect of interrupting a WAIT, effectively ending it. Note that not all methods that are somehow waiting (e.g. blocking IO) respond to interruption in that way, as they don't put the thread in a waiting state. Lastly a thread that has its interrupt flag set, that enters a blocking method (i.e. tries to get into a waiting state), will immediately throw an InterruptedException and the interrupt flag will be cleared.

Other than these mechanics, Java does not assign any special semantic meaning to interruption. Code is free to interpret an interrupt any way it likes. But most often interruption is used to signal to a thread it should stop running at its earliest convenience. But, as should be clear from the above, it is up to the code on that thread to react to that interruption appropriately in order to stop running. Stopping a thread is a collaboration. When a thread is interrupted its running code can be several levels deep into the stacktrace. Most code doesn't call a blocking method, and finishes timely enough to not delay the stopping of the thread unduly. The code that should mostly be concerned with being responsive to interruption, is code that is in a loop handling tasks until there are none left, or until a flag is set signalling it to stop that loop. Loops that handle possibly infinite tasks (i.e. they keep running in principle) should check the interrupt flag in order to exit the loop. For finite loops the semantics may dictate that all tasks must be finished before ending, or it may be appropriate to leave some tasks unhandled. Code that calls blocking methods will be forced to deal with the InterruptedException. If at all semantically possible, it can simply propagate the InterruptedException and declare to throw it. As such it becomes a blocking method itself in regard to its callers. If it cannot propagate the exception, it should at the very least set the interrupted flag, so callers higher up the stack also know the thread was interrupted. In some cases the method needs to continue waiting regardless of the InterruptedException, in which case it must delay setting the interrupted flag until after it is done waiting, this may involve setting a local variable, which is to be checked prior to exiting the method to then interrupt its thread.

Examples :

Example of code that stops handling tasks upon interruption

```
class TaskHandler implements Runnable {  
  
    private final BlockingQueue<Task> queue;  
  
    TaskHandler(BlockingQueue<Task> queue) {  
        this.queue = queue;  
    }  
  
    @Override  
    public void run() {  
        while (!Thread.currentThread().isInterrupted()) { // check for interrupt flag, exit loop  
            // handle task  
        }  
    }  
}
```

当被中断时

```

try {
任务 task = 队列.take(); // 阻塞调用, 响应中断
    处理(任务);
} catch (InterruptedException e) {
    线程.当前线程().中断(); // 由于Runnable接口限制, 不能抛出InterruptedException, 因此通过设置标志
来表示中断
}
}

私有 无返回值 处理(任务 task) {
    // 实际处理
}
}

```

延迟设置中断标志直到完全完成的代码示例：

```

类 必须完成处理器 实现 Runnable {

private final BlockingQueue<Task> queue;

必须完成处理器(阻塞队列<任务> 队列) {
    this.queue = queue;
}

@Override
公共 无返回值 运行() {
    布尔 应该中断 = 假;

    当 (真) {
        尝试 {
任务 task = 队列取出();
        如果 (task.是任务结束()) {
            如果 (应该中断) {
                线程.当前线程().中断();
            }
            return;
        }
    }
    处理(task);
} catch (InterruptedException e) {
    应该中断 = 真; // 必须完成, 完成后记得设置中断标志
}
}

私有 无返回值 处理(任务 task) {
    // 实际处理
}
}

```

具有固定任务列表但可能在中断时提前退出的代码示例

```

类 尽可能前进 实现 可运行接口 {

私有最终 列表<任务> 任务列表 = 新建 数组列表<>();

@Override
公共 无返回值 运行() {
    对于 (任务 task : 任务列表) {
}
}

```

when interrupted

```

try {
    Task task = queue.take(); // blocking call, responsive to interruption
    handle(task);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt(); // cannot throw InterruptedException (due to
Runnable interface restriction) so indicating interruption by setting the flag
}
}

private void handle(Task task) {
    // actual handling
}
}

```

Example of code that delays setting the interrupt flag until completely done :

```

class MustFinishHandler implements Runnable {

private final BlockingQueue<Task> queue;

MustFinishHandler(BlockingQueue<Task> queue) {
    this.queue = queue;
}

@Override
public void run() {
    boolean shouldInterrupt = false;

    while (true) {
        try {
            Task task = queue.take();
            if (task.isEndOfTasks()) {
                if (shouldInterrupt) {
                    Thread.currentThread().interrupt();
                }
                return;
            }
            handle(task);
        } catch (InterruptedException e) {
            shouldInterrupt = true; // must finish, remember to set interrupt flag when we're
done
        }
    }
}

private void handle(Task task) {
    // actual handling
}
}

```

Example of code that has a fixed list of tasks but may quit early when interrupted

```

class GetAsFarAsPossible implements Runnable {

private final List<Task> tasks = new ArrayList<>();

@Override
public void run() {
    for (Task task : tasks) {
}
}

```

```
if (Thread.currentThread().isInterrupted()) {  
    return;  
}  
handle(任务);  
}  
  
私有 无返回值 处理(任务 task) {  
    // 实际处理  
}
```

```
if (Thread.currentThread().isInterrupted()) {  
    return;  
}  
handle(task);  
}  
  
private void handle(Task task) {  
    // actual handling  
}
```

第127章：执行器、执行器服务和线程池

Java中的Executor接口提供了一种将任务提交与任务执行机制（包括线程使用、调度等细节）解耦的方式。通常使用Executor来代替显式创建线程。通过Executor，开发者无需大幅重写代码即可轻松调整程序的任务执行策略。

第127.1节：ThreadPoolExecutor

常用的Executor是ThreadPoolExecutor，它负责线程的管理。你可以配置执行器在任务较少时始终保持的最小线程数（称为核心线程数），以及线程池在任务增多时可扩展到的最大线程数。工作量减少时，线程池会逐渐减少线程数，直到达到最小线程数。

```
ThreadPoolExecutor 池 = new ThreadPoolExecutor(  
    1, // 保持至少一个线程准备就绪,  
        // 即使没有Runnable被执行  
    5, // 最多五个Runnable/线程  
        // 并行执行  
    1, TimeUnit.MILLISECONDS, // 空闲线程在超过最小线程池大小后  
        // 一分钟后终止  
    new ArrayBlockingQueue<Runnable>(10)); // 待执行的Runnable保存在这里  
  
pool.execute(new Runnable() {  
    @Override public void run() {  
        //运行的代码  
    }  
});
```

注意 如果你配置了一个ThreadPoolExecutor使用一个无界队列，那么线程数不会超过corePoolSize，因为只有当队列满时才会创建新线程：

带所有参数的ThreadPoolExecutor：

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,  
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,  
RejectedExecutionHandler handler)
```

摘自[JavaDoc](#)

如果正在运行的线程数超过corePoolSize但少于maximumPoolSize，只有当队列满时才会创建新线程。

优点：

1. 可以控制BlockingQueue的大小，避免内存溢出场景。有限的有界队列大小不会降低应用性能。
2. 你可以使用现有的或创建新的拒绝处理策略。

1. 在默认的ThreadPoolExecutor.AbortPolicy中，处理器会抛出运行时异常

Chapter 127: Executor, ExecutorService and Thread pools

The [Executor](#) interface in Java provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used instead of explicitly creating threads. With Executors, developers won't have to significantly rewrite their code to be able to easily tune their program's task-execution policy.

Section 127.1: ThreadPoolExecutor

A common Executor used is the ThreadPoolExecutor, which takes care of Thread handling. You can configure the minimal amount of Threads the executor always has to maintain when there's not much to do (it's called core size) and a maximal Thread size to which the Pool can grow, if there is more work to do. Once the workload declines, the Pool slowly reduces the Thread count again until it reaches min size.

```
ThreadPoolExecutor pool = new ThreadPoolExecutor(  
    1, // keep at least one thread ready,  
        // even if no Runnables are executed  
    5, // at most five Runnables/Threads  
        // executed in parallel  
    1, TimeUnit.MILLISECONDS, // idle Threads terminated after one  
        // minute, when min Pool size exceeded  
    new ArrayBlockingQueue<Runnable>(10)); // outstanding Runnables are kept here  
  
pool.execute(new Runnable() {  
    @Override public void run() {  
        //code to run  
    }  
});
```

Note If you configure the ThreadPoolExecutor with an *unbounded* queue, then the thread count will not exceed corePoolSize since new threads are only created if the queue is full:

ThreadPoolExecutor with all parameters:

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,  
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,  
RejectedExecutionHandler handler)
```

from [JavaDoc](#)

If there are more than corePoolSize but less than maximumPoolSize threads running, a new thread will be created only if the queue is full.

Advantages:

1. BlockingQueue size can be controlled and out-of-memory scenarios can be avoided. Application performance won't be degraded with limited bounded queue size.
2. You can use existing or create new Rejection Handler policies.
 1. In the default ThreadPoolExecutor.AbortPolicy, the handler throws a runtime

拒绝时抛出 RejectedExecutionException。

2. 在ThreadPoolExecutor.CallerRunsPolicy中，调用 execute 的线程本身会运行该任务。这提供了一种简单的反馈控制机制，可以减缓新任务提交的速度。
3. 在ThreadPoolExecutor.DiscardPolicy中，无法执行的任务会被直接丢弃。
4. 在ThreadPoolExecutor.DiscardOldestPolicy中，如果执行器未关闭，工作队列头部的任务会被丢弃，然后重新尝试执行（可能再次失败，导致重复此过程）。
3. 可以配置自定义的ThreadFactory，这很有用：

1. 设置更具描述性的线程名称
2. 设置线程守护状态
3. 设置线程优先级

以下是如何使用 ThreadPoolExecutor 的示例

第127.2节：从计算中获取值 - 可调用

如果你的计算产生了一些后续需要的返回值，简单的Runnable任务是不够的。对于这种情况，你可以使用ExecutorService.submit(Callable<T>)，它在执行完成后返回一个值。

该服务将返回一个Future，你可以用它来获取任务执行的结果。

```
// 提交一个可调用任务进行执行
ExecutorService 线程池 = anExecutorService;
Future<Integer> future = 线程池.submit(new Callable<Integer>() {
    @Override public Integer call() {
        // 执行一些计算
        return new Random().nextInt();
    }
});
// ... 在future在另一个线程执行时执行其他任务
```

当你需要获取future的结果时，调用future.get()

- 无限期等待未来完成并返回结果。

```
try {
    // 阻塞当前线程直到 future 完成
    Integer result = future.get();
} catch (InterruptedException || ExecutionException e) {
    // 适当处理
}
```

- 等待 future 完成，但不超过指定时间。

```
try {
    // 阻塞当前线程最多 500 毫秒。
    // 如果 future 在此之前完成，则返回结果，
    // 否则抛出 TimeoutException。
    Integer result = future.get(500, TimeUnit.MILLISECONDS);
} catch (InterruptedException || ExecutionException || TimeoutException e) {
```

RejectedExecutionException upon rejection.

2. In ThreadPoolExecutor.CallerRunsPolicy, the thread that invokes execute itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.
3. In ThreadPoolExecutor.DiscardPolicy, a task that cannot be executed is simply dropped.
4. In ThreadPoolExecutor.DiscardOldestPolicy, if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)
3. Custom ThreadFactory can be configured, which is useful :

1. To set a more descriptive thread name
2. To set thread daemon status
3. To set thread priority

[Here](#) is a example of how to use ThreadPoolExecutor

Section 127.2: Retrieving value from computation - Callable

If your computation produces some return value which later is required, a simple Runnable task isn't sufficient. For such cases you can use ExecutorService.submit(Callable<T>) which returns a value after execution completes.

The Service will return a Future which you can use to retrieve the result of the task execution.

```
// Submit a callable for execution
ExecutorService pool = anExecutorService;
Future<Integer> future = pool.submit(new Callable<Integer>() {
    @Override public Integer call() {
        //do some computation
        return new Random().nextInt();
    }
});
// ... perform other tasks while future is executed in a different thread
```

When you need to get the result of the future, call future.get()

- Wait indefinitely for future to finish with a result.

```
try {
    // Blocks current thread until future is completed
    Integer result = future.get();
} catch (InterruptedException || ExecutionException e) {
    // handle appropriately
}
```

- Wait for future to finish, but no longer than specified time.

```
try {
    // Blocks current thread for a maximum of 500 milliseconds.
    // If the future finishes before that, result is returned,
    // otherwise TimeoutException is thrown.
    Integer result = future.get(500, TimeUnit.MILLISECONDS);
} catch (InterruptedException || ExecutionException || TimeoutException e) {
```

```
// 适当处理  
}
```

如果不再需要已调度或正在运行任务的结果，可以调用 Future.cancel(boolean) 来取消它。

- 调用 cancel(false) 只会将任务从待运行任务队列中移除。
- 调用cancel(true)也会中断当前正在运行的任务。

第127.3节：submit() 与 execute() 异常处理差异

通常 execute() 命令用于“发送即忘”调用（不需要分析结果），而 submit() 命令用于分析 Future 对象的结果。

我们应当注意这两个命令在异常处理机制上的关键区别。

如果未捕获异常，submit() 抛出的异常会被框架吞掉。

理解差异的代码示例：

案例1：使用 execute() 命令提交 Runnable，会报告异常。

```
import java.util.concurrent.*;  
import java.util.*;  
  
public class ExecuteSubmitDemo {  
    public ExecuteSubmitDemo() {  
        System.out.println("创建服务");  
        ExecutorService service = Executors.newFixedThreadPool(2);  
        //ExtendedExecutor service = new ExtendedExecutor();  
        for (int i = 0; i < 2; i++) {  
            service.execute(new Runnable(){  
                public void run(){  
                    int a = 4, b = 0;  
                    System.out.println("a and b=" + a + ":" + b);  
                    System.out.println("a/b:" + (a / b));  
                    System.out.println("Thread Name in Runnable after divide by  
zero:"+Thread.currentThread().getName());  
                }  
            });  
        }  
        service.shutdown();  
    }  
    public static void main(String args[]){  
        ExecuteSubmitDemo demo = new ExecuteSubmitDemo();  
    }  
}  
  
class ExtendedExecutor extends ThreadPoolExecutor {  
  
    public ExtendedExecutor() {  
        super(1, 1, 60, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(100));  
    }  
    // ...  
    protected void afterExecute(Runnable r, Throwable t) {  
        super.afterExecute(r, t);  
        if (t == null && r instanceof Future<?>) {  
            try {  
                Object result = ((Future<?>) r).get();  
            }  
        }  
    }  
}
```

```
// handle appropriately  
}
```

If the result of a scheduled or running task is no longer required, you can call Future.cancel(boolean) to cancel it.

- Calling cancel(**false**) will just remove the task from the queue of tasks to be run.
- Calling cancel(**true**) will also interrupt the task if it is currently running.

Section 127.3: submit() vs execute() exception handling differences

Generally execute() command is used for fire and forget calls (without need of analyzing the result) and submit() command is used for analyzing the result of Future object.

We should be aware of key difference of Exception Handling mechanisms between these two commands.

Exceptions from submit() are swallowed by framework if you did not catch them.

Code example to understand the difference:

Case 1: submit the Runnable with execute() command, which reports the Exception.

```
import java.util.concurrent.*;  
import java.util.*;  
  
public class ExecuteSubmitDemo {  
    public ExecuteSubmitDemo() {  
        System.out.println("creating service");  
        ExecutorService service = Executors.newFixedThreadPool(2);  
        //ExtendedExecutor service = new ExtendedExecutor();  
        for (int i = 0; i < 2; i++) {  
            service.execute(new Runnable(){  
                public void run(){  
                    int a = 4, b = 0;  
                    System.out.println("a and b=" + a + ":" + b);  
                    System.out.println("a/b:" + (a / b));  
                    System.out.println("Thread Name in Runnable after divide by  
zero:"+Thread.currentThread().getName());  
                }  
            });  
        }  
        service.shutdown();  
    }  
    public static void main(String args[]){  
        ExecuteSubmitDemo demo = new ExecuteSubmitDemo();  
    }  
}  
  
class ExtendedExecutor extends ThreadPoolExecutor {  
  
    public ExtendedExecutor() {  
        super(1, 1, 60, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(100));  
    }  
    // ...  
    protected void afterExecute(Runnable r, Throwable t) {  
        super.afterExecute(r, t);  
        if (t == null && r instanceof Future<?>) {  
            try {  
                Object result = ((Future<?>) r).get();  
            }  
        }  
    }  
}
```

```

} catch (CancellationException ce) {
    t = ce;
} catch (ExecutionException ee) {
    t = ee.getCause();
} catch (InterruptedException ie) {
    Thread.currentThread().interrupt(); // ignore/reset
}
if (t != null)
    System.out.println(t);
}

```

输出：

```

创建服务
a 和 b=4:0
a 和 b=4:0
线程“pool-1-thread-1”中发生异常 线程“pool-1-thread-2”中发生异常
java.lang.ArithmetricException: 除以零
at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:744)
java.lang.ArithmetricException: / by zero
at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:744)

```

案例2：用submit()替换execute()：service.submit(new Runnable(){ 在这种情况下，由于run()方法没有显式捕获异常，异常会被框架吞掉。

输出：

```

创建服务
a 和 b=4:0
a 和 b=4:0

```

案例3：将newFixedThreadPool改为ExtendedExecutor

```

//ExecutorService service = Executors.newFixedThreadPool(2);
ExtendedExecutor service = new ExtendedExecutor();

```

输出：

```

创建服务
a 和 b=4:0
java.lang.ArithmetricException: / by zero
a 和 b=4:0
java.lang.ArithmetricException: / by zero

```

我演示了这个例子以涵盖两个主题：使用自定义的ThreadPoolExecutor以及使用自定义ThreadPoolExecutor处理异常。

```

} catch (CancellationException ce) {
    t = ce;
} catch (ExecutionException ee) {
    t = ee.getCause();
} catch (InterruptedException ie) {
    Thread.currentThread().interrupt(); // ignore/reset
}
if (t != null)
    System.out.println(t);
}

```

输出：

```

creating service
a 和 b=4:0
a 和 b=4:0
Exception in thread "pool-1-thread-1" Exception in thread "pool-1-thread-2"
java.lang.ArithmetricException: / by zero
at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:744)
java.lang.ArithmetricException: / by zero
at ExecuteSubmitDemo$1.run(ExecuteSubmitDemo.java:15)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:744)

```

Case 2: Replace execute() with submit(): service.submit(new Runnable(){ In this case, Exceptions are swallowed by framework since run() method did not catch them explicitly.

输出：

```

creating service
a 和 b=4:0
a 和 b=4:0

```

Case 3: Change the newFixedThreadPool to ExtendedExecutor

```

//ExecutorService service = Executors.newFixedThreadPool(2);
ExtendedExecutor service = new ExtendedExecutor();

```

输出：

```

creating service
a 和 b=4:0
java.lang.ArithmetricException: / by zero
a 和 b=4:0
java.lang.ArithmetricException: / by zero

```

I have demonstrated this example to cover two topics : Use your custom ThreadPoolExecutor and handle Exception with custom ThreadPoolExecutor.

解决上述问题的另一种简单方法：当你使用普通的ExecutorService和submit命令时，从submit()命令获取Future对象，调用Future的get()方法。捕获afterExecute方法实现中提到的三种异常。自定义ThreadPoolExecutor相比这种方法的优势：
你必须只在一个地方处理异常处理机制——自定义 ThreadPoolExecutor。

第127.4节：处理拒绝执行

如果

1. 你尝试向已关闭的执行器提交任务，或2. 队列已饱和
(仅限有界队列)，且线程数已达到最大，

RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor) 将被调用。

默认行为是调用者会收到一个 RejectedExecutionException 异常。但还有更多预定义的行为可用：

- **ThreadPoolExecutor.AbortPolicy** (默认，会抛出 REE)
- **ThreadPoolExecutor.CallerRunsPolicy** (在调用者线程上执行任务——阻塞它)
- **ThreadPoolExecutor.DiscardPolicy** (静默丢弃任务)
- **ThreadPoolExecutor.DiscardOldestPolicy** (静默丢弃队列中最旧的任务并重试执行新任务)

你可以通过 ThreadPool 的构造函数之一来设置它们：

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler) // <--
```

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) // <--
```

你也可以通过扩展RejectedExecutionHandler接口来实现你自己的行为：

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

第127.5节：火并忘记 - Runnable任务

Executors 接受一个java.lang.Runnable，它包含（可能是计算密集型或其他长时间运行或耗时）代码，在另一个线程中运行。

用法如下：

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //卸载的工作，无需获取返回结果
    }
});
```

Other simple solution to above problem : When you are using normal ExecutorService & submit command, get the Future object from submit() command call get() API on Future. Catch the three exceptions, which have been quoted in afterExecute method implementation. Advantage of custom ThreadPoolExecutor over this approach : You have to handle Exception handling mechanism in only one place - Custom ThreadPoolExecutor.

Section 127.4: Handle Rejected Execution

If

1. you try to submit tasks to a shutdown Executor or
2. the queue is saturated (only possible with bounded ones) and maximum number of Threads has been reached,

RejectedExecutionHandler.rejectedExecution(Runnable, ThreadPoolExecutor) will be called.

The default behavior is that you'll get a RejectedExecutionException thrown at the caller. But there are more predefined behaviors available:

- **ThreadPoolExecutor.AbortPolicy** (default, will throw REE)
- **ThreadPoolExecutor.CallerRunsPolicy** (executes task on caller's thread - *blocking it*)
- **ThreadPoolExecutor.DiscardPolicy** (silently discard task)
- **ThreadPoolExecutor.DiscardOldestPolicy** (silently discard **oldest** task in queue and retry execution of the new task)

You can set them using one of the ThreadPool [constructors](#):

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          RejectedExecutionHandler handler) // <--
```

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) // <--
```

You can as well implement your own behavior by extending [RejectedExecutionHandler](#) interface:

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

Section 127.5: Fire and Forget - Runnable Tasks

Executors accept a java.lang.Runnable which contains (potentially computationally or otherwise long-running or heavy) code to be run in another Thread.

Usage would be:

```
Executor exec = anExecutor;
exec.execute(new Runnable() {
    @Override public void run() {
        //offloaded work, no need to get result back
    }
});
```

```
});
```

注意，使用此执行器，你无法获取任何计算结果。
使用 Java 8，可以利用 lambda 表达式来简化代码示例。

```
版本 ≥ Java SE 8
Executor exec = anExecutor;
exec.execute(() -> {
    //卸载的工作，无需获取返回结果
});
```

```
});
```

Note that with this executor, you have no means to get any computed value back.
With Java 8, one can utilize lambdas to shorten the code example.

```
Version ≥ Java SE 8
Executor exec = anExecutor;
exec.execute(() -> {
    //offloaded work, no need to get result back
});
```

第127.6节：不同类型并发结构的使用场景

1. ExecutorService

```
ExecutorService executor = Executors.newFixedThreadPool(50);
```

它简单易用，隐藏了ThreadPoolExecutor的底层细节。

当Callable/Runnable任务数量较少，且任务堆积在无界队列中不会增加内存占用或降低系统性能时，我更倾向于使用这个。
如果有CPU/内存限制，我更倾向于使用带容量限制的ThreadPoolExecutor，并配合RejectedExecutionHandler来处理任务拒绝。

2. CountDownLatch

CountDownLatch 将使用给定的计数进行初始化。该计数通过调用 countDown() 方法递减。等待计数归零的线程可以调用 await() 方法之一。调用 await() 会阻塞线程，直到计数达到零。该类使得一个 Java 线程能够等待其他一组线程完成它们的任务。

用例：

1. 实现最大并行性：有时我们希望同时启动多个线程以实现最大并行性
 2. 等待N个线程完成后再开始执行
 3. 死锁检测。
3. [ThreadPoolExecutor](#)：它提供了更多的控制。如果应用程序受限于待处理的Runnable/Callable任务数量，可以通过设置最大容量来使用有界队列。一旦队列达到最大容量，可以定义拒绝处理器。Java提供了四种类型的RejectedExecutionHandler策略。

1. ThreadPoolExecutor.AbortPolicy，拒绝时处理器会抛出运行时的RejectedExecutionException。
- 2.ThreadPoolExecutor.CallerRunsPolicy，调用execute的线程本身运行该任务。这提供了一种简单的反馈控制机制，可以减缓新任务提交的速度。

Section 127.6: Use cases for different types of concurrency constructs

1. ExecutorService

```
ExecutorService executor = Executors.newFixedThreadPool(50);
```

It is simple and easy to use. It hides low level details of ThreadPoolExecutor.

I prefer this one when number of Callable / Runnable tasks are small in number and piling of tasks in unbounded queue does not increase memory & degrade the performance of the system. If you have CPU / Memory constraints, I prefer to use ThreadPoolExecutor with capacity constraints & RejectedExecutionHandler to handle rejection of tasks.

2. CountDownLatch

CountDownLatch will be initialized with a given count. This count is decremented by calls to the countDown() method. Threads waiting for this count to reach zero can call one of the await() methods. Calling await() blocks the thread until the count reaches zero. *This class enables a java thread to wait until other set of threads completes their tasks.*

Use cases:

1. Achieving Maximum Parallelism: Sometimes we want to start a number of threads at the same time to achieve maximum parallelism
 2. Wait N threads to completes before start execution
 3. Deadlock detection.
3. [ThreadPoolExecutor](#) : It provides more control. If application is constrained by number of pending Runnable/Callable tasks, you can use bounded queue by setting the max capacity. Once the queue reaches maximum capacity, you can define RejectionHandler. Java provides four types of RejectedExecutionHandler policies.
1. ThreadPoolExecutor.AbortPolicy, the handler throws a runtime RejectedExecutionException upon rejection.
 2. ThreadPoolExecutor.CallerRunsPolicy` , the thread that invokes execute itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.

3. 在ThreadPoolExecutor.DiscardPolicy中，无法执行的任务会被直接丢弃。
4. ThreadPoolExecutor.DiscardOldestPolicy，如果执行器未关闭，则丢弃队列头部的任务工作队列被丢弃，然后重新尝试执行（这可能再次失败，导致重复发生）。

如果你想模拟CountDownLatch的行为，可以使用invokeAll()方法。

4. 你没有提到的另一个机制是ForkJoinPool

ForkJoinPool是在Java 7中引入的。它类似于Java的ExecutorService，但有一个区别。ForkJoinPool使任务能够轻松地将工作拆分成更小的任务，这些任务也会提交给ForkJoinPool。当空闲的工作线程从繁忙的工作线程队列中窃取任务时，就会发生任务窃取。

Java 8在ExecutorService中引入了一个新的API来创建工作窃取线程池。你不必创建RecursiveTask和RecursiveAction，但仍然可以使用ForkJoinPool。

```
public static ExecutorService newWorkStealingPool()
```

使用所有可用处理器作为目标并行级别，创建一个工作窃取线程池。

默认情况下，它将使用CPU核心数作为参数。

这四种机制是相辅相成的。根据你想控制的粒度级别，你需要选择合适的机制。

第127.7节：等待ExecutorService中所有任务完成

让我们来看一下等待提交给Executor的任务完成的各种选项

1. [ExecutorService invokeAll\(\)](#)

执行给定的任务，返回一个包含它们状态和结果的Future列表，当所有任务完成时。

示例：

```
import java.util.concurrent.*;
import java.util.*;

public class InvokeAllDemo{
    public InvokeAllDemo(){
        System.out.println("创建服务");
        ExecutorService service =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
        List<MyCallable> futureList = new ArrayList<MyCallable>();
```

3. In ThreadPoolExecutor.DiscardPolicy, a task that cannot be executed is simply dropped.
4. ThreadPoolExecutor.DiscardOldestPolicy, if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)

If you want to simulate CountDownLatch behaviour, you can use [invokeAll\(\)](#) method.

4. One more mechanism you did not quote is [ForkJoinPool](#)

The ForkJoinPool was added to Java in Java 7. The ForkJoinPool is similar to the Java ExecutorService but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Task stealing happens in ForkJoinPool when free worker threads steal tasks from busy worker thread queue.

Java 8 has introduced one more API in [ExecutorService](#) to create work stealing pool. You don't have to create RecursiveTask and RecursiveAction but still can use ForkJoinPool.

```
public static ExecutorService newWorkStealingPool()
```

Creates a work-stealing thread pool using all available processors as its target parallelism level.

By default, it will take number of CPU cores as parameter.

All these four mechanism are complimentary to each other. Depending on level of granularity you want to control, you have to chose right ones.

Section 127.7: Wait for completion of all tasks in ExecutorService

Let's have a look at various options to wait for completion of tasks submitted to [Executor](#)

1. [ExecutorService invokeAll\(\)](#)

Executes the given tasks, returning a list of Futures holding their status and results when everything is completed.

Example:

```
import java.util.concurrent.*;
import java.util.*;

public class InvokeAllDemo{
    public InvokeAllDemo(){
        System.out.println("creating service");
        ExecutorService service =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
        List<MyCallable> futureList = new ArrayList<MyCallable>();
```

```

        for (int i = 0; i < 10; i++){
    MyCallable myCallable = new MyCallable((long)i);
        futureList.add(myCallable);
    }
    System.out.println("Start");
    try{
List<Future<Long>> futures = service.invokeAll(futureList);
    } catch(Exception err){
err.printStackTrace();
}
    System.out.println("Completed");
    service.shutdown();
}
public static void main(String args[]){
    InvokeAllDemo demo = new InvokeAllDemo();
}
class MyCallable implements Callable<Long>{
    Long id = 0L;
    public MyCallable(Long val){
        this.id = val;
    }
    public Long call(){
        // 添加您的业务逻辑
        return id;
    }
}
}

```

2. CountDownLatch

一种同步辅助工具，允许一个或多个线程等待，直到其他线程中执行的一组操作完成。

CountDownLatch（倒计时锁存器）以给定的计数初始化。await 方法会阻塞，直到当前计数因调用 countDown() 方法而达到零，随后所有等待的线程被释放，任何后续的 await 调用会立即返回。这是一次性现象——计数不能重置。如果需要可重置计数的版本，请考虑使用 CyclicBarrier（循环屏障）。

3. Executors 中的 ForkJoinPool 或 newWorkStealingPool()

4. 遍历所有在提交给ExecutorService后创建的Future对象

5. Oracle文档中关于ExecutorService的推荐关闭方式：

```

void shutdownAndAwaitTermination(ExecutorService pool) {
    pool.shutdown(); // 禁止提交新任务
    try {
        // 等待一段时间以让现有任务终止
        if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
            pool.shutdownNow(); // 取消目前正在执行的任务
        // 等待一段时间以让任务响应取消
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("线程池未能终止");
    }
}

```

```

for (int i = 0; i < 10; i++){
    MyCallable myCallable = new MyCallable((long)i);
    futureList.add(myCallable);
}
System.out.println("Start");
try{
List<Future<Long>> futures = service.invokeAll(futureList);
} catch(Exception err){
    err.printStackTrace();
}
System.out.println("Completed");
service.shutdown();
}
public static void main(String args[]){
    InvokeAllDemo demo = new InvokeAllDemo();
}
class MyCallable implements Callable<Long>{
    Long id = 0L;
    public MyCallable(Long val){
        this.id = val;
    }
    public Long call(){
        // Add your business logic
        return id;
    }
}

```

2. CountDownLatch

A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

A **CountDownLatch** is initialized with a given count. The await methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of await return immediately. This is a one-shot phenomenon -- the count cannot be reset. If you need a version that resets the count, consider using a **CyclicBarrier**.

3. ForkJoinPool or newWorkStealingPool() in Executors

4. Iterate through all Future objects created after submitting to ExecutorService

5. Recommended way of shutdown from oracle documentation page of ExecutorService:

```

void shutdownAndAwaitTermination(ExecutorService pool) {
    pool.shutdown(); // Disable new tasks from being submitted
    try {
        // Wait a while for existing tasks to terminate
        if (!pool.awaitTermination(60, TimeUnit.SECONDS)) {
            pool.shutdownNow(); // Cancel currently executing tasks
        // Wait a while for tasks to respond to being cancelled
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
            System.err.println("Pool did not terminate");
    }
}

```

```

} catch (InterruptedException ie) {
    // 如果当前线程也被中断，则（重新）取消
    pool.shutdownNow();
    // 保留中断状态
    Thread.currentThread().interrupt();
}

```

shutdown(): 启动有序关闭，已提交的任务会被执行，但不接受新任务。

shutdownNow(): 尝试停止所有正在执行的任务，停止处理等待任务，并返回等待执行任务的列表。

在上述示例中，如果您的任务完成时间较长，您可以将 if 条件更改为 while 条件

替换

```
if (!pool.awaitTermination(60, TimeUnit.SECONDS))
```

为

```

while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
    Thread.sleep(60000);
}

```

```

} catch (InterruptedException ie) {
    // (Re-)Cancel if current thread also interrupted
    pool.shutdownNow();
    // Preserve interrupt status
    Thread.currentThread().interrupt();
}

```

shutdown(): Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

shutdownNow(): Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

In above example, if your tasks are taking more time to complete, you can change if condition to while condition

Replace

```
if (!pool.awaitTermination(60, TimeUnit.SECONDS))
```

with

```

while(!pool.awaitTermination(60, TimeUnit.SECONDS)) {
    Thread.sleep(60000);
}

```

第127.8节：不同类型

ExecutorService 的使用场景

[Executors](#) 返回满足特定需求的不同类型线程池。

1. public static ExecutorService newSingleThreadExecutor()

创建一个使用单个工作线程并基于无界队列运行的执行器

正如 Java 文档所述，newFixedThreadPool(1) 和 newSingleThreadExecutor() 之间存在差异，后者说明为：

与其他等效的 newFixedThreadPool(1) 不同，返回的执行器保证不能重新配置为使用额外的线程。

这意味着 newFixedThreadPool 可以在程序后续通过以下方式重新配置：

((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10) 这对于 newSingleThreadExecutor 是不可能的

用例：

1. 您希望按顺序执行提交的任务。
2. 您只需要一个线程来处理所有请求。

Section 127.8: Use cases for different types of ExecutorService

[Executors](#) returns different type of ThreadPools catering to specific need.

1. **public static ExecutorService newSingleThreadExecutor()**

Creates an Executor that uses a single worker thread operating off an unbounded queue

There is a difference between newFixedThreadPool(1) and newSingleThreadExecutor() as the java doc says for the latter:

Unlike the otherwise equivalent newFixedThreadPool(1) the returned executor is guaranteed not to be reconfigurable to use additional threads.

Which means that a newFixedThreadPool can be reconfigured later in the program by:

((ThreadPoolExecutor) fixedThreadPool).setMaximumPoolSize(10) This is not possible for newSingleThreadExecutor

Use cases:

1. You want to execute the submitted tasks in a sequence.
2. You need only one Thread to handle all your request

缺点：

1. 无界队列是有害的。
2. `public static ExecutorService newFixedThreadPool(int nThreads)`

创建一个线程池，重用固定数量的线程，这些线程从共享的无界队列中获取任务。在任何时刻，最多有 `nThreads` 个线程处于活动状态处理任务。如果所有线程都处于活动状态时提交了额外任务，这些任务将等待在队列中，直到有线程可用。

用例：

1. 有效利用可用核心。将 `nThreads` 配置为
`Runtime.getRuntime().availableProcessors()`
2. 当你决定线程池中的线程数不应超过某个数量时

缺点：

1. 无界队列是有害的。
3. `public static ExecutorService newCachedThreadPool()`

创建一个线程池，根据需要创建新线程，但当之前创建的线程可用时会重用它们

用例：

1. 适用于短生命周期的异步任务

缺点：

1. 无界队列是有害的。
2. 如果所有现有线程都忙，每个新任务都会创建一个新线程。如果任务持续时间较长，将创建更多线程，这会降低系统性能。
此情况下的替代方案：`newFixedThreadPool`

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

创建一个线程池，可以安排命令在给定延迟后运行，或周期性执行。

用例：

1. 处理将来在一定时间间隔内发生的带有延迟的重复事件

缺点：

1. 无界队列是有害的。

5. `public static ExecutorService newWorkStealingPool()`

Cons:

1. Unbounded queue is harmful.
2. `public static ExecutorService newFixedThreadPool(int nThreads)`

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most `nThreads` threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available

Use cases:

1. Effective use of available cores. Configure `nThreads` as
`Runtime.getRuntime().availableProcessors()`
2. When you decide that number of thread should not exceed a number in the thread pool

Cons:

1. Unbounded queue is harmful.
3. `public static ExecutorService newCachedThreadPool()`

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available

Use cases:

1. For short-lived asynchronous tasks

Cons:

1. Unbounded queue is harmful.
2. Each new task will create a new thread if all existing threads are busy. If the task is taking long duration, more number of threads will be created, which will degrade the performance of the system.
Alternative in this case: `newFixedThreadPool`

4. `public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)`

Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

Use cases:

1. Handling recurring events with delays, which will happen in future at certain interval of times

Cons:

1. Unbounded queue is harmful.
5. `public static ExecutorService newWorkStealingPool()`

创建一个使用所有可用处理器作为目标并行级别的工作窃取线程池

用例：

- 适用于分治类型的问题。
- 有效利用空闲线程。空闲线程会从繁忙线程中窃取任务。

缺点：

- 无界队列大小是有害的。

你可以看到所有这些 ExecutorService 的一个共同缺点：无界队列。这个问题将在

[ThreadPoolExecutor 中得到解决](#)

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

使用 ThreadPoolExecutor，你可以

- 动态控制线程池大小
- 设置 BlockingQueue 的容量
- 定义队列满时的 RejectionExecutionHandler
- CustomThreadFactory 用于在创建线程时添加一些额外功能 (public Thread newThread(Runnable r))

第127.9节：在固定时间、延迟后或重复运行任务的调度

ScheduledExecutorService 类提供了多种调度单个或重复任务的方法。以下代码示例假设 pool 已声明并初始化如下：

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

除了常规的 ExecutorService 方法外，ScheduledExecutorService API 还增加了4个调度任务并返回 ScheduledFuture 对象的方法。后者可用于获取结果（在某些情况下）和取消任务。

在固定延迟后启动任务

下面的示例安排一个任务在十分钟后启动。

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
    @Override public Integer call() {
        // 执行某些操作
        return 42;
    }
},
10, TimeUnit.MINUTES);
```

以固定频率启动任务

下面的示例安排一个任务在十分钟后开始，然后以每一分钟一次的频率重复执行。

Creates a work-stealing thread pool using all available processors as its target parallelism level

Use cases:

- For divide and conquer type of problems.
- Effective use of idle threads. Idle threads steals tasks from busy threads.

Cons:

- Unbounded queue size is harmful.

You can see one common drawbacks in all these ExecutorService : unbounded queue. This will be addressed with [ThreadPoolExecutor](#)

```
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,
RejectedExecutionHandler handler)
```

With ThreadPoolExecutor, you can

- Control Thread pool size dynamically
- Set the capacity for BlockingQueue
- Define RejectionExecutionHandler when queue is full
- CustomThreadFactory to add some additional functionality during Thread creation (public Thread newThread(Runnable r))

Section 127.9: Scheduling tasks to run at a fixed time, after a delay or repeatedly

The ScheduledExecutorService class provides a methods for scheduling single or repeated tasks in a number of ways. The following code sample assume that pool has been declared and initialized as follows:

```
ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);
```

In addition to the normal ExecutorService methods, the ScheduledExecutorService API adds 4 methods that schedule tasks and return ScheduledFuture objects. The latter can be used to retrieve results (in some cases) and cancel tasks.

Starting a task after a fixed delay

The following example schedules a task to start after ten minutes.

```
ScheduledFuture<Integer> future = pool.schedule(new Callable<>() {
    @Override public Integer call() {
        // do something
        return 42;
    }
},
10, TimeUnit.MINUTES);
```

Starting tasks at a fixed rate

The following example schedules a task to start after ten minutes, and then repeatedly at a rate of once every one minute.

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
    @Override public void run() {
        // 执行某些操作
    }
},
10, 1, TimeUnit.MINUTES);
```

任务执行将按照计划继续，直到pool被关闭，future被取消，或者某个任务遇到异常。

保证由给定的scheduledAtFixedRate调用调度的任务不会时间重叠。如果任务执行时间超过规定周期，则下一次及后续任务的执行可能会延迟开始。

以固定延迟启动任务

以下示例安排一个任务在十分钟后开始，然后在一个任务结束和下一个任务开始之间以一分钟的延迟重复执行。

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
    @Override public void run() {
        // 执行某些操作
    }
},
10, 1, TimeUnit.MINUTES);
```

任务执行将按照计划继续，直到pool被关闭，future被取消，或者某个任务遇到异常。

```
ScheduledFuture<?> future = pool.scheduleAtFixedRate(new Runnable() {
    @Override public void run() {
        // do something
    }
},
10, 1, TimeUnit.MINUTES);
```

Task execution will continue according to the schedule until the pool is shut down, the future is canceled, or one of the tasks encounters an exception.

It is guaranteed that the tasks scheduled by a given scheduledAtFixedRate call will not overlap in time. If a task takes longer than the prescribed period, then the next and subsequent task executions may start late.

Starting tasks with a fixed delay

The following example schedules a task to start after ten minutes, and then repeatedly with a delay of one minute between one task ending and the next one starting.

```
ScheduledFuture<?> future = pool.scheduleWithFixedDelay(new Runnable() {
    @Override public void run() {
        // do something
    }
},
10, 1, TimeUnit.MINUTES);
```

Task execution will continue according to the schedule until the pool is shut down, the future is canceled, or one of the tasks encounters an exception.

第127.10节：使用线程池

线程池主要用于调用ExecutorService中的方法。

以下方法可用于提交工作以执行：

方法	描述
submit	执行提交的工作并返回一个future，可用于获取结果
execute	在未来某个时间执行任务，但不获取任何返回值
invokeAll	执行一组任务并返回一个Future列表
invokeAny	执行所有任务，但只返回其中一个成功（无异常）的结果

完成线程池操作后，可以调用shutdown()来终止线程池。这会执行所有待处理的任务。若想等待所有任务执行完毕，可以循环调用awaitTermination或isShutdown()。

Section 127.10: Using Thread Pools

Thread Pools are used mostly calling methods in ExecutorService.

The following methods can be used to submit work for execution:

Method	Description
submit	Executes a the submitted work and return a future which can be used to get the result
execute	Execute the task sometime in the future without getting any return value
invokeAll	Execute a list of tasks and return a list of Futures
invokeAny	Executes all the but return only the result of one that has been successful (without exceptions)

Once you are done with the Thread Pool you can call shutdown() to terminate the Thread Pool. This executes all pending tasks. To wait for all tasks to execute you can can loop around awaitTermination or isShutdown().

第128章 : ThreadLocal

第128.1节 : ThreadLocal的基本用法

Java `ThreadLocal` 用于创建线程局部变量。众所周知，一个对象的线程共享其变量，因此该变量不是线程安全的。我们可以使用同步来保证线程安全，但如果想避免同步，`ThreadLocal` 允许我们创建线程局部变量，即只有该线程可以读取或写入这些变量，因此执行相同代码的其他线程无法访问彼此的`ThreadLocal` 变量。

这可以用于有线程池的场景，比如在 web 服务中。例如，每次请求都创建一个 `SimpleDateFormat` 对象非常耗时，且由于 `SimpleDateFormat` 不是线程安全的，无法创建静态对象，因此我们可以创建一个 `ThreadLocal`，从而在线程安全的前提下避免每次都创建 `SimpleDateFormat` 的开销。

下面的代码片段展示了它的用法：

每个线程都有自己的 `ThreadLocal` 变量，可以使用 `get()` 和 `set()` 方法获取默认值或更改该线程局部的值。

`ThreadLocal` 实例通常是希望将状态与线程关联的类中的私有静态字段。

下面是一个小示例，展示了 Java 程序中 `ThreadLocal` 的使用，并证明每个线程都有自己的 `ThreadLocal` 变量副本。

```
package com.examples.threads;

import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat 不是线程安全的，所以给每个线程一个实例
    // SimpleDateFormat 不是线程安全的，所以给每个线程一个实例
    private static final ThreadLocal<SimpleDateFormat> formatter = new
    ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue()
        {
            return new SimpleDateFormat("yyyyMMdd HHmm");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("线程名= "+Thread.currentThread().getName()+" 默认格式化器 =
"+formatter.get().toPattern());
        try {

```

Chapter 128: ThreadLocal

Section 128.1: Basic ThreadLocal usage

Java `ThreadLocal` 用于创建线程局部变量。已知线程共享其变量，因此该变量不是线程安全的。我们可以使用同步来保证线程安全，但如果想避免同步，`ThreadLocal` 允许我们创建线程局部变量，即只有该线程可以读取或写入这些变量，因此执行相同代码的其他线程无法访问彼此的 `ThreadLocal` 变量。

This can be used we can use `ThreadLocal` variables in situations where you have a thread pool like for example in a web service. For example, Creating a `SimpleDateFormat` object every time for every request is time consuming and a Static one cannot be created as `SimpleDateFormat` is not thread safe, so we can create a `ThreadLocal` so that we can perform thread safe operations without the overhead of creating `SimpleDateFormat` every time.

The below piece of code shows how it can be used:

Every thread has its own `ThreadLocal` variable and they can use its `get()` and `set()` methods to get the default value or change its value local to Thread.

`ThreadLocal` instances are typically private static fields in classes that wish to associate state with a thread.

Here is a small example showing use of `ThreadLocal` in java program and proving that every thread has its own copy of `ThreadLocal` variable.

```
package com.examples.threads;

import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat is not thread-safe, so give one to each thread
    // SimpleDateFormat is not thread-safe, so give one to each thread
    private static final ThreadLocal<SimpleDateFormat> formatter = new
    ThreadLocal<SimpleDateFormat>(){
        @Override
        protected SimpleDateFormat initialValue()
        {
            return new SimpleDateFormat("yyyyMMdd HHmm");
        }
    };

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name= "+Thread.currentThread().getName()+" default Formatter =
"+formatter.get().toPattern());
        try {

```

```

        Thread.sleep(new Random().nextInt(1000));
    } catch (InterruptedException e) {
e.printStackTrace();
}

formatter.set(new SimpleDateFormat());

System.out.println("线程名= "+Thread.currentThread().getName()+" formatter =
"+formatter.get().toPattern());
}

```

输出：

线程名= 0 默认 格式化器 = yyyyMMdd HHmm

线程名= 1 默认 格式化器 = yyyyMMdd HHmm

线程名= 0 格式化器 = M/d/yy h:mm a

线程名= 2 默认 格式化器 = yyyyMMdd HHmm

线程名= 1 格式化器 = M/d/yy h:mm a

线程名= 3 默认 格式化器 = yyyyMMdd HHmm

线程名= 4 默认 格式化器 = yyyyMMdd HHmm

线程名= 4 格式化器 = M/d/yy h:mm a

线程名= 5 默认 格式化器 = yyyyMMdd HHmm

线程名= 2 格式化器 = M/d/yy h:mm a

线程名= 3 格式化器 = M/d/yy h:mm a

线程名= 6 默认 格式化器 = yyyyMMdd HHmm

线程名= 5 格式化器 = M/d/yy h:mm a

线程名= 6 格式化器 = M/d/yy h:mm a

线程名= 7 默认 格式化器 = yyyyMMdd HHmm

线程名= 8 默认 格式化器 = yyyyMMdd HHmm

线程名= 8 格式化器 = M/d/yy h:mm a

线程名= 7 格式化器 = M/d/yy h:mm a

线程名= 9 默认 格式化器 = yyyyMMdd HHmm

线程名= 9 格式化器 = M/d/yy h:mm a

从输出中可以看到，线程-0 已经更改了格式化器的值，但线程-2 的默认格式化器仍然与初始化值相同。

第128.2节：ThreadLocal Java 8函数式初始化

```

public static class ThreadLocalExample
{

```

```

        Thread.sleep(new Random().nextInt(1000));
    } catch (InterruptedException e) {
e.printStackTrace();
}

formatter.set(new SimpleDateFormat());

System.out.println("Thread Name= "+Thread.currentThread().getName()+" formatter =
"+formatter.get().toPattern());
}

```

Output:

Thread Name= 0 **default** Formatter = yyyyMMdd HHmm

Thread Name= 1 **default** Formatter = yyyyMMdd HHmm

Thread Name= 0 formatter = M/d/yy h:mm a

Thread Name= 2 **default** Formatter = yyyyMMdd HHmm

Thread Name= 1 formatter = M/d/yy h:mm a

Thread Name= 3 **default** Formatter = yyyyMMdd HHmm

Thread Name= 4 **default** Formatter = yyyyMMdd HHmm

Thread Name= 4 formatter = M/d/yy h:mm a

Thread Name= 5 **default** Formatter = yyyyMMdd HHmm

Thread Name= 2 formatter = M/d/yy h:mm a

Thread Name= 3 formatter = M/d/yy h:mm a

Thread Name= 6 **default** Formatter = yyyyMMdd HHmm

Thread Name= 5 formatter = M/d/yy h:mm a

Thread Name= 6 formatter = M/d/yy h:mm a

Thread Name= 7 **default** Formatter = yyyyMMdd HHmm

Thread Name= 8 **default** Formatter = yyyyMMdd HHmm

Thread Name= 8 formatter = M/d/yy h:mm a

Thread Name= 7 formatter = M/d/yy h:mm a

Thread Name= 9 **default** Formatter = yyyyMMdd HHmm

Thread Name= 9 formatter = M/d/yy h:mm a

As we can see from the output that Thread-0 has changed the value of formatter but still thread-2 default formatter is same as the initialized value.

Section 128.2: ThreadLocal Java 8 functional initialization

```

public static class ThreadLocalExample
{

```

```

private static final ThreadLocal<SimpleDateFormat> format =
    ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd_HHmm"));

public String formatDate(Date date)
{
    return format.get().format(date);
}

```

第128.3节：多个线程共享一个对象

在这个例子中，我们只有一个对象，但它被不同线程共享/执行。普通使用字段保存状态是不可能的，因为其他线程也会看到（或者可能看不到）该状态。

```

public class Test {
    public static void main(String[] args) {
        Foo foo = new Foo();
        new Thread(foo, "Thread 1").start();
        new Thread(foo, "Thread 2").start();
    }
}

```

在 Foo 中，我们从零开始计数。我们没有将状态保存到字段中，而是将当前数字存储在静态可访问的 ThreadLocal 对象中。注意，本例中的同步与 ThreadLocal 的使用无关，而是为了确保更好的控制台输出效果。

```

public class Foo implements Runnable {
    private static final int ITERATIONS = 10;
    private static final ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    @Override
    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            synchronized (threadLocal) {
                //虽然访问的是静态字段，但我们获得的是自己的（之前保存的）值。
                int value = threadLocal.get();
                System.out.println(Thread.currentThread().getName() + ": " + value);

                //更新我们自己的变量
                threadLocal.set(value + 1);

                try {
                    threadLocal.notifyAll();
                    if (i < ITERATIONS - 1) {
                        threadLocal.wait();
                    }
                } catch (InterruptedException ex) {
                }
            }
        }
    }
}

```

从输出中我们可以看到，每个线程都是自己计数的，并没有使用另一个线程的值：

```

private static final ThreadLocal<SimpleDateFormat> format =
    ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd_HHmm"));

public String formatDate(Date date)
{
    return format.get().format(date);
}

```

Section 128.3: Multiple threads with one shared object

In this example we have only one object but it is shared between/executed on different threads. Ordinary usage of fields to save state would not be possible because the other thread would see that too (or probably not see).

```

public class Test {
    public static void main(String[] args) {
        Foo foo = new Foo();
        new Thread(foo, "Thread 1").start();
        new Thread(foo, "Thread 2").start();
    }
}

```

In Foo we count starting from zero. Instead of saving the state to a field we store our current number in the ThreadLocal object which is statically accessible. Note that the synchronization in this example is not related to the usage of ThreadLocal but rather ensures a better console output.

```

public class Foo implements Runnable {
    private static final int ITERATIONS = 10;
    private static final ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>() {
        @Override
        protected Integer initialValue() {
            return 0;
        }
    };

    @Override
    public void run() {
        for (int i = 0; i < ITERATIONS; i++) {
            synchronized (threadLocal) {
                //Although accessing a static field, we get our own (previously saved) value.
                int value = threadLocal.get();
                System.out.println(Thread.currentThread().getName() + ": " + value);

                //Update our own variable
                threadLocal.set(value + 1);

                try {
                    threadLocal.notifyAll();
                    if (i < ITERATIONS - 1) {
                        threadLocal.wait();
                    }
                } catch (InterruptedException ex) {
                }
            }
        }
    }
}

```

From the output we can see that each thread counts for itself and does not use the value of the other one:

```
线程 1: 0  
线程 2: 0  
线程 1: 1  
线程 2: 1  
线程 1: 2  
线程 2: 2  
线程 1: 3  
线程 2: 3  
线程 1: 4  
线程 2: 4  
线程 1: 5  
线程 2: 5  
线程 1: 6  
线程 2: 6  
线程 1: 7  
线程 2: 7  
线程 1: 8  
线程 2: 8  
线程 1: 9  
线程 2: 9
```

```
Thread 1: 0  
Thread 2: 0  
Thread 1: 1  
Thread 2: 1  
Thread 1: 2  
Thread 2: 2  
Thread 1: 3  
Thread 2: 3  
Thread 1: 4  
Thread 2: 4  
Thread 1: 5  
Thread 2: 5  
Thread 1: 6  
Thread 2: 6  
Thread 1: 7  
Thread 2: 7  
Thread 1: 8  
Thread 2: 8  
Thread 1: 9  
Thread 2: 9
```

第129章：在多线程应用中使用ThreadPoolExecutor。

在创建高性能且数据驱动的应用时，以异步方式完成耗时任务并让多个任务并发运行是非常有帮助的。本章节将介绍使用ThreadPoolExecutor来并发完成多个异步任务的概念。

第129.1节：使用Runnable类实例执行无返回值的异步任务

有些应用可能希望创建所谓的“火并忘记”（Fire & Forget）任务，这类任务可以周期性触发，且在完成分配的任务后不需要返回任何类型的值（例如，清理旧临时文件、日志轮转、自动保存状态）。

在此示例中，我们将创建两个类：一个实现 Runnable 接口，另一个包含 main() 方法。

AsyncMaintenanceTaskCompleter.java

```
import lombok.extern.java.Log;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncMaintenanceTaskCompleter implements Runnable {
    private int taskNumber;

    public AsyncMaintenanceTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    public void run() {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping for %d seconds", taskNumber, timeout));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));
        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
    }
}
```

AsyncExample1

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AsyncExample1 {
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        for(int i = 0; i < 10; i++){
            executorService.execute(new AsyncMaintenanceTaskCompleter(i));
        }
    }
}
```

Chapter 129: Using ThreadPoolExecutor in MultiThreaded applications.

When creating a performant and data-driven application, it can be very helpful to complete time-intensive tasks in an asynchronous manner and to have multiple tasks running concurrently. This topic will introduce the concept of using ThreadPoolExecutors to complete multiple asynchronous tasks concurrently.

Section 129.1: Performing Asynchronous Tasks Where No Return Value Is Needed Using a Runnable Class Instance

Some applications may want to create so-called "Fire & Forget" tasks which can be periodically triggered and do not need to return any type of value returned upon completion of the assigned task (for example, purging old temp files, rotating logs, autosaving state).

In this example, we will create two classes: One which implements the Runnable interface, and one which contains a main() method.

AsyncMaintenanceTaskCompleter.java

```
import lombok.extern.java.Log;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncMaintenanceTaskCompleter implements Runnable {
    private int taskNumber;

    public AsyncMaintenanceTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    public void run() {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping for %d seconds", taskNumber, timeout));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));
        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
    }
}
```

AsyncExample1

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class AsyncExample1 {
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        for(int i = 0; i < 10; i++){
            executorService.execute(new AsyncMaintenanceTaskCompleter(i));
        }
    }
}
```

```
executorService.shutdown();  
}  
}
```

运行 AsyncExample1.main() 产生了以下输出：

```
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务 8 正在休眠 18 秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务 6 正在休眠 4 秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务 2 正在休眠 6 秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务3正在休眠4秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务9正在休眠14秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务4正在休眠9秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务5正在休眠10秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务0正在休眠7秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务1正在休眠9秒  
2016年12月28日 下午2:21:03 AsyncMaintenanceTaskCompleter 运行  
信息：任务7正在休眠8秒  
2016年12月28日 下午2:21:07 AsyncMaintenanceTaskCompleter 运行  
信息：任务6休眠结束  
2016年12月28日 下午2:21:07 AsyncMaintenanceTaskCompleter 运行  
信息：任务3休眠结束  
2016年12月28日 下午2:21:09 AsyncMaintenanceTaskCompleter 运行  
信息：任务2休眠结束  
2016年12月28日 下午2:21:10 AsyncMaintenanceTaskCompleter 运行  
信息：任务0休眠结束  
2016年12月28日 下午2:21:11 AsyncMaintenanceTaskCompleter 运行  
信息：任务7已完成休眠  
2016年12月28日 下午2:21:12 AsyncMaintenanceTaskCompleter 运行  
信息：任务4已完成休眠  
2016年12月28日 下午2:21:12 AsyncMaintenanceTaskCompleter 运行  
信息：任务1已完成休眠  
2016年12月28日 下午2:21:13 AsyncMaintenanceTaskCompleter 运行  
信息：任务5已完成休眠  
2016年12月28日 下午2:21:17 AsyncMaintenanceTaskCompleter 运行  
信息：任务9已完成休眠  
2016年12月28日 下午2:21:21 AsyncMaintenanceTaskCompleter 运行  
信息：任务8已完成休眠
```

进程以退出代码0结束

值得注意的观察： 上述输出中有几点需要注意，

1. 任务执行的顺序不可预测。
2. 由于每个任务休眠的时间是（伪）随机的，它们不一定按照调用的顺序完成。

第129.2节：使用可调用类实例执行需要返回值的异步任务

通常需要执行一个长时间运行的任务，并在该任务完成后使用其结果。

```
executorService.shutdown();  
}  
}
```

Running AsyncExample1.main() resulted in the following output:

```
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 8 is sleeping for 18 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 6 is sleeping for 4 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 2 is sleeping for 6 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 3 is sleeping for 4 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 9 is sleeping for 14 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 4 is sleeping for 9 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 5 is sleeping for 10 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 0 is sleeping for 7 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 1 is sleeping for 9 seconds  
Dec 28, 2016 2:21:03 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 7 is sleeping for 8 seconds  
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 6 is done sleeping  
Dec 28, 2016 2:21:07 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 3 is done sleeping  
Dec 28, 2016 2:21:09 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 2 is done sleeping  
Dec 28, 2016 2:21:10 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 0 is done sleeping  
Dec 28, 2016 2:21:11 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 7 is done sleeping  
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 4 is done sleeping  
Dec 28, 2016 2:21:12 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 1 is done sleeping  
Dec 28, 2016 2:21:13 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 5 is done sleeping  
Dec 28, 2016 2:21:17 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 9 is done sleeping  
Dec 28, 2016 2:21:21 PM AsyncMaintenanceTaskCompleter run  
INFO: Task 8 is done sleeping
```

Process finished with exit code 0

Observations of Note: There are several things to note in the output above,

1. The tasks did not execute in a predictable order.
2. Since each task was sleeping for a (pseudo)random amount of time, they did not necessarily complete in the order in which they were invoked.

Section 129.2: Performing Asynchronous Tasks Where a Return Value Is Needed Using a Callable Class Instance

It is often necessary to execute a long-running task and use the result of that task once it has completed.

在此示例中，我们将创建两个类：一个实现 Callable<T> 接口（其中 T 是我们希望返回的类型），另一个包含 main() 方法。

AsyncValueTypeTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncValueTypeTaskCompleter implements Callable<Integer> {
    private int taskNumber;

    public AsyncValueTypeTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    @Override
    public Integer call() throws Exception {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("任务 %d 正在休眠", taskNumber));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));

        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
        return timeout;
    }
}
```

AsyncExample2.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

@Log
public class AsyncExample2 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            Future<Integer> submittedFuture = executorService.submit(new
                AsyncValueTypeTaskCompleter(i));
            futures.add(submittedFuture);
        }
        executorService.shutdown();
        while (!futures.isEmpty()) {
            for (int j = 0; j < futures.size(); j++) {
                Future<Integer> f = futures.get(j);
                if (f.isDone()) {
                    try {
```

In this example, we will create two classes: One which implements the Callable<T> interface (where T is the type we wish to return), and one which contains a main() method.

AsyncValueTypeTaskCompleter.java

```
import lombok.extern.java.Log;

import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.TimeUnit;

@Log
public class AsyncValueTypeTaskCompleter implements Callable<Integer> {
    private int taskNumber;

    public AsyncValueTypeTaskCompleter(int taskNumber) {
        this.taskNumber = taskNumber;
    }

    @Override
    public Integer call() throws Exception {
        int timeout = ThreadLocalRandom.current().nextInt(1, 20);
        try {
            log.info(String.format("Task %d is sleeping", taskNumber));
            TimeUnit.SECONDS.sleep(timeout);
            log.info(String.format("Task %d is done sleeping", taskNumber));

        } catch (InterruptedException e) {
            log.warning(e.getMessage());
        }
        return timeout;
    }
}
```

AsyncExample2.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

@Log
public class AsyncExample2 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            Future<Integer> submittedFuture = executorService.submit(new
                AsyncValueTypeTaskCompleter(i));
            futures.add(submittedFuture);
        }
        executorService.shutdown();
        while (!futures.isEmpty()) {
            for (int j = 0; j < futures.size(); j++) {
                Future<Integer> f = futures.get(j);
                if (f.isDone()) {
                    try {
```

```
        int timeout = f.get();
        log.info(String.format("一个任务刚刚完成，睡眠了 %d
秒", timeout));
        futures.remove(f);
    } catch (InterruptedException | ExecutionException e) {
        log.warning(e.getMessage());
    }
}
}
}
}
```

运行 AsyncExample2.main() 产生了以下输出：

```
2016年12月28日 下午3:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务7正在睡眠
2016年12月28日 下午3:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务8正在睡眠
2016年12月28日 下午3:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务2正在睡眠
2016年12月28日 下午3:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务1正在睡眠
2016年12月28日 下午3:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务4正在睡眠
2016年12月28日 15:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务9正在休眠
2016年12月28日 15:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务0正在休眠
2016年12月28日 15:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务6正在休眠
2016年12月28日 15:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务5正在休眠
2016年12月28日 15:07:15 AsyncValueTypeTaskCompleter 调用
信息：任务3正在休眠
2016年12月28日 15:07:16 AsyncValueTypeTaskCompleter 调用
信息：任务8休眠结束
2016年12月28日 15:07:16 AsyncExample2 主程序
信息：一个任务刚刚完成，休眠了1秒
2016年12月28日 15:07:17 AsyncValueTypeTaskCompleter 调用
信息：任务2休眠结束
2016年12月28日 15:07:17 AsyncExample2 主程序
信息：一个任务刚刚完成，休眠了2秒
2016年12月28日 15:07:17 AsyncValueTypeTaskCompleter 调用
信息：任务9休眠结束
2016年12月28日 15:07:17 AsyncExample2 主程序
信息：一个任务刚刚完成，休眠了2秒
2016年12月28日 15:07:19 AsyncValueTypeTaskCompleter 调用
信息：任务3休眠结束
2016年12月28日 15:07:19 AsyncExample2 主程序
信息：一个任务刚刚完成，休眠了4秒
2016年12月28日 15:07:20 AsyncValueTypeTaskCompleter 调用
信息：任务0休眠结束
2016年12月28日 下午3:07:20 AsyncExample2 主程序
信息：一个任务在休眠5秒后刚刚完成
2016年12月28日 下午3:07:21 AsyncValueTypeTaskCompleter 调用
信息：任务5的休眠已完成
2016年12月28日 下午3:07:21 AsyncExample2 主程序
信息：一个任务在休眠6秒后刚刚完成
2016年12月28日 下午3:07:25 AsyncValueTypeTaskCompleter 调用
信息：任务1已完成休眠
2016年12月28日 下午3:07:25 AsyncExample2 主程序
```

```
        int timeout = f.get();
        log.info(String.format("A task just completed after sleeping for %d
seconds", timeout));
        futures.remove(f);
    } catch (InterruptedException | ExecutionException e) {
        log.warning(e.getMessage());
    }
}
}
}
}
```

Running AsyncExample2.main() resulted in the following output:

```
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 7 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 4 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 6 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is sleeping
Dec 28, 2016 3:07:15 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is sleeping
Dec 28, 2016 3:07:16 PM AsyncValueTypeTaskCompleter call
INFO: Task 8 is done sleeping
Dec 28, 2016 3:07:16 PM AsyncExample2 main
INFO: A task just completed after sleeping for 1 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 2 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:17 PM AsyncValueTypeTaskCompleter call
INFO: Task 9 is done sleeping
Dec 28, 2016 3:07:17 PM AsyncExample2 main
INFO: A task just completed after sleeping for 2 seconds
Dec 28, 2016 3:07:19 PM AsyncValueTypeTaskCompleter call
INFO: Task 3 is done sleeping
Dec 28, 2016 3:07:19 PM AsyncExample2 main
INFO: A task just completed after sleeping for 4 seconds
Dec 28, 2016 3:07:20 PM AsyncValueTypeTaskCompleter call
INFO: Task 0 is done sleeping
Dec 28, 2016 3:07:20 PM AsyncExample2 main
INFO: A task just completed after sleeping for 5 seconds
Dec 28, 2016 3:07:21 PM AsyncValueTypeTaskCompleter call
INFO: Task 5 is done sleeping
Dec 28, 2016 3:07:21 PM AsyncExample2 main
INFO: A task just completed after sleeping for 6 seconds
Dec 28, 2016 3:07:25 PM AsyncValueTypeTaskCompleter call
INFO: Task 1 is done sleeping
Dec 28, 2016 3:07:25 PM AsyncExample2 main
```

信息：一个任务在休眠10秒后刚刚完成
 2016年12月28日 下午3:07:27 AsyncValueTypeTaskCompleter 调用
 信息：任务6已完成休眠
 2016年12月28日 下午3:07:27 AsyncExample2 主程序
 信息：一个任务在休眠12秒后刚刚完成
 2016年12月28日 下午3:07:29 AsyncValueTypeTaskCompleter 调用
 信息：任务7已完成休眠
 2016年12月28日 下午3:07:29 AsyncExample2 主程序
 信息：一个任务在休眠14秒后刚刚完成
 2016年12月28日 下午3:07:31 AsyncValueTypeTaskCompleter 调用
 信息：任务4已完成休眠
 2016年12月28日 下午3:07:31 AsyncExample2 主程序
 信息：一个任务在休眠16秒后刚刚完成

INFO: A task just completed after sleeping for 10 seconds
 Dec 28, 2016 3:07:27 PM AsyncValueTypeTaskCompleter call
 INFO: Task 6 is done sleeping
 Dec 28, 2016 3:07:27 PM AsyncExample2 main
 INFO: A task just completed after sleeping for 12 seconds
 Dec 28, 2016 3:07:29 PM AsyncValueTypeTaskCompleter call
 INFO: Task 7 is done sleeping
 Dec 28, 2016 3:07:29 PM AsyncExample2 main
 INFO: A task just completed after sleeping for 14 seconds
 Dec 28, 2016 3:07:31 PM AsyncValueTypeTaskCompleter call
 INFO: Task 4 is done sleeping
 Dec 28, 2016 3:07:31 PM AsyncExample2 main
 INFO: A task just completed after sleeping for 16 seconds

值得注意的观察：

以上输出中有几点需要注意，

- 每次调用 `ExecutorService.submit()` 都返回一个 `Future` 实例，该实例被存储在列表中以备后用。2. `Future` 包含一个名为 `isDone()` 的方法，可用于在尝试检查返回值之前确认任务是否已完成。对尚未完成的 `Future` 调用 `Future.get()` 方法会阻塞当前线程，直到任务完成，这可能会抵消异步执行任务所带来的许多好处。
- 在检查 `Future` 对象的返回值之前调用了 `executorService.shutdown()` 方法。
 这不是必须的，但这样做是为了展示这是可行的。`executorService.shutdown()` 方法不会阻止已经提交给 `ExecutorService` 的任务完成，而是阻止新的任务被添加到队列中。

第129.3节：使用 Lambda 内联定义异步任务

虽然良好的软件设计通常最大化代码的可重用性，但有时通过 `Lambda` 表达式在代码中内联定义异步任务可以最大化代码的可读性。

在本例中，我们将创建一个包含 `main()` 方法的单一类。在该方法内部，我们将使用 `Lambda` 表达式创建并执行 `Callable` 和 `Runnable<T>` 的实例。

AsyncExample3.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

@Log
public class AsyncExample3 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for(int i = 0; i < 5; i++){
            final int index = i;
            executorService.execute(() -> {
                int timeout = getTimeout();
                log.info(String.format("Runnable %d 已提交，将休眠 %d 秒", index, timeout));
                try {

```

Observations of Note:

There are several things to note in the output above,

- Each call to `ExecutorService.submit()` returned an instance of `Future`, which was stored in a list for later use
- `Future` contains a method called `isDone()` which can be used to check whether our task has been completed before attempting to check its return value. Calling the `Future.get()` method on a `Future` that is not yet done will block the current thread until the task is complete, potentially negating many benefits gained from performing the task Asynchronously.
- The `executorService.shutdown()` method was called prior to checking the return values of the `Future` objects. This is not required, but was done in this way to show that it is possible. The `executorService.shutdown()` method does not prevent the completion of tasks which have already been submitted to the `ExecutorService`, but rather prevents new tasks from being added to the Queue.

Section 129.3: Defining Asynchronous Tasks Inline using Lambdas

While good software design often maximizes code reusability, sometimes it can be useful to define asynchronous tasks inline in your code via `Lambda` expressions to maximize code readability.

In this example, we will create a single class which contains a `main()` method. Inside this method, we will use `Lambda` expressions to create and execute instances of `Callable` and `Runnable<T>`.

AsyncExample3.java

```
import lombok.extern.java.Log;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

@Log
public class AsyncExample3 {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<Integer>> futures = new ArrayList<>();
        for(int i = 0; i < 5; i++){
            final int index = i;
            executorService.execute(() -> {
                int timeout = getTimeout();
                log.info(String.format("Runnable %d has been submitted and will sleep for %d seconds", index, timeout));
                try {

```

```

        TimeUnit.SECONDS.sleep(timeout);
    } catch (InterruptedException e) {
        log.warning(e.getMessage());
    }
    log.info(String.format("Runnable %d 已完成休眠", index));
}
Future<Integer> submittedFuture = executorService.submit(() -> {
    int timeout = getTimeout();
    log.info(String.format("Callable %d 将开始休眠", index));
    try {
        TimeUnit.SECONDS.sleep(timeout);
    } catch (InterruptedException e) {
        log.warning(e.getMessage());
    }
    log.info(String.format("Callable %d 已完成休眠", index));
    return timeout;
});
futures.add(submittedFuture);
}
executorService.shutdown();
while(!futures.isEmpty()){
    for(int j = 0; j < futures.size(); j++){
        Future<Integer> f = futures.get(j);
        if(f.isDone()){
            try {
                int timeout = f.get();
                log.info(String.format("一个任务刚刚完成，睡眠了 %d 秒", timeout));
                futures.remove(f);
            } catch (InterruptedException | ExecutionException e) {
                log.warning(e.getMessage());
            }
        }
    }
}

public static int getTimeout(){
    return ThreadLocalRandom.current().nextInt(1, 20);
}
}

```

值得注意的观察：

以上输出中有几点需要注意，

1. Lambda 表达式可以访问定义它们的作用域中可用的变量和方法，但所有变量必须是 final (或实际上是 final) 的，才能在 lambda 表达式内部使用。
2. 我们不必显式指定 Lambda 表达式是 Callable 还是 Runnable<T>，返回类型会自动根据返回值推断。

```

        TimeUnit.SECONDS.sleep(timeout);
    } catch (InterruptedException e) {
        log.warning(e.getMessage());
    }
    log.info(String.format("Runnable %d has finished sleeping", index));
}
Future<Integer> submittedFuture = executorService.submit(() -> {
    int timeout = getTimeout();
    log.info(String.format("Callable %d will begin sleeping", index));
    try {
        TimeUnit.SECONDS.sleep(timeout);
    } catch (InterruptedException e) {
        log.warning(e.getMessage());
    }
    log.info(String.format("Callable %d is done sleeping", index));
    return timeout;
});
futures.add(submittedFuture);
}
executorService.shutdown();
while(!futures.isEmpty()){
    for(int j = 0; j < futures.size(); j++){
        Future<Integer> f = futures.get(j);
        if(f.isDone()){
            try {
                int timeout = f.get();
                log.info(String.format("A task just completed after sleeping for %d seconds", timeout));
                futures.remove(f);
            } catch (InterruptedException | ExecutionException e) {
                log.warning(e.getMessage());
            }
        }
    }
}

public static int getTimeout(){
    return ThreadLocalRandom.current().nextInt(1, 20);
}
}

```

Observations of Note:

There are several things to note in the output above,

1. Lambda expressions have access to variables and methods which are available to the scope in which they are defined, but all variables must be final (or effectively final) for use inside a lambda expression.
2. We do not have to specify whether our Lambda expression is a Callable or a Runnable<T> explicitly, the return type is inferred automatically by the return type.

第130章：常见的 Java 陷阱

本主题概述了 Java 初学者常犯的一些错误。

这包括在使用 Java 语言或理解运行时环境时的常见错误。

与特定 API 相关的错误可以在针对这些 API 的专题中描述。字符串是一个特殊情况；它们在 Java 语言规范中有说明。除常见错误外，关于字符串的其他细节可以在本主题中描述。

第130.1节：陷阱：使用 == 比较原始包装类对象，如 Integer

(此陷阱同样适用于所有原始包装类型，但我们将以Integer和int为例说明。) 在处理Integer对象时，使用==比较值是很诱人的，因为这正是你对int值所做的操作。在某些情况下，这似乎是可行的：

```
Integer int1_1 = Integer.valueOf("1");
Integer int1_2 = Integer.valueOf(1);

System.out.println("int1_1 == int1_2: " + (int1_1 == int1_2));          // true
System.out.println("int1_1 equals int1_2: " + int1_1.equals(int1_2));    // true
```

这里我们创建了两个值为Integer对象1，并对它们进行了比较（在本例中，我们创建了一个来自字符串和一个来自int字面量。还有其他选择）。此外，我们观察到两种比较方法（==和equals）都返回了true。

当我们选择不同的值时，这种行为会发生变化：

```
Integer int2_1 = Integer.valueOf("1000");
Integer int2_2 = Integer.valueOf(1000);

System.out.println("int2_1 == int2_2: " + (int2_1 == int2_2));          // false
System.out.println("int2_1 equals int2_2: " + int2_1.equals(int2_2));    // true
```

在这种情况下，只有equals比较返回了正确的结果。

这种行为差异的原因是，JVM维护了一个Integer对象的缓存，范围是-128到127。（上限值可以通过系统属性"java.lang.Integer.IntegerCache.high"或JVM参数"-XX:AutoBoxCacheMax=size"来覆盖）。对于该范围内的值，Integer.valueOf()会返回缓存的值，而不是创建新的对象。

因此，在第一个例子中，Integer.valueOf(1)和Integer.valueOf("1")调用返回了相同的缓存Integer实例。相比之下，在第二个例子中，Integer.valueOf(1000)和Integer.valueOf("1000")都创建并返回了新的Integer对象。

引用类型的==运算符测试的是引用相等性（即是否是同一个对象）。因此，在第一个例子中，int1_1 == int1_2为true，因为引用相同。在第二个例子中，int2_1 == int2_2为false，因为引用不同。

第130.2节：陷阱：使用 == 比较字符串

Java初学者常犯的一个错误是使用==运算符来测试两个字符串是否相等。例如：

Chapter 130: Common Java Pitfalls

This topic outlines some of the common mistakes made by beginners in Java.

This includes any common mistakes in use of the Java language or understanding of the run-time environment.

Mistakes associated with specific APIs can be described in topics specific to those APIs. Strings are a special case; they're covered in the Java Language Specification. Details other than common mistakes can be described in this topic on Strings.

Section 130.1: Pitfall: using == to compare primitive wrappers objects such as Integer

(This pitfall applies equally to all primitive wrapper types, but we will illustrate it for Integer and int.)

When working with Integer objects, it is tempting to use == to compare values, because that is what you would do with int values. And in some cases this will seem to work:

```
Integer int1_1 = Integer.valueOf("1");
Integer int1_2 = Integer.valueOf(1);

System.out.println("int1_1 == int1_2: " + (int1_1 == int1_2));          // true
System.out.println("int1_1 equals int1_2: " + int1_1.equals(int1_2));    // true
```

Here we created two Integer objects with the value 1 and compare them (In this case we created one from a String and one from an int literal. There are other alternatives). Also, we observe that the two comparison methods (== and equals) both yield true.

This behavior changes when we choose different values:

```
Integer int2_1 = Integer.valueOf("1000");
Integer int2_2 = Integer.valueOf(1000);

System.out.println("int2_1 == int2_2: " + (int2_1 == int2_2));          // false
System.out.println("int2_1 equals int2_2: " + int2_1.equals(int2_2));    // true
```

In this case, only the equals comparison yields the correct result.

The reason for this difference in behavior is, that the JVM maintains a cache of Integer objects for the range -128 to 127. (The upper value can be overridden with the system property "java.lang.Integer.IntegerCache.high" or the JVM argument "-XX:AutoBoxCacheMax=size"). For values in this range, the Integer.valueOf() will return the cached value rather than creating a new one.

Thus, in the first example the Integer.valueOf(1) and Integer.valueOf("1") calls returned the same cached Integer instance. By contrast, in the second example the Integer.valueOf(1000) and Integer.valueOf("1000") both created and returned new Integer objects.

The == operator for reference types tests for reference equality (i.e. the same object). Therefore, in the first example int1_1 == int1_2 is true because the references are the same. In the second example int2_1 == int2_2 is false because the references are different.

Section 130.2: Pitfall: using == to compare strings

A common mistake for Java beginners is to use the == operator to test if two strings are equal. For example:

```

public class Hello {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0] == "hello") {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}

```

上述程序本应测试第一个命令行参数，并在其是或不是单词“hello”时打印不同的消息。但问题是它无法正常工作。无论第一个命令行参数是什么，该程序都会输出“你今天心情不好吗？”

在这种特定情况下，String “hello” 被放入字符串池中，而String args[0] 存储在堆上。这意味着有两个对象表示相同的字面量，每个都有自己的引用。由于==测试的是引用而非实际的内容相等，因此比较大多数情况下会返回false。这并不意味着它总是如此。

当你使用==来测试字符串时，实际上你是在测试两个String对象是否是同一个Java对象。
不幸的是，这并不是Java中字符串相等的含义。实际上，测试字符串的正确方法是使用equals(Object)方法。对于一对字符串，我们通常想测试它们是否由相同顺序的相同字符组成。

```

public class Hello2 {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0].equals("hello")) {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}

```

但情况实际上更糟。问题在于==在某些情况下会给出预期的答案。例如

```

public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        if (s1 == s2) {
            System.out.println("same");
        } else {
            System.out.println("different");
        }
    }
}

```

有趣的是，即使我们用错误的方式测试字符串，这也会打印“same”。这是为什么呢？因为Java语言规范（第3.10.5节：字符串字面量）规定，任何两个由相同字符组成的字符串>>字面量<<实际上会被表示为同一个Java对象。因此，==测试对于相等的字面量会返回true。（字符串字面量会被“驻留”并添加到一个共享的“字符串池”中，当你的代码被加载时，但这实际上是一个实现细节。）

```

public class Hello {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0] == "hello") {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}

```

The above program is supposed to test the first command line argument and print different messages when it and isn't the word "hello". But the problem is that it won't work. That program will output "Are you feeling grumpy today?" no matter what the first command line argument is.

In this particular case the `String` "hello" is put in the string pool while the `String` args[0] resides on the heap. This means there are two objects representing the same literal, each with its reference. Since == tests for references, not actual equality, the comparison will yield a false most of the times. This doesn't mean that it will always do so.

When you use == to test strings, what you are actually testing is if two `String` objects are the same Java object. Unfortunately, that is not what string equality means in Java. In fact, the correct way to test strings is to use the equals(`Object`) method. For a pair of strings, we usually want to test if they consist of the same characters in the same order.

```

public class Hello2 {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (args[0].equals("hello")) {
                System.out.println("Hello back to you");
            } else {
                System.out.println("Are you feeling grumpy today?");
            }
        }
    }
}

```

But it actually gets worse. The problem is that == will give the expected answer in some circumstances. For example

```

public class Test1 {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        if (s1 == s2) {
            System.out.println("same");
        } else {
            System.out.println("different");
        }
    }
}

```

Interestingly, this will print "same", even though we are testing the strings the wrong way. Why is that? Because the [Java Language Specification \(Section 3.10.5: String Literals\)](#) stipulates that any two string >> literals<< consisting of the same characters will actually be represented by the same Java object. Hence, the == test will give true for equal literals. (The string literals are "interned" and added to a shared "string pool" when your code is loaded, but that is actually an implementation detail.)

为了增加混淆，Java语言规范还规定，当你有一个编译时常量表达式连接两个字符串字面量时，这等同于一个单一的字面量。因此：

```
public class Test1 {  
    public static void main(String[] args) {  
        String s1 = "hello";  
        String s2 = "hel" + "lo";  
        String s3 = " mum";  
        if (s1 == s2) {  
            System.out.println("1. same");  
        } else {  
            System.out.println("1. different");  
        }  
        if (s1 + s3 == "hello mum") {  
            System.out.println("2. same");  
        } else {  
            System.out.println("2. different");  
        }  
    }  
}
```

这将输出“1. 相同”和“2. 不同”。在第一种情况下，+ 表达式在编译时被计算，我们比较一个String对象与它自身。在第二种情况下，它在运行时被计算，我们比较两个不同的字符串对象

总之，在Java中使用==来测试字符串几乎总是错误的，但这并不保证一定会给出错误的答案。

第130.3节：陷阱：忘记释放资源

每当程序打开一个资源，例如文件或网络连接时，使用完毕后释放该资源非常重要。如果在操作这些资源时抛出任何异常，也应采取类似的谨慎措施。有人可能会认为FileInputStream有一个finalizer，在垃圾回收事件中调用close()方法；然而，由于我们无法确定垃圾回收周期何时开始，输入流可能会无限期地占用计算机资源。资源必须在try-catch块的finally部分关闭：

```
版本 < Java SE 7  
  
private static void printFileJava6() throws IOException {  
    FileInputStream input;  
    try {  
        input = new FileInputStream("file.txt");  
        int data = input.read();  
        while (data != -1){  
            System.out.print((char) data);  
            data = input.read();  
        }  
    } finally {  
        if (input != null) {  
            input.close();  
        }  
    }  
}
```

自Java 7起，特别为这种情况引入了一个非常有用且简洁的语句，称为try-with-resources：

版本 ≥ Java SE 7

To add to the confusion, the Java Language Specification also stipulates that when you have a compile-time constant expression that concatenates two string literals, that is equivalent to a single literal. Thus:

```
public class Test1 {  
    public static void main(String[] args) {  
        String s1 = "hello";  
        String s2 = "hel" + "lo";  
        String s3 = " mum";  
        if (s1 == s2) {  
            System.out.println("1. same");  
        } else {  
            System.out.println("1. different");  
        }  
        if (s1 + s3 == "hello mum") {  
            System.out.println("2. same");  
        } else {  
            System.out.println("2. different");  
        }  
    }  
}
```

This will output "1. same" and "2. different". In the first case, the + expression is evaluated at compile time and we compare one `String` object with itself. In the second case, it is evaluated at run time and we compare two different `String` objects

In summary, using == to test strings in Java is almost always incorrect, but it is not guaranteed to give the wrong answer.

Section 130.3: Pitfall: forgetting to free resources

Every time a program opens a resource, such as a file or network connection, it is important to free the resource once you are done using it. Similar caution should be taken if any exception were to be thrown during operations on such resources. One could argue that the `FileInputStream` has a `finalizer` that invokes the `close()` method on a garbage collection event; however, since we can't be sure when a garbage collection cycle will start, the input stream can consume computer resources for an indefinite period of time. The resource must be closed in a `finally` section of a try-catch block:

```
Version < Java SE 7  
  
private static void printFileJava6() throws IOException {  
    FileInputStream input;  
    try {  
        input = new FileInputStream("file.txt");  
        int data = input.read();  
        while (data != -1){  
            System.out.print((char) data);  
            data = input.read();  
        }  
    } finally {  
        if (input != null) {  
            input.close();  
        }  
    }  
}
```

Since Java 7 there is a really useful and neat statement introduced in Java 7 particularly for this case, called try-with-resources:

Version ≥ Java SE 7

```

private static void printFileJava7() throws IOException {
    try (FileInputStream input = new FileInputStream("file.txt")) {
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}

```

try-with-resources语句可以用于任何实现了Closeable或AutoCloseable接口的对象。它确保每个资源在语句结束时被关闭。这两个接口的区别在于，Closeable的close()方法会抛出一个IOException，必须以某种方式处理该异常。

在资源已经被打开但应在使用后安全关闭的情况下，可以将其赋值给try-with-resources中的局部变量

```

版本 ≥ Java SE 7
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //访问资源
    }
}

```

在try-with-resources构造函数中创建的局部资源变量实际上是final的。

第130.4节：陷阱：在尝试打开文件之前测试文件

有些人建议在尝试打开文件之前应对文件进行各种测试，以提供更好的诊断或避免处理异常。例如，该方法尝试检查path对应一个可读文件：

```

public static File getValidatedFile(String path) throws IOException {
    File f = new File(path);
    if (!f.exists()) throw new IOException("错误：未找到文件：" + path);
    if (!f.isFile()) throw new IOException("错误：是一个目录：" + path);
    if (!f.canRead()) throw new IOException("错误：无法读取文件：" + path);
    return f;
}

```

你可以像这样使用上述方法：

```

File f = null;
try {
    f = getValidatedFile("somefile");
} catch (IOException ex) {
    System.err.println(ex.getMessage());
    return;
}
try (InputStream is = new FileInputStream(file)) {
    // 读取数据等。
}

```

第一个问题出现在 `FileInputStream(File)` 的方法签名中，因为编译器仍然会坚持要求我们捕获 `IOException`，或者在调用栈的更高层捕获。

```

private static void printFileJava7() throws IOException {
    try (FileInputStream input = new FileInputStream("file.txt")) {
        int data = input.read();
        while (data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}

```

The *try-with-resources* statement can be used with any object that implements the Closeable or AutoCloseable interface. It ensures that each resource is closed by the end of the statement. The difference between the two interfaces is, that the `close()` method of Closeable throws an `IOException` which has to be handled in some way.

In cases where the resource has already been opened but should be safely closed after use, one can assign it to a local variable inside the *try-with-resources*

```

Version ≥ Java SE 7
private static void printFileJava7(InputStream extResource) throws IOException {
    try (InputStream input = extResource) {
        ... //access resource
    }
}

```

The local resource variable created in the *try-with-resources* constructor is effectively final.

Section 130.4: Pitfall: testing a file before attempting to open it

Some people recommend that you should apply various tests to a file before attempting to open it either to provide better diagnostics or avoid dealing with exceptions. For example, this method attempts to check if path corresponds to a readable file:

```

public static File getValidatedFile(String path) throws IOException {
    File f = new File(path);
    if (!f.exists()) throw new IOException("Error: not found: " + path);
    if (!f.isFile()) throw new IOException("Error: Is a directory: " + path);
    if (!f.canRead()) throw new IOException("Error: cannot read file: " + path);
    return f;
}

```

You might use the above method like this:

```

File f = null;
try {
    f = getValidatedFile("somefile");
} catch (IOException ex) {
    System.err.println(ex.getMessage());
    return;
}
try (InputStream is = new FileInputStream(file)) {
    // Read data etc.
}

```

The first problem is in the signature for `FileInputStream(File)` because the compiler will still insist we catch `IOException` here, or further up the stack.

第二个问题是 `getValidatedFile` 进行的检查并不能保证 `FileInputStream` 会成功。

- 竞态条件：另一个线程或独立进程可能会在`getValidatedFile`返回后重命名文件、删除文件或移除读取权限。这将导致一个“普通”的`IOException`，而没有自定义消息。
- 有些边缘情况未被这些测试覆盖。例如，在启用SELinux“强制”模式的系统上，尽管`canRead()`返回true，尝试读取文件仍可能失败。

第三个问题是测试效率低下。例如，`exists`、`isFile`和`canRead`调用都会各自执行一次`syscall`来完成所需检查。随后又会进行另一次`syscall`来打开文件，这在后台重复了相同的检查。

简而言之，像`getValidatedFile`这样的方法是误导性的。更好的做法是直接尝试打开文件并处理异常：

```
try (InputStream is = new FileInputStream("somefile")) {  
    // 读取数据等。  
} catch (IOException ex) {  
    System.err.println("处理 'somefile' 时的IO错误：" + ex.getMessage());  
    return;  
}
```

如果你想区分打开和读取时抛出的IO错误，可以使用嵌套的try / catch。如果你想为打开失败提供更好的诊断，可以在异常处理器中执行`exists`、`isFile`和`canRead`检查。

第130.5节：陷阱：将变量视为对象

没有任何Java变量代表一个对象。

```
字符串 foo; // 不是对象
```

Java数组中也不包含对象。

```
String bar[] = new String[100]; // 没有成员是对象。
```

如果你错误地将变量视为对象，Java语言的实际行为会让你感到惊讶。

- 对于具有原始类型（如int或float）的Java变量，变量保存的是值的副本。所有原始值的副本都是无法区分的；即数字1只有一个int值。原始值不是对象，也不表现得像对象。
- 对于具有引用类型（类或数组类型）的Java变量，变量保存的是引用。所有引用的副本都是无法区分的。引用可能指向对象，或者可能是null，表示它们不指向任何对象。然而，它们不是对象，也不表现得像对象。

无论哪种情况，变量都不是对象，也不包含对象。它们可能包含指向对象的引用，但那是另一回事。

示例类

以下示例使用此类，该类表示二维空间中的一个点。

The second problem is that checks performed by `getValidatedFile` do not guarantee that the `FileInputStream` will succeed.

- Race conditions: another thread or a separate process could rename the file, delete the file, or remove read access after the `getValidatedFile` returns. That would lead to a "plain" `IOException` without the custom message.
- There are edge cases not covered by those tests. For example, on a system with SELinux in "enforcing" mode, an attempt to read a file can fail despite `canRead()` returning `true`.

The third problem is that the tests are inefficient. For example, the `exists`, `isFile` and `canRead` calls will each make a `syscall` to perform the required check. Another `syscall` is then made to open the file, which repeats the same checks behind the scenes.

In short, methods like `getValidatedFile` are misguided. It is better to simply attempt to open the file and handle the exception:

```
try (InputStream is = new FileInputStream("somefile")) {  
    // Read data etc.  
} catch (IOException ex) {  
    System.err.println("IO Error processing 'somefile': " + ex.getMessage());  
    return;  
}
```

If you wanted to distinguish IO errors thrown while opening and reading, you could use a nested try / catch. If you wanted to produce better diagnostics for open failures, you could perform the `exists`, `isFile` and `canRead` checks in the handler.

Section 130.5: Pitfall: thinking of variables as objects

No Java variable represents an object.

```
String foo; // NOT AN OBJECT
```

Neither does any Java array contain objects.

```
String bar[] = new String[100]; // No member is an object.
```

If you mistakenly think of variables as objects, the actual behavior of the Java language will surprise you.

- For Java variables which have a primitive type (such as `int` or `float`) the variable holds a copy of the value. All copies of a primitive value are indistinguishable; i.e. there is only one `int` value for the number one. Primitive values are not objects and they do not behave like objects.
- For Java variables which have a reference type (either a class or an array type) the variable holds a reference. All copies of a reference are indistinguishable. References may point to objects, or they may be `null` which means that they point to no object. However, they are not objects and they don't behave like objects.

Variables are not objects in either case, and they don't contain objects in either case. They may contain *references to objects*, but that is saying something different.

Example class

The examples that follow use this class, which represents a point in 2D space.

```

public final class MutableLocation {
    public int x;
    public int y;

    public MutableLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object other) {
        if (!(other instanceof MutableLocation)) {
            return false;
        }
        MutableLocation that = (MutableLocation) other;
        return this.x == that.x && this.y == that.y;
    }
}

```

该类的一个实例是一个具有两个字段 `x` 和 `y` 的对象，这两个字段的类型都是 `int`。

我们可以拥有多个 `MutableLocation` 类的实例。其中一些将表示二维空间中的相同位置；即 `x` 和 `y` 的相应值将匹配。其他的将表示不同的位置。

多个变量可以指向同一个对象

```

MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

```

在上面，我们声明了三个变量 `here`、`there` 和 `elsewhere`，它们可以保存对 `MutableLocation` 对象的引用。

如果你（错误地）将这些变量视为对象，那么你很可能会误解这些语句的意思：

1. 将位置 "[1, 2]" 复制到 `here`
2. 将位置 "[1, 2]" 复制到 `there`
3. 将位置 "[1, 2]" 复制到 `elsewhere`

由此，你可能会推断我们在三个变量中有三个独立的对象。实际上，上述代码只创建了 两个 对象。`变量 here 和 there 实际上引用的是同一个对象。`

我们可以证明这一点。假设变量声明如上：

```

System.out.println("BEFORE: here.x is " + here.x + ", there.x is " + there.x +
                    "elsewhere.x is " + elsewhere.x);
here.x = 42;
System.out.println("AFTER: here.x is " + here.x + ", there.x is " + there.x +
                    "elsewhere.x is " + elsewhere.x);

```

这将输出以下内容：

```

BEFORE: here.x is 1, there.x is 1, elsewhere.x is 1
AFTER: here.x is 42, there.x is 42, elsewhere.x is 1

```

我们给 `here.x` 赋了一个新值，它改变了我们通过 `there.x` 看到的值。它们指的是同一个对象。但我们通过 `elsewhere.x` 看到的值没有改变，所以 `elsewhere` 必须指向一个不同的

```

public final class MutableLocation {
    public int x;
    public int y;

    public MutableLocation(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public boolean equals(Object other) {
        if (!(other instanceof MutableLocation)) {
            return false;
        }
        MutableLocation that = (MutableLocation) other;
        return this.x == that.x && this.y == that.y;
    }
}

```

An instance of this class is an object that has two fields `x` and `y` which have the type `int`.

We can have many instances of the `MutableLocation` class. Some will represent the same locations in 2D space; i.e. the respective values of `x` and `y` will match. Others will represent different locations.

Multiple variables can point to the same object

```

MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

```

In the above, we have declared three variables `here`, `there` and `elsewhere` that can hold references to `MutableLocation` objects.

If you (incorrectly) think of these variables as being objects, then you are likely to misread the statements as saying:

1. Copy the location "[1, 2]" to `here`
2. Copy the location "[1, 2]" to `there`
3. Copy the location "[1, 2]" to `elsewhere`

From that, you are likely to infer we have three independent objects in the three variables. In fact there are *only two objects created* by the above. The variables `here` and `there` actually refer to the same object.

We can demonstrate this. Assuming the variable declarations as above:

```

System.out.println("BEFORE: here.x is " + here.x + ", there.x is " + there.x +
                    "elsewhere.x is " + elsewhere.x);
here.x = 42;
System.out.println("AFTER: here.x is " + here.x + ", there.x is " + there.x +
                    "elsewhere.x is " + elsewhere.x);

```

This will output the following:

```

BEFORE: here.x is 1, there.x is 1, elsewhere.x is 1
AFTER: here.x is 42, there.x is 42, elsewhere.x is 1

```

We assigned a new value to `here.x` and it changed the value that we see via `there.x`. They are referring to the same object. But the value that we see via `elsewhere.x` has not changed, so `elsewhere` must refer to a different

对象。

如果一个变量是对象，那么赋值 `here.x = 42` 不会改变 `there.x` 的值。

等号运算符不会测试两个对象是否相等

对引用值使用等号 (`==`) 运算符测试的是这些值是否引用同一个对象。它不测试两个 (不同) 对象在直观意义上是否“相等”。

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

if (here == there) {
    System.out.println("here is there");
}
if (here == elsewhere) {
    System.out.println("here is elsewhere");
}
```

这将打印 "here is there"，但不会打印 "here is elsewhere"。 (`here` 和 `elsewhere` 中的引用是指两个不同的对象。)

相比之下，如果我们调用上面实现的 `equals(Object)` 方法，我们将测试两个 `MutableLocation` 实例的位置是否相等。

```
if (here.equals(there)) {
    System.out.println("here equals there");
}
if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
}
```

这将打印两个消息。特别是，`here.equals(elsewhere)` 返回 `true`，因为我们选择的两个 `MutableLocation` 对象相等的语义标准已被满足。

方法调用根本不传递对象

Java 方法调用使用 按值传递1 来传递参数并返回结果。

当你将一个引用值传递给方法时，实际上是按值传递了一个对象的引用，这意味着它创建了对象引用的一个副本。

只要两个对象引用仍然指向同一个对象，你就可以通过任一引用修改该对象，这也是一些人感到困惑的原因。

但是，你并不是通过引用传递对象。区别在于，如果对象引用的副本被修改为指向另一个对象，原始对象引用仍然会指向原始对象。

```
void f(MutableLocation foo) {
    foo = new MutableLocation(3, 4); // 让局部变量 foo 指向不同的对象。
}

void g() {
    MutableLocation foo = MutableLocation(1, 2);
    f(foo);
    System.out.println("foo.x is " + foo.x); // 输出 "foo.x is 1".
}
```

object.

If a variable was an object, then the assignment `here.x = 42` would not change `there.x`.

The equality operator does NOT test that two objects are equal

Applying the equality (`==`) operator to reference values tests if the values refer to the same object. It does *not* test whether two (different) objects are "equal" in the intuitive sense.

```
MutableLocation here = new MutableLocation(1, 2);
MutableLocation there = here;
MutableLocation elsewhere = new MutableLocation(1, 2);

if (here == there) {
    System.out.println("here is there");
}
if (here == elsewhere) {
    System.out.println("here is elsewhere");
}
```

This will print "here is there", but it won't print "here is elsewhere". (The references in `here` and `elsewhere` are for two distinct objects.)

By contrast, if we call the `equals(Object)` method that we implemented above, we are going to test if two `MutableLocation` instances have an equal location.

```
if (here.equals(there)) {
    System.out.println("here equals there");
}
if (here.equals(elsewhere)) {
    System.out.println("here equals elsewhere");
}
```

This will print both messages. In particular, `here.equals(elsewhere)` returns `true` because the semantic criteria we chose for equality of two `MutableLocation` objects has been satisfied.

Method calls do NOT pass objects at all

Java method calls use *pass by value*1 to pass arguments and return a result.

When you pass a reference value to a method, you're actually passing a reference to an object *by value*, which means that it is creating a copy of the object reference.

As long as both object references are still pointing to the same object, you can modify that object from either reference, and this is what causes confusion for some.

However, you are *not* passing an object by reference2. The distinction is that if the object reference copy is modified to point to another object, the original object reference will still point to the original object.

```
void f(MutableLocation foo) {
    foo = new MutableLocation(3, 4); // Point local foo at a different object.
}

void g() {
    MutableLocation foo = MutableLocation(1, 2);
    f(foo);
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 1".
}
```

}

你也不是传递了对象的副本。

```
void f(MutableLocation foo) {  
    foo.x = 42;  
}  
  
void g() {  
    MutableLocation foo = new MutableLocation(0, 0);  
    f(foo);  
    System.out.println("foo.x is " + foo.x); // 打印 "foo.x is 42"  
}
```

1 - 在像 Python 和 Ruby 这样的语言中，术语“通过共享传递”更常用于表示对象/引用的“按值传递”。

2 - “按引用传递”或“按引用调用”在编程语言术语中有非常具体的含义。实际上，它意味着传递变量或数组元素的地址，因此当被调用的方法给形式参数赋新值时，会改变原始变量中的值。Java 不支持这一点。有关不同参数传递机制的更详细描述，请参阅 https://en.wikipedia.org/wiki/Evaluation_strategy。

}

Neither are you passing a copy of the object.

```
void f(MutableLocation foo) {  
    foo.x = 42;  
}  
  
void g() {  
    MutableLocation foo = new MutableLocation(0, 0);  
    f(foo);  
    System.out.println("foo.x is " + foo.x); // Prints "foo.x is 42"  
}
```

1 - In languages like Python and Ruby, the term "pass by sharing" is preferred for "pass by value" of an object / reference.

2 - The term "pass by reference" or "call by reference" has a very specific meaning in programming language terminology. In effect, it means that you pass the address of a variable or an array element, so that when the called method assigns a new value to the formal argument, it changes the value in the original variable. Java does not support this. For a more fulsome description of different mechanisms for passing parameters, please refer to https://en.wikipedia.org/wiki/Evaluation_strategy.

第130.6节：陷阱：内存泄漏

Java 自动管理内存。你不需要手动释放内存。当对象不再被活动线程“可达”时，垃圾回收器可能会释放堆上的对象内存。

然而，你可以通过让不再需要的对象保持可达状态来阻止内存被释放。无论你称之为内存泄漏还是内存囤积，结果都是一样的——分配的内存不必要地增加。

Java 中的内存泄漏可能以多种方式发生，但最常见的原因是永远存在的对象引用，因为只要还有引用，垃圾回收器就无法从堆中移除对象。

静态字段

可以通过定义包含某些对象集合的静态字段的类来创建这样的引用，并且在集合不再需要后忘记将该静态字段设置为 null。静态字段被视为垃圾回收根（GC roots），永远不会被回收。另一个问题是在使用 JNI 时非堆内存的泄漏。

类加载器泄漏

迄今为止，最隐蔽的内存泄漏类型是类加载器泄漏。类加载器持有它加载的每个类的引用，每个类又持有其类加载器的引用。每个对象也持有其类的引用。因此，如果类加载器加载的某个类的单个对象未被垃圾回收，那么该类加载器加载的任何类都无法被回收。由于每个类还引用其静态字段，这些静态字段也无法被回收。

累积泄漏 累积泄漏示例可能如下所示：

```
final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);  
final Deque<BigDecimal> numbers = new LinkedBlockingDeque<>();  
final BigDecimal divisor = new BigDecimal(51);  
  
scheduledExecutorService.scheduleAtFixedRate(() -> {
```

Section 130.6: Pitfall: memory leaks

Java manages memory automatically. You are not required to free memory manually. An object's memory on the heap may be freed by a garbage collector when the object is no longer *reachable* by a live thread.

However, you can prevent memory from being freed, by allowing objects to be reachable that are no longer needed. Whether you call this a memory leak or memory packratting, the result is the same -- an unnecessary increase in allocated memory.

Memory leaks in Java can happen in various ways, but the most common reason is everlasting object references, because the garbage collector can't remove objects from the heap while there are still references to them.

Static fields

One can create such a reference by defining class with a **static** field containing some collection of objects, and forgetting to set that **static** field to **null** after the collection is no longer needed. **static** fields are considered GC roots and are never collected. Another issue is leaks in non-heap memory when [JNI](#) is used.

Classloader leak

By far, though, the most insidious type of memory leak is the [classloader leak](#). A classloader holds a reference to every class it has loaded, and every class holds a reference to its classloader. Every object holds a reference to its class as well. Therefore, if even a *single* object of a class loaded by a classloader is not garbage, not a single class that that classloader has loaded can be collected. Since each class also refers to its static fields, they cannot be collected either.

Accumulation leak The accumulation leak example could look like the following:

```
final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);  
final Deque<BigDecimal> numbers = new LinkedBlockingDeque<>();  
final BigDecimal divisor = new BigDecimal(51);  
  
scheduledExecutorService.scheduleAtFixedRate(() -> {
```

```

BigDecimal number = numbers.peekLast();
if (number != null && number.remainder(divisor).byteValue() == 0) {
    System.out.println("Number: " + number);
    System.out.println("Deque size: " + numbers.size());
}
}, 10, 10, TimeUnit.MILLISECONDS);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    numbers.add(new BigDecimal(System.currentTimeMillis()));
}, 10, 10, TimeUnit.MILLISECONDS);

try {
    scheduledExecutorService.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}

```

该示例创建了两个定时任务。第一个任务从名为 `numbers` 的双端队列中取出最后一个数字，如果该数字能被51整除，则打印该数字和队列的大小。第二个任务向队列中添加数字。两个任务都以固定频率调度，每10毫秒运行一次。

如果执行该代码，你会看到双端队列的大小持续增加。这最终会导致双端队列被占满，消耗所有可用的堆内存。

为了在保持该程序语义的同时防止此问题，我们可以使用另一种从双端队列中取数字的方法：`pollLast`。与方法`peekLast`相反，`pollLast`返回元素并将其从双端队列中移除，而`peekLast`仅返回最后一个元素。

第130.7节：陷阱：不了解String是不可变类

新的Java程序员经常忘记或未能完全理解Java的String类是不可变的。这会导致如下示例中的问题：

```

public class Shout {
    public static void main(String[] args) {
        for (String s : args) {
            s.toUpperCase();
            System.out.print(s);
            System.out.print(" ");
        }
        System.out.println();
    }
}

```

上述代码本应打印命令行参数的大写形式。不幸的是，它不起作用，参数的大小写没有改变。问题出在这条语句：

```
s.toUpperCase();
```

你可能认为调用`toUpperCase()`会将 `s` 转换为大写字符串。事实并非如此。它不能！String对象是不可变的，不能被改变。

实际上，`toUpperCase()`方法返回一个String对象，该对象是你调用它的String的大写版本。这个对象很可能是一个新的String对象，但如果 `s` 已经全部是大写，结果可能是现有的字符串。

```

BigDecimal number = numbers.peekLast();
if (number != null && number.remainder(divisor).byteValue() == 0) {
    System.out.println("Number: " + number);
    System.out.println("Deque size: " + numbers.size());
}
}, 10, 10, TimeUnit.MILLISECONDS);

scheduledExecutorService.scheduleAtFixedRate(() -> {
    numbers.add(new BigDecimal(System.currentTimeMillis()));
}, 10, 10, TimeUnit.MILLISECONDS);

try {
    scheduledExecutorService.awaitTermination(1, TimeUnit.DAYS);
} catch (InterruptedException e) {
    e.printStackTrace();
}

```

This example creates two scheduled tasks. The first task takes the last number from a deque called `numbers`, and, if the number is divisible by 51, it prints the number and the deque's size. The second task puts numbers into the deque. Both tasks are scheduled at a fixed rate, and they run every 10 ms.

If the code is executed, you'll see that the size of the deque is permanently increasing. This will eventually cause the deque to be filled with objects that consume all available heap memory.

To prevent this while preserving the semantics of this program, we can use a different method for taking numbers from the deque: `pollLast`. Contrary to the method `peekLast`, `pollLast` returns the element and removes it from the deque while `peekLast` only returns the last element.

Section 130.7: Pitfall: Not understanding that String is an immutable class

New Java programmers often forget, or fail to fully comprehend, that the Java `String` class is immutable. This leads to problems like the one in the following example:

```

public class Shout {
    public static void main(String[] args) {
        for (String s : args) {
            s.toUpperCase();
            System.out.print(s);
            System.out.print(" ");
        }
        System.out.println();
    }
}

```

The above code is supposed to print command line arguments in upper case. Unfortunately, it does not work, the case of the arguments is not changed. The problem is this statement:

```
s.toUpperCase();
```

You might think that calling `toUpperCase()` is going to change `s` to an uppercase string. It doesn't. It can't! `String` objects are immutable. They cannot be changed.

In reality, the `toUpperCase()` method *returns* a `String` object which is an uppercase version of the `String` that you call it on. This will probably be a new `String` object, but if `s` was already all uppercase, the result could be the existing string.

因此，为了有效使用此方法，您需要使用方法调用返回的对象；例如：

```
s = s.toUpperCase();
```

事实上，“字符串永远不变”的规则适用于所有String方法。如果您记住这一点，就可以避免一整类初学者的错误。

第130.8节：陷阱：结合赋值和副作用

我们偶尔会看到StackOverflow上的Java问题（以及C或C++问题）询问类似以下代码的结果：

```
i += a[i++] + b[i--];
```

对于已知的初始状态的i、a和b，这段代码的计算结果是.....

一般来说：

- 对于Java，答案总是有规定的1，但不明显，且通常难以弄清楚
- 对于C和C++，答案通常是不确定的。

此类示例常用于考试或面试中，目的是测试学生或面试者是否真正理解Java编程语言中表达式求值的工作原理。

这可以说是作为“知识测试”是合理的，但这并不意味着你在实际程序中应该这样做。

举例来说，以下这个看似简单的例子曾多次出现在StackOverflow的问题中（比如[this one](#)）。在某些情况下，它表现为某人代码中的一个真实错误。

```
int a = 1;
a = a++;
System.out.println(a); // 这将打印什么。
```

大多数程序员（包括Java专家）快速阅读这些语句时会说它输出2。实际上，它输出1。关于原因的详细解释，请阅读[this Answer](#)。

然而，从这个及类似例子中真正应吸取的教训是，任何Java语句既对同一变量进行赋值又产生副作用，充其量都难以理解，最坏则极具误导性。你应该避免编写这样的代码。

1 - 除非变量或对象对其他线程可见，否则可能存在Java内存模型相关的问题。

So in order to use this method effectively, you need to use the object returned by the method call; for example:

```
s = s.toUpperCase();
```

In fact, the "strings never change" rule applies to all `String` methods. If you remember that, then you can avoid a whole category of beginner's mistakes.

Section 130.8: Pitfall: combining assignment and side-effects

Occasionally we see StackOverflow Java questions (and C or C++ questions) that ask what something like this:

```
i += a[i++] + b[i--];
```

evaluates to ... for some known initial states of i, a and b.

Generally speaking:

- for Java the answer is always specified¹, but non-obvious, and often difficult to figure out
- for C and C++ the answer is often unspecified.

Such examples are often used in exams or job interviews as an attempt to see if the student or interviewee understands how expression evaluation really works in the Java programming language. This is arguably legitimate as a "test of knowledge", but that does not mean that you should ever do this in a real program.

To illustrate, the following seemingly simple example has appeared a few times in StackOverflow questions (like [this one](#)). In some cases, it appears as a genuine mistake in someone's code.

```
int a = 1;
a = a++;
System.out.println(a); // What does this print.
```

Most programmers (including Java experts) reading those statements quickly would say that it outputs 2. In fact, it outputs 1. For a detailed explanation of why, please read [this Answer](#).

However the real takeaway from this and similar examples is that *any* Java statement that *both* assigns to *and* side-effects the same variable is going to be *at best* hard to understand, and *at worst* downright misleading. You should avoid writing code like this.

1 - modulo potential issues with the Java Memory Model if the variables or objects are visible to other threads.

第131章：Java陷阱 - 异常使用

尽管编译正确，若Java编程语言使用不当，程序仍可能产生错误结果。本主题的主要目的是列出与异常处理相关的常见陷阱，并提出避免这些陷阱的正确方法。

第131.1节：陷阱——捕获Throwable、Exception、Error或Runtimeexception

对于缺乏经验的Java程序员来说，一个常见的思维模式是认为异常是“问题”或“负担”，而处理它们的最佳方式是尽早捕获所有异常¹。这导致了如下代码：

```
....  
try {  
    InputStream is = new FileInputStream(fileName);  
    // 处理输入  
} catch (Exception ex) {  
    System.out.println("无法打开文件 " + fileName);  
}
```

上述代码存在一个重大缺陷。该 `catch` 实际上会捕获比程序员预期更多的异常。假设 `fileName` 的值为 `null`，因应用程序其他地方的错误导致。这将使 `FileInputStream` 构造函数抛出 `NullPointerException`。该处理器会捕获此异常，并向用户报告：

```
无法打开文件 null
```

这既无帮助又令人困惑。更糟的是，假设是“处理输入”代码抛出了意外异常（无论是检查异常还是非检查异常！）。现在用户将收到一个误导性的消息，提示文件打开时出现问题，而实际上问题可能根本与I/O无关。

问题的根源在于程序员为 `Exception` 编写了一个处理器。这几乎总是一个错误：

- 捕获`Exception`将捕获所有已检查异常，以及大多数未检查异常。
- 捕获`RuntimeException`将捕获大多数未检查异常。
- 捕获`Error`将捕获表示JVM内部错误的未检查异常。这些错误通常不可恢复，且不应被捕获。
- 捕获`Throwable`将捕获所有可能的异常。

捕获过于广泛的异常集合的问题在于，处理程序通常无法适当处理所有异常。以`Exception`等为例，程序员很难预测可能被捕获的异常；即无法预期。

一般来说，正确的解决方案是处理被抛出的异常。例如，可以捕获它们并在原地处理：

```
try {  
    InputStream is = new FileInputStream(fileName);  
    // 处理输入  
} catch (FileNotFoundException ex) {  
    System.out.println("无法打开文件 " + fileName);  
}
```

Chapter 131: Java Pitfalls - Exception usage

Several Java programming language misusage might conduct a program to generate incorrect results despite being compiled correctly. This topic main purpose is to list common **pitfalls** related to **exception handling**, and to propose the correct way to avoid having such pitfalls.

Section 131.1: Pitfall - Catching Throwable, Exception, Error or RuntimeException

A common thought pattern for inexperienced Java programmers is that exceptions are "a problem" or "a burden" and the best way to deal with this is catch them all¹ as soon as possible. This leads to code like this:

```
....  
try {  
    InputStream is = new FileInputStream(fileName);  
    // process the input  
} catch (Exception ex) {  
    System.out.println("Could not open file " + fileName);  
}
```

The above code has a significant flaw. The `catch` is actually going to catch more exceptions than the programmer is expecting. Suppose that the value of the `fileName` is `null`, due to a bug elsewhere in the application. This will cause the `FileInputStream` constructor to throw a `NullPointerException`. The handler will catch this, and report to the user:

```
Could not open file null
```

which is unhelpful and confusing. Worse still, suppose that it was the "process the input" code that threw the unexpected exception (checked or unchecked!). Now the user will get the misleading message for a problem that didn't occur while opening the file, and may not be related to I/O at all.

The root of the problem is that the programmer has coded a handler for `Exception`. This is almost always a mistake:

- Catching `Exception` will catch all checked exceptions, and most unchecked exceptions as well.
- Catching `RuntimeException` will catch most unchecked exceptions.
- Catching `Error` will catch unchecked exceptions that signal JVM internal errors. These errors are generally not recoverable, and should not be caught.
- Catching `Throwable` will catch all possible exceptions.

The problem with catching too broad a set of exceptions is that the handler typically cannot handle all of them appropriately. In the case of the `Exception` and so on, it is difficult for the programmer to predict what could be caught; i.e. what to expect.

In general, the correct solution is to deal with the exceptions that are thrown. For example, you can catch them and handle them in situ:

```
try {  
    InputStream is = new FileInputStream(fileName);  
    // process the input  
} catch (FileNotFoundException ex) {  
    System.out.println("Could not open file " + fileName);  
}
```

或者可以将它们声明为由包含方法抛出。

很少有情况适合捕获Exception。唯一常见的情况类似于：

```
public static void main(String[] args) {
    try {
        // 执行操作
    } catch (Exception ex) {
        System.err.println("很遗憾，发生了错误。 " +
                           "请将此报告给 X Y Z");
        // 将堆栈跟踪写入日志文件。
        System.exit(1);
    }
}
```

这里我们确实想处理所有异常，所以捕获Exception（甚至Throwable）是正确的。

1 - 也称为宝可梦异常处理。

第131.2节：陷阱 - 忽略或压制异常

这个例子是关于故意忽略或“压制”异常。更准确地说，是关于如何捕获并处理异常而忽略它。然而，在描述如何做到这一点之前，我们首先应该指出，压制异常通常不是处理它们的正确方式。

异常通常是由某些东西抛出，以通知程序的其他部分发生了一些重要的（即“异常”）事件。通常（虽然不总是）异常意味着出现了问题。如果你编写程序来压制异常，很可能问题会以另一种形式重新出现。更糟的是，当你压制异常时，你丢弃了异常对象及其相关堆栈跟踪中的信息。这很可能会使得查明问题的最初来源更加困难。

实际上，当你使用IDE的自动更正功能来“修复”由未处理异常引起的编译错误时，经常会发生异常压制。例如，你可能会看到如下代码：

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* 在此处添加异常处理代码 */
}
```

显然，程序员接受了IDE的建议以消除编译错误，但该建议是不合适的。（如果文件打开失败，程序很可能应该对此做出处理。）

使用上述“修正”后，程序可能会在后续失败；例如，由于`NullPointerException`，因为`inputStream`现在是`null`。）

话虽如此，下面是一个故意忽略异常的示例。（为了论证，假设我们已经确定在显示自拍时被中断是无害的。）注释告诉读者我们是故意忽略异常的，以及这样做的原因。

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // 即使显示自拍被中断也无所谓。
}
```

or you can declare them as thrown by the enclosing method.

There are very few situations where catching `Exception` is appropriate. The only one that arises commonly is something like this:

```
public static void main(String[] args) {
    try {
        // do stuff
    } catch (Exception ex) {
        System.err.println("Unfortunately an error has occurred. " +
                           "Please report this to X Y Z");
        // Write stacktrace to a log file.
        System.exit(1);
    }
}
```

Here we genuinely want to deal with all exceptions, so catching `Exception` (or even `Throwable`) is correct.

1 - Also known as [Pokemon Exception Handling](#).

Section 131.2: Pitfall - Ignoring or squashing exceptions

This example is about deliberately ignoring or “squashing” exceptions. Or to be more precise, it is about how to catch and handle an exception in a way that ignores it. However, before we describe how to do this, we should first point out that squashing exceptions is generally not the correct way to deal with them.

Exceptions are usually thrown (by something) to notify other parts of the program that some significant (i.e. “exceptional”) event has occurred. Generally (though not always) an exception means that something has gone wrong. If you code your program to squash the exception, there is a fair chance that the problem will reappear in another form. To make things worse, when you squash the exception, you are throwing away the information in the exception object and its associated stack trace. That is likely to make it harder to figure out what the original source of the problem was.

In practice, exception squashing frequently happens when you use an IDE's auto-correction feature to “fix” a compilation error caused by an unhandled exception. For example, you might see code like this:

```
try {
    inputStream = new FileInputStream("someFile");
} catch (IOException e) {
    /* add exception handling code here */
}
```

Clearly, the programmer has accepted the IDE's suggestion to make the compilation error go away, but the suggestion was inappropriate. (If the file open has failed, the program should most likely do something about it. With the above “correction”, the program is liable to fail later; e.g. with a `NullPointerException` because `inputStream` is now `null`.)

Having said that, here is an example of deliberately squashing an exception. (For the purposes of argument, assume that we have determined that an interrupt while showing the selfie is harmless.) The comment tells the reader that we squashed the exception deliberately, and why we did that.

```
try {
    selfie.show();
} catch (InterruptedException e) {
    // It doesn't matter if showing the selfie is interrupted.
}
```

}

另一种常见的方式是通过异常变量名来突出我们是故意忽略异常，但不说明原因，如下所示：

```
try {  
    selfie.show();  
} catch (InterruptedException ignored) { }
```

一些IDE（如IntelliJ IDEA）如果变量名设置为ignored。

第131.3节：陷阱 - 抛出Throwable、Exception、Error或RuntimeException

虽然捕获Throwable、Exception、Error和RuntimeException异常是不好的，但抛出它们更糟糕。

基本问题是，当您的应用程序需要处理异常时，顶层异常的存在使得区分不同的错误情况变得困难。例如

```
try {  
    InputStream = new FileInputStream(someFile); // 可能抛出IOException  
    ...  
    if (somethingBad) {  
        throw new Exception(); // 错误  
    }  
} catch (IOException ex) {  
    System.err.println("无法打开 ...");  
} catch (Exception ex) {  
    System.err.println("发生了错误"); // 错误  
}
```

问题在于，因为我们抛出了一个Exception实例，我们被迫捕获它。然而如另一个例子所述，捕获Exception是不好的。在这种情况下，区分“预期”的Exception（当somethingBad为true时抛出）和意外捕获的未检查异常（如NullPointerException）变得困难。

如果允许顶层异常传播，我们会遇到其他问题：

- 我们现在必须记住所有我们抛弃顶层的不同原因，并区分/处理它们。
- 在Exception和Throwable的情况下，如果我们希望异常能够传播，还需要将这些异常添加到方法的throws子句中。这是有问题的，如下所述。

简而言之，不要抛出这些异常。抛出一个更具体的异常，更准确地描述发生的“异常事件”。如果需要，可以定义并使用自定义异常类。

在方法的“throws”中声明Throwable或Exception是有问题的。

用Exception甚至Throwable替换方法的throws子句中长长的异常列表是很诱人的。但这是一个坏主意：

1. 它强制调用者处理（或传播）Exception。
2. 我们不能再依赖编译器告诉我们需要处理的具体受检异常。

}

Another conventional way to highlight that we are *deliberately* squashing an exception without saying why is to indicate this with the exception variable's name, like this:

```
try {  
    selfie.show();  
} catch (InterruptedException ignored) { }
```

Some IDEs (like IntelliJ IDEA) won't display a warning about the empty catch block if the variable name is set to ignored.

Section 131.3: Pitfall - Throwing Throwable, Exception, Error or RuntimeException

While catching the `Throwable`, `Exception`, `Error` and `RuntimeException` exceptions is bad, throwing them is even worse.

The basic problem is that when your application needs to handle exceptions, the presence of the top level exceptions make it hard to discriminate between different error conditions. For example

```
try {  
    InputStream is = new FileInputStream(someFile); // could throw IOException  
    ...  
    if (somethingBad) {  
        throw new Exception(); // WRONG  
    }  
} catch (IOException ex) {  
    System.err.println("cannot open ...");  
} catch (Exception ex) {  
    System.err.println("something bad happened"); // WRONG  
}
```

The problem is that because we threw an `Exception` instance, we are forced to catch it. However as described in another example, catching `Exception` is bad. In this situation, it becomes difficult to discriminate between the "expected" case of an `Exception` that gets thrown if `somethingBad` is `true`, and the unexpected case where we actually catch an unchecked exception such as `NullPointerException`.

If the top-level exception is allowed to propagate, we run into other problems:

- We now have to remember all of the different reasons that we threw the top-level, and discriminate / handle them.
- In the case of `Exception` and `Throwable` we also need to add these exceptions to the `throws` clause of methods if we want the exception to propagate. This is problematic, as described below.

In short, don't throw these exceptions. Throw a more specific exception that more closely describes the "exceptional event" that has happened. If you need to, define and use a custom exception class.

Declaring `Throwable` or `Exception` in a method's "throws" is problematic.

It is tempting to replace a long list of thrown exceptions in a method's `throws` clause with `Exception` or even `Throwable`. This is a bad idea:

1. It forces the caller to handle (or propagate) `Exception`.
2. We can no longer rely on the compiler to tell us about specific checked exceptions that need to be handled.

3. 正确处理Exception很困难。很难知道实际可能捕获到哪些异常，如果不知道可能捕获到什么异常，就很难知道适当的恢复策略是什么。
4. 处理Throwable更难，因为现在你还必须应对那些本不应该发生的潜在失败。可以从中恢复。

这条建议意味着应避免某些其他模式。例如：

```
try {
    doSomething();
} catch (Exception ex) {
    report(ex);
    throw ex;
}
```

上述代码试图在异常传递时记录所有异常，但并未明确处理它们。不幸的是，在Java 7之前，`throw ex;`语句会让编译器认为任何Exception都可能被抛出。这可能会迫使你将包含该代码的方法声明为`throws Exception`。从Java 7开始，编译器知道这里可能（重新抛出）的异常集合更小。

第131.4节：陷阱——将异常用于正常流程控制

有一句口头禅是一些 Java 专家常常念叨的：

“异常应该只用于异常情况。”

（例如：<http://programmers.stackexchange.com/questions/184654>）其

核心是，在 Java 中使用异常和异常处理来实现正常流程控制是个坏主意。例如，比较以下两种处理可能为 `null` 的参数的方法。

```
public String truncateWordOrNull(String word, int maxLength) {
    if (word == null) {
        return "";
    } else {
        return word.substring(0, Math.min(word.length(), maxLength));
    }
}

public String truncateWordOrNull(String word, int maxLength) {
    try {
        return word.substring(0, Math.min(word.length(), maxLength));
    } catch (NullPointerException ex) {
        return "";
    }
}
```

在此示例中，我们（按设计）将`word`为`null`的情况视为一个空词。两个版本分别使用传统的`if ... else`和`try ... catch`来处理`null`。我们应该如何决定哪个版本更好？

第一个标准是可读性。虽然可读性很难客观量化，但大多数程序员都会同意第一个版本的核心含义更容易理解。实际上，要真正理解第二种形式，你需要明白`NullPointerException`不能由`Math.min`或

字符串.子字符串 方法。

第二个标准是效率。在Java 8之前发布的Java版本中，第二个版本明显（数量级上）

3. Handling `Exception` properly is difficult. It is hard to know what actual exceptions may be caught, and if you don't know what could be caught, it is hard to know what recovery strategy is appropriate.
4. Handling `Throwable` is even harder, since now you also have to cope with potential failures that should never be recovered from.

This advice means that certain other patterns should be avoided. For example:

```
try {
    doSomething();
} catch (Exception ex) {
    report(ex);
    throw ex;
}
```

The above attempts to log all exceptions as they pass, without definitively handling them. Unfortunately, prior to Java 7, the `throw ex;` statement caused the compiler to think that any `Exception` could be thrown. That could force you to declare the enclosing method as `throws Exception`. From Java 7 onwards, the compiler knows that the set of exceptions that could be (re-thrown) there is smaller.

Section 131.4: Pitfall - Using exceptions for normal flowcontrol

There is a mantra that some Java experts are won't to recite:

“Exceptions should only be used for exceptional cases.”

（For example: <http://programmers.stackexchange.com/questions/184654>）

The essence of this is that it is a bad idea (in Java) to use exceptions and exception handling to implement normal flow control. For example, compare these two ways of dealing with a parameter that could be `null`.

```
public String truncateWordOrNull(String word, int maxLength) {
    if (word == null) {
        return "";
    } else {
        return word.substring(0, Math.min(word.length(), maxLength));
    }
}

public String truncateWordOrNull(String word, int maxLength) {
    try {
        return word.substring(0, Math.min(word.length(), maxLength));
    } catch (NullPointerException ex) {
        return "";
    }
}
```

In this example, we are (by design) treating the case where `word` is `null` as if it is an empty word. The two versions deal with `null` either using conventional `if ... else` and or `try ... catch`. How should we decide which version is better?

The first criterion is readability. While readability is hard to quantify objectively, most programmers would agree that the essential meaning of the first version is easier to discern. Indeed, in order to truly understand the second form, you need to understand that a `NullPointerException` cannot be thrown by the `Math.min` or `String.substring` methods.

The second criterion is efficiency. In releases of Java prior to Java 8, the second version is significantly (orders of

比第一个版本慢（数量级）。特别是，构造异常对象需要捕获并记录堆栈帧，以防需要堆栈跟踪。

另一方面，在许多情况下，使用异常处理比使用条件代码来处理“异常”事件更具可读性、更高效且（有时）更正确。实际上，也存在极少数情况需要对“非异常”事件使用异常处理；即那些相对频繁发生的事件。

对于后者，值得考虑减少创建异常对象开销的方法。

第131.5节：陷阱——直接继承 `Throwable`

`Throwable` 有两个直接子类，`Exception` 和 `Error`。虽然可以创建一个直接继承 `Throwable` 的新类，但这并不建议，因为许多应用程序假设只有 `Exception` 和 `Error` 存在。

更重要的是，直接继承 `Throwable` 没有实际好处，因为生成的类实际上只是一个受检异常。继承 `Exception` 会产生相同的行为，但能更清楚地表达你的意图。

第131.6节：陷阱——捕获 `InterruptedException`

正如其他陷阱中已经指出的，使用以下方式捕获所有异常

```
try {
    // 一些代码
} catch (Exception) {
    // 一些错误处理
}
```

会带来许多不同的问题。但其中一个特别的问题是，它可能导致死锁，因为在编写多线程应用时破坏了中断机制。

如果你启动一个线程，通常也需要能够因各种原因突然停止它。

```
线程 t = new 线程(新 可运行对象) {
    public void 运行() {
        while (true) {
            //无限执行某些操作
        }
    }
}

t.start();

//执行其他操作

// 如果线程仍然处于活动状态，应取消该线程。
// 解决此问题的更好方法是使用一个共享变量，线程定期检测该变量以实现干净退出，// 但在此示例中，我们尝试强制中断该线程。

if (t.isAlive()) {
    t.interrupt();
    t.join();
}

//继续执行程序
```

`t.interrupt()` 会在线程中抛出 `InterruptedException`，目的是关闭该线程。但是如果线程在完全停止之前需要清理一些资源怎么办？为此，它可以捕获该异常

magnitude) slower than the first version. In particular, the construction of an exception object entails capturing and recording the stackframes, just in case the stacktrace is required.

On the other hand, there are many situations where using exceptions is more readable, more efficient and (sometimes) more correct than using conditional code to deal with "exceptional" events. Indeed, there are rare situations where it is necessary to use them for "non-exceptional" events; i.e. events that occur relatively frequently. For the latter, it is worth looking at ways to reduce the overheads of creating exception objects.

Section 131.5: Pitfall - Directly subclassing `Throwable`

`Throwable` has two direct subclasses, `Exception` and `Error`. While it's possible to create a new class that extends `Throwable` directly, this is inadvisable as many applications assume only `Exception` and `Error` exist.

More to the point there is no practical benefit to directly subclassing `Throwable`, as the resulting class is, in effect, simply a checked exception. Subclassing `Exception` instead will result in the same behavior, but will more clearly convey your intent.

Section 131.6: Pitfall - Catching `InterruptedException`

As already pointed out in other pitfalls, catching all exceptions by using

```
try {
    // Some code
} catch (Exception) {
    // Some error handling
}
```

Comes with a lot of different problems. But one particular problem is that it can lead to deadlocks as it breaks the interrupt system when writing multi-threaded applications.

If you start a thread you usually also need to be able to stop it abruptly for various reasons.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        while (true) {
            //Do something indefinitely
        }
    }
}

t.start();

//Do something else

// The thread should be canceled if it is still active.
// A better way to solve this is with a shared variable that is tested
// regularly by the thread for a clean exit, but for this example we try to
// forcibly interrupt this thread.
if (t.isAlive()) {
    t.interrupt();
    t.join();
}

//Continue with program
```

The `t.interrupt()` will raise an `InterruptedException` in that thread, than is intended to shut down the thread. But what if the Thread needs to clean up some resources before its completely stopped? For this it can catch the

InterruptedException 并进行一些清理工作。

```
线程 t = new 线程(new 可运行对象() {
    public void 运行() {
        try {
            while (true) {
                //无限执行某些操作
            }
        } 捕获 (InterruptedException ex) {
            // 进行一些快速清理

            // 在这种情况下，简单的返回就可以了。
            // 但如果你不能百分之百确定线程在捕获 InterruptedException 后会结束
            // 你需要为包围这段代码的层抛出另一个异常。
        }

        Thread.currentThread().interrupt();
    }
})
```

但是如果你的代码中有一个捕获所有异常的表达式，InterruptedException 也会被它捕获，中断将不会继续。这种情况下可能导致死锁，因为父线程会无限期等待这个线程通过 `t.join()` 停止。

```
线程 t = new 线程(new 可运行对象() {
    public void 运行() {
        try {
            while (true) {
                try {
                    //无限执行某些操作
                }
            }
        } 捕获 (Exception ex) {
            ex.printStackTrace();
        }
    } 捕获 (InterruptedException ex) {
        // 这是死代码，因为中断异常已经在
        // 内层的 try-catch 中被捕获了
        Thread.currentThread().interrupt();
    }
})
```

因此，最好单独捕获异常，但如果你坚持使用通用捕获，至少要先单独捕获InterruptedException。

```
线程 t = new 线程(new 可运行对象() {
    public void 运行() {
        try {
            while (true) {
                try {
                    //无限执行某些操作
                }
            }
        } catch (InterruptedException ex) {
            throw ex; //将异常抛出到调用链上
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    } catch (InterruptedException ex) {
        // 一些快速清理代码
    }
})
```

InterruptedException and do some cleanup.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                //Do something indefinitely
            }
        } catch (InterruptedException ex) {
            //Do some quick cleanup

            // In this case a simple return would do.
            // But if you are not 100% sure that the thread ends after
            // catching the InterruptedException you will need to raise another
            // one for the layers surrounding this code.
            Thread.currentThread().interrupt();
        }
    }
})
```

But if you have a catch-all expression in your code, the InterruptedException will be caught by it as well and the interruption will not continue. Which in this case could lead to a deadlock as the parent thread waits indefinitely for this thread to stop with `t.join()`.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinitely
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    } catch (InterruptedException ex) {
        // Dead code as the interrupt exception was already caught in
        // the inner try-catch
        Thread.currentThread().interrupt();
    }
})
```

So it is better to catch Exceptions individually, but if you insist on using a catch-all, at least catch the InterruptedException individually beforehand.

```
Thread t = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                try {
                    //Do something indefinitely
                } catch (InterruptedException ex) {
                    throw ex; //Send it up in the chain
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        } catch (InterruptedException ex) {
            // Some quick cleanup code
        }
    }
})
```

```
    Thread.currentThread().interrupt();
}
}
```

第131.7节：陷阱 - 过多或不恰当的堆栈跟踪

程序员最令人恼火的行为之一就是在代码中到处散布调用printStackTrace()的语句。

问题在于printStackTrace()会将堆栈跟踪写入标准输出。

- 对于面向非Java程序员的终端用户的应用程序来说，堆栈跟踪充其量是无用的信息，最坏情况下会引起恐慌。
- 对于服务器端应用程序，几乎没人会查看标准输出。

更好的做法是不直接调用printStackTrace，或者如果调用，也应以将堆栈跟踪写入日志文件或错误文件而非终端用户控制台的方式调用。

一种方法是使用日志框架，并将异常对象作为日志事件的参数传递。

然而，即使记录异常，如果不谨慎也可能有害。请考虑以下情况：

```
public void method1() throws SomeException {
    try {
method2();
        // 执行某些操作
    } catch (SomeException ex) {
Logger.getLogger().warn("method1 中出现问题", ex);
        throw ex;
    }
}

public void method2() throws SomeException {
    try {
        // 执行其他操作
    } catch (SomeException ex) {
Logger.getLogger().warn("方法2中出现问题", ex);
        throw ex;
    }
}
```

如果异常在method2中抛出，你很可能会在日志文件中看到同一个堆栈跟踪的两个副本，
对应同一个失败。

简而言之，要么记录异常，要么将其重新抛出（可能用另一个异常包装）。不要两者都做。

```
    Thread.currentThread().interrupt();
}
}
```

Section 131.7: Pitfall - Excessive or inappropriate stacktraces

One of the more annoying things that programmers can do is to scatter calls to printStackTrace() throughout their code.

The problem is that the printStackTrace() is going to write the stacktrace to standard output.

- For an application that is intended for end-users who are not Java programmers, a stacktrace is uninformative at best, and alarming at worst.
- For a server-side application, the chances are that nobody will look at the standard output.

A better idea is to not call printStackTrace directly, or if you do call it, do it in a way that the stack trace is written to a log file or error file rather than to the end-user's console.

One way to do this is to use a logging framework, and pass the exception object as a parameter of the log event. However, even logging the exception can be harmful if done injudiciously. Consider the following:

```
public void method1() throws SomeException {
    try {
method2();
        // Do something
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method1", ex);
        throw ex;
    }
}

public void method2() throws SomeException {
    try {
        // Do something else
    } catch (SomeException ex) {
        Logger.getLogger().warn("Something bad in method2", ex);
        throw ex;
    }
}
```

If the exception is thrown in method2, you are likely to see two copies of the same stacktrace in the logfile, corresponding to the same failure.

In short, either log the exception or re-throw it further (possibly wrapped with another exception). Don't do both.

第132章：Java陷阱 - 语言语法

尽管编译正确，若错误使用Java编程语言，程序仍可能产生错误结果。本主题的主要目的是列出常见陷阱及其原因，并提出避免这些问题的正确方法。

第132.1节：陷阱 - 在'switch'语句的case中遗漏'break'

这些Java问题可能非常尴尬，有时直到生产环境运行时才被发现。

switch语句中的贯穿行为通常很有用；然而，当不希望出现这种行为时，遗漏“break”关键字可能导致灾难性后果。如果你忘记在下面代码示例中的“case 0”中放置“break”，程序将先输出“Zero”，随后输出“One”，因为这里的控制流会贯穿整个

“switch”语句，直到遇到“break”为止。例如：

```
public static void switchCasePrimer() {  
    int caseIndex = 0;  
    switch (caseIndex) {  
        case 0:  
            System.out.println("Zero");  
        case 1:  
            System.out.println("One");  
            break;  
        case 2:  
            System.out.println("Two");  
            break;  
        default:  
            System.out.println("Default");  
    }  
}
```

在大多数情况下，更简洁的解决方案是使用接口，并将具有特定行为的代码移入单独的实现中（组合优于继承）

如果无法避免使用switch语句，建议对发生的“预期”贯穿（fallthrough）进行文档说明。这样可以向其他开发者表明你知道缺少break是有意为之，这是预期的行为。

```
switch(caseIndex) {  
    [...]  
    case 2:  
        System.out.println("Two");  
        // fallthrough  
    default:  
        System.out.println("Default");
```

第132.2节：陷阱——声明与标准类同名的类

有时，刚接触Java的程序员会犯一个错误，即定义一个与广泛使用的类同名的类。例如：

```
包com.example;  
  
/**
```

Chapter 132: Java Pitfalls - Language syntax

Several Java programming language misusage might conduct a program to generate incorrect results despite being compiled correctly. This topic main purpose is to list common pitfalls with their causes, and to propose the correct way to avoid falling in such problems.

Section 132.1: Pitfall - Missing a 'break' in a 'switch' case

These Java issues can be very embarrassing, and sometimes remain undiscovered until run in production.

Fallthrough behavior in switch statements is often useful; however, missing a "break" keyword when such behavior is not desired can lead to disastrous results. If you have forgotten to put a "break" in "case 0" in the code example below, the program will write "Zero" followed by "One", since the control flow inside here will go through the entire "switch" statement until it reaches a "break". For example:

```
public static void switchCasePrimer() {  
    int caseIndex = 0;  
    switch (caseIndex) {  
        case 0:  
            System.out.println("Zero");  
        case 1:  
            System.out.println("One");  
            break;  
        case 2:  
            System.out.println("Two");  
            break;  
        default:  
            System.out.println("Default");  
    }  
}
```

In most cases, the cleaner solution would be to use interfaces and move code with specific behaviour into separate implementations (*composition over inheritance*)

If a switch-statement is unavoidable it is recommended to document "expected" fallthroughs if they occur. That way you show fellow developers that you are aware of the missing break, and that this is expected behaviour.

```
switch(caseIndex) {  
    [...]  
    case 2:  
        System.out.println("Two");  
        // fallthrough  
    default:  
        System.out.println("Default");
```

Section 132.2: Pitfall - Declaring classes with the same names as standard classes

Sometimes, programmers who are new to Java make the mistake of defining a class with a name that is the same as a widely used class. For example:

```
package com.example;  
  
/**
```

```
* 我的字符串工具类
*/
公共类String{
    ...
}
```

然后他们会疑惑为什么会出现意想不到的错误。例如：

```
包com.example;

public class Test {
    公共静态void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

如果你编译并尝试运行上述类，将会得到一个错误：

```
$ javac com/example/*.java
$ java com.example.Test
错误：在类 test.Test 中未找到主方法，请定义主方法为：
public static void main(String[] args)
或者 JavaFX 应用程序类必须继承 javafx.application.Application
```

查看 Test 类代码的人会看到 main 的声明并查看其签名，
会想知道 java 命令在抱怨什么。但实际上，java 命令说的是实话。

当我们在与 Test 相同的包中声明了一个版本的 String 时，该版本优先于自动导入的 java.lang.String。
因此，Test.main 方法的签名实际上是

```
void main(com.example.String[] args)
```

而不是

```
void main(java.lang.String[] args)
```

并且 java 命令不会将 that 识别为入口方法。

课程：不要定义与java.lang中已有类同名的类，或Java SE库中其他常用类同名的类。如果这样做，你将面临各种难以察觉的错误。

第132.3节：陷阱——遗漏大括号：“悬挂的if”和“悬挂的else”问题

Oracle Java风格指南的最新版本规定，if语句中的“then”和“else”语句应始终用“花括号”或“大括号”括起来。类似规则也适用于各种循环语句的主体部分。

```
if (a) {           // <- 开括号
    doSomething();
}                   // <- 关括号
```

实际上并非Java语言语法所强制要求。事实上，如果if语句的“then”部分只有一条语句，则可以省略大括号

```
* My string utilities
*/
public class String {
    ...
}
```

Then they wonder why they get unexpected errors. For example:

```
package com.example;

public class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

If you compile and then attempt to run the above classes you will get an error:

```
$ javac com/example/*.java
$ java com.example.Test
Error: Main method not found in class test.Test, please define the main method as:
public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Someone looking at the code for the Test class would see the declaration of main and look at its signature and
wonder what the java command is complaining about. But in fact, the java command is telling the truth.

When we declare a version of String in the same package as Test, this version takes precedence over the
automatic import of java.lang.String. Thus, the signature of the Test.main method is actually

```
void main(com.example.String[] args)
```

instead of

```
void main(java.lang.String[] args)
```

and the java command will not recognize that as an entrypoint method.

Lesson: Do not define classes that have the same name as existing classes in java.lang, or other commonly used
classes in the Java SE library. If you do that, you are setting yourself open for all sorts of obscure errors.

Section 132.3: Pitfall - Leaving out braces: the "dangling if" and "dangling else" problems

The latest version of the Oracle Java style guide mandates that the "then" and "else" statements in an if statement
should always be enclosed in "braces" or "curly brackets". Similar rules apply to the bodies of various loop
statements.

```
if (a) {           // <- open brace
    doSomething();
    doSomeMore();
}                   // <- close brace
```

This is not actually required by Java language syntax. Indeed, if the "then" part of an if statement is a single
statement, it is legal to leave out the braces

```
if (a)  
doSomething();
```

甚至

```
if (a) doSomething();
```

但是，忽略 Java 风格规则并省略大括号存在风险。具体来说，这会大大增加因代码缩进错误而被误读的风险。

“悬挂的 if” 问题：

考虑上面示例代码，去掉大括号后的写法。

```
if (a)  
doSomething();  
doSomeMore();
```

这段代码似乎表示只有当 a 为 true 时，才会调用 doSomething 和 doSomeMore。实际上，代码缩进错误。Java 语言规范规定 doSomeMore() 调用是紧跟在 if 语句之后的独立语句。

正确的缩进如下：

```
if (a)  
doSomething();  
doSomeMore();
```

“悬挂的 else” 问题

当我们加入 else 时，会出现第二个问题。考虑以下缺少大括号的示例。

```
if (a)  
    if (b)  
        doX();  
    else if (c)  
        doY();  
else  
    doZ();
```

上面的代码似乎表示当a为false时会调用doZ。实际上，缩进再次不正确。

代码的正确缩进是：

```
if (a)  
    if (b)  
        doX();  
    else if (c)  
        doY();  
    else  
        doZ();
```

如果代码按照Java风格规则编写，实际上会是这样：

```
if (a) {  
    if (b) {  
        doX();  
    } else if (c) {  
        doY();  
    } else {
```

```
if (a)  
    doSomething();
```

or even

```
if (a) doSomething();
```

However there are dangers in ignoring Java style rules and leaving out the braces. Specifically, you significantly increase the risk that code with faulty indentation will be misread.

The "dangling if" problem:

Consider the example code from above, rewritten without braces.

```
if (a)  
doSomething();  
doSomeMore();
```

This code *seems to say* that the calls to doSomething and doSomeMore will both occur *if and only if* a is **true**. In fact, the code is incorrectly indented. The Java Language Specification states that the doSomeMore() call is a separate statement following the if statement. The correct indentation is as follows:

```
if (a)  
    doSomething();  
doSomeMore();
```

The "dangling else" problem

A second problem appears when we add **else** to the mix. Consider the following example with missing braces.

```
if (a)  
    if (b)  
        doX();  
    else if (c)  
        doY();  
else  
    doZ();
```

The code above *seems to say* that doZ will be called when a is **false**. In fact, the indentation is incorrect once again. The correct indentation for the code is:

```
if (a)  
    if (b)  
        doX();  
    else if (c)  
        doY();  
    else  
        doZ();
```

If the code was written according to the Java style rules, it would actually look like this:

```
if (a) {  
    if (b) {  
        doX();  
    } else if (c) {  
        doY();  
    } else {
```

```
doZ();  
}  
}
```

为了说明为什么那样更好，假设你不小心缩进错误了代码。你可能会得到类似这样的代码：

```
if (a) {  
    if (b) {  
        doX();  
    } else if (c) {  
        doY();  
    } else {  
        doZ();  
    }  
}  
  
if (a) {  
    if (b) {  
        doX();  
    } else if (c) {  
        doY();  
    } else {  
        doZ();  
    }  
}
```

但在这两种情况下，缩进错误的代码对有经验的Java程序员来说“看起来都是错的”。

第132.4节：陷阱 - 八进制字面量

考虑以下代码片段：

```
// 打印数字1到10的和  
int count = 0;  
for (int i = 1; i < 010; i++) { // 这里有错误 ....  
count = count + i;  
}  
System.out.println("1到10的和是 " + count);
```

一个Java初学者可能会惊讶地发现，上述程序输出了错误的结果。它实际上打印的是数字1到8的和。

原因是以数字零 ('0') 开头的整数字面量被Java编译器解释为八进制字面量，而不是你可能期望的十进制字面量。因此，010是八进制数10，换算成十进制是8。

第132.5节：陷阱——使用'=='测试布尔值

有时新手Java程序员会写出如下代码：

```
public void check(boolean ok) {  
    if (ok == true) { // 注意 'ok == true'  
    }  
}
```

有经验的程序员会觉得这样写很笨拙，想改写成：

```
public void check(boolean ok) {  
    if (ok) {  
        System.out.println("没问题");  
    }  
}
```

然而，`ok == true`的问题不仅仅是笨拙。考虑下面这个变体：

```
public void check(boolean ok) {
```

```
    doZ();  
}  
}
```

To illustrate why that is better, suppose that you had accidentally mis-indented the code. You might end up with something like this:

```
if (a) {  
    if (b) {  
        doX();  
    } else if (c) {  
        doY();  
    } else {  
        doZ();  
    }  
}  
  
if (a) {  
    if (b) {  
        doX();  
    } else if (c) {  
        doY();  
    } else {  
        doZ();  
    }  
}
```

But in both cases, the mis-indented code "looks wrong" to the eye of an experienced Java programmer.

Section 132.4: Pitfall - Octal literals

Consider the following code snippet:

```
// Print the sum of the numbers 1 to 10  
int count = 0;  
for (int i = 1; i < 010; i++) { // Mistake here ....  
count = count + i;  
}  
System.out.println("The sum of 1 to 10 is " + count);
```

A Java beginner might be surprised to know that the above program prints the wrong answer. It actually prints the sum of the numbers 1 to 8.

The reason is that an integer literal that starts with the digit zero ('0') is interpreted by the Java compiler as an octal literal, not a decimal literal as you might expect. Thus, 010 is the octal number 10, which is 8 in decimal.

Section 132.5: Pitfall - Using '==' to test a boolean

Sometimes a new Java programmer will write code like this:

```
public void check(boolean ok) {  
    if (ok == true) { // Note 'ok == true'  
        System.out.println("It is OK");  
    }  
}
```

An experienced programmer would spot that as being clumsy and want to rewrite it as:

```
public void check(boolean ok) {  
    if (ok) {  
        System.out.println("It is OK");  
    }  
}
```

However, there is more wrong with `ok == true` than simple clumsiness. Consider this variation:

```
public void check(boolean ok) {
```

```
if (ok = true) {           // 哎呀！  
    System.out.println("没问题");  
}
```

这里程序员错误地将 `==` 写成了 `= ...`。现在代码中有一个微妙的错误。表达式 `x = true` 无条件地将 `true` 赋值给 `x`，随后结果为 `true`。换句话说，`check` 方法现在无论参数是什么都会打印 "It is OK"。

这里的教训是不要养成使用 `== false` 和 `== true` 的习惯。除了冗长之外，它们还会使你的编码更容易出错。

注意：避免 `ok == true` 陷阱的一个可能替代方法是使用 Yoda 条件；即将字面量放在关系运算符的左侧，如 `true == ok`。这样可以，但大多数程序员可能会觉得 Yoda 条件看起来很奇怪。当然，`ok`（或 `!ok`）更简洁、更自然。

```
if (ok = true) {           // Ooops!  
    System.out.println("It is OK");  
}
```

Here the programmer has mistyped `==` as `= ...` and now the code has a subtle bug. The expression `x = true` unconditionally assigns `true` to `x` and then evaluates to `true`. In other words, the `check` method will now print "It is OK" no matter what the parameter was.

The lesson here is to get out of the habit of using `== false` and `== true`. In addition to being verbose, they make your coding more error prone.

Note: A possible alternative to `ok == true` that avoids the pitfall is to use [Yoda conditions](#); i.e. put the literal on the left side of the relational operator, as in `true == ok`. This works, but most programmers would probably agree that Yoda conditions look odd. Certainly `ok` (or `!ok`) is more concise and more natural.

第132.6节：陷阱——忽视方法的可见性

即使是有经验的Java开发者也往往认为Java只有三种访问修饰符。实际上语言有四种！`package private`（又称默认）级别的可见性经常被忽视。

你应该注意哪些方法设置为`public`。应用程序中的`public`方法是应用程序的可见API。这个API应尽可能小且紧凑，尤其是当你编写可重用库时（参见SOLID原则）。同样重要的是考虑所有方法的可见性，并且只在适当情况下使用`protected`或`package private`访问权限。

当你将本应为`private`的方法声明为`public`时，你就暴露了类的内部实现细节。

一个推论是，你只对类的公共方法进行单元测试——实际上你只能测试公共方法。为了能够对那些方法进行单元测试而增加私有方法的可见性是不好的做法。测试调用了更严格可见性方法的公共方法，应该足以测试整个API。你绝不应该仅为了单元测试而增加更多公共方法来扩展你的API。

第132.7节：陷阱：使用“assert”进行参数或用户输入验证

StackOverflow上偶尔会有人问，是否适合使用`assert`来验证传递给方法的参数，甚至是用户提供的输入。

简单的回答是不适合。

更好的替代方案包括：

- 使用自定义代码抛出`IllegalArgumentException`（非法参数异常）。
- 使用Google Guava库中提供的`Preconditions`方法。
- 使用Apache Commons Lang3库中提供的`Validate`方法。

这是Java语言规范 ([JLS 14.10, 针对Java 8](#)) 对此事的建议：

通常，断言检查在程序开发和测试期间启用，部署时禁用，以提高性能。

由于断言可能被禁用，程序不能假设其中包含的表达式

Section 132.6: Pitfall - Ignoring method visibility

Even experienced Java developers tend to think that Java has only three protection modifiers. The language actually has four! The `package private` (a.k.a. default) level of visibility is often forgotten.

You should pay attention to what methods you make public. The public methods in an application are the application's visible API. This should be as small and compact as possible, especially if you are writing a reusable library (see also the [SOLID](#) principle). It is important to similarly consider the visibility of all methods, and to only use protected or package private access where appropriate.

When you declare methods that should be `private` as public, you expose the internal implementation details of the class.

A corollary to this is that you only [unit test](#) the public methods of your class - in fact you can **only** test public methods. It is bad practice to increase the visibility of private methods just to be able to run unit tests against those methods. Testing public methods that call the methods with more restrictive visibility should be sufficient to test an entire API. You should **never** expand your API with more public methods only to allow unit testing.

Section 132.7: Pitfall: Using 'assert' for argument or user input validation

A question that occasionally on StackOverflow is whether it is appropriate to use `assert` to validate arguments supplied to a method, or even inputs provided by the user.

The simple answer is that it is not appropriate.

Better alternatives include:

- Throwing an `IllegalArgumentException` using custom code.
- Using the `Preconditions` methods available in Google Guava library.
- Using the `Validate` methods available in Apache Commons Lang3 library.

This is what the [Java Language Specification \(JLS 14.10, for Java 8\)](#) advises on this matter:

Typically, assertion checking is enabled during program development and testing, and disabled for deployment, to improve performance.

Because assertions may be disabled, programs must not assume that the expressions contained in

断言将被评估。因此，这些布尔表达式通常应无副作用。

评估此类布尔表达式不应影响评估完成后可见的任何状态。断言中包含布尔表达式具有副作用并非非法，但通常不合适，因为这可能导致程序行为因断言是否启用而异。

鉴于此，断言不应用于公共方法中的参数检查。参数检查通常是方法契约的一部分，无论断言是否启用，该契约都必须得到遵守。

使用断言进行参数检查的另一个问题是，错误的参数应导致适当的运行时异常（例如IllegalArgumentExcept ion、

ArrayIndexOutOfBoundsException或NullPointerException）。断言失败不会抛出适当的异常。同样，使用断言对公共方法进行参数检查并非非法，但通常不合适。断言错误（AssertionError）本不应被捕获，但实际上可以捕获，因此try语句的规则应将try块中出现的断言视为与当前对throw语句的处理类似。

assertions will be evaluated. Thus, these boolean expressions should generally be free of side effects. Evaluating such a boolean expression should not affect any state that is visible after the evaluation is complete. It is not illegal for a boolean expression contained in an assertion to have a side effect, but it is generally inappropriate, as it could cause program behavior to vary depending on whether assertions were enabled or disabled.

In light of this, assertions should not be used for argument checking in public methods. Argument checking is typically part of the contract of a method, and this contract must be upheld whether assertions are enabled or disabled.

A secondary problem with using assertions for argument checking is that erroneous arguments should result in an appropriate run-time exception (such as [IllegalArgumentException](#), [ArrayIndexOutOfBoundsException](#), or [NullPointerException](#)). An assertion failure will not throw an appropriate exception. Again, it is not illegal to use assertions for argument checking on public methods, but it is generally inappropriate. It is intended that Assertion Error never be caught, but it is possible to do so, thus the rules for try statements should treat assertions appearing in a try block similarly to the current treatment of throw statements.

第132.8节：陷阱——通配符导入可能使代码变得脆弱

考虑以下部分示例：

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context(); // 来自 com.example.somelib
    ...
}
```

假设当你最初针对版本 1.0 的 somelib 和版本 1.0 的 otherlib 开发代码时。

然后在某个后续时间点，你需要将依赖项升级到更高版本，并且你决定使用 otherlib 版本 2.0。还假设他们在 otherlib 从 1.0 到 2.0 之间所做的更改之一是添加了一个 Context 类。

现在当你重新编译 Test 时，你会收到一个编译错误，告诉你 Context 是一个模糊的导入。

如果你熟悉代码库，这可能只是一个小麻烦。如果不熟悉，那么你需要做一些工作来解决这个问题，这里以及可能的其他地方。

这里的问题是通配符导入。一方面，使用通配符可以让你的类少写几行代码。另一方面：

- 对代码库其他部分、Java 标准库或第三方库的向上兼容更改可能导致编译错误。
- 可读性下降。除非你使用集成开发环境（IDE），否则很难弄清楚哪个通配符导入引入了某个命名类。

教训是，在需要长期维护的代码中使用通配符导入是个坏主意。如果你使用 IDE，维护具体的（非通配符）导入并不费力，而且这份努力是值得的。

Section 132.8: Pitfall - Wildcard imports can make your code fragile

Consider the following partial example:

```
import com.example.somelib.*;
import com.acme.otherlib.*;

public class Test {
    private Context x = new Context(); // from com.example.somelib
    ...
}
```

Suppose that when you first developed the code against version 1.0 of somelib and version 1.0 of otherlib. Then at some later point, you need to upgrade your dependencies to a later versions, and you decide to use otherlib version 2.0. Also suppose that one of the changes that they made to otherlib between 1.0 and 2.0 was to add a [Context](#) class.

Now when you recompile Test, you will get a compilation error telling you that [Context](#) is an ambiguous import.

If you are familiar with the codebase, this probably is just a minor inconvenience. If not, then you have some work to do to address this problem, here and potentially elsewhere.

The problem here is the wildcard imports. On the one hand, using wildcards can make your classes a few lines shorter. On the other hand:

- Upwards compatible changes to other parts of your codebase, to Java standard libraries or to 3rd party libraries can lead to compilation errors.
- Readability suffers. Unless you are using an IDE, figuring out which of the wildcard imports is pulling in a named class can be difficult.

The lesson is that it is a bad idea to use wildcard imports in code that needs to be long lived. Specific (non-wildcard) imports are not much effort to maintain if you use an IDE, and the effort is worthwhile.

第132.9节：陷阱 - 分号位置错误和缺失的大括号

这是一个会让Java初学者真正困惑的错误，至少他们第一次犯错时会如此。正确的写法应该是：

```
if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");
```

但他们却不小心写成了：

```
if (feeling == HAPPY);
    System.out.println("Smile");
else
    System.out.println("Frown");
```

当Java编译器告诉他们else位置错误时，他们感到困惑。Java编译器会将上述代码解释为：

```
if (feeling == HAPPY)
    /*空语句*/
System.out.println("Smile"); // 这是无条件执行的
else
    // 这是位置错误的。语句不能
    // 以'else'开头
System.out.println("Frown");
```

在其他情况下，虽然不会有编译错误，但代码不会按程序员的意图执行。例如：

```
for (int i = 0; i < 5; i++)
    System.out.println("Hello");
```

只打印一次“Hello”。再次强调，多余的分号意味着for循环体是一个空语句。这意味着紧随其后的println调用是无条件执行的。

另一种变体：

```
for (int i = 0; i < 5; i++)
    System.out.println("数字是 " + i);
```

这会导致 i 的“找不到符号”错误。多余的分号意味着 println 调用试图在 i 的作用域之外使用它。

在这些例子中，有一个直接的解决方案：只需删除多余的分号即可。然而，这些例子还带来了一些更深层的教训：

1. Java中的分号不是“语法噪音”。分号的有无会改变程序的含义。不要只是简单地在每行末尾添加分号。
2. 不要相信代码的缩进。在Java语言中，行首的额外空白符会被编译器忽略。
3. 使用自动缩进工具。所有IDE和许多简单的文本编辑器都能正确缩进Java

Section 132.9: Pitfall - Misplaced semicolons and missing braces

This is a mistake that causes real confusion for Java beginners, at least the first time that they do it. Instead of writing this:

```
if (feeling == HAPPY)
    System.out.println("Smile");
else
    System.out.println("Frown");
```

they accidentally write this:

```
if (feeling == HAPPY);
    System.out.println("Smile");
else
    System.out.println("Frown");
```

and are puzzled when the Java compiler tells them that the `else` is misplaced. The Java compiler will interpret the above as follows:

```
if (feeling == HAPPY)
    /*empty statement*/
System.out.println("Smile"); // This is unconditional
else
    // This is misplaced. A statement cannot
    // start with 'else'
System.out.println("Frown");
```

In other cases, there will be no compilation errors, but the code won't do what the programmer intends. For example:

```
for (int i = 0; i < 5; i++)
    System.out.println("Hello");
```

only prints "Hello" once. Once again, the spurious semicolon means that the body of the `for` loop is an empty statement. That means that the `println` call that follows is unconditional.

Another variation:

```
for (int i = 0; i < 5; i++)
    System.out.println("The number is " + i);
```

This will give a "Cannot find symbol" error for `i`. The presence of the spurious semicolon means that the `println` call is attempting to use `i` outside of its scope.

In those examples, there is a straight-forward solution: simply delete the spurious semicolon. However, there are some deeper lessons to be drawn from these examples:

1. The semicolon in Java is not "syntactic noise". The presence or absence of a semicolon can change the meaning of your program. Don't just add them at the end of every line.
2. Don't trust your code's indentation. In the Java language, extra whitespace at the beginning of a line is ignored by the compiler.
3. Use an automatic indenter. All IDEs and many simple text editors understand how to correctly indent Java

代码。

4. 这是最重要的教训。遵循最新的Java风格指南，并在“then”和“else”语句以及循环体语句周围加上大括号。左大括号({})不应另起一行。

如果程序员遵循了风格规则，那么带有错误分号的if示例应该是这样的：

```
if (feeling == HAPPY); {  
    System.out.println("Smile");  
} else {  
    System.out.println("Frown");  
}
```

这对有经验的人来说看起来很奇怪。如果你自动缩进这段代码，它可能看起来像这样：

```
if (feeling == HAPPY); {  
    System.out.println("Smile");  
} else {  
    System.out.println("Frown");  
}
```

即使是初学者也应该能看出这是错误的。

第132.10节：陷阱——重载而非重写

考虑以下示例：

```
public final class Person {  
    private final String firstName;  
    private final String lastName;  
  
    public Person(String firstName, String lastName) {  
        this.firstName = (firstName == null) ? "" : firstName;  
        this.lastName = (lastName == null) ? "" : lastName;  
    }  
  
    public boolean equals(String other) {  
        if (!(other instanceof Person)) {  
            return false;  
        }  
        Person p = (Person) other;  
        return firstName.equals(p.firstName) &&  
               lastName.equals(p.lastName);  
    }  
  
    public int hashCode() {  
        return firstName.hashCode() + 31 * lastName.hashCode();  
    }  
}
```

这段代码不会按预期运行。问题在于Person的equals和hashCode方法没有覆盖Object定义的标准方法。

- equals方法的签名错误。它应该声明为equals(Object)而不是equals(String)。
- hashCode方法的名称错误。它应该是hashCode()（注意大写的C）。

这些错误意味着我们声明了意外的重载，如果Person被使用，这些重载将不会被调用。

code.

4. This is the most important lesson. Follow the latest Java style guidelines, and put braces around the "then" and "else" statements and the body statement of a loop. The open brace ({}) should not be on a new line.

If the programmer followed the style rules then the if example with a misplaced semicolons would look like this:

```
if (feeling == HAPPY); {  
    System.out.println("Smile");  
} else {  
    System.out.println("Frown");  
}
```

That looks odd to an experienced eye. If you auto-indented that code, it would probably look like this:

```
if (feeling == HAPPY); {  
    System.out.println("Smile");  
} else {  
    System.out.println("Frown");  
}
```

which should stand out as wrong to even a beginner.

Section 132.10: Pitfall - Overloading instead of overriding

Consider the following example:

```
public final class Person {  
    private final String firstName;  
    private final String lastName;  
  
    public Person(String firstName, String lastName) {  
        this.firstName = (firstName == null) ? "" : firstName;  
        this.lastName = (lastName == null) ? "" : lastName;  
    }  
  
    public boolean equals(String other) {  
        if (!(other instanceof Person)) {  
            return false;  
        }  
        Person p = (Person) other;  
        return firstName.equals(p.firstName) &&  
               lastName.equals(p.lastName);  
    }  
  
    public int hashCode() {  
        return firstName.hashCode() + 31 * lastName.hashCode();  
    }  
}
```

This code is not going to behave as expected. The problem is that the equals and hashCode methods for Person do not override the standard methods defined by Object.

- The equals method has the wrong signature. It should be declared as equals(Object) not equals(String).
- The hashCode method has the wrong name. It should be hashCode() (note the capital C).

These mistakes mean that we have declared accidental overloads, and these won't be used if Person is used in a

多态上下文。

但是，有一个简单的方法可以解决这个问题（从Java 5开始）。每当你打算让你的方法成为重写时，使用@Override注解：

```
版本 ≥ Java SE 5
public final class Person {
    ...
    @Override
    public boolean equals(String other) {
        ...
    }

    @Override
    public hashCode() {
        ...
    }
}
```

当我们在方法声明中添加@Override注解时，编译器会检查该方法是否确实重写（或实现）了超类或接口中声明的方法。因此，在上面的例子中，编译器会给出两个编译错误，这足以提醒我们存在错误。

第132.11节：自动拆箱空对象为基本类型的陷阱

```
public class Foobar {
    public static void main(String[] args) {

        // 示例：
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

这里的陷阱是将`null`与`false`进行比较。由于我们是在比较原始类型的`boolean`与`Boolean`对象，Java尝试将`Boolean`对象拆箱为对应的原始类型，以便进行比较。然而，由于该值为`null`，因此会抛出`NullPointerException`异常。

Java无法将原始类型与`null`值进行比较，这会导致运行时抛出`NullPointerException`异常。
考虑原始类型条件`false == null`；这会在编译时产生错误，
错误信息为不可比较的类型：`int`和`<null>`。

polymorphic context.

However, there is a simple way to deal with this (from Java 5 onwards). Use the `@Override` annotation whenever you intend your method to be an override:

```
Version ≥ Java SE 5
public final class Person {
    ...
    @Override
    public boolean equals(String other) {
        ...
    }

    @Override
    public hashCode() {
        ...
    }
}
```

When we add an `@Override` annotation to a method declaration, the compiler will check that the method *does* override (or implement) a method declared in a superclass or interface. So in the example above, the compiler will give us two compilation errors, which should be enough to alert us to the mistake.

Section 132.11: Pitfall of Auto-Unboxing Null Objects into Primitives

```
public class Foobar {
    public static void main(String[] args) {

        // example:
        Boolean ignore = null;
        if (ignore == false) {
            System.out.println("Do not ignore!");
        }
    }
}
```

The pitfall here is that `null` is compared to `false`. Since we're comparing a primitive `boolean` against a `Boolean`, Java attempts to *unbox* the the `Boolean Object` into a primitive equivalent, ready for comparison. However, since that value is `null`, a `NullPointerException` is thrown.

Java is incapable of comparing primitive types against `null` values, which causes a `NullPointerException` at runtime. Consider the primitive case of the condition `false == null`; this would generate a *compile time* error for incomparable types: `int` and `<null>`.

第133章：Java陷阱——线程与并发

第133.1节：陷阱——继承“java.lang.Thread”

Thread类的javadoc展示了定义和使用线程的两种方式：

使用自定义线程类：

```
class PrimeThread extends Thread {  
    long minPrime;  
PrimeThread(long minPrime) {  
    this.minPrime = minPrime;  
}  
  
public void run() {  
    // 计算大于minPrime的素数  
    . . .  
}  
}
```

```
PrimeThread p = new PrimeThread(143);  
p.start();
```

使用Runnable：

```
class PrimeRun implements Runnable {  
    long minPrime;  
PrimeRun(long minPrime) {  
    this.minPrime = minPrime;  
}  
  
public void run() {  
    // 计算大于minPrime的素数  
    . . .  
}  
}
```

```
PrimeRun p = new PrimeRun(143);  
new Thread(p).start();
```

(来源：[java.lang.Thread javadoc](#).)

自定义线程类的方法可行，但存在一些问题：

1. 在使用经典线程池、执行器（executor）或ForkJoin框架的环境中使用PrimeThread很不方便。
(这并非不可能，因为PrimeThread间接实现了Runnable接口，但将自定义的Thread类作为Runnable使用显然很笨拙，且可能不可行.....这取决于该类的其他方面。)
2. 其他方法更容易出错。例如，如果你声明了一个PrimeThread.start()如果不委托给 Thread.start()，你最终会得到一个在当前线程上运行的“线程”。

将线程逻辑放入 Runnable 的方法避免了这些问题。实际上，如果你使用匿名类（Java 1.1 及以后版本）来实现 Runnable，结果会比上述示例更简洁、更易读。

Chapter 133: Java Pitfalls - Threads and Concurrency

Section 133.1: Pitfall - Extending 'java.lang.Thread'

The javadoc for the [Thread](#) class shows two ways to define and use a thread:

Using a custom thread class:

```
class PrimeThread extends Thread {  
    long minPrime;  
PrimeThread(long minPrime) {  
    this.minPrime = minPrime;  
}  
  
public void run() {  
    // compute primes larger than minPrime  
    . . .  
}  
}  
  
PrimeThread p = new PrimeThread(143);  
p.start();
```

Using a [Runnable](#):

```
class PrimeRun implements Runnable {  
    long minPrime;  
PrimeRun(long minPrime) {  
    this.minPrime = minPrime;  
}  
  
public void run() {  
    // compute primes larger than minPrime  
    . . .  
}  
}  
  
PrimeRun p = new PrimeRun(143);  
new Thread(p).start();
```

(Source: [java.lang.Thread javadoc](#).)

The custom thread class approach works, but it has a few problems:

1. It is awkward to use PrimeThread in a context that uses a classic thread pool, an executor, or the ForkJoin framework. (It is not impossible, because PrimeThread indirectly implements [Runnable](#), but using a custom [Thread](#) class as a [Runnable](#) is certainly clumsy, and may not be viable ... depending on other aspects of the class.)
2. There is more opportunity for mistakes in other methods. For example, if you declared a `PrimeThread.start()` without delegating to `Thread.start()`, you would end up with a "thread" that ran on the current thread.

The approach of putting the thread logic into a [Runnable](#) avoids these problems. Indeed, if you use an anonymous class (Java 1.1 onwards) to implement the [Runnable](#) the result is more succinct, and more readable than the

上述示例。

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // 计算大于minPrime的素数
    }
}).start();
```

使用 lambda 表达式 (Java 8 及以后版本) , 上述示例会变得更加优雅 :

```
final long minPrime = ...
new Thread(() -> {
    // 计算大于 minPrime 的素数
}).start();
```

第133.2节：陷阱 - 线程过多会使应用程序变慢

许多刚接触多线程的人认为使用线程会自动让应用程序运行得更快。事实上，情况要复杂得多。但我们可以肯定地说，对于任何计算机来说，同时运行的线程数量是有限制的：

- 计算机有固定数量的核心（或超线程）。
- Java线程必须被调度到一个核心或超线程上才能运行。
- 如果可运行的Java线程数量超过了（可用的）核心/超线程数量，其中一些线程必须等待。

这告诉我们，单纯创建越来越多的Java线程不能让应用程序运行得越来越快。但还有其他因素需要考虑：

- 每个线程都需要一个堆外内存区域来存放其线程栈。典型的（默认）线程栈大小是512KB或1MB。如果线程数量较多，内存使用量可能会很大。
- 每个活动线程都会引用堆中的多个对象。这会增加可达对象的工作集，进而影响垃圾回收和物理内存的使用。
- 线程切换的开销不可忽视。它通常涉及切换到操作系统内核空间以做出线程调度决策。
- 线程同步和线程间信号（例如wait()、notify() / notifyAll()）的开销可能很大。

根据应用程序的具体情况，这些因素通常意味着线程数量存在一个“最佳点”。超过这个点，增加线程数量带来的性能提升很小，甚至可能导致性能下降。

如果您的应用程序为每个新任务创建实例，那么工作负载的意外增加（例如，高请求率）可能导致灾难性行为。

处理此问题的更好方法是使用大小可控（静态或动态）的有界线程池。

当工作量过大时，应用程序需要将请求排队。如果使用ExecutorService，它将负责线程池管理和任务排队。

examples above.

```
final long minPrime = ...
new Thread(new Runnable() {
    public void run() {
        // compute primes larger than minPrime
    }
}).start();
```

With a lambda expression (Java 8 onwards), the above example would become even more elegant:

```
final long minPrime = ...
new Thread(() -> {
    // compute primes larger than minPrime
}).start();
```

Section 133.2: Pitfall - Too many threads makes an application slower

A lot of people who are new to multi-threading think that using threads automatically make an application go faster. In fact, it is a lot more complicated than that. But one thing that we can state with certainty is that for any computer there is a limit on the number of threads that can be run at the same time:

- A computer has a fixed number of cores (or hyperthreads).
- A Java thread has to be scheduled to a core or hyperthread in order to run.
- If there are more runnable Java threads than (available) cores / hyperthreads, some of them must wait.

This tells us that simply creating more and more Java threads *cannot* make the application go faster and faster. But there are other considerations as well:

- Each thread requires an off-heap memory region for its thread stack. The typical (default) thread stack size is 512Kbytes or 1Mbytes. If you have a significant number of threads, the memory usage can be significant.
- Each active thread will refer to a number of objects in the heap. That increases the working set of *reachable* objects, which impacts on garbage collection and on physical memory usage.
- The overheads of switching between threads is non-trivial. It typically entails a switch into the OS kernel space to make a thread scheduling decision.
- The overheads of thread synchronization and inter-thread signaling (e.g. wait(), notify() / notifyAll()) *can be* significant.

Depending on the details of your application, these factors generally mean that there is a "sweet spot" for the number of threads. Beyond that, adding more threads gives minimal performance improvement, and can make performance worse.

If your application creates for each new task, then an unexpected increase in the workload (e.g. a high request rate) can lead to catastrophic behavior.

A better way to deal with this is to use bounded thread pool whose size you can control (statically or dynamically). When there is too much work to do, the application needs to queue the requests. If you use an ExecutorService, it will take care of the thread pool management and task queuing.

第133.3节：陷阱：错误使用wait() / notify()

方法object.wait()、object.notify()和object.notifyAll()应以非常特定的方式使用。（参见 <http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307>）

“丢失通知”问题

一个常见的新手错误是无条件调用object.wait()

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait(); // 不要这样做！
    }
    doSomething();
}
```

错误的原因在于它依赖于其他线程调用lock.notify()或lock.notifyAll()，但没有任何保证其他线程不会在消费者线程调用lock.wait()之前就已经进行了该调用。

如果没有其他线程已经等待通知，lock.notify()和lock.notifyAll()根本不会执行任何操作。示例中调用myConsumer()的线程如果来不及捕获通知，将会永远挂起。

“非法监视器状态”错误

如果在未持有锁的情况下调用wait()或notify()，JVM将抛出IllegalMonitorStateException异常。

```
public void myConsumer() {
    lock.wait(); // 抛出异常
    consume();
}

public void myProducer() {
    produce();
    lock.notify(); // 抛出异常
}
```

(wait() / notify()设计要求必须持有锁，因为这是避免系统性竞态条件所必需的。如果可以在不加锁的情况下调用wait()或notify()，那么就无法实现这些原语的主要用例：等待某个条件的发生。)

等待/通知过于底层

避免使用wait()和notify()的最佳方法是根本不使用它们。大多数同步问题都可以通过使用java.util.concurrent包中提供的更高级别的同步对象（队列、屏障、信号量等）来解决。

第133.4节：陷阱：共享变量需要适当的同步

考虑以下示例：

```
public class ThreadTest implements Runnable {
```

Section 133.3: Pitfall: incorrect use of wait() / notify()

The methods object.wait(), object.notify() and object.notifyAll() are meant to be used in a very specific way. (see <http://stackoverflow.com/documentation/java/5409/wait-notify#t=20160811161648303307>)

The "Lost Notification" problem

One common beginner mistake is to unconditionally call object.wait()

```
private final Object lock = new Object();

public void myConsumer() {
    synchronized (lock) {
        lock.wait(); // DON'T DO THIS!
    }
    doSomething();
}
```

The reason this is wrong is that it depends on some other thread to call lock.notify() or lock.notifyAll(), but nothing guarantees that the other thread did not make that call before the consumer thread called lock.wait().

lock.notify() and lock.notifyAll() do not do anything at all if some other thread is not already waiting for the notification. The thread that calls myConsumer() in this example will hang forever if it is too late to catch the notification.

The "Illegal Monitor State" bug

If you call wait() or notify() on an object without holding the lock, then the JVM will throw IllegalMonitorStateException.

```
public void myConsumer() {
    lock.wait(); // throws exception
    consume();
}

public void myProducer() {
    produce();
    lock.notify(); // throws exception
}
```

(The design for wait() / notify() requires that the lock is held because this is necessary to avoid systemic race conditions. If it was possible to call wait() or notify() without locking, then it would be impossible to implement the primary use-case for these primitives: waiting for a condition to occur.)

Wait / notify is too low-level

The best way to avoid problems with wait() and notify() is to not use them. Most synchronization problems can be solved by using the higher-level synchronization objects (queues, barriers, semaphores, etc.) that are available in the java.util.concurrent package.

Section 133.4: Pitfall: Shared variables require proper synchronization

Consider this example:

```
public class ThreadTest implements Runnable {
```

```

private boolean stop = false;

public void run() {
    long counter = 0;
    while (!stop) {
        counter = counter + 1;
        System.out.println("Counted " + counter);
    }
}

public static void main(String[] args) {
    ThreadTest tt = new ThreadTest();
    new Thread(tt).start(); // 创建并启动子线程
    Thread.sleep(1000);
    tt.stop = true; // 告诉子线程停止。
}
}

```

该程序的意图是启动一个线程，让它运行1000毫秒，然后通过设置stop标志使其停止。

它会按预期工作吗？

可能会，也可能不会。

当main方法返回时，应用程序不一定会停止。如果创建了另一个线程，且该线程未被标记为守护线程，那么主线程结束后应用程序将继续运行。在本例中，这意味着应用程序会一直运行，直到子线程结束。子线程结束的条件是当 tt.stop 被设置为true时。

但这实际上并不完全正确。事实上，子线程会在它观察到stop的值为true之后停止。

这会发生吗？可能会，也可能不会。

Java语言规范保证线程中进行的内存读写操作对该线程是可见的，且按照源代码中语句的顺序执行。然而，通常情况下，当一个线程写入，另一个线程随后读取时，这种可见性并不保证。要保证可见性，写操作和随后的读操作之间必须存在一条happens-before关系链。在上述示例中，更新stop标志没有这样的关系链，因此不能保证子线程会看到stop变为true。

(作者注：关于Java内存模型的详细技术细节应另设专题讨论。)

我们如何解决这个问题？

在这种情况下，有两种简单的方法可以确保stop更新是可见的：

1. 声明stop为volatile；即

```
private volatile boolean stop = false;
```

对于volatile变量，JLS规定一个线程的写操作与第二个线程随后读操作之间存在happens-before关系。

2. 使用互斥锁进行如下同步：

```
public class ThreadTest implements Runnable {
```

```

private boolean stop = false;

public void run() {
    long counter = 0;
    while (!stop) {
        counter = counter + 1;
    }
    System.out.println("Counted " + counter);
}

public static void main(String[] args) {
    ThreadTest tt = new ThreadTest();
    new Thread(tt).start(); // Create and start child thread
    Thread.sleep(1000);
    tt.stop = true; // Tell child thread to stop.
}
}

```

The intent of this program is intended to start a thread, let it run for 1000 milliseconds, and then cause it to stop by setting the stop flag.

Will it work as intended?

Maybe yes, maybe no.

An application does not necessarily stop when the main method returns. If another thread has been created, and that thread has not been marked as a daemon thread, then the application will continue to run after the main thread has ended. In this example, that means that the application will keep running until child thread ends. That should happen when tt.stop is set to true.

But that is actually not strictly true. In fact, the child thread will stop after it has observed stop with the value true. Will that happen? Maybe yes, maybe no.

The Java Language Specification guarantees that memory reads and writes made in a thread are visible to that thread, as per the order of the statements in the source code. However, in general, this is NOT guaranteed when one thread writes and another thread (subsequently) reads. To get guaranteed visibility, there needs to be a chain of happens-before relations between a write and a subsequent read. In the example above, there is no such chain for the update to the stop flag, and therefore it is not guaranteed that the child thread will see stop change to true.

(Note to authors: There should be a separate Topic on the Java Memory Model to go into the deep technical details.)

How do we fix the problem?

In this case, there are two simple ways to ensure that the stop update is visible:

1. Declare stop to be volatile; i.e.

```
private volatile boolean stop = false;
```

For a volatile variable, the JLS specifies that there is a happens-before relation between a write by one thread and a later read by a second thread.

2. Use a mutex to synchronize as follows:

```
public class ThreadTest implements Runnable {
```

```

private boolean stop = false;

public void run() {
    long counter = 0;
    while (true) {
synchronize (this) {
        if (stop) {
            break;
        }
    }
    counter = counter + 1;
    System.out.println("Counted " + counter);
}
}

public static void main(String[] args) {
    ThreadTest tt = new ThreadTest();
    new Thread(tt).start(); // 创建并启动子线程
    Thread.sleep(1000);
synchronize (tt) {
    tt.stop = true; // 告诉子线程停止。
}
}
}

```

除了确保互斥外，JLS还规定一个线程释放互斥锁与第二个线程获得同一互斥锁之间存在happens-before关系。

但赋值不是原子操作吗？

是的，确实是！

然而，这一事实并不意味着更新的效果会同时对所有线程可见。只有一条合适的happens-before关系链才能保证这一点。

他们为什么要这么做？

第一次进行Java多线程编程的程序员会发现内存模型很有挑战性。

程序的行为往往不符合直觉，因为人们自然期望写操作是统一可见的。那么，为什么Java设计者要这样设计内存模型呢？

这实际上是性能和易用性（对程序员而言）之间的折衷。

现代计算机架构由多个处理器（核心）组成，每个核心有独立的寄存器集。主内存可以被所有处理器或处理器组访问。现代计算机硬件的另一个特点是访问寄存器的速度通常比访问主内存快几个数量级。随着核心数量的增加，很容易看出读写主内存可能成为系统的主要性能瓶颈。

为了解决这一不匹配问题，处理器核心和主内存之间实现了一级或多级内存缓存。每个核心通过其缓存访问内存单元。通常，只有在缓存未命中时才会读取主内存，只有在缓存行需要刷新时才会写入主内存。对于每个核心的工作集内存位置都能装入其缓存的应用，核心速度就不再受限于主内存的速度或带宽。

但当多个核心同时读取和写入共享变量时，这就带来了一个新问题。变量的最新版本可能存在于某个核心的缓存中。除非该核心将缓存行刷新到主内存，并且其他核心

```

private boolean stop = false;

public void run() {
    long counter = 0;
    while (true) {
synchronize (this) {
        if (stop) {
            break;
        }
    }
    counter = counter + 1;
    System.out.println("Counted " + counter);
}
}

public static void main(String[] args) {
    ThreadTest tt = new ThreadTest();
    new Thread(tt).start(); // Create and start child thread
    Thread.sleep(1000);
synchronize (tt) {
    tt.stop = true; // Tell child thread to stop.
}
}
}

```

In addition to ensuring that there is mutual exclusion, the JLS specifies that there is a *happens-before* relation between the releasing a mutex in one thread and gaining the same mutex in a second thread.

But isn't assignment atomic?

Yes it is!

However, that fact does not mean that the effects of update will be visible simultaneously to all threads. Only a proper chain of *happens-before* relations will guarantee that.

Why did they do this?

Programmers doing multi-threaded programming in Java for the first time find the Memory Model is challenging. Programs behave in an unintuitive way because the natural expectation is that writes are visible uniformly. So why the Java designers design the Memory Model this way.

It actually comes down to a compromise between performance and ease of use (for the programmer).

A modern computer architecture consists of multiple processors (cores) with individual register sets. Main memory is accessible either to all processors or to groups of processors. Another property of modern computer hardware is that access to registers is typically orders of magnitude faster to access than access to main memory. As the number of cores scales up, it is easy to see that reading and writing to main memory can become a system's main performance bottleneck.

This mismatch is addressed by implementing one or more levels of memory caching between the processor cores and main memory. Each core access memory cells via its cache. Normally, a main memory read only happens when there is a cache miss, and a main memory write only happens when a cache line needs to be flushed. For an application where each core's working set of memory locations will fit into its cache, the core speed is no longer limited by main memory speed / bandwidth.

But that gives us a new problem when multiple cores are reading and writing shared variables. The latest version of a variable may sit in one core's cache. Unless the that core flushes the cache line to main memory, AND other cores

使它们缓存的旧版本失效，其中一些可能会看到变量的过时版本。但如果每次缓存写入时都将缓存刷新到内存（“以防”另一个核心进行了读取），那将不必要地消耗主内存带宽。

硬件指令集层面采用的标准解决方案是提供缓存失效和缓存写直达的指令，并由编译器决定何时使用它们。

回到Java。内存模型的设计使得Java编译器不必在不真正需要的情况下发出缓存失效和写直达指令。假设程序员会使用适当的同步机制（例如原始互斥锁、volatile、高级并发类等）来表明需要内存可见性。在没有happens-before关系的情况下，Java编译器可以自由地假设不需要任何缓存操作（或类似操作）。

这对多线程应用程序有显著的性能优势，但缺点是编写正确的多线程应用程序并非易事。程序员确实需要理解自己在做什么。

为什么我无法重现这个问题？

这类问题难以重现有多种原因：

1. 如上所述，不正确处理内存可见性问题的后果通常是
你的编译应用程序未正确处理内存缓存。然而，正如我们上面提到的，内存缓存通常会被刷新。
2. 当你更换硬件平台时，内存缓存的特性可能会发生变化。如果你的应用程序没有正确同步，这可能导致不同的行为。
3. 您可能正在观察到偶然的同步效应。例如，如果您添加traceprints，它们通常在I/O流的幕后会发生一些同步，导致缓存刷新。
因此，添加traceprints经常会导致应用程序表现不同。
4. 在调试器下运行应用程序会导致JIT编译器以不同方式编译它。
断点和单步执行会加剧这种情况。这些效应通常会改变应用程序的行为。

这些情况使得由于同步不足引起的错误特别难以解决。

第133.5节：陷阱 - 线程创建相对昂贵

考虑以下两个微基准测试：

第一个基准测试仅创建、启动并加入线程。线程的Runnable不执行任何工作。

```
public class ThreadTest {  
    public static void main(String[] args) throws Exception {  
        while (true) {  
            long start = System.nanoTime();  
            for (int i = 0; i < 100_000; i++) {  
                Thread t = new Thread(new Runnable() {  
                    public void run() {}  
                });  
                t.start();  
                t.join();  
            }  
            long end = System.nanoTime();  
        }  
    }  
}
```

invalidate their cached copy of older versions, some of them are liable to see stale versions of the variable. But if the caches were flushed to memory each time there is a cache write ("just in case" there was a read by another core) that would consume main memory bandwidth unnecessarily.

The standard solution used at the hardware instruction set level is to provide instructions for cache invalidation and a cache write-through, and leave it to the compiler to decide when to use them.

Returning to Java. the Memory Model is designed so that the Java compilers are not required to issue cache invalidation and write-through instructions where they are not really needed. The assumption is that the programmer will use an appropriate synchronization mechanism (e.g. primitive mutexes, **volatile**, higher-level concurrency classes and so on) to indicate that it needs memory visibility. In the absence of a *happens-before* relation, the Java compilers are free to *assume* that no cache operations (or similar) are required.

This has significant performance advantages for multi-threaded applications, but the downside is that writing correct multi-threaded applications is not a simple matter. The programmer *does* have to understand what he or she is doing.

Why can't I reproduce this?

There are a number of reasons why problems like this are difficult to reproduce:

1. As explained above, the consequence of not dealing with memory visibility issues problems properly is typically that your compiled application does not handle the memory caches correctly. However, as we alluded to above, memory caches often get flushed anyway.
2. When you change the hardware platform, the characteristics of the memory caches may change. This can lead to different behavior if your application does not synchronize correctly.
3. You may be observing the effects of *serendipitous* synchronization. For example, if you add traceprints, there is typically some synchronization happening behind the scenes in the I/O streams that causes cache flushes. So adding traceprints often causes the application to behave differently.
4. Running an application under a debugger causes it to be compiled differently by the JIT compiler. Breakpoints and single stepping exacerbate this. These effects will often change the way an application behaves.

These things make bugs that are due to inadequate synchronization particularly difficult to solve.

Section 133.5: Pitfall - Thread creation is relatively expensive

Consider these two micro-benchmarks:

The first benchmark simply creates, starts and joins threads. The thread's **Runnable** does no work.

```
public class ThreadTest {  
    public static void main(String[] args) throws Exception {  
        while (true) {  
            long start = System.nanoTime();  
            for (int i = 0; i < 100_000; i++) {  
                Thread t = new Thread(new Runnable() {  
                    public void run() {}  
                });  
                t.start();  
                t.join();  
            }  
            long end = System.nanoTime();  
        }  
    }  
}
```

```

        System.out.println((end - start) / 100_000.0);
    }
}

$ java ThreadTest
34627.91355
33596.66021
33661.19084
33699.44895
33603.097
33759.3928
33671.5719
33619.46809
33679.92508
33500.32862
33409.70188
33475.70541
33925.87848
33672.89529
^C

```

在一台运行64位Java 8 u101的典型现代Linux电脑上，该基准测试显示创建、启动和加入线程的平均时间在33.6到33.9微秒之间。

第二个基准测试与第一个等效，但使用ExecutorService提交任务，并使用Future与任务结束进行同步。

```

import java.util.concurrent.*;

public class ExecutorTest {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Future<?> future = exec.submit(new Runnable() {
                    public void run() {
                });
            }
            future.get();
        }
        long end = System.nanoTime();
        System.out.println((end - start) / 100_000.0);
    }
}

```

```

$ java ExecutorTest
6714.66053
5418.24901
5571.65213
5307.83651
5294.44132
5370.69978
5291.83493
5386.23932
5384.06842
5293.14126
5445.17405
5389.70685

```

```

        System.out.println((end - start) / 100_000.0);
    }
}

$ java ThreadTest
34627.91355
33596.66021
33661.19084
33699.44895
33603.097
33759.3928
33671.5719
33619.46809
33679.92508
33500.32862
33409.70188
33475.70541
33925.87848
33672.89529
^C

```

On a typical modern PC running Linux with 64bit Java 8 u101, this benchmark shows an average time taken to create, start and join thread of between 33.6 and 33.9 microseconds.

The second benchmark does the equivalent to the first but using an ExecutorService to submit tasks and a Future to rendezvous with the end of the task.

```

import java.util.concurrent.*;

public class ExecutorTest {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        while (true) {
            long start = System.nanoTime();
            for (int i = 0; i < 100_000; i++) {
                Future<?> future = exec.submit(new Runnable() {
                    public void run() {
                });
            }
            future.get();
        }
        long end = System.nanoTime();
        System.out.println((end - start) / 100_000.0);
    }
}

```

```

$ java ExecutorTest
6714.66053
5418.24901
5571.65213
5307.83651
5294.44132
5370.69978
5291.83493
5386.23932
5384.06842
5293.14126
5445.17405
5389.70685

```

如您所见，平均时间介于5.3到5.6微妙之间。

虽然实际时间会受到多种因素的影响，但这两个结果之间的差异是显著的。使用线程池回收线程明显比创建新线程更快。

As you can see, the averages are between 5.3 and 5.6 microseconds.

While the actual times will depend on a variety of factors, the difference between these two results is significant. It is clearly faster to use a thread pool to recycle threads than it is to create new threads.

第134章：Java陷阱——空值和空指针异常

第134.1节：陷阱——“修复”意外的空值

在StackOverflow上，我们经常在回答中看到这样的代码：

```
public String joinStrings(String a, String b) {  
    if (a == null) {  
        a = "";  
    }  
    if (b == null) {  
        b = "";  
    }  
    return a + " : " + b;  
}
```

通常，伴随着一种断言，即“最佳实践”是这样测试null以避免NullPointerException。

这是最佳实践吗？简而言之：不是。

在我们说在joinStrings中这样做是否是个好主意之前，有一些潜在的假设需要被质疑：

“a”或“b”为null意味着什么？

一个String值可以是零个或多个字符，所以我们已经有了一种表示空字符串的方法。null是否意味着与""不同的东西？如果不是，那么用两种方式表示空字符串就是有问题的。

空值是来自未初始化的变量吗？

一个null可能来自未初始化的字段，或未初始化的数组元素。该值可能是设计上未初始化，或是意外未初始化。如果是意外，则这是一个错误。

这个null表示“未知”还是“缺失值”？

有时一个null可以有真实的含义；例如变量的真实值未知、不可用或“可选”。在Java 8中，Optional类提供了更好的表达方式。

如果这是一个错误（或设计缺陷），我们是否应该“修正”它？

对代码的一种解释是，我们通过使用空字符串来“修正”意外的null。这个策略正确吗？是否更好让NullPointerException发生，然后在调用栈上层捕获该异常并将其记录为错误？

“修正”问题的弊端是它可能掩盖问题，或使问题更难诊断。

这样做效率高吗/对代码质量有益吗？

如果始终采用“修正”方法，代码中将包含大量“防御性”的null检测。这会使代码变长且难以阅读。此外，所有这些检测和“修正”可能会影响应用程序的性能。

总结

Chapter 134: Java Pitfalls - Nulls and NullPointerException

Section 134.1: Pitfall - "Making good" unexpected nulls

On StackOverflow, we often see code like this in Answers:

```
public String joinStrings(String a, String b) {  
    if (a == null) {  
        a = "";  
    }  
    if (b == null) {  
        b = "";  
    }  
    return a + " : " + b;  
}
```

Often, this is accompanied with an assertion that is "best practice" to test for `null` like this to avoid `NullPointerException`.

Is it best practice? In short: No.

There are some underlying assumptions that need to be questioned before we can say if it is a good idea to do this in our `joinStrings`:

What does it mean for "a" or "b" to be null?

A `String` value can be zero or more characters, so we already have a way of representing an empty string. Does `null` mean something different to ""? If no, then it is problematic to have two ways to represent an empty string.

Did the null come from an uninitialized variable?

A `null` can come from an uninitialized field, or an uninitialized array element. The value could be uninitialized by design, or by accident. If it was by accident then this is a bug.

Does the null represent a "don't know" or "missing value"?

Sometimes a `null` can have a genuine meaning; e.g. that the real value of a variable is unknown or unavailable or "optional". In Java 8, the `Optional` class provides a better way of expressing that.

If this is a bug (or a design error) should we "make good"?

One interpretation of the code is that we are "making good" an unexpected `null` by using an empty string in its place. Is the correct strategy? Would it be better to let the `NullPointerException` happen, and then catch the exception further up the stack and log it as a bug?

The problem with "making good" is that it is liable to either hide the problem, or make it harder to diagnose.

Is this efficient / good for code quality?

If the "make good" approach is used consistently, your code is going to contain a lot of "defensive" null tests. This is going to make it longer and harder to read. Furthermore, all of this testing and "making good" is liable to impact on the performance of your application.

In summary

如果 `null` 是一个有意义的值，那么测试 `null` 情况是正确的方法。推论是，如果 `null` 值是有意义的，那么这应当在任何接受 `null`

值或返回它的方法的 javadoc 中明确说明。

否则，更好的做法是将意外的 `null` 视为编程错误，并让 `NullPointerException` 发生，以便开发者知道代码中存在问题。

第134.2节：陷阱——使用 `null` 表示空数组或集合

有些程序员认为用 `null` 表示空数组或集合可以节省空间。虽然确实可以节省少量空间，但另一方面，这会使代码更复杂且更脆弱。比较下面两个用于求数组和的方法版本：

第一个版本是你通常编写该方法的方式：

```
/**  
 * 对整数数组中的值求和。  
 * @arg values 要求和的数组  
 * @return 和  
 */  
  
public int sum(int[] values) {  
    int sum = 0;  
    for (int value : values) {  
        sum += value;  
    }  
    return sum;  
}
```

第二个版本是当你习惯使用 `null` 来表示一个空数组时，如何编写该方法。

```
/**  
 * 对整数数组中的值求和。  
 * @arg values 要求和的数组，或为 null。  
 * @return 和，如果数组为 null 则返回零。  
 */  
  
public int sum(int[] values) {  
    int sum = 0;  
    if (values != null) {  
        for (int value : values) {  
            sum += value;  
        }  
    }  
    return sum;  
}
```

如你所见，代码稍微复杂一些。这直接归因于决定以这种方式使用 `null`。

现在考虑如果这个可能为 `null` 的数组在很多地方被使用。在每个使用它的地方，你都需要考虑是否需要测试 `null`。如果你漏掉了必须存在的 `null` 测试，就有可能导致 `NullPointerException`。因此，以这种方式使用 `null` 的策略会使你的应用程序更脆弱；即更容易受到程序员错误后果的影响。

这里的教训是，当你想表达空时，应该使用空数组和空列表。

If `null` is a meaningful value, then testing for the `null` case is the correct approach. The corollary is that if a `null` value is meaningful, then this should be clearly documented in the javadocs of any methods that accept the `null` value or return it.

Otherwise, it is a better idea to treat an unexpected `null` as a programming error, and let the `NullPointerException` happen so that the developer gets to know there is a problem in the code.

Section 134.2: Pitfall - Using `null` to represent an empty array or collection

Some programmers think that it is a good idea to save space by using a `null` to represent an empty array or collection. While it is true that you can save a small amount of space, the flipside is that it makes your code more complicated, and more fragile. Compare these two versions of a method for summing an array:

The first version is how you would normally code the method:

```
/**  
 * Sum the values in an array of integers.  
 * @arg values the array to be summed  
 * @return the sum  
 */  
  
public int sum(int[] values) {  
    int sum = 0;  
    for (int value : values) {  
        sum += value;  
    }  
    return sum;  
}
```

The second version is how you need to code the method if you are in the habit of using `null` to represent an empty array.

```
/**  
 * Sum the values in an array of integers.  
 * @arg values the array to be summed, or null.  
 * @return the sum, or zero if the array is null.  
 */  
  
public int sum(int[] values) {  
    int sum = 0;  
    if (values != null) {  
        for (int value : values) {  
            sum += value;  
        }  
    }  
    return sum;  
}
```

As you can see, the code is a bit more complicated. This is directly attributable to the decision to use `null` in this way.

Now consider if this array that might be a `null` is used in lots of places. At each place where you use it, you need to consider whether you need to test for `null`. If you miss a `null` test that needs to be there, you risk a `NullPointerException`. Hence, the strategy of using `null` in this way leads to your application being more fragile; i.e. more vulnerable to the consequences of programmer errors.

The lesson here is to use empty arrays and empty lists when that is what you mean.

```
int[] values = new int[0]; // 始终为空
List<Integer> list = new ArrayList(); // 初始为空
List<Integer> list = Collections.emptyList(); // 始终为空
```

空间开销很小，如果这是值得做的事情，还有其他方法可以最小化它。

第134.3节：陷阱——关闭I/O流时未检查其是否已初始化

为了防止内存泄漏，不应忘记关闭已完成任务的输入流或输出流。

这通常通过一个try-catch-finally语句来完成，但不包含catch部分：

```
void writeNullBytesToFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        out.close();
    }
}
```

虽然上述代码看起来无害，但它存在一个缺陷，可能导致调试变得不可能。如果初始化out的那一行（`out = new FileOutputStream(filename)`）抛出异常，那么执行`out.close()`时，`out`将是null，导致严重的NullPointerException！

为防止这种情况，只需在尝试关闭流之前确保流不是null即可。

```
void writeNullBytesToFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        if (out != null)
            out.close();
    }
}
```

一个更好的方法是使用try-with-resources，因为它会自动关闭流，且抛出空指针异常（NPE）的概率为0，无需使用finally块。

```
void writeNullBytesToFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

第134.4节：陷阱——返回null而不是抛出异常

一些Java程序员普遍不喜欢抛出或传播异常。这导致了如下代码：

```
int[] values = new int[0]; // always empty
List<Integer> list = new ArrayList(); // initially empty
List<Integer> list = Collections.emptyList(); // always empty
```

The space overhead is small, and there are other ways to minimize it if this is a worthwhile thing to do.

Section 134.3: Pitfall - Not checking if an I/O stream isn't even initialized when closing it

To prevent memory leaks, one should not forget to close an input stream or an output stream whose job is done. This is usually done with a **try-catch-finally** statement without the **catch** part:

```
void writeNullBytesToFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        out.close();
    }
}
```

While the above code might look innocent, it has a flaw that can make debugging impossible. If the line where `out` is initialized (`out = new FileOutputStream(filename)`) throws an exception, then `out` will be `null` when `out.close()` is executed, resulting in a nasty `NullPointerException`!

To prevent this, simply make sure the stream isn't `null` before trying to close it.

```
void writeNullBytesToFile(int count, String filename) throws IOException {
    FileOutputStream out = null;
    try {
        out = new FileOutputStream(filename);
        for(; count > 0; count--)
            out.write(0);
    } finally {
        if (out != null)
            out.close();
    }
}
```

An even better approach is to **try-with-resources**, since it'll automatically close the stream with a probability of 0 to 1.

```
void writeNullBytesToFile(int count, String filename) throws IOException {
    try (FileOutputStream out = new FileOutputStream(filename)) {
        for(; count > 0; count--)
            out.write(0);
    }
}
```

Section 134.4: Pitfall - Returning null instead of throwing an exception

Some Java programmers have a general aversion to throwing or propagating exceptions. This leads to code like the following:

```

public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("打开失败: " + ex.getMessage());
        return null;
    }
}

```

那么，这样做的问题是什么？

问题在于，getReader 返回了一个 null 作为特殊值，表示无法打开 Reader。现在返回的值需要在使用前进行 null 检测。如果省略了检测，结果将会是 NullPointerException。

这里实际上有三个问题：

1. IOException 异常被过早捕获了。
2. 这段代码的结构意味着存在资源泄漏的风险。
3. 使用了 null 然后返回，因为没有可返回的“真正的”Reader。

事实上，假设异常确实需要像这样早期捕获，那么除了返回 null 之外，还有几个替代方案：

1. 可以实现一个 NullReader 类；例如，其 API 操作表现得好像读取器已经处于“文件末尾”位置。
2. 在 Java 8 中，可以将 getReader 声明为返回一个 Optional<Reader>。

第134.5节：陷阱 - 不必要地使用原始包装类可能导致 NullPointerException

有时，Java 新手程序员会将原始类型和包装类混用。这可能导致问题。考虑以下示例：

```

public class MyRecord {
    public int a, b;
    public Integer c, d;
}

...
MyRecord record = new MyRecord();
record.a = 1;          // 正确
record.b = record.b + 1; // 正确
record.c = 1;          // 正确
record.d = record.d + 1; // 抛出 NullPointerException

```

我们的 MyRecord 类依赖默认初始化来初始化其字段的值。因此，当我们 new 一个记录时，字段 a 和 b 会被设置为零，而字段 c 和 d 会被设置为 null。

当我们尝试使用默认初始化的字段时，会发现 int 类型的字段始终有效，但 Integer 类型的字段在某些情况下有效，在其他情况下无效。具体来说，在失败的情况下（使用 d）中，右侧表达式尝试对 null 引用进行拆箱，这就是导致抛出 NullPointerException 的原因。

对此有几种看法：

```

public Reader getReader(String pathname) {
    try {
        return new BufferedReader(FileReader(pathname));
    } catch (IOException ex) {
        System.out.println("Open failed: " + ex.getMessage());
        return null;
    }
}

```

So what is the problem with that?

The problem is that the getReader is returning a null as a special value to indicate that the Reader could not be opened. Now the returned value needs to be tested to see if it is null before it is used. If the test is left out, the result will be a NullPointerException.

There are actually three problems here:

1. The IOException was caught too soon.
2. The structure of this code means that there is a risk of leaking a resource.
3. A null was used then returned because no "real" Reader was available to return.

In fact, assuming that the exception did need to be caught early like this, there were a couple of alternatives to returning null:

1. It would be possible to implement a NullReader class; e.g. one where API's operations behaves as if the reader was already at the "end of file" position.
2. With Java 8, it would be possible to declare getReader as returning an Optional<Reader>.

Section 134.5: Pitfall - Unnecessary use of Primitive Wrappers can lead to NullPointerExceptions

Sometimes, programmers who are new Java will use primitive types and wrappers interchangeably. This can lead to problems. Consider this example:

```

public class MyRecord {
    public int a, b;
    public Integer c, d;
}

...
MyRecord record = new MyRecord();
record.a = 1;          // OK
record.b = record.b + 1; // OK
record.c = 1;          // OK
record.d = record.d + 1; // throws a NullPointerException

```

Our MyRecord class1 relies on default initialization to initialize the values on its fields. Thus, when we new a record, the a and b fields will be set to zero, and the c and d fields will be set to null.

When we try to use the default initialized fields, we see that the int fields works all of the time, but the Integer fields work in some cases and not others. Specifically, in the case that fails (with d), what happens is that the expression on the right-hand side attempts to unbox a null reference, and that is what causes the NullPointerException to be thrown.

There are a couple of ways to look at this:

- 如果字段 c 和 d 需要是基本类型的包装类，那么我们不应该依赖默认初始化，要么应该检测 null。除非字段在 null 状态下有明确含义，否则前者是正确的做法。

- 如果字段不需要是基本类型的包装类，那么将它们设为包装类就是错误的。除了这个问题，包装类相较于基本类型还有额外的开销。

这里的教训是，除非确实需要，否则不要使用基本类型的包装类。

1 - 这个类不是良好编码实践的示例。例如，一个设计良好的类不会有公共字段。但这不是本例的重点。

第134.6节：陷阱——使用“尤达式”来避免空指针异常

StackOverflow上发布的许多示例代码都包含如下片段：

```
if ("A".equals(someString)) {  
    // 执行某些操作  
}
```

这种写法确实“防止”或“避免”了当someString为null时可能出现的NullPointerException。此外，有人认为

```
"A".equals(someString)
```

比以下写法更好：

```
someString != null && someString.equals("A")
```

(它更简洁，在某些情况下可能更高效。然而，正如我们下面所论述的，简洁性也可能是一个负面因素。)

然而，真正的陷阱是习惯性地使用尤达式测试来避免NullPointerExceptions。

当你写"A".equals(someString)时，实际上是在“补救”someString恰好为null的情况。但正如另一个例子（陷阱——“补救”意外的null）所说明的，“补救”null值可能因多种原因而有害。

这意味着约达条件并不是“最佳实践”¹。除非预期出现null，否则最好让NullPointerException发生，这样你就可以得到单元测试失败（或错误报告）。这使你能够找到并修复导致意外/不希望出现的null的错误。

只有在null是预期的情况下才应使用Yoda条件，因为你测试的对象来自一个文档化为返回null的API。可以说，使用一些不那么优雅的表达测试方式可能更好，因为这有助于让审查你代码的人更容易注意到null测试。

1 - 根据维基百科：“最佳编码实践是一套软件开发社区随着时间积累的非正式规则，能够帮助提高软件质量。”使用Yoda表示法并不能达到这一点。在很多情况下，它反而使代码更糟。

- If the fields c and d need to be primitive wrappers, then either we should not be relying on default initialization, or we should be testing for `null`. For former is the correct approach *unless* there is a definite meaning for the fields in the `null` state.

- If the fields don't need to be primitive wrappers, then it is a mistake to make them primitive wrappers. In addition to this problem, the primitive wrappers have extra overheads relative to primitive types.

The lesson here is to not use primitive wrapper types unless you really need to.

1 - This class is not an example of good coding practice. For instance, a well-designed class would not have public fields. However, that is not the point of this example.

Section 134.6: Pitfall - Using "Yoda notation" to avoid NullPointerException

A lot of example code posted on StackOverflow includes snippets like this:

```
if ("A".equals(someString)) {  
    // do something  
}
```

This does "prevent" or "avoid" a possible NullPointerException in the case that someString is `null`. Furthermore, it is arguable that

```
"A".equals(someString)
```

is better than:

```
someString != null && someString.equals("A")
```

(It is more concise, and in some circumstances it might be more efficient. However, as we argue below, conciseness could be a negative.)

However, the real pitfall is using the Yoda test **to avoid NullPointerExceptions** as a matter of habit.

When you write `"A".equals(someString)` you are actually "making good" the case where someString happens to be `null`. But as another example (Pitfall - "Making good" unexpected nulls) explains, "making good" `null` values can be harmful for a variety of reasons.

This means that Yoda conditions are not "best practice"¹. Unless the `null` is expected, it is better to let the NullPointerException happen so that you can get a unit test failure (or a bug report). That allows you to find and fix the bug that caused the unexpected / unwanted `null` to appear.

Yoda conditions should only be used in cases where the `null` is *expected* because the object you are testing has come from an API that is *documented* as returning a `null`. And arguably, it could be better to use one of the less pretty ways expressing the test because that helps to highlight the `null` test to someone who is reviewing your code.

1 - According to [Wikipedia](#): "Best coding practices are a set of informal rules that the software development community has learned over time which can help improve the quality of software.". Using Yoda notation does not achieve this. In a lot of situations, it makes the code worse.

第135章：Java陷阱 - 性能问题

本主题描述了若干“陷阱”（即初学Java程序员常犯的错误），这些错误与Java应用程序性能相关。

第135.1节：陷阱 - 循环中的字符串连接不具备扩展性

以下代码作为示例：

```
public String joinWords(List<String> words) {  
    String message = "";  
    for (String word : words) {  
        message = message + " " + word;  
    }  
    return message;  
}
```

遗憾的是，如果words列表很长，这段代码效率很低。问题的根源在于这条语句：

```
message = message + " " + word;
```

在每次循环迭代中，这条语句都会创建一个新的message字符串，包含原始message字符串的所有字符副本，并在其后附加额外字符。这会生成大量临时字符串，并进行大量复制。

当我们分析joinWords时，假设有N个单词，平均长度为M，我们发现创建了O(N)个临时字符串，并且在此过程中会复制O(M.N²)个字符。N²这一项尤其令人担忧。

针对这类问题，推荐的方法是使用StringBuilder代替字符串连接，具体如下：

```
public String joinWords2(List<String> words) {  
    StringBuilder message = new StringBuilder();  
    for (String word : words) {  
        message.append(" ").append(word);  
    }  
    return message.toString();  
}
```

对joinWords2的分析需要考虑“扩展”StringBuilder底层字符数组的开销。然而，事实证明新创建的对象数量为O(logN)，复制的字符数量为O(M.N)。后者包括在最终调用toString()时复制的字符。

（通过一开始使用正确容量创建StringBuilder，可能还能进一步调整。）
然而，整体复杂度保持不变。）

回到原始的joinWords方法，结果是关键语句会被典型的Java编译器优化成类似这样的代码：

```
StringBuilder tmp = new StringBuilder();
```

Chapter 135: Java Pitfalls - Performance Issues

This topic describes a number of "pitfalls" (i.e. mistakes that novice java programmers make) that relate to Java application performance.

Section 135.1: Pitfall - String concatenation in a loop does not scale

Consider the following code as an illustration:

```
public String joinWords(List<String> words) {  
    String message = "";  
    for (String word : words) {  
        message = message + " " + word;  
    }  
    return message;  
}
```

Unfortunate this code is inefficient if the words list is long. The root of the problem is this statement:

```
message = message + " " + word;
```

For each loop iteration, this statement creates a new message string containing a copy of all characters in the original message string with extra characters appended to it. This generates a lot of temporary strings, and does a lot of copying.

When we analyse joinWords, assuming that there are N words with an average length of M, we find that O(N) temporary strings are created and O(M.N²) characters will be copied in the process. The N² component is particularly troubling.

The recommended approach for this kind of problem1 is to use a StringBuilder instead of string concatenation as follows:

```
public String joinWords2(List<String> words) {  
    StringBuilder message = new StringBuilder();  
    for (String word : words) {  
        message.append(" ").append(word);  
    }  
    return message.toString();  
}
```

The analysis of joinWords2 needs to take account of the overheads of "growing" the StringBuilder backing array that holds the builder's characters. However, it turns out that the number of new objects created is O(logN) and that the number of characters copied is O(M.N) characters. The latter includes characters copied in the final toString() call.

(It may be possible to tune this further, by creating the StringBuilder with the correct capacity to start with. However, the overall complexity remains the same.)

Returning to the original joinWords method, it turns out that the critical statement will be optimized by a typical Java compiler to something like this:

```
StringBuilder tmp = new StringBuilder();
```

```
tmp.append(message).append(" ").append(word);
message = tmp.toString();
```

然而, Java编译器不会像我们在

参考 :

- [“Java中循环内的字符串+'操作符慢吗？”](#)

1 - 在Java 8及以后版本, 可以使用Joiner类来解决这个特定问题。但这并不是这个示例真正想要讨论的内容。

第135.2节：陷阱——使用size()来判断集合是否为空效率低下

Java集合框架为所有Collection对象提供了两个相关的方法：

- `size()` 返回Collection中的条目数,
- `isEmpty()` 方法仅当Collection为空时返回true。

这两种方法都可以用来测试集合是否为空。例如：

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // 避免这样写
boolean isEmpty = strings.isEmpty(); // 推荐写法
```

虽然这些方法看起来相同, 但有些集合实现并不存储大小。对于这样的集合, `size()` 的实现需要每次调用时计算大小。例如：

- 一个简单的链表类 (但不是`java.util.LinkedList`) 可能需要遍历列表来计数元素。
- `ConcurrentHashMap` 类需要汇总映射中所有“段”的条目数。
- 一个惰性实现的集合可能需要将整个集合加载到内存中才能计数元素。

相比之下, `isEmpty()` 方法只需测试集合中是否至少有一个元素。这不需要计数元素。

虽然 `size() == 0` 并不总是比 `isEmpty()` 效率低, 但可以想象不到一个正确实现的 `isEmpty()` 会比 `size() == 0` 效率低。因此推荐使用 `isEmpty()`。

第135.3节：陷阱——为了使用 == 而对字符串进行驻留 (interning) 是个坏主意

当一些程序员看到这个建议时：

使用==来测试字符串是不正确的 (除非字符串已经被常量池化)

他们最初的反应是将字符串常量池化, 以便可以使用==。 (毕竟==比调用`String.equals(...)`要快, 不是吗。)

这是错误的方法, 从多个角度来看都是如此：

```
tmp.append(message).append(" ").append(word);
message = tmp.toString();
```

However, the Java compiler will not "hoist" the `StringBuilder` out of the loop, as we did by hand in the code for `joinWords2`.

Reference:

- [“Is Java's String '+' operator in a loop slow?”](#)

1 - In Java 8 and later, the Joiner class can be used to solve this particular problem. However, that is not what this example is *really supposed to be about*.

Section 135.2: Pitfall - Using size() to test if a collection is empty is inefficient

The Java Collections Framework provides two related methods for all `Collection` objects:

- `size()` returns the number of entries in a `Collection`, and
- `isEmpty()` method returns true if (and only if) the `Collection` is empty.

Both methods can be used to test for collection emptiness. For example:

```
Collection<String> strings = new ArrayList<>();
boolean isEmpty_wrong = strings.size() == 0; // Avoid this
boolean isEmpty = strings.isEmpty(); // Best
```

While these approaches look the same, some collection implementations do not store the size. For such a collection, the implementation of `size()` needs to calculate the size each time it is called. For instance:

- A simple linked list class (but not the `java.util.LinkedList`) might need to traverse the list to count the elements.
- The `ConcurrentHashMap` class needs to sum the entries in all of the map's "segments".
- A lazy implementation of a collection might need to realize the entire collection in memory in order to count the elements.

By contrast, an `isEmpty()` method only needs to test if there is *at least one* element in the collection. This does not entail counting the elements.

While `size() == 0` is not always less efficient than `isEmpty()`, it is inconceivable for a properly implemented `isEmpty()` to be less efficient than `size() == 0`. Hence `isEmpty()` is preferred.

Section 135.3: Pitfall - Interning strings so that you can use == is a bad idea

When some programmers see this advice:

"Testing strings using == is incorrect (unless the strings are interned)"

their initial reaction is to intern strings so that they can use ==. (After all == is faster than calling `String.equals(...)`, isn't it.)

This is the wrong approach, from a number of perspectives:

首先，只有当你确定你测试的所有String对象都已经被常量池化时，才能安全地使用`==`。Java语言规范保证你源代码中的字符串字面量会被常量池化。然而，除了`String.intern(String)`方法本身，标准的Java SE API并不保证返回的字符串是常量池化的。如果你遗漏了一个未被常量池化的String对象来源，你的应用程序将变得不可靠。这种不可靠性表现为假阴性，而不是异常，这会使问题更难被发现。

使用'intern()'的代价

在底层，常量池化通过维护一个包含先前常量池化的String对象的哈希表来实现。

使用某种弱引用机制，以防止驻留哈希表成为存储泄漏。

虽然哈希表是用本地代码实现的（不同于HashMap、HashTable等），但intern调用在CPU和内存使用方面仍然相对昂贵。

这种成本必须与使用`==`代替`equals`所节省的开销进行比较。实际上，除非每个驻留字符串与其他字符串比较“几次”，否则我们无法达到收支平衡。

（顺便说一下，少数情况下驻留是值得的，通常是为了减少应用程序的内存占用，这些应用中相同的字符串多次出现，并且这些字符串的生命周期较长。）

对垃圾回收的影响

除了上述直接的CPU和内存开销外，驻留字符串还会影响垃圾回收器的性能。

在Java 7之前的版本中，驻留字符串存放在“PermGen”空间，该空间的回收频率较低。

如果需要回收PermGen，通常会触发一次完整的垃圾回收。如果PermGen空间完全填满，即使常规堆空间有空闲，JVM也会崩溃。

在Java 7中，字符串池从“PermGen”移到了普通堆中。然而，哈希表仍然是一个长生命周期的数据结构，这会导致任何驻留字符串也具有长生命周期。（即使驻留字符串对象分配在Eden区，它们很可能在被回收前就被晋升。）因此，在所有情况下，驻留字符串都会相对于普通字符串延长其生命周期。这将增加JVM生命周期内的垃圾回收开销。

第二个问题是哈希表需要使用某种弱引用机制来防止字符串驻留导致内存泄漏。但这种机制会增加垃圾回收器的工作量。

这些垃圾回收开销难以量化，但毫无疑问它们确实存在。如果你使用`intern`大量使用时，可能会产生显著影响。

字符串池哈希表大小

根据该来源，从Java 6开始，字符串池被实现为固定大小的哈希表，使用链表来处理哈希到同一桶的字符串。在Java 6的早期版本中，哈希表具有（硬编码的）固定大小。作为Java 6中期更新，添加了一个调优参数（`-XX:StringTableSize`）。随后在Java 7的中期更新中，池的默认大小从1009更改为60013。

关键是，如果你确实打算在代码中大量使用`intern`，建议选择一个哈希表大小可调的Java版本，并确保适当调优其大小。否则，随着池变大，`intern`的性能可能会下降。

作为潜在拒绝服务攻击向量的字符串驻留

Fragility

First of all, you can only safely use `==` if you know that *all* of the `String` objects you are testing have been interned. The JLS guarantees that String literals in your source code will have been interned. However, none of the standard Java SE APIs guarantee to return interned strings, apart from `String.intern(String)` itself. If you miss just one source of `String` objects that haven't been interned, your application will be unreliable. That unreliability will manifest itself as false negatives rather than exceptions which is liable to make it harder to detect.

Costs of using 'intern()'

Under the hood, interning works by maintaining a hash table that contains previously interned `String` objects. Some kind of weak reference mechanism is used so that the interning hash table does not become a storage leak. While the hash table is implemented in native code (unlike `HashMap`, `HashTable` and so on), the `intern` calls are still relatively costly in terms of CPU and memory used.

This cost has to be compared with the saving of we are going to get by using `==` instead of `equals`. In fact, we are not going to break even unless each interned string is compared with other strings "a few" times.

(Aside: the few situations where interning is worthwhile tend to be about reducing the memory foot print of an application where the same strings recur many times, *and* those strings have a long lifetime.)

The impact on garbage collection

In addition to the direct CPU and memory costs described above, interned Strings impact on the garbage collector performance.

For versions of Java prior to Java 7, interned strings are held in the "PermGen" space which is collected infrequently. If PermGen needs to be collected, this (typically) triggers a full garbage collection. If the PermGen space fills completely, the JVM crashes, even if there was free space in the regular heap spaces.

In Java 7, the string pool was moved out of "PermGen" into the normal heap. However, the hash table is still going to be a long-lived data structure, which is going to cause any interned strings to be long-lived. (Even if the interned string objects were allocated in Eden space they would most likely be promoted before they were collected.)

Thus in all cases, interning a string is going to prolong its lifetime relative to an ordinary string. That will increase the garbage collection overheads over the lifetime of the JVM.

The second issue is that the hash table needs to use a weak reference mechanism of some kind to prevent string interning leaking memory. But such a mechanism is more work for the garbage collector.

These garbage collection overheads are difficult to quantify, but there is little doubt that they do exist. If you use `intern` extensively, they could be significant.

The string pool hashtable size

According to [this source](#), from Java 6 onwards, the string pool is implemented as fixed sized hash table with chains to deal with strings that hash to the same bucket. In early releases of Java 6, the hash table had a (hard-wired) constant size. A tuning parameter (`-XX:StringTableSize`) was added as a mid-life update to Java 6. Then in a mid-life update to Java 7, the default size of the pool was changed from 1009 to 60013.

The bottom line is that if you do intend to use `intern` intensively in your code, it is *advisable* to pick a version of Java where the hashtable size is tunable and make sure that you tune the size it appropriately. Otherwise, the performance of `intern` is liable to degrade as the pool gets larger.

Interning as a potential denial of service vector

字符串的哈希码算法是众所周知的。如果你对恶意用户或应用提供的字符串进行驻留，这可能被用作拒绝服务（DoS）攻击的一部分。如果恶意方安排所有提供的字符串具有相同的哈希码，这可能导致哈希表不平衡，intern的性能降至O(N)...其中N是发生冲突的字符串数量。

（有更简单/更有效的方法对服务发起DoS攻击。然而，如果DoS攻击的目标是破坏安全或规避第一线DoS防御，则可能使用此向量。）

第135.4节：陷阱 - 使用'new'创建原始类型包装器实例效率低下

Java语言允许你使用new来创建Integer、Boolean等实例，但这通常是一个不好的做法。更好的方法是使用自动装箱（Java 5及以后版本）或valueOf方法。

```
Integer i1 = new Integer(1);      // 不推荐
Integer i2 = 2;                  // 推荐（自动装箱）
Integer i3 = Integer.valueOf(3); // 可以
```

显式使用new Integer(int)是不好的原因是它会创建一个新对象（除非被JIT编译器优化掉）。相比之下，当使用自动装箱或显式调用valueOf时，Java运行时会尝试从缓存的预先存在的Integer对象中重用对象。每当运行时缓存命中时，就避免了创建新对象。这也节省了堆内存并减少了因对象频繁创建而导致的垃圾回收开销。

注意事项：

- 在最近的Java实现中，自动装箱是通过调用valueOf实现的，并且存在以下类型的缓存 Boolean、Byte、Short、Integer、Long和Character。
- 对于整型的缓存行为是由Java语言规范强制规定的。

第135.5节：陷阱 - 正则表达式的效率问题

正则表达式匹配是一个强大的工具（在Java及其他环境中），但它确实存在一些缺点。其中之一是正则表达式往往比较耗费资源。

Pattern和Matcher实例应该被重用

考虑以下示例：

```
/**
 * 测试列表中的所有字符串是否都由英文字母和数字组成。
 * @param strings 要检查的字符串列表
 * @return 当且仅当所有字符串都满足条件时返回 'true'
 * @throws NullPointerException 如果 'strings' 为 'null' 或包含 'null' 元素。
 */
public boolean allAlphanumeric(List<String> strings) {
    for (String s : strings) {
        if (!s.matches("[A-Za-z0-9]*")) {
            return false;
        }
    }
    return true;
}
```

这段代码是正确的，但效率不高。问题出在 matches(...) 调用上。实际上，s.matches("[A-

The hashCode algorithm for strings is well-known. If you intern strings supplied by malicious users or applications, this could be used as part of a denial of service (DoS) attack. If the malicious agent arranges that all of the strings it provides have the same hash code, this could lead to an unbalanced hash table and O(N) performance for intern ... where N is the number of collided strings.

(There are simpler / more effective ways to launch a DoS attack against a service. However, this vector could be used if the goal of the DoS attack is to break security, or to evade first-line DoS defences.)

Section 135.4: Pitfall - Using 'new' to create primitive wrapper instances is inefficient

The Java language allows you to use new to create instances Integer, Boolean and so on, but it is generally a bad idea. It is better to either use autoboxing (Java 5 and later) or the valueOf method.

```
Integer i1 = new Integer(1);      // BAD
Integer i2 = 2;                  // BEST (autoboxing)
Integer i3 = Integer.valueOf(3); // OK
```

The reason that using new Integer(int) explicitly is a bad idea is that it creates a new object (unless optimized out by JIT compiler). By contrast, when autoboxing or an explicit valueOf call are used, the Java runtime will try to reuse an Integer object from a cache of pre-existing objects. Each time the runtime has a cache "hit", it avoids creating an object. This also saves heap memory and reduces GC overheads caused by object churn.

Notes:

- In recent Java implementations, autoboxing is implemented by calling valueOf, and there are caches for Boolean, Byte, Short, Integer, Long and Character.
- The caching behavior for the integral types is mandated by the Java Language Specification.

Section 135.5: Pitfall - Efficiency concerns with regular expressions

Regular expression matching is a powerful tool (in Java, and in other contexts) but it does have some drawbacks. One of these is that regular expressions tends to be rather expensive.

Pattern and Matcher instances should be reused

Consider the following example:

```
/**
 * Test if all strings in a list consist of English letters and numbers.
 * @param strings the list to be checked
 * @return 'true' if all strings satisfy the criteria
 * @throws NullPointerException if 'strings' is 'null' or a 'null' element.
 */
public boolean allAlphanumeric(List<String> strings) {
    for (String s : strings) {
        if (!s.matches("[A-Za-z0-9]*")) {
            return false;
        }
    }
    return true;
}
```

This code is correct, but it is inefficient. The problem is in the matches(...) call. Under the hood, s.matches("[A-

Za-z0-9]*)" 等价于：

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

这又等价于

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

Pattern.compile("[A-Za-z0-9]*")调用会解析正则表达式，分析它，并构造一个Pattern对象，该对象保存正则表达式引擎将使用的数据结构。这是一个非平凡的计算。然后创建一个Matcher对象来包装 s参数。最后调用match()来执行实际的模式匹配。

问题是这些工作在每次循环迭代中都会重复进行。解决方案是将代码重构为如下：

```
private static Pattern ALPHA_NUMERIC = Pattern.compile("[A-Za-z0-9]*");

public boolean allAlphanumeric(List<String> strings) {
    Matcher matcher = ALPHA_NUMERIC.matcher("");
    for (String s : strings) {
        matcher.reset(s);
        if (!matcher.matches()) {
            return false;
        }
    }
    return true;
}
```

注意Pattern的javadoc中说明：

该类的实例是不可变的，且可安全用于多个并发线程。Matcher类的实例则不适合此类使用。

当应使用find()时，不要使用match()

假设你想测试字符串 s 是否包含三个或更多连续的数字。你可以用多种方式表达这一点包括：

```
if (s.matches(".*[0-9]{3}.*")) {
    System.out.println("matches");
}
```

或者

```
if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
    System.out.println("matches");
}
```

第一个更简洁，但效率可能较低。表面上看，第一个版本会尝试将整个字符串与模式匹配。此外，由于 ".*" 是一个“贪婪”模式，模式匹配器很可能会“急切”地推进到字符串末尾，然后回溯直到找到匹配。

相比之下，第二个版本会从左到右搜索，并且一旦找到连续的三个数字就会停止搜索。

Za-z0-9]*)" is equivalent to this:

```
Pattern.matches(s, "[A-Za-z0-9]*")
```

which is in turn equivalent to

```
Pattern.compile("[A-Za-z0-9]*").matcher(s).matches()
```

The Pattern.compile("[A-Za-z0-9]*") call parses the regular expression, analyze it, and construct a Pattern object that holds the data structure that will be used by the regex engine. This is a non-trivial computation. Then a Matcher object is created to wrap the s argument. Finally we call match() to do the actual pattern matching.

The problem is that this work is all repeated for each loop iteration. The solution is to restructure the code as follows:

```
private static Pattern ALPHA_NUMERIC = Pattern.compile("[A-Za-z0-9]*");

public boolean allAlphanumeric(List<String> strings) {
    Matcher matcher = ALPHA_NUMERIC.matcher("");
    for (String s : strings) {
        matcher.reset(s);
        if (!matcher.matches()) {
            return false;
        }
    }
    return true;
}
```

Note that the [javadoc](#) for Pattern states:

Instances of this class are immutable and are safe for use by multiple concurrent threads. Instances of the Matcher class are not safe for such use.

Don't use match() when you should use find()

Suppose you want to test if a string s contains three or more digits in a row. You can express this in various ways including:

```
if (s.matches(".*[0-9]{3}.*")) {
    System.out.println("matches");
}
```

or

```
if (Pattern.compile("[0-9]{3}").matcher(s).find()) {
    System.out.println("matches");
}
```

The first one is more concise, but it is also likely to be less efficient. On the face of it, the first version is going to try to match the entire string against the pattern. Furthermore, since ".*" is a "greedy" pattern, the pattern matcher is likely to advance "eagerly" try to the end of the string, and backtrack until it finds a match.

By contrast, the second version will search from left to right and will stop searching as soon as it finds the 3 digits in a row.

使用比正则表达式更高效的替代方案

正则表达式是一个强大的工具，但不应成为你唯一的工具。许多任务可以通过其他方式更高效地完成。例如：

```
Pattern.compile("ABC").matcher(s).find()
```

与以下代码功能相同：

```
s.contains("ABC")
```

只是后者效率高得多。（即使你能摊销编译正则表达式的成本。）

通常，非正则表达式形式更复杂。例如，`matches()`调用执行的测试比之前的`allAlphanumeric`方法可以重写为：

```
public boolean matches(String s) {  
    for (char c : s) {  
        if ((c >= 'A' && c <= 'Z') ||  
            (c >= 'a' && c <= 'z') ||  
            (c >= '0' && c <= '9')) {  
            return false;  
        }  
    }  
    return true;  
}
```

这比使用Matcher写的代码多，但速度也会快得多。

灾难性回溯

（这可能是所有正则表达式实现中的一个问题，但我们在那里提及它，因为它是Pattern使用中的一个陷阱。）

考虑这个（人为设计的）例子：

```
Pattern pat = Pattern.compile("(A+)+B");  
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAB").matches());  
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAC").matches());
```

第一次`println`调用将快速打印`true`。第二次将打印`false`。最终。实际上，如果你尝试上述代码，你会发现每次在C前添加一个A，所需时间都会翻倍。

这种行为是灾难性回溯的一个例子。实现正则表达式匹配的模式匹配引擎徒劳地尝试所有可能的方式来匹配该模式。

让我们看看`(A+)+B`实际上是什么意思。表面上看，它似乎表示“一个或多个A字符后跟一个B值”，但实际上它表示一个或多个组，每个组由一个或多个A字符组成。例如：

- 'AB'只有一种匹配方式：'(A)B'
- 'AAB'有两种匹配方式：'(AA)B'或'(A)(A)B'
- 'AAAB'有四种匹配方式：'(AAA)B'或'(AA)(A)B'或'(A)(AA)B'或'(A)(A)(A)B'
- 依此类推

换句话说，可能的匹配数量是 2^N 次方，其中N是A字符的数量。

Use more efficient alternatives to regular expressions

Regular expressions are a powerful tool, but they should not be your only tool. A lot of tasks can be done more efficiently in other ways. For example:

```
Pattern.compile("ABC").matcher(s).find()
```

does the same thing as:

```
s.contains("ABC")
```

except that the latter is a lot more efficient. (Even if you can amortize the cost of compiling the regular expression.)

Often, the non-regex form is more complicated. For example, the test performed by the `matches()` call the earlier `allAlphanumeric` method can be rewritten as:

```
public boolean matches(String s) {  
    for (char c : s) {  
        if ((c >= 'A' && c <= 'Z') ||  
            (c >= 'a' && c <= 'z') ||  
            (c >= '0' && c <= '9')) {  
            return false;  
        }  
    }  
    return true;  
}
```

Now that is more code than using a Matcher, but it is also going to be significantly faster.

Catastrophic Backtracking

(This is potentially a problem with all implementations of regular expressions, but we will mention it here because it is a pitfall for Pattern usage.)

Consider this (contrived) example:

```
Pattern pat = Pattern.compile("(A+)+B");  
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAAAAAAB").matches());  
System.out.println(pat.matcher("AAAAAAAAAAAAAAAAC").matches());
```

The first `println` call will quickly print `true`. The second one will print `false`. Eventually. Indeed, if you experiment with the code above, you will see that each time you add an A before the C, the time take will double.

This is behavior is an example of *catastrophic backtracking*. The pattern matching engine that implements the regex matching is fruitlessly trying all of the *possible* ways that the pattern *might* match.

Let us look at what `(A+)+B` actually means. Superficially, it seems to say "one or more A characters followed by a B value", but in reality it says one or more groups, each of which consists of one or more A characters. So, for example:

- 'AB' matches one way only: '(A)B'
- 'AAB' matches two ways: '(AA)B' or '(A)(A)B'
- 'AAAB' matches four ways: '(AAA)B' or '(AA)(A)B' or '(A)(AA)B' or '(A)(A)(A)B'
- and so on

In other words, the number of possible matches is 2^N where N is the number of A characters.

上述例子显然是人为设计的，但表现出这种性能特征的模式（即 $O(2^N)$ 或 $O(N^K)$ 对于较大的 K ）在使用考虑不周的正则表达式时经常出现。有一些标准的解决方法：

- 避免在重复模式中嵌套其他重复模式。
- 避免使用过多的重复模式。
- 根据需要使用非回溯重复。
- 不要使用正则表达式来处理复杂的解析任务。（应编写一个合适的解析器。）

最后，要警惕用户或API客户端可能提供具有病态特征的正则表达式字符串的情况。这可能导致意外或故意的“拒绝服务”。

参考资料：

- 正则表达式标签，特别是
<http://stackoverflow.com/documentation/regex/977/backtracking#t=201610010339131361163> 和
<http://stackoverflow.com/documentation/regex/4527/when-you-should-not-use-regular-expressions#t=201610010339593564913>
- “[正则表达式性能](#)” 作者：Jeff Atwood。
- “[如何用正则表达式杀死Java](#)” 作者：Andreas Haufler。

第135.6节：陷阱——对无缓冲流进行小规模读写效率低下

考虑以下代码将一个文件复制到另一个文件：

```
import java.io.*;

public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new FileInputStream(args[0]));
            OutputStream os = new FileOutputStream(args[1])) {
            int octet;
            while ((octet = is.read()) != -1) {
                os.write(octet);
            }
        }
    }
}
```

（我们故意省略了常规的参数检查、错误报告等，因为它们与本例的重点无关。）

如果你编译上述代码并用它复制一个大文件，你会注意到速度非常慢。事实上，它至少比标准操作系统的文件复制工具慢几个数量级。

（此处添加实际性能测量数据！）

上述示例在大文件情况下运行缓慢的主要原因是它在无缓冲的字节流上执行单字节读取和单字节写入。提高性能的简单方法是用缓冲流包装这些流。例如：

```
import java.io.*;

public class FileCopy {
```

The above example is clearly contrived, but patterns that exhibit this kind of performance characteristics (i.e. $O(2^N)$ or $O(N^K)$ for a large K) arise frequently when ill-considered regular expressions are used. There are some standard remedies:

- Avoid nesting repeating patterns within other repeating patterns.
- Avoid using too many repeating patterns.
- Use non-backtracking repetition as appropriate.
- Don't use regexes for complicated parsing tasks. (Write a proper parser instead.)

Finally, beware of situations where a user or an API client can supply a regex string with pathological characteristics. That can lead to accidental or deliberate "denial of service".

References:

- The Regular Expressions tag, particularly
<http://stackoverflow.com/documentation/regex/977/backtracking#t=201610010339131361163> and
<http://stackoverflow.com/documentation/regex/4527/when-you-should-not-use-regular-expressions#t=201610010339593564913>
- ["Regex Performance"](#) by Jeff Atwood.
- ["How to kill Java with a Regular Expression"](#) by Andreas Haufler.

Section 135.6: Pitfall - Small reads / writes on unbuffered streams are inefficient

Consider the following code to copy one file to another:

```
import java.io.*;

public class FileCopy {

    public static void main(String[] args) throws Exception {
        try (InputStream is = new FileInputStream(args[0]));
            OutputStream os = new FileOutputStream(args[1])) {
            int octet;
            while ((octet = is.read()) != -1) {
                os.write(octet);
            }
        }
    }
}
```

（We have deliberately omitted normal argument checking, error reporting and so on because they are not relevant to point of this example.）

If you compile the above code and use it to copy a huge file, you will notice that it is very slow. In fact, it will be at least a couple of orders of magnitude slower than the standard OS file copy utilities.

（Add actual performance measurements here!）

The primary reason that the example above is slow (in the large file case) is that it is performing one-byte reads and one-byte writes on unbuffered byte streams. The simple way to improve performance is to wrap the streams with buffered streams. For example:

```
import java.io.*;

public class FileCopy {
```

```

public static void main(String[] args) throws Exception {
    try (InputStream is = new BufferedInputStream(
        new FileInputStream(args[0]));
        OutputStream os = new BufferedOutputStream(
        new FileOutputStream(args[1]))) {
        int octet;
        while ((octet = is.read()) != -1) {
            os.write(octet);
        }
    }
}

```

这些小改动将根据不同的平台相关因素，至少提升数据复制速率几个数量级。缓冲流包装器使数据以更大的块进行读写。这两个实例都实现了作为字节数组的缓冲区。

- 对于 `is`, 数据每次从文件读取几千字节到缓冲区。当调用 `read()` 时, 通常会从缓冲区返回一个字节。只有当缓冲区被清空时, 才会从底层输入流读取数据。
- `os` 的行为类似。调用 `os.write(int)` 会将单个字节写入缓冲区。只有当缓冲区满了, 或者 `os` 被刷新或关闭时, 数据才会写入输出流。

字符流呢？

正如你应该知道的, Java I/O 提供了不同的 API 用于读取和写入二进制数据和文本数据。

- `InputStream` 和 `OutputStream` 是基于流的二进制 I/O 的基础 API
- `Reader` 和 `Writer` 是基于流的文本 I/O 的基础 API。

对于文本 I/O, `BufferedReader` 和 `BufferedWriter` 是 `BufferedInputStream` 和 `BufferedOutputStream` 的对应版本。

为什么缓冲流会产生如此大的差异？

缓冲流提升性能的真正原因与应用程序与操作系统的交互方式有关：

- Java 应用程序中的 Java 方法, 或 JVM 本地运行时库中的本地过程调用执行速度很快。它们通常只需几条机器指令, 性能影响极小。
- 相比之下, JVM 运行时调用操作系统并不快。这涉及一种称为“系统调用 (syscall)”的操作。系统调用的典型流程如下：

- 将系统调用参数放入寄存器。
- 执行 SYSENTER 陷阱指令。
- 陷阱处理器切换到特权状态并更改虚拟内存映射。然后它分派到处理特定系统调用的代码。
- 系统调用处理器检查参数, 确保不会访问用户进程不应看到的内存。
- 执行系统调用的具体工作。以 `read` 系统调用为例, 可能包括：
 - 检查文件描述符当前的位置是否有数据可读
 - 调用文件系统处理器, 将所需数据从磁盘 (或其他存储位置) 取入缓冲区缓存,
 - 将数据从缓冲区缓存复制到 JVM 提供的地址

```

public static void main(String[] args) throws Exception {
    try (InputStream is = new BufferedInputStream(
        new FileInputStream(args[0]));
        OutputStream os = new BufferedOutputStream(
        new FileOutputStream(args[1]))) {
        int octet;
        while ((octet = is.read()) != -1) {
            os.write(octet);
        }
    }
}

```

These small changes will improve data copy rate by *at least* a couple of orders of magnitude, depending on various platform-related factors. The buffered stream wrappers cause the data to be read and written in larger chunks. The instances both have buffers implemented as byte arrays.

- With `is`, data is read from the file into the buffer a few kilobytes at a time. When `read()` is called, the implementation will typically return a byte from the buffer. It will only read from the underlying input stream if the buffer has been emptied.
- The behavior for `os` is analogous. Calls to `os.write(int)` write single bytes into the buffer. Data is only written to the output stream when the buffer is full, or when `os` is flushed or closed.

What about character-based streams?

As you should be aware, Java I/O provides different APIs for reading and writing binary and text data.

- `InputStream` and `OutputStream` are the base APIs for stream-based binary I/O
- `Reader` and `Writer` are the base APIs for stream-based text I/O.

For text I/O, `BufferedReader` and `BufferedWriter` are the equivalents for `BufferedInputStream` and `BufferedOutputStream`.

Why do buffered streams make this much difference?

The real reason that buffered streams help performance is to do with the way that an application talks to the operating system:

- Java method in a Java application, or native procedure calls in the JVM's native runtime libraries are fast. They typically take a couple of machine instructions and have minimal performance impact.
- By contrast, JVM runtime calls to the operating system are not fast. They involve something known as a "syscall". The typical pattern for a syscall is as follows:

- Put the syscall arguments into registers.
- Execute a SYSENTER trap instruction.
- The trap handler switched to privileged state and changes the virtual memory mappings. Then it dispatches to the code to handle the specific syscall.
- The syscall handler checks the arguments, taking care that it isn't being told to access memory that the user process should not see.
- The syscall specific work is performed. In the case of a `read` syscall, this may involve:
 - checking that there is data to be read at the file descriptor's current position
 - calling the file system handler to fetch the required data from disk (or wherever it is stored) into the buffer cache,
 - copying data from the buffer cache to the JVM-supplied address

4. 调整流指针和文件描述符位置
6. 从系统调用返回。这涉及再次更改虚拟内存映射并切换出特权状态。

正如你可以想象的，执行一次系统调用可能需要成千上万条机器指令。保守估计，至少比普通方法调用长两个数量级。（可能是三个或更多。）

鉴于此，缓冲流之所以能带来巨大差异，是因为它们大幅减少了系统调用的次数。缓冲输入流不是对每次read()调用都执行系统调用，而是根据需要将大量数据读入缓冲区。大多数对缓冲流的read()调用只是做一些简单的边界检查，并返回之前读取的字节。类似的道理也适用于输出流和字符流的情况。

（有些人认为缓冲I/O性能的提升来自于读取请求大小与磁盘块大小、磁盘旋转延迟等因素的不匹配。事实上，现代操作系统采用多种策略，确保应用程序通常不需要等待磁盘。这并不是性能提升的真正原因。）

缓冲流总是有优势吗？

不总是。如果你的应用程序将进行大量“小”读写操作，缓冲流绝对是有优势的。然而，如果你的应用程序只需要对一个大的byte[]或char[]进行大块读写，那么缓冲流不会带来实质性的好处。实际上，可能还会有（微小的）性能损失。

这是Java中复制文件最快的方法吗？

不是。当你使用Java基于流的API复制文件时，至少会产生一次额外的数据内存到内存的拷贝。如果使用NIO的ByteBuffer和Channel API，可以避免这种情况。（这里添加一个单独示例的链接。）

第135.7节：陷阱——过度使用原始包装类型效率低下

考虑以下两段代码：

```
int a = 1000;
int b = a + 1;
```

和

```
Integer a = 1000;
Integer b = a + 1;
```

问题：哪个版本更高效？

答案：这两个版本看起来几乎相同，但第一个版本比第二个版本高效得多。

第二个版本使用的数字表示占用更多空间，并且在后台依赖自动装箱和自动拆箱。实际上，第二个版本等同于以下代码：

```
Integer a = Integer.valueOf(1000);           // 装箱1000
Integer b = Integer.valueOf(a.intValue() + 1); // 拆箱1000, 加1, 装箱1001
```

与使用int的版本相比，使用Integer时显然多了三个额外的方法调用。在valueOf的情况下，每次调用都会创建并初始化一个新的Integer对象。所有这些额外的装箱和拆箱操作很可能使第二个版本比第一个版本慢一个数量级。

4. adjusting the stream pointer
6. Return from the syscall. This entails changing VM mappings again and switching out of privileged state.

As you can imagine, performing a single syscall can thousands of machine instructions. Conservatively, *at least* two orders of magnitude longer than a regular method call. (Probably three or more.)

Given this, the reason that buffered streams make a big difference is that they drastically reduce the number of syscalls. Instead of doing a syscall for each read() call, the buffered input stream reads a large amount of data into a buffer as required. Most read() calls on the buffered stream do some simple bounds checking and return a byte that was read previously. Similar reasoning applies in the output stream case, and also the character stream cases.

(Some people think that buffered I/O performance comes from the mismatch between the read request size and the size of a disk block, disk rotational latency and things like that. In fact, a modern OS uses a number of strategies to ensure that the application *typically* doesn't need to wait for the disk. This is not the real explanation.)

Are buffered streams always a win?

Not always. Buffered streams are definitely a win if your application is going to do lots of "small" reads or writes. However, if your application only needs to perform large reads or writes to / from a large byte[] or char[], then buffered streams will give you no real benefits. Indeed there might even be a (tiny) performance penalty.

Is this the fastest way to copy a file in Java?

No it isn't. When you use Java's stream-based APIs to copy a file, you incur the cost of at least one extra memory-to-memory copy of the data. It is possible to avoid this if you use the NIO ByteBuffer and Channel APIs. (Add a link to a separate example here.)

Section 135.7: Pitfall - Over-use of primitive wrapper types is inefficient

Consider these two pieces of code:

```
int a = 1000;
int b = a + 1;
```

and

```
Integer a = 1000;
Integer b = a + 1;
```

Question: Which version is more efficient?

Answer: The two versions look almost the identical, but the first version is a lot more efficient than the second one.

The second version is using a representation for the numbers that uses more space, and is relying on auto-boxing and auto-unboxing behind the scenes. In fact the second version is directly equivalent to the following code:

```
Integer a = Integer.valueOf(1000);           // box 1000
Integer b = Integer.valueOf(a.intValue() + 1); // unbox 1000, add 1, box 1001
```

Comparing this to the other version that uses int, there are clearly three extra method calls when Integer is used. In the case of valueOf, the calls are each going to create and initialize a new Integer object. All of this extra boxing and unboxing work is likely to make the second version an order of magnitude slower than the first one.

此外，第二个版本在每次`valueOf`调用时都会在堆上分配对象。虽然空间利用率取决于平台，但每个`Integer`对象大约需要16字节左右。相比之下，`int`版本不需要额外的堆空间，假设`a`和`b`是局部变量。

另一个原始类型比其装箱类型更快的重要原因是它们各自的数组类型在内存中的布局方式不同。

以`int[]`和`Integer[]`为例，`int[]`中的`int`值在内存中是连续排列的。但在`Integer[]`中，排列的不是值本身，而是指向`Integer`对象的引用（指针），这些对象中才包含实际的`int`值。

除了增加了一层间接访问外，这在遍历值时对缓存局部性影响很大。对于`int[]`，CPU可以一次性将数组中的所有值加载到缓存中，因为它们在内存中是连续的。但对于`Integer[]`，CPU可能需要为每个元素额外进行一次内存访问，因为数组中只包含指向实际值的引用。

简而言之，使用原始类型的包装类在CPU和内存资源上相对昂贵。不必要的使用它们是低效的。

第135.8节：陷阱——创建日志消息的开销

TRACE和DEBUG日志级别用于传达代码运行时的高细节信息。通常建议将日志级别设置高于这两个级别，但需要注意，即使这些语句看似“关闭”，也不能影响性能。

考虑以下日志语句：

```
// 处理某种请求，记录参数
LOG.debug("请求来自 " + myInetAddress.toString()
+ " 参数: " + Arrays.toString(veryLongParamArray));
```

即使日志级别设置为`INFO`，传递给`debug()`的参数也会在每次执行该行时被计算。这会在多个方面造成不必要的资源消耗：

- `字符串` 拼接：会创建多个`字符串`实例
- `InetAddress` 甚至可能进行 DNS 查询。
- `veryLongParamArray` 可能非常长——将其转换为字符串会消耗内存，耗费时间

解决方案

大多数日志框架提供了使用固定字符串和对象引用创建日志消息的方法。日志消息只有在实际记录时才会被计算。示例：

```
// 如果调试被禁用，则不会调用 toString()，也不会进行字符串拼接
LOG.debug("来自 {} 的请求，参数: {}", myInetAddress, parameters);
```

只要所有参数都能通过`String.valueOf(Object)`转换为字符串，这种方法效果很好。如果日志消息的计算更复杂，可以在记录日志前检查日志级别：

```
if (LOG.isDebugEnabled()) {
    // 仅在启用 DEBUG 时计算参数表达式
    LOG.debug("请求来自 {}, 参数: {}", myInetAddress,
        Arrays.toString(veryLongParamArray));
```

In addition to that, the second version is allocating objects on the heap in each `valueOf` call. While the space utilization is platform specific, it is likely to be in the region of 16 bytes for each `Integer` object. By contrast, the `int` version needs zero extra heap space, assuming that `a` and `b` are local variables.

Another big reason why primitives are faster than their boxed equivalent is how their respective array types are laid out in memory.

If you take `int[]` and `Integer[]` as an example, in the case of an `int[]` the `int` values are contiguously laid out in memory. But in the case of an `Integer[]` it's not the values that are laid out, but references (pointers) to `Integer` objects, which in turn contain the actual `int` values.

Besides being an extra level of indirection, this can be a big tank when it comes to cache locality when iterating over the values. In the case of an `int[]` the CPU could fetch all the values in the array, into its cache at once, because they are contiguous in memory. But in the case of an `Integer[]` the CPU potentially has to do an additional memory fetch for each element, since the array only contains references to the actual values.

In short, using primitive wrapper types is relatively expensive in both CPU and memory resources. Using them unnecessarily is inefficient.

Section 135.8: Pitfall - The overheads of creating log messages

TRACE and DEBUG log levels are there to be able to convey high detail about the operation of the given code at runtime. Setting the log level above these is usually recommended, however some care must be taken for these statements to not affect performance even when seemingly "turned off".

Consider this log statement:

```
// Processing a request of some kind, logging the parameters
LOG.debug("Request coming from " + myInetAddress.toString()
+ " parameters: " + Arrays.toString(veryLongParamArray));
```

Even when the log level is set to `INFO`, arguments passed to `debug()` will be evaluated on each execution of the line. This makes it unnecessarily consuming on several counts:

- `String` concatenation: multiple `String` instances will be created
- `InetAddress` might even do a DNS lookup.
- the `veryLongParamArray` might be very long - creating a `String` out of it consumes memory, takes time

Solution

Most logging framework provide means to create log messages using fix strings and object references. The log message will be evaluated only if the message is actually logged. Example:

```
// No toString() evaluation, no string concatenation if debug is disabled
LOG.debug("Request coming from {} parameters: {}", myInetAddress, parameters);
```

This works very well as long as all parameters can be converted to strings using `String.valueOf(Object)`. If the log message computation is more complex, the log level can be checked before logging:

```
if (LOG.isDebugEnabled()) {
    // Argument expression evaluated only when DEBUG is enabled
    LOG.debug("Request coming from {}, parameters: {}", myInetAddress,
        Arrays.toString(veryLongParamArray));
```

}

这里，LOG.debug() 中耗时的 Arrays.toString(Object[]) 计算仅在实际启用 DEBUG 时执行。

第135.9节：陷阱 - 遍历 Map 的键可能效率低下

以下示例代码比实际需要的要慢：

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // 对 key 和 value 进行某些操作
}
```

这是因为它对 map 中的每个键都需要进行一次查找（调用 get() 方法）。这种查找可能效率不高（在 HashMap 中，它涉及调用键的 hashCode 方法，然后在内部数据结构中查找正确的桶，有时还需要调用 equals 方法）。对于大型 map，这可能不是一个微不足道的开销。

避免这种情况的正确方法是对映射的条目进行迭代，详细内容见集合 (Collections) 主题。

第135.10节：陷阱——调用System.gc()效率低下

调用System.gc()（几乎总是）是一个不好的主意。

gc()方法的javadoc说明如下：

“调用gc方法表示建议Java虚拟机努力回收未使用的对象，以便使它们当前占用的内存可快速重用。当方法调用返回时，Java虚拟机已尽最大努力回收所有被丢弃对象的空间。”

可以从中得出几个要点：

1. 使用“suggests”（建议）一词，而不是（比如）“tells”（告诉），意味着JVM可以自由选择是否忽略该建议。默认的JVM行为（最近版本）是遵循该建议，但可以通过启动JVM时设置-XX:+DisableExplicitGC来覆盖此行为。
2. “尽最大努力回收所有被丢弃对象的空间”这句话意味着调用gc将触发一次“完全”的垃圾回收。

那么为什么调用System.gc()是个坏主意呢？

首先，运行一次完整的垃圾回收代价很高。完整的垃圾回收需要访问并“标记”所有仍然可达的对象；也就是说，所有不是垃圾的对象。如果你在垃圾不多的时候触发它，那么垃圾回收会做大量工作，但收益却相对较小。

其次，完整的垃圾回收容易破坏未被回收对象的“局部性”特性。由同一线程在大致相同时刻分配的对象往往在同一内存中被分配得很接近。这是有益的。同期分配的对象很可能相关；即相互引用。如果你的应用程序使用这些引用，那么由于各种内存和页面缓存效应，内存访问的速度可能会更快。不幸的是，完整的垃圾回收往往会移动对象，导致原本相近的对象现在变得更远。

}

Here, LOG.debug() with the costly `Arrays.toString(0bect[])` computation is processed only when DEBUG is actually enabled.

Section 135.9: Pitfall - Iterating a Map's keys can be inefficient

The following example code is slower than it needs to be :

```
Map<String, String> map = new HashMap<>();
for (String key : map.keySet()) {
    String value = map.get(key);
    // Do something with key and value
}
```

That is because it requires a map lookup (the get() method) for each key in the map. This lookup may not be efficient (in a HashMap, it entails calling hashCode on the key, then looking up the correct bucket in internal data structures, and sometimes even calling equals). On a large map, this may not be a trivial overhead.

The correct way of avoiding this is to iterate on the map's entries, which is detailed in the Collections topic

Section 135.10: Pitfall - Calling System.gc() is inefficient

It is (almost always) a bad idea to call `System.gc()`.

The javadoc for the `gc()` method specifies the following:

“Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for quick reuse. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all discarded objects.”

There are a couple of important points that can be drawn from this:

1. The use of the word "suggests" rather than (say) "tells" means that the JVM is free to ignore the suggestion. The default JVM behavior (recent releases) is to follow the suggestion, but this can be overridden by setting -XX:+DisableExplicitGC when launching the JVM.
2. The phrase "a best effort to reclaim space from all discarded objects" implies that calling `gc` will trigger a "full" garbage collection.

So why is calling `System.gc()` a bad idea?

First, running a full garbage collection is expensive. A full GC involves visiting and "marking" every object that is still reachable; i.e. every object that is not garbage. If you trigger this when there isn't much garbage to be collected, then the GC does a lot of work for relatively little benefit.

Second, a full garbage collection is liable to disturb the "locality" properties of the objects that are not collected. Objects that are allocated by the same thread at roughly the same time tend to be allocated close together in memory. This is good. Objects that are allocated at the same time are likely to be related; i.e. reference each other. If your application uses those references, then the chances are that memory access will be faster because of various memory and page caching effects. Unfortunately, a full garbage collection tends to move objects around so that objects that were once close are now further apart.

第三，运行完整的垃圾回收可能会导致你的应用程序暂停，直到回收完成。
在此期间，你的应用程序将无响应。

事实上，最好的策略是让JVM决定何时运行垃圾回收，以及运行哪种类型的回收。如果你不干预，JVM会选择一个优化吞吐量或最小化垃圾回收暂停时间的时机和回收类型。

一开始我们说过“.....（几乎总是）一个坏主意.....”。实际上，有几个场景可能是个好主意：

1. 如果你正在为某些对垃圾回收敏感的代码（例如涉及终结器或弱引用/软引用/虚引用的代码）编写单元测试，那么调用`System.gc()`可能是必要的。
2. 在某些交互式应用中，可能会有特定的时间点，用户不会在意是否发生垃圾回收暂停。一个例子是游戏中存在自然暂停的“游戏”阶段；例如加载新关卡时。

第135.11节：陷阱——调用 '`new String(String)`' 效率低下

使用`newString(String)`来复制字符串效率低下且几乎总是不必要的。

- 字符串对象是不可变的，因此无需复制它们来防止更改。
- 在某些较旧版本的Java中，`String`对象可能与其他`String`对象共享底层数组。在这些版本中，通过创建一个（小的）子字符串来保留（大的）字符串，可能会导致内存泄漏。
然而，从Java 7开始，`String`的底层数组不再共享。

在没有任何实际好处的情况下，调用`newString(String)`纯粹是浪费：

- 复制操作会消耗CPU时间。
- 复制会使用更多内存，增加应用程序的内存占用和/或增加垃圾回收开销。
- 如果字符串对象被复制，像`equals(Object)`和`hashCode()`这样的操作可能会变慢。

Third, running a full garbage collection is liable to make your application pause until the collection is complete.
While this is happening, your application will be non-responsive.

In fact, the best strategy is to let the JVM decide when to run the GC, and what kind of collection to run. If you don't interfere, the JVM will choose a time and collection type that optimizes throughput or minimizes GC pause times.

At the beginning we said "... (almost always) a bad idea ...". In fact there are a couple of scenarios where it *might* be a good idea:

1. If you are implementing a unit test for some code that is garbage collection sensitive (e.g. something involving finalizers or weak / soft / phantom references) then calling `System.gc()` may be necessary.
2. In some interactive applications, there can be particular points in time where the user won't care if there is a garbage collection pause. One example is a game where there are natural pauses in the "play"; e.g. when loading a new level.

Section 135.11: Pitfall - Calling '`new String(String)`' is inefficient

Using `new String(String)` to duplicate a string is inefficient and almost always unnecessary.

- `String` objects are immutable, so there is no need to copy them to protect against changes.
- In some older versions of Java, `String` objects can share backing arrays with other `String` objects. In those versions, it is possible to leak memory by creating a (small) substring of a (large) string and retaining it.
However, from Java 7 onwards, `String` backing arrays are not shared.

In the absence of any tangible benefit, calling `new String(String)` is simply wasteful:

- Making the copy takes CPU time.
- The copy uses more memory which increases the application's memory footprint and / or increases GC overheads.
- Operations like `equals(Object)` and `hashCode()` can be slower if `String` objects are copied.

第136章：ServiceLoader

第136.1节：简单的ServiceLoader示例

ServiceLoader是一种简单易用的内置机制，用于动态加载接口实现。通过服务加载器——提供实例化手段（但不负责连接）——可以在Java SE中构建一个简单的依赖注入机制。借助ServiceLoader接口和实现的分离，程序扩展变得自然且方便。实际上，许多Java API都是基于ServiceLoader实现的。

基本概念包括

- 操作服务的接口
- 通过ServiceLoader获取服务的实现
- 提供服务的实现

让我们从接口开始，并将其放入一个jar包中，例如命名为accounting-api.jar

```
包example;  
  
公共接口AccountingService{  
  
    long getBalance();  
}
```

现在我们提供了该服务的一个实现，打包在名为accounting-impl.jar的jar文件中，包含该服务的一个实现

```
package example.impl;  
import example.AccountingService;  
  
public interface DefaultAccountingService implements AccountingService {  
  
    public long getBalance() {  
        return balanceFromDB();  
    }  
  
    private long balanceFromDB(){  
        ...  
    }  
}
```

此外，accounting-impl.jar 包含一个文件，声明该 jar 提供了 AccountingService 的实现。该文件路径必须以 META-INF/services/ 开头，且文件名必须与接口的全限定名相同：

- META-INF/services/example.AccountingService

该文件的内容是实现类的全限定名：

```
example.impl.DefaultAccountingService
```

假设这两个 jar 都在使用 AccountingService 的程序的类路径中，可以通过 ServiceLauncher 获取该服务的实例

Chapter 136: ServiceLoader

Section 136.1: Simple ServiceLoader Example

The ServiceLoader is a simple and easy to use built-in mechanism for dynamic loading of interface implementations. With the service loader - providing means for instantiation (but not the wiring) - a simple dependency injection mechanism can be built in Java SE. With the ServiceLoader interface and implementation separation becomes natural and programs can be conveniently extended. Actually a lot of Java API are implemented based on the ServiceLoader

The basic concepts are

- Operating on *interfaces* of services
- Obtaining implementation(s) of the service via ServiceLoader
- Providing implementation of services

Lets start with the interface and put it in a jar, named for example accounting-api.jar

```
package example;  
  
public interface AccountingService {  
  
    long getBalance();  
}
```

Now we provide an implementation of that service in a jar named accounting-impl.jar, containing an implementation of the service

```
package example.impl;  
import example.AccountingService;  
  
public interface DefaultAccountingService implements AccountingService {  
  
    public long getBalance() {  
        return balanceFromDB();  
    }  
  
    private long balanceFromDB(){  
        ...  
    }  
}
```

further, the accounting-impl.jar contains a file declaring that this jar provides an implementation of AccountingService. The file has to have a path starting with META-INF/services/ and must have the same name as the *fully-qualified* name of the interface:

- META-INF/services/example.AccountingService

The content of the file is the *fully-qualified* name of the implementation:

```
example.impl.DefaultAccountingService
```

Given both jars are in the classpath of the program, that consumes the AccountingService, an instance of the Service can be obtained by using the ServiceLauncher

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

由于ServiceLoader是一个Iterable，它支持多个实现提供者，程序可以从中选择：

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
    //...
}
```

注意，调用next()时总会创建一个新实例。如果想重用实例，必须使用ServiceLoader的iterator()方法，或者如上所示使用for-each循环。

第136.2节：日志服务

下面的示例展示了如何通过ServiceLoader实例化一个用于日志记录的类。

Service

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;
}
```

服务的实现

以下实现简单地将消息写入System.err

```
package servicetest.logger;

import servicetest.Logger;

public class ConsoleLogger implements Logger {

    @Override
    public void log(String message) {
        System.err.println(message);
    }

    @Override
    public void close() {
    }
}
```

以下实现将消息写入文本文件：

```
package servicetest.logger;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
```

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
AccountingService service = loader.next();
long balance = service.getBalance();
```

As the ServiceLoader is an Iterable, it supports multiple implementation providers, where the program may choose from:

```
ServiceLoader<AccountingService> loader = ServiceLoader.load(AccountingService.class)
for(AccountingService service : loader) {
    //...
}
```

Note that when invoking next() a new instance will always be created. If you want to re-use an instance, you have to use the iterator() method of the ServiceLoader or the for-each loop as shown above.

Section 136.2: Logger Service

The following example shows how to instantiate a class for logging via the ServiceLoader.

Service

```
package servicetest;

import java.io.IOException;

public interface Logger extends AutoCloseable {

    void log(String message) throws IOException;
}
```

Implementations of the service

The following implementation simply writes the message to System.err

```
package servicetest.logger;

import servicetest.Logger;

public class ConsoleLogger implements Logger {

    @Override
    public void log(String message) {
        System.err.println(message);
    }

    @Override
    public void close() {
    }
}
```

The following implementation writes the messages to a text file:

```
package servicetest.logger;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
```

```

import servicetest.Logger;

public class 文件记录器 implements Logger {

    private final BufferedWriter 写入器;

    public 文件记录器() throws IOException {
        写入器 = new BufferedWriter(new FileWriter("log.txt"));
    }

    @Override
    public void 记录(String 消息) throws IOException {
        写入器.append(消息);
        写入器.newLine();
    }

    @Override
    public void 关闭() throws IOException {
        写入器.close();
    }
}

```

META-INF/services/servicetest.Logger

文件 META-INF/services/servicetest.Logger 列出了 Logger 实现的名称。

servicetest.logger.控制台记录器
servicetest.logger.文件记录器

用法

以下 main 方法将消息写入所有可用的记录器。记录器是通过 ServiceLoader 实例化的。

```

public static void main(String[] args) throws Exception {
    final String message = "Hello World!";

    // 获取 Logger 的 ServiceLoader
    ServiceLoader<Logger> loader = ServiceLoader.load(servicetest.Logger.class);

    // 遍历可用 Logger 的实例，将消息写入每一个
    Iterator<Logger> iterator = loader.iterator();
    while (iterator.hasNext()) {
        try (Logger logger = iterator.next()) {
            logger.log(message);
        }
    }
}

```

```

import servicetest.Logger;

public class FileLogger implements Logger {

    private final BufferedWriter writer;

    public FileLogger() throws IOException {
        writer = new BufferedWriter(new FileWriter("log.txt"));
    }

    @Override
    public void log(String message) throws IOException {
        writer.append(message);
        writer.newLine();
    }

    @Override
    public void close() throws IOException {
        writer.close();
    }
}

```

META-INF/services/servicetest.Logger

The META-INF/services/servicetest.Logger file lists the names of the Logger implementations.

servicetest.logger.ConsoleLogger
servicetest.logger.FileLogger

Usage

The following main method writes a message to all available loggers. The loggers are instantiated using ServiceLoader.

```

public static void main(String[] args) throws Exception {
    final String message = "Hello World!";

    // get ServiceLoader for Logger
    ServiceLoader<Logger> loader = ServiceLoader.load(servicetest.Logger.class);

    // iterate through instances of available loggers, writing the message to each one
    Iterator<Logger> iterator = loader.iterator();
    while (iterator.hasNext()) {
        try (Logger logger = iterator.next()) {
            logger.log(message);
        }
    }
}

```

第137章：类加载器

第137.1节：实现自定义类加载器

每个自定义加载器必须直接或间接继承java.lang.ClassLoader类。主要的扩展点是以下方法：

- `findClass(String)` - 如果你的类加载器遵循标准的委托模型进行类加载，则重载此方法。
- `loadClass(String, boolean)` - 重载此方法以实现替代的委托模型。
- `findResource` 和 `findResources` - 重载这些方法以自定义资源加载。

负责从字节数组实际加载类的 `defineClass` 方法是 `final` 的，以防止重载。任何自定义行为都需要在调用 `defineClass` 之前执行。

这里有一个简单的示例，从字节数组加载特定类：

```
public class ByteArrayClassLoader extends ClassLoader {  
    private String classname;  
    private byte[] classfile;  
  
    public ByteArrayClassLoader(String classname, byte[] classfile) {  
        this.classname = classname;  
        this.classfile = classfile.clone();  
    }  
  
    @Override  
    protected Class findClass(String classname) throws ClassNotFoundException {  
        if (classname.equals(this.classname)) {  
            return defineClass(classname, classfile, 0, classfile.length);  
        } else {  
            throw new ClassNotFoundException(classname);  
        }  
    }  
}
```

由于我们只重写了 `findClass` 方法，当调用

`loadClass` 时，这个自定义类加载器将表现如下。

1. 类加载器的 `loadClass` 方法调用 `findLoadedClass` 以查看是否已经存在具有该名称的类由此类加载器加载。如果成功，生成的 `Class` 对象将返回给请求者。
2. 然后，`loadClass` 方法通过调用其父类加载器的 `loadClass` 方法将任务委托给父类加载器。如果父类加载器能够处理请求时，它将返回一个 `Class` 对象，然后将该对象返回给请求者。
3. 如果父类加载器无法加载该类，`findClass` 则调用我们重写的 `findClass` 方法，传递要加载的类名。
4. 如果请求的名称与 `this.classname` 匹配，我们调用 `defineClass` 从中加载实际的类这个 `.class` 文件字节数组。然后返回生成的 `Class` 对象。
5. 如果名称不匹配，我们将抛出 `ClassNotFoundException`。

第137.2节：加载外部.class文件

要加载一个类，我们首先需要定义它。类是由 `ClassLoader` 定义的。只有一个问题，Oracle 没有为 `ClassLoader` 编写带有此功能的代码。要定义类，我们需要访问一个名为 `defineClass()` 的方法，它是 `ClassLoader` 的私有方法。

Chapter 137: Classloaders

Section 137.1: Implementing a custom classLoader

Every custom loader must directly or indirectly extend the `java.lang.ClassLoader` class. The main extension points are the following methods:

- `findClass(String)` - overload this method if your classloader follows the standard delegation model for class loading.
- `loadClass(String, boolean)` - overload this method to implement an alternative delegation model.
- `findResource` and `findResources` - overload these methods to customize resource loading.

The `defineClass` methods which are responsible for actually loading the class from a byte array are `final` to prevent overloading. Any custom behavior needs to be performed prior to calling `defineClass`.

Here is a simple that loads a specific class from a byte array:

```
public class ByteArrayClassLoader extends ClassLoader {  
    private String classname;  
    private byte[] classfile;  
  
    public ByteArrayClassLoader(String classname, byte[] classfile) {  
        this.classname = classname;  
        this.classfile = classfile.clone();  
    }  
  
    @Override  
    protected Class findClass(String classname) throws ClassNotFoundException {  
        if (classname.equals(this.classname)) {  
            return defineClass(classname, classfile, 0, classfile.length);  
        } else {  
            throw new ClassNotFoundException(classname);  
        }  
    }  
}
```

Since we have only overridden the `findClass` method, this custom class loader is going to behave as follows when `loadClass` is called.

1. The classloader's `loadClass` method calls `findLoadedClass` to see if a class with this name has already been loaded by this classloader. If that succeeds, the resulting `Class` object is returned to the requestor.
2. The `loadClass` method then delegates to the parent classloader by calling its `loadClass` call. If the parent can deal with the request, it will return a `Class` object which is then returned to the requestor.
3. If the parent classloader cannot load the class, `findClass` then calls our override `findClass` method, passing the name of the class to be loaded.
4. If the requested name matches `this.classname`, we call `defineClass` to load the actual class from the `this.classfile` byte array. The resulting `Class` object is then returned.
5. If the name did not match, we throw `ClassNotFoundException`.

Section 137.2: Loading an external .class file

To load a class we first need to define it. The class is defined by the `ClassLoader`. There's just one problem, Oracle didn't write the `ClassLoader`'s code with this feature available. To define the class we will need to access a method named `defineClass()` which is a private method of the `ClassLoader`.

为了访问它，我们将创建一个新类ByteClassLoader，并继承ClassLoader。现在我们已经将类继承自ClassLoader，就可以访问ClassLoader的私有方法。为了使defineClass()可用，我们将创建一个新方法，作为私有defineClass()方法的镜像。调用私有方法时需要类名name、类字节数组classBytes、第一个字节的偏移量（为0，因为classBytes的数据从classBytes[0]开始），以及最后一个字节的偏移量（为classBytes.length，因为它表示数据的大小，即最后的偏移量）。

```
public class ByteClassLoader extends ClassLoader {  
  
    public Class<?> defineClass(String name, byte[] classBytes) {  
        return defineClass(name, classBytes, 0, classBytes.length);  
    }  
}
```

现在，我们有了一个公共的defineClass()方法。可以通过传入类名和类字节作为参数来调用它。

假设我们有一个名为MyClass的类，位于包stackoverflow中.....

要调用该方法，我们需要类字节，因此通过使用Paths.get()方法并传入二进制类的路径作为参数，创建一个表示类路径的Path对象。现在，我们可以获取类字节了Files.readAllBytes(path)。于是我们创建了一个ByteClassLoader实例，并使用我们创建的方法，defineClass()。我们已经有了类的字节码，但要调用我们的方法，还需要类名，类名由包名（点号）和类的规范名组成，在本例中是stackoverflow.MyClass。

```
Path path = Paths.get("MyClass.class");  
  
ByteClassLoader loader = new ByteClassLoader();  
loader.defineClass("stackoverflow.MyClass", Files.readAllBytes(path));
```

注意：defineClass()方法返回一个Class<?>对象。如果需要，可以保存它。

要加载类，我们只需调用loadClass()并传入类名。此方法可能抛出ClassNotFoundException，因此需要使用try catch块

```
try{  
    loader.loadClass("stackoverflow.MyClass");  
} catch(ClassNotFoundException e){  
    e.printStackTrace();  
}
```

第137.3节：实例化和使用类加载器

这个基本示例展示了应用程序如何实例化一个类加载器并使用它动态加载类。

```
URL[] urls = new URL[] {new URL("file:/home/me/extras.jar")};  
ClassLoader loader = new URLClassLoader(urls);  
Class<?> myObjectClass = loader.findClass("com.example.MyObject");
```

本示例中创建的类加载器将以默认类加载器作为其父加载器，并且会先尝试在父类加载器中查找任何类，然后才会在"extra.jar"中查找。如果请求的类已经被加载，findClass调用将返回先前加载类的引用。

调用findClass可能以多种方式失败。最常见的有：

To access it, what we will do is create a new class, ByteClassLoader, and extend it to `ClassLoader`. Now that we have extended our class to `ClassLoader`, we can access the `ClassLoader`'s private methods. To make `defineClass()` available, we will create a new method that will act like a mirror for the private `defineClass()` method. To call the private method we will need the class name, name, the class bytes, `classBytes`, the first byte's offset, which will be 0 because `classBytes`' data starts at `classBytes[0]`, and the last byte's offset, which will be `classBytes.length` because it represents the size of the data, which will be the last offset.

```
public class ByteClassLoader extends ClassLoader {  
  
    public Class<?> defineClass(String name, byte[] classBytes) {  
        return defineClass(name, classBytes, 0, classBytes.length);  
    }  
}
```

Now, we have a public `defineClass()` method. It can be called by passing the name of the class and the class bytes as arguments.

Let's say we have class named `MyClass` in the package `stackoverflow`...

To call the method we need the class bytes so we create a `Path` object representing our class' path by using the `Paths.get()` method and passing the path of the binary class as an argument. Now, we can get the class bytes with `Files.readAllBytes(path)`. So we create a `ByteClassLoader` instance and use the method we created, `defineClass()`. We already have the class bytes but to call our method we also need the class name which is given by the package name (dot) the class canonical name, in this case `stackoverflow.MyClass`.

```
Path path = Paths.get("MyClass.class");  
  
ByteClassLoader loader = new ByteClassLoader();  
loader.defineClass("stackoverflow.MyClass", Files.readAllBytes(path));
```

Note: The `defineClass()` method returns a `Class<?>` object. You can save it if you want.

To load the class, we just call `loadClass()` and pass the class name. This method can throw an `ClassNotFoundException` so we need to use a try catch block

```
try{  
    loader.loadClass("stackoverflow.MyClass");  
} catch(ClassNotFoundException e){  
    e.printStackTrace();  
}
```

Section 137.3: Instantiating and using a classloader

This basic example shows how an application can instantiate a classloader and use it to dynamically load a class.

```
URL[] urls = new URL[] {new URL("file:/home/me/extras.jar")};  
ClassLoader loader = new URLClassLoader(urls);  
Class<?> myObjectClass = loader.findClass("com.example.MyObject");
```

The classloader created in this example will have the default classloader as its parent, and will first try to find any class in the parent classloader before looking in "extra.jar". If the requested class has already been loaded, the `findClass` call will return the reference to the previously loaded class.

The `findClass` call can fail in a variety of ways. The most common are:

- 如果找不到指定的类，调用将抛出`ClassNotFoundException`。
- 如果指定的类依赖于某个找不到的其他类，调用将抛出
`NoClassDefFoundError`。

- If the named class cannot be found, the call will throw `ClassNotFoundException`.
- If the named class depends on some other class that cannot be found, the call will throw `NoClassDefFoundError`.

第138章：编程创建图像

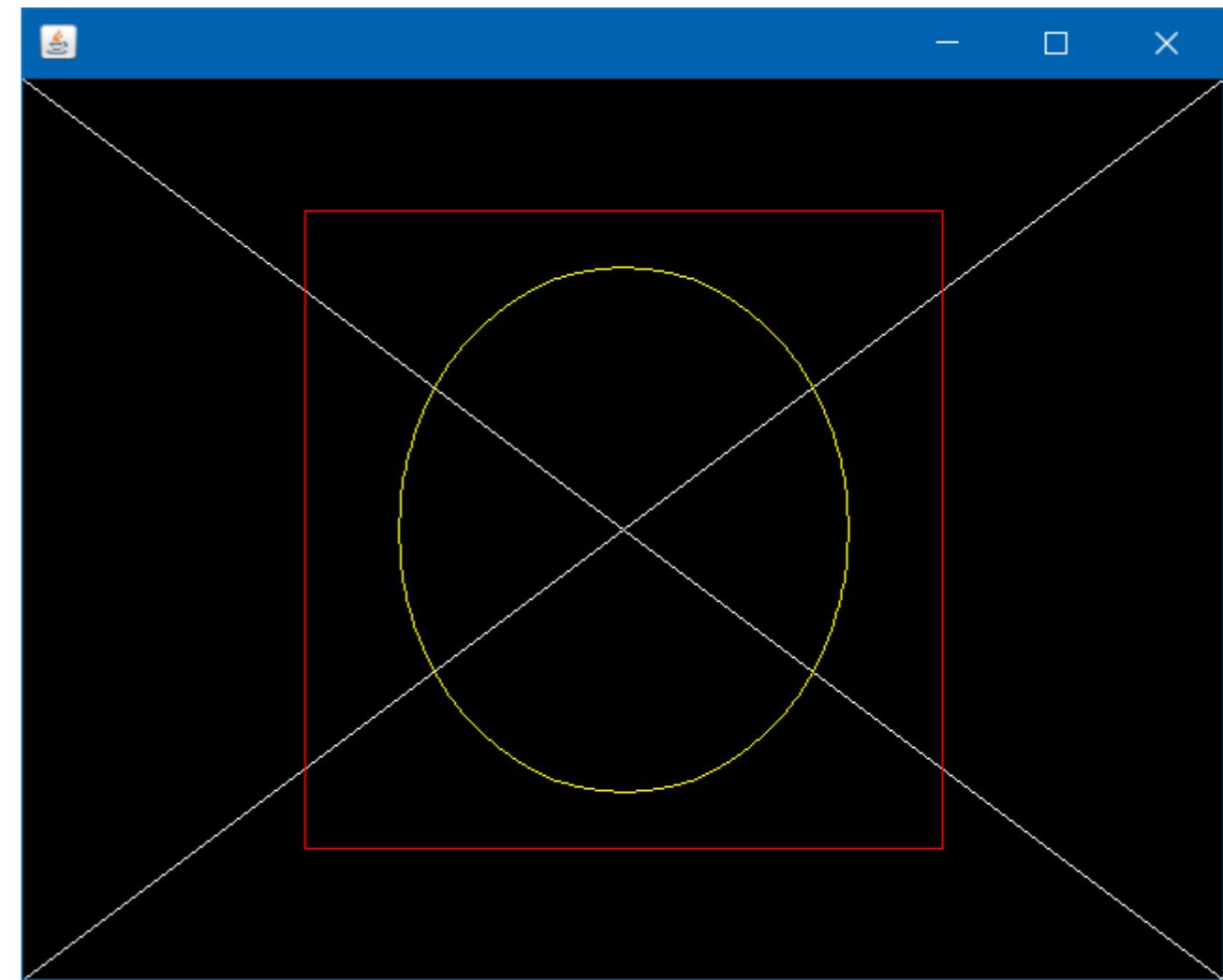
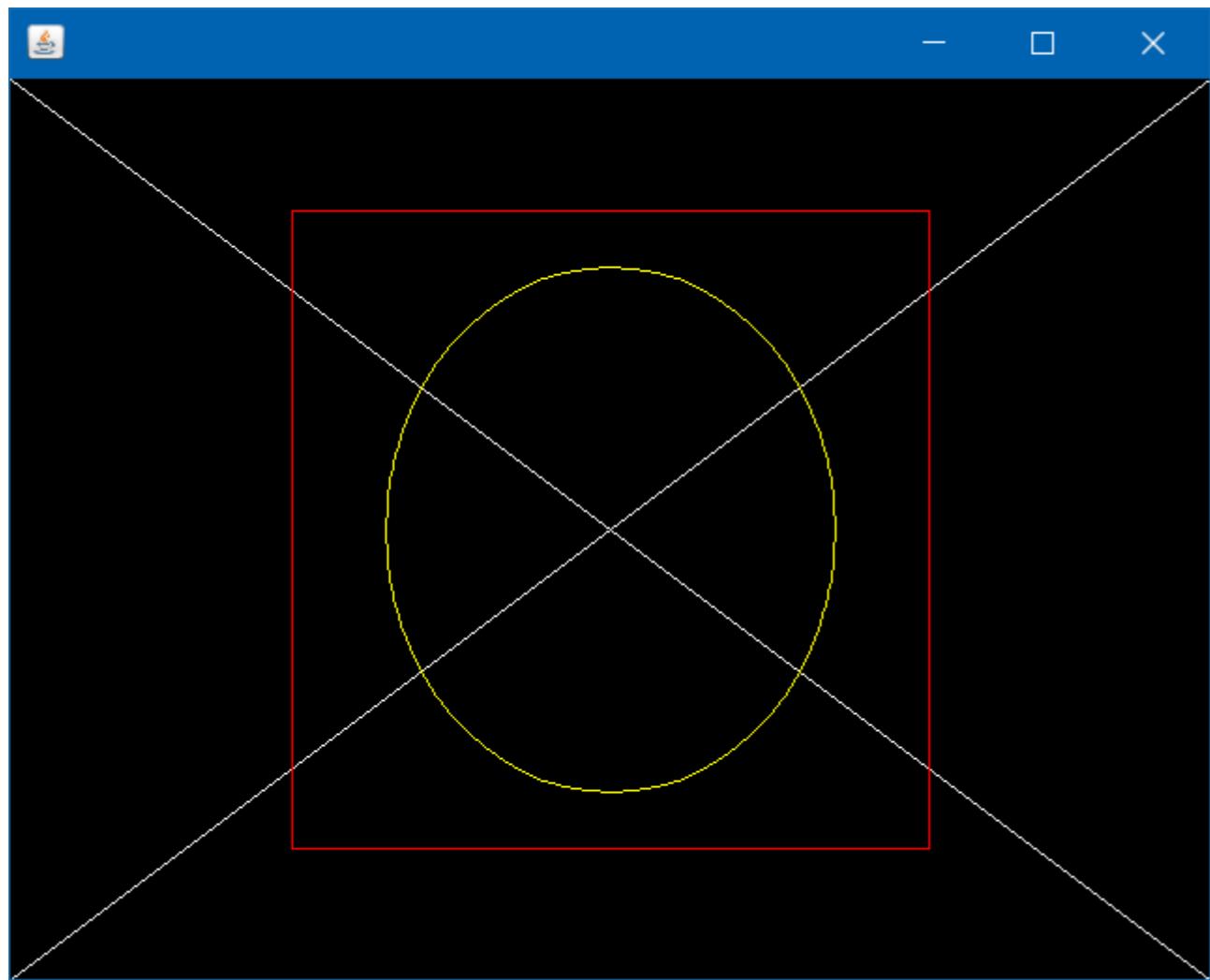
第138.1节：编程创建简单图像并显示

```
类 ImageCreationExample {  
  
    静态 Image createSampleImage() {  
        // 实例化一个新的 BufferedImage (Image 的子类) 实例  
        BufferedImage img = new BufferedImage(640, 480, BufferedImage.TYPE_INT_ARGB);  
  
        // 在图像上绘制内容  
        paintOnImage(img);  
  
        return img;  
    }  
  
    static void paintOnImage(BufferedImage img) {  
        // 获取一个可绘制的 Graphics2D (Graphics 的子类) 对象  
        Graphics2D g2d = (Graphics2D) img.getGraphics();  
  
        // 一些示例绘制  
        g2d.setColor(Color.BLACK);  
        g2d.fillRect(0, 0, 640, 480);  
        g2d.setColor(Color.WHITE);  
        g2d.drawLine(0, 0, 640, 480);  
        g2d.drawLine(0, 480, 640, 0);  
        g2d.setColor(Color.YELLOW);  
        g2d.drawOval(200, 100, 240, 280);  
        g2d.setColor(Color.RED);  
        g2d.drawRect(150, 70, 340, 340);  
  
        // 在图像上绘制可能非常耗费内存  
        // 因此最好尽早释放资源  
        // 虽然这不是必须的  
        g2d.dispose();  
    }  
  
    public static void main(String[] args) {  
        JFrame frame = new JFrame();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Image img = createSampleImage();  
        ImageIcon icon = new ImageIcon(img);  
        frame.add(new JLabel(icon));  
        frame.pack();  
        frame.setVisible(true);  
    }  
}
```

Chapter 138: Creating Images Programmatically

Section 138.1: Creating a simple image programmatically and displaying it

```
class ImageCreationExample {  
  
    static Image createSampleImage() {  
        // instantiate a new BufferedImage (subclass of Image) instance  
        BufferedImage img = new BufferedImage(640, 480, BufferedImage.TYPE_INT_ARGB);  
  
        // draw something on the image  
        paintOnImage(img);  
  
        return img;  
    }  
  
    static void paintOnImage(BufferedImage img) {  
        // get a drawable Graphics2D (subclass of Graphics) object  
        Graphics2D g2d = (Graphics2D) img.getGraphics();  
  
        // some sample drawing  
        g2d.setColor(Color.BLACK);  
        g2d.fillRect(0, 0, 640, 480);  
        g2d.setColor(Color.WHITE);  
        g2d.drawLine(0, 0, 640, 480);  
        g2d.drawLine(0, 480, 640, 0);  
        g2d.setColor(Color.YELLOW);  
        g2d.drawOval(200, 100, 240, 280);  
        g2d.setColor(Color.RED);  
        g2d.drawRect(150, 70, 340, 340);  
  
        // drawing on images can be very memory-consuming  
        // so it's better to free resources early  
        // it's not necessary, though  
        g2d.dispose();  
    }  
  
    public static void main(String[] args) {  
        JFrame frame = new JFrame();  
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        Image img = createSampleImage();  
        ImageIcon icon = new ImageIcon(img);  
        frame.add(new JLabel(icon));  
        frame.pack();  
        frame.setVisible(true);  
    }  
}
```



第138.2节：将图像保存到磁盘

```
public static void saveImage(String destination) throws IOException {
    // 方法在“以编程方式创建简单图像并显示”示例中实现
    BufferedImage img = createSampleImage();

    // ImageIO 提供了多种写入方法，支持不同的输出
    ImageIO.write(img, "png", new File(destination));
}
```

Section 138.2: Save an Image to disk

```
public static void saveImage(String destination) throws IOException {
    // method implemented in "Creating a simple image Programmatically and displaying it" example
    BufferedImage img = createSampleImage();

    // ImageIO provides several write methods with different outputs
    ImageIO.write(img, "png", new File(destination));
}
```

第138.3节：在 BufferedImage 中设置单个像素的颜色

```
BufferedImage image = new BufferedImage(256, 256, BufferedImage.TYPE_INT_ARGB);

//你不必使用Graphics对象，可以单独读取和设置像素颜色
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
        int alpha = 255; //别忘了这个，或者改用BufferedImage.TYPE_INT_RGB
        int red = i; //或者任何你喜欢的公式
        int green = j; //或者任何你喜欢的公式
        int blue = 50; //或者任何你喜欢的公式
        int color = (alpha << 24) | (red << 16) | (green << 8) | blue;
        image.setRGB(i, j, color);
    }
}

ImageIO.write(image, "png", new File("computed.png"));
```

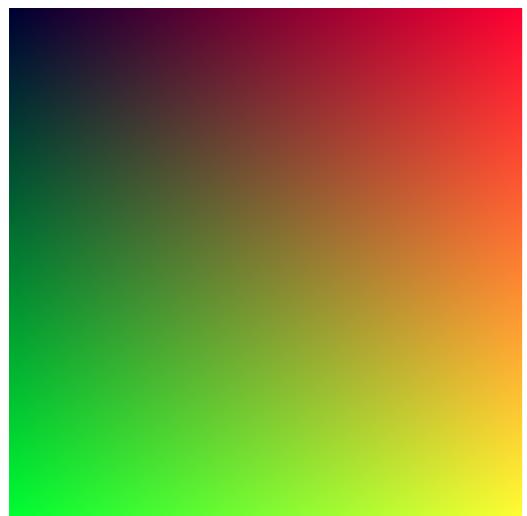
Section 138.3: Setting individual pixel's color in BufferedImage

```
BufferedImage image = new BufferedImage(256, 256, BufferedImage.TYPE_INT_ARGB);

//you don't have to use the Graphics object, you can read and set pixel color individually
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 256; j++) {
        int alpha = 255; //don't forget this, or use BufferedImage.TYPE_INT_RGB instead
        int red = i; //or any formula you like
        int green = j; //or any formula you like
        int blue = 50; //or any formula you like
        int color = (alpha << 24) | (red << 16) | (green << 8) | blue;
        image.setRGB(i, j, color);
    }
}

ImageIO.write(image, "png", new File("computed.png"));
```

输出：

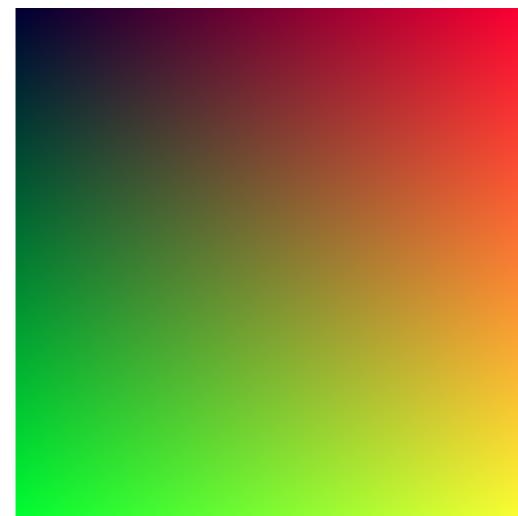


第138.4节：指定图像渲染质量

```
static void setupQualityHigh(Graphics2D g2d) {  
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);  
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);  
    // 许多其他 RenderingHints 键/值对可供指定  
}  
  
static void setupQualityLow(Graphics2D g2d) {  
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_OFF);  
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_SPEED);  
}
```

示例图像的质量与速度渲染比较：

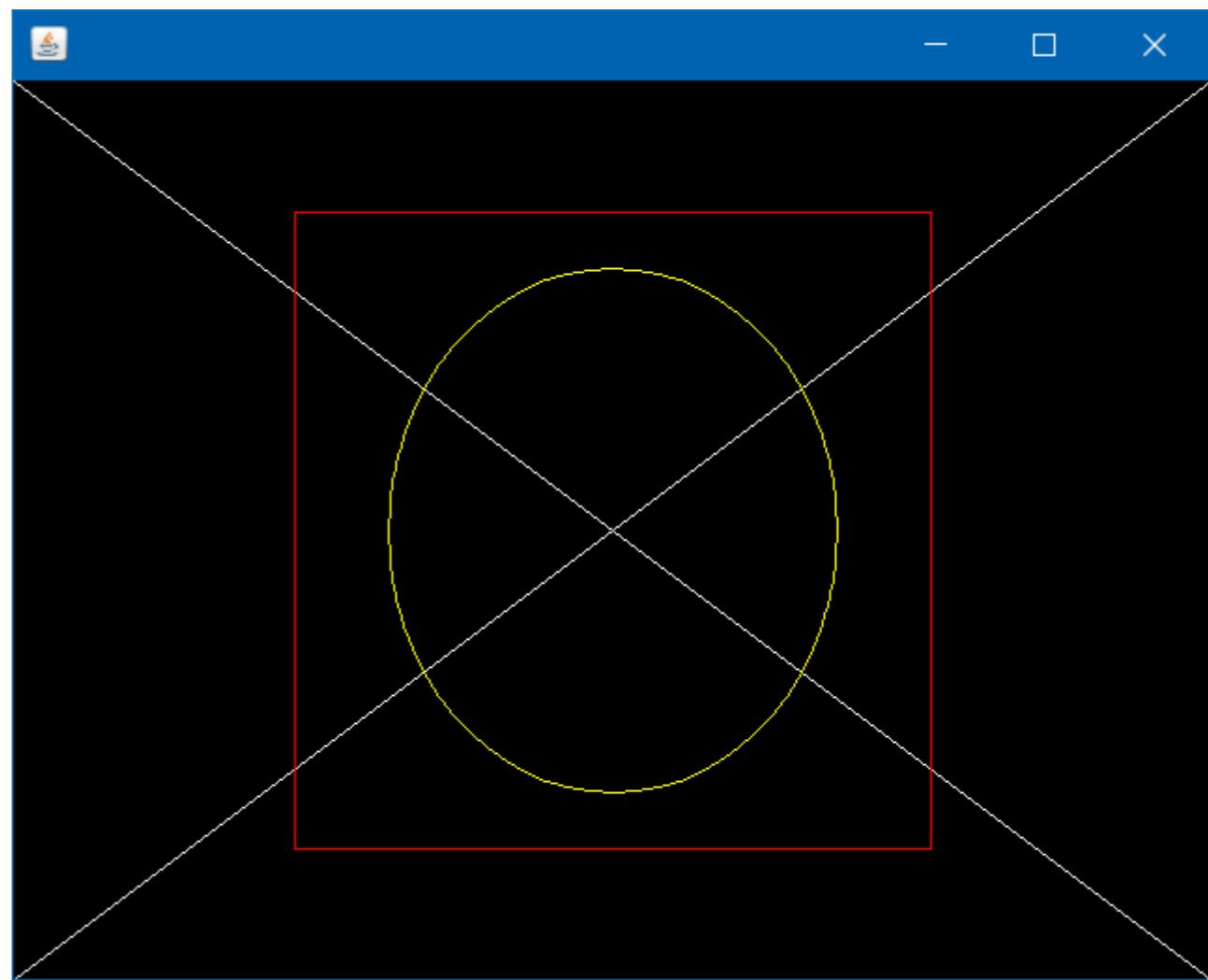
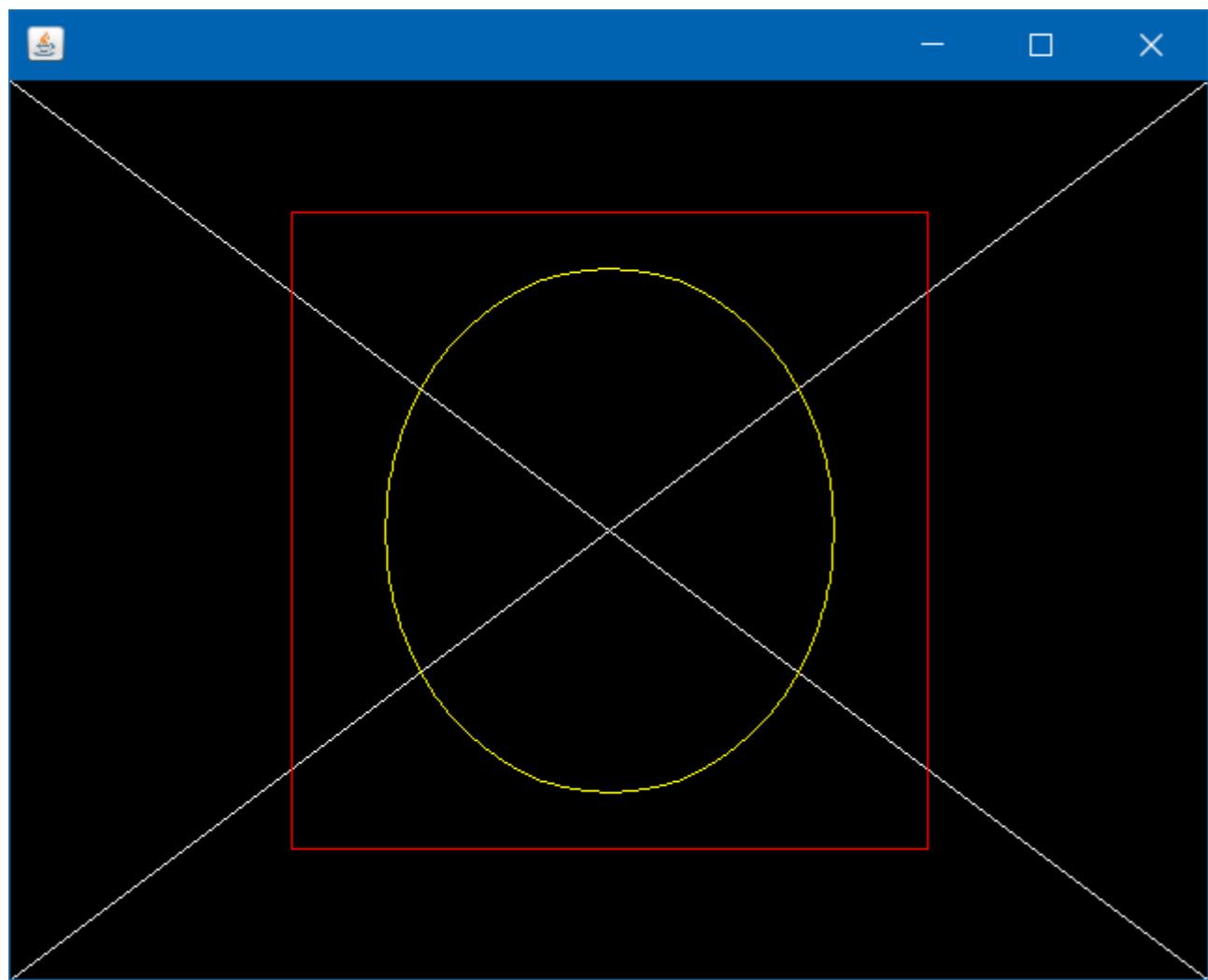
Output:

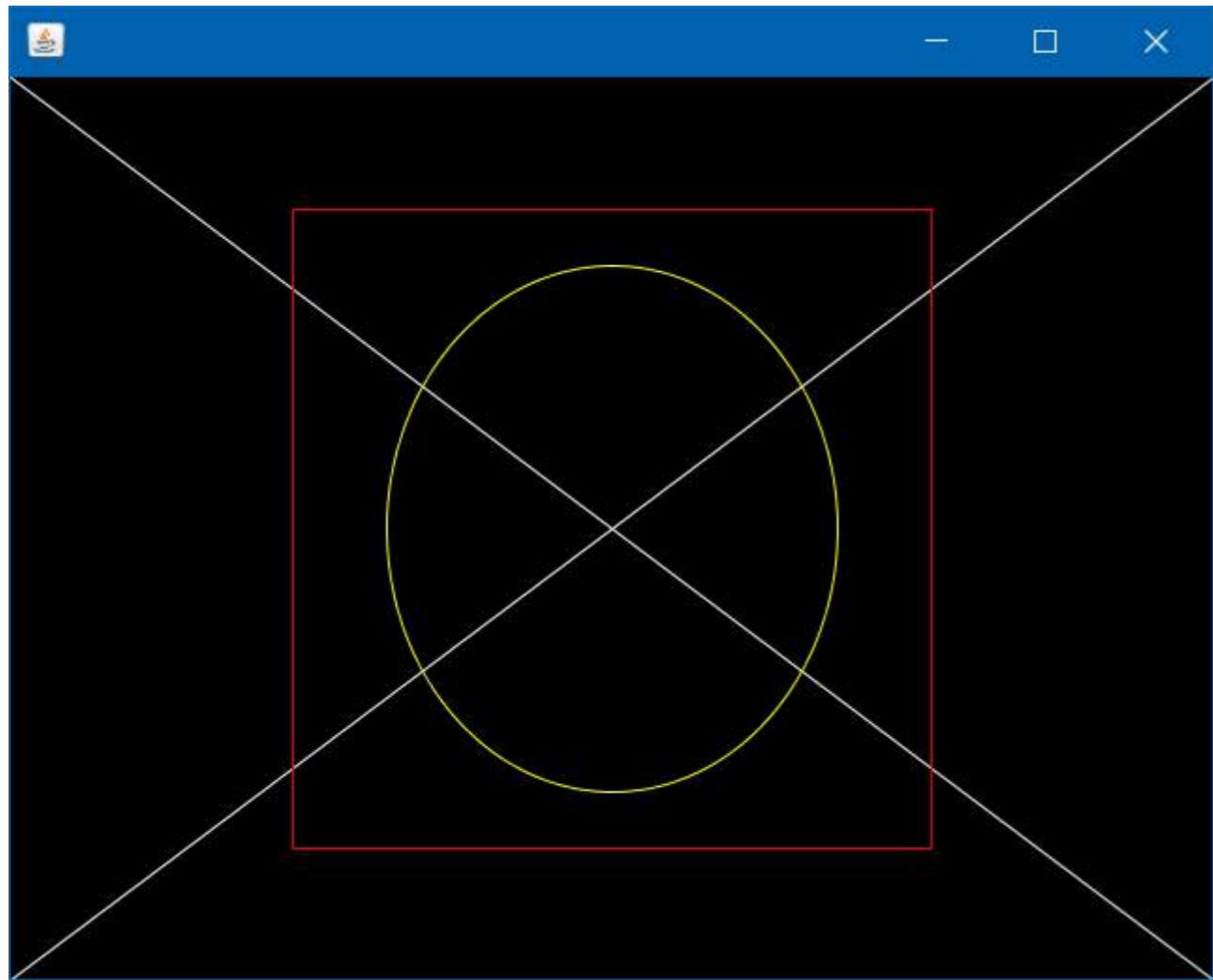


Section 138.4: Specifying image rendering quality

```
static void setupQualityHigh(Graphics2D g2d) {  
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);  
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY);  
    // many other RenderingHints KEY/VALUE pairs to specify  
}  
  
static void setupQualityLow(Graphics2D g2d) {  
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_OFF);  
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_SPEED);  
}
```

A comparison of QUALITY and SPEED rendering of the sample image:





第138.5节：使用 BufferedImage 类创建图像

```
int width = 256; // 像素宽度
int height = 256; // 像素高度
BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_4BYTE_ABGR);
// BufferedImage.TYPE_4BYTE_ABGR - 存储 RGB 颜色和透明度 (alpha) , 详见 javadoc

Graphics g = image.createGraphics();

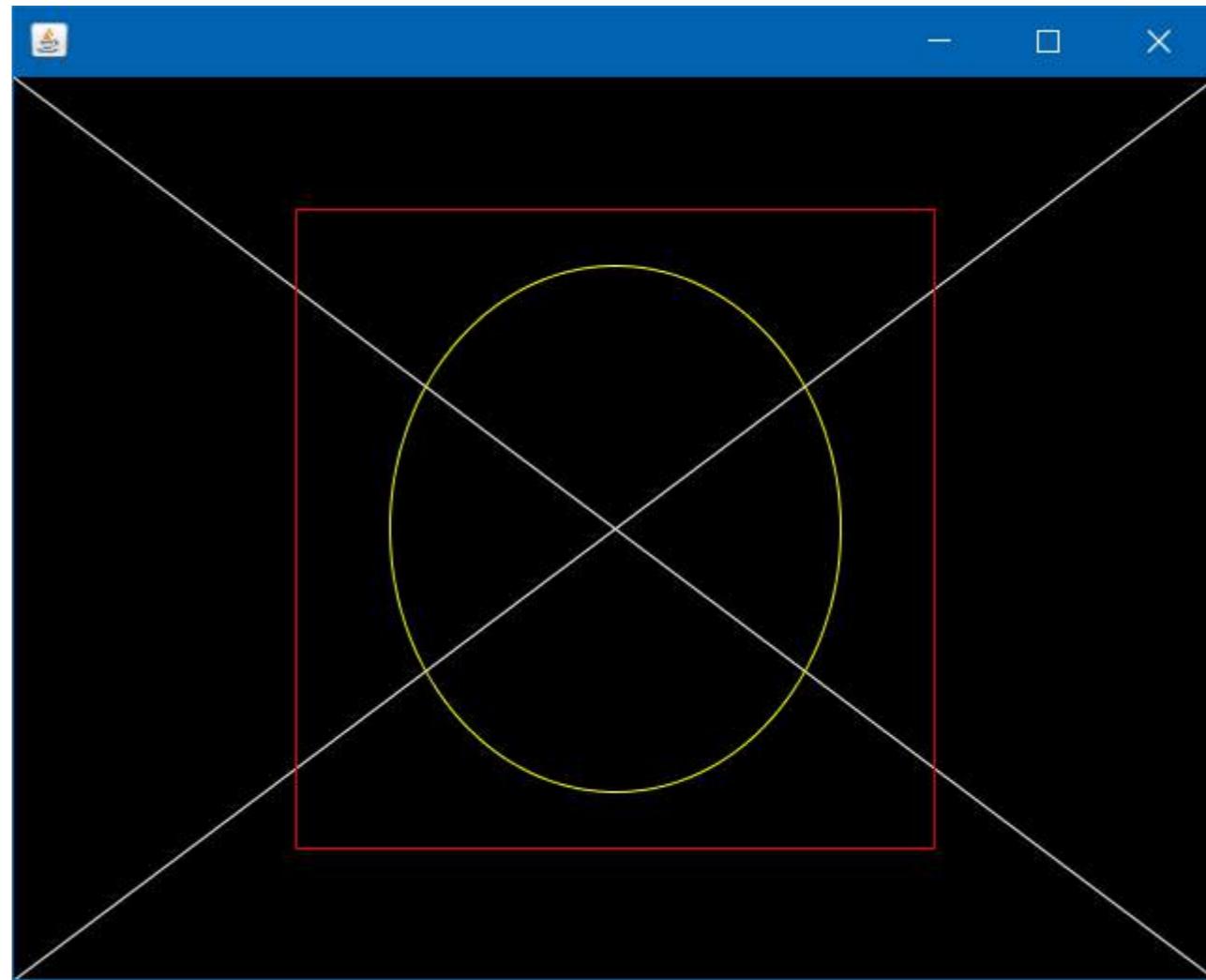
        //随意绘制, 就像在UI应用程序中的drawComponent(Graphics g)方法中那样绘制
g.setColor(Color.RED);
g.fillRect(20, 30, 50, 50);

g.setColor(Color.BLUE);
g.drawOval(120, 120, 80, 40);

g.dispose(); //当图形对象不再需要时释放它们//现在图像已经通过程序生成内容, 可以在graphics.drawImage()中使用它来绘制到其他地方

//或者简单地保存到文件
ImageIO.write(image, "png", new File("myimage.png"));
```

输出：



Section 138.5: Creating an image with BufferedImage class

```
int width = 256; //in pixels
int height = 256; //in pixels
BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE_4BYTE_ABGR);
//BufferedImage.TYPE_4BYTE_ABGR - store RGB color and visibility (alpha), see javadoc for more info

Graphics g = image.createGraphics();

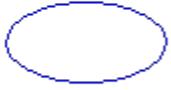
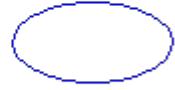
        //draw whatever you like, like you would in a drawComponent(Graphics g) method in an UI application
g.setColor(Color.RED);
g.fillRect(20, 30, 50, 50);

g.setColor(Color.BLUE);
g.drawOval(120, 120, 80, 40);

g.dispose(); //dispose graphics objects when they are no longer needed

//now image has programmatically generated content, you can use it in graphics.drawImage() to draw it somewhere else
//or just simply save it to a file
ImageIO.write(image, "png", new File("myimage.png"));
```

Output:



第138.6节：使用BufferedImage编辑和重用图像

```
BufferedImage cat = ImageIO.read(new File("cat.jpg")); //读取现有文件

//修改它
Graphics g = cat.createGraphics();
g.setColor(Color.RED);
g.drawString("Cat", 10, 10);
g.dispose();

//现在创建一个新图像
BufferedImage cats = new BufferedImage(256, 256, BufferedImage.TYPE_4BYTE_ABGR);

//并在其上绘制旧图像，16次
g = cats.createGraphics();
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        g.drawImage(cat, i * 64, j * 64, null);
    }
}

g.setColor(Color.BLUE);
g.drawRect(0, 0, 255, 255); //添加一些漂亮的边框
g.dispose(); //完成

ImageIO.write(cats, "png", new File("cats.png"));
```

原始 cat 文件：



生成的文件：

Section 138.6: Editing and re-using image with BufferedImage

```
BufferedImage cat = ImageIO.read(new File("cat.jpg")); //read existing file

//modify it
Graphics g = cat.createGraphics();
g.setColor(Color.RED);
g.drawString("Cat", 10, 10);
g.dispose();

//now create a new image
BufferedImage cats = new BufferedImage(256, 256, BufferedImage.TYPE_4BYTE_ABGR);

//and draw the old one on it, 16 times
g = cats.createGraphics();
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        g.drawImage(cat, i * 64, j * 64, null);
    }
}

g.setColor(Color.BLUE);
g.drawRect(0, 0, 255, 255); //add some nice border
g.dispose(); //and done

ImageIO.write(cats, "png", new File("cats.png"));
```

Original cat file:



Produced file:



第 138.7 节：如何缩放 BufferedImage

```
/**  
 * 使用由 BufferedImage 支持的 Graphics2D 对象调整图像大小。  
 * @param srcImg - 要缩放的源图像  
 * @param w - 期望的宽度  
 * @param h - 期望的高度  
 * @return - 新的调整大小后的图像  
 */  
private BufferedImage getScaledImage(Image srcImg, int w, int h){  
  
    // 创建一个新的合适大小的图像，该图像包含或可能包含介于 0.0 和 1.0 之间（包括两端）的任意透明度值。  
    BufferedImage resizedImg = new BufferedImage(w, h, BufferedImage.TRANSLUCENT);  
  
    // 创建一个与设备无关的对象来绘制调整大小后的图像  
    Graphics2D g2 = resizedImg.createGraphics();  
  
    // 这可以被更改，参见  
    // http://stackoverflow.com/documentation/java/5482/creating-images-programmatically/19498/specifying-image-rendering-quality  
    g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,  
    RenderingHints.VALUE_INTERPOLATION_BILINEAR);  
  
    // 最后在 Graphics2D 中以所需大小绘制源图像。  
    g2.drawImage(srcImg, 0, 0, w, h, null);  
  
    // 释放此图形上下文并释放其使用的任何系统资源  
    g2.dispose();  
  
    // 返回用于创建 Graphics2D 的图像  
    return resizedImg;  
}
```



Section 138.7: How to scale a BufferedImage

```
/**  
 * Resizes an image using a Graphics2D object backed by a BufferedImage.  
 * @param srcImg - source image to scale  
 * @param w - desired width  
 * @param h - desired height  
 * @return - the new resized image  
 */  
private BufferedImage getScaledImage(Image srcImg, int w, int h){  
  
    // Create a new image with good size that contains or might contain arbitrary alpha values between  
    // and including 0.0 and 1.0.  
    BufferedImage resizedImg = new BufferedImage(w, h, BufferedImage.TRANSLUCENT);  
  
    // Create a device-independant object to draw the resized image  
    Graphics2D g2 = resizedImg.createGraphics();  
  
    // This could be changed, Cf.  
    // http://stackoverflow.com/documentation/java/5482/creating-images-programmatically/19498/specifying-image-rendering-quality  
    g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,  
    RenderingHints.VALUE_INTERPOLATION_BILINEAR);  
  
    // Finally draw the source image in the Graphics2D with the desired size.  
    g2.drawImage(srcImg, 0, 0, w, h, null);  
  
    // Disposes of this graphics context and releases any system resources that it is using  
    g2.dispose();  
  
    // Return the image used to create the Graphics2D  
    return resizedImg;  
}
```

第139章：原子类型

参数	描述
设置	字段的易失性设置
获取	字段的易失性读取
延迟设置	这是字段的有序存储操作
compareAndSet	如果值是预期值，则将其设置为新值
getAndSet	获取当前值并更新

Java 原子类型是简单的可变类型，提供基本的线程安全且原子性的操作，无需使用锁定。它们适用于锁定会成为并发瓶颈，或存在死锁或活锁风险的情况。

第139.1节：创建原子类型

对于简单的多线程代码，使用同步是可以接受的。然而，使用同步确实会影响活性，并且随着代码库变得更加复杂，发生死锁、饥饿或活锁的可能性也会增加。

在更复杂的并发情况下，使用原子变量通常是更好的选择，因为它允许以线程安全的方式访问单个变量，而无需使用同步方法或代码块的开销。

创建一个AtomicInteger类型：

```
AtomicInteger aInt = new AtomicInteger() // 创建默认值为0  
AtomicInteger aInt = new AtomicInteger(1) // 创建初始值为1
```

其他实例类型同理。

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // 创建指定长度的数组  
= new AtomicIntegerArray(new int[] {1, 2, 3}) // 用另一个数组初始化数组
```

其他原子类型同理。

有一个显著的例外是没有float和double类型。这些可以通过使用模拟实现
`Float.floatToIntBits(float)` 和 `Float.intBitsToFloat(int)` 用于 float，
`Double.doubleToLongBits(double)` 和 `Double.longBitsToDouble(long)` 用于 double 类型。

如果您愿意使用 sun.misc.Unsafe，可以通过使用 sun.misc.Unsafe 中的原子操作将任何原始变量作为原子变量使用。所有原始类型都应转换或编码为 int 或 long，以便以这种方式使用。更多内容请参见：sun.misc.Unsafe。

第139.2节：原子类型的动机

实现多线程应用程序的简单方法是使用 Java 内置的同步和锁定原语；例如 `synchronized` 关键字。以下示例展示了如何使用 `synchronized` 来累加计数。

```
public class 计数器 {  
    private final int[] 计数器;
```

Chapter 139: Atomic Types

Parameter	Description
set	Volatile set of the field
get	Volatile read of the field
lazySet	This is a store ordered operation of the field
compareAndSet	If the value is the expected value then sent it to the new value
getAndSet	get the current value and update

Java Atomic Types are simple mutable types that provide basic operations that are thread-safe and atomic without resorting to locking. They are intended for use in cases where locking would be a concurrency bottleneck, or where there is risk of deadlock or livelock.

Section 139.1: Creating Atomic Types

For simple multi-threaded code, using synchronization is acceptable. However, using synchronization does have a liveness impact, and as a codebase becomes more complex, the likelihood goes up that you will end up with [Deadlock](#), [Starvation](#), or [Livelock](#).

In cases of more complex concurrency, using Atomic Variables is often a better alternative, as it allows an individual variable to be accessed in a thread-safe manner without the overhead of using synchronized methods or code blocks.

Creating an AtomicInteger type:

```
AtomicInteger aInt = new AtomicInteger() // Create with default value 0  
AtomicInteger aInt = new AtomicInteger(1) // Create with initial value 1
```

Similarly for other instance types.

```
AtomicIntegerArray aIntArray = new AtomicIntegerArray(10) // Create array of specific length  
AtomicIntegerArray aIntArray = new AtomicIntegerArray(new int[] {1, 2, 3}) // Initialize array with another array
```

Similarly for other atomic types.

There is a notable exception that there is no `float` and `double` types. These can be simulated through the use of `Float.floatToIntBits(float)` and `Float.intBitsToFloat(int)` for `float` as well as `Double.doubleToLongBits(double)` and `Double.longBitsToDouble(long)` for doubles.

If you are willing to use `sun.misc.Unsafe` you can use any primitive variable as atomic by using the atomic operation in `sun.misc.Unsafe`. All primitive types should be converted or encoded in int or longs to so use it in this way. For more on this see: `sun.misc.Unsafe`.

Section 139.2: Motivation for Atomic Types

The simple way to implement multi-threaded applications is to use Java's built-in synchronization and locking primitives; e.g. the `synchronized` keyword. The following example shows how we might use `synchronized` to accumulate counts.

```
public class Counters {  
    private final int[] counters;
```

```

public 计数器(int 计数器数量) {
    计数器 = new int[计数器数量];
}

/**
 * 增加给定索引处的整数值
 */
public synchronized void 计数(int 数字) {
    if (数字 >= 0 && 数字 < 计数器.length) {
        计数器[数字]++;
    }
}

/**
 * 获取给定索引处数字的当前计数,
 * 如果该索引处没有数字, 则返回0。
 */
public synchronized int 获取计数(int 数字) {
    return (数字 >= 0 && 数字 < 计数器.length) ? 计数器[数字] : 0;
}

```

此实现将正常工作。然而，如果有大量线程对同一个Counters对象进行大量同时调用，同步机制可能会成为瓶颈。具体来说：

1. 每个同步方法调用都会从当前线程获取Counters实例的锁开始。
2. 线程在检查number值并更新计数器时会持有该锁。
3. 最后，它将释放锁，允许其他线程访问。

如果一个线程尝试获取锁时另一个线程已持有该锁，尝试的线程将在步骤1被阻塞（停止），直到锁被释放。如果有多个线程在等待，其中一个将获得锁，其他线程将继续被阻塞。

这可能导致几个问题：

- 如果锁的争用很大（即许多线程尝试获取锁），那么某些线程可能会被阻塞很长时间。
- 当线程因等待锁而被阻塞时，操作系统通常会尝试切换执行到另一个线程。这种上下文切换对处理器性能有较大影响。
- 当多个线程被阻塞在同一个锁上时，不保证其中任何一个线程会被“公平”对待（即保证每个线程都能被调度运行）。这可能导致线程饥饿现象。

如何实现原子类型？

让我们先用AtomicInteger计数器重写上面的示例：

```

public class Counters {
    private final AtomicInteger[] counters;

    public Counters(int nosCounters) {
        counters = new AtomicInteger[nosCounters];
        for (int i = 0; i < nosCounters; i++) {
            counters[i] = new AtomicInteger();
        }
    }
}

```

```

public Counters(int nosCounters) {
    counters = new int[nosCounters];
}

/**
 * Increments the integer at the given index
 */
public synchronized void count(int number) {
    if (number >= 0 && number < counters.length) {
        counters[number]++;
    }
}

/**
 * Obtains the current count of the number at the given index,
 * or if there is no number at that index, returns 0.
 */
public synchronized int getCount(int number) {
    return (number >= 0 && number < counters.length) ? counters[number] : 0;
}

```

This implementation will work correctly. However, if you have a large number of threads making lots of simultaneous calls on the same Counters object, the synchronization is liable to be a bottleneck. Specifically:

1. Each **synchronized** method call will start with the current thread acquiring the lock for the Counters instance.
2. The thread will hold the lock while it checks number value and updates the counter.
3. Finally, the it will release the lock, allowing other threads access.

If one thread attempts to acquire the lock while another one holds it, the attempting thread will be blocked (stopped) at step 1 until the lock is released. If multiple threads are waiting, one of them will get it, and the others will continue to be blocked.

This can lead to a couple of problems:

- If there is a lot of contention for the lock (i.e. lots of threads try to acquire it), then some threads can be blocked for a long time.
- When a thread is blocked waiting for the lock, the operating system will typically try switch execution to a different thread. This context switching incurs a relatively large performance impact on the processor.
- When there are multiple threads blocked on the same lock, there are no guarantees that any one of them will be treated “fairly” (i.e. each thread is guaranteed to be scheduled to run). This can lead to *thread starvation*.

How does one implement Atomic Types?

Let us start by rewriting the example above using AtomicInteger counters:

```

public class Counters {
    private final AtomicInteger[] counters;

    public Counters(int nosCounters) {
        counters = new AtomicInteger[nosCounters];
        for (int i = 0; i < nosCounters; i++) {
            counters[i] = new AtomicInteger();
        }
    }
}

```

```

    }

    /**
     * 增加给定索引处的整数值
     */
    public void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number].incrementAndGet();
        }
    }

    /**
     * 获取给定索引处对象的当前计数,
     * 如果该索引处没有数字, 则返回0。
     */
    public int getCount(int number) {
        return (number >= 0 && number < counters.length) ?
            counters[number].get() : 0;
    }
}

```

我们用AtomicInteger[]替换了int[], 并在每个元素中初始化了一个实例。我们还将对int值的操作替换为incrementAndGet()和get()的调用。

但最重要的是, 我们可以去掉 synchronized关键字, 因为不再需要锁定。这是因为incrementAndGet()和get()操作是原子性且线程安全的。在此上下文中, 这意味着:

- 数组中的每个计数器只会在操作的“前”状态 (例如“递增”) 或“后”状态中被观察到。
- 假设操作发生在时间T, 任何线程在时间T之后都无法看到“前”状态。

此外, 虽然两个线程可能实际上会在同一时间尝试更新同一个AtomicInteger实例, 但操作的实现确保在给定实例上一次只发生一次递增。这是在无锁的情况下完成的, 通常能带来更好的性能。

原子类型是如何工作的?

原子类型通常依赖于目标机器指令集中的专用硬件指令。例如, 基于英特尔的指令集提供了一个CAS (比较并交换) 指令, 该指令能够原子地执行一系列特定的内存操作。

这些底层指令被用来实现相应API中更高级的操作
AtomicXxx类。例如, (同样是类似C的伪代码) :

```

private volatile num;

int increment() {
    while (TRUE) {
        int old = num;
        int new = old + 1;
        if (old == compare_and_swap(&num, old, new)) {
            return new;
        }
    }
}

```

```

    }

    /**
     * Increments the integer at the given index
     */
    public void count(int number) {
        if (number >= 0 && number < counters.length) {
            counters[number].incrementAndGet();
        }
    }

    /**
     * Obtains the current count of the object at the given index,
     * or if there is no number at that index, returns 0.
     */
    public int getCount(int number) {
        return (number >= 0 && number < counters.length) ?
            counters[number].get() : 0;
    }
}

```

We have replaced the `int[]` with an `AtomicInteger[]`, and initialized it with an instance in each element. We have also added calls to `incrementAndGet()` and `get()` in place of operations on `int` values.

But the most important thing is that we can remove the `synchronized` keyword because locking is no longer required. This works because the `incrementAndGet()` and `get()` operations are *atomic* and *thread-safe*. In this context, it means that:

- Each counter in the array will only be *observable* in either the “before” state for an operation (like an “increment”) or in the “after” state.
- Assuming that the operation occurs at time T, no thread will be able to see the “before” state after time T.

Furthermore, while two threads might actually attempt to update the same `AtomicInteger` instance at the same time, the implementations of the operations ensure that only one increment happens at a time on the given instance. This is done without locking, often resulting in better performance.

How do Atomic Types work?

Atomic types typically rely on specialized hardware instructions in the instruction set of the target machine. For example, Intel-based instruction sets provide a CAS ([Compare and Swap](#)) instruction that will perform a specific sequence of memory operations atomically.

These low-level instructions are used to implement higher-level operations in the APIs of the respective `AtomicXxx` classes. For example, (again, in C-like pseudocode):

```

private volatile num;

int increment() {
    while (TRUE) {
        int old = num;
        int new = old + 1;
        if (old == compare_and_swap(&num, old, new)) {
            return new;
        }
    }
}

```

如果对AtomicXxxx没有争用，if测试将成功，循环会立即结束。如果存在争用，除了一个线程外，所有线程的if都会失败，它们将在循环中“自旋”少量循环周期。实际上，自旋比挂起线程并切换到另一个线程快几个数量级（除非在不现实的高争用级别下，此时同步（synchronized）比原子类表现更好，因为当CAS操作失败时，重试只会增加更多争用）。

顺便提一下，CAS指令通常被JVM用来实现无争用锁定。如果JVM看到锁当前未被锁定，它会尝试使用CAS来获取锁。如果CAS成功，则无需进行昂贵的线程调度、上下文切换等操作。有关所用技术的更多信息，请参见HotSpot中的偏向锁定（Biased Locking）。

If there is no contention on the AtomicXxxx, the if test will succeed and the loop will end immediately. If there is contention, then the if will fail for all but one of the threads, and they will "spin" in the loop for a small number of cycles of the loop. In practice, the spinning is orders of magnitude faster (except at *unrealistically high* levels of contention, where synchronized performs better than atomic classes because when the CAS operation fails, then the retry will only add more contention) than suspending the thread and switching to another one.

Incidentally, CAS instructions are typically used by the JVM to implement *uncontented locking*. If the JVM can see that a lock is not currently locked, it will attempt to use a CAS to acquire the lock. If the CAS succeeds, then there is no need to do the expensive thread scheduling, context switching and so on. For more information on the techniques used, see [Biased Locking in HotSpot](#).

第140章：RSA加密

第140.1节：使用由OAEP和GCM组成的混合密码系统的示例

以下示例使用由AES GCM和OAEP组成的混合密码系统加密数据，使用它们的默认参数大小和128位AES密钥大小。

OAEP比PKCS#1 v1.5填充对填充oracle攻击的抵抗力更强。GCM也能防护填充oracle攻击。

解密可以先通过获取封装密钥的长度，然后获取封装密钥。随后，使用与公钥配对的RSA私钥对封装密钥进行解密。之后，AES/GCM加密的密文可以被解密回原始明文。

该协议包括：

1. 一个用于包装密钥的长度字段 (RSAPrivateKey 缺少 getKeySize() 方法)；
2. 包装/封装密钥，大小与RSA密钥的字节大小相同；
3. GCM密文和128位认证标签（由Java自动添加）。

注意事项：

- 要正确使用此代码，您应提供至少2048位的RSA密钥，密钥越大越好（但速度越慢，尤其是在解密时）；
- 要使用AES-256，您应先安装无限制密码策略文件；与其自己创建协议，不如使用
- 诸如加密消息语法 (CMS / PKCS#7) 或PGP等容器格式。

下面是示例：

```
/**  
 * 使用混合加密系统加密数据，该系统使用GCM加密数据，使用OAEP加密AES密钥。  
  
 * AES加密的密钥大小为128位。  
 * OAEP和GCM均使用所有默认参数选择。  
 *  
 * @param publicKey 用于包装 AES 密钥的 RSA 公钥  
 * @param plaintext 要加密的明文，未被修改  
 * @return 密文  
 * @throws InvalidKeyException 如果密钥不是 RSA 公钥  
 * @throws NullPointerException 如果明文为 null  
 */  
public static byte[] encryptData(PublicKey publicKey, byte[] plaintext)  
    throws InvalidKeyException, NullPointerException {  
  
    // --- 创建 RSA OAEP 密码器 ---  
  
    Cipher oaep;  
    try {  
        // SHA-1 是默认且在此设置下不易受攻击  
        // 使用 OAEPParameterSpec 配置不仅仅是哈希算法  
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");  
    } catch (NoSuchAlgorithmException e) {  
        throw new RuntimeException(  
            "运行时不支持 RSA 密码 (运行时的必备算法) ",  
            e);  
    }
```

Chapter 140: RSA Encryption

Section 140.1: An example using a hybrid cryptosystem consisting of OAEP and GCM

The following example encrypts data by using a [hybrid cryptosystem](#) consisting of AES GCM and OAEP, using their default parameter sizes and an AES key size of 128 bits.

OAEP is less vulnerable to padding oracle attacks than PKCS#1 v1.5 padding. GCM is also protected against padding oracle attacks.

Decryption can be performed by first retrieving the length of the encapsulated key and then by retrieving the encapsulated key. The encapsulated key can then be decrypted using the RSA private key that forms a key pair with the public key. After that the AES/GCM encrypted ciphertext can be decrypted to the original plaintext.

The protocol consists of:

1. a length field for the wrapped key (RSAPrivateKey misses a getKeySize() method);
2. the wrapped/encapsulated key, of the same size as the RSA key size in bytes;
3. the GCM ciphertext and 128 bit authentication tag (automatically added by Java).

Notes:

- To correctly use this code you should supply an RSA key of at least 2048 bits, bigger is better (but slower, especially during decryption);
- To use AES-256 you should install the [unlimited cryptography policy files](#) first;
- Instead creating your own protocol you might want to use a container format such as the Cryptographic Message Syntax (CMS / PKCS#7) or PGP instead.

So here's the example:

```
/**  
 * Encrypts the data using a hybrid crypto-system which uses GCM to encrypt the data and OAEP to  
 * encrypt the AES key.  
 * The key size of the AES encryption will be 128 bit.  
 * All the default parameter choices are used for OAEP and GCM.  
 *  
 * @param publicKey the RSA public key used to wrap the AES key  
 * @param plaintext the plaintext to be encrypted, not altered  
 * @return the ciphertext  
 * @throws InvalidKeyException if the key is not an RSA public key  
 * @throws NullPointerException if the plaintext is null  
 */  
public static byte[] encryptData(PublicKey publicKey, byte[] plaintext)  
    throws InvalidKeyException, NullPointerException {  
  
    // --- create the RSA OAEP cipher ---  
  
    Cipher oaep;  
    try {  
        // SHA-1 is the default and not vulnerable in this setting  
        // use OAEPParameterSpec to configure more than just the hash  
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");  
    } catch (NoSuchAlgorithmException e) {  
        throw new RuntimeException(  
            "Runtime doesn't have support for RSA cipher (mandatory algorithm for runtimes)",  
            e);  
    }
```

```

} 捕获 (NoSuchPaddingException e) {
    throw new RuntimeException(
        "运行时不支持OAEP填充 (自标准Java
运行时XX版本起支持) ", e);
}
oaep.初始化(Cipher.WRAP_MODE, 公钥);

// --- 将明文包装到缓冲区中

// 如果明文为null, 将抛出NullPointerException
ByteBuffer 明文缓冲区 = ByteBuffer.包装(明文);

// --- 生成新的AES密钥 ---

KeyGenerator aes密钥生成器;
try {
    aes密钥生成器 = KeyGenerator.获取实例("AES");
} 捕获 (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "运行时不支持AES密钥生成器 (运行时必备算法) ", e);
}

// 对于AES-192和256, 请确保您拥有相应权限 (安装
// 无限制加密策略文件)
aesKeyGenerator.init(128);
SecretKey aesKey = aesKeyGenerator.generateKey();

// --- 包装新的 AES 秘钥 ---

byte[] wrappedKey;
try {
    wrappedKey = oaep.wrap(aesKey);
} catch (IllegalBlockSizeException e) {
    throw new RuntimeException(
        "AES 密钥应始终适配使用正常大小 RSA 密钥的 OAEP", e);
}

// --- 设置 AES GCM 密码模式 ---

Cipher aesGCM;
try {
    aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
    // 由于密钥是随机生成的, 我们可以使用全零随机数
    // 128 位是标签大小的推荐 (最大) 值
    // 12 字节 (96 位) 是 GCM 模式加密的默认随机数大小
    GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128, new byte[12]);
    aesGCM.init(Cipher.ENCRYPT_MODE, aesKey, staticParameterSpec);
} 捕获 (NoSuchAlgorithmException e) {
    抛出新的 RuntimeException(
        "运行时不支持 AES 加密算法 (运行时的必备算法) ",
        e);
} 捕获 (NoSuchPaddingException e) {
    throw new RuntimeException(
        "运行时不支持 GCM (自标准 Java 运行时版本
XX 起支持) ", e);
} 捕获 (InvalidAlgorithmParameterException e) {
    throw new RuntimeException(
        "此 GCM 实现不接受 IvParameterSpec", e);
}

// --- 为我们自己的协议创建一个合适大小的缓冲区 ---

```

```

} 捕获 (NoSuchPaddingException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for OAEP padding (present in the standard Java
runtime sinze XX)", e);
}
oaep.init(Cipher.WRAP_MODE, publicKey);

// --- wrap the plaintext in a buffer

// will throw NullPointerException if plaintext is null
ByteBuffer plaintextBuffer = ByteBuffer.wrap(plaintext);

// --- generate a new AES secret key ---

KeyGenerator aesKeyGenerator;
try {
    aesKeyGenerator = KeyGenerator.getInstance("AES");
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for AES key generator (mandatory algorithm for
runtimes)", e);
}
// for AES-192 and 256 make sure you've got the rights (install the
// Unlimited Crypto Policy files)
aesKeyGenerator.init(128);
SecretKey aesKey = aesKeyGenerator.generateKey();

// --- wrap the new AES secret key ---

byte[] wrappedKey;
try {
    wrappedKey = oaep.wrap(aesKey);
} catch (IllegalBlockSizeException e) {
    throw new RuntimeException(
        "AES key should always fit OAEP with normal sized RSA key", e);
}

// --- setup the AES GCM cipher mode ---

Cipher aesGCM;
try {
    aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
    // we can get away with a zero nonce since the key is randomly generated
    // 128 bits is the recommended (maximum) value for the tag size
    // 12 bytes (96 bits) is the default nonce size for GCM mode encryption
    GCMParameterSpec staticParameterSpec = new GCMParameterSpec(128, new byte[12]);
    aesGCM.init(Cipher.ENCRYPT_MODE, aesKey, staticParameterSpec);
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for AES cipher (mandatory algorithm for runtimes)",
        e);
} catch (NoSuchPaddingException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for GCM (present in the standard Java runtime sinze
XX)", e);
} catch (InvalidAlgorithmParameterException e) {
    throw new RuntimeException(
        "IvParameterSpec not accepted by this implementation of GCM", e);
}

// --- create a buffer of the right size for our own protocol ---

```

```

ByteBuffer ciphertextBuffer = ByteBuffer.allocate(
    Short.字节数
    +oaep.获取输出大小(128 / Byte.大小)
    +aesGCM.获取输出大小(plaintext.长度));

// - 元素1：确保我们知道封装密钥的大小
ciphertextBuffer.putShort((short) wrappedKey.length);

// - 元素2：放入封装密钥
ciphertextBuffer.put(wrappedKey);

// - 元素3：使用 GCM 加密到缓冲区
try {
    aesGCM.doFinal(plaintextBuffer, ciphertextBuffer);
} catch (ShortBufferException | IllegalBlockSizeException | BadPaddingException e) {
    throw new RuntimeException("加密异常，AES/GCM 加密不应在此失败", e);
}

return ciphertextBuffer.array();
}

```

当然，没有解密，加密并没有太大用处。请注意，如果解密失败，将返回最少的信息。

```

/**
 * 使用混合加密系统解密数据，该系统使用 GCM 加密
 * 数据，使用 OAEP 加密 AES 密钥。OAEP 和 GCM 均使用所有默认参数
 * 选项。
 *
 * @param privateKey
 *      用于解包AES密钥的RSA私钥
 * @param ciphertext
 *      要加密的密文，未被修改
 * @return 明文
 * @throws InvalidKeyException
 *      如果密钥不是RSA私钥
 * @throws NullPointerException
 *      如果密文为null
 * @throws IllegalArgumentException
 *      如果密文无效，则抛出消息为"Invalid ciphertext"的异常（尽量减少信息泄露）
 */
public static byte[] decryptData(PrivateKey privateKey, byte[] ciphertext)
    throws InvalidKeyException, NullPointerException {

    // --- 创建 RSA OAEP 密码器 ---
}

Cipher oaep;
try {
    // SHA-1 是默认且在此设置下不易受攻击
    // 使用 OAEPParameterSpec 配置不仅仅是哈希算法
    oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "运行时不支持RSA密码（运行时的必备算法）",
        e);
} 捕获 (NoSuchPaddingException e) {
    throw new RuntimeException(
        "运行时不支持OAEP填充（自XX版本的标准Java运行时起支持）",
        e);
}

```

```

    oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "运行时不支持RSA密码（运行时的必备算法）",
        e);
} 捕获 (NoSuchPaddingException e) {
    throw new RuntimeException(
        "运行时不支持OAEP填充（自XX版本的标准Java运行时起支持）",
        e);
}

```

```

ByteBuffer ciphertextBuffer = ByteBuffer.allocate(
    Short.BYTES
    + oaep.getOutputSize(128 / Byte.SIZE)
    + aesGCM.getOutputSize(plaintext.length));

// - element 1: make sure that we know the size of the wrapped key
ciphertextBuffer.putShort((short) wrappedKey.length);

// - element 2: put in the wrapped key
ciphertextBuffer.put(wrappedKey);

// - element 3: GCM encrypt into buffer
try {
    aesGCM.doFinal(plaintextBuffer, ciphertextBuffer);
} catch (ShortBufferException | IllegalBlockSizeException | BadPaddingException e) {
    throw new RuntimeException("Cryptographic exception, AES/GCM encryption should not fail here", e);
}

return ciphertextBuffer.array();
}

```

Of course, encryption is not very useful without decryption. Note that this will return minimal information if decryption fails.

```

/**
 * Decrypts the data using a hybrid crypto-system which uses GCM to encrypt
 * the data and OAEP to encrypt the AES key. All the default parameter
 * choices are used for OAEP and GCM.
 *
 * @param privateKey
 *      the RSA private key used to unwrap the AES key
 * @param ciphertext
 *      the ciphertext to be encrypted, not altered
 * @return the plaintext
 * @throws InvalidKeyException
 *      if the key is not an RSA private key
 * @throws NullPointerException
 *      if the ciphertext is null
 * @throws IllegalArgumentException
 *      with the message "Invalid ciphertext" if the ciphertext is invalid (minimize
 *      information leakage)
 */
public static byte[] decryptData(PrivateKey privateKey, byte[] ciphertext)
    throws InvalidKeyException, NullPointerException {

    // --- create the RSA OAEP cipher ---

    Cipher oaep;
    try {
        // SHA-1 is the default and not vulnerable in this setting
        // use OAEPParameterSpec to configure more than just the hash
        oaep = Cipher.getInstance("RSA/ECB/OAEPwithSHA1andMGF1Padding");
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(
            "Runtime doesn't have support for RSA cipher (mandatory algorithm for runtimes)",
            e);
    } catch (NoSuchPaddingException e) {
        throw new RuntimeException(
            "Runtime doesn't have support for OAEP padding (present in the standard Java
            runtime since XX)",
            e);
    }
}

```

```

e);
}
oaep.init(Cipher.UNWRAP_MODE, privateKey);

// --- 将密文包装到缓冲区中

// 如果密文为null, 将抛出NullPointerException
ByteBuffer ciphertextBuffer = ByteBuffer.wrap(ciphertext);

// 合理性检查 #1
if (ciphertextBuffer.remaining() < 2) {
    throw new IllegalArgumentException("无效的密文");
}
// - 元素1：封装密钥的长度
int wrappedKeySize = ciphertextBuffer.getShort() & 0xFFFF;
// 合理性检查 #2
if (ciphertextBuffer.remaining() < wrappedKeySize + 128 / Byte.SIZE) {
    throw new IllegalArgumentException("无效的密文");
}

// --- 解包AES秘密密钥 ---

byte[] wrappedKey = new byte[wrappedKeySize];
// - 元素2：封装的密钥
ciphertextBuffer.get(wrappedKey);
SecretKey aesKey;
try {
    aesKey = (SecretKey) oaep.unwrap(wrappedKey, "AES",
        Cipher.SECRET_KEY);
} 捕获 (NoSuchAlgorithmException e) {
    抛出新的 RuntimeException(
        "运行时不支持AES密码 (运行时的强制算法) ",
        e);
} catch (InvalidKeyException e) {
    throw new RuntimeException(
        "无效的密文");
}

// --- 设置 AES GCM 密码模式 ---

Cipher aesGCM;
try {
    aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
    // 由于密钥是随机生成的, 我们可以使用零nonce

    // 128位是标签大小的推荐 (最大) 值// 12字节 (96位) 是GCM模式加密的默认nonce大小
    GCMPParameterSpec staticParameterSpec = new GCMPParameterSpec(128,
        new byte[12]);
    aesGCM.init(Cipher.DECRYPT_MODE, aesKey, staticParameterSpec);
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "运行时不支持AES密码 (运行时的强制算法) ",
        e);
} 捕获 (NoSuchPaddingException e) {
    throw new RuntimeException(
        "运行时不支持GCM (自XX起标准Java运行时中存在) ",
        e);
} 捕获 (InvalidAlgorithmParameterException e) {
    throw new RuntimeException(

```

```

        e);
}
oaep.init(Cipher.UNWRAP_MODE, privateKey);

// --- wrap the ciphertext in a buffer

// will throw NullPointerException if ciphertext is null
ByteBuffer ciphertextBuffer = ByteBuffer.wrap(ciphertext);

// sanity check #1
if (ciphertextBuffer.remaining() < 2) {
    throw new IllegalArgumentException("Invalid ciphertext");
}
// - element 1: the length of the encapsulated key
int wrappedKeySize = ciphertextBuffer.getShort() & 0xFFFF;
// sanity check #2
if (ciphertextBuffer.remaining() < wrappedKeySize + 128 / Byte.SIZE) {
    throw new IllegalArgumentException("Invalid ciphertext");
}

// --- unwrap the AES secret key ---

byte[] wrappedKey = new byte[wrappedKeySize];
// - element 2: the encapsulated key
ciphertextBuffer.get(wrappedKey);
SecretKey aesKey;
try {
    aesKey = (SecretKey) oaep.unwrap(wrappedKey, "AES",
        Cipher.SECRET_KEY);
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for AES cipher (mandatory algorithm for runtimes)",
        e);
} catch (InvalidKeyException e) {
    throw new RuntimeException(
        "Invalid ciphertext");
}

// --- setup the AES GCM cipher mode ---

Cipher aesGCM;
try {
    aesGCM = Cipher.getInstance("AES/GCM/Nopadding");
    // we can get away with a zero nonce since the key is randomly
    // generated
    // 128 bits is the recommended (maximum) value for the tag size
    // 12 bytes (96 bits) is the default nonce size for GCM mode
    // encryption
    GCMPParameterSpec staticParameterSpec = new GCMPParameterSpec(128,
        new byte[12]);
    aesGCM.init(Cipher.DECRYPT_MODE, aesKey, staticParameterSpec);
} catch (NoSuchAlgorithmException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for AES cipher (mandatory algorithm for runtimes)",
        e);
} catch (NoSuchPaddingException e) {
    throw new RuntimeException(
        "Runtime doesn't have support for GCM (present in the standard Java runtime sinze
        XX)",
        e);
} catch (InvalidAlgorithmParameterException e) {
    throw new RuntimeException(

```

```

    "IvParameterSpec 不被此 GCM 实现接受",
    e);
}

// --- 为我们自己的协议创建一个合适大小的缓冲区 ---

ByteBuffer 明文缓冲区 = ByteBuffer.allocate(aesGCM
    .getOutputSize(ciphertextBuffer.remaining()));

// - 元素 3: GCM 密文
try {
aesGCM.doFinal(ciphertextBuffer, plaintextBuffer);
} catch (ShortBufferException | IllegalBlockSizeException
    | BadPaddingException e) {
    throw new RuntimeException(
        "无效的密文");
}

return plaintextBuffer.array();
}

```

```

    "IvParameterSpec not accepted by this implementation of GCM",
    e);
}

// --- create a buffer of the right size for our own protocol ---

ByteBuffer plaintextBuffer = ByteBuffer.allocate(aesGCM
    .getOutputSize(ciphertextBuffer.remaining()));

// - element 3: GCM ciphertext
try {
    aesGCM.doFinal(ciphertextBuffer, plaintextBuffer);
} catch (ShortBufferException | IllegalBlockSizeException
    | BadPaddingException e) {
    throw new RuntimeException(
        "Invalid ciphertext");
}

return plaintextBuffer.array();
}

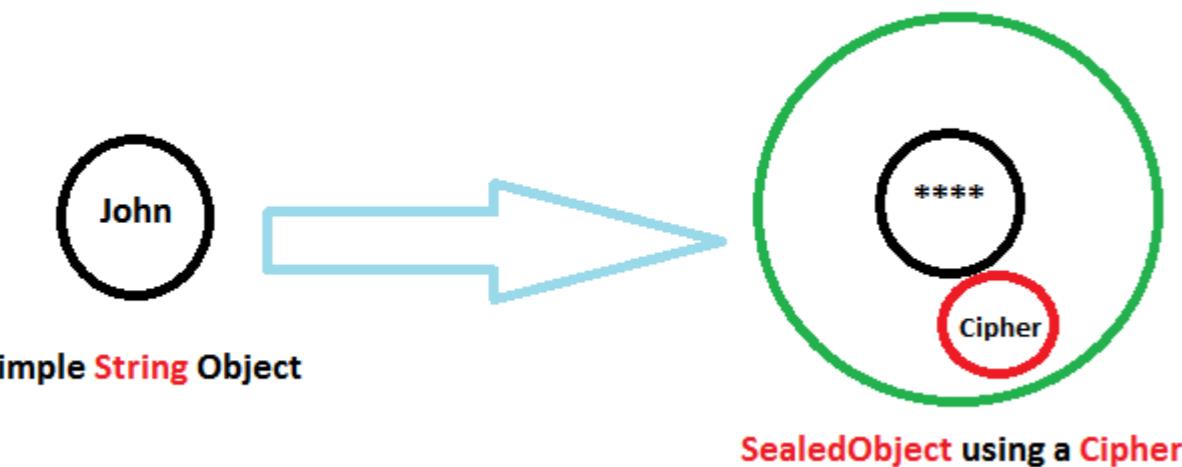
```

第141章：安全对象

第141.1节：SealedObject (javax.crypto.SealedObject)

此类使程序员能够创建一个对象，并使用加密算法保护其机密性。

给定任何可序列化对象，都可以创建一个SealedObject，该对象封装了原始对象的序列化格式（即“深拷贝”），并使用诸如AES、DES等加密算法对其序列化内容进行加密，以保护其机密性。加密内容随后可以使用相应算法和正确的解密密钥进行解密，并反序列化，从而得到原始对象。



```
Serializable obj = new String("John");
// 生成密钥
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

第141.2节：SignedObject (java.security.SignedObject)

SignedObject是一个用于创建真实运行时对象的类，其完整性无法被破坏而不被检测到。

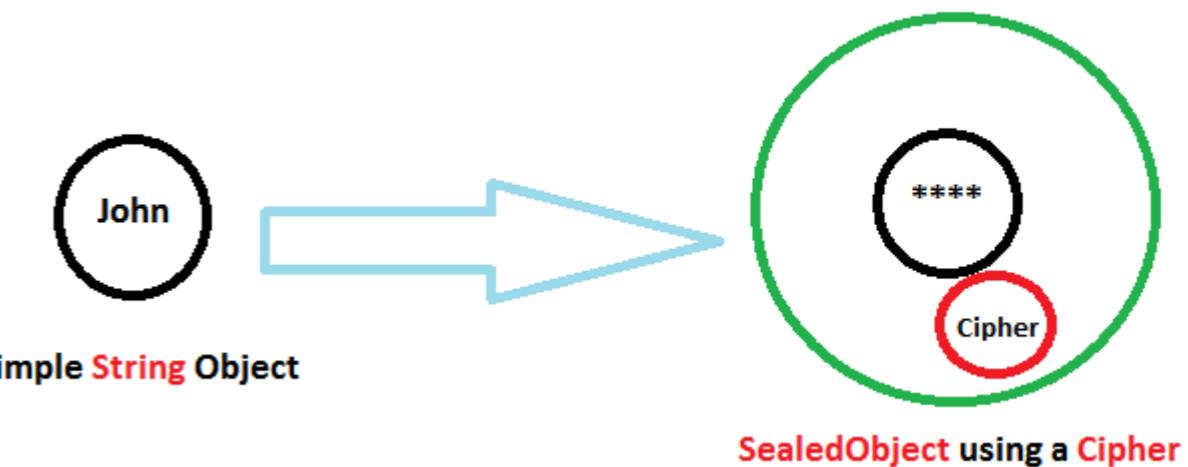
更具体地说，SignedObject包含另一个可序列化对象，即（待）签名对象及其签名。

Chapter 141: Secure objects

Section 141.1: SealedObject (javax.crypto.SealedObject)

This class enables a programmer to create an object and protect its confidentiality with a cryptographic algorithm.

Given any Serializable object, one can create a **SealedObject** that encapsulates the original object, in serialized format (i.e., a "deep copy"), and seals (encrypts) its serialized contents, using a cryptographic algorithm such as AES, DES, to protect its confidentiality. The encrypted content can later be decrypted (with the corresponding algorithm using the correct decryption key) and de-serialized, yielding the original object.

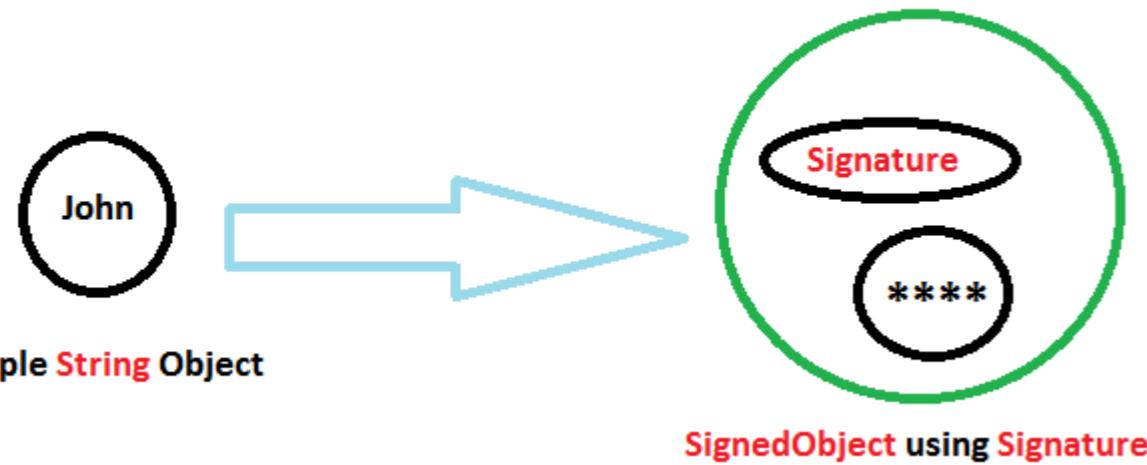


```
Serializable obj = new String("John");
// Generate key
KeyGenerator kgen = KeyGenerator.getInstance("AES");
kgen.init(128);
SecretKey aesKey = kgen.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, aesKey);
SealedObject sealedObject = new SealedObject(obj, cipher);
System.out.println("sealedObject-" + sealedObject);
System.out.println("sealedObject Data-" + sealedObject.getObject(aesKey));
```

Section 141.2: SignedObject (java.security.SignedObject)

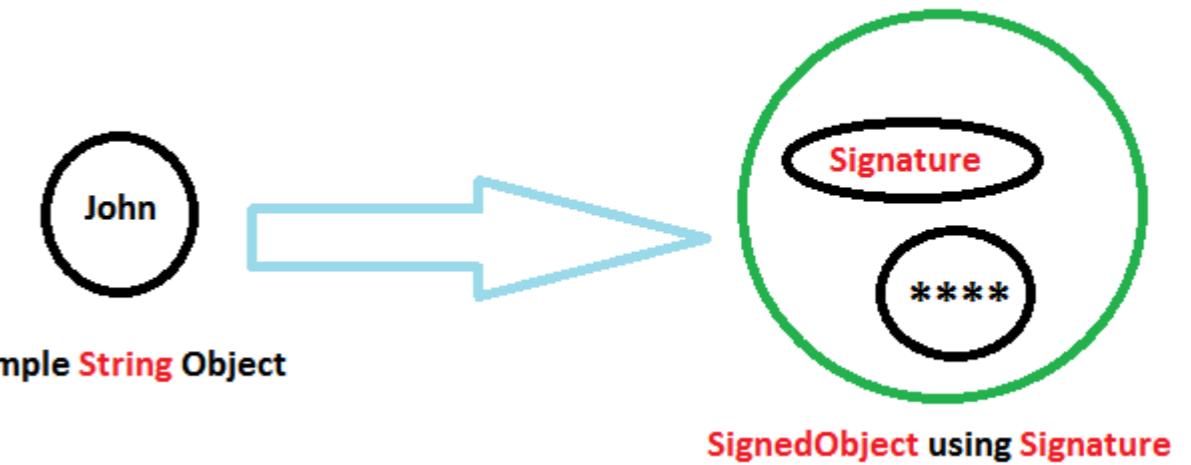
SignedObject is a class for the purpose of creating authentic runtime objects whose integrity cannot be compromised without being detected.

More specifically, a SignedObject contains another Serializable object, the (to-be-)signed object and its signature.



```
//创建一个密钥
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
// 创建私钥
PrivateKey signingKey = keyGen.generateKeyPair().getPrivate();
// 创建签名对象
Signature signingEngine = Signature.getInstance("DSA");
signingEngine.initSign(signingKey);
// 创建一个简单对象
Serializable obj = new String("John");
// 对我们的对象进行签名
SignedObject signedObject = new SignedObject(obj, signingKey, signingEngine);

System.out.println("signedObject-" + signedObject);
System.out.println("signedObject Data-" + signedObject.getObject());
```



```
//Create a key
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG", "SUN");
keyGen.initialize(1024, random);
// create a private key
PrivateKey signingKey = keyGen.generateKeyPair().getPrivate();
// create a Signature
Signature signingEngine = Signature.getInstance("DSA");
signingEngine.initSign(signingKey);
// create a simple object
Serializable obj = new String("John");
// sign our object
SignedObject signedObject = new SignedObject(obj, signingKey, signingEngine);

System.out.println("signedObject-" + signedObject);
System.out.println("signedObject Data-" + signedObject.getObject());
```

第142章：安全与密码学

第142.1节：计算密码哈希

使用不同算法计算相对较小数据块的哈希值：

```
final MessageDigest md5 = MessageDigest.getInstance("MD5");
final MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
final MessageDigest sha256 = MessageDigest.getInstance("SHA-256");

final byte[] data = "FOO BAR".getBytes();

System.out.println("MD5      hash: " + DatatypeConverter.printHexBinary(md5.digest(data)));
System.out.println("SHA1     hash: " + DatatypeConverter.printHexBinary(sha1.digest(data)));
System.out.println("SHA256   hash: " + DatatypeConverter.printHexBinary(sha256.digest(data)));
```

产生以下输出：

```
MD5      hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1     hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256   hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

根据您所使用的Java平台实现，可能会提供其他算法。

第142.2节：使用公钥/私钥加密和解密数据

使用公钥加密数据：

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
rsa.update(message.getBytes());
final byte[] result = rsa.doFinal();

System.out.println("Message: " + message);
System.out.println("Encrypted: " + DatatypeConverter.printHexBinary(result));
```

输出类似于：

```
消息: Hello
加密后: 5641FBB9558ECFA9ED...
```

注意，在创建Cipher对象时，必须指定与所使用密钥类型兼容的转换方式。（参见JCA标准算法名称以获取支持的转换列表。）对于RSA加密，[数据message.getBytes\(\)](#)的长度必须小于密钥大小。详情请参见此SO回答。

解密数据：

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
rsa.update(cipherText);
final String result = new String(rsa.doFinal());
```

Chapter 142: Security & Cryptography

Section 142.1: Compute Cryptographic Hashes

To compute the hashes of relatively small blocks of data using different algorithms:

```
final MessageDigest md5 = MessageDigest.getInstance("MD5");
final MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
final MessageDigest sha256 = MessageDigest.getInstance("SHA-256");

final byte[] data = "FOO BAR".getBytes();

System.out.println("MD5      hash: " + DatatypeConverter.printHexBinary(md5.digest(data)));
System.out.println("SHA1     hash: " + DatatypeConverter.printHexBinary(sha1.digest(data)));
System.out.println("SHA256   hash: " + DatatypeConverter.printHexBinary(sha256.digest(data)));
```

Produces this output:

```
MD5      hash: E99E768582F6DD5A3BA2D9C849DF736E
SHA1     hash: 0135FAA6323685BA8A8FF8D3F955F0C36949D8FB
SHA256   hash: 8D35C97BCD902B96D1B551741BBE8A7F50BB5A690B4D0225482EAA63DBFB9DED
```

Additional algorithms may be available depending on your implementation of the Java platform.

Section 142.2: Encrypt and Decrypt Data with Public / Private Keys

To encrypt data with a public key:

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());
rsa.update(message.getBytes());
final byte[] result = rsa.doFinal();

System.out.println("Message: " + message);
System.out.println("Encrypted: " + DatatypeConverter.printHexBinary(result));
```

Produces output similar to:

```
Message: Hello
Encrypted: 5641FBB9558ECFA9ED...
```

Note that when creating the Cipher object, you have to specify a transformation that is compatible with the type of key being used. (See [JCA Standard Algorithm Names](#) for a list of supported transformations.). For RSA encryption data [message.getBytes\(\)](#) length must be smaller than the key size. See this [SO Answer](#) for detail.

To decrypt the data:

```
final Cipher rsa = Cipher.getInstance("RSA");

rsa.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
rsa.update(cipherText);
final String result = new String(rsa.doFinal());
```

```
System.out.println("Decrypted: " + result);
```

输出如下内容：

解密结果: Hello

第142.3节：生成密码学随机数据

生成密码学随机数据样本：

```
final byte[] sample = new byte[16];
new SecureRandom().nextBytes(sample);
System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

输出类似于：

样本：E4F14CEA2384F70B706B53A6DF8C5EFE

请注意，根据所使用的算法，调用 `nextBytes()` 可能在收集熵时阻塞。

指定算法和提供者的方法：

```
final byte[] sample = new byte[16];
final SecureRandom randomness = SecureRandom.getInstance("SHA1PRNG", "SUN");
randomness.nextBytes(sample);

System.out.println("提供者: " + randomness.getProvider());
System.out.println("算法: " + randomness.getAlgorithm());
System.out.println("样本: " + DatatypeConverter.printHexBinary(sample));
```

输出类似于：

提供者: SUN 版本 1.8

算法: SHA1PRNG

样本: C80C44BAEB352FD29FBBE20489E4C0B9

第142.4节：生成公钥/私钥对

使用不同算法和密钥大小生成密钥对：

```
final KeyPairGenerator dhGenerator = KeyPairGenerator.getInstance("DiffieHellman");
final KeyPairGenerator dsaGenerator = KeyPairGenerator.getInstance("DSA");
final KeyPairGenerator rsaGenerator = KeyPairGenerator.getInstance("RSA");

dhGenerator.initialize(1024);
dsaGenerator.initialize(1024);
rsaGenerator.initialize(2048);

final KeyPair dhPair = dhGenerator.generateKeyPair();
final KeyPair dsaPair = dsaGenerator.generateKeyPair();
final KeyPair rsaPair = rsaGenerator.generateKeyPair();
```

```
System.out.println("Decrypted: " + result);
```

Produces the following output:

Decrypted: Hello

Section 142.3: Generate Cryptographically Random Data

To generate samples of cryptographically random data:

```
final byte[] sample = new byte[16];
new SecureRandom().nextBytes(sample);
System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Produces output similar to:

Sample: E4F14CEA2384F70B706B53A6DF8C5EFE

Note that the call to `nextBytes()` may block while entropy is gathered depending on the algorithm being used.

To specify the algorithm and provider:

```
final byte[] sample = new byte[16];
final SecureRandom randomness = SecureRandom.getInstance("SHA1PRNG", "SUN");
randomness.nextBytes(sample);

System.out.println("Provider: " + randomness.getProvider());
System.out.println("Algorithm: " + randomness.getAlgorithm());
System.out.println("Sample: " + DatatypeConverter.printHexBinary(sample));
```

Produces output similar to:

Provider: SUN version 1.8
Algorithm: SHA1PRNG
Sample: C80C44BAEB352FD29FBBE20489E4C0B9

Section 142.4: Generate Public / Private Key Pairs

To generate key pairs using different algorithms and key sizes:

```
final KeyPairGenerator dhGenerator = KeyPairGenerator.getInstance("DiffieHellman");
final KeyPairGenerator dsaGenerator = KeyPairGenerator.getInstance("DSA");
final KeyPairGenerator rsaGenerator = KeyPairGenerator.getInstance("RSA");

dhGenerator.initialize(1024);
dsaGenerator.initialize(1024);
rsaGenerator.initialize(2048);

final KeyPair dhPair = dhGenerator.generateKeyPair();
final KeyPair dsaPair = dsaGenerator.generateKeyPair();
final KeyPair rsaPair = rsaGenerator.generateKeyPair();
```

您的 Java 平台实现可能支持其他算法和密钥大小。

指定生成密钥时使用的随机源：

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));
final KeyPair pair = generator.generateKeyPair();
```

第142.5节：计算和验证数字签名

计算签名：

```
final PrivateKey privateKey = keyPair.getPrivate();
final byte[] data = "FOO BAR".getBytes();
final Signature signer = Signature.getInstance("SHA1withRSA");

signer.initSign(privateKey);
signer.update(data);

final byte[] signature = signer.sign();
```

注意，签名算法必须与用于生成密钥对的算法兼容。

验证签名：

```
final PublicKey publicKey = keyPair.getPublic();
final Signature verifier = Signature.getInstance("SHA1withRSA");

verifier.initVerify(publicKey);
verifier.update(data);

System.out.println("Signature: " + verifier.verify(signature));
```

产生以下输出：

签名：true

Additional algorithms and key sizes may be available on your implementation of the Java platform.

To specify a source of randomness to use when generating the keys:

```
final KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
generator.initialize(2048, SecureRandom.getInstance("SHA1PRNG", "SUN"));
final KeyPair pair = generator.generateKeyPair();
```

Section 142.5: Compute and Verify Digital Signatures

To compute a signature:

```
final PrivateKey privateKey = keyPair.getPrivate();
final byte[] data = "FOO BAR".getBytes();
final Signature signer = Signature.getInstance("SHA1withRSA");

signer.initSign(privateKey);
signer.update(data);

final byte[] signature = signer.sign();
```

Note that the signature algorithm must be compatible with the algorithm used to generate the key pair.

To verify a signature:

```
final PublicKey publicKey = keyPair.getPublic();
final Signature verifier = Signature.getInstance("SHA1withRSA");

verifier.initVerify(publicKey);
verifier.update(data);

System.out.println("Signature: " + verifier.verify(signature));
```

Produces this output:

Signature: true

第143章：安全与密码学

Java中的安全实践大致可以分为两个宽泛且定义模糊的类别：Java平台安全和安全的Java编程。

Java平台安全实践涉及管理JVM的安全性和完整性。它包括管理JCE提供者和安全策略等主题。

安全的Java编程实践关注编写安全Java程序的最佳方法。它包括使用随机数和密码学以及防止漏洞等主题。

第143.1节：JCE

Java密码扩展（JCE）是内置于JVM的一个框架，允许开发者轻松且安全地在程序中使用密码学。它通过为程序员提供一个简单、可移植的接口，同时使用JCE提供者系统来安全地实现底层的密码操作。

第143.2节：密钥及密钥管理

虽然JCE保障了密码操作和密钥生成的安全，但实际管理密钥的责任在于开发者。这里需要提供更多信息。

一种被广泛接受的运行时密钥处理最佳实践是仅将密钥存储为byte数组，绝不以字符串形式存储。这是因为Java字符串是不可变的，无法手动在内存中“清除”或“置零”；虽然可以移除对字符串的引用，但该字符串的具体内容会一直保留在内存中，直到其内存段被垃圾回收并重用。攻击者有很长的时间窗口可以转储程序内存并轻松找到密钥。相反，byte数组是可变的，可以原地覆盖其内容；因此，一旦不再需要密钥，最好立即将其“置零”。

第143.3节：常见的Java漏洞

第143.4节：网络问题

第143.5节：随机性与您

对于大多数应用来说，`java.utils.Random`类是一个完全合适的“随机”数据来源。如果您需要从数组中选择一个随机元素，或生成一个随机字符串，或创建一个临时的“唯一”标识符，您可能应该使用`Random`。

然而，许多加密系统依赖随机性来保证其安全性，而`Random`提供的随机性质量并不够高。

对于任何需要随机输入的加密操作，您应该改用`SecureRandom`。

第143.6节：哈希与验证

加密哈希函数属于一类具有三个重要属性的函数；一致性、唯一性和不可逆性。

一致性：给定相同的数据，哈希函数总是返回相同的值。也就是说，如果 $X = Y$ ，则对于哈希函数 f ， $f(x)$ 总是等于 $f(y)$ 。

唯一性：没有两个不同的输入会导致哈希函数产生相同的输出。也就是说，如果 $X \neq Y$ ，则对于任何

Chapter 143: Security & Cryptography

Security practices in Java can be separated into two broad, vaguely defined categories; Java platform security, and secure Java programming.

Java platform security practices deal with managing the security and integrity of the JVM. It includes such topics as managing JCE providers and security policies.

Secure Java programming practices concern the best ways to write secure Java programs. It includes such topics as using random numbers and cryptography, and preventing vulnerabilities.

Section 143.1: The JCE

The Java Cryptography Extension (JCE) is a framework built into the JVM to allow developers to easily and securely use cryptography in their programs. It does this by providing a simple, portable interface to programmers, while using a system of JCE Providers to securely implement the underlying cryptographic operations.

Section 143.2: Keys and Key Management

While the JCE secures cryptographic operations and key generation, it is up to the developer to actually manage their keys. More information needs to be provided here.

One commonly-accepted best practice for handling keys at runtime is to store them only as `byte` arrays, and never as strings. This is because Java strings are immutable, and cannot be manually "cleared" or "zeroed out" in memory; while a reference to a string can be removed, the exact string will remain in memory until its segment of memory is garbage-collected and reused. An attacker would have a large window in which they could dump the program's memory and easily find the key. Contrarily, `byte` arrays are mutable, and can have their contents overwritten in place; it is a good idea to 'zero-out' your keys as soon as you no longer need them.

Section 143.3: Common Java vulnerabilities

Section 143.4: Networking Concerns

Section 143.5: Randomness and You

For most applications, the `java.utils.Random` class is a perfectly fine source of "random" data. If you need to choose a random element from an array, or generate a random string, or create a temporary "unique" identifier, you should probably use `Random`.

However, many cryptographic systems rely on randomness for their security, and the randomness provided by `Random` simply isn't of high enough quality. For any cryptographic operation that requires a random input, you should use `SecureRandom` instead.

Section 143.6: Hashing and Validation

A cryptographic hash function is a member of a class of functions with three vital properties; consistency, uniqueness, and irreversibility.

Consistency: Given the same data, a hash function will always return the same value. That is, if $X = Y$, $f(x)$ will always equal $f(y)$ for hash function f .

Uniqueness: No two inputs to a hash function will ever result in the same output. That is, if $X \neq Y$, $f(x) \neq f(y)$, for any

X 和 Y 的值。

不可逆性：“反转”哈希函数在实际操作中非常困难，甚至不可能。也就是说，给定仅有的 $f(X)$ ，不应该有任何方法能找到原始的 X，除非对所有可能的 X 值进行函数 f 的计算（暴力破解）。不应该存在函数 f_1 使得 $f_1(f(X)) = X$ 。

许多函数缺少至少一个这些属性。例如，MD5 和 SHA1 已知存在碰撞，即两个输入有相同的输出，因此它们缺乏唯一性。目前被认为安全的一些函数是 SHA-256 和 SHA-512。

values of X and Y.

Irreversibility: It is impractically difficult, if not impossible, to "reverse" a hash function. That is, given only $f(X)$, there should be no way of finding the original X short of putting every possible value of X through the function f (brute-force). There should be no function f_1 such that $f_1(f(X)) = X$.

Many functions lack at least one of these attributes. For example, MD5 and SHA1 are known to have collisions, i.e. two inputs that have the same output, so they lack uniqueness. Some functions that are currently believed to be secure are SHA-256 and SHA-512.

第144章：SecurityManager (安全管理器)

第144.1节：对由 ClassLoader 加载的类进行沙箱保护

ClassLoader 需要提供一个ProtectionDomain来标识代码的来源：

```
public class PluginClassLoader extends ClassLoader {
    private final ClassProvider provider;

    private final ProtectionDomain pd;

    public PluginClassLoader(ClassProvider provider) {
        this.provider = provider;
        Permissions permissions = new Permissions();

        this.pd = new ProtectionDomain(provider.getCodeSource(), permissions, this, null);
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classDef = provider.getClass(name);
        Class<?> clazz = defineClass(name, classDef, 0, classDef.length, pd);
        return clazz;
    }
}
```

通过重写findClass而非loadClass，保持了委托模型，PluginClassLoader将首先查询系统和父类加载器以获取类定义。

创建策略：

```
public class PluginSecurityPolicy extends Policy {
    private final Permissions appPermissions = new Permissions();
    private final Permissions pluginPermissions = new Permissions();

    public PluginSecurityPolicy() {
        // 根据需要修改此处
        appPermissions.add(new AllPermission());
        // 向pluginPermissions添加插件应拥有的任何权限
    }

    @Override
    public Provider getProvider() {
        return super.getProvider();
    }

    @Override
    public String getType() {
        return super.getType();
    }

    @Override
    public Parameters getParameters() {
        return super.getParameters();
    }

    @Override
```

Chapter 144: SecurityManager

Section 144.1: Sandboxing classes loaded by a ClassLoader

The ClassLoader needs to provide a [ProtectionDomain](#) identifying the source of the code:

```
public class PluginClassLoader extends ClassLoader {
    private final ClassProvider provider;

    private final ProtectionDomain pd;

    public PluginClassLoader(ClassProvider provider) {
        this.provider = provider;
        Permissions permissions = new Permissions();

        this.pd = new ProtectionDomain(provider.getCodeSource(), permissions, this, null);
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] classDef = provider.getClass(name);
        Class<?> clazz = defineClass(name, classDef, 0, classDef.length, pd);
        return clazz;
    }
}
```

By overriding `findClass` rather than `loadClass` the delegational model is preserved, and the `PluginClassLoader` will first query the system and parent classloader for class definitions.

Create a Policy:

```
public class PluginSecurityPolicy extends Policy {
    private final Permissions appPermissions = new Permissions();
    private final Permissions pluginPermissions = new Permissions();

    public PluginSecurityPolicy() {
        // amend this as appropriate
        appPermissions.add(new AllPermission());
        // add any permissions plugins should have to pluginPermissions
    }

    @Override
    public Provider getProvider() {
        return super.getProvider();
    }

    @Override
    public String getType() {
        return super.getType();
    }

    @Override
    public Parameters getParameters() {
        return super.getParameters();
    }

    @Override
```

```

public PermissionCollection getPermissions(CodeSource codesource) {
    return new Permissions();
}

@Override
public PermissionCollection getPermissions(ProtectionDomain domain) {
    return isPlugin(domain)?pluginPermissions:appPermissions;
}

private boolean isPlugin(ProtectionDomain pd){
    return pd.getClassLoader() instanceof PluginClassLoader;
}

}

```

最后，设置策略和安全管理器（默认实现即可）：

```

Policy.setPolicy(new PluginSecurityPolicy());
System.setSecurityManager(new SecurityManager());

```

第144.2节：启用安全管理器

Java虚拟机（JVM）可以安装安全管理器运行。安全管理器根据代码的加载来源和签名证书等因素，管理JVM中运行的代码允许执行的操作。

可以通过在启动JVM时在命令行设置java.security.manager系统属性来安装安全管理器：

```
java -Djava.security.manager <main class name>
```

或者通过Java代码以编程方式安装：

```
System.setSecurityManager(new SecurityManager())
```

标准Java安全管理器基于策略授予权限，策略定义在策略文件中。如果未指定策略文件，则使用位于\$JAVA_HOME/lib/security/java.policy下的默认策略文件。

第144.3节：实现策略拒绝规则

有时希望拒绝某个权限给某个保护域，无论该域获得了任何其他权限。此示例演示了满足此类需求的所有可能方法中的一种。它引入了一个“负面”权限类，以及一个包装器，使得默认的策略可以作为此类权限的存储库重复使用。

注意事项：

- 标准策略文件语法和权限分配机制总体上保持不变。这意味着策略文件中的拒绝规则仍然以授予的形式表达。
- 该策略包装器专门用于封装默认的基于文件的策略（假定为 com.sun.security.provider.PolicyFile）。
- 被拒绝的权限仅在策略层面被视为拒绝。如果静态分配给某个域，默认情况下该域将把它们视为普通的“正面”权限。

DeniedPermission类

```
包com.example;
```

```

public PermissionCollection getPermissions(CodeSource codesource) {
    return new Permissions();
}

@Override
public PermissionCollection getPermissions(ProtectionDomain domain) {
    return isPlugin(domain)?pluginPermissions:appPermissions;
}

private boolean isPlugin(ProtectionDomain pd){
    return pd.getClassLoader() instanceof PluginClassLoader;
}

}

```

Finally, set the policy and a SecurityManager (default implementation is fine):

```

Policy.setPolicy(new PluginSecurityPolicy());
System.setSecurityManager(new SecurityManager());

```

Section 144.2: Enabling the SecurityManager

Java Virtual Machines (JVMs) can be run with a SecurityManager installed. The SecurityManager governs what the code running in the JVM is allowed to do, based on factors such as where the code was loaded from and what certificates were used to sign the code.

The SecurityManager can be installed by setting the java.security.manager system property on the command line when starting the JVM:

```
java -Djava.security.manager <main class name>
```

or programmatically from within Java code:

```
System.setSecurityManager(new SecurityManager())
```

The standard Java SecurityManager grants permissions on the basis of a Policy, which is defined in a policy file. If no policy file is specified, the default policy file under \$JAVA_HOME/lib/security/java.policy will be used.

Section 144.3: Implementing policy deny rules

It is occasionally desirable to *deny* a certain [Permission](#) to some [ProtectionDomain](#), *regardless* of any other permissions that domain accrues. This example demonstrates just one of all the possible approaches for satisfying this kind of requirement. It introduces a "negative" permission class, along with a wrapper that enables the default [Policy](#) to be reused as a repository of such permissions.

Notes:

- The standard policy file syntax and mechanism for permission assignment in general remain unaffected. This means that *deny* rules within policy files are still expressed as *grants*.
- The policy wrapper is meant to specifically encapsulate the default file-backed [Policy](#) (assumed to be com.sun.security.provider.PolicyFile).
- Denied permissions are only processed as such at the policy level. If statically assigned to a domain, they will by default be treated by that domain as ordinary "positive" permissions.

The DeniedPermission class

```
package com.example;
```

```

导入java.lang.reflect.Constructor;
导入java.lang.reflect.InvocationTargetException;
导入java.lang.reflect.Modifier;
导入java.security.BasicPermission;
导入java.security.Permission;
导入java.security.UnresolvedPermission;
导入java.text.MessageFormat;

/**
 * "负面"权限的表示。
 * <p>
 * 一个 DeniedPermission, 当被"授予" (给某个 ProtectionDomain 和/或 * Principal) 时, 表示一种无论拥有任何正向权限 (包括 AllPermission) 都不能行使的特权。换句话说, 如果 * 一组被授予的权限, P, 包含了该类权限 D, 那么实际被授予的权限集合是<br/>
 * <br/>
 * &nbsp;&nbsp;&nbsp;<em>{ P<sub>implied</sub> - D<sub>implied</sub> }</em>.
 * </p>
 * <p>
 * 该类的每个实例封装了一个目标权限, 表示被拒绝的"正向"权限。
 * </p>
 * 被拒绝的权限采用以下命名规则:<br/>
 * <br/>
 *
&nbsp;&nbsp;&nbsp;<em>&lt;target_class_name&gt;:&lt;target_name&gt;(&lt;target_actions&gt;)</em><br/>

* <br/>
* 其中:
* <ul>
* <li><em>target_class_name</em> 是目标权限类的名称, </li>
* <li><em>target_name</em> 是目标权限的名称, 且</li>
* <li><em>target_actions</em> 是目标权限的操作字符串 (可选)。</li>
* </ul>
* 拒绝的权限, 具有目标权限 <em>t</em>, 如果满足以下条件, 则称其<em>隐含</em>另一个权限 <em>p</em>:
* <ul>
* <li>p <em>本身不是</em>拒绝的权限, 且 <code>(t.implies(p) == true)</code>,
* 或者</li>
* <li>p <em>是</em>拒绝的权限, 具有目标 <em>t1</em>, 且
* <code>(t.implies(t1) == true)</code>。
* </ul>
* <p>
* 策略决策点 (例如, Policy 提供者) 有责任在发布授权声明时考虑拒绝权限的语义。
* </p>
*/
public final class DeniedPermission extends BasicPermission {

    private final Permission target;
    private static final long serialVersionUID = 473625163869800679L;

    /**
     * 实例化一个 DeniedPermission, 封装指定类的目标权限, 指定名称和可选的操作。
     *
     * @throws IllegalArgumentException
     *          如果出现以下情况:
     *          <ul>
     *          <li><code>targetClassName</code> 为 <code>null</code>、空字符串、未指向具体的 <code>Permission</code> 子类, 或指向
    
```

```

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Modifier;
import java.security.BasicPermission;
import java.security.Permission;
import java.security.UnresolvedPermission;
import java.text.MessageFormat;

/**
 * A representation of a "negative" privilege.
 * <p>
 * A DeniedPermission, when "granted" (to some ProtectionDomain and/or
 * Principal), represents a privilege which cannot be exercised, regardless of
 * any positive permissions (AllPermission included) possessed. In other words, if a
 * set of granted permissions, P, contains a permission of this class, D, then the
 * set of effectively granted permissions is<br/>
 * <br/>
 * &nbsp;&nbsp;&nbsp;<em>{ P<sub>implied</sub> - D<sub>implied</sub> }</em>.
 * </p>
 * <p>
 * Each instance of this class encapsulates a target permission, representing the
 * "positive" permission being denied.
 * </p>
 * Denied permissions employ the following naming scheme:<br/>
 * <br/>
 *
&nbsp;&nbsp;&nbsp;<em>&lt;target_class_name&gt;:&lt;target_name&gt;(&lt;target_actions&gt;)</em><br/>
* <br/>
* where:
* <ul>
* <li><em>target_class_name</em> is the name of the target permission's class,</li>
* <li><em>target_name</em> is the name of the target permission, and</li>
* <li><em>target_actions</em> is, optionally, the actions string of the target permission.</li>
* </ul>
* A denied permission, having a target permission <em>t</em>, is said to <em>imply</em> another
* permission <em>p</em>, if:
* <ul>
* <li>p <em>is not</em> itself a denied permission, and <code>(t.implies(p) == true)</code>,
* or</li>
* <li>p <em>is</em> a denied permission, with a target <em>t1</em>, and
* <code>(t.implies(t1) == true)</code>.
* </ul>
* <p>
* It is the responsibility of the policy decision point (e.g., the Policy provider) to
* take denied permission semantics into account when issuing authorization statements.
* </p>
*/
public final class DeniedPermission extends BasicPermission {

    private final Permission target;
    private static final long serialVersionUID = 473625163869800679L;

    /**
     * Instantiates a DeniedPermission that encapsulates a target permission of the
     * indicated class, specified name and, optionally, actions.
     *
     * @throws IllegalArgumentException
     *          if:
     *          <ul>
     *          <li><code>targetClassName</code> is <code>null</code>, the empty string, does not
     *          refer to a concrete <code>Permission</code> descendant, or refers to
    
```

```

*           <code>DeniedPermission.class</code> 或
<code>UnresolvedPermission.class</code>。 </li>
*           <li><code>targetName</code> 为 <code>null</code>。 </li>      *
<li><code>targetClassName</code> 无法被实例化，且这是调用者的错误；

*
*           例如，因为 <code>targetName</code> 和/或 <code>targetActions</code> 不符合目标类的命名约束；或者由于目标类未公
开
*
*           <code>(String name)</code> 构造函数，或 <code>(String name, String
actions)</code>
*
*           构造函数，具体取决于 <code>targetActions</code> 是否为 <code>null</code>。
或
*
*           not.</li>
*
*           </ul>
*/
public static DeniedPermission newDeniedPermission(String targetClassName, String targetName,
                                                 String targetActions) {
    if (targetClassName == null || targetClassName.trim().isEmpty() || targetName == null) {
        throw new IllegalArgumentException(
            "传入了空或空字符串的[targetClassName]，或空的[targetName]参数。");
    }
    StringBuilder sb = new StringBuilder(targetClassName).append(":").append(targetName);
    if (targetName != null) {
        sb.append(":").append(targetName);
    }
    return new DeniedPermission(sb.toString());
}

/**
* 实例化一个<code>DeniedPermission</code>，封装了目标权限的
类名、
* 名称以及可选的操作，这些共同作为<code>name</code>参数提供。
*
* @throws IllegalArgumentException
*           如果出现以下情况：
*           <ul>
*           <li><code>name</code>的目标权限类名部分为空，未
*               指向具体的<code>Permission</code>子类，或指向
*               <code>DeniedPermission.class</code> 或
*               <code>UnresolvedPermission.class</code>。 </li>
*           <li><code>name</code> 的目标名称组件是 <code>empty</code></li>      *           <li>目
标权限类无法被实例化，且这是调用者的责任；

*
*           例如，因为 <code>name</code> 的目标名称和/或目标动作组件
不符合目标类的命名约束；或由于目标
类
*
*           不公开 <code>(String name)</code> 构造函数，或      *
<code>(String name, String actions)</code> 构造函数，具体取决于目标操作组件是否为空。
*
*           </ul>
*/
public DeniedPermission(String name) {
    super(name);
    String[] comps = name.split(":");
    if (comps.length < 2) {
        throw new IllegalArgumentException(MessageFormat.format("名称格式错误 [{0}]
参数。", name));
    }
    this.目标=initTarget(comps[0], comps[1], ((comps.length < 3) ? null : comps[2]));
}

```

```

        <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>. </li>
        * <li><code>targetName</code> is <code>null</code>. </li>
        * <li><code>targetClassName</code> cannot be instantiated, and it's the caller's
fault;
        * e.g., because <code>targetName</code> and/or <code>targetActions</code> do not
adhere
        * to the naming constraints of the target class; or due to the target class not
exposing a <code>(String name)</code>, or <code>(String name, String
actions)</code>
        *
        * constructor, depending on whether <code>targetActions</code> is <code>null</code>
or
        * not.</li>
        </ul>
    */
    public static DeniedPermission newDeniedPermission(String targetClassName, String targetName,
                                                    String targetActions) {
        if (targetClassName == null || targetClassName.trim().isEmpty() || targetName == null) {
            throw new IllegalArgumentException(
                    "Null or empty [targetClassName], or null [targetName] argument was
supplied.");
        }
        StringBuilder sb = new StringBuilder(targetClassName).append(":").append(targetName);
        if (targetName != null) {
            sb.append(":").append(targetName);
        }
        return new DeniedPermission(sb.toString());
    }

    /**
     * Instantiates a <code>DeniedPermission</code> that encapsulates a target permission of the
class,
     * name and, optionally, actions, collectively provided as the <code>name</code> argument.
     *
     * @throws IllegalArgumentException
     *         if:
     *         <ul>
     *             <li><code>name</code>'s target permission class name component is empty, does not
refer to a concrete <code>Permission</code> descendant, or refers to
     *             <code>DeniedPermission.class</code> or
<code>UnresolvedPermission.class</code>. </li>
     *             <li><code>name</code>'s target name component is <code>empty</code></li>
     *             <li>the target permission class cannot be instantiated, and it's the caller's
fault;
     *             e.g., because <code>name</code>'s target name and/or target actions component(s)
do
     * not adhere to the naming constraints of the target class; or due to the target
class
     * not exposing a <code>(String name)</code>, or
     * <code>(String name, String actions)</code> constructor, depending on whether the
target actions component is empty or not.</li>
     * </ul>
    */
    public DeniedPermission(String name) {
        super(name);
        String[] comps = name.split(":");
        if (comps.length < 2) {
            throw new IllegalArgumentException(MessageFormat.format("Malformed name [{0}]
argument.", name));
        }
        this.target = initTarget(comps[0], comps[1], ((comps.length < 3) ? null : comps[2]));
    }
}

```

```

/**
 * 实例化一个封装了给定目标权限的 <code>DeniedPermission</code>。
 *
 * @throws IllegalArgumentException
 * 如果 <code>target</code> 是 <code>null</code>、<code>DeniedPermission</code>, 或
 * 一个
 * <code>UnresolvedPermission</code>。
 */
public static DeniedPermission newDeniedPermission(Permission target) {
    if (target == null) {
        throw new IllegalArgumentException("Null [target] argument.");
    }
    if (target instanceof DeniedPermission || target instanceof UnresolvedPermission) {
        throw new IllegalArgumentException("[target] must not be a DeniedPermission or an
UnresolvedPermission.");
    }
    StringBuilder sb = new
    StringBuilder(target.getClass().getName()).append(":").append(target.getName());
    String targetActions = target.getActions();
    if (targetActions != null) {
        sb.append(":").append(targetActions);
    }
    return new DeniedPermission(sb.toString(), target);
}

private DeniedPermission(String name, Permission target) {
    super(name);
    this.目标=目标;
}

private Permission 初始化目标(String 目标类名, String 目标名称, String 目标操作)
{
Class<?> 目标类;
try {
    目标类 = Class.forName(目标类名);
}
catch (ClassNotFoundException cnfe) {
    if (目标类名.trim().isEmpty()) {
        目标类名 = "<empty>";
    }
    throw new IllegalArgumentException(
        MessageFormat.format("未找到目标权限类 [{0}]。",
        目标类名));
}
if (!Permission.class.isAssignableFrom(目标类) ||
Modifier.isAbstract(目标类.getModifiers())) {
    throw new IllegalArgumentException(MessageFormat
        .format("目标权限类 [{0}] 不是 (具体的) 权限类。",
        目标类名));
}
if (目标类 == DeniedPermission.class || 目标类 == UnresolvedPermission.class) {
    throw new IllegalArgumentException("目标权限类不能是
DeniedPermission 本身。");
}
构造函数<?> targetCtor;
try {
    if (targetActions == null) {
targetCtor = targetClass.getConstructor(String.class);
    }
    else {
targetCtor = targetClass.getConstructor(String.class, String.class);
    }
}

```

```

/**
 * Instantiates a <code>DeniedPermission</code> that encapsulates the given target permission.
 *
 * @throws IllegalArgumentException
 * if <code>target</code> is <code>null</code>, a <code>DeniedPermission</code>, or
an
 * <code>UnresolvedPermission</code>.
*/
public static DeniedPermission newDeniedPermission(Permission target) {
    if (target == null) {
        throw new IllegalArgumentException("Null [target] argument.");
    }
    if (target instanceof DeniedPermission || target instanceof UnresolvedPermission) {
        throw new IllegalArgumentException("[target] must not be a DeniedPermission or an
UnresolvedPermission.");
    }
    StringBuilder sb = new
    StringBuilder(target.getClass().getName()).append(":").append(target.getName());
    String targetActions = target.getActions();
    if (targetActions != null) {
        sb.append(":").append(targetActions);
    }
    return new DeniedPermission(sb.toString(), target);
}

private DeniedPermission(String name, Permission target) {
    super(name);
    this.target = target;
}

private Permission initTarget(String targetClassName, String targetName, String targetActions)
{
    Class<?> targetClass;
    try {
        targetClass = Class.forName(targetClassName);
    }
    catch (ClassNotFoundException cnfe) {
        if (targetClassName.trim().isEmpty()) {
            targetClassName = "<empty>";
        }
        throw new IllegalArgumentException(
            MessageFormat.format("Target Permission class [{0}] not found。",
            targetClassName));
    }
    if (!Permission.class.isAssignableFrom(targetClass) ||
Modifier.isAbstract(targetClass.getModifiers())) {
        throw new IllegalArgumentException(MessageFormat
            .format("Target Permission class [{0}] is not a (concrete) Permission。",
            targetClassName));
    }
    if (targetClass == DeniedPermission.class || targetClass == UnresolvedPermission.class) {
        throw new IllegalArgumentException("Target Permission class cannot be a
DeniedPermission itself.");
    }
    Constructor<?> targetCtor;
    try {
        if (targetActions == null) {
            targetCtor = targetClass.getConstructor(String.class);
        }
        else {
            targetCtor = targetClass.getConstructor(String.class, String.class);
        }
    }
}

```

```

    }
    catch (NoSuchMethodException nsme) {
        throw new IllegalArgumentException(MessageFormat.format(
            "目标权限类 [{0}] 未提供或暴露 (String name) 或
            (String name, String actions) 构造函数。",
            targetClassName));
    }
    try {
        return (Permission) targetCtor
.newInstance(((targetCtor.getParameterCount() == 1) ? new Object[] { targetName
} : new Object[] { targetName, targetActions }));
    }
    catch (ReflectiveOperationException roe) {
        if (roe instanceof InvocationTargetException) {
            if (targetName == null) {
                targetName = "<null>";
            }
            else if (targetName.trim().isEmpty()) {
                targetName = "<empty>";
            }
            if (targetActions == null) {
                targetActions = "<null>";
            }
            else if (targetActions.trim().isEmpty()) {
                targetActions = "<empty>";
            }
            throw new IllegalArgumentException(MessageFormat.format(
                "无法实例化目标权限类 [{0}]；提供的目标名称
                [{1}] 和/或目标操作 [{2}] 可能有误。",
                targetClassName, targetName, targetActions), roe);
        }
        throw new RuntimeException(
            "无法实例化目标权限类 [{0}]；发生了意外错误
            — 详情请参见附带的原因",
            roe);
    }
}

/**
 * 检查给定的权限是否被此权限所包含，参见{@link DeniedPermission
 * 概述}。
 */
@Override
public boolean implies(Permission p) {
    if (p instanceof DeniedPermission) {
        return target.implies(((DeniedPermission) p).target);
    }
    return target.implies(p);
}

/**
 * 返回此被拒绝权限的目标权限（实际的正向权限），该权限不是
 * 待授予）。
 */
public Permission getTargetPermission() {
    return target;
}
}

```

拒绝策略 (DenyingPolicy) 类

```

    }
    catch (NoSuchMethodException nsme) {
        throw new IllegalArgumentException(MessageFormat.format(
            "Target Permission class [{0}] does not provide or expose a (String name) or
            (String name, String actions) constructor.",
            targetClassName));
    }
    try {
        return (Permission) targetCtor
.newInstance(((targetCtor.getParameterCount() == 1) ? new Object[] { targetName
} : new Object[] { targetName, targetActions }));
    }
    catch (ReflectiveOperationException roe) {
        if (roe instanceof InvocationTargetException) {
            if (targetName == null) {
                targetName = "<null>";
            }
            else if (targetName.trim().isEmpty()) {
                targetName = "<empty>";
            }
            if (targetActions == null) {
                targetActions = "<null>";
            }
            else if (targetActions.trim().isEmpty()) {
                targetActions = "<empty>";
            }
            throw new IllegalArgumentException(MessageFormat.format(
                "Could not instantiate target Permission class [{0}]； provided target name
                [{1}] and/or target actions [{2}] potentially erroneous.",
                targetClassName, targetName, targetActions), roe);
        }
        throw new RuntimeException(
            "Could not instantiate target Permission class [{0}]； an unforeseen error
            occurred - see attached cause for details",
            roe);
    }
}

/**
 * Checks whether the given permission is implied by this one, as per the {@link DeniedPermission
 * overview}.
 */
@Override
public boolean implies(Permission p) {
    if (p instanceof DeniedPermission) {
        return target.implies(((DeniedPermission) p).target);
    }
    return target.implies(p);
}

/**
 * Returns this denied permission's target permission (the actual positive permission which is
 * not
 * to be granted).
 */
public Permission getTargetPermission() {
    return target;
}
}

```

The DenyingPolicy class

```

包com.example;

import java.security.CodeSource;
import java.security.NoSuchAlgorithmException;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Policy;
import java.security.ProtectionDomain;
import java.security.UnresolvedPermission;
import java.util.Enumeration;

/**
 * 包装器，为标准的基于文件的 <code>Policy</code> 添加了基本的 {@link DeniedPermission} 处理功能。
 */
public final class DenyingPolicy extends Policy {

    {
        try {
defaultPolicy = Policy.getInstance("javaPolicy", null);
        }
        catch (NoSuchAlgorithmException nsae) {
            throw new RuntimeException("无法获取默认策略。", nsae);
        }
    }

    private final Policy defaultPolicy;

    @Override
    public PermissionCollection getPermissions(CodeSource codesource) {
        return defaultPolicy.getPermissions(codesource);
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return defaultPolicy.getPermissions(domain);
    }

    /**
     * @return
     *      <ul>
     *          <li><code>true</code> 如果满足以下条件：</li>
     *          <ul>
     *              <li><code>permission</code> <em>不是</em> <code>DeniedPermission</code> 的实例，</li>
     *              <li>在系统默认的 <code>Policy</code> 上调用 <code>implies(domain, permission)</code> 返回 <code>true</code>，且</li>
     *              <li><code>permission</code> <em>不被</em> 任何可能已分配给 <code>domain</code> 的 <code>DeniedPermission</code> 所隐含。</li>
     *          </ul>
     *      </ul>
     *      <li>否则返回 <code>false</code>。
     *  </ul>
    */

    @Override
    public boolean implies(ProtectionDomain domain, Permission permission) {
        if (permission instanceof DeniedPermission) {
            /*
             * 在策略决策层面，DeniedPermissions 只能自身隐含，而不能被隐含（如
             * 他们是剥夺特权，而非授予特权）。此外，客户不应使用此
             */
    }
}

package com.example;

import java.security.CodeSource;
import java.security.NoSuchAlgorithmException;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Policy;
import java.security.ProtectionDomain;
import java.security.UnresolvedPermission;
import java.util.Enumeration;

/**
 * Wrapper that adds rudimentary {@link DeniedPermission} processing capabilities to the standard
 * file-backed <code>Policy</code>.
 */
public final class DenyingPolicy extends Policy {

    {
        try {
            defaultPolicy = Policy.getInstance("javaPolicy", null);
        }
        catch (NoSuchAlgorithmException nsae) {
            throw new RuntimeException("Could not acquire default Policy.", nsae);
        }
    }

    private final Policy defaultPolicy;

    @Override
    public PermissionCollection getPermissions(CodeSource codesource) {
        return defaultPolicy.getPermissions(codesource);
    }

    @Override
    public PermissionCollection getPermissions(ProtectionDomain domain) {
        return defaultPolicy.getPermissions(domain);
    }

    /**
     * @return
     *      <ul>
     *          <li><code>true</code> if:</li>
     *          <ul>
     *              <li><code>permission</code> <em>is not</em> an instance of
     *                  <code>DeniedPermission</code>, </li>
     *              <li>an <code>implies(domain, permission)</code> invocation on the system-default
     *                  <code>Policy</code> yields <code>true</code>, and</li>
     *              <li><code>permission</code> <em>is not</em> implied by any
     *                  <code>DeniedPermission</code>s
     *                  having potentially been assigned to <code>domain</code>. </li>
     *          </ul>
     *      </ul>
     *      <li><code>false</code>, otherwise.
     *  </ul>
    */

    @Override
    public boolean implies(ProtectionDomain domain, Permission permission) {
        if (permission instanceof DeniedPermission) {
            /*
             * At the policy decision level, DeniedPermissions can only themselves imply, not be
             * implied (as
             * they take away, rather than grant, privileges). Furthermore, clients aren't supposed
             * to use this
             */
    }
}

```

```

* 用于检查某个域是否_没有_某个权限的方法 (这毕竟是DeniedPermissions所表达的内容)。
    */
    return false;
}

if (!defaultPolicy.implies(domain, permission)) {
    // 未授予权限, 因此无需检查是否被拒绝
    return false;
}

/*
* 权限已授予——现在检查是否存在覆盖的DeniedPermission。以下
* 假设previousPolicy是sun.security.provider.PolicyFile (不同的
* 实现可能不支持#getPermissions(ProtectionDomain)和/或以不同方式处理UnresolvedPermissions)。
*/

```

```

Enumeration<Permission> perms = defaultPolicy.getPermissions(domain).elements();
while (perms.hasMoreElements()) {
    Permission p = perms.nextElement();
    /*
     * DeniedPermissions 通常会保持未解析状态, 因为没有代码会检查
     * 其他代码是否被“授予”了此类权限。
     */
    if (p instanceof UnresolvedPermission) {
        UnresolvedPermission up = (UnresolvedPermission) p;
        if (up.getUnresolvedType().equals(DeniedPermission.class.getName())) {
            // 强制解析
            defaultPolicy.implies(domain, up);
            // 立即评估, 避免重复遍历集合
            p = new DeniedPermission(up.getUnresolvedName());
        }
        if (p instanceof DeniedPermission && p.implies(permission)) {
            // 权限被拒绝
            return false;
        }
    }
    // 权限被授予
    return true;
}

@Override
public void refresh() {
    defaultPolicy.refresh();
}

```

演示

```

package com.example;

import java.security.Policy;

public class Main {

    public static void main(String... args) {
        Policy.setPolicy(new DenyingPolicy());
    }
}

```

```

* method for checking whether some domain _does not_ have a permission (which is what
* DeniedPermissions express after all).
*/
return false;
}

if (!defaultPolicy.implies(domain, permission)) {
    // permission not granted, so no need to check whether denied
    return false;
}

/*
 * Permission granted--now check whether there's an overriding DeniedPermission. The
following
 * assumes that previousPolicy is a sun.security.provider.PolicyFile (different
implementations
 * might not support #getPermissions(ProtectionDomain) and/or handle UnresolvedPermissions
 * differently).
*/

Enumeration<Permission> perms = defaultPolicy.getPermissions(domain).elements();
while (perms.hasMoreElements()) {
    Permission p = perms.nextElement();
    /*
     * DeniedPermissions will generally remain unresolved, as no code is expected to check
whether other
     * code has been "granted" such a permission.
     */
    if (p instanceof UnresolvedPermission) {
        UnresolvedPermission up = (UnresolvedPermission) p;
        if (up.getUnresolvedType().equals(DeniedPermission.class.getName())) {
            // force resolution
            defaultPolicy.implies(domain, up);
            // evaluate right away, to avoid reiterating over the collection
            p = new DeniedPermission(up.getUnresolvedName());
        }
    }
    if (p instanceof DeniedPermission && p.implies(permission)) {
        // permission denied
        return false;
    }
}
// permission granted
return true;
}

@Override
public void refresh() {
    defaultPolicy.refresh();
}
}


```

Demo

```

package com.example;

import java.security.Policy;

public class Main {

    public static void main(String... args) {
        Policy.setPolicy(new DenyingPolicy());
    }
}

```

```
System.setSecurityManager(new SecurityManager());
// 应该失败
System.getProperty("foo.bar");
}
```

分配一些权限：

```
grant codeBase "file:///path/to/classes/bin/-"
    permission java.util.PropertyPermission "*", "read,write";
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";
};
```

最后，运行Main并观察其失败，原因是“拒绝”规则（DeniedPermission）覆盖了grant（其PropertyPermission）。注意，调用setProperty("foo.baz", "xyz")则会成功，因为被拒绝的权限仅涵盖“读取”操作，且仅针对“foo.bar”属性。

```
System.setSecurityManager(new SecurityManager());
// should fail
System.getProperty("foo.bar");
}
```

Assign some permissions:

```
grant codeBase "file:///path/to/classes/bin/-"
    permission java.util.PropertyPermission "*", "read,write";
    permission com.example.DeniedPermission "java.util.PropertyPermission:foo.bar:read";
};
```

Lastly, run the Main and watch it fail, due to the "deny" rule (the DeniedPermission) overriding the grant (its PropertyPermission). Note that a setProperty("foo.baz", "xyz") invocation would instead have succeeded, since the denied permission only covers the "read" action, and solely for the "foo.bar" property.

第145章：JNDI

第145.1节：通过JNDI的RMI

本例展示了JNDI在RMI中的工作原理。它有两个作用：

- 为服务器提供绑定/解绑/重新绑定到RMI注册表的API
- 为客户端提供查找/列出RMI注册表的API。

RMI注册表是RMI的一部分，而非JNDI。

为简化起见，我们将使用`java.rmi.registry.CreateRegistry()`来创建RMI注册表。

1.Server.java (JNDI服务器)

```
包com.neohope.jndi.test;

导入javax.naming.Context;
导入javax.naming.InitialContext;
导入javax.naming.NamingException;
导入java.io.IOException;
导入java.rmi.RemoteException;
导入java.rmi.registry.LocateRegistry;
导入java.util.Hashtable;

/**
 * JNDI服务器
 * 1. 在端口1234上创建注册表
 * 2. 绑定JNDI
 * 3. 等待连接
 * 4. 清理并结束
 */
public class Server {
    private static Registry registry;
    private static InitialContext ctx;

    public static void initJNDI() {
        try {
registry = LocateRegistry.createRegistry(1234);
            final Hashtable jndiProperties = new Hashtable();
            jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
            ctx = new InitialContext(jndiProperties);
        } catch (NamingException e) {
e.printStackTrace();
        } catch (RemoteException e) {
e.printStackTrace();
        }
    }

    public static void bindJNDI(String name, Object obj) throws NamingException {
        ctx.bind(name, obj);
    }

    public static void unbindJNDI(String name) throws NamingException {
        ctx.unbind(name);
    }
}
```

Chapter 145: JNDI

Section 145.1: RMI through JNDI

This example shows how JNDI works in RMI. It has two roles:

- to provide the server with a bind/unbind/rebind API to the RMI Registry
- to provide the client with a lookup/list API to the RMI Registry.

The RMI Registry is part of RMI, not JNDI.

To make this simple, we will use `java.rmi.registry.CreateRegistry()` to create the RMI Registry.

1. Server.java(the JNDI server)

```
package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.util.Hashtable;

/**
 * JNDI Server
 * 1.create a registry on port 1234
 * 2.bind JNDI
 * 3.wait for connection
 * 4.clean up and end
 */
public class Server {
    private static Registry registry;
    private static InitialContext ctx;

    public static void initJNDI() {
        try {
registry = LocateRegistry.createRegistry(1234);
            final Hashtable jndiProperties = new Hashtable();
            jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");
            ctx = new InitialContext(jndiProperties);
        } catch (NamingException e) {
e.printStackTrace();
        } catch (RemoteException e) {
e.printStackTrace();
        }
    }

    public static void bindJNDI(String name, Object obj) throws NamingException {
        ctx.bind(name, obj);
    }

    public static void unbindJNDI(String name) throws NamingException {
        ctx.unbind(name);
    }
}
```

```

public static void unInitJNDI() throws NamingException {
    ctx.close();
}

public static void main(String[] args) throws NamingException, IOException {
    initJNDI();
    NMessage msg = new NMessage("Just A Message");
    bindJNDI("/neohope/jndi/test01", msg);
    System.in.read();
    unbindJNDI("/neohope/jndi/test01");
    unInitJNDI();
}
}

```

2.Client.java (JNDI客户端)

```

package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;

/**
 * 1.初始化上下文
 * 2.在注册表中查找服务
 * 3.使用该服务
 * 4.end
 */
public class Client {
    public static void main(String[] args) throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
        jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");

        InitialContext ctx = new InitialContext(jndiProperties);
        NMessage msg = (NMessage) ctx.lookup("/neohope/jndi/test01");
        System.out.println(msg.message);
        ctx.close();
    }
}

```

3.NMessage.java (RMI服务器类)

```

package com.neohope.jndi.test;

import java.io.Serializable;
import java.rmi.Remote;

/**
 * NMessage
 * RMI服务器类
 * 必须实现 Remote 和 Serializable
 */
public class NMessage implements Remote, Serializable {
    public String message = "";

    public NMessage(String message)
    {
        this.message = message;
    }
}

```

```

public static void unInitJNDI() throws NamingException {
    ctx.close();
}

public static void main(String[] args) throws NamingException, IOException {
    initJNDI();
    NMessage msg = new NMessage("Just A Message");
    bindJNDI("/neohope/jndi/test01", msg);
    System.in.read();
    unbindJNDI("/neohope/jndi/test01");
    unInitJNDI();
}
}

```

2. Client.java(the JNDI client)

```

package com.neohope.jndi.test;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;

/**
 * 1.init context
 * 2.lookup registry for the service
 * 3.use the service
 * 4.end
 */
public class Client {
    public static void main(String[] args) throws NamingException {
        final Hashtable jndiProperties = new Hashtable();
        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.rmi.registry.RegistryContextFactory");
        jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");

        InitialContext ctx = new InitialContext(jndiProperties);
        NMessage msg = (NMessage) ctx.lookup("/neohope/jndi/test01");
        System.out.println(msg.message);
        ctx.close();
    }
}

```

3. NMessage.java (RMI server class)

```

package com.neohope.jndi.test;

import java.io.Serializable;
import java.rmi.Remote;

/**
 * NMessage
 * RMI server class
 * must implements Remote and Serializable
 */
public class NMessage implements Remote, Serializable {
    public String message = "";

    public NMessage(String message)
    {
        this.message = message;
    }
}

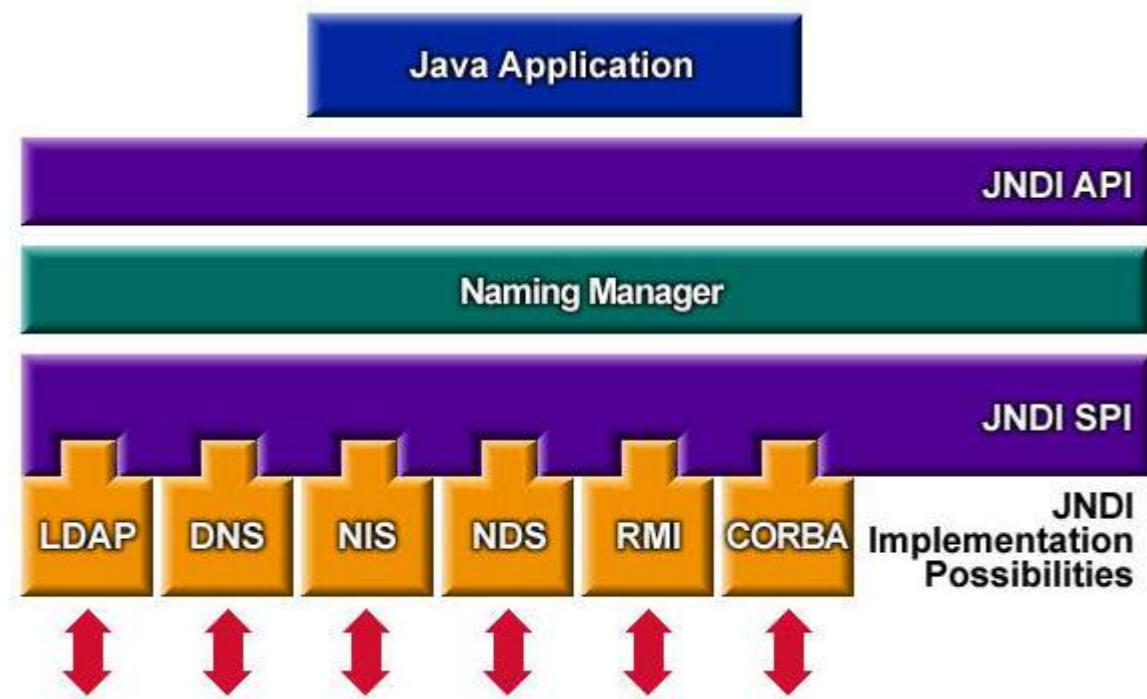
```

```
}
```

如何运行示例：

1. 构建并启动服务器
2. 构建并启动客户端

介绍



Java 命名和目录接口 (JNDI) 是一个用于目录服务的 Java API，允许 Java 软件客户端通过名称发现和查找数据及对象。它设计为独立于任何特定的命名或目录服务实现。

JNDI 架构由一个 API (应用程序编程接口) 和一个 SPI (服务提供者接口) 组成。Java 应用程序使用该 API 访问各种命名和目录服务。SPI 使各种命名和目录服务能够透明地插入，从而允许使用 JNDI 技术 API 的 Java 应用程序访问它们的服务。

如上图所示，JNDI 支持 LDAP、DNS、NIS、NDS、RMI 和 CORBA。当然，你也可以扩展它。

工作原理

在此示例中，Java RMI 使用 JNDI API 在网络中查找对象。如果您想查找一个对象，您至少需要两条信息：

- 对象的位置

RMI注册表管理名称绑定，它告诉你在哪里可以找到对象。

- 对象的名称

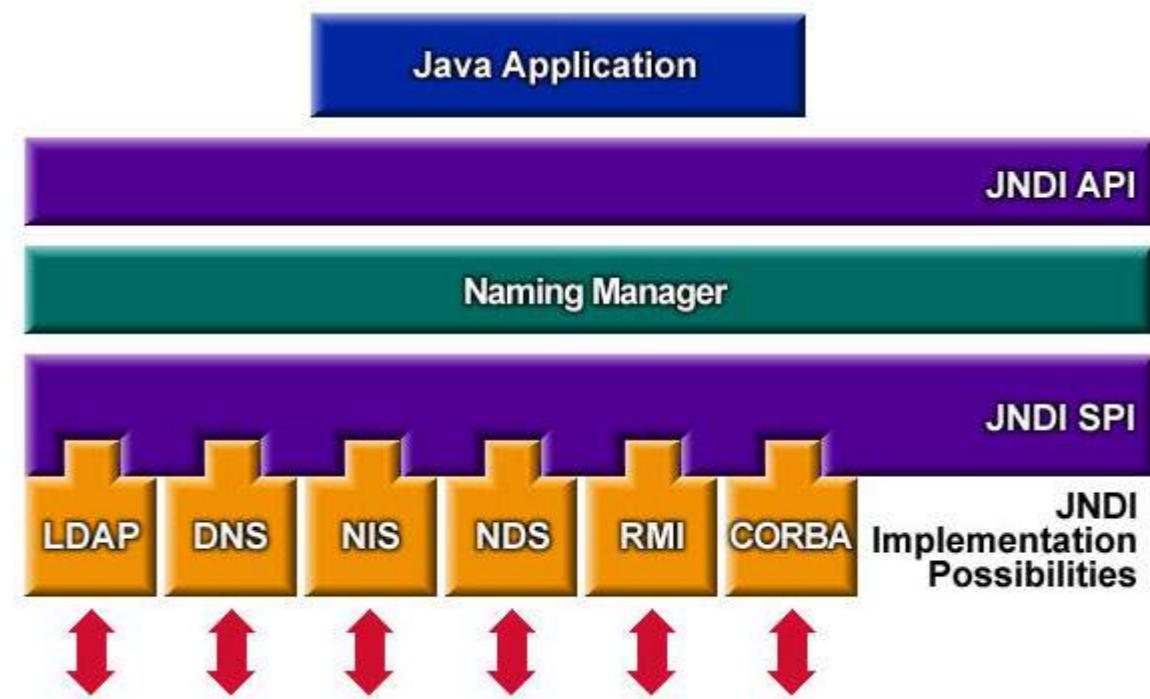
对象的名称是什么？通常是一个字符串，也可以是实现了Name接口的对象。

```
}
```

How to run the example:

1. build and start the server
2. build and start the client

Introduce



The **Java Naming and Directory Interface (JNDI)** is a Java API for a directory service that allows Java software clients to discover and look up data and objects via a name. It is designed to be independent of any specific naming or directory service implementation.

The JNDI architecture consists of an **API** (Application Programming Interface) and an **SPI** (Service Provider Interface). Java applications use this API to access a variety of naming and directory services. The SPI enables a variety of naming and directory services to be plugged in transparently, allowing the Java application using the API of the JNDI technology to access their services.

As you can see from the picture above, JNDI supports LDAP, DNS, NIS, NDS, RMI and CORBA. Of course, you can extend it.

How it works

In this example, the Java RMI use the JNDI API to look up objects in a network. If you want to look up an object, you need at least two pieces of information:

- Where to find the object

The RMI Registry manages the name bindings, it tells you where to find the object.

- The name of the object

What is an object's name? It is usually a string, it can also be an object that implements the Name interface.

一步步来

- 首先你需要一个注册表，用来管理名称绑定。在这个例子中，我们使用

```
java.rmi.registry.LocateRegistry.
```

```
//这将在本地主机的1234端口启动一个注册表  
registry = LocateRegistry.createRegistry(1234);
```

- 客户端和服务器端都需要一个上下文（Context）。服务器使用上下文绑定名称和对象。客户端使用上下文查找名称并获取对象。

```
// 我们使用 com.sun.jndi.rmi.registry.RegistryContextFactory 作为 InitialContextFactory  
final Hashtable jndiProperties = new Hashtable();  
jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,  
"com.sun.jndi.rmi.registry.RegistryContextFactory");  
// 注册表 URL 是 "rmi://localhost:1234"  
jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");  
InitialContext ctx = new InitialContext(jndiProperties);
```

- 服务器绑定名称和对象

```
// jndi 名称是 "/neohope/jndi/test01"  
bindJNDI("/neohope/jndi/test01", msg);
```

- 客户端通过名称 "/neohope/jndi/test01" 查找对象

```
// 通过名称 "java:com/neohope/jndi/test01" 查找对象  
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

- 现在客户端可以使用该对象

- 当服务器结束时，需要进行清理。

```
ctx.unbind("/neohope/jndi/test01");  
ctx.close();
```

Step by step

- First you need a registry, which manage the name binding. In this example, we use

```
java.rmi.registry.LocateRegistry.
```

```
//This will start a registry on localhost, port 1234  
registry = LocateRegistry.createRegistry(1234);
```

- Both client and server need a Context. Server use the Context to bind the name and object. Client use the Context to lookup the name and get the object.

```
//We use com.sun.jndi.rmi.registry.RegistryContextFactory as the InitialContextFactory  
final Hashtable jndiProperties = new Hashtable();  
jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY,  
"com.sun.jndi.rmi.registry.RegistryContextFactory");  
//the registry usrl is "rmi://localhost:1234"  
jndiProperties.put(Context.PROVIDER_URL, "rmi://localhost:1234");  
InitialContext ctx = new InitialContext(jndiProperties);
```

- The server bind the name and object

```
//The jndi name is "/neohope/jndi/test01"  
bindJNDI("/neohope/jndi/test01", msg);
```

- The client look up the object by the name "/neohope/jndi/test01"

```
//look up the object by name "java:com/neohope/jndi/test01"  
NeoMessage msg = (NeoMessage) ctx.lookup("/neohope/jndi/test01");
```

- Now the client can use the object

- When the server is ending, need to clean up.

```
ctx.unbind("/neohope/jndi/test01");  
ctx.close();
```

第146章：sun.misc.Unsafe

第146.1节：通过反射实例化sun.misc.Unsafe

```
public static Unsafe getUnsafe() {
    try {
        Field unsafe = Unsafe.class.getDeclaredField("theUnsafe");
        unsafe.setAccessible(true);
        return (Unsafe) unsafe.get(null);
    } catch (IllegalAccessException e) {
        // 处理
    } catch (IllegalArgumentException e) {
        // 处理
    } catch (NoSuchFieldException e) {
        // 处理
    } catch (SecurityException e) {
        // 处理
    }
}
```

sun.misc.Unsafe 有一个私有构造函数，且静态方法 getUnsafe() 通过检查类加载器来确保代码是由主类加载器加载的。因此，加载该实例的一种方法是使用反射获取静态字段。

第146.2节：通过启动类路径实例化 sun.misc.Unsafe

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
}
```

虽然此示例可以编译，但除非 Unsafe 类是由主类加载器加载，否则很可能在运行时失败。为确保这一点，JVM 应该使用适当的参数启动，例如：

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

然后 foo.bar.MyApp 类可以使用 UnsafeLoader.loadUnsafe()。

第146.3节：获取 Unsafe 实例

Unsafe 被存储为一个私有字段，无法直接访问。构造函数是私有的，唯一访问方法 public static Unsafe getUnsafe() 具有特权访问权限。通过反射，有一种变通方法可以使私有字段可访问：

```
public static final Unsafe UNSAFE;

static {
    Unsafe unsafe = null;

    try {
        final PrivilegedExceptionAction<Unsafe> action = () -> {
            final Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);

            return (Unsafe) f.get(null);
        };
    }
```

Chapter 146: sun.misc.Unsafe

Section 146.1: Instantiating sun.misc.Unsafe via reflection

```
public static Unsafe getUnsafe() {
    try {
        Field unsafe = Unsafe.class.getDeclaredField("theUnsafe");
        unsafe.setAccessible(true);
        return (Unsafe) unsafe.get(null);
    } catch (IllegalAccessException e) {
        // Handle
    } catch (IllegalArgumentException e) {
        // Handle
    } catch (NoSuchFieldException e) {
        // Handle
    } catch (SecurityException e) {
        // Handle
    }
}
```

sun.misc.Unsafe has a Private constructor, and the static getUnsafe() method is guarded with a check of the classloader to ensure that the code was loaded with the primary classloader. Therefore, one method of loading the instance is to use reflection to get the static field.

Section 146.2: Instantiating sun.misc.Unsafe via bootclasspath

```
public class UnsafeLoader {
    public static Unsafe loadUnsafe() {
        return Unsafe.getUnsafe();
    }
}
```

While this example will compile, it is likely to fail at runtime unless the Unsafe class was loaded with the primary classloader. To ensure that happens the JVM should be loaded with the appropriate arguments, like:

```
java -Xbootclasspath:$JAVA_HOME/jre/lib/rt.jar:./UnsafeLoader.jar foo.bar.MyApp
```

The foo.bar.MyApp class can then use UnsafeLoader.loadUnsafe().

Section 146.3: Getting Instance of Unsafe

Unsafe is stored as a private field that cannot be accessed directly. The constructor is private and the only method to access public static Unsafe getUnsafe() has privileged access. By use of reflection, there is a work-around to make private fields accessible:

```
public static final Unsafe UNSAFE;

static {
    Unsafe unsafe = null;

    try {
        final PrivilegedExceptionAction<Unsafe> action = () -> {
            final Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);

            return (Unsafe) f.get(null);
        };
    }
```

```

};

unsafe = AccessController.doPrivileged(action);
} catch (final Throwable t) {
    throw new RuntimeException("Exception accessing Unsafe", t);
}

UNSAFE = unsafe;
}

```

第146.4节：Unsafe的用途

Unsafe的一些用途如下：

用途	API
堆外/直接内存的分配、重新分配和释放	allocateMemory(字节), reallocateMemory(地址, 字节) 和 freeMemory(地址)
内存栅栏	loadFence(), storeFence(), fullFence()
挂起当前线程	park(是否绝对, 时间), unpark(线程)
直接字段和/或内存访问	get* 和 put* 方法族
抛出未检查异常	throwException(e)
CAS 和原子操作	compareAndSwap* 方法族
设置内存	setMemory
易失性或并发操作	get*易失性, put*易失性, putOrdered*

get 和 put 系列方法是针对给定对象的。如果对象为 null，则视为绝对地址。

```

// 向字段写入值
protected static long fieldOffset = UNSAFE.objectFieldOffset(getClass().getField("theField"));
UNSAFE.putLong(this, fieldOffset, newValue);

// 写入绝对值
UNSAFE.putLong(null, address, newValue);
UNSAFE.putLong(address, newValue);

```

某些方法仅定义于 int 和 long 类型。你可以通过以下方式在 float 和 double 上使用这些方法
floatToRawIntBits, intBitsToFloat, doubleToRawLongBits, longBitsToDouble`

```

};

unsafe = AccessController.doPrivileged(action);
} catch (final Throwable t) {
    throw new RuntimeException("Exception accessing Unsafe", t);
}

UNSAFE = unsafe;
}

```

Section 146.4: Uses of Unsafe

Some uses of unsafe is as follows:

Use	API
Off heap / direct memory allocation, reallocation and deallocation	allocateMemory(bytes), reallocateMemory(address, bytes) and freeMemory(address)
Memory fences	loadFence(), storeFence(), fullFence()
Parking current thread	park(isAbsolute, time), unpark(thread)
Direct field and or memory access	get* and put* family of methods
Throwing unchecked exceptions	throwException(e)
CAS and Atomic Operations	compareAndSwap* family of methods
Setting out memory	setMemory
Volatile or concurrent operations	get*Volatile, put*Volatile, putOrdered*

The get and put family of methods are relative to a given object. If the object is null then it is treated as an absolute address.

```

// Putting a value to a field
protected static long fieldOffset = UNSAFE.objectFieldOffset(getClass().getField("theField"));
UNSAFE.putLong(this, fieldOffset, newValue);

// Putting an absolute value
UNSAFE.putLong(null, address, newValue);
UNSAFE.putLong(address, newValue);

```

Some methods are only defined for int and longs. You can use these methods on floats and doubles using floatToRawIntBits, intBitsToFloat, doubleToRawLongBits, longBitsToDouble`

第147章：Java内存模型

第147.1节：内存模型的动机

考虑以下示例：

```
public class 示例 {
    public int a, b, c, d;

    public void 执行() {
        a = b + 1;
        c = d + 1;
    }
}
```

如果这个类在单线程应用中使用，那么可观察到的行为将完全符合你的预期。例如：

```
public class 单线程 {
    public static void main(String[] 参数) {
        示例 eg = new 示例();
        System.out.println(eg.a + ", " + eg.c);
        eg.执行();
        System.out.println(eg.a + ", " + eg.c);
    }
}
```

将输出：

```
0, 0
1, 1
```

就“主”线程而言，main()方法和doIt()方法中的语句将按照它们在源代码中编写的顺序执行。这是Java语言规范 (JLS) 的明确要求。

现在考虑在多线程应用程序中使用相同的类。

```
public class MultiThreaded {
    public static void main(String[] args) {
        final Example eg = new Example();
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    eg.doIt();
                }
            }
        }).start();
        while (true) {
            System.out.println(eg.a + ", " + eg.c);
        }
    }
}
```

这将打印什么？

Chapter 147: Java Memory Model

Section 147.1: Motivation for the Memory Model

Consider the following example:

```
public class Example {
    public int a, b, c, d;

    public void doIt() {
        a = b + 1;
        c = d + 1;
    }
}
```

If this class is used in a single-threaded application, then the observable behavior will be exactly as you would expect. For instance:

```
public class SingleThreaded {
    public static void main(String[] args) {
        Example eg = new Example();
        System.out.println(eg.a + ", " + eg.c);
        eg.doIt();
        System.out.println(eg.a + ", " + eg.c);
    }
}
```

will output:

```
0, 0
1, 1
```

As far as the “main” thread can tell, the statements in the main() method and the doIt() method will be executed in the order that they are written in the source code. This is a clear requirement of the Java Language Specification (JLS).

Now consider the same class used in a multi-threaded application.

```
public class MultiThreaded {
    public static void main(String[] args) {
        final Example eg = new Example();
        new Thread(new Runnable() {
            public void run() {
                while (true) {
                    eg.doIt();
                }
            }
        }).start();
        while (true) {
            System.out.println(eg.a + ", " + eg.c);
        }
    }
}
```

What will this print?

事实上，根据JLS，无法预测这将打印：

- 你可能会先看到几行0, 0。
- 然后你可能会看到类似N, N或N, N + 1的行。
- 你可能会看到类似 N + 1, N 这样的行。
- 理论上，你甚至可能看到 0, 0 这些行无限持续1。

1 - 实际上，`println` 语句的存在可能会导致某种偶然的同步和内存缓存刷新。这很可能掩盖导致上述行为的一些效果。

那么我们如何解释这些现象呢？

赋值的重排序

一个可能的解释是 JIT 编译器改变了 `doIt()` 方法中赋值的顺序。Java语言规范 (JLS) 要求语句必须“看起来”按照当前线程的视角顺序执行。在这种情况下，`doIt()` 方法中的代码无法观察到这两个语句（假设的）重排序的效果。这意味着 JIT 编译器被允许这样做。

为什么它会这样做？

在典型的现代硬件上，机器指令通过指令流水线执行，允许一系列指令处于不同阶段。某些指令执行阶段比其他阶段耗时更长，内存操作通常耗时更久。一个智能的编译器可以通过调整指令顺序来优化流水线的指令吞吐量，以最大化重叠执行的数量。这可能导致部分语句的执行顺序被打乱。JLS 允许这种做法，前提是它不会影响“从当前线程视角看”的计算结果。

内存缓存的影响

第二个可能的解释是内存缓存的影响。在经典计算机架构中，每个处理器有一小部分寄存器和较大的内存。访问寄存器比访问主内存快得多。在现代架构中，存在比寄存器慢但比主内存快的内存缓存。

编译器会通过尝试将变量的副本保存在寄存器或内存缓存中来利用这一点。如果变量不需要刷新到主内存，或者不需要从内存读取，那么不执行这些操作会带来显著的性能提升。在Java语言规范 (JLS) 不要求内存操作对其他线程可见的情况下，Java即时编译器 (JIT) 很可能不会添加“读屏障”和“写屏障”指令，这些指令会强制进行主内存的读写操作。再次强调，这样做的性能提升是显著的。

适当的同步

到目前为止，我们已经看到JLS允许JIT编译器通过重新排序或避免内存操作来加快单线程代码的执行速度。但当其他线程可以观察主内存中（共享）变量的状态时，会发生什么呢？

答案是，其他线程可能会观察到基于Java语句代码顺序看似不可能出现的变量状态.....

解决方案是使用适当的同步。三种主要方法是：

- 使用原始互斥锁和 `synchronized` 结构。
- 使用 `volatile` 变量。
- 使用更高级的并发支持；例如 `java.util.concurrent` 包中的类。

In fact, according to the JLS it is not possible to predict that this will print:

- You will probably see a few lines of 0, 0 to start with.
- Then you probably see lines like N, N or N, N + 1.
- You might see lines like N + 1, N.
- In theory, you might even see that the 0, 0 lines continue forever1.

1 - In practice the presence of the `println` statements is liable to cause some serendipitous synchronization and memory cache flushing. That is likely to hide some of the effects that would cause the above behavior.

So how can we explain these?

Reordering of assignments

One possible explanation for unexpected results is that the JIT compiler has changed the order of the assignments in the `doIt()` method. The JLS requires that statements *appear to execute in order from the perspective of the current thread*. In this case, nothing in the code of the `doIt()` method can observe the effect of a (hypothetical) reordering of those two statement. This means that the JIT compiler would be permitted to do that.

Why would it do that?

On typical modern hardware, machine instructions are executed using a instruction pipeline which allows a sequence of instructions to be in different stages. Some phases of instruction execution take longer than others, and memory operations tend to take a longer time. A smart compiler can optimize the instruction throughput of the pipeline by ordering the instructions to maximize the amount of overlap. This may lead to executing parts of statements out of order. The JLS permits this provided that not affect the result of the computation *from the perspective of the current thread*.

Effects of memory caches

A second possible explanation is effect of memory caching. In a classical computer architecture, each processor has a small set of registers, and a larger amount of memory. Access to registers is much faster than access to main memory. In modern architectures, there are memory caches that are slower than registers, but faster than main memory.

A compiler will exploit this by trying to keep copies of variables in registers, or in the memory caches. If a variable does not *need* to be flushed to main memory, or does not *need* to be read from memory, there are significant performance benefits in not doing this. In cases where the JLS does not require memory operations to be visible to another thread, the Java JIT compiler is likely to not add the "read barrier" and "write barrier" instructions that will force main memory reads and writes. Once again, the performance benefits of doing this are significant.

Proper synchronization

So far, we have seen that the JLS allows the JIT compiler to generate code that makes single-threaded code faster by reordering or avoiding memory operations. But what happens when other threads can observe the state of the (shared) variables in main memory?

The answer is, that the other threads are liable to observe variable states which would appear to be impossible ... based on the code order of the Java statements. The solution to this is to use appropriate synchronization. The three main approaches are:

- Using primitive mutexes and the `synchronized` constructs.
- Using `volatile` variables.
- Using higher level concurrency support; e.g. classes in the `java.util.concurrent` packages.

但即使如此，理解何处需要同步以及可以依赖哪些效果仍然很重要。这就是Java内存模型的作用所在。

内存模型

Java内存模型是Java语言规范 (JLS) 中规定一个线程在何种条件下能够保证看到另一个线程所做内存写入效果的部分。内存模型以相当严谨的形式规范，因此需要详细且仔细地阅读才能理解。但基本原则是，某些构造会在一个线程对变量的写入与另一个线程随后对同一变量的读取之间创建“先行发生” (happens-before) 关系。如果存在“先行发生”关系，JIT编译器必须生成代码，确保读取操作能够看到写入操作写入的值。

有了这个基础，就可以推理Java程序中的内存一致性，并判断其在所有执行平台上是否可预测且一致。

第147.2节：先行发生关系

(以下是Java语言规范的简化版本。若要深入理解，需要阅读规范原文。)

先行发生关系是内存模型中使我们能够理解和推理内存可见性的部分。正如JLS所述 (JLS 17.4.5)：

“两个操作可以通过先行发生关系排序。如果一个操作先行发生另一个操作，那么第一个操作对第二个操作是可见的且在其之前发生。”

这是什么意思？

操作

上述引用中提到的操作在JLS 17.4.2中有规定。规范定义了5种操作类型：

- 读取：读取一个非易失性变量。
- 写入：写入一个非易失性变量。
- 同步操作：
 - 易失性读取：读取一个易失性变量。
 - 易失性写入：写入一个易失性变量。
 - 锁定。锁定一个监视器
 - 解锁。解锁一个监视器。
 - 线程的（合成的）第一个和最后一个操作。
 - 启动线程或检测线程已终止的操作。
- 外部操作。操作的结果依赖于程序所处的环境。
- 线程分歧操作。这些操作模拟某些类型无限循环的行为。

But even with this, it is important to understand where synchronization is needed, and what effects that you can rely on. This is where the Java Memory Model comes in.

The Memory Model

The Java Memory Model is the section of the JLS that specifies the conditions under which one thread is guaranteed to see the effects of memory writes made by another thread. The Memory Model is specified with a fair degree of *formal rigor*, and (as a result) requires detailed and careful reading to understand. But the basic principle is that certain constructs create a "happens-before" relation between write of a variable by one thread, and a subsequent read of the same variable by another thread. If the "happens before" relation exists, the JIT compiler is *obliged* to generate code that will ensure that the read operation sees the value written by the write.

Armed with this, it is possible to reason about memory coherency in a Java program, and decide whether this will be predictable and consistent for *all* execution platforms.

Section 147.2: Happens-before relationships

(The following is a simplified version of what the Java Language Specification says. For a deeper understanding, you need to read the specification itself.)

Happens-before relationships are the part of the Memory Model that allow us to understand and reason about memory visibility. As the JLS says ([JLS 17.4.5](#)):

"Two *actions* can be ordered by a *happens-before* relationship. If one action *happens-before* another, then the first is visible to and ordered before the second."

What does this mean?

Actions

The actions that the above quote refers to are specified in [JLS 17.4.2](#). There are 5 kinds of action listed defined by the spec:

- Read: Reading a non-volatile variable.
- Write: Writing a non-volatile variable.
- Synchronization actions:
 - Volatile read: Reading a volatile variable.
 - Volatile write: Writing a volatile variable.
 - Lock. Locking a monitor
 - Unlock. Unlocking a monitor.
 - The (synthetic) first and last actions of a thread.
 - Actions that start a thread or detect that a thread has terminated.
- External Actions. An action that has a result that depends on the environment in which the program.
- Thread divergence actions. These model the behavior of certain kinds of infinite loop.

程序顺序和同步顺序

这两种顺序 ([JLS 17.4.3](#) 和 [JLS 17.4.4](#)) 控制Java程序中语句的执行顺序。

程序顺序描述单线程内语句执行的顺序。

同步顺序描述由同步连接的两个语句的执行顺序：

- 对监视器的解锁操作与该监视器上所有后续的锁定操作同步。
- 对volatile变量的写操作与任何线程对同一变量的所有后续读操作同步。
- 启动线程的操作（即调用Thread.start()）与该线程中第一个操作同步（即调用线程的 run()方法）。
- 字段的默认初始化与每个线程中的第一个操作同步。（详见JLS中的解释。）
- 线程中的最后一个操作与检测到该线程终止的另一个线程中的任何操作同步；例如 join()调用的返回或isTerminated()调用返回true。
- 如果一个线程中断另一个线程，第一线程中的中断调用与另一个线程检测到该线程被中断的点同步。

先行发生顺序

这种排序 ([JLS 17.4.5](#)) 决定了内存写入是否保证对后续的内存读取可见。

更具体地说，变量v的读取保证能观察到对v的写入，当且仅当write(v)发生在read(v)之前，并且在此期间没有对v的其他写入。如果存在中间写入，那么read(v)可能会看到这些中间写入的结果，而不是较早的写入。

定义happens-before顺序的规则如下：

- Happens-Before规则#1** - 如果x和y是同一线程的操作，且x在程序顺序中位于y之前，那么x 先于 y。
- Happens-Before规则#2** - 从一个对象的构造函数结束到该对象的终结器开始之间存在一个happens-before边。
- Happens-Before规则#3** - 如果操作x与后续操作y同步，则x 先于 y。
- Happens-Before规则#4** - 如果x 先于 y，且y 先于 z，则x 先于 z。

此外，Java标准库中的各种类被指定为定义happens-before关系。你可以将其理解为某种方式发生了这种关系，而无需确切知道该保证是如何实现的。

第147.3节：如何避免需要理解内存模型

内存模型难以理解，也难以应用。如果你需要推理多线程代码的正确性，它是有用的，但你不希望为你编写的每个多线程应用都进行这种推理。

Program Order and Synchronization Order

These two orderings ([JLS 17.4.3](#) and [JLS 17.4.4](#)) govern the execution of statements in a Java

Program order describes the order of statement execution within a single thread.

Synchronization order describes the order of statement execution for two statements connected by a synchronization:

- An unlock action on monitor *synchronizes-with* all subsequent lock actions on that monitor.
- A write to a volatile variable *synchronizes-with* all subsequent reads of the same variable by any thread.
- An action that starts a thread (i.e. the call to `Thread.start()`) *synchronizes-with* the first action in the thread it starts (i.e. the call to the thread's `run()` method).
- The default initialization of fields *synchronizes-with* the first action in every thread. (See the JLS for an explanation of this.)
- The final action in a thread *synchronizes-with* any action in another thread that detects the termination; e.g. the return of a `join()` call or `isTerminated()` call that returns `true`.
- If one thread interrupts another thread, the interrupt call in the first thread *synchronizes-with* the point where another thread detects that the thread was interrupted.

Happens-before Order

This ordering ([JLS 17.4.5](#)) is what determines whether a memory write is guaranteed to be visible to a subsequent memory read.

More specifically, a read of a variable v is guaranteed to observe a write to v if and only if `write(v)` *happens-before* `read(v)` AND there is no intervening write to v. If there are intervening writes, then the `read(v)` may see the results of them rather than the earlier one.

The rules that define the *happens-before* ordering are as follows:

- Happens-Before Rule #1** - If x and y are actions of the same thread and x comes before y in *program order*, then x *happens-before* y.
- Happens-Before Rule #2** - There is a happens-before edge from the end of a constructor of an object to the start of a finalizer for that object.
- Happens-Before Rule #3** - If an action x *synchronizes-with* a subsequent action y, then x *happens-before* y.
- Happens-Before Rule #4** - If x *happens-before* y and y *happens-before* z then x *happens-before* z.

In addition, various classes in the Java standard libraries are specified as defining *happens-before* relationships. You can interpret this as meaning that it happens *somehow*, without needing to know exactly how the guarantee is going to be met.

Section 147.3: How to avoid needing to understand the Memory Model

The Memory Model is difficult to understand, and difficult to apply. It is useful if you need to reason about the correctness of multi-threaded code, but you do not want to have to do this reasoning for every multi-threaded application that you write.

如果你在编写Java并发代码时采用以下原则，基本上可以避免使用“happens-before”推理。

- 尽可能使用不可变数据结构。一个正确实现的不可变类将是线程安全的，并且在与其他类一起使用时不会引入线程安全问题。
- 理解并避免“非安全发布”(unsafe publication)。
- 使用原始互斥锁或Lock对象来同步对需要线程安全的可变对象状态的访问。
- 使用Executor / ExecutorService或分叉加入框架，而不是尝试直接创建和管理线程。
- 使用`java.util.concurrent`类，这些类提供高级锁、信号量、闩锁和屏障，而不是直接使用wait/notify/notifyAll。
- 使用java.util.concurrent版本的映射、集合、列表、队列和双端队列，而不是对非并发集合进行外部同步。

总体原则是尽量使用Java内置的并发库，而不是“自己动手”实现并发。

如果正确使用，你可以依赖它们的正常工作。

1 - 并非所有对象都需要线程安全。例如，如果一个对象或多个对象是线程限定的（即仅被一个线程访问），那么它们的线程安全性就不相关。

第147.4节：应用于一些示例的先行发生 (Happens-before) 推理

我们将展示一些示例，说明如何应用happens-before推理来检查写操作是否对后续读操作可见。

单线程代码

正如你所预期的，在单线程程序中，写操作总是对后续的读操作可见。

```
public class SingleThreadExample {  
    public int a, b;  
  
    public int add() {  
        a = 1;          // 写入(a)  
        b = 2;          // 写入(b)  
        return a + b; // 先读(a)后读(b)  
    }  
}
```

根据 Happens-Before 规则 #1：

1. write(a) 操作 happens-before write(b) 操作。
2. write(b) 操作 happens-before read(a) 操作。
3. read(a) 操作 happens-before read(a) 操作。

根据先行发生规则#4：

4. write(a) 先行发生于write(b) 且 write(b) 先行发生于read(a) 意味着 write(a) 先行发生于 read(a)。

If you adopt the following principals when writing concurrent code in Java, you can *largely* avoid the need to resort to *happens-before* reasoning.

- Use immutable data structures where possible. A properly implemented immutable class will be thread-safe, and will not introduce thread-safety issues when you use it with other classes.
- Understand and avoid "unsafe publication".
- Use primitive mutexes or Lock objects to synchronize access to state in mutable objects that need to be thread-safe1.
- Use Executor / ExecutorService or the fork join framework rather than attempting to create manage threads directly.
- Use the `java.util.concurrent classes that provide advanced locks, semaphores, latches and barriers, instead of using wait/notify/notifyAll directly.
- Use the `java.util.concurrent` versions of maps, sets, lists, queues and deques rather than external synchronization of non-concurrent collections.

The general principle is to try to use Java's built-in concurrency libraries rather than "rolling your own" concurrency. You can rely on them working, if you use them properly.

1 - Not all objects need to be thread safe. For example, if an object or objects is *thread-confined* (i.e. it is only accessible to one thread), then its thread-safety is not relevant.

Section 147.4: Happens-before reasoning applied to some examples

We will present some examples to show how to apply *happens-before* reasoning to check that writes are visible to subsequent reads.

Single-threaded code

As you would expect, writes are always visible to subsequent reads in a single-threaded program.

```
public class SingleThreadExample {  
    public int a, b;  
  
    public int add() {  
        a = 1;          // write(a)  
        b = 2;          // write(b)  
        return a + b; // read(a) followed by read(b)  
    }  
}
```

By Happens-Before Rule #1:

1. The write(a) action *happens-before* the write(b) action.
2. The write(b) action *happens-before* the read(a) action.
3. The read(a) action *happens-before* the read(a) action.

By Happens-Before Rule #4:

4. write(a) *happens-before* write(b) AND write(b) *happens-before* read(a) IMPLIES write(a) *happens-before* read(a).

5. write(b) 先行发生于read(a) 且 read(a) 先行发生于read(b) 意味着 write(b) 先行发生于 read(b)。

总结：

6. write(a) 先行发生于read(a) 关系意味着 a + b 语句保证能看到 a 的正确值。
7. 写入(b) 先于读取(b)关系意味着 a + b 语句保证能看到正确的b值。

两个线程示例中'volatile'的行为

我们将使用以下示例代码来探讨内存模型对`volatile`的一些影响。

```
public class VolatileExample {  
    private volatile int a;  
    private int b; // 非volatile  
  
    public void update(int first, int second) {  
        b = first; // 写入(b)  
        a = second; // 写入-volatile(a)  
    }  
  
    public int observe() {  
        return a + b; // 先读-volatile(a), 再读(b)  
    }  
}
```

首先，考虑涉及两个线程的以下语句序列：

1. 创建了一个VolatileExample的单一实例；称之为ve，
2. ve.update(1, 2) 在一个线程中被调用，且
3. ve.observe() 在另一个线程中被调用。

根据 Happens-Before 规则 #1：

1. write(a) 操作先于 volatile-write(a) 操作发生。
2. volatile-read(a) 操作先于 read(b) 操作发生。

根据 Happens-Before 规则 #2：

3. 第一个线程中的 volatile-write(a) 操作先于第二个线程中的 volatile-read(a) 操作发生

根据先行发生规则#4：

4. 第一个线程中的 write(b) 操作先于第二个线程中的 read(b) 操作发生。

换句话说，对于这个特定的序列，我们可以保证第二个线程会看到第一个线程对非 volatile 变量 b 所做的更新。然而，也应该清楚，如果 update方法中的赋值顺序相反，或者 observe() 方法在读取 a 之前读取了变量 b，那么 happens-before链将被打断。如果第二个线程中的 volatile-read(a) 操作不是紧随第一个线程中的 volatile-write(a) 操作之后，链条也会被打断。

当链条被打断时，无法 guarantee observe() 会看到 b 的正确值。

具有三个线程的易失性

5. write(b) happens-before read(a) AND read(a) happens-before read(b) IMPLIES write(b) happens-before read(b).

Summing up:

6. The write(a) happens-before read(a) relation means that the a + b statement is guaranteed to see the correct value of a.
7. The write(b) happens-before read(b) relation means that the a + b statement is guaranteed to see the correct value of b.

Behavior of 'volatile' in an example with 2 threads

We will use the following example code to explore some implications of the Memory Model for `volatile`.

```
public class VolatileExample {  
    private volatile int a;  
    private int b; // NOT volatile  
  
    public void update(int first, int second) {  
        b = first; // write(b)  
        a = second; // write-volatile(a)  
    }  
  
    public int observe() {  
        return a + b; // read-volatile(a) followed by read(b)  
    }  
}
```

First, consider the following sequence of statements involving 2 threads:

1. A single instance of VolatileExample is created; call it ve,
2. ve.update(1, 2) is called in one thread, and
3. ve.observe() is called in another thread.

By Happens-Before Rule #1:

1. The write(a) action happens-before the volatile-write(a) action.
2. The volatile-read(a) action happens-before the read(b) action.

By Happens-Before Rule #2:

3. The volatile-write(a) action in the first thread happens-before the volatile-read(a) action in the second thread.

By Happens-Before Rule #4:

4. The write(b) action in the first thread happens-before the read(b) action in the second thread.

In other words, for this particular sequence, we are guaranteed that the 2nd thread will see the update to the non-volatile variable b made by the first thread. However, it is also clear that if the assignments in the update method were the other way around, or the observe() method read the variable b before a, then the happens-before chain would be broken. The chain would also be broken if volatile-read(a) in the second thread was not subsequent to the volatile-write(a) in the first thread.

When the chain is broken, there is no guarantee that observe() will see the correct value of b.

Volatile with three threads

假设我们在前面的例子中添加第三个线程：

1. 创建了一个VolatileExample的单一实例；称之为ve，
2. 两个线程调用update：
 - ve.update(1, 2)在一个线程中被调用，
 - ve.update(3, 4)在第二个线程中被调用，
3. ve.observe()随后在第三个线程中被调用。

要完全分析这个情况，我们需要考虑线程一
和线程二中语句的所有可能交错执行。这里，我们只考虑其中两种情况。

场景#1 - 假设update(1, 2)先于update(3,4)执行，我们得到以下序列：

```
write(b, 1), write-volatile(a, 2)    // 第一个线程
write(b, 3), write-volatile(a, 4)    // 第二个线程
read-volatile(a), read(b)           // 第三个线程
```

在这种情况下，很容易看出从write(b, 3)到read(b)存在一条连续的happens-before链。
此外，中间没有对b的写操作。因此，对于该场景，第三个线程保证能看到b的值为3。

场景#2 - 假设update(1, 2)和update(3,4)重叠执行，操作交错如下：

```
write(b, 3)                      // 第二个线程
write(b, 1)                      // 第一个线程
write-volatile(a, 2)              // 第一个线程
write-volatile(a, 4)              // 第二个线程
read-volatile(a), read(b)         // 第三个线程
```

现在，虽然存在从write(b, 3)到read(b)的happens-before链，但中间有另一个线程执行的write(b, 1)操作。这意味着我们无法确定read(b)会看到哪个值。

(顺便说一句：这表明我们不能依赖volatile来确保非volatile变量的可见性，除非在非常有限的情况下。)

Suppose we add a third thread into the previous example:

1. A single instance of VolatileExample is created; call it ve,
2. Two threads call update:
 - ve.update(1, 2) is called in one thread,
 - ve.update(3, 4) is called in the second thread,
3. ve.observe() is subsequently called in a third thread.

To analyse this completely, we need to consider all of the possible interleavings of the statements in thread one
and thread two. Instead, we will consider just two of them.

Scenario #1 - suppose that update(1, 2) precedes update(3, 4) we get this sequence:

```
write(b, 1), write-volatile(a, 2)    // first thread
write(b, 3), write-volatile(a, 4)    // second thread
read-volatile(a), read(b)           // third thread
```

In this case, it is easy to see that there is an unbroken *happens-before* chain from write(b, 3) to read(b).
Furthermore there is no intervening write to b. So, for this scenario, the third thread is guaranteed to see b as
having value 3.

Scenario #2 - suppose that update(1, 2) and update(3, 4) overlap and the actions are interleaved as follows:

```
write(b, 3)                      // second thread
write(b, 1)                      // first thread
write-volatile(a, 2)              // first thread
write-volatile(a, 4)              // second thread
read-volatile(a), read(b)         // third thread
```

Now, while there is a *happens-before* chain from write(b, 3) to read(b), there is an intervening write(b, 1)
action performed by the other thread. This means we cannot be certain which value read(b) will see.

(Aside: This demonstrates that we cannot rely on **volatile** for ensuring visibility of non-volatile variables, except in
very limited situations.)

第148章：Java部署

有多种技术用于“打包”Java应用程序、Web应用等，以便部署到它们将运行的平台。它们包括简单的库或可执行的JAR文件、WAR和EAR文件，以及安装程序和独立可执行文件。

第148.1节：从命令行制作可执行的JAR

制作JAR需要一个或多个类文件。如果要通过双击运行，类文件应包含main方法。

在本例中，我们将使用：

```
import javax.swing.*;
import java.awt.Container;

public class HelloWorld {

    public static void main(String[] args) {
        JFrame f = new JFrame("Hello, World");
        JLabel label = new JLabel("Hello, World");
        Container cont = f.getContentPane();
        cont.add(label);
        f.setSize(400,100);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

它被命名为 HelloWorld.java

接下来，我们要编译这个程序。

你可以使用任何程序来完成这一步。要从命令行运行，请参阅关于编译和运行你的第一个 Java 程序的文档。

一旦你有了 HelloWorld.class，创建一个新文件夹，命名为你想要的名字。

再创建一个名为 manifest.txt 的文件，并将以下内容粘贴进去

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

将它放在与 HelloWorld.class 相同的文件夹中

使用命令行将当前目录（Windows 上为cd C:\Your\Folder\Path\Here）设置为你的文件夹。

使用终端并切换目录到你的文件夹（Mac 上为cd /Users/user/Documents/Java/jarfolder）。

完成后，输入jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class并按回车。这会使用指定的.class文件创建一个jar文件（在包含manifest和HelloWorld.class的文件夹中），命名为HelloWorld.jar。有关选项（如-m和-v）的信息，请参见语法部分。

完成这些步骤后，进入包含manifest文件的目录，你应该能找到HelloWorld.jar。点击它应会在文本框中显示Hello, World。

Chapter 148: Java deployment

There are a variety of technologies for "packaging" Java applications, webapps and so forth, for deployment to the platform on which they will run. They range from simple library or executable JAR files, WAR and EAR files, through to installers and self-contained executables.

Section 148.1: Making an executable JAR from the command line

To make a jar, you need one or more class files. This should have a main method if it is to be run by a double click.

For this example, we will use:

```
import javax.swing.*;
import java.awt.Container;

public class HelloWorld {

    public static void main(String[] args) {
        JFrame f = new JFrame("Hello, World");
        JLabel label = new JLabel("Hello, World");
        Container cont = f.getContentPane();
        cont.add(label);
        f.setSize(400,100);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

It has been named HelloWorld.java

Next, we want to compile this program.

You may use any program you want to do this. To run from the command line, see the documentation on compiling and running your first java program.

Once you have HelloWorld.class, make a new folder and call it whatever you want.

Make another file called manifest.txt and paste into it

```
Main-Class: HelloWorld
Class-Path: HelloWorld.jar
```

Put it in the same folder with HelloWorld.class

Use the command line to make your current directory (cd C:\Your\Folder\Path\Here on windows) your folder.

Use Terminal and change directory to the directory (cd /Users/user/Documents/Java/jarfolder on Mac) your folder

When that is done, type in jar -cvfm HelloWorld.jar manifest.txt HelloWorld.class and press enter. This makes a jar file (in the folder with your manifest and HelloWorld.class) using the .class files specified and named HelloWorld.jar. See the Syntax section for information about the options (like -m and -v). After these steps, go to your directory with the manifest file and you should find HelloWorld.jar. Clicking on it should display Hello, World in a text box.

第148.2节：为应用程序及其依赖项创建UberJAR

Java应用程序的一个常见需求是能够通过复制单个文件来部署。对于仅依赖标准Java SE类库的简单应用程序，这一需求可以通过创建包含所有（已编译）应用程序类的JAR文件来满足。

如果应用程序依赖第三方库，情况就不那么简单了。如果你只是将依赖的JAR文件放入应用程序JAR中，标准的Java类加载器将无法找到库类，应用程序也无法启动。相反，你需要创建一个包含应用程序类及相关资源以及依赖类和资源的单一JAR文件。这些内容需要组织成类加载器可以搜索的单一命名空间。

这样的JAR文件通常被称为UberJAR。

使用“jar”命令创建UberJAR

创建 UberJAR 的过程很简单。（为了简便，我将使用 Linux 命令。命令在 Mac OS 上应完全相同，在 Windows 上则类似。）

1. 创建一个临时目录，并切换到该目录。

```
$ mkdir tempDir  
$ cd tempDir
```

2. 对于每个依赖的JAR文件，按照它们需要出现在应用程序类路径上的逆序，使用 jar命令将JAR解包到临时目录中。

```
$ jar -xf <path/to/file.jar>
```

对多个JAR文件执行此操作将会覆盖JAR文件的内容。

3. 将构建树中的应用程序类复制到临时目录中

```
$ cp -r path/to/classes .
```

4. 从临时目录的内容创建UberJAR：

```
$ jar -cf ..//myApplication.jar
```

如果您正在创建可执行的JAR文件，请按照此处所述包含适当的MANIFEST.MF。

使用Maven创建UberJAR

如果您的项目是使用Maven构建的，您可以使用“maven-assembly”或“maven-shade”插件来创建UberJAR。详情请参见Maven文档中的Maven Assembly主题。

UberJAR的优点和缺点

UberJAR的一些优点是不言而喻的：

- UberJAR易于分发。

Section 148.2: Creating an UberJAR for an application and its dependencies

A common requirement for a Java application is that can be deployed by copying a single file. For simple applications that depend only on the standard Java SE class libraries, this requirement is satisfied by creating a JAR file containing all of the (compiled) application classes.

Things are not so straightforward if the application depends on third-party libraries. If you simply put dependency JAR files inside an application JAR, the standard Java class loader will not be able to find the library classes, and your application will not start. Instead, you need to create a single JAR file that contains the application classes and associated resources together with the dependency classes and resources. These need to be organized as a single namespace for the classloader to search.

The such a JAR file is often referred to as an UberJAR.

Creating an UberJAR using the "jar" command

The procedure for creating an UberJAR is straight-forward. (I will use Linux commands for simplicity. The commands should be identical for Mac OS, and similar for Windows.)

1. Create a temporary directory, and change directory to it.

```
$ mkdir tempDir  
$ cd tempDir
```

2. For each dependent JAR file, *in the reverse order* that they need to appear on the application's classpath, used the jar command to unpack the JAR into the temporary directory.

```
$ jar -xf <path/to/file.jar>
```

Doing this for multiple JAR files will *overlay* contents of the JARs.

3. Copy the application classes from the build tree into the temporary directory

```
$ cp -r path/to/classes .
```

4. Create the UberJAR from the contents of the temporary directory:

```
$ jar -cf ..//myApplication.jar
```

If you are creating an executable JAR file, include an appropriate MANIFEST.MF as described here.

Creating an UberJAR using Maven

If your project is built using Maven, you can get it to create an UberJAR using either the "maven-assembly" or "maven-shade" plugins. See the Maven Assembly topic (in the Maven documentation) for details.

The advantages and drawbacks of UberJARs

Some of advantages of UberJARs are self-evident:

- An UberJAR is easy to distribute.

- 您无法破坏UberJAR的库依赖关系，因为库是自包含的。

此外，如果您使用适当的工具来创建UberJAR，您可以选择从JAR文件中排除未使用的库类。然而，这通常是通过对类的静态分析完成的。如果您的应用程序使用反射、注解处理和类似技术，您需要小心不要错误地排除类。

UberJAR也有一些缺点：

- 如果你有很多包含相同依赖项的UberJAR，那么每个UberJAR都会包含一份依赖项的副本。
- 一些开源库的许可证可能会限制它们在UberJAR中的使用。

1 - 一些开源库许可证允许您使用该库，前提是最终用户能够替换库的一个版本为另一个版本。UberJAR可能会使版本依赖的替换变得困难。

第148.3节：创建JAR、WAR和EAR文件

JAR、WAR和EAR文件类型本质上是带有“清单”文件的ZIP文件，并且（对于WAR和EAR文件）具有特定的内部目录/文件结构。

创建这些文件的推荐方法是使用专门针对Java的构建工具，该工具“理解”各自文件类型的要求。如果不使用构建工具，那么IDE的“导出”功能是下一个可尝试的选项。

*(编辑注：关于如何创建这些文件的描述最好放在各自工具的文档中。
请放在那里。请自我克制，别把它们强行塞进这个示例中！)*

使用Maven创建JAR和WAR文件

使用Maven创建JAR或WAR文件，只需在POM文件中放入正确的<packaging>元素；例如，

<packaging>jar</packaging>

或者

<packaging>war</packaging>

有关更多细节，Maven可以通过为 maven jar 插件添加入口类和外部依赖的相关信息作为插件属性，配置为创建“可执行”JAR文件。甚至还有一个插件可以创建将应用程序及其依赖合并为单个JAR文件的“uberJAR”。

请参阅 Maven 文档 (<http://stackoverflow.com/documentation/maven/topics>) 以获取更多信息。

使用 Ant 创建 JAR、WAR 和 EAR 文件

Ant 构建工具有专门的“任务”用于构建 JAR、WAR 和 EAR。请参阅 Ant 文档 (<http://stackoverflow.com/documentation/ant/topics>) 以获取更多信息。

使用 IDE 创建 JAR、WAR 和 EAR 文件

三大流行的 Java IDE 都内置了创建部署文件的支持。该功能通常被描述为“导出”。

- You cannot break the library dependencies for an UberJAR, since the libraries are self-contained.

In addition, if you use an appropriate tooling to create the UberJAR, you will have the option of excluding library classes that are not used from the JAR file. However, that this is typically done by static analysis of the classes. If your application uses reflection, annotation processing and similar techniques, you need to be careful that classes are not excluded incorrectly.

UberJARs also have some disadvantages:

- If you have lots of UberJARs with the same dependencies, then each one will contain a copy of the dependencies.
- Some open source libraries have licenses which *may* preclude their use in an UberJAR.

1 - Some open source library licenses allow you to use the library only if the end-user is able to replace one version of the library with another. UberJARs can make replacement of version dependencies difficult.

Section 148.3: Creating JAR, WAR and EAR files

The JAR, WAR and EAR files types are fundamentally ZIP files with a "manifest" file and (for WAR and EAR files) a particular internal directory / file structure.

The recommended way to create these files is to use a Java-specific build tool which "understands" the requirements for the respective file types. If you don't use a build tool, then IDE "export" is the next option to try.

*(Editorial note: the descriptions of how to create these files are best placed in the documentation for the respective tools.
Put them there. Please show some self-restraint and DON'T shoe-horn them into this example!)*

Creating JAR and WAR files using Maven

Creating a JAR or WAR using Maven is simply a matter of putting the correct <packaging> element into the POM file; e.g,

<packaging>jar</packaging>

or

<packaging>war</packaging>

For more details. Maven can be configured to create "executable" JAR files by adding the requisite information about the entry-point class and external dependencies as plugin properties for the maven jar plugin. There is even a plugin for creating "uberJAR" files that combine an application and its dependencies into a single JAR file.

Please refer to the Maven documentation (<http://stackoverflow.com/documentation/maven/topics>) for more information.

Creating JAR, WAR and EAR files using Ant

The Ant build tool has separate "tasks" for building JAR, WAR and EAR. Please refer to the Ant documentation (<http://stackoverflow.com/documentation/ant/topics>) for more information.

Creating JAR, WAR and EAR files using an IDE

The three most popular Java IDEs all have built-in support for creating deployment files. The functionality is often described as "exporting".

- Eclipse - <http://stackoverflow.com/documentation/eclipse/topics>
- NetBeans - <http://stackoverflow.com/documentation/netbeans/topics>
- IntelliJ-IDEA - 导出

使用 jar 命令创建 JAR、WAR 和 EAR 文件。

也可以使用jar命令“手动”创建这些文件。只需将正确的组件文件按正确的位置组装成文件树，创建一个清单文件，然后运行jar来创建JAR文件。

请参阅jar命令主题（创建和修改JAR文件）以获取更多信息

第148.4节：Java Web Start简介

Oracle Java教程将Web Start总结如下：

Java Web Start软件提供了一键启动全功能应用程序的能力。用户可以下载并启动应用程序，例如完整的电子表格程序或互联网聊天客户端，而无需经过繁琐的安装过程。

Java Web Start的其他优点包括支持签名代码和明确声明平台依赖性，以及支持代码缓存和应用程序更新的部署。

Java Web Start也称为JavaWS和JAWS。主要的信息来源有：

- [Java教程 - 课程 : Java Web Start](#)
- [Java Web Start指南](#)
- [Java Web Start常见问题解答](#)
- [JNLP规范](#)
- [javax.jnlp API 文档](#)
- [Java Web Start 开发者网站](#)

先决条件

从高层次来看，Web Start 通过从远程网页服务器分发打包为 JAR 文件的 Java 应用程序来工作。先决条件包括：

- 目标机器上必须预先安装 Java (JRE 或 JDK)。要求 Java 版本为 1.2.2 或更高版本：
 - 从 Java 5.0 开始，Web Start 支持已包含在 JRE / JDK 中。
 - 对于早期版本，Web Start 支持需要单独安装。
 - Web Start 基础设施包括一些可以嵌入网页的 Javascript，用于帮助用户安装所需的软件。
- 托管软件的网页服务器必须对目标机器可访问。
- 如果用户打算通过网页中的链接启动 Web Start 应用程序，则：
 - 他们需要一个兼容的网页浏览器，且
 - 对于现代（安全）浏览器，需要告诉浏览器如何允许Java运行.....而不影响网页浏览器的安全性。

- Eclipse - <http://stackoverflow.com/documentation/eclipse/topics>
- NetBeans - <http://stackoverflow.com/documentation/netbeans/topics>
- IntelliJ-IDEA - Exporting

Creating JAR, WAR and EAR files using the jar command.

It is also possible to create these files "by hand" using the jar command. It is simply a matter of assembling a file tree with the correct component files in the correct place, creating a manifest file, and running jar to create the JAR file.

Please refer to the jar command Topic (Creating and modifying JAR files) for more information

Section 148.4: Introduction to Java Web Start

The Oracle Java Tutorials summarize [Web Start](#) as follows:

Java Web Start software provides the power to launch full-featured applications with a single click. Users can download and launch applications, such as a complete spreadsheet program or an Internet chat client, without going through lengthy installation procedures.

Other advantages of Java Web Start are support for signed code and explicit declaration of platform dependencies, and support for code caching and deployment of application updates.

Java Web Start is also referred to as JavaWS and JAWS. The primary sources of information are:

- [The Java Tutorials - Lesson: Java Web Start](#)
- [Java Web Start Guide](#)
- [Java Web Start FAQ](#)
- [JNLP Specification](#)
- [javax.jnlp API Documentation](#)
- [Java Web Start Developers Site](#)

Prerequisites

At a high level, Web Start works by distributing Java applications packed as JAR files from a remote webserver. The prerequisites are:

- A pre-existing Java installation (JRE or JDK) on the target machine where the application is to run. Java 1.2.2 or higher is required:
 - From Java 5.0 onwards, Web Start support is included in the JRE / JDK.
 - For earlier releases, Web Start support is installed separately.
 - The Web Start infrastructure includes some Javascript that can be included in a web page to assist the user to install the necessary software.
- The webserver that hosts the software must be accessible to the target machine.
- If the user is going to launch a Web Start application using a link in a web page, then:
 - they need a compatible web browser, and
 - for modern (secure) browsers, they need to be told how to tell the browser to allow Java to run ... without compromising web browser security.

一个示例JNLP文件

下面的示例旨在说明JNLP的基本功能。

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="https://www.example.com/demo"
      href="demo_webstart.jnlp">
  <information>
    <title>演示</title>
    <vendor>Example.com团队</vendor>
  </information>
  <resources>
    <!-- 应用资源 -->
    <j2se version="1.7+" href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="Demo.jar" main="true"/>
  </resources>
  <application-desc
    name="Demo Application"
    main-class="com.example.jwsdemo.Main"
    width="300"
    height="300">
  </application-desc>
  <update check="background"/>
</jnlp>
```

如您所见，JNLP 文件是基于 XML 的，所有信息都包含在 `<jnlp>` 元素中。

- `spec` 属性给出了该文件所遵循的 JNLP 规范版本。
- `codebase` 属性给出了用于解析文件中相对 `href` URL 的基础 URL。
- `href`属性为此JNLP文件提供了确定的URL。
- 信息`<information>` 元素包含应用程序的元数据，包括其标题、作者、描述和帮助网站。
- 资源`<resources>` 元素描述应用程序的依赖项，包括所需的Java版本、操作系统平台和JAR文件。
- 应用描述`<application-desc>`（或`<applet-desc>`）元素提供启动

应用程序所需的信息。

设置网络服务器

必须将网络服务器配置为使用`application/x-java-jnlp-file`作为.jnlp文件的MIME类型。

JNLP文件和应用程序的JAR文件必须安装在网络服务器上，以便通过JNLP文件指定的URL访问。

启用通过网页启动

如果应用程序通过网页链接启动，则必须在网络服务器上创建包含该链接的网页。

- 如果可以假设用户的计算机上已安装Java Web Start，则网页只需包含一个用于启动应用程序的链接。例如。

```
<a href="https://www.example.com/demo_webstart.jnlp">启动应用程序</a>
```

- 否则，页面还应包含一些脚本，用于检测用户所使用的浏览器类型，并请求下载和安装所需版本的Java。

An example JNLP file

The following example is intended to illustrate the basic functionality of JNLP.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="https://www.example.com/demo"
      href="demo_webstart.jnlp">
  <information>
    <title>Demo</title>
    <vendor>The Example.com Team</vendor>
  </information>
  <resources>
    <!-- Application Resources -->
    <j2se version="1.7+" href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="Demo.jar" main="true"/>
  </resources>
  <application-desc
    name="Demo Application"
    main-class="com.example.jwsdemo.Main"
    width="300"
    height="300">
  </application-desc>
  <update check="background"/>
</jnlp>
```

As you can see, a JNLP file XML-based, and the information is all contained in the `<jnlp>` element.

- The `spec` attribute gives the version of the JNLP spec that this file conforms to.
- The `codebase` attribute gives the base URL for resolving relative `href` URLs in the rest of the file.
- The `href` attribute gives the definitive URL for this JNLP file.
- The `<information>` element contains metadata the application including its title, authors, description and help website.
- The `<resources>` element describes the dependencies for the application including the required Java version, OS platform and JAR files.
- The `<application-desc>` (or `<applet-desc>`) element provides information needed to launch the application.

Setting up the web server

The webserver must be configured to use `application/x-java-jnlp-file` as the MIMETYPE for `.jnlp` files.

The JNLP file and the application's JAR files must be installed on the webserver so that they are available using the URLs indicated by the JNLP file.

Enabling launch via a web page

If the application is to be launched via a web link, the page that contains the link must be created on the webserver.

- If you can assume that Java Web Start is already installed on the user's machine, then the web page simply needs to contain a link for launching the application. For example.

```
<a href="https://www.example.com/demo_webstart.jnlp">Launch the application</a>
```

- Otherwise, the page should also include some scripting to detect the kind of browser the user is using and request to download and install the required version of Java.

注意： 鼓励用户通过这种方式安装Java，甚至启用浏览器中的Java以使JNLP网页启动功能正常工作，是一个不好的做法。

从命令行启动Web Start应用程序

从命令行启动Web Start应用程序的说明很简单。假设用户已经安装了Java 5.0的JRE或JDK，只需运行以下命令：

```
$ javaws <url>
```

其中<url>是远程服务器上JNLP文件的URL。

NOTE: It is a bad idea to encourage users to encourage to install Java this way, or even to enable Java in their web browsers so that JNLP web page launch will work.

Launching Web Start applications from the command line

The instructions for launching an Web Start application from the command line are simple. Assuming that the user has a Java 5.0 JRE or JDK installed, the simply need to run this:

```
$ javaws <url>
```

where **<url>** is the URL for the JNLP file on the remote server.

第149章：Java插件系统实现

第149.1节：使用URLClassLoader

有几种方法可以为Java应用程序实现插件系统。其中最简单的方法之一是使用URLClassLoader。以下示例将涉及一些JavaFX代码。

假设我们有一个主应用程序的模块。该模块应该从

'plugins'文件夹中加载以jar形式存在的插件。初始代码：

```
package main;

public class MainApplication extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins"); //任意目录
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        VBox loadedPlugins=new VBox(6); //一个用于稍后显示视觉信息的容器
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
        launch(a);
    }
}
```

然后，我们创建一个表示插件的接口。

```
package main;

public interface Plugin
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

我们想加载实现此接口的类，因此首先需要过滤扩展名为".jar"的文件：

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

如果有任何文件，我们需要创建 URL 和类名的集合：

```
if(files!=null && files.length>0)
{
    ArrayList<String> classes=new ArrayList<>();
    ArrayList<URL> urls=new ArrayList<>(files.length);
    for(File file:files)
    {
```

Chapter 149: Java plugin system implementations

Section 149.1: Using URLClassLoader

There are several ways to implement a plugin system for a Java application. One of the simplest is to use *URLClassLoader*. The following example will involve a bit of JavaFX code.

Suppose we have a module of a main application. This module is supposed to load plugins in form of Jars from 'plugins' folder. Initial code:

```
package main;

public class MainApplication extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins"); //arbitrary directory
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
        VBox loadedPlugins=new VBox(6); //a container to show the visual info later
        Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
        Scene scene=new Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] a)
    {
        launch(a);
    }
}
```

Then, we create an interface which will represent a plugin.

```
package main;

public interface Plugin
{
    default void initialize()
    {
        System.out.println("Initialized "+this.getClass().getName());
    }
    default String name(){return getClass().getSimpleName();}
}
```

We want to load classes which implement this interface, so first we need to filter files which have a '.jar' extension:

```
File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
```

If there are any files, we need to create collections of URLs and class names:

```
if(files!=null && files.length>0)
{
    ArrayList<String> classes=new ArrayList<>();
    ArrayList<URL> urls=new ArrayList<>(files.length);
    for(File file:files)
    {
```

```

JarFile jar=new JarFile(file);
jar.stream().forEach(jarEntry -> {
    if(jarEntry.getName().endsWith(".class"))
    {
        classes.add(jarEntry.getName());
    }
});
URL url=file.toURI().toURL();
urls.add(url);
}
}

```

让我们向MainApplication添加一个静态的HashSet，用于保存已加载的插件：

```
static HashSet<Plugin> plugins=new HashSet<>();
```

接下来，我们实例化一个URLClassLoader，并遍历类名，实例化实现了Plugin接口的类：

```

URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
classes.forEach(className->{
    try
    {
        Class cls=urlClassLoader.loadClass(className.replaceAll("/", ".").replace(".class", ""));
        //转换为二进制名称
        Class[] interfaces=cls.getInterfaces();
        for(Class intface:interfaces)
        {
            if(intface.equals(Plugin.class)) //检查是否存在 Plugin 接口
            {
                Plugin plugin=(Plugin) cls.newInstance(); //实例化 Plugin
                plugins.add(plugin);
                break;
            }
        }
    } catch (Exception e){e.printStackTrace();}
});

```

然后，我们可以调用插件的方法，例如，初始化它们：

```

if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("已加载的插件:"));
plugins.forEach(plugin -> {
plugin.initialize();
loadedPlugins.getChildren().add(new Label(plugin.name()));
});

```

主应用程序的最终代码：

```

package main;
public class MainApplication extends Application
{
    static HashSet<Plugin> plugins=new HashSet<>();
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins");
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
    }
}

```

```

JarFile jar=new JarFile(file);
jar.stream().forEach(jarEntry -> {
    if(jarEntry.getName().endsWith(".class"))
    {
        classes.add(jarEntry.getName());
    }
});
URL url=file.toURI().toURL();
urls.add(url);
}
}

```

Let's add a static HashSet to *MainApplication* which will hold loaded plugins:

```
static HashSet<Plugin> plugins=new HashSet<>();
```

Next, we instantiate a *URLClassLoader*, and iterate over class names, instantiating classes which implement *Plugin* interface:

```

URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
classes.forEach(className->{
    try
    {
        Class cls=urlClassLoader.loadClass(className.replaceAll("/", ".").replace(".class", ""));
        //transforming to binary name
        Class[] interfaces=cls.getInterfaces();
        for(Class intface:interfaces)
        {
            if(intface.equals(Plugin.class)) //checking presence of Plugin interface
            {
                Plugin plugin=(Plugin) cls.newInstance(); //instantiating the Plugin
                plugins.add(plugin);
                break;
            }
        }
    } catch (Exception e){e.printStackTrace();}
});

```

Then, we can call plugin's methods, for example, to initialize them:

```

if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
plugins.forEach(plugin -> {
plugin.initialize();
loadedPlugins.getChildren().add(new Label(plugin.name()));
});

```

The final code of *MainApplication*:

```

package main;
public class MainApplication extends Application
{
    static HashSet<Plugin> plugins=new HashSet<>();
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        File pluginDirectory=new File("plugins");
        if(!pluginDirectory.exists())pluginDirectory.mkdir();
    }
}

```

```

File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
VBox loadedPlugins=new VBox(6);
loadedPlugins.setAlignment(Pos.CENTER);
if(files!=null && files.length>0)
{
    ArrayList<String> classes=new ArrayList<>();
    ArrayList<URL> urls=new ArrayList<>(files.length);
    for(File file:files)
    {
        JarFile jar=new JarFile(file);
        jar.stream().forEach(jarEntry -> {
            if(jarEntry.getName().endsWith(".class"))
            {
                classes.add(jarEntry.getName());
            }
        });
        URL url=file.toURI().toURL();
        urls.add(url);
    }
    URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
    classes.forEach(className->{
        try
        {
            Class
            cls=urlClassLoader.loadClass(className.replaceAll("/", ".").replace(".class", ""));
            Class[] interfaces=cls.getInterfaces();
            for(Class intface:interfaces)
            {
                if(intface.equals(Plugin.class))
                {
                    Plugin plugin=(Plugin) cls.newInstance();
                    plugins.add(plugin);
                    break;
                }
            }
        }
        catch (Exception e){e.printStackTrace();}
    });
    if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
    plugins.forEach(plugin -> {
        plugin.initialize();
        loadedPlugins.getChildren().add(new Label(plugin.name()));
    });
}
Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
Scene scene=new Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
primaryStage.setScene(scene);
primaryStage.show();
}

public static void main(String[] a)
{
    launch(a);
}
}

```

让我们创建两个插件。显然，插件的源代码应该放在一个单独的模块中。

```

package plugins;

import main.Plugin;

```

```

File[] files=pluginDirectory.listFiles((dir, name) -> name.endsWith(".jar"));
VBox loadedPlugins=new VBox(6);
loadedPlugins.setAlignment(Pos.CENTER);
if(files!=null && files.length>0)
{
    ArrayList<String> classes=new ArrayList<>();
    ArrayList<URL> urls=new ArrayList<>(files.length);
    for(File file:files)
    {
        JarFile jar=new JarFile(file);
        jar.stream().forEach(jarEntry -> {
            if(jarEntry.getName().endsWith(".class"))
            {
                classes.add(jarEntry.getName());
            }
        });
        URL url=file.toURI().toURL();
        urls.add(url);
    }
    URLClassLoader urlClassLoader=new URLClassLoader(urls.toArray(new URL[urls.size()]));
    classes.forEach(className->{
        try
        {
            Class
            cls=urlClassLoader.loadClass(className.replaceAll("/", ".").replace(".class", ""));
            Class[] interfaces=cls.getInterfaces();
            for(Class intface:interfaces)
            {
                if(intface.equals(Plugin.class))
                {
                    Plugin plugin=(Plugin) cls.newInstance();
                    plugins.add(plugin);
                    break;
                }
            }
        }
        catch (Exception e){e.printStackTrace();}
    });
    if(!plugins.isEmpty())loadedPlugins.getChildren().add(new Label("Loaded plugins:"));
    plugins.forEach(plugin -> {
        plugin.initialize();
        loadedPlugins.getChildren().add(new Label(plugin.name()));
    });
}
Rectangle2D screenbounds=Screen.getPrimary().getVisualBounds();
Scene scene=new Scene(loadedPlugins,screenbounds.getWidth()/2,screenbounds.getHeight()/2);
primaryStage.setScene(scene);
primaryStage.show();
}

public static void main(String[] a)
{
    launch(a);
}
}

```

Let's create two plugins. Obviously, the plugin's source should be in a separate module.

```

package plugins;

import main.Plugin;

```

```
public class FirstPlugin implements Plugin
{
    //这个插件具有默认行为
}
```

第二个插件：

```
package plugins;

import main.Plugin;

public class AnotherPlugin implements Plugin
{
    @Override
    public void initialize() //重写以显示用户的主目录
    {
        System.out.println("用户主目录: "+System.getProperty("user.home"));
    }
}
```

这些插件必须打包成标准的Jar包——这个过程取决于你的IDE或其他工具。

当Jar包直接放入“plugins”目录时，MainApplication 会检测它们并实例化相应的类。

```
public class FirstPlugin implements Plugin
{
    //this plugin has default behaviour
}
```

Second plugin:

```
package plugins;

import main.Plugin;

public class AnotherPlugin implements Plugin
{
    @Override
    public void initialize() //overrided to show user's home directory
    {
        System.out.println("User home directory: "+System.getProperty("user.home"));
    }
}
```

These plugins have to be packaged into standard Jars - this process depends on your IDE or other tools.

When Jars will be put into 'plugins' directly, *MainApplication* will detect them and instantiate appropriate classes.

第150章：JavaBean

JavaBeans (商标) 是一种设计Java类API的模式，允许实例 (bean) 在各种环境中使用并通过各种工具无需显式编写Java代码。该模式包括定义属性的getter和setter方法的约定，定义构造函数，以及定义事件监听器API的约定。

第150.1节：基础Java Bean

```
public class BasicJavaBean implements java.io.Serializable{  
  
    private int value1;  
    private String value2;  
    private boolean value3;  
  
    public BasicJavaBean(){  
  
        public void setValue1(int value1){  
            this.value1 = value1;  
        }  
  
        public int getValue1(){  
            return value1;  
        }  
  
        public void setValue2(String value2){  
            this.value2 = value2;  
        }  
  
        public String getValue2(){  
            return value2;  
        }  
  
        public void setValue3(boolean value3){  
            this.value3 = value3;  
        }  
  
        public boolean isValue3(){  
            return value3;  
        }  
    }
```

Chapter 150: JavaBean

JavaBeans (TM) is a pattern for designing Java class APIs that allows instances (beans) to be used in various contexts and using various tools *without* explicitly writing Java code. The patterns consists of conventions for defining getters and setters for *properties*, for defining constructors, and for defining event listener APIs.

Section 150.1: Basic Java Bean

```
public class BasicJavaBean implements java.io.Serializable{  
  
    private int value1;  
    private String value2;  
    private boolean value3;  
  
    public BasicJavaBean(){  
  
        public void setValue1(int value1){  
            this.value1 = value1;  
        }  
  
        public int getValue1(){  
            return value1;  
        }  
  
        public void setValue2(String value2){  
            this.value2 = value2;  
        }  
  
        public String getValue2(){  
            return value2;  
        }  
  
        public void setValue3(boolean value3){  
            this.value3 = value3;  
        }  
  
        public boolean isValue3(){  
            return value3;  
        }  
    }
```

第151章：Java SE 7特性

本主题将总结Java SE 7中Java编程语言新增的特性。其他领域如JDBC和Java虚拟机（JVM）也有许多新特性，但本主题不涉及这些内容。

第151.1节：Java SE 7编程语言新特性

- [二进制字面量](#)：整型（byte、short、int和long）也可以用二进制数表示。要指定二进制字面量，在数字前加上前缀0b或0B。
- [switch语句中的字符串](#)：可以在switch语句的表达式中使用String对象。
- [try-with-resources语句](#)是声明一个或多个资源的try语句。资源是程序使用完后必须关闭的对象。try-with-resources语句确保每个资源在语句结束时被关闭。任何实现了java.lang.AutoCloseable接口的对象（包括所有实现了java.io.Closeable的对象）都可以作为资源使用。
- [捕获多种异常类型及改进类型检查的重新抛出异常](#)：单个catch块可以处理多种类型的异常。此特性可以减少代码重复，降低捕获过于宽泛异常的诱惑。
- [数字字面量中的下划线](#)：数字字面量中的数字之间可以出现任意数量的下划线字符（_）。此特性允许你例如在数字字面量中分隔数字组，从而提高代码的可读性。
- [泛型实例创建的类型推断](#)：只要编译器能够从上下文中推断出类型参数，就可以用一对空的类型参数(<>)替代调用泛型类构造函数时所需的类型参数。这对尖括号非正式地称为“菱形”。
- [使用不可重构的形式参数和可变参数时改进的编译器警告和错误方法](#)

第151.2节：二进制字面量

```
// 一个8位的"byte"值：  
byte aByte = (byte)0b00100001;  
  
// 一个16位的"short"值：  
short aShort = (short)0b1010000101000101;  
  
// 一些32位的"int"值：  
int anInt1 = 0b10100001010001011010000101000101;  
int anInt2 = 0b101;  
int anInt3 = 0B101; // B可以是大写或小写。  
  
// 一个64位的"long"值。注意"L"后缀：  
long aLong = 0b101000010100010110100001010001011010000101000101L;
```

第151.3节：带资源的try语句

该示例从文件中读取第一行。它使用BufferedReader的实例从文件中读取数据。

BufferedReader是一种资源，程序使用完后必须关闭：

```
static String readFirstLineFromFile(String path) throws IOException {  
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }
```

Chapter 151: Java SE 7 Features

In this topic you'll find a summary of the new features added to the Java programming language in Java SE 7. There are many other new features in other fields such as JDBC and Java Virtual Machine (JVM) that are not going to be covered in this topic.

Section 151.1: New Java SE 7 programming language features

- [Binary Literals](#): The integral types (byte, short, int, and long) can also be expressed using the binary number system. To specify a binary literal, add the prefix 0b or 0B to the number.
- [Strings in switch Statements](#): You can use a String object in the expression of a switch statement
- [The try-with-resources Statement](#): The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.
- [Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking](#): a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.
- [Underscores in Numeric Literals](#): Any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.
- [Type Inference for Generic Instance Creation](#): You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.
- [Improved Compiler Warnings and Errors When Using Non-Reifiable Formal Parameters with Varargs Methods](#)

Section 151.2: Binary Literals

```
// An 8-bit 'byte' value:  
byte aByte = (byte)0b00100001;  
  
// A 16-bit 'short' value:  
short aShort = (short)0b1010000101000101;  
  
// Some 32-bit 'int' values:  
int anInt1 = 0b10100001010001011010000101000101;  
int anInt2 = 0b101;  
int anInt3 = 0B101; // The B can be upper or lower case.  
  
// A 64-bit 'long' value. Note the "L" suffix:  
long aLong = 0b101000010100010110100001010001011010000101000101L;
```

Section 151.3: The try-with-resources statement

The example reads the first line from a file. It uses an instance of [BufferedReader](#) to read data from the file. [BufferedReader](#) is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {  
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }
```

```
}
```

在此示例中，try-with-resources语句中声明的资源是BufferedReader。声明语句出现在try关键字后紧跟的括号内。Java SE 7及更高版本中的BufferedReader类实现了接口java.lang.AutoCloseable。由于BufferedReader实例是在try-with-resources语句中声明的，无论try语句是正常完成还是异常终止（例如BufferedReader.readLine方法抛出IOException），该资源都会被关闭。

第151.4节：数字字面量中的下划线

下面的示例展示了在数字字面量中使用下划线的其他方式：

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

下划线只能放在数字之间；不能将下划线放在以下位置：

- 数字的开头或结尾
- 浮点字面量中小数点的相邻位置
- F或L后缀之前
- 期望数字串的位置

第151.5节：泛型实例创建的类型推断

你可以使用

```
Map<String, List<String>> myMap = new HashMap<>();
```

而不是

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

但是，你不能使用

```
List<String> list = new ArrayList<>();
list.add("A");

// 以下语句应失败，因为 addAll 期望的是
// Collection<? extends String>

list.addAll(new ArrayList<>());
```

因为它无法编译。请注意，菱形语法通常在方法调用中有效；然而，建议主要在变量声明时使用菱形语法。

第151.6节：switch语句中的字符串

```
public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg) {
    String typeOfDay;
```

```
}
```

In this example, the resource declared in the try-with-resources statement is a `BufferedReader`. The declaration statement appears within parentheses immediately after the `try` keyword. The class `BufferedReader`, in Java SE 7 and later, implements the interface `java.lang.AutoCloseable`. Because the `BufferedReader` instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method `BufferedReader.readLine` throwing an `IOException`).

Section 151.4: Underscores in Numeric Literals

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

You can place underscores only between digits; you cannot place underscores in the following places:

- At the beginning or end of a number
- Adjacent to a decimal point in a floating point literal
- Prior to an F or L suffix
- In positions where a string of digits is expected

Section 151.5: Type Inference for Generic Instance Creation

You can use

```
Map<String, List<String>> myMap = new HashMap<>();
```

instead of

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

However, you can't use

```
List<String> list = new ArrayList<>();
list.add("A");

// The following statement should fail since addAll expects
// Collection<? extends String>

list.addAll(new ArrayList<>());
```

because it can't compile. Note that the diamond often works in method calls; however, it is suggested that you use the diamond primarily for variable declarations.

Section 151.6: Strings in switch Statements

```
public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg) {
    String typeOfDay;
```

```
switch (dayOfWeekArg) {
    case "Monday":
        typeOfDay = "工作周开始";
        break;
    case "Tuesday":
    case "Wednesday":
    case "Thursday":
        typeOfDay = "周中";
        break;
    case "Friday":
        typeOfDay = "工作周末";
        break;
    case "Saturday":
    case "Sunday":
        typeOfDay = "周末";
        break;
    default:
        throw new IllegalArgumentException("无效的星期几: " + dayOfWeekArg);
}
return typeOfDay;
}
```

```
switch (dayOfWeekArg) {
    case "Monday":
        typeOfDay = "Start of work week";
        break;
    case "Tuesday":
    case "Wednesday":
    case "Thursday":
        typeOfDay = "Midweek";
        break;
    case "Friday":
        typeOfDay = "End of work week";
        break;
    case "Saturday":
    case "Sunday":
        typeOfDay = "Weekend";
        break;
    default:
        throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeekArg);
}
return typeOfDay;
}
```

第152章：Java SE 8特性

在本主题中，您将找到Java SE 8中Java编程语言新增功能的摘要。本主题不会涵盖其他领域的许多新功能，例如JDBC和Java虚拟机（JVM）。

第152.1节：Java SE 8编程语言的新特性

- 引入了Lambda表达式这一新的语言特性。它们使您能够将功能作为方法参数，或将代码视为数据。Lambda表达式让您能够更简洁地表达单方法接口（称为函数式接口）的实例。
 - 方法引用为已有名称的方法提供了易读的Lambda表达式。
 - 默认方法使得可以向库的接口添加新功能，并确保与为这些接口旧版本编写的代码的二进制兼容性。
 - 利用Java SE 8中的Lambda表达式和流的新增强API描述了利用Lambda表达式和流的新类和增强类。
- 改进的类型推断——Java编译器利用目标类型推断泛型方法调用的类型参数。表达式的目标类型是Java编译器根据表达式出现的位置所期望的数据类型。例如，您可以在Java SE 7中使用赋值语句的目标类型进行类型推断。然而，在Java SE 8中，您可以在更多上下文中使用目标类型进行类型推断。
 - [Lambda表达式中的目标类型类型推断](#)
 - [重复注解](#)
- 重复注解提供了在同一声明或类型使用上多次应用相同注解类型的能力。
- 类型注解提供了在类型使用的任何位置应用注解的能力，而不仅限于声明。配合可插拔类型系统使用，该特性能够改进代码的类型检查。
- 方法参数反射 - 你可以通过方法`java.lang.reflect.Executable.getParameters`获取任何方法或构造函数的形式参数名称。（类Method和Constructor继承自Executable类，因此继承了方法`Executable.getParameters`）然而，.class文件默认不存储形式参数名称。要在特定的.class文件中存储形式参数名称，从而使反射API能够检索形式参数名称，请使用javac编译器的-parameters选项编译源文件。
- 日期时间API - 在`java.time`中新增了新的时间API。如果使用该API，则无需指定时区。

Chapter 152: Java SE 8 Features

In this topic you'll find a summary of the new features added to the Java programming language in Java SE 8. There are many other new features in other fields such as JDBC and Java Virtual Machine (JVM) that are not going to be covered in this topic.

Section 152.1: New Java SE 8 programming language features

- [Lambda Expressions](#), a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
 - [Method references](#) provide easy-to-read lambda expressions for methods that already have a name.
 - [Default methods](#) enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
 - [New and Enhanced APIs That Take Advantage of Lambda Expressions and Streams](#) in Java SE 8 describe new and enhanced classes that take advantage of lambda expressions and streams.
- Improved Type Inference - The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The target type of an expression is the data type that the Java compiler expects depending on where the expression appears. For example, you can use an assignment statement's target type for type inference in Java SE 7. However, in Java SE 8, you can use the target type for type inference in more contexts.
 - [Target Typing in Lambda Expressions](#)
 - [Type Inference](#)
- [Repeating Annotations](#) provide the ability to apply the same annotation type more than once to the same declaration or type use.
- [Type Annotations](#) provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
- [Method parameter reflection](#) - You can obtain the names of the formal parameters of any method or constructor with the method `java.lang.reflect.Executable.getParameters`. (The classes Method and Constructor extend the class Executable and therefore inherit the method `Executable.getParameters`) However, .class files do not store formal parameter names by default. To store formal parameter names in a particular .class file, and thus enable the Reflection API to retrieve formal parameter names, compile the source file with the -parameters option of the javac compiler.
- Date-time-api - Added new time api in `java.time`. If used this, you don't need to designate timezone.

第153章：动态方法调度

什么是动态方法调度？

动态方法调度是一个过程，其中对重写方法的调用在运行时而非编译时被解析。当通过引用调用重写方法时，Java根据该引用所指向对象的类型确定执行哪个版本的方法。这也被称为运行时多态性。

我们将通过一个示例来说明这一点。

第153.1节：动态方法调度 - 示例代码

抽象类：

```
package base;

/*
抽象类不能被实例化，但可以被子类化
*/
public abstract class ClsVirusScanner {

    //包含一个抽象方法
    public abstract void fnStartScan();

    protected void fnCheckForUpdateVersion(){
        System.out.println("执行病毒扫描器版本检查");
    }

    protected void fnBootTimeScan(){
        System.out.println("执行启动时扫描");
    }
    protected void fnInternetSecurity(){
        System.out.println("扫描互联网安全");
    }

    protected void fnRealTimeScan(){
        System.out.println("执行实时扫描");
    }

    protected void fnVirusMalwareScan(){
        System.out.println("检测病毒和恶意软件");
    }
}
```

子类中重写抽象方法：

```
import base.ClsVirusScanner;

//所有三个子类都继承自基类ClsVirusScanner
//子类1
class Cls付费版本 extends ClsVirusScanner{
    @Override
    public void fn开始扫描() {
        super.fn检查更新版本();
        super.fn启动时扫描();
    }
}
```

Chapter 153: Dynamic Method Dispatch

What is Dynamic Method Dispatch?

Dynamic Method Dispatch is a process in which the call to an overridden method is resolved at runtime rather than at compile-time. When an overridden method is called by a reference, Java determines which version of that method to execute based on the type of object it refers to. This is also known as runtime polymorphism.

We will see this through an example.

Section 153.1: Dynamic Method Dispatch - Example Code

Abstract Class :

```
package base;

/*
Abstract classes cannot be instantiated, but they can be subclassed
*/
public abstract class ClsVirusScanner {

    //With One Abstract method
    public abstract void fnStartScan();

    protected void fnCheckForUpdateVersion(){
        System.out.println("Perform Virus Scanner Version Check");
    }

    protected void fnBootTimeScan(){
        System.out.println("Perform BootTime Scan");
    }
    protected void fnInternetSecurity(){
        System.out.println("Scan for Internet Security");
    }

    protected void fnRealTimeScan(){
        System.out.println("Perform RealTime Scan");
    }

    protected void fnVirusMalwareScan(){
        System.out.println("Detect Virus & Malware");
    }
}
```

Overriding Abstract Method in Child Class :

```
import base.ClsVirusScanner;

//All the 3 child classes inherits the base class ClsVirusScanner
//Child Class 1
class ClsPaidVersion extends ClsVirusScanner{
    @Override
    public void fnStartScan() {
        super.fnCheckForUpdateVersion();
        super.fnBootTimeScan();
    }
}
```

```

super.fn互联网安全();
super.fn实时扫描();
super.fn病毒恶意软件扫描();
}
//Cls付费版本 是一个 ClsVirusScanner
//子类2

class Cls试用版本 extends ClsVirusScanner{
@Override
public void fn开始扫描() {
    super.fn互联网安全();
    super.fn病毒恶意软件扫描();
}
//Cls试用版本 是一个 ClsVirusScanner

//子类3
class Cls免费版本 extends ClsVirusScanner{
@Override
public void fnStartScan() {
    super.fnVirusMalwareScan();
}
//ClsTrialVersion 是 ClsVirusScanner 的子类

```

动态/延迟绑定导致动态方法调度：

```

//调用类
public class ClsRunTheApplication {

public static void main(String[] args) {

    final String VIRUS_SCANNER_VERSION = "TRIAL_VERSION";

    //父类引用为空
    ClsVirusScanner objVS=null;

    //Java SE 7 支持的字符串案例
    switch (VIRUS_SCANNER_VERSION){
        case "FREE_VERSION":

            //父类引用子类对象 3
            //ClsFreeVersion 是 ClsVirusScanner 的子类
            objVS = new ClsFreeVersion(); //动态或运行时绑定
            break;
        case "PAID_VERSION":

            //父类引用子类对象 1
            //ClsPaidVersion 是 ClsVirusScanner 的子类
            objVS = new ClsPaidVersion(); //动态或运行时绑定
            break;
        case "TRIAL_VERSION":

            //父类引用子类对象 2
            objVS = new ClsTrialVersion(); //动态或运行时绑定
            break;
    }

    //方法 fnStartScan() 是 ClsTrialVersion() 的版本
    objVS.fnStartScan();
}

```

```

super.fnInternetSecutiry();
super.fnRealTimeScan();
super.fnVirusMalwareScan();
}
//ClsPaidVersion IS-A ClsVirusScanner
//Child Class 2

class ClsTrialVersion extends ClsVirusScanner{
@Override
public void fnStartScan() {
    super.fnInternetSecutiry();
    super.fnVirusMalwareScan();
}
//ClsTrialVersion IS-A ClsVirusScanner

//Child Class 3
class ClsFreeVersion extends ClsVirusScanner{
@Override
public void fnStartScan() {
    super.fnVirusMalwareScan();
}
//ClsTrialVersion IS-A ClsVirusScanner

```

Dynamic/Late Binding leads to Dynamic method dispatch :

```

//Calling Class
public class ClsRunTheApplication {

public static void main(String[] args) {

    final String VIRUS_SCANNER_VERSION = "TRIAL_VERSION";

    //Parent Refers Null
    ClsVirusScanner objVS=null;

    //String Cases Supported from Java SE 7
    switch (VIRUS_SCANNER_VERSION){
        case "FREE_VERSION":

            //Parent Refers Child Object 3
            //ClsFreeVersion IS-A ClsVirusScanner
            objVS = new ClsFreeVersion(); //Dynamic or Runtime Binding
            break;
        case "PAID_VERSION":

            //Parent Refers Child Object 1
            //ClsPaidVersion IS-A ClsVirusScanner
            objVS = new ClsPaidVersion(); //Dynamic or Runtime Binding
            break;
        case "TRIAL_VERSION":

            //Parent Refers Child Object 2
            objVS = new ClsTrialVersion(); //Dynamic or Runtime Binding
            break;
    }

    //Method fnStartScan() is the Version of ClsTrialVersion()
    objVS.fnStartScan();
}

```

}

结果：

扫描互联网安全
检测病毒&恶意软件

向上转型：

```
objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()
```

}

Result :

Scan **for** Internet Security
Detect Virus & Malware

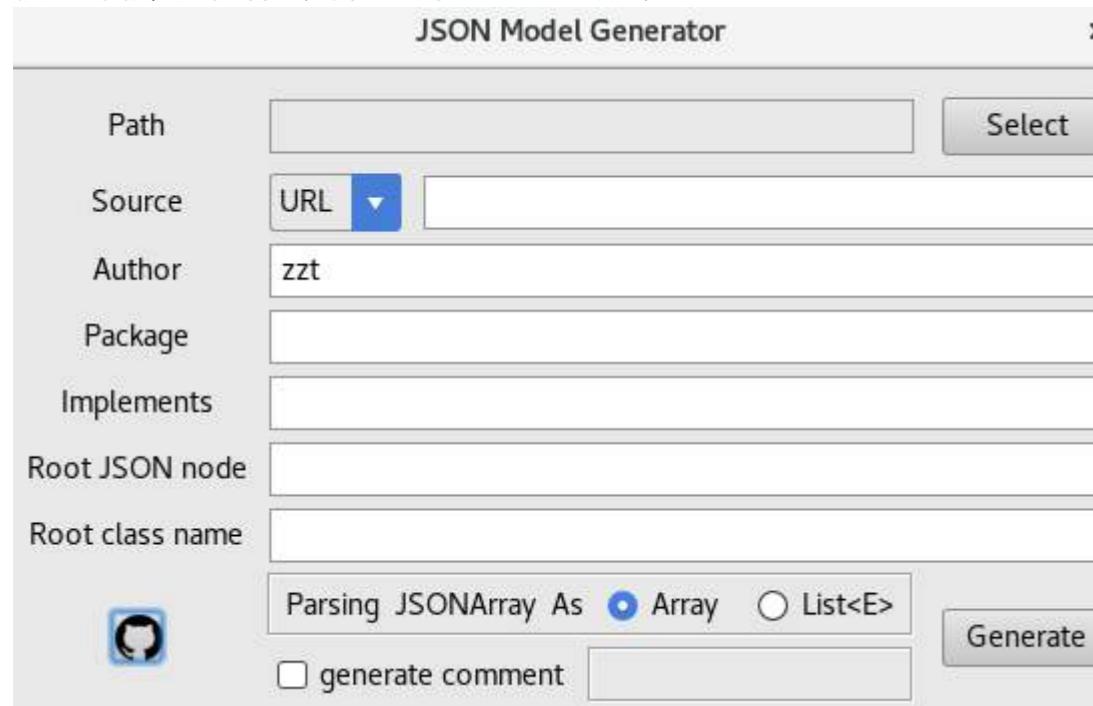
Upcasting :

```
objVS = new ClsFreeVersion();
objVS = new ClsPaidVersion();
objVS = new ClsTrialVersion()
```

第154章：生成Java代码

第154.1节：从JSON生成POJO

- 通过在IntelliJ设置中搜索，安装IntelliJ的JSON模型生成器插件。
- 从“工具”启动插件
- 输入UI字段，如下所示（“路径”、“源”、“包”为必填项）：

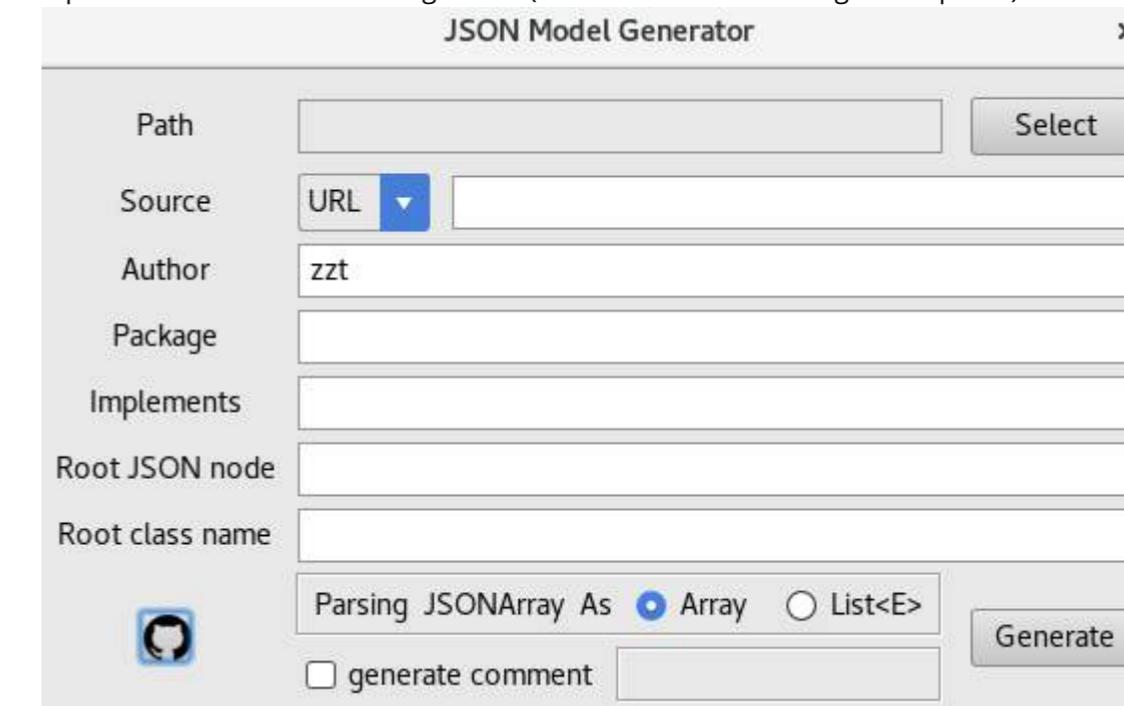


- 点击“生成”按钮，完成操作。

Chapter 154: Generating Java Code

Section 154.1: Generate POJO From JSON

- Install [JSON Model Generator plugin](#) of IntelliJ by searching in IntelliJ setting.
- Start the plugin from 'Tools'
- Input the field of UI as following shows ('Path', 'Source', 'Package' is required):



- Click 'Generate' button and your are done.

第155章：JShell

JShell是JDK 9中新增的Java交互式REPL。它允许开发者即时计算表达式、测试类并尝试Java语言。JDK 9的早期访问版本可从以下地址获取：<http://jdk.java.net/9/>

第155.1节：编辑代码片段

JShell使用的基本代码单元是代码片段（snippet），或称为源条目（source entry）。每次声明局部变量或定义局部方法或类时，都会创建一个代码片段，其名称即为变量/方法/类的标识符。你可以随时使用/edit命令编辑已创建的代码片段。例如，假设我创建了一个名为Foo的类，里面有一个方法bar：

```
jshell> class Foo {  
...> void bar() {  
...> }  
...> }
```

现在，我想填写方法体。与其重写整个类，不如直接编辑它：

```
jshell> /edit Foo
```

默认情况下，会弹出一个带有最基本功能的Swing编辑器。不过你可以更改JShell使用的编辑器：

```
jshell> /设置编辑器 emacs  
jshell> /设置编辑器 vi  
jshell> /设置编辑器 nano  
jshell> /设置编辑器 -默认
```

请注意，如果代码片段的新版本包含任何语法错误，可能无法保存。同样，只有当原始声明/定义在语法上正确时，才会创建代码片段；以下情况无效：

```
jshell> String st = String 3  
//错误省略  
jshell> /edit st  
| 没有这样的代码片段: st
```

然而，尽管存在某些编译时错误（例如类型不匹配），代码片段仍可能被编译并因此可编辑——以下情况有效：

```
jshell> int i = "hello"  
//错误省略  
jshell> /edit i
```

最后，可以使用/drop命令删除代码片段：

```
jshell> int i = 13  
jshell> /drop i  
jshell> System.out.println(i)  
| 错误:  
| 找不到符号  
|   符号: 变量 i  
| System.out.println(i)  
|
```

Chapter 155: JShell

JShell is an interactive REPL for Java added in JDK 9. It allows developers to instantly evaluate expressions, test classes, and experiment with the Java language. Early access for jdk 9 can be obtained from: <http://jdk.java.net/9/>

Section 155.1: Editing Snippets

The basic unit of code used by JShell is the **snippet**, or **source entry**. Every time you declare a local variable or define a local method or class, you create a snippet whose name is the identifier of the variable/method/class. At any time, you can edit a snippet you have created with the /edit command. For example, let's say I have created the class Foo with a single, method, bar:

```
jshell> class Foo {  
...> void bar() {  
...> }  
...> }
```

Now, I want to fill in the body of my method. Rather than rewrite the entire class, I can edit it:

```
jshell> /edit Foo
```

By default, a swing editor will pop up with the most basic features possible. However you can change the editor that JShell uses:

```
jshell> /set editor emacs  
jshell> /set editor vi  
jshell> /set editor nano  
jshell> /set editor -default
```

Note that if **the new version of the snippet contains any syntax errors, it may not be saved**. Likewise, a snippet is only created if the original declaration/definition is syntactically correct; the following does not work:

```
jshell> String st = String 3  
//error omitted  
jshell> /edit st  
| No such snippet: st
```

However, snippets may be compiled and hence editable despite certain compile-time errors, such as mismatched types—the following works:

```
jshell> int i = "hello"  
//error omitted  
jshell> /edit i
```

Finally, snippets may be deleted using the /drop command:

```
jshell> int i = 13  
jshell> /drop i  
jshell> System.out.println(i)  
| Error:  
| cannot find symbol  
|   symbol: variable i  
| System.out.println(i)  
|
```

要删除所有代码片段，从而重置JVM的状态，请使用\reset：

```
jshell> int i = 2
jshell> String s = "hi"
jshell> /reset
| 正在重置状态。
jshell> i
| 错误：
| 找不到符号
| 符号： 变量 i
| i
| ^
jshell> s
| 错误：
| 找不到符号
| 符号： 变量 s
| s
| ^
```

第155.2节：进入和退出JShell

启动 JShell

在尝试启动 JShell 之前，请确保您的JAVA_HOME环境变量指向 JDK 9 安装目录。要启动 JShell，请运行以下命令：

```
$ jshell
```

如果一切顺利，您应该会看到一个jshell>提示符。

退出 JShell

要退出 JShell，请在 JShell 提示符下运行以下命令：

```
jshell> /exit
```

第 155.3 节：表达式

在 JShell 中，您可以计算 Java 表达式，带或不带分号。这些表达式可以是基本的表达式和语句，也可以是更复杂的：

```
jshell> 4+2
jshell> System.out.printf("我今年 %d 岁。", 421)
```

循环和条件语句也可以：

```
jshell> for (int i = 0; i<3; i++) {
...> System.out.println(i);
...> }
```

需要注意的是，代码块内的表达式必须以分号结尾！

To delete all snippets, thereby resetting the state of the JVM, use \reset:

```
jshell> int i = 2
jshell> String s = "hi"
jshell> /reset
| Resetting state.
jshell> i
| Error:
| cannot find symbol
|   symbol:   variable i
| i
| ^
jshell> s
| Error:
| cannot find symbol
|   symbol:   variable s
| s
| ^
```

Section 155.2: Entering and Exiting JShell

Starting JShell

Before trying to start JShell, make sure your JAVA_HOME environment variable points to a JDK 9 installation. To start JShell, run the following command:

```
$ jshell
```

If all goes well, you should see a jshell> prompt.

Exiting JShell

To exit JShell, run the following command from the JShell prompt:

```
jshell> /exit
```

Section 155.3: Expressions

Within JShell, you can evaluate Java expressions, with or without semicolons. These can range from basic expressions and statements to more complex ones:

```
jshell> 4+2
jshell> System.out.printf("I am %d years old.\n", 421)
```

Loops and conditionals are fine, too:

```
jshell> for (int i = 0; i<3; i++) {
...> System.out.println(i);
...> }
```

It is important to note that **expressions within blocks must have semicolons!**

第155.4节：方法和类

你可以在JShell中定义方法和类：

```
jshell> void speak() {  
...> System.out.println("hello");  
...> }  
  
jshell> class MyClass {  
...> void doNothing() {}  
...> }
```

不需要访问修饰符。与其他代码块一样，方法体内也需要分号。请记住，与变量一样，方法和类也可以重新定义。要查看方法或类的列表，分别在JShell提示符下输入 /methods 或 /types。

第155.5节：变量

您可以在 JShell 中声明局部变量：

```
jshell> String s = "hi"  
jshell> int i = s.length
```

请记住，变量可以用不同的类型重新声明；这在 JShell 中是完全有效的：

```
jshell> String var = "hi"  
jshell> int var = 3
```

要查看变量列表，请在 JShell 提示符下输入 /vars。

Section 155.4: Methods and Classes

You can define methods and classes within JShell:

```
jshell> void speak() {  
...> System.out.println("hello");  
...> }  
  
jshell> class MyClass {  
...> void doNothing() {}  
...> }
```

No access modifiers are necessary. As with other blocks, semicolons are required inside of method bodies. Keep in mind that, as with variables, it is possible to redefine methods and classes. To see a list of methods or classes, enter /methods or /types at the JShell prompt, respectively.

Section 155.5: Variables

You can declare local variables within JShell:

```
jshell> String s = "hi"  
jshell> int i = s.length
```

Keep in mind that variables can be redeclared with different types; this is perfectly valid in JShell:

```
jshell> String var = "hi"  
jshell> int var = 3
```

To see a list of variables, enter /vars at the JShell prompt.

第156章：堆栈遍历 API

在 Java 9 之前，访问线程堆栈帧仅限于内部类 `sun.reflect.Reflection`。

具体来说是方法 `sun.reflect.Reflection::getCallerClass`。一些库依赖于此方法，但该方法已被弃用。

JDK 9 现在通过 `java.lang.StackWalker` 类提供了一个替代的标准 API，设计上更高效，允许对堆栈帧进行惰性访问。一些应用程序可能使用此 API 来遍历执行堆栈并对类进行过滤。

第156.1节：打印当前线程的所有堆栈帧

以下代码打印当前线程的所有堆栈帧：

```
1 包 test;
2
3 导入 java.lang.StackWalker.StackFrame;
4 导入 java.lang.reflect.InvocationTargetException;
5 导入 java.lang.reflect.Method;
6 导入 java.util.List;
7 导入 java.util.stream.Collectors;
8
9 公共类 StackWalker示例 {
10
11     public static void main(String[] args) throws NoSuchMethodException, SecurityException,
12         IllegalAccessException, IllegalArgumentException, InvocationTargetException {
13         Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
14         fooMethod.invoke(null, (Object[]) null);
15     }
16
17     class FooHelper {
18         protected static void foo() {
19             BarHelper.bar();
20         }
21     }
22
23     class BarHelper {
24         protected static void bar() {
25             List<StackFrame> stack = StackWalker.getInstance()
26                 .walk((s) -> s.collect(Collectors.toList()));
27             for(StackFrame frame : stack) {
28                 System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
29                     frame.getMethodName());
30             }
31         }
32     }
33 }
```

输出：

```
test.BarHelper 26 bar
test.FooHelper 19 foo
test.StackWalkerExample 13 main
```

Chapter 156: Stack-Walking API

Prior to Java 9, access to the thread stack frames was limited to an internal class `sun.reflect.Reflection`. Specifically the method `sun.reflect.Reflection::getCallerClass`. Some libraries relies on this method which is deprecated.

An alternative standard API is now provided in JDK 9 via the `java.lang.StackWalker` class, and is designed to be efficient by allowing lazy access to the stack frames. Some applications may use this API to traverse the execution stack and filter on classes.

Section 156.1: Print all stack frames of the current thread

The following prints all stack frames of the current thread:

```
1 package test;
2
3 import java.lang.StackWalker.StackFrame;
4 import java.lang.reflect.InvocationTargetException;
5 import java.lang.reflect.Method;
6 import java.util.List;
7 import java.util.stream.Collectors;
8
9 public class StackWalkerExample {
10
11     public static void main(String[] args) throws NoSuchMethodException, SecurityException,
12         IllegalAccessException, IllegalArgumentException, InvocationTargetException {
13         Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);
14         fooMethod.invoke(null, (Object[]) null);
15     }
16
17     class FooHelper {
18         protected static void foo() {
19             BarHelper.bar();
20         }
21     }
22
23     class BarHelper {
24         protected static void bar() {
25             List<StackFrame> stack = StackWalker.getInstance()
26                 .walk((s) -> s.collect(Collectors.toList()));
27             for(StackFrame frame : stack) {
28                 System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
29                     frame.getMethodName());
29             }
30         }
31     }
32 }
```

Output:

```
test.BarHelper 26 bar
test.FooHelper 19 foo
test.StackWalkerExample 13 main
```

第156.2节：打印当前调用者类

下面打印当前调用者类。请注意，在这种情况下，StackWalker需要使用选项RETAIN_CLASS_REFERENCE创建，以便在StackFrame对象中保留Class实例。否则会发生异常。

```
public class StackWalkerExample {  
  
    public static void main(String[] args) {  
        FooHelper.foo();  
    }  
}  
  
class FooHelper {  
    protected static void foo() {  
        BarHelper.bar();  
    }  
}  
  
class BarHelper {  
    protected static void bar() {  
  
        System.out.println(StackWalker.getInstance(Option.RETAIN_CLASS_REFERENCE).getCallerClass());  
    }  
}
```

输出：

```
class 测试.FooHelper
```

第156.3节：显示反射和其他隐藏帧

还有其他几个选项允许堆栈跟踪包含实现和/或反射帧。这对于调试可能很有用。例如，我们可以在创建StackWalker实例时添加SHOW_REFLECT_FRAMES选项，这样反射方法的帧也会被打印出来：

```
包 test;  
  
导入 java.lang.StackWalker.Option;  
导入 java.lang.StackWalker.StackFrame;  
导入 java.lang.reflect.InvocationTargetException;  
导入 java.lang.reflect.Method;  
导入 java.util.List;  
导入 java.util.stream.Collectors;  
  
公共类 StackWalkerExample {  
  
    公共静态 void main(String[] args) 抛出 NoSuchMethodException, SecurityException,  
    IllegalAccessException, IllegalArgumentException, InvocationTargetException {  
        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);  
        fooMethod.invoke(null, (Object[]) null);  
    }  
}  
  
class FooHelper {  
    protected static void foo() {
```

Section 156.2: Print current caller class

The following prints the current caller class. Note that in this case, the [StackWalker](#) needs to be created with the option [RETAIN_CLASS_REFERENCE](#), so that **Class** instances are retained in the [StackFrame](#) objects. Otherwise an exception would occur.

```
public class StackWalkerExample {  
  
    public static void main(String[] args) {  
        FooHelper.foo();  
    }  
}  
  
class FooHelper {  
    protected static void foo() {  
        BarHelper.bar();  
    }  
}  
  
class BarHelper {  
    protected static void bar() {  
  
        System.out.println(StackWalker.getInstance(Option.RETAIN_CLASS_REFERENCE).getCallerClass());  
    }  
}
```

Output:

```
class test.FooHelper
```

Section 156.3: Showing reflection and other hidden frames

A couple of other options allow stack traces to include implementation and/or reflection frames. This may be useful for debugging purposes. For instance, we can add the [SHOW_REFLECT_FRAMES](#) option to the [StackWalker](#) instance upon creation, so that the frames for the reflective methods are printed as well:

```
package test;  
  
import java.lang.StackWalker.Option;  
import java.lang.StackWalker.StackFrame;  
import java.lang.reflect.InvocationTargetException;  
import java.lang.reflect.Method;  
import java.util.List;  
import java.util.stream.Collectors;  
  
public class StackWalkerExample {  
  
    public static void main(String[] args) throws NoSuchMethodException, SecurityException,  
    IllegalAccessException, IllegalArgumentException, InvocationTargetException {  
        Method fooMethod = FooHelper.class.getDeclaredMethod("foo", (Class<?>[])null);  
        fooMethod.invoke(null, (Object[]) null);  
    }  
}  
  
class FooHelper {  
    protected static void foo() {
```

```

BarHelper.bar();
}

class BarHelper {
    protected static void bar() {
        // 显示反射方法
        List<StackFrame> stack = StackWalker.getInstance(Option.SHOW_REFLECT_FRAMES)
            .walk((s) -> s.collect(Collectors.toList()));
        for(StackFrame frame : stack) {
            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
                frame.getMethodName());
        }
    }
}

```

输出：

```

test.BarHelper 27 bar
test.FooHelper 20 foo
jdk.internal.reflect.NativeMethodAccessorImpl -2 invoke0
jdk.internal.reflect.NativeMethodAccessorImpl 62 invoke
jdk.internal.reflect.DelegatingMethodAccessorImpl 43 invoke
java.lang.reflect.Method 563 invoke
test.StackWalkerExample 14 main

```

请注意，某些反射方法的行号可能不可用，因此`StackFrame.getLineNumber()`可能返回负值。

```

BarHelper.bar();
}

class BarHelper {
    protected static void bar() {
        // show reflection methods
        List<StackFrame> stack = StackWalker.getInstance(Option.SHOW_REFLECT_FRAMES)
            .walk((s) -> s.collect(Collectors.toList()));
        for(StackFrame frame : stack) {
            System.out.println(frame.getClassName() + " " + frame.getLineNumber() + " " +
                frame.getMethodName());
        }
    }
}

```

Output:

```

test.BarHelper 27 bar
test.FooHelper 20 foo
jdk.internal.reflect.NativeMethodAccessorImpl -2 invoke0
jdk.internal.reflect.NativeMethodAccessorImpl 62 invoke
jdk.internal.reflect.DelegatingMethodAccessorImpl 43 invoke
java.lang.reflect.Method 563 invoke
test.StackWalkerExample 14 main

```

Note that line numbers for some reflection methods may not be available so `StackFrame.getLineNumber()` may return negative values.

第157章：套接字

套接字是网络上两个程序之间双向通信链路的一个端点。

第157.1节：从套接字读取

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream())));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in)))
) {
    //使用套接字
}
```

Chapter 157: Sockets

A socket is one end-point of a two-way communication link between two programs running on the network.

Section 157.1: Read from socket

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out =
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream())));
    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in)))
) {
    //Use the socket
}
```

第158章：Java套接字

套接字是一种低级网络接口，有助于在两个程序之间建立连接，主要是客户端。这些程序可能运行在同一台机器上，也可能不在同一台机器上。

套接字编程是最广泛使用的网络概念之一。

第158.1节：一个简单的TCP回显服务器

我们的TCP回显服务器将是一个独立的线程。它很简单，因为这是一个起点。它只会将你发送的内容以大写形式回显回来。

```
public class CAPECHOServer extends Thread{

    // 该类实现服务器套接字。服务器套接字仅在本地防火墙允许时
    // 等待网络上的请求到来
    ServerSocket serverSocket;

    public CAPECHOServer(int port, int timeout){
        try {
            // 在指定端口创建一个新的服务器。
            serverSocket = new ServerSocket(port);
            // SoTimeout 基本上是套接字超时。
            // timeout 是套接字超时的时间，单位为毫秒
            serverSocket.setSoTimeout(timeout);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run(){
        try {
            // 我们希望服务器持续接受连接
            while(!Thread.interrupted()){
                }
            // 完成后关闭服务器。
            serverSocket.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

现在开始接受连接。让我们更新 run 方法。

```
@Override
public void run(){
    while(!Thread.interrupted()){
        try {
            // 记录端口号和机器 IP
            Logger.getLogger(this.getClass().getName()).log(Level.INFO, "监听客户端于
{0} 在 {1}", new Object[]{serverSocket.getLocalPort(),
InetAddress.getLocalHost().getHostAddress()});
            Socket client = serverSocket.accept(); // 接受客户端连接
            // 现在获取 DataInputStream 和 DataOutputStream
        }
    }
}
```

Chapter 158: Java Sockets

Sockets are a low-level network interface that helps in creating a connection between two programs mainly clients which may or may not be running on the same machine.

Socket Programming is one of the most widely used networking concepts.

Section 158.1: A simple TCP echo back server

Our TCP echo back server will be a separate thread. It's simple as it's a start. It will just echo back whatever you send it but in capitalised form.

```
public class CAPECHOServer extends Thread{

    // This class implements server sockets. A server socket waits for requests to come
    // in over the network only when it is allowed through the local firewall
    ServerSocket serverSocket;

    public CAPECHOServer(int port, int timeout){
        try {
            // Create a new Server on specified port.
            serverSocket = new ServerSocket(port);
            // SoTimeout is basically the socket timeout.
            // timeout is the time until socket timeout in milliseconds
            serverSocket.setSoTimeout(timeout);
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run(){
        try {
            // We want the server to continuously accept connections
            while(!Thread.interrupted()){
                }
            // Close the server once done.
            serverSocket.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOServer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Now to accept connections. Let's update the run method.

```
@Override
public void run(){
    while(!Thread.interrupted()){
        try {
            // Log with the port number and machine ip
            Logger.getLogger(this.getClass().getName()).log(Level.INFO, "Listening for Clients at
{0} on {1}", new Object[]{serverSocket.getLocalPort(),
InetAddress.getLocalHost().getHostAddress()});
            Socket client = serverSocket.accept(); // Accept client connection
            // Now get DataInputStream and DataOutputStream
        }
    }
}
```

```

DataInputStream istream = new DataInputStream(client.getInputStream()); // 来自客户端的输入流
DataOutputStream ostream = new DataOutputStream(client.getOutputStream());
// 重要说明
/*
服务器的输入是客户端的输出
客户端的输入是服务器的输出
*/
// 发送欢迎消息
ostream.writeUTF("Welcome!");

// 关闭连接
istream.close();
ostream.close();
client.close();
} catch (IOException ex) {
Logger.getLogger(CAPECHOserver.class.getName()).log(Level.SEVERE, null, ex);
}
}

// 完成后关闭服务器

try {
serverSocket.close();
} catch (IOException ex) {
Logger.getLogger(CAPECHOserver.class.getName()).log(Level.SEVERE, null, ex);
}
}

```

现在如果你能打开telnet并尝试连接，你会看到一条欢迎消息。

你必须使用你指定的端口和IP地址进行连接。

你应该看到类似这样的结果：

欢迎！

与主机的连接已断开。

连接断开是因为我们终止了它。有时我们需要编写自己的TCP客户端。
在这种情况下，我们需要一个客户端来请求用户输入并通过网络发送，接收大写的输入。

如果服务器先发送数据，那么客户端必须先读取数据。

```

public class CAPECHOClient extends Thread{

Socket server;
Scanner key; // 用于输入的扫描器

public CAPECHOClient(String ip, int port){
    try {
        server = new Socket(ip, port);
        key = new Scanner(System.in);
    } catch (IOException ex) {
        Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
    }
}

@Override

```

```

DataInputStream istream = new DataInputStream(client.getInputStream()); // From
client's input stream
DataOutputStream ostream = new DataOutputStream(client.getOutputStream());
// Important Note
/*
The server's input is the client's output
The client's input is the server's output
*/
// Send a welcome message
ostream.writeUTF("Welcome!");

// Close the connection
istream.close();
ostream.close();
client.close();
} catch (IOException ex) {
Logger.getLogger(CAPECHOserver.class.getName()).log(Level.SEVERE, null, ex);
}

// Close the server once done

try {
    serverSocket.close();
} catch (IOException ex) {
    Logger.getLogger(CAPECHOserver.class.getName()).log(Level.SEVERE, null, ex);
}
}

```

Now if you can open telnet and try connecting You'll see a Welcome message.

You must connect with the port you specified and IP Adress.

You should see a result similar to this:

Welcome!

Connection to host lost.

Well, the connection was lost because we terminated it. Sometimes we would have to program our own TCP client.
In this case, we need a client to request input from the user and send it across the network, receive the capitalised input.

If the server sends data first, then the client must read the data first.

```

public class CAPECHOClient extends Thread{

Socket server;
Scanner key; // Scanner for input

public CAPECHOClient(String ip, int port){
    try {
        server = new Socket(ip, port);
        key = new Scanner(System.in);
    } catch (IOException ex) {
        Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
    }
}

@Override

```

```

public void run(){
    DataInputStream istream = null;
    DataOutputStream ostream = null;
    try {
        istream = new DataInputStream(server.getInputStream()); // 熟悉的代码行
        ostream = new DataOutputStream(server.getOutputStream());
        System.out.println(istream.readUTF()); // 打印服务器发送的内容
        System.out.print(">");
        String tosend = key.nextLine();
        ostream.writeUTF(tosend); // 发送用户输入的内容到服务器
        System.out.println(istream.readUTF()); // 最后读取服务器发送的内容后退出。
    } catch (IOException ex) {
        Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
    } finally {
        try {
            istream.close();
            ostream.close();
            server.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

现在更新服务器

```

ostream.writeUTF("Welcome!");

String inString = istream.readUTF(); // 读取用户发送的内容
String outString = inString.toUpperCase(); // 转换为大写
ostream.writeUTF(outString);

// 关闭连接
istream.close();

```

现在运行服务器和客户端，你应该会看到类似如下的输出

```

Welcome!
>

```

```

public void run(){
    DataInputStream istream = null;
    DataOutputStream ostream = null;
    try {
        istream = new DataInputStream(server.getInputStream()); // Familiar lines
        ostream = new DataOutputStream(server.getOutputStream());
        System.out.println(istream.readUTF()); // Print what the server sends
        System.out.print(">");
        String tosend = key.nextLine();
        ostream.writeUTF(tosend); // Send whatever the user typed to the server
        System.out.println(istream.readUTF()); // Finally read what the server sends before exiting.
    } catch (IOException ex) {
        Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
    } finally {
        try {
            istream.close();
            ostream.close();
            server.close();
        } catch (IOException ex) {
            Logger.getLogger(CAPECHOClient.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}

```

Now update the server

```

ostream.writeUTF("Welcome!");

String inString = istream.readUTF(); // Read what the user sent
String outString = inString.toUpperCase(); // Change it to caps
ostream.writeUTF(outString);

// Close the connection
istream.close();

```

And now run the server and client, You should have an output similar to this

```

Welcome!
>

```

第159章：FTP（文件传输协议）

参数	详情
主机	FTP服务器的主机名或IP地址
端口	FTP服务器端口
用户名	FTP服务器用户名
密码	FTP服务器密码

第159.1节：连接并登录FTP服务器

要开始使用Java进行FTP，您需要创建一个新的FTPClient，然后使用该客户端连接并登录服务器
.connect(字符串 服务器, 整数 端口) 和 .login(字符串 用户名, 字符串 密码)。

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;
//导入本项目所需的所有资源。

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // 设置这些以匹配您的FTP服务器 //
        String server = "www.server.com"; //服务器可以是主机名或IP地址。
        int port = 21;
        String user = "Username";
        String pass = "Password";

        FTPClient ftp = new FTPClient();
        ftp.connect(server, port);
        ftp.login(user, pass);
    }
}
```

现在我们已经完成了基础部分。但如果连接服务器时出现错误怎么办？我们需要知道何时出现问题并获取错误信息。
让我们添加一些代码来捕获连接时的错误。

```
try {
    ftp.connect(server, port);
    showServerReply(ftp);
    int replyCode = ftp.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
        System.out.println("操作失败。服务器回复代码: " + replyCode)
        return;
    }
    ftp.login(user, pass);
} catch {
}
```

让我们一步步分解刚才所做的操作。

```
showServerReply(ftp);
```

这指的是我们将在后续步骤中创建的一个函数。

```
int replyCode = ftp.getReplyCode();
```

Chapter 159: FTP (File Transfer Protocol)

Parameters	Details
host	Either the host name or IP address of the FTP server
port	The FTP server port
username	The FTP server username
password	The FTP server password

Section 159.1: Connecting and Logging Into a FTP Server

To start using FTP with Java, you will need to create a new FTPClient and then connect and login to the server using
.connect(String server, int port) and .login(String username, String password).

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;
//Import all the required resource for this project.

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com"; //Server can be either host name or IP address.
        int port = 21;
        String user = "Username";
        String pass = "Password";

        FTPClient ftp = new FTPClient();
        ftp.connect(server, port);
        ftp.login(user, pass);
    }
}
```

Now we have the basics done. But what if we have an error connecting to the server? We'll want to know when something goes wrong and get the error message. Let's add some code to catch errors while connecting.

```
try {
    ftp.connect(server, port);
    showServerReply(ftp);
    int replyCode = ftp.getReplyCode();
    if (!FTPReply.isPositiveCompletion(replyCode)) {
        System.out.println("Operation failed. Server reply code: " + replyCode)
        return;
    }
    ftp.login(user, pass);
} catch {
}
```

Let's break down what we just did, step by step.

```
showServerReply(ftp);
```

This refers to a function we will be making in a later step.

```
int replyCode = ftp.getReplyCode();
```

这会获取服务器的回复/错误代码并将其存储为整数。

```
if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.println("操作失败。服务器回复代码: " + replyCode)
    return;
}
```

这会检查回复代码以判断是否有错误。如果有错误，它将简单地打印“操作失败”。

服务器回复代码：“后跟错误代码。我们还添加了一个try/catch块，下一步将继续完善。接下来，我们还将创建一个检查`ftp.login()`是否有错误的函数。

```
boolean success = ftp.login(user, pass);
showServerReply(ftp);
if (!success) {
    System.out.println("登录服务器失败");
    return;
} else {
    System.out.println("已登录服务器");
}
```

我们也来拆解这个代码块。

```
boolean success = ftp.login(user, pass);
```

这不仅会尝试登录FTP服务器，还会将结果存储为布尔值。

```
showServerReply(ftp);
```

这将检查服务器是否发送了任何消息，但我们首先需要在下一步创建该函数。

```
if (!success) {
    System.out.println("登录服务器失败");
    return;
} else {
    System.out.println("已登录服务器");
}
```

该语句将检查我们是否成功登录；如果成功，将打印“已登录服务器”，否则将打印“登录服务器失败”。这是我们目前的脚本：

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // 请根据您的FTP服务器设置这些参数 //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {

```

This grabs the reply/error code from the server and stores it as an integer.

```
if (!FTPReply.isPositiveCompletion(replyCode)) {
    System.out.println("Operation failed. Server reply code: " + replyCode)
    return;
}
```

This checks the reply code to see if there was an error. If there was an error, it will simply print "Operation failed. Server reply code: " followed by the error code. We also added a try/catch block which we will add to in the next step. Next, let's also create a function that checks `ftp.login()` for errors.

```
boolean success = ftp.login(user, pass);
showServerReply(ftp);
if (!success) {
    System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}
```

Let's break this block down too.

```
boolean success = ftp.login(user, pass);
```

This will not just attempt to login to the FTP server, it will also store the result as a boolean.

```
showServerReply(ftp);
```

This will check if the server sent us any messages, but we will first need to create the function in the next step.

```
if (!success) {
    System.out.println("Failed to log into the server");
    return;
} else {
    System.out.println("LOGGED IN SERVER");
}
```

This statement will check if we logged in successfully; if so, it will print "LOGGED IN SERVER", otherwise it will print "Failed to log into the server". This is our script so far:

```
import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {

```

```

        System.out.println("操作失败。服务器回复代码: " + replyCode);
        return;
    }
    boolean success = ftp.login(user, pass);
    showServerReply(ftp);
    if (!success) {
        System.out.println("登录服务器失败");
        return;
    } else {
        System.out.println("已登录服务器");
    }
} catch {
}
}
}

```

现在接下来让我们完成 Catch 块，以防整个过程中出现任何错误。

```

} catch (IOException ex) {
    System.out.println("哎呀！出了点问题。");
    ex.printStackTrace();
}

```

完成的 catch 块现在将在出现错误时打印“哎呀！出了点问题。”以及堆栈跟踪。
现在我们的最后一步是创建我们已经使用了一段时间的showServerReply()函数。

```

private static void showServerReply(FTPClient ftp) {
    String[] replies = ftp.getReplyStrings();
    if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
        }
    }
}

```

此函数以一个FTPClient作为变量，命名为“ftp”。随后它将服务器的任何回复存储在一个字符串数组中。接着检查是否存储了任何消息。如果有，则将每条消息打印为“SERVER: [reply]”。现在该函数完成，以下是完整脚本：

```

import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        String[] replies = ftp.getReplyStrings();
        if (replies != null && replies.length > 0) {
            for (String aReply : replies) {
                System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // 请根据您的FTP服务器设置这些参数 //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"
    }
}

```

```

        System.out.println("Operation failed. Server reply code: " + replyCode);
        return;
    }
    boolean success = ftp.login(user, pass);
    showServerReply(ftp);
    if (!success) {
        System.out.println("Failed to log into the server");
        return;
    } else {
        System.out.println("LOGGED IN SERVER");
    }
} catch {
}
}

```

Now next let's create complete the Catch block in case we run into any errors with the whole process.

```

} catch (IOException ex) {
    System.out.println("Oops! Something went wrong.");
    ex.printStackTrace();
}

```

The completed catch block will now print "Oops! Something went wrong." and the stacktrace if there is an error.
Now our final step is to create the showServerReply() we have been using for a while now.

```

private static void showServerReply(FTPClient ftp) {
    String[] replies = ftp.getReplyStrings();
    if (replies != null && replies.length > 0) {
        for (String aReply : replies) {
            System.out.println("SERVER: " + aReply);
        }
    }
}

```

This function takes an FTPClient as a variable, and calls it "ftp". After that it stores any server replies from the server in a string array. Next it checks if any messages were stored. If there is any, it prints each of them as "SERVER: [reply]". Now that we have that function done, this is the completed script:

```

import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        String[] replies = ftp.getReplyStrings();
        if (replies != null && replies.length > 0) {
            for (String aReply : replies) {
                System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"
    }
}

```

```

FTPClient ftp = new FTPClient
    try {
        ftp.connect(server, port)
        showServerReply(ftp);
        int replyCode = ftpClient.getReplyCode();
        if (!FTPReply.isPositiveCompletion(replyCode)) {
            System.out.println("操作失败。服务器回复代码: " + replyCode);
            return;
        }
        boolean success = ftp.login(user, pass);
        showServerReply(ftp);
        if (!success) {
            System.out.println("登录服务器失败");
            return;
        } else {
            System.out.println("已登录服务器");
        }
    } catch (IOException ex) {
        System.out.println("哎呀！出了点问题。");
        ex.printStackTrace();
    }
}

```

我们首先需要创建一个新的FTPClient，尝试连接服务器并使用`.connect(String 服务器, int 端口)` 和 `.login(String 用户名, String 密码)`。重要的是要使用 try/catch 块来连接和登录，以防我们的代码无法连接到服务器。我们还需要创建一个函数，用于检查并显示在尝试连接和登录时可能从服务器接收到的任何消息。我们将调用此函数为 "showServerReply(FTPClient ftp)"。

```

import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        if (replies != null && replies.length > 0) {
            for (String aReply : replies) {
                System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // 请根据您的FTP服务器设置这些参数 //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(服务器, 端口)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("操作失败。服务器回复代码: " + replyCode);
                return;
            }
            boolean success = ftp.login(用户, 密码);
            showServerReply(ftp);
        }
    }
}

```

```

FTPClient ftp = new FTPClient
    try {
        ftp.connect(server, port)
        showServerReply(ftp);
        int replyCode = ftpClient.getReplyCode();
        if (!FTPReply.isPositiveCompletion(replyCode)) {
            System.out.println("Operation failed. Server reply code: " + replyCode);
            return;
        }
        boolean success = ftp.login(user, pass);
        showServerReply(ftp);
        if (!success) {
            System.out.println("Failed to log into the server");
            return;
        } else {
            System.out.println("LOGGED IN SERVER");
        }
    } catch (IOException ex) {
        System.out.println("Oops! Something went wrong.");
        ex.printStackTrace();
    }
}

```

We first need to create a new FTPClient and try connecting to the server it and logging into it using `.connect(String server, int port)` and `.login(String username, String password)`. It is important to connect and login using a try/catch block in case our code fails to connect with the server. We will also need to create a function that checks and displays any messages we may receive from the server as we try connecting and logging in. We will call this function "showServerReply(FTPClient ftp)".

```

import java.io.IOException;
import org.apache.commons.net.ftp.FTPClient;
import org.apache.commons.net.ftp.FTPReply;

public class FTPConnectAndLogin {
    private static void showServerReply(FTPClient ftp) {
        if (replies != null && replies.length > 0) {
            for (String aReply : replies) {
                System.out.println("SERVER: " + aReply);
            }
        }
    }

    public static void main(String[] args) {
        // SET THESE TO MATCH YOUR FTP SERVER //
        String server = "www.server.com";
        int port = 21;
        String user = "username"
        String pass = "password"

        FTPClient ftp = new FTPClient
        try {
            ftp.connect(server, port)
            showServerReply(ftp);
            int replyCode = ftpClient.getReplyCode();
            if (!FTPReply.isPositiveCompletion(replyCode)) {
                System.out.println("Operation failed. Server reply code: " + replyCode);
                return;
            }
            boolean success = ftp.login(user, pass);
            showServerReply(ftp);
        }
    }
}

```

```
if (!success) {  
    System.out.println("登录服务器失败");  
    return;  
} else {  
    System.out.println("已登录服务器");  
}  
} catch (IOException ex) {  
    System.out.println("哎呀！出了点问题。");  
    ex.printStackTrace();  
}  
}
```

之后，你现在应该已经将 FTP 服务器连接到你的 Java 脚本了。

```
if (!success) {  
    System.out.println("Failed to log into the server");  
    return;  
} else {  
    System.out.println("LOGGED IN SERVER");  
}  
} catch (IOException ex) {  
    System.out.println("Oops! Something went wrong.");  
    ex.printStackTrace();  
}  
}
```

After this, you should now have your FTP server connected to your Java script.

第160章：在 Java 中使用其他脚本语言

Java 本身是一种非常强大的语言，但由于 JSR223 (Java 规范请求 223) 引入了脚本引擎，其功能可以进一步扩展

第160.1节：在nashorn的脚本模式下评估JavaScript文件

```
public class JSEngine {  
  
    /*  
     * 注意 Nashorn 仅适用于 Java 8 及以上版本  
     * 你可以使用 ScriptEngineManager.getEngineByName("js") 来使用 rhino ;  
     */  
  
    ScriptEngine engine;  
    ScriptContext context;  
    public Bindings scope;  
  
    // 从其工厂以脚本模式初始化引擎  
    public JSEngine(){  
        engine = new NashornScriptEngineFactory().getScriptEngine("-scripting");  
        // ScriptContext 是一个接口，因此我们需要它的一个实现类  
        context = new SimpleScriptContext();  
        // 创建绑定以暴露变量  
        scope = engine.createBindings();  
    }  
  
    // 清除绑定以移除之前的变量  
    public void newBatch(){  
        scope.clear();  
    }  
  
    public void execute(String file){  
        try {  
            // 获取用于输入的缓冲读取器  
            BufferedReader br = new BufferedReader(new FileReader(file));  
            // 评估代码，输入为缓冲读取器  
            engine.eval(br);  
        } catch (FileNotFoundException ex) {  
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);  
        } catch (ScriptException ex) {  
            // 脚本异常基本上是指脚本中出现错误  
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
  
    public void eval(String code){  
        try {  
            // Engine.eval 基本上将任何字符串视为一行代码并进行评估、执行  
            engine.eval(code);  
        } catch (ScriptException ex) {  
            // 脚本异常基本上是指脚本中出现错误  
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
}
```

Chapter 160: Using Other Scripting Languages in Java

Java in itself is an extremely powerful language, but its power can further be extended Thanks to JSR223 (Java Specification Request 223) introducing a script engine

Section 160.1: Evaluating A JavaScript file in -scripting mode of nashorn

```
public class JSEngine {  
  
    /*  
     * Note Nashorn is only available for Java-8 onwards  
     * You can use rhino from ScriptEngineManager.getEngineByName("js");  
     */  
  
    ScriptEngine engine;  
    ScriptContext context;  
    public Bindings scope;  
  
    // Initialize the Engine from its factory in scripting mode  
    public JSEngine(){  
        engine = new NashornScriptEngineFactory().getScriptEngine("-scripting");  
        // Script context is an interface so we need an implementation of it  
        context = new SimpleScriptContext();  
        // Create bindings to expose variables into  
        scope = engine.createBindings();  
    }  
  
    // Clear the bindings to remove the previous variables  
    public void newBatch(){  
        scope.clear();  
    }  
  
    public void execute(String file){  
        try {  
            // Get a buffered reader for input  
            BufferedReader br = new BufferedReader(new FileReader(file));  
            // Evaluate code, with input as bufferedReader  
            engine.eval(br);  
        } catch (FileNotFoundException ex) {  
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);  
        } catch (ScriptException ex) {  
            // Script Exception is basically when there is an error in script  
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
  
    public void eval(String code){  
        try {  
            // Engine.eval basically treats any string as a line of code and evaluates it, executes  
            engine.eval(code);  
        } catch (ScriptException ex) {  
            // Script Exception is basically when there is an error in script  
            Logger.getLogger(JSEngine.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
}
```

```

// 将绑定应用到上下文并设置引擎的默认上下文
public void startBatch(int SCP){
    context.setBindings(scope, SCP);
    engine.setContext(context);
}

// 我们使用 Invocable 接口来访问脚本中的方法
// Invocable 是一个可选接口, 请检查您的引擎是否实现了它
public Invocable invocable(){
    return (Invocable)engine;
}

}

```

现在是主方法

```

public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // 创建一个新的批处理, 可能不必要
    jse.newBatch();
    // 将变量 x 以值 "hello world" 暴露给脚本
    jse.scope.put("x", "hello world");
    // 应用绑定并启动批处理
    jse.startBatch(ExecutionContext.ENGINE_SCOPE);
    // 执行代码
    jse.eval("print(x);");
}

```

你的输出应类似于此

hello world

如您所见, 已打印出暴露的变量 x。现在进行文件测试。

这里是 test.js

```

print(x);
function test(){
    print("hello test.js:test");
}
test();

```

以及更新后的主方法

```

public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // 创建一个新的批处理, 可能不必要
    jse.newBatch();
    // 将变量 x 以值 "hello world" 暴露给脚本
    jse.scope.put("x", "hello world");
    // 应用绑定并启动批处理
    jse.startBatch(ExecutionContext.ENGINE_SCOPE);
    // 执行代码
    jse.execute("./test.js");
}

```

```

// Apply the bindings to the context and set the engine's default context
public void startBatch(int SCP){
    context.setBindings(scope, SCP);
    engine.setContext(context);
}

// We use the invocable interface to access methods from the script
// Invocable is an optional interface, please check if your engine implements it
public Invocable invocable(){
    return (Invocable)engine;
}

}

```

Now the main method

```

public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unnecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ExecutionContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.eval("print(x);");
}

```

Your output should be similar to this

hello world

As you can see the exposed variable x has been printed. Now testing with a file.

Here we have test.js

```

print(x);
function test(){
    print("hello test.js:test");
}
test();

```

And the updated main method

```

public static void main(String[] args) {
    JSEngine jse = new JSEngine();
    // Create a new batch probably unnecessary
    jse.newBatch();
    // Expose variable x into script with value of hello world
    jse.scope.put("x", "hello world");
    // Apply the bindings and start the batch
    jse.startBatch(ExecutionContext.ENGINE_SCOPE);
    // Evaluate the code
    jse.execute("./test.js");
}

```

假设 test.js 与您的应用程序在同一目录下，您应该会看到类似如下的输出

```
hello world  
hello test.js:test
```

Assuming that test.js is in the same directory as your application You should have output similar to this

```
hello world  
hello test.js:test
```

第161章：C++比较

Java和C++是相似的语言。本主题作为Java和C++工程师的快速参考指南。

第161.1节：静态类成员

静态成员具有类范围，而非对象范围

C++示例

```
// 在头文件中定义
class Singleton {
    public:
        static Singleton *getInstance();

    private:
    Singleton() {}
        static Singleton *instance;
};

// 在.cpp中初始化
Singleton* Singleton::instance = 0;
```

Java 示例

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Chapter 161: C++ Comparison

Java and C++ are similar languages. This topic serves as a quick reference guide for Java and C++ Engineers.

Section 161.1: Static Class Members

Static members have class scope as opposed to object scope

C++ Example

```
// define in header
class Singleton {
    public:
        static Singleton *getInstance();

    private:
    Singleton() {}
        static Singleton *instance;
};

// initialize in .cpp
Singleton* Singleton::instance = 0;
```

Java Example

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

第161.2节：定义在其他结构内的类

定义在另一个类中

C++

嵌套类[ref]（需要对封闭类的引用）

```
class Outer {
    class Inner {
        public:
            Inner(Outer* o) :outer(o) {}

        private:
        Outer* outer;
    };
};
```

Java

[非静态] 嵌套类（又称内部类或成员类）

Section 161.2: Classes Defined within Other Constructs

Defined within Another Class

C++

Nested Class[ref] (needs a reference to enclosing class)

```
class Outer {
    class Inner {
        public:
            Inner(Outer* o) :outer(o) {}

        private:
        Outer* outer;
    };
};
```

Java

[non-static] Nested Class (aka Inner Class or Member Class)

```
class OuterClass {  
    ...  
    class InnerClass {  
        ...  
    }  
}
```

静态定义于另一个类中

C++

静态嵌套类

```
class Outer {  
    class Inner {  
        ...  
    };  
};
```

Java

静态嵌套类（又称静态成员类）[ref] ——

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
}
```

定义在方法内部

(例如事件处理)

C++

局部类[ref] ——

```
void fun() {  
    class Test {  
        /* Test类的成员 */  
    };  
}
```

Java

局部类[ref] ——

```
class Test {  
    void f() {  
        new Thread(new Runnable() {  
            public void run() {  
                doSomethingBackgroundish();  
            }  
        }).start();  
    }  
}
```

```
class OuterClass {  
    ...  
    class InnerClass {  
        ...  
    }  
}
```

Statically Defined within Another Class

C++

Static Nested Class

```
class Outer {  
    class Inner {  
        ...  
    };  
};
```

Java

Static Nested Class (aka Static Member Class)[ref]

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
}
```

Defined within a Method

(e.g. event handling)

C++

Local Class[ref]

```
void fun() {  
    class Test {  
        /* members of Test class */  
    };  
}
```

Java

Local Class[ref]

```
class Test {  
    void f() {  
        new Thread(new Runnable() {  
            public void run() {  
                doSomethingBackgroundish();  
            }  
        }).start();  
    }  
}
```

第161.3节：按值传递与按引用传递

许多人认为Java仅仅是按值传递，但情况比这更复杂。比较以下C++和Java示例，可以看到按值传递（也称复制）和按引用传递（也称别名）的多种形式。

C++示例 (完整代码)

```
// 传递对象的副本
static void passByCopy(PassIt obj) {
    obj.i = 22; // 仅是“局部”修改
}

// 传递指针
static void passByPointer(PassIt* ptr) {
    ptr->i = 33;
ptr = 0; // 如果是'0'，最好用nullptr代替
}

// 传递别名 (也称引用)
static void passByAlias(PassIt& ref) {
    ref.i = 44;
}

// 这是一个老派的做法。
// 查看 std::swap 以获得最佳做法
static void swap(PassIt** pptr1, PassIt** pptr2) {
    PassIt* tmp = *pptr1;
    *pptr1 = *pptr2;
    *pptr2 = tmp;
}
```

Java 示例 (完整代码)

```
// 传递变量的副本
// 注意：在 Java 中只有基本类型是按值传递
public static void passByCopy(int copy) {
    copy = 33; // 仅是“局部”更改
}

// Java 中没有指针的概念
/*
public static void passByPointer(PassIt *ptr) {
    ptr->i = 33;
ptr = 0; // 如果是'0'，最好用 nullptr 代替
}
*/

// 传递别名 (也称引用)
public static void passByAlias(PassIt ref) {
    ref.i = 44;
}

// 传递别名 (即引用) ,
// 但需要进行“手动”的、可能代价较高的复制
public static void swap(PassIt ref1, PassIt ref2) {
    PassIt tmp = new PassIt(ref1);
ref1.copy(ref2);
    ref2.copy(tmp);
}
```

Section 161.3: Pass-by-value & Pass-by-reference

Many argue that Java is ONLY pass-by-value, but it's more nuanced than that. Compare the following C++ and Java examples to see the many flavors of pass-by-value (aka copy) and pass-by-reference (aka alias).

C++ Example (complete code)

```
// passes a COPY of the object
static void passByCopy(PassIt obj) {
    obj.i = 22; // only a "local" change
}

// passes a pointer
static void passByPointer(PassIt* ptr) {
    ptr->i = 33;
    ptr = 0; // better to use nullptr instead if '0'
}

// passes an alias (aka reference)
static void passByAlias(PassIt& ref) {
    ref.i = 44;
}

// This is an old-school way of doing it.
// Check out std::swap for the best way to do this
static void swap(PassIt** pptr1, PassIt** pptr2) {
    PassIt* tmp = *pptr1;
    *pptr1 = *pptr2;
    *pptr2 = tmp;
}
```

Java Example (complete code)

```
// passes a copy of the variable
// NOTE: in java only primitives are pass-by-copy
public static void passByCopy(int copy) {
    copy = 33; // only a "local" change
}

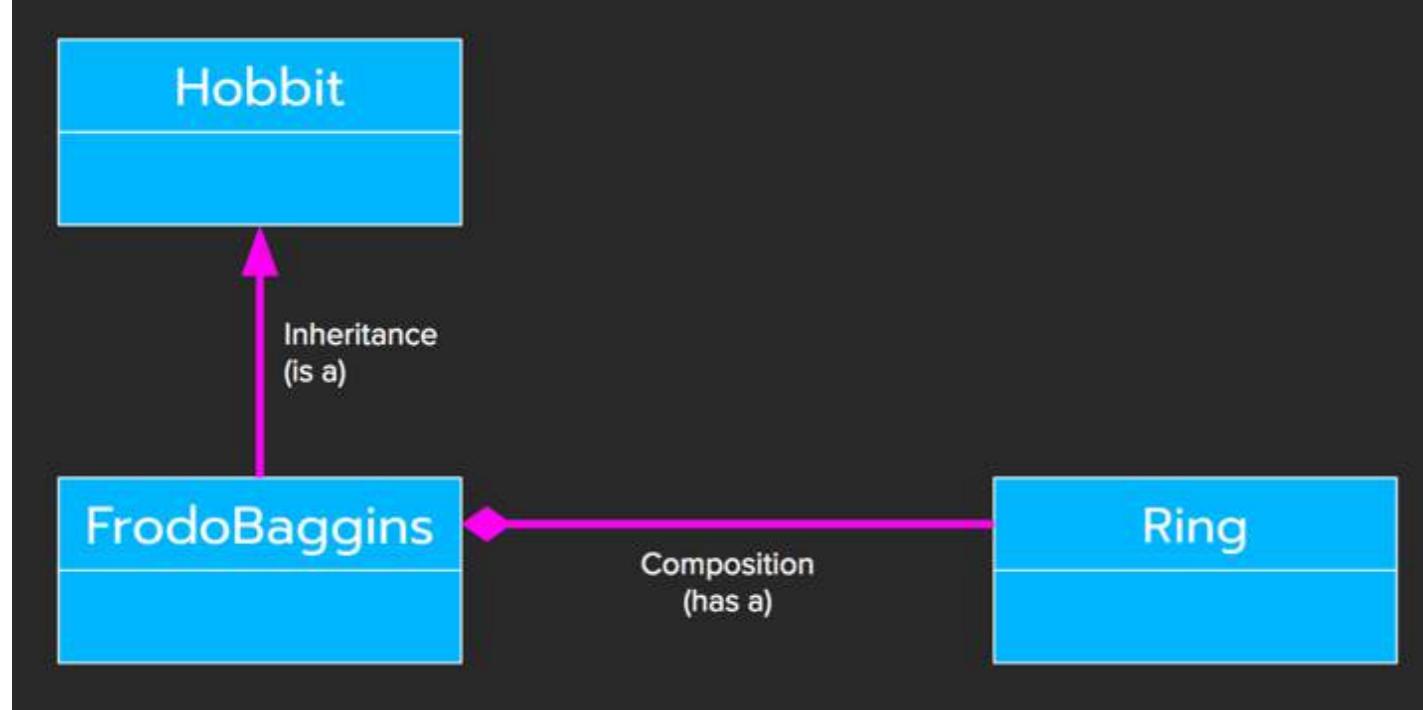
// No such thing as pointers in Java
/*
public static void passByPointer(PassIt *ptr) {
    ptr->i = 33;
    ptr = 0; // better to use nullptr instead if '0'
}
*/

// passes an alias (aka reference)
public static void passByAlias(PassIt ref) {
    ref.i = 44;
}

// passes aliases (aka references),
// but need to do "manual", potentially expensive copies
public static void swap(PassIt ref1, PassIt ref2) {
    PassIt tmp = new PassIt(ref1);
    ref1.copy(ref2);
    ref2.copy(tmp);
}
```

第161.4节：继承与组合

C++ 和 Java 都是面向对象语言，因此以下图示适用于两者。



第161.5节：不当的向下转型

注意使用“向下转型”——向下转型是指从基类向继承层次结构中的子类进行转型（即多态的反向操作）。通常，应使用多态和重写来代替 instanceof 和向下转型。

C++ 示例

```
// 需要显式类型转换  
Child *pChild = (Child *) &parent;
```

Java 示例

```
if(mySubClass instanceof SubClass) {  
    SubClass mySubClass = (SubClass)someBaseClass;  
    mySubClass.nonInheritedMethod();  
}
```

第161.6节：抽象方法与类

抽象方法

声明时无实现

C++

纯虚方法

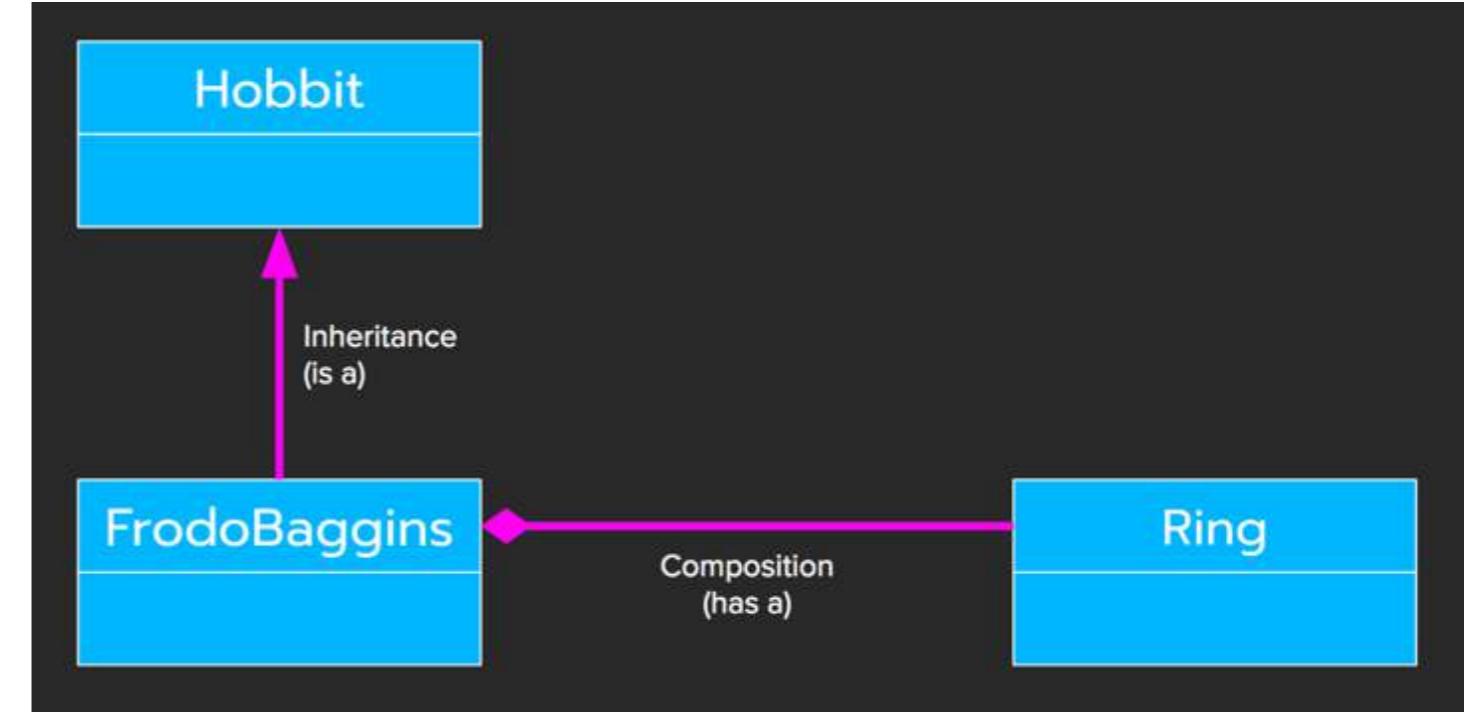
```
virtual void eat(void) = 0;
```

Java

抽象方法

Section 161.4: Inheritance vs Composition

C++ & Java are both object-oriented languages, thus the following diagram applies to both.



Section 161.5: Outcast Downcasting

Beware of using "downcasting" - Downcasting is casting down the inheritance hierarchy from a base class to a subclass (i.e. opposite of polymorphism). In general, use polymorphism & overriding instead of instanceof & downcasting.

C++ Example

```
// explicit type case required  
Child *pChild = (Child *) &parent;
```

Java Example

```
if(mySubClass instanceof SubClass) {  
    SubClass mySubClass = (SubClass)someBaseClass;  
    mySubClass.nonInheritedMethod();  
}
```

Section 161.6: Abstract Methods & Classes

Abstract Method

declared without an implementation

C++

pure virtual method

```
virtual void eat(void) = 0;
```

Java

abstract method

抽象 无返回值 绘制();

抽象类

不能被实例化

C++

不能被实例化；至少有一个纯虚函数

```
类 AB {公有: 虚函数 无返回值 f() = 0;};
```

Java

不能被实例化；可以有非抽象方法

抽象类 图形对象 {}

接口

无实例字段

C++

没有类似于Java的东西

Java

非常类似于抽象类，但1) 支持多重继承；2) 无实例字段

```
interface TestInterface {}
```

abstract void draw();

Abstract Class

cannot be instantiated

C++

cannot be instantiated; has at least 1 pure virtual method

```
class AB {public: virtual void f() = 0;};
```

Java

cannot be instantiated; can have non-abstract methods

```
abstract class GraphicObject {}
```

Interface

no instance fields

C++

nothing comparable to Java

Java

very similar to abstract class, but 1) supports multiple inheritance; 2) no instance fields

```
interface TestInterface {}
```

第162章：音频

第162.1节：播放MIDI文件

可以使用javax.sound.midi包中的多个类来播放MIDI文件。Sequencer执行MIDI文件的播放，其许多方法可用于设置播放控制，如循环次数、速度、轨道静音等。

一般可以通过以下方式播放MIDI数据：

```
import java.io.File;
import java.io.IOException;
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Sequence;
import javax.sound.midi.Sequencer;

public class MidiPlayback {
    public static void main(String[] args) {
        try {
            Sequencer sequencer = MidiSystem.getSequencer(); // 获取默认Sequencer
            if (sequencer == null) {
                System.err.println("不支持Sequencer设备");
                return;
            }
            sequencer.open(); // 打开设备
            // 创建序列，文件必须包含MIDI文件数据。
            Sequence sequence = MidiSystem.getSequence(new File(args[0]));
            sequencer.setSequence(sequence); // 加载到sequencer中
            sequencer.start(); // 开始播放
        } catch (MidiUnavailableException | InvalidMidiDataException | IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

停止播放使用：

```
sequencer.stop(); // 停止播放
```

sequencer可以在播放时将序列中的一个或多个轨道静音，使指定轨道中的乐器不发声。
以下示例将序列中的第一个轨道设置为静音：

```
import javax.sound.midi.Track;
// ...

Track[] track = sequence.getTracks();
sequencer.setTrackMute(track[0]);
```

sequencer可以设置循环次数来重复播放序列。以下设置sequencer播放序列四次和无限次：

```
sequencer.setLoopCount(3);
sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
```

序列器不一定总是从头开始播放序列，也不一定必须播放

Chapter 162: Audio

Section 162.1: Play a MIDI file

MIDI files can be played by using several classes from the `javax.sound.midi` package. A `Sequencer` performs playback of the MIDI file, and many of its methods can be used to set playback controls such as loop count, tempo, track muting, and others.

General playback of MIDI data can be done in this way:

```
import java.io.File;
import java.io.IOException;
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Sequence;
import javax.sound.midi.Sequencer;

public class MidiPlayback {
    public static void main(String[] args) {
        try {
            Sequencer sequencer = MidiSystem.getSequencer(); // Get the default Sequencer
            if (sequencer==null) {
                System.err.println("Sequencer device not supported");
                return;
            }
            sequencer.open(); // Open device
            // Create sequence, the File must contain MIDI file data.
            Sequence sequence = MidiSystem.getSequence(new File(args[0]));
            sequencer.setSequence(sequence); // load it into sequencer
            sequencer.start(); // start the playback
        } catch (MidiUnavailableException | InvalidMidiDataException | IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

To stop the playback use:

```
sequencer.stop(); // Stop the playback
```

A sequencer can be set to mute one or more of the sequence's tracks during playback so none of the instruments in those specified play. The following example sets the first track in the sequence to be muted:

```
import javax.sound.midi.Track;
// ...

Track[] track = sequence.getTracks();
sequencer.setTrackMute(track[0]);
```

A sequencer can play a sequence repeatedly if the loop count is given. The following sets the sequencer to play a sequence four times and indefinitely:

```
sequencer.setLoopCount(3);
sequencer.setLoopCount(Sequencer.LOOP_CONTINUOUSLY);
```

The sequencer does not always have to play the sequence from the beginning, nor does it have to play the

序列直到结束。可以通过指定序列中开始和结束的tick来从任意点开始和结束。
也可以手动指定序列中应该从哪个tick开始播放：

```
sequencer.setLoopStartPoint(512);
sequencer.setLoopEndPoint(32768);
sequencer.setTickPosition(8192);
```

序列器还可以以一定的节奏播放MIDI文件，节奏可以通过指定每分钟节拍数（BPM）或每四分音符微秒数（MPQ）来控制。播放序列的速度因子也可以调整。

```
sequencer.setTempoInBPM(1250f);
sequencer.setTempoInMPQ(4750f);
sequencer.setTempoFactor(1.5f);
```

使用完Sequencer后，记得关闭它

```
sequencer.close();
```

第162.2节：循环播放音频文件

所需导入：

```
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
```

这段代码将创建一个剪辑并在启动后持续播放：

```
Clip clip = AudioSystem.getClip();
clip.open(AudioSystem.getAudioInputStream(new URL(filename)));
clip.start();
clip.loop(Clip.LOOP_CONTINUOUSLY);
```

获取包含所有支持文件类型的数组：

```
AudioFileFormat.Type [] audioFileTypes = AudioSystem.getAudioFileTypes();
```

第162.3节：基本音频输出

Java的Hello Audio! 用于播放本地或互联网存储的音频文件，代码如下。它适用于未压缩的.wav文件，不应用于播放mp3或压缩文件。

```
import java.io.*;
import java.net.URL;
import javax.sound.sampled.*;

public class SoundClipTest {

    // 构造函数
    public SoundClipTest() {
        try {
            // 打开音频输入流。
            File soundFile = new File("/usr/share/sounds/alsa/Front_Center.wav"); //你也可以通过URL获取音频文件
            AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);
            AudioFormat format = audioIn.getFormat();
        }
    }
}
```

sequence until the end. It can start and end at any point by specifying the tick in the sequence to start and end at. It is also possible to specify manually which tick in the sequence the sequencer should play from:

```
sequencer.setLoopStartPoint(512);
sequencer.setLoopEndPoint(32768);
sequencer.setTickPosition(8192);
```

Sequencers can also play a MIDI file at a certain tempo, which can be controlled by specifying the tempo in beats per minute (BPM) or microseconds per quarter note (MPQ). The factor at which the sequence is played can be adjusted as well.

```
sequencer.setTempoInBPM(1250f);
sequencer.setTempoInMPQ(4750f);
sequencer.setTempoFactor(1.5f);
```

When you finished using the [Sequencer](#), remember to close it

```
sequencer.close();
```

Section 162.2: Play an Audio file Looped

Needed imports:

```
import javax.sound.sampled.AudioSystem;
import javax.sound.sampled.Clip;
```

This code will create a clip and play it continuously once started:

```
Clip clip = AudioSystem.getClip();
clip.open(AudioSystem.getAudioInputStream(new URL(filename)));
clip.start();
clip.loop(Clip.LOOP_CONTINUOUSLY);
```

Get an Array with all supported file types:

```
AudioFileFormat.Type [] audioFileTypes = AudioSystem.getAudioFileTypes();
```

Section 162.3: Basic audio output

The Hello Audio! of Java that plays a sound file from local or internet storage looks as follows. It works for uncompressed .wav files and should not be used for playing mp3 or compressed files.

```
import java.io.*;
import java.net.URL;
import javax.sound.sampled.*;

public class SoundClipTest {

    // Constructor
    public SoundClipTest() {
        try {
            // Open an audio input stream.
            File soundFile = new File("/usr/share/sounds/alsa/Front_Center.wav"); //you could also
            get the sound file with an URL
            AudioInputStream audioIn = AudioSystem.getAudioInputStream(soundFile);
            AudioFormat format = audioIn.getFormat();
        }
    }
}
```

```

// 获取一个声音剪辑资源。
DataLine.Info info = new DataLine.Info(Clip.class, format);
Clip clip = (Clip)AudioSystem.getLine(info);
// 打开音频剪辑并从音频输入流加载样本。
clip.open(audioIn);
clip.start();
} catch (UnsupportedAudioFileException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
} catch (LineUnavailableException e) {
e.printStackTrace();
}
}

public static void main(String[] args) {
new SoundClipTest();
}
}

```

第162.4节：裸机声音

使用Java制作声音时，你也可以几乎做到裸机操作。此代码将原始二进制数据写入操作系统的音频缓冲区以生成声音。理解生成此类声音的限制和必要计算非常重要。由于播放基本上是即时的，计算需要几乎实时完成。

因此，此方法不适用于更复杂的声音采样。对于此类用途，使用专业工具是更好的方法。

下面的方法生成并直接输出给定频率、给定音量和给定持续时间的矩形波。

```

public void rectangleWave(byte volume, int hertz, int msecs) {
final SourceDataLine dataLine;
// 24 kHz x 8位，单声道，有符号小端AudioFormat
AudioFormat af = new AudioFormat(24_000, 8, 1, true, false);
try {
dataLine = AudioSystem.getSourceDataLine(af);
dataLine.open(af, 10_000); // 音频缓冲区大小：10k样本
} catch (LineUnavailableException e) {
throw new RuntimeException(e);
}

int waveHalf = 24_000 / hertz; // 半周期的采样数
byte[] buffer = new byte[waveHalf * 20];
int samples = msecs * (24_000 / 1000); // 24k (采样数/秒) / 1000 (毫秒/秒) * 时间 (毫秒)

dataLine.start(); // 开始播放
int sign = 1;

for (int i = 0; i < samples; i += buffer.length) {
for (int j = 0; j < 20; j++) { // 生成10个波形到缓冲区
sign *= -1;
// 用音量填充从第j个半波到第j+1个半波
Arrays.fill(buffer, waveHalf * j, waveHalf * (j+1), (byte) (volume * sign));
}
dataLine.write(buffer, 0, buffer.length); //
}
dataLine.drain(); // 强制缓冲区排空到硬件
}

```

```

// Get a sound clip resource.
DataLine.Info info = new DataLine.Info(Clip.class, format);
Clip clip = (Clip)AudioSystem.getLine(info);
// Open audio clip and load samples from the audio input stream.
clip.open(audioIn);
clip.start();
} catch (UnsupportedAudioFileException e) {
e.printStackTrace();
} catch (IOException e) {
e.printStackTrace();
} catch (LineUnavailableException e) {
e.printStackTrace();
}

public static void main(String[] args) {
new SoundClipTest();
}
}

```

Section 162.4: Bare metal sound

You can also go almost bare-metal when producing sound with java. This code will write raw binary data into the OS audio buffer to generate sound. It's extremely important to understand the limitations and necessary calculations to generating sound like this. Since playback is basically instantaneous, calculations need to be performed at almost real-time.

As such this method is unusable for more complicated sound-sampling. For such purposes using specialized tools is the better approach.

The following method generates and directly outputs a rectangle-wave of a given frequency in a given volume for a given duration.

```

public void rectangleWave(byte volume, int hertz, int msecs) {
final SourceDataLine dataLine;
// 24 kHz x 8bit, single-channel, signed little endian AudioFormat
AudioFormat af = new AudioFormat(24_000, 8, 1, true, false);
try {
dataLine = AudioSystem.getSourceDataLine(af);
dataLine.open(af, 10_000); // audio buffer size: 10k samples
} catch (LineUnavailableException e) {
throw new RuntimeException(e);
}

int waveHalf = 24_000 / hertz; // samples for half a period
byte[] buffer = new byte[waveHalf * 20];
int samples = msecs * (24_000 / 1000); // 24k (samples / sec) / 1000 (ms/sec) * time(ms)

dataLine.start(); // starts playback
int sign = 1;

for (int i = 0; i < samples; i += buffer.length) {
for (int j = 0; j < 20; j++) { // generate 10 waves into buffer
sign *= -1;
// fill from the jth wave-half to the j+1th wave-half with volume
Arrays.fill(buffer, waveHalf * j, waveHalf * (j+1), (byte) (volume * sign));
}
dataLine.write(buffer, 0, buffer.length); //
}
dataLine.drain(); // forces buffer drain to hardware
}

```

```
dataLine.stop(); // 结束播放  
}
```

为了更细致地生成不同的声波，正弦计算和可能更大的采样大小是必要的。
这会导致代码显著复杂，因此此处省略。

```
dataLine.stop(); // ends playback  
}
```

For a more differentiated way to generate different soundwaves sinus calculations and possibly larger sample sizes are necessary. This results in significantly more complex code and is accordingly omitted here.

第163章：Java打印服务

Java打印服务API提供了发现打印服务和发送打印请求的功能。

它包括基于IETF规范RFC 2911中互联网打印协议（IPP）1.1标准属性的可扩展打印属性。

第163.1节：构建将被打印的文档

Doc是一个接口，Java打印服务API提供了一个简单的实现，称为SimpleDoc。

每个Doc实例基本上由两个方面组成：

- 打印数据内容本身（电子邮件、图像、文档等）
- 打印数据格式，称为DocFlavor（MIME类型 + 表示类）。

在创建Doc对象之前，我们需要从某处加载文档。在示例中，我们将从磁盘加载一个特定文件：

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

所以现在，我们必须选择一个与我们的内容匹配的DocFlavor。DocFlavor类有许多常量来表示最常见的数据类型。让我们选择INPUT_STREAM.PDF这一种：

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

现在，我们可以创建一个新的 SimpleDoc 实例：

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor, null);
```

现在 doc 对象可以发送到打印作业请求（参见从打印服务创建打印作业）。

第163.2节：发现可用的打印服务

要发现所有可用的打印服务，我们可以使用 PrintServiceLookup 类。来看一下如何操作：

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringAvailablePrintServices {

    public static void main(String[] args) {
        discoverPrintServices();
    }

    public static void discoverPrintServices() {
        PrintService[] allPrintServices = PrintServiceLookup.lookupPrintServices(null, null);

        for (Printservice printService : allPrintServices) {
            System.out.println("打印服务名称: " + printService.getName());
        }
    }
}
```

Chapter 163: Java Print Service

The [Java Print Service API](#) provides functionalities to discover print services and send print requests for them.

It includes extensible print attributes based on the standard attributes specified in the [Internet Printing Protocol \(IPP\) 1.1](#) from the IETF Specification, [RFC 2911](#).

Section 163.1: Building the Doc that will be printed

Doc is an interface and the Java Print Service API provide a simple implementation called SimpleDoc.

Every Doc instance is basically made of two aspects:

- the print data content itself (an E-mail, an image, a document etc)
- the print data format, called DocFlavor (MIME type + Representation class).

Before creating the Doc object, we need to load our document from somewhere. In the example, we will load an specific file from the disk:

```
FileInputStream pdfFileInputStream = new FileInputStream("something.pdf");
```

So now, we have to choose a DocFlavor that matches our content. The DocFlavor class has a bunch of constants to represent the most usual types of data. Let's pick the INPUT_STREAM.PDF one:

```
DocFlavor pdfDocFlavor = DocFlavor.INPUT_STREAM.PDF;
```

Now, we can create a new instance of SimpleDoc:

```
Doc doc = new SimpleDoc(pdfFileInputStream, pdfDocFlavor, null);
```

The doc object now can be sent to the print job request (see Creating a print job from a print service).

Section 163.2: Discovering the available print services

To discovery all the available print services, we can use the PrintServiceLookup class. Let's see how:

```
import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringAvailablePrintServices {

    public static void main(String[] args) {
        discoverPrintServices();
    }

    public static void discoverPrintServices() {
        PrintService[] allPrintServices = PrintServiceLookup.lookupPrintServices(null, null);

        for (Printservice printService : allPrintServices) {
            System.out.println("Print service name: " + printService.getName());
        }
    }
}
```

当该程序在Windows环境下执行时，将打印如下内容：

```
打印服务名称: 传真  
打印服务名称: Microsoft Print to PDF  
打印服务名称: Microsoft XPS 文档查看器
```

第163.3节：定义打印请求属性

有时我们需要确定打印请求的某些方面。我们称之为属性。

打印请求属性的示例有：

- 份数 (1, 2等) ,
- 方向 (纵向或横向)
- 色彩 (单色, 彩色)
- 质量 (草稿, 普通, 高级)
- 单双面 (单面, 双面等)
- 等等...

在选择其中一个以及每个属性的值之前，首先需要构建一组属性：

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

现在我们可以添加它们。一些示例是：

```
pras.add(new Copies(5));  
pras.add(MediaSize.ISO_A4);  
pras.add(OrientationRequested.PORTRAIT);  
pras.add(PrintQuality.NORMAL);
```

现在pras对象可以发送到打印作业请求（参见从打印服务创建打印作业）。

第163.4节：监听打印作业请求状态变化

对于大多数打印客户端来说，了解打印作业是否完成或失败非常有用。

Java打印服务API提供了一些功能来获取这些场景的信息。我们所要做的就是：

- 为PrintJobListener接口提供一个实现，并
- 在打印作业中注册该实现。

当打印作业状态发生变化时，我们将收到通知。我们可以执行任何需要的操作，例如：

- 更新用户界面，
- 启动另一个业务流程，
- 在数据库中记录某些内容，
- 或者简单地记录日志。

在下面的示例中，我们将记录每个打印作业状态的变化：

```
import javax.print.event.PrintJobEvent;  
import javax.print.event.PrintJobListener;  
  
public class LoggerPrintJobListener implements PrintJobListener {
```

This program, when executed on a Windows environment, will print something like this:

```
Print service name: Fax  
Print service name: Microsoft Print to PDF  
Print service name: Microsoft XPS Document Viewer
```

Section 163.3: Defining print request attributes

Sometimes we need to determine some aspects of the print request. We will call them *attribute*.

Are examples of print request attributes:

- amount of copies (1, 2 etc),
- orientation (portrait or landscape)
- chromacity (monochrome, color)
- quality (draft, normal, high)
- sides (one-sided, two-sided etc)
- and so on...

Before choosing one of them and which value each one will have, first we need to build a set of attributes:

```
PrintRequestAttributeSet pras = new HashPrintRequestAttributeSet();
```

Now we can add them. Some examples are:

```
pras.add(new Copies(5));  
pras.add(MediaSize.ISO_A4);  
pras.add(OrientationRequested.PORTRAIT);  
pras.add(PrintQuality.NORMAL);
```

The pras object now can be sent to the print job request (see Creating a print job from a print service).

Section 163.4: Listening print job request status change

For the most printing clients, is extremely useful to know if a print job has finished or failed.

The Java Print Service API provide some functionalities to get informed about these scenarios. All we have to do is:

- provide an implementation for PrintJobListener interface and
- register this implementation at the print job.

When the print job state changes, we will be notified. We can do anything is needed, for example:

- update a user interface,
- start another business process,
- record something in the database,
- or simply log it.

In the example bellow, we will log every print job status change:

```
import javax.print.event.PrintJobEvent;  
import javax.print.event.PrintJobListener;  
  
public class LoggerPrintJobListener implements PrintJobListener {
```

```
// 你喜欢的 Logger 类写在这里!
private static final Logger LOG = Logger.getLogger(LoggerPrintJobListener.class);

public void printDataTransferCompleted(PrintJobEvent pje) {
    LOG.info("打印数据传输完成 ;)");
}

public void printJobCompleted(PrintJobEvent pje) {
    LOG.info("打印任务完成 =)");
}

public void printJobFailed(PrintJobEvent pje) {
    LOG.info("打印任务失败 =( ");
}

public void printJobCanceled(PrintJobEvent pje) {
    LOG.info("打印任务已取消 :| ");
}

public void printJobNoMoreEvents(PrintJobEvent pje) {
    LOG.info("任务没有更多事件 ");
}

public void printJobRequiresAttention(PrintJobEvent pje) {
    LOG.info("打印任务需要注意 :O ");
}
}
```

最后，我们可以在打印请求之前，将打印作业监听器的实现添加到打印作业上，具体如下：

```
DocPrintJob printJob = printService.createPrintJob();

printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

PrintJobEvent pje 参数

注意每个方法都有一个 PrintJobEvent pje 参数。为了简化示例，我们没有使用它，但你可以用它来查看状态。例如：

```
pje.getPrintJob().getAttributes();
```

将返回一个 PrintJobAttributeSet 对象实例，你可以用 foreach 方式遍历它们。

实现相同目标的另一种方法

实现相同目标的另一种选择是继承 PrintJobAdapter 类，顾名思义，它是 PrintJobListener 的适配器。实现接口时必须实现所有方法，而这种方式的优点是只需重写想要的方法。来看一下它是如何工作的：

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobAdapter;

public class LoggerPrintJobAdapter extends PrintJobAdapter {

    // 你喜欢的 Logger 类写在这里!
}
```

```
// Your favorite Logger class goes here!
private static final Logger LOG = Logger.getLogger(LoggerPrintJobListener.class);

public void printDataTransferCompleted(PrintJobEvent pje) {
    LOG.info("Print data transfer completed ;)");
}

public void printJobCompleted(PrintJobEvent pje) {
    LOG.info("Print job completed =)");
}

public void printJobFailed(PrintJobEvent pje) {
    LOG.info("Print job failed =( ");
}

public void printJobCanceled(PrintJobEvent pje) {
    LOG.info("Print job canceled :| ");
}

public void printJobNoMoreEvents(PrintJobEvent pje) {
    LOG.info("No more events to the job ");
}

public void printJobRequiresAttention(PrintJobEvent pje) {
    LOG.info("Print job requires attention :O ");
}
}
```

Finally, we can add our print job listener implementation on the print job before the print request itself, as follows:

```
DocPrintJob printJob = printService.createPrintJob();

printJob.addPrintJobListener(new LoggerPrintJobListener());

printJob.print(doc, pras);
```

The PrintJobEvent pje argument

Notice that every method has a PrintJobEvent pje argument. We don't use it in this example for simplicity purposes, but you can use it to explore the status. For example:

```
pje.getPrintJob().getAttributes();
```

Will return a PrintJobAttributeSet object instance and you can run them in a for-each way.

Another way to achieve the same goal

Another option to achieve the same goal is extending the PrintJobAdapter class, as the name says, is an adapter for PrintJobListener. Implementing the interface we compulsorily have to implement all of them. The advantage of this way it's we need to override only the methods we want. Let's see how it works:

```
import javax.print.event.PrintJobEvent;
import javax.print.event.PrintJobAdapter;

public class LoggerPrintJobAdapter extends PrintJobAdapter {

    // Your favorite Logger class goes here!
}
```

```

private static final Logger LOG = Logger.getLogger(LoggerPrintJobAdapter.class);

public void printJobCompleted(PrintJobEvent pje) {
    LOG.info("打印任务完成 =");
}

public void printJobFailed(PrintJobEvent pje) {
    LOG.info("打印任务失败 =( ");
}

```

注意我们只重写了一些特定的方法。

与实现接口PrintJobListener的示例相同，我们在发送打印任务之前将监听器添加到打印任务中：

```

printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);

```

第163.5节：发现默认打印服务

要发现默认打印服务，我们可以使用PrintServiceLookup类。来看一下如何操作：

```

import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringDefaultPrintService {

    public static void main(String[] args) {
        discoverDefaultPrintService();
    }

    public static void discoverDefaultPrintService() {
        PrintService defaultPrintService = PrintServiceLookup.lookupDefaultPrintService();
        System.out.println("默认打印服务名称: " + defaultPrintService.getName());
    }
}

```

第163.6节：从打印服务创建打印任务

打印任务是对特定打印服务中打印某物的请求。它基本上包括：

- 将要打印的数据（参见构建将要打印的文档）
- 一组属性

选择合适的打印服务实例后，我们可以请求创建一个打印任务：

```
DocPrintJob printJob = printService.createPrintJob();
```

DocPrintJob接口为我们提供了print方法：

```
printJob.print(doc, pras);
```

doc参数是一个Doc：将要打印的数据。

```

private static final Logger LOG = Logger.getLogger(LoggerPrintJobAdapter.class);

public void printJobCompleted(PrintJobEvent pje) {
    LOG.info("Print job completed = ");
}

public void printJobFailed(PrintJobEvent pje) {
    LOG.info("Print job failed =( ");
}

```

Notice that we override only some specific methods.

As the same way in the example implementing the interface PrintJobListener, we add the listener to the print job before sending it to print:

```

printJob.addPrintJobListener(new LoggerPrintJobAdapter());

printJob.print(doc, pras);

```

Section 163.5: Discovering the default print service

To discovery the default print service, we can use the PrintServiceLookup class. Let's see how::

```

import javax.print.PrintService;
import javax.print.PrintServiceLookup;

public class DiscoveringDefaultPrintService {

    public static void main(String[] args) {
        discoverDefaultPrintService();
    }

    public static void discoverDefaultPrintService() {
        PrintService defaultPrintService = PrintServiceLookup.lookupDefaultPrintService();
        System.out.println("Default print service name: " + defaultPrintService.getName());
    }
}

```

Section 163.6: Creating a print job from a print service

A print job is a request of printing something in a specific print service. It consists, basically, by:

- the data that will be printed (see Building the Doc that will be printed)
- a set of attributes

After picking-up the right print service instance, we can request the creation of a print job:

```
DocPrintJob printJob = printService.createPrintJob();
```

The DocPrintJob interface provide us the print method:

```
printJob.print(doc, pras);
```

The doc argument is a Doc: the data that will be printed.

pras参数是一个PrintRequestAttributeSet接口：一组PrintRequestAttribute。打印请求属性的示例有：

- 份数 (1, 2等) ,
- 方向 (纵向或横向)
- 色彩 (单色, 彩色)
- 质量 (草稿, 普通, 高级)
- 单双面 (单面, 双面等)
- 等等...

print方法可能会抛出一个PrintException。

And the pras argument is a PrintRequestAttributeSet interface: a set of PrintRequestAttribute. Are examples of print request attributes:

- amount of copies (1, 2 etc),
- orientation (portrait or landscape)
- chromacity (monochrome, color)
- quality (draft, normal, high)
- sides (one-sided, two-sided etc)
- and so on...

The print method may throw a PrintException.

第164章：CompletableFuture

CompletableFuture是Java SE 8新增的一个类，实现了Java SE 5中的Future接口。除了支持Future接口外，它还添加了许多方法，允许在future完成时进行异步回调。

第164.1节：CompletableFuture的简单示例

在下面的示例中，calculateShippingPrice方法计算运费，这需要一些处理时间。在实际应用中，这可能是联系另一个服务器，根据产品重量和运输方式返回价格。

通过使用CompletableFuture以异步方式建模，我们可以在方法中继续进行其他工作（即计算包装费用）。

```
public static void main(String[] args) {
    int price = 15; // 这里为了简单起见，使用整数价格
    int weightInGrams = 900;

    calculateShippingPrice(weightInGrams) // 这里，我们获得future
        .thenAccept(shippingPrice -> { // 然后立即对其进行处理！
            // 这种流式风格非常有助于保持代码简洁
            System.out.println("您的总价是: " + (price + shippingPrice));
        });
    System.out.println("请稍候。我们正在计算您的总价。");
}

public static CompletableFuture<Integer> calculateShippingPrice(int weightInGrams) {
    return CompletableFuture.supplyAsync(() -> {
        // supplyAsync 是一个工厂方法，将给定的
        // Supplier<U> 转换为 CompletableFuture<U>

        // 假设每 200 克是运费增加 1 美元
        int shippingCosts = weightInGrams / 200;

        try {
            Thread.sleep(2000L); // 现在模拟一些等待时间...
        } catch(InterruptedException e) { /* 我们可以安全地忽略它 */ }

        return shippingCosts; // 并返回费用！
    });
}
```

Chapter 164: CompletableFuture

CompletableFuture is a class added to Java SE 8 which implements the Future interface from Java SE 5. In addition to supporting the Future interface it adds many methods that allow asynchronous callback when the future is completed.

Section 164.1: Simple Example of CompletableFuture

In the example below, the calculateShippingPrice method calculates shipping cost, which takes some processing time. In a real world example, this would e.g. be contacting another server which returns the price based on the weight of the product and the shipping method.

By modeling this in an async way via CompletableFuture, we can continue different work in the method (i.e. calculating packaging costs).

```
public static void main(String[] args) {
    int price = 15; // Let's keep it simple and work with whole number prices here
    int weightInGrams = 900;

    calculateShippingPrice(weightInGrams) // Here, we get the future
        .thenAccept(shippingPrice -> { // And then immediately work on it!
            // This fluent style is very useful for keeping it concise
            System.out.println("Your total price is: " + (price + shippingPrice));
        });
    System.out.println("Please stand by. We are calculating your total price.");
}

public static CompletableFuture<Integer> calculateShippingPrice(int weightInGrams) {
    return CompletableFuture.supplyAsync(() -> {
        // supplyAsync is a factory method that turns a given
        // Supplier<U> into a CompletableFuture<U>

        // Let's just say each 200 grams is a new dollar on your shipping costs
        int shippingCosts = weightInGrams / 200;

        try {
            Thread.sleep(2000L); // Now let's simulate some waiting time...
        } catch(InterruptedException e) { /* We can safely ignore that */ }

        return shippingCosts; // And send the costs back!
    });
}
```

第165章：运行时命令

第165.1节：添加关闭钩子

有时你需要在程序停止时执行一段代码，比如释放你打开的系统资源。你可以使用addShutdownHook方法让线程在程序停止时运行：

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    ImportantStuff.someImportantInputStream.close();
}));
```

Chapter 165: Runtime Commands

Section 165.1: Adding shutdown hooks

Sometimes you need a piece of code to execute when the program stops, such as releasing system resources that you open. You can make a thread run when the program stops with the addShutdownHook method:

```
Runtime.getRuntime().addShutdownHook(new Thread(() -> {
    ImportantStuff.someImportantInputStream.close();
}));
```

第166章：单元测试

单元测试是测试驱动开发的重要组成部分，也是构建任何健壮应用程序的重要特性。在Java中，单元测试几乎完全依赖于外部库和框架进行，大多数都有自己的文档标签。本文档作为引导读者了解可用工具及其相应文档的入门介绍。

第166.1节：什么是单元测试？

这部分有点入门性质。主要是因为文档必须包含示例，即使它只是一个示范文章。如果你已经了解单元测试的基础知识，可以直接跳到备注部分，那里提到了具体的框架。

单元测试是确保某个模块按预期行为运行。在大型应用中，确保模块在独立环境下的正确执行，是保证应用程序准确性的关键部分。

请考虑以下（简单的）伪代码示例：

```
public class Example {
    public static void main (String args[]) {
        new Example();
    }

    // 应用层测试。
    public Example() {
        Consumer c = new Consumer();
        System.out.println("VALUE = " + c.getVal());
    }

    // 你的模块。
    class Consumer {
        private Capitalizer c;

        public Consumer() {
            c = new Capitalizer();
        }

        public String getVal() {
            return c.getVal();
        }
    }

    // 另一个团队的模块。
    class Capitalizer {
        private DataReader dr;

        public Capitalizer() {
            dr = new DataReader();
        }

        public String getVal() {
            return dr.readVal().toUpperCase();
        }
    }

    // 另一个团队的模块。
    class DataReader {
        public String readVal() {

```

Chapter 166: Unit Testing

Unit testing is an integral part of test-driven development, and an important feature for building any robust application. In Java, Unit testing is almost exclusively performed using external libraries and frameworks, most of which have their own documentation tag. This stub serves as a means of introducing the reader to the tools available, and their respective documentation.

Section 166.1: What is Unit Testing?

This is a bit of a primer. It's mostly put it in because documentation is forced to have an example, even if it's intended as a stub article. If you already know unit-testing basics, feel free to skip forward to the remarks, where specific frameworks are mentioned.

Unit testing is ensuring that a given module behaves as expected. In large-scale applications, ensuring the appropriate execution of modules in a vacuum is an integral part of ensuring application fidelity.

Consider the following (trivial) pseudo-example:

```
public class Example {
    public static void main (String args[]) {
        new Example();
    }

    // Application-level test.
    public Example() {
        Consumer c = new Consumer();
        System.out.println("VALUE = " + c.getVal());
    }

    // Your Module.
    class Consumer {
        private Capitalizer c;

        public Consumer() {
            c = new Capitalizer();
        }

        public String getVal() {
            return c.getVal();
        }
    }

    // Another team's module.
    class Capitalizer {
        private DataReader dr;

        public Capitalizer() {
            dr = new DataReader();
        }

        public String getVal() {
            return dr.readVal().toUpperCase();
        }
    }

    // Another team's module.
    class DataReader {
        public String readVal() {

```

```
// 指的是应用程序部署中的某个文件，或者
// 可能通过特定部署网络获取的文件。
File f;
String s = "data";
// ... 从 f 中读取数据到 s ...
return s;
}
}
```

所以这个例子很简单；DataReader 从文件中获取数据，传递给 Capitalizer，后者将所有字符转换为大写，然后传递给 Consumer。但 DataReader 与我们的应用环境紧密关联，因此我们推迟测试这条链，直到准备好发布测试版本。

现在，假设在某次发布过程中，出于未知原因，Capitalizer 中的 getVal() 方法从返回 toUpperCase() 字符串变成了返回 toLowerCase() 字符串：

```
// 另一个团队的模块。
class Capitalizer {
...
public String getVal() {
    return dr.readVal().toLowerCase();
}
}
```

显然，这打破了预期的行为。但由于执行DataReader涉及的繁琐过程，我们直到下一次测试部署时才会注意到这个问题。DataReader，我们不会注意到这一点，直到下一次测试部署。因此，几天/几周/几个月过去了，这个错误一直存在于我们的系统中，然后产品经理看到了这个问题，立刻转向你，作为与Consumer相关的团队负责人。“为什么会这样？你们改了什么？”显然，你一头雾水。你根本不知道发生了什么。你没有更改任何应该涉及这部分的代码；为什么它突然坏了？

最终，在团队之间的讨论和协作后，问题被追踪并解决了。但是，这引出了一个问题；这本可以如何避免？

有两个显而易见的方面：

测试需要自动化

我们对手动测试的依赖让这个错误长时间未被发现。我们需要一种方法，能够即时自动化检测引入的错误。不是5周后，不是5天后，也不是5分钟后。
就是现在。

你必须理解，在这个例子中，我表达的是一个非常微不足道的错误被引入且未被发现。在工业应用中，随着数十个模块不断更新，这类错误可能无处不在。你修复了一个模块的问题，却发现你“修复”的行为在其他地方（无论是内部还是外部）以某种方式被依赖。

如果没有严格的验证，问题会逐渐渗入系统。如果被忽视得足够久，可能会导致修复变更所需的额外工作量大幅增加（然后还要修复这些修复，等等），以至于产品的剩余工作量实际上会随着投入的努力而增加。你绝不想陷入这种境地。

测试需要细化

我们上面例子中提到的第二个问题是追踪错误所花费的时间。当测试人员发现问题时，产品经理联系了你，你调查后发现Capitalizer返回了

```
// Refers to a file somewhere in your application deployment, or
// perhaps retrieved over a deployment-specific network.
File f;
String s = "data";
// ... Read data from f into s ...
return s;
}
}
```

So this example is trivial; DataReader gets the data from a file, passes it to the Capitalizer, which converts all the characters to upper-case, which then gets passed to the Consumer. But the DataReader is heavily-linked to our application environment, so we defer testing of this chain until we are ready to deploy a test release.

Now, assume, somewhere along the way in a release, for reasons unknown, the getVal() method in Capitalizer changed from returning a toUpperCase() String to a toLowerCase() String:

```
// Another team's module.
class Capitalizer {
...
public String getVal() {
    return dr.readVal().toLowerCase();
}
}
```

Clearly, this breaks expected behavior. But, because of the arduous processes involved with execution of the DataReader, we won't notice this until our next test deployment. So days/weeks/months go by with this bug sitting in our system, and then the product manager sees this, and instantly turns to you, the team leader associated with the Consumer. "Why is this happening? What did you guys change?" Obviously, you're clueless. You have no idea what's going on. You didn't change any code that should be touching this; why is it suddenly broken?

Eventually, after discussion between the teams and collaboration, the issue is traced, and the problem solved. But, it begs the question; how could this have been prevented?

There are two obvious things:

Tests need to be automated

Our reliance upon manual testing let this bug go by unnoticed far too long. We need a way to automate the process under which bugs are introduced **instantly**. Not 5 weeks from now. Not 5 days from now. Not 5 minutes from now. Right now.

You have to appreciate that, in this example, I've expressed one **very trivial** bug that was introduced and unnoticed. In an industrial application, with dozens of modules constantly being updated, these can creep in all over the place. You fix something with one module, only to realize that the very behavior you "fixed" was relied upon in some manner elsewhere (either internally or externally).

Without rigorous validation, things will creep into the system. It's possible that, if neglected far enough, this will result in so much extra work trying to fix changes (and then fixing those fixes, etc.), that a product will actually **increase** in remaining work as effort is put into it. You do not want to be in this situation.

Tests need to be fine-grained

The second problem noted in our above example is the amount of time it took to trace the bug. The product manager pinged you when the testers noticed it, you investigated and found that the Capitalizer was returning

看似错误的数据，你将你的发现反馈给了Capitalizer团队，他们进行了调查，等等，等等。

我之前提到的关于这个简单例子中问题数量和难度的观点在这里同样适用。显然，任何对Java相当熟悉的人都能快速找到引入的问题。但追踪和沟通问题往往要困难得多。也许Capitalizer团队只给了你一个没有源码的JAR包。也许他们位于世界另一端，沟通时间非常有限（可能每天只发一次邮件）。这可能导致错误追踪需要数周甚至更长时间（而且，对于某个版本发布，可能有多个这样的错误）。

为了减轻这种情况，我们希望在尽可能细粒度的层面上进行严格测试（你也需要粗粒度测试以确保模块间正确交互，但这不是我们这里的重点）。我们希望严格规范所有对外功能（至少是对外功能）的操作，并为这些功能编写测试。

引入单元测试

想象一下，如果我们有一个测试，专门确保Capitalizer的getVal()方法对于给定的输入字符串返回一个大写字符串。更进一步，想象这个测试在我们提交任何代码之前就已经运行。系统中引入的错误（即将toUpperCase()替换为toLowerCase()）将不会造成任何问题，因为错误根本不会被引入系统。我们会在测试中发现它，开发者（希望如此）会意识到他们的错误，并会找到替代方案来实现他们想要的效果。

这里对如何实现这些测试有所省略，但这些内容在框架特定的文档中有详细说明（链接在备注中）。希望这能作为单元测试重要性的一个示例。

seemingly bad data, you pinged the Capitalizer team with your findings, they investigated, etc. etc. etc.

The same point I made above about the quantity and difficulty of this trivial example hold here. Obviously anyone reasonably well-versed with Java could find the introduced problem quickly. But it's often much, much more difficult to trace and communicate issues. Maybe the Capitalizer team provided you a JAR with no source. Maybe they're located on the other side of the world, and communication hours are very limited (perhaps to e-mails that get sent once daily). It can result in bugs taking weeks or longer to trace (and, again, there could be several of these for a given release).

In order to mitigate against this, we want rigorous testing on as **fine** a level as possible (you also want coarse-grained testing to ensure modules interact properly, but that's not our focal point here). We want to rigorously specify how all outward-facing functionality (at minimum) operates, and tests for that functionality.

Enter unit-testing

Imagine if we had a test, specifically ensuring that the getVal() method of Capitalizer returned a capitalized string for a given input string. Furthermore, imagine that test was run before we even committed any code. The bug introduced into the system (that is, toUpperCase() being replaced with toLowerCase()) would cause no issues because the bug would never **be** introduced into the system. We would catch it in a test, the developer would (hopefully) realize their mistake, and an alternative solution would be reached as to how to introduce their intended effect.

There's some omissions made here as to **how** to implement these tests, but those are covered in the framework-specific documentation (linked in the remarks). Hopefully, this serves as an example of **why** unit testing is important.

第167章：断言

参数

详情

expression1 断言语句在该表达式计算结果为false时抛出AssertionError。

expression2 可选。当使用时，断言语句抛出的AssertionError将带有此消息。

第167.1节：使用断言检查算术运算

```
a = 1 - Math.abs(1 - a % 2);
```

// 如果上述算术运算有误，将抛出错误。

```
assert a >= 0 && a <= 1 : "计算值 " + a + " 超出预期范围";
```

```
return a;
```

Chapter 167: Asserting

Parameter

Details

expression1 The assertion statement throws an AssertionException if this expression evaluates to **false**.

expression2 Optional. When used, AssertionErrors thrown by the assert statement have this message.

Section 167.1: Checking arithmetic with assert

```
a = 1 - Math.abs(1 - a % 2);
```

// This will throw an error if my arithmetic above is wrong.

```
assert a >= 0 && a <= 1 : "Calculated value of " + a + " is outside of expected bounds";
```

```
return a;
```

第168章：多版本发布的JAR文件

Java 9引入的特性之一是多版本Jar (MRJAR) , 它允许在同一个Jar文件中捆绑针对多个Java版本的代码。该特性在 JEP 238中进行了规范。

第168.1节：多版本Jar文件内容示例

通过在 `MANIFEST.MF` 文件中设置 `Multi-Release: true`, Jar 文件变成多版本 Jar, Java运行时（只要支持 MRJAR 格式）将根据当前的主版本选择合适版本的类。

这种 Jar 的结构如下：

```
jar 根目录
- A.class
- B.class
- C.class
- D.class
- META-INF
  - versions
    - 9
      - A.class
      - B.class
    - 10
      - A.class
```

- 在 JDK 9 之前，只有根入口中的类对 Java 运行时可见。
- 在 JDK 9 上，类 A 和 B 将从目录 `root/META-INF/versions/9` 加载，而类 C 和 D 将从基础入口加载。
- 在 JDK 10 上，类 A 将从目录 `root/META-INF/versions/10` 加载。

第 168.2 节：使用 jar 工具创建多版本 Jar

`jar` 命令可以用来创建一个多版本 Jar，其中包含针对 Java 8 和 Java 9 编译的同一类的两个版本，尽管会有警告提示这些类是相同的：

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
警告: 条目 META-INF/versions/9/demo/SampleClass.class 包含一个 class,
与 jar 中已存在的条目相同
```

`--release 9` 选项告诉 `jar` 将后续所有内容 (`sampleproject-9` 目录中的 `demo` 包) 包含在 MRJAR 的版本条目中，即位于 `root/META-INF/versions/9` 下。结果内容如下：

```
jar 根目录
- demo
  - SampleClass.class
-META-INF
  - versions
    - 9
      - demo
        - SampleClass.class
```

现在让我们创建一个名为 `Main` 的类，该类打印`SampleClass`的 URL，并将其添加到 Java 9 版本中：

Chapter 168: Multi-Release JAR Files

One of the features introduced in Java 9 is the multi-release Jar (MRJAR) which allows bundling code targeting multiple Java releases within the same Jar file. The feature is specified in [JEP 238](#).

Section 168.1: Example of a multi-release Jar file's contents

By setting `Multi-Release: true` in the `MANIFEST.MF` file, the Jar file becomes a multi-release Jar and the Java runtime (as long as it supports the MRJAR format) will pick the appropriate versions of classes depending on the current major version.

The structure of such a Jar is the following:

```
jar root
- A.class
- B.class
- C.class
- D.class
- META-INF
  - versions
    - 9
      - A.class
      - B.class
    - 10
      - A.class
```

- On JDks < 9, only the classes in the root entry are visible to the Java runtime.
- On a JDK 9, the classes A and B will be loaded from the directory `root/META-INF/versions/9`, while C and D will be loaded from the base entry.
- On a JDK 10, class A would be loaded from the directory `root/META-INF/versions/10`.

Section 168.2: Creating a multi-release Jar using the jar tool

The `jar` command can be used to create a multi-release Jar containing two versions of the same class compiled for both Java 8 and Java 9, albeit with a warning telling that the classes are identical:

```
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
```

The `--release 9` option tells `jar` to include everything that follows (the `demo` package inside the `sampleproject-9` directory) inside a versioned entry in the MRJAR, namely under `root/META-INF/versions/9`. The result is the following contents:

```
jar root
- demo
  - SampleClass.class
-META-INF
  - versions
    - 9
      - demo
        - SampleClass.class
```

Let us now create a class called `Main` that prints the URL of the `SampleClass`, and add it for the Java 9 version:

```

package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}

```

如果我们编译这个类并重新运行 jar 命令，会出现错误：

```

C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demoentry: META-INF/versions/9/demo/Main.class, 包含一个新的公共类，在基础条目中未找到
警告：条目 META-INF/versions/9/demo/Main.java，存在多个同名资源
警告：条目 META-INF/versions/9/demo/SampleClass.class 包含一个 class，该
与jar中已有条目相同
无效的多-发布jar文件MR.jar已删除

```

原因是jar工具阻止将公共类添加到版本化条目中，除非它们也被添加到基础条目中。这样做是为了使MRJAR对不同的Java版本暴露相同的公共API。请注意，在运行时，这条规则并非必须。它可能仅由像jar这样的工具应用。在这个特定情况下，目的在于 Main 是运行示例代码，因此我们可以简单地在基本条目中添加一个副本。如果该类是我们仅在 Java 9 中需要的较新实现的一部分，则可以将其设为非公开。

要将Main添加到根条目，我们首先需要将其编译为针对Java 9之前版本的目标。这可以使用javac的新--release选项来完成：

```

C:\Users\manouti\sampleproject-base\demo>javac --release 8 Main.java
C:\Users\manouti\sampleproject-base\demo>cd ../..
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo

```

运行 Main 类显示 SampleClass 是从版本目录加载的：

```

C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class

```

第168.3节：多版本Jar中已加载类的URL

给定以下多版本Jar：

```

jar 根目录
- demo
  - SampleClass.class
-META-INF
  - versions
    - 9
      - demo
        - SampleClass.class

```

下面的类打印了SampleClass的URL：

```

包 demo;

```

```

package demo;

import java.net.URL;

public class Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}

```

If we compile this class and re-run the jar command, we get an error:

```

C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demoentry: META-INF/versions/9/demo/Main.class, contains a new public class not
found in base entries
Warning: entry META-INF/versions/9/demo/Main.java, multiple resources with same name
Warning: entry META-INF/versions/9/demo/SampleClass.class contains a class that
is identical to an entry already in the jar
invalid multi-release jar file MR.jar deleted

```

The reason is that the jar tool prevents adding public classes to versioned entries if they are not added to the base entries as well. This is done so that the MRJAR exposes the same public API for the different Java versions. Note that at runtime, this rule is not required. It may be only applied by tools like jar. In this particular case, the purpose of Main is to run sample code, so we can simply add a copy in the base entry. If the class were part of a newer implementation that we only need for Java 9, it could be made non-public.

To add Main to the root entry, we first need to compile it to target a pre-Java 9 release. This can be done using the new --release option of javac:

```

C:\Users\manouti\sampleproject-base\demo>javac --release 8 Main.java
C:\Users\manouti\sampleproject-base\demo>cd ../..
C:\Users\manouti>jar --create --file MR.jar -C sampleproject-base demo --release 9 -C
sampleproject-9 demo

```

Running the Main class shows that the SampleClass gets loaded from the versioned directory:

```

C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class

```

Section 168.3: URL of a loaded class inside a multi-release Jar

Given the following multi-release Jar:

```

jar root
- demo
  - SampleClass.class
-META-INF
  - versions
    - 9
      - demo
        - SampleClass.class

```

The following class prints the URL of the SampleClass:

```

package demo;

```

```
导入 java.net.URL;

公共类 Main {

    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

如果该类被编译并添加到MRJAR中针对Java 9的版本条目上，运行它将导致：

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

```
import java.net.URL;

public class Main {

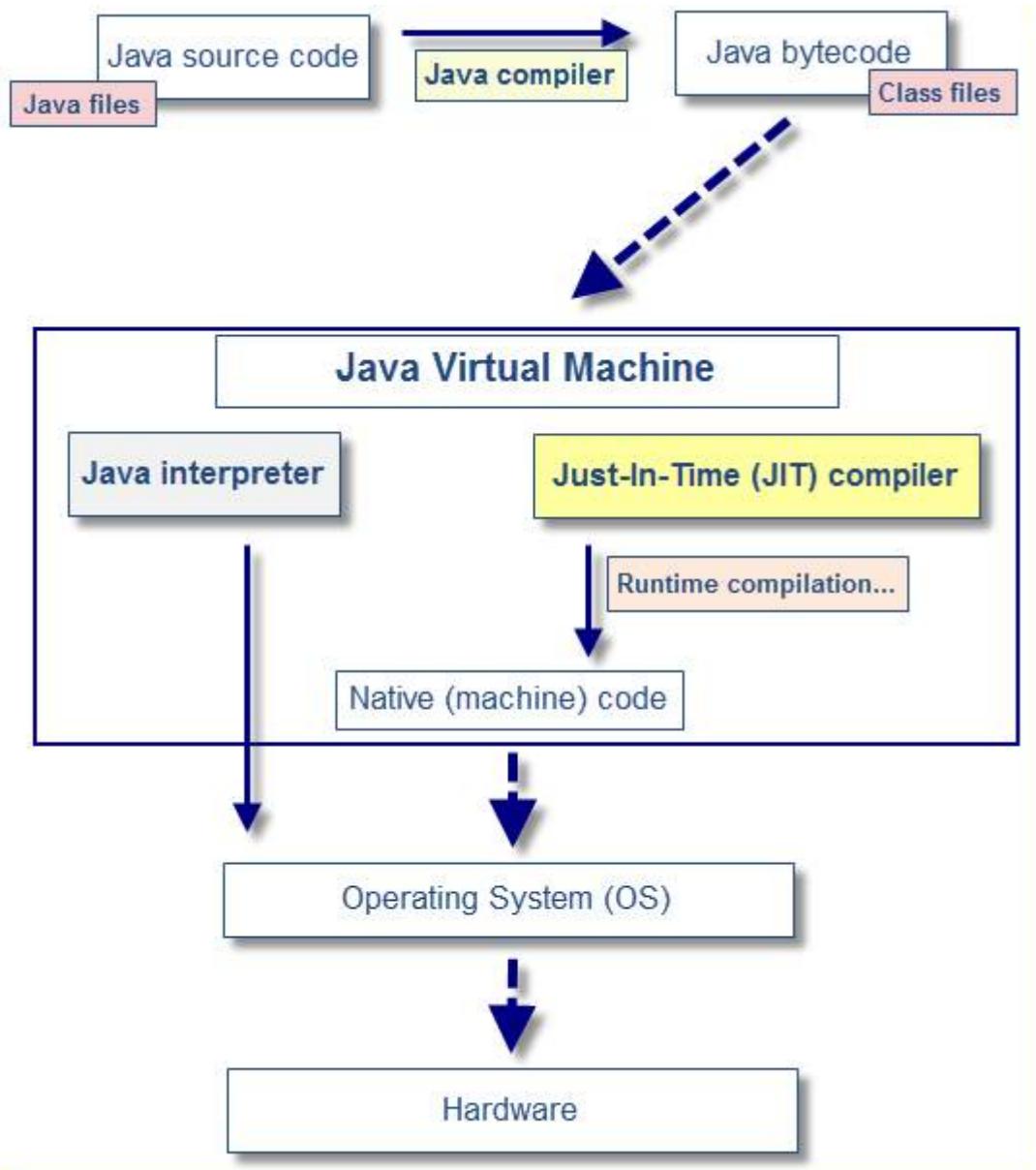
    public static void main(String[] args) throws Exception {
        URL url = Main.class.getClassLoader().getResource("demo/SampleClass.class");
        System.out.println(url);
    }
}
```

If the class is compiled and added on the versioned entry for Java 9 in the MRJAR, running it would result in:

```
C:\Users\manouti>java --class-path MR.jar demo.Main
jar:file:/C:/Users/manouti/MR.jar!/META-INF/versions/9/demo/SampleClass.class
```

第169章：即时编译器 (JIT)

169.1节：概述



即时编译器 (JIT) 是Java™运行环境的一个组件，用于提升Java应用程序在运行时的性能。

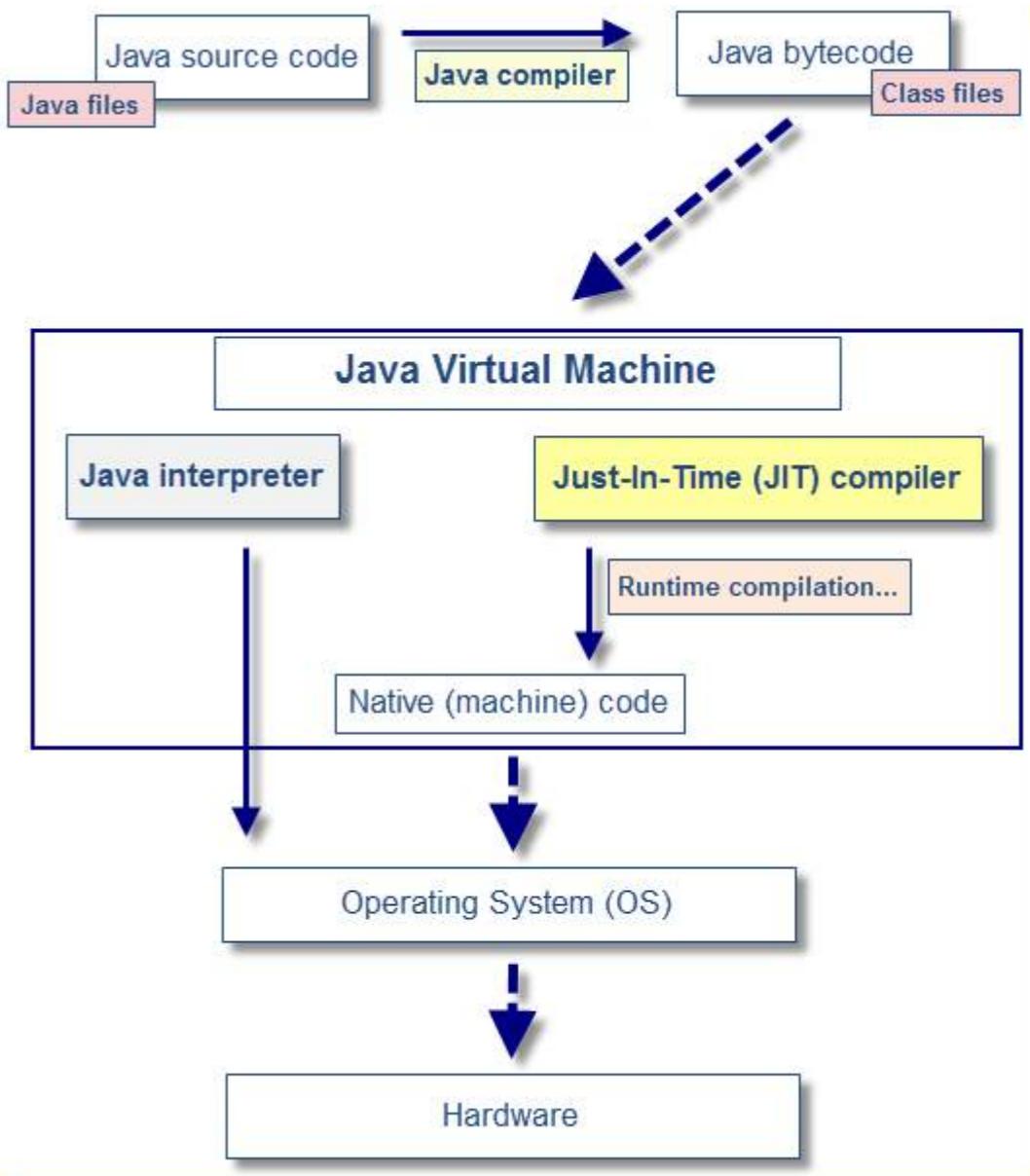
- Java程序由类组成，这些类包含平台无关的字节码，可以被JVM在多种不同计算机架构上解释执行。
- 在运行时，JVM加载类文件，确定每个字节码的语义，并执行相应的计算。

解释执行期间额外的处理器和内存使用导致Java应用程序的性能比本地应用程序更慢。

JIT编译器通过在运行时将字节码编译成本地机器码，帮助提升Java程序的性能。

Chapter 169: Just in Time (JIT) compiler

Section 169.1: Overview



The Just-In-Time (JIT) compiler is a component of the Java™ Runtime Environment that improves the performance of Java applications at run time.

- Java programs consists of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures.
- At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation.

The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application.

The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.

JIT 编译器默认启用，并在调用 Java 方法时激活。JIT 编译器将该方法的字节码编译成本地机器码，“即时”编译以运行。

当一个方法被编译后，JVM 会直接调用该方法的编译代码，而不是解释执行它。理论上，如果编译不需要处理器时间和内存使用，编译每个方法可以使 Java 程序的速度接近本地应用程序的速度。

JIT 编译确实需要处理器时间和内存使用。当 JVM 刚启动时，会调用成千上万个方法。编译所有这些方法会显著影响启动时间，即使程序最终能达到非常好的峰值性能。

- 实际上，方法在第一次被调用时不会被编译。对于每个方法，JVM 会维护一个调用计数 每次调用该方法时该计数都会增加。
- JVM 会解释执行一个方法，直到其调用计数超过 JIT 编译阈值。
- 因此，常用方法会在 JVM 启动后不久被编译，而不常用的方法则会在更晚的时候编译，或者根本不编译。
- JIT 编译阈值帮助 JVM 快速启动，同时仍能提升性能。
- 该阈值经过精心选择，以在启动时间和长期性能之间取得最佳平衡。
- 方法被编译后，其调用计数会被重置为零，随后对该方法的调用会继续增加其计数。
- 当一个方法的调用计数达到 JIT 重新编译阈值时，JIT 编译器会第二次编译该方法，应用比上一次编译更多的优化。
- 该过程会重复进行，直到达到最大优化级别。

Java程序中最繁忙的方法总是被最积极地优化，从而最大化使用JIT编译器的性能优势。

JIT编译器还可以在运行时测量操作数据，并利用这些数据来提高后续重新编译的质量。

JIT编译器可以被禁用，此时整个Java程序将被解释执行。除非用于诊断或解决JIT编译问题，否则不建议禁用JIT编译器。

The JIT compiler is enabled by default, and is activated when a Java method is called. The JIT compiler compiles the bytecodes of that method into native machine code, compiling it "[just in time](#)" to run.

When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to approach that of a native application.

JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

- In practice, methods are not compiled the first time they are called. For each method, the JVM maintains a call count which is incremented every time the method is called.
- The JVM interprets a method until its call count exceeds a JIT compilation threshold.
- Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all.
- The JIT compilation threshold helps the JVM start quickly and still have improved performance.
- The threshold has been carefully selected to obtain an optimal balance between startup times and long term performance.
- After a method is compiled, its call count is reset to zero and subsequent calls to the method continue to increment its count.
- When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles it a second time, applying a larger selection of optimizations than on the previous compilation.
- This process is repeated until the maximum optimization level is reached.

The busiest methods of a Java program are always optimized most aggressively, maximizing the performance benefits of using the JIT compiler.

The JIT compiler can also measure operational data at run time, and use that data to improve the quality of further recompilations.

The JIT compiler can be disabled, in which case the entire Java program will be interpreted. Disabling the JIT compiler is not recommended except to diagnose or work around JIT compilation problems.

第170章：字节码修改

第170.1节：什么是字节码？

字节码是JVM使用的一组指令。为说明这一点，我们来看这个Hello World程序。

```
public static void main(String[] args){  
    System.out.println("Hello World");  
}
```

编译成字节码后，它变成了这样。

```
public static main([Ljava/lang/String; args)V  
    getstatic java/lang/System out Ljava/io/PrintStream;  
    ldc "Hello World"  
    invokevirtual java/io/PrintStream print(Ljava/lang/String;)V
```

这背后的逻辑是什么？

getstatic - 获取类的静态字段的值。在这里，是System的PrintStream "Out"。

ldc - 将一个常量压入栈中。在这里，是字符串 "Hello World"

invokevirtual - 调用栈上已加载引用的方法，并将结果放回栈中。方法的参数也从栈中获取。

嗯，肯定还有更多吧？

共有255个操作码，但并非全部都已实现。所有当前操作码的表格可以在这里找到：[Java字节码指令列表](#)。

我如何编写/编辑字节码？

有多种方法可以编写和编辑字节码。你可以使用编译器、使用库，或者使用程序。

用于编写：

- [茉莉](#)
- [喀拉喀托](#)

用于编辑：

- 库
 - [ASM](#)
 - [Javassist](#)
 - [BCEL](#) - 不支持 Java 8+
- 工具
 - [字节码查看器](#)
 - [JBytedit](#)
 - [rej](#) - 不支持 Java 8+
 - [JBE](#) - 不支持 Java 8+

我想了解更多关于字节码的内容！

可能有专门针对字节码的文档页面。此页面重点介绍的是修改

Chapter 170: Bytecode Modification

Section 170.1: What is Bytecode?

Bytecode is the set of instructions used by the JVM. To illustrate this let's take this Hello World program.

```
public static void main(String[] args){  
    System.out.println("Hello World");  
}
```

This is what it turns into when compiled into bytecode.

```
public static main([Ljava/lang/String; args)V  
    getstatic java/lang/System out Ljava/io/PrintStream;  
    ldc "Hello World"  
    invokevirtual java/io/PrintStream print(Ljava/lang/String;)V
```

What's the logic behind this?

getstatic - Retrieves the value of a static field of a class. In this case, the *PrintStream "Out"* of *System*.

ldc - Push a constant onto the stack. In this case, the String "Hello World"

invokevirtual - Invokes a method on a loaded reference on the stack and puts the result on the stack. Parameters of the method are also taken from the stack.

Well, there has to be more right?

There are 255 opcodes, but not all of them are implemented yet. A table with all of the current opcodes can be found here: [Java bytecode instruction listings](#).

How can I write / edit bytecode?

There's multiple ways to write and edit bytecode. You can use a compiler, use a library, or use a program.

For writing:

- [Jasmin](#)
- [Krakatau](#)

For editing:

- Libraries
 - [ASM](#)
 - [Javassist](#)
 - [BCEL](#) - Doesn't support Java 8+
- Tools
 - [Bytecode-Viewer](#)
 - [JBytedit](#)
 - [rej](#) - Doesn't support Java 8+
 - [JBE](#) - Doesn't support Java 8+

I'd like to learn more about bytecode!

There's probably a specific documentation page specifically for bytecode. This page focuses on the modification of

使用不同的库和工具的字节码。

第170.2节：如何使用ASM编辑jar文件

首先需要加载jar中的类。我们将使用三种方法来完成此过程：

- loadClasses(File)
- readJar(JarFile, JarEntry, Map)
- getNode(byte[])

```
Map<String, ClassNode> loadClasses(File jarFile) throws IOException {
    Map<String, ClassNode> classes = new HashMap<String, ClassNode>();
    JarFile jar = new JarFile(jarFile);
    Stream<JarEntry> str = jar.stream();
    str.forEach(z -> readJar(jar, z, classes));
    jar.close();
    return classes;
}

Map<String, ClassNode> readJar(JarFile jar, JarEntry entry, Map<String, ClassNode> classes) {
    String name = entry.getName();
    try (InputStream jis = jar.getInputStream(entry)){
        if (name.endsWith(".class")) {
            byte[] bytes = IOUtils.toByteArray(jis);
            String cafebabe = String.format("%02X%02X%02X%02X", bytes[0], bytes[1], bytes[2], bytes[3]);
            if (!cafebabe.toLowerCase().equals("cafebabe")) {
                // 该类没有有效的魔数
                return classes;
            }
            try {
                ClassNode cn = getNode(bytes);
                classes.put(cn.name, cn);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return classes;
}

ClassNode getNode(byte[] bytes) {
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
    } catch (Exception e) {
        e.printStackTrace();
    }
    cr = null;
    return cn;
}
```

通过这些方法，加载和修改jar文件变成了在映射中更改ClassNode的简单操作。
在本例中，我们将使用Tree API将jar中的所有字符串替换为大写形式。

```
File jarFile = new File("sample.jar");
Map<String, ClassNode> nodes = loadClasses(jarFile);
```

bytecode using different libraries and tools.

Section 170.2: How to edit jar files with ASM

Firstly the classes from the jar need to be loaded. We'll use three methods for this process:

- loadClasses(File)
- readJar(JarFile, JarEntry, Map)
- getNode(byte[])

```
Map<String, ClassNode> loadClasses(File jarFile) throws IOException {
    Map<String, ClassNode> classes = new HashMap<String, ClassNode>();
    JarFile jar = new JarFile(jarFile);
    Stream<JarEntry> str = jar.stream();
    str.forEach(z -> readJar(jar, z, classes));
    jar.close();
    return classes;
}

Map<String, ClassNode> readJar(JarFile jar, JarEntry entry, Map<String, ClassNode> classes) {
    String name = entry.getName();
    try (InputStream jis = jar.getInputStream(entry)){
        if (name.endsWith(".class")) {
            byte[] bytes = IOUtils.toByteArray(jis);
            String cafebabe = String.format("%02X%02X%02X%02X", bytes[0], bytes[1], bytes[2], bytes[3]);
            if (!cafebabe.toLowerCase().equals("cafebabe")) {
                // This class doesn't have a valid magic
                return classes;
            }
            try {
                ClassNode cn = getNode(bytes);
                classes.put(cn.name, cn);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return classes;
}

ClassNode getNode(byte[] bytes) {
    ClassReader cr = new ClassReader(bytes);
    ClassNode cn = new ClassNode();
    try {
        cr.accept(cn, ClassReader.EXPAND_FRAMES);
    } catch (Exception e) {
        e.printStackTrace();
    }
    cr = null;
    return cn;
}
```

With these methods loading and changing a jar file becomes a simple matter of changing ClassNodes in a map. In this example we will replace all Strings in the jar with capitalized ones using the Tree API.

```
File jarFile = new File("sample.jar");
Map<String, ClassNode> nodes = loadClasses(jarFile);
```

```

// 遍历 ClassNodes
for (ClassNode cn : nodes.values()){
    // 遍历类中的方法
    for (MethodNode mn : cn.methods){
        // 遍历方法中的指令
        for (AbstractInsnNode ain : mn.instructions.toArray()){
            // 如果指令是加载常量值
            if (ain.getOpcode() == Opcodes.LDC){
                // 将当前指令强制转换为 Ldc
                // 如果常量是字符串，则将其首字母大写。
                LdcInsnNode ldc = (LdcInsnNode) ain;
                if (ldc.cst instanceof String){
                    ldc.cst = ldc.cst.toString().toUpperCase();
                }
            }
        }
    }
}

```

既然所有 ClassNode 的字符串都已被修改，我们需要保存这些更改。为了保存更改并生成可用的输出，需要完成以下几件事：

- 导出 ClassNode 为字节数组
- 加载非类的 jar 条目（例如：Manifest.mf / jar 中的其他二进制资源）为字节数组
- 将所有字节保存到新的 jar 文件中

根据上面最后一部分内容，我们将创建三个方法。

- processNodes(Map<String, ClassNode> nodes)
- loadNonClasses(File jarFile)
- saveAsJar(Map<String, byte[]> outBytes, String fileName)

用法：

```

Map<String, byte[]> out = process(nodes, new HashMap<String, MappedClass>());
out.putAll(loadNonClassEntries(jarFile));
saveAsJar(out, "sample-edit.jar");

```

使用的方法：

```

static Map<String, byte[]> processNodes(Map<String, ClassNode> nodes, Map<String, MappedClass> mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    // 遍历节点并将它们添加到 <类名, 类字节> 的映射中
    // 使用 Compute_Frames 确保堆栈帧会被自动重新计算
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name).getNewName() : cn.name,
                cw.toByteArray());
    }
    return out;
}

static Map<String, byte[]> loadNonClasses(File jarFile) throws IOException {
    Map<String, byte[]> entries = new HashMap<String, byte[]>();
    ZipInputStream jis = new ZipInputStream(new FileInputStream(jarFile));
    ZipEntry entry;
    // 遍历所有条目
    while ((entry = jis.getNextEntry()) != null) {

```

```

// Iterate ClassNodes
for (ClassNode cn : nodes.values()){
    // Iterate methods in class
    for (MethodNode mn : cn.methods){
        // Iterate instructions in method
        for (AbstractInsnNode ain : mn.instructions.toArray()){
            // If the instruction is loading a constant value
            if (ain.getOpcode() == Opcodes.LDC){
                // Cast current instruction to Ldc
                // If the constant is a string then capitalize it.
                LdcInsnNode ldc = (LdcInsnNode) ain;
                if (ldc.cst instanceof String){
                    ldc.cst = ldc.cst.toString().toUpperCase();
                }
            }
        }
    }
}

```

Now that all of the ClassNode's strings have been modified we need to save the changes. In order to save the changes and have a working output a few things have to be done:

- Export ClassNodes to bytes
- Load non-class jar entries (Ex: Manifest.mf / other binary resources in jar) as bytes
- Save all bytes to a new jar

From the last portion above, we'll create three methods.

- processNodes(Map<String, ClassNode> nodes)
- loadNonClasses(File jarFile)
- saveAsJar(Map<String, byte[]> outBytes, String fileName)

Usage:

```

Map<String, byte[]> out = process(nodes, new HashMap<String, MappedClass>());
out.putAll(loadNonClassEntries(jarFile));
saveAsJar(out, "sample-edit.jar");

```

The methods used:

```

static Map<String, byte[]> processNodes(Map<String, ClassNode> nodes, Map<String, MappedClass> mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    // Iterate nodes and add them to the map of <Class names , Class bytes>
    // Using Compute_Frames ensures that stack-frames will be re-calculated automatically
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name).getNewName() : cn.name,
                cw.toByteArray());
    }
    return out;
}

static Map<String, byte[]> loadNonClasses(File jarFile) throws IOException {
    Map<String, byte[]> entries = new HashMap<String, byte[]>();
    ZipInputStream jis = new ZipInputStream(new FileInputStream(jarFile));
    ZipEntry entry;
    // Iterate all entries
    while ((entry = jis.getNextEntry()) != null) {

```

```

try {
    String name = entry.getName();
    if (!name.endsWith(".class") && !entry.isDirectory()) {
        // Apache Commons - byte[] toByteArray(InputStream input)
        //
        // 将每个条目添加到映射 <条目名称, 条目字节>
        byte[] bytes = IOUtils.toByteArray(jis);
        entries.put(name, bytes);
    }
} catch (Exception e) {
e.printStackTrace();
} finally {
jis.closeEntry();
}
jis.close();
return entries;
}

static void saveAsJar(Map<String, byte[]> outBytes, String fileName) {
try {
    // 创建 jar 输出流
    JarOutputStream out = new JarOutputStream(new FileOutputStream(fileName));
    // 对映射中的每个条目, 保存字节
    for (String entry : outBytes.keySet()) {
        // 将类名追加到类条目中
        String ext = entry.contains(".") ? "" : ".class";
        out.putNextEntry(new ZipEntry(entry + ext));
        out.write(outBytes.get(entry));
    }
    out.closeEntry();
}
out.close();
} catch (IOException e) {
e.printStackTrace();
}
}

```

就是这样。所有更改将保存到 "sample-edit.jar" 文件中。

第170.3节：如何将 ClassNode 加载为 Class

```

/**
 * 从 ClassNode 加载类
 *
 * @param cn
 *          要加载的 ClassNode
 * @return
 */
public static Class<?> load(ClassNode cn) {
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
    return new ClassDefiner(ClassLoader.getSystemClassLoader()).get(cn.name.replace("/", "."),
cw.toByteArray());
}

/**
 * 从字节加载类的类加载器。
 */
static class ClassDefiner extends ClassLoader {
    public ClassDefiner(ClassLoader parent) {
        super(parent);
    }
}

```

```

try {
    String name = entry.getName();
    if (!name.endsWith(".class") && !entry.isDirectory()) {
        // Apache Commons - byte[] toByteArray(InputStream input)
        //
        // Add each entry to the map <Entry name , Entry bytes>
        byte[] bytes = IOUtils.toByteArray(jis);
        entries.put(name, bytes);
    }
} catch (Exception e) {
e.printStackTrace();
} finally {
jis.closeEntry();
}
jis.close();
return entries;
}

static void saveAsJar(Map<String, byte[]> outBytes, String fileName) {
try {
    // Create jar output stream
    JarOutputStream out = new JarOutputStream(new FileOutputStream(fileName));
    // For each entry in the map, save the bytes
    for (String entry : outBytes.keySet()) {
        // Append class names to class entries
        String ext = entry.contains(".") ? "" : ".class";
        out.putNextEntry(new ZipEntry(entry + ext));
        out.write(outBytes.get(entry));
        out.closeEntry();
    }
    out.close();
} catch (IOException e) {
e.printStackTrace();
}
}

```

That's it. All the changes will be saved to "sample-edit.jar".

Section 170.3: How to load a ClassNode as a Class

```

/**
 * Load a class by from a ClassNode
 *
 * @param cn
 *          ClassNode to load
 * @return
 */
public static Class<?> load(ClassNode cn) {
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
    return new ClassDefiner(ClassLoader.getSystemClassLoader()).get(cn.name.replace("/", "."),
cw.toByteArray());
}

/**
 * Classloader that loads a class from bytes.
 */
static class ClassDefiner extends ClassLoader {
    public ClassDefiner(ClassLoader parent) {
        super(parent);
    }
}

```

```

public Class<?> get(String name, byte[] bytes) {
    Class<?> c = defineClass(name, bytes, 0, bytes.length);
    resolveClass(c);
    return c;
}

```

第170.4节：如何重命名jar文件中的类

```

public static void main(String[] args) throws Exception {
    File jarFile = new File("Input.jar");
    Map<String, ClassNode> nodes = JarUtils.loadClasses(jarFile);

    Map<String, byte[]> out = JarUtils.loadNonClassEntries(jarFile);
    Map<String, String> mappings = new HashMap<String, String>();
    mappings.put("me/example/ExampleClass", "me/example/ExampleRenamed");
    out.putAll(process(nodes, mappings));
    JarUtils.saveAsJar(out, "Input-new.jar");
}

static Map<String, byte[]> process(Map<String, ClassNode> nodes, Map<String, String> mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    Remapper mapper = new SimpleRemapper(mappings);
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        ClassVisitor remapper = new ClassRemapper(cw, mapper);
        cn.accept(remapper);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name) : cn.name, cw.toByteArray());
    }
    return out;
}

```

SimpleRemapper 是 ASM 库中已有的一个类。但它只允许更改类名。如果你想重命名字段和方法，应该自己实现 Remapper 类。

第 170.5 节：Javassist 基础

Javassist 是一个字节码插桩库，允许你修改字节码，注入 Java 代码，这些代码会被 Javassist 转换成字节码，并在运行时添加到被插桩的类/方法中。

让我们编写第一个转换器，实际操作一个假设的类 "com.my.to.be.instrumented.MyClass"，并在每个方法的指令中添加一个日志调用。

```

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;

public class DynamicTransformer implements ClassFileTransformer {

    public byte[] transform(ClassLoader loader, String className, Class classBeingRedefined,
                           ProtectionDomain protectionDomain, byte[] classfileBuffer) throws
    IllegalClassFormatException {

        byte[] byteCode = classfileBuffer;
        // 进入转换器的将是每个加载的类，因此我们进行过滤
    }
}

```

```

public Class<?> get(String name, byte[] bytes) {
    Class<?> c = defineClass(name, bytes, 0, bytes.length);
    resolveClass(c);
    return c;
}

```

Section 170.4: How to rename classes in a jar file

```

public static void main(String[] args) throws Exception {
    File jarFile = new File("Input.jar");
    Map<String, ClassNode> nodes = JarUtils.loadClasses(jarFile);

    Map<String, byte[]> out = JarUtils.loadNonClassEntries(jarFile);
    Map<String, String> mappings = new HashMap<String, String>();
    mappings.put("me/example/ExampleClass", "me/example/ExampleRenamed");
    out.putAll(process(nodes, mappings));
    JarUtils.saveAsJar(out, "Input-new.jar");
}

static Map<String, byte[]> process(Map<String, ClassNode> nodes, Map<String, String> mappings) {
    Map<String, byte[]> out = new HashMap<String, byte[]>();
    Remapper mapper = new SimpleRemapper(mappings);
    for (ClassNode cn : nodes.values()) {
        ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
        ClassVisitor remapper = new ClassRemapper(cw, mapper);
        cn.accept(remapper);
        out.put(mappings.containsKey(cn.name) ? mappings.get(cn.name) : cn.name, cw.toByteArray());
    }
    return out;
}

```

SimpleRemapper is an existing class in the ASM library. However it only allows for class names to be changed. If you wish to rename fields and methods you should create your own implementation of the Remapper class.

Section 170.5: Javassist Basic

Javassist is a bytecode instrumentation library that allows you to modify bytecode injecting Java code that will be converted to bytecode by Javassist and added to the instrumented class/method at runtime.

Lets write the first transformer that actually take an hypothetical class "com.my.to.be.instrumented.MyClass" and add to the instructions of each method a log call.

```

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
import javassist.ClassPool;
import javassist.CtClass;
import javassist.CtMethod;

public class DynamicTransformer implements ClassFileTransformer {

    public byte[] transform(ClassLoader loader, String className, Class classBeingRedefined,
                           ProtectionDomain protectionDomain, byte[] classfileBuffer) throws
    IllegalClassFormatException {

        byte[] byteCode = classfileBuffer;
        // into the transformer will arrive every class loaded so we filter
    }
}

```

```

// 以匹配我们需要的内容
if (className.equals("com.my.to/be/instrumented/MyClass")) {

    try {
        // 获取默认的Javassist类池
        ClassPool cp = ClassPool.getDefault();
        // 从类池中获取具有此限定名的类
        CtClass cc = cp.get("com.my.to.be.instrumented.MyClass");
        // 获取检索到的类的所有方法
        CtMethod[] methods = cc.getDeclaredMethods()
        for(CtMethod meth : methods) {
            // 要返回并注入的插装代码
            final StringBuffer buffer = new StringBuffer();
            String name = meth.getName();
            // 仅在缓冲区中打印一个日志示例
            buffer.append("System.out.println(\"方法 " + name + " 执行了\");");
            meth.insertBefore(buffer.toString())
        }
        // 创建类的字节码
        byteCode = cc.toBytecode();
        // 从类池中移除CtClass
        cc.detach();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

return byteCode;
}

```

现在为了使用这个转换器（使我们的JVM在加载每个类时调用transform方法），我们需要通过一个代理将这个instrumentor添加进去：

```

import java.lang.instrument.Instrumentation;

public class EasyAgent {

    public static void premain(String agentArgs, Instrumentation inst) {

        // 注册转换器
        inst.addTransformer(new DynamicTransformer());
    }
}

```

启动我们第一个instrumentor实验的最后一步是将这个代理类实际注册到VM机器执行中。最简单的方式是通过shell命令的一个选项来注册它：

```
java -javaagent:myAgent.jar MyJavaApplication
```

正如我们所见，agent/transformer项目作为一个jar包被添加到任何名为MyJavaApplication的应用程序执行中，该应用程序应包含一个名为“com.my.to.be.instrumented.MyClass”的类，以实际执行我们注入的代码。

```

// to match only what we need
if (className.equals("com.my.to/be/instrumented/MyClass")) {

    try {
        // retrieve default Javassist class pool
        ClassPool cp = ClassPool.getDefault();
        // get from the class pool our class with this qualified name
        CtClass cc = cp.get("com.my.to.be.instrumented.MyClass");
        // get all the methods of the retrieved class
        CtMethod[] methods = cc.getDeclaredMethods()
        for(CtMethod meth : methods) {
            // The instrumentation code to be returned and injected
            final StringBuffer buffer = new StringBuffer();
            String name = meth.getName();
            // just print into the buffer a log for example
            buffer.append("System.out.println(\"Method " + name + " executed\");");
            meth.insertBefore(buffer.toString())
        }
        // create the bytecode of the class
        byteCode = cc.toBytecode();
        // remove the CtClass from the ClassPool
        cc.detach();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

return byteCode;
}

```

Now in order to use this transformer (so that our JVM will call the method transform on each class at load time) we need to add this instrumentor this with an agent:

```

import java.lang.instrument.Instrumentation;

public class EasyAgent {

    public static void premain(String agentArgs, Instrumentation inst) {

        // registers the transformer
        inst.addTransformer(new DynamicTransformer());
    }
}

```

Last step to start our first instrumentor experiment is to actually register this agent class to the JVM machine execution. The easiest way to actually do it is to register it with an option into the shell command:

```
java -javaagent:myAgent.jar MyJavaApplication
```

As we can see the agent/transformer project is added as a jar to the execution of any application named MyJavaApplication that is supposed to contain a class named "com.my.to.be.instrumented.MyClass" to actually execute our injected code.

名称	描述
<classes>	要反汇编的类列表。格式可以是package1.package2.Classname，或者package1/package2/Classname格式。请勿包含.class扩展名。
-help, --help, -?	打印此使用说明
-version	版本信息
-v, -verbose	打印附加信息
-l	打印行号和局部变量表
-公共	仅显示公共类和成员
-受保护	显示受保护/公共类和成员
-包	显示包/受保护/公共类和成员（默认）
-p, -私有	显示所有类和成员
-c	拆解代码
-s	打印内部类型签名
-sysinfo	显示正在处理的类的系统信息（路径、大小、日期、MD5哈希）
-constants	显示最终常量
-classpath <path>	指定用户类文件的位置
-cp <path>	指定用户类文件的位置
-bootclasspath <path>	覆盖引导类文件的位置

第171.1节：使用javap查看字节码

如果你想查看Java程序生成的字节码，可以使用提供的javap命令来查看。

假设我们有以下Java源文件：

```
包com.stackoverflow.documentation;
导入org.springframework.stereotype.Service;
导入java.io.IOException;
导入java.io.InputStream;
导入java.util.List;

@Service
公共类HelloWorldService{

    公共void sayHello() {
        System.out.println("Hello, World!");
    }

    私有Object[] pvtMethod(List<String> strings) {
        返回 新的Object[]{strings};
    }

    受保护的String tryCatchResources(String filename) 抛出IOException {
        尝试 (InputStream inputStream = getClass().getResourceAsStream(filename)) {
            字节[] bytes = 新的字节[8192];
            int read = inputStream.read(bytes);
            return new String(bytes, 0, read);
        } catch (IOException | RuntimeException e) {
    }
}
```

Chapter 171: Disassembling and Decompiling

Name	Description
<classes>	List of classes to disassemble. Can be in either package1.package2.Classname format, or package1/package2/Classname format. Do not include the .class extension.
-help, --help, -?	Print this usage message
-version	Version information
-v, -verbose	Print additional information
-l	Print line number and local variable tables
-public	Show only public classes and members
-protected	Show protected/public classes and members
-package	Show package/protected/public classes and members (default)
-p, -private	Show all classes and members
-c	Disassemble the code
-s	Print internal type signatures
-sysinfo	Show system info (path, size, date, MD5 hash) of class being processed
-constants	Show final constants
-classpath <path>	Specify where to find user class files
-cp <path>	Specify where to find user class files
-bootclasspath <path>	Override location of bootstrap class files

Section 171.1: Viewing bytecode with javap

If you want to see the generated bytecode for a Java program, you can use the provided javap command to view it.

Assuming that we have the following Java source file:

```
package com.stackoverflow.documentation;

import org.springframework.stereotype.Service;
import java.io.IOException;
import java.io.InputStream;
import java.util.List;

@Service
public class HelloWorldService {

    public void sayHello() {
        System.out.println("Hello, World!");
    }

    private Object[] pvtMethod(List<String> strings) {
        return new Object[]{strings};
    }

    protected String tryCatchResources(String filename) throws IOException {
        try (InputStream inputStream = getClass().getResourceAsStream(filename)) {
            byte[] bytes = new byte[8192];
            int read = inputStream.read(bytes);
            return new String(bytes, 0, read);
        } catch (IOException | RuntimeException e) {
    }
}
```

```

e.printStackTrace();
    throw e;
}

void stuff() {
    System.out.println("stuff");
}
}

```

编译源文件后，最简单的用法是：

```

cd <包含类的目录> (例如 target/classes)
javap com/stackoverflow/documentation/SpringExample

```

这将产生如下输出

```

Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService {
public com.stackoverflow.documentation.HelloWorldService();
public void sayHello();
protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
void stuff();
}

```

这列出了类中所有非私有方法，但这对大多数用途来说并不是特别有用。以下命令更有用：

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

输出结果为：

```

Classfile /Users/pivotal/IdeaProjects/stackoverflow-spring-
docs/target/classes/com/stackoverflow/documentation/HelloWorldService.class
最后修改时间 2016年7月22日；大小 2167 字节
MD5 校验和 6e33b5c292ead21701906353b7f06330
编译自 "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService
次版本号: 0
主版本号: 51
标志: ACC_PUBLIC, ACC_SUPER
常量池:
#1 = Methodref      #5.#60      // java/lang/Object."":()V
#2 = Fieldref       #61.#62      // java/lang/System.out:Ljava/io/PrintStream;
#3 = String          #63          // Hello, World!
#4 = Methodref       #64.#65      // java/io/PrintStream.println:(Ljava/lang/String;)V
#5 = Class           #66          // java/lang/Object
#6 = Methodref       #5.#67      // java/lang/Object.getClass:()Ljava/lang/Class;
#7 = Methodref       #68.#69      //
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
#8 = Methodref       #70.#71      // java/io/InputStream.read:([B)I
#9 = Class           #72          // java/lang/String
#10 = Methodref      #9.#73      // java/lang/String."":([BII)V
#11 = Methodref      #70.#74      // java/io/InputStream.close:()V
#12 = Class           #75          // java/lang/Throwable
#13 = Methodref      #12.#76     //
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
#14 = Class           #77          // java/io/IOException
#15 = Class           #78          // java/lang/RuntimeException

```

```

e.printStackTrace();
    throw e;
}

void stuff() {
    System.out.println("stuff");
}
}

```

After compiling the source file, the most simple usage is:

```

cd <directory containing classes> (e.g. target/classes)
javap com/stackoverflow/documentation/SpringExample

```

Which produces the output

```

Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService {
public com.stackoverflow.documentation.HelloWorldService();
public void sayHello();
protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
void stuff();
}

```

This lists all non-private methods in the class, but that is not particularly useful for most purposes. The following command is a lot more useful:

```
javap -p -c -s -constants -l -v com/stackoverflow/documentation/HelloWorldService
```

Which produces the output:

```

Classfile /Users/pivotal/IdeaProjects/stackoverflow-spring-
docs/target/classes/com/stackoverflow/documentation/HelloWorldService.class
Last modified Jul 22, 2016; size 2167 bytes
MD5 checksum 6e33b5c292ead21701906353b7f06330
Compiled from "HelloWorldService.java"
public class com.stackoverflow.documentation.HelloWorldService
minor version: 0
major version: 51
flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
#1 = Methodref      #5.#60      // java/lang/Object."":()V
#2 = Fieldref       #61.#62      // java/lang/System.out:Ljava/io/PrintStream;
#3 = String          #63          // Hello, World!
#4 = Methodref       #64.#65      // java/io/PrintStream.println:(Ljava/lang/String;)V
#5 = Class           #66          // java/lang/Object
#6 = Methodref       #5.#67      // java/lang/Object.getClass:()Ljava/lang/Class;
#7 = Methodref       #68.#69      //
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
#8 = Methodref       #70.#71      // java/io/InputStream.read:([B)I
#9 = Class           #72          // java/lang/String
#10 = Methodref      #9.#73      // java/lang/String."":([BII)V
#11 = Methodref      #70.#74      // java/io/InputStream.close:()V
#12 = Class           #75          // java/lang/Throwable
#13 = Methodref      #12.#76     //
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
#14 = Class           #77          // java/io/IOException
#15 = Class           #78          // java/lang/RuntimeException

```

```

#16 = Methodref      #79.#80      // java/lang/Exception.printStackTrace():V
#17 = String         #55          // stuff
#18 = Class          #81          // com/stackoverflow/documentation/HelloWorldService
#19 = Utf8
#20 = Utf8           ()V
#21 = Utf8           Code
#22 = Utf8           LineNumberTable
#23 = Utf8           LocalVariableTable
#24 = Utf8           this
#25 = Utf8           Lcom/stackoverflow/documentation/HelloWorldService;
#26 = Utf8           sayHello
#27 = Utf8           pvtMethod
#28 = Utf8           (Ljava/util/List;)[Ljava/lang/Object;
#29 = Utf8           strings
#30 = Utf8           Ljava/util/List;
#31 = Utf8           LocalVariableTypeTable
#32 = Utf8           Ljava/util/List;
#33 = Utf8           签名
#34 = Utf8           (Ljava/util/List;)[Ljava/lang/Object;
#35 = Utf8           tryCatchResources
#36 = Utf8           (Ljava/lang/String;)Ljava/lang/String;
#37 = Utf8           字节
#38 = Utf8           [B
#39 = Utf8           读取
#40 = Utf8           I
#41 = Utf8           输入流
#42 = Utf8           Ljava/io/InputStream;
#43 = Utf8           e
#44 = Utf8           Ljava/lang/Exception;
#45 = Utf8           文件名
#46 = Utf8           Ljava/lang/String;
#47 = Utf8           StackMapTable
#48 = Class          #81          // com/stackoverflow/documentation/HelloWorldService
#49 = Class          #72          // java/lang/String
#50 = Class          #82          // java/io/InputStream
#51 = Class          #75          // java/lang/Throwable
#52 = Class          #38          // "[B"
#53 = Class          #83          // java/lang/Exception
#54 = Utf8           异常
#55 = Utf8           物品
#56 = Utf8           源文件
#57 = Utf8           HelloWorldService.java
#58 = Utf8           RuntimeVisibleAnnotations
#59 = Utf8           Lorg/springframework/stereotype/Service;
#60 = NameAndType    #19:#20     // ":"()
#61 = 类             #84          // java/lang/System
#62 = 名称和类型    #85:#86     // out:Ljava/io/PrintStream;
#63 = Utf8           Hello, World!
#64 = 类             #87          // java/io/PrintStream
#65 = 名称和类型    #88:#89     // println:(Ljava/lang/String;)V
#66 = Utf8           java/lang/Object
#67 = 名称和类型    #90:#91     // getClass():Ljava/lang/Class;
#68 = 类             #92          // java/lang/Class
#69 = 名称和类型    #93:#94     //
getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
#70 = 类             #82          // java/io/InputStream
#71 = 名称和类型    #39:#95     // read:([B)I
#72 = Utf8           java/lang/String
#73 = 名称和类型    #19:#96     // "":(BII)V
#74 = 名称和类型    #97:#20     // close:()V
#75 = Utf8           java/lang/Throwable
#76 = 名称和类型    #98:#99     // addSuppressed:(Ljava/lang/Throwable;)V
#77 = Utf8           java/io/IOException
#78 = Utf8           java/lang/RuntimeException

```

```

#16 = Methodref      #79.#80      // java/lang/Exception.printStackTrace():V
#17 = String          #55          // stuff
#18 = Class          #81          // com/stackoverflow/documentation/HelloWorldService
#19 = Utf8
#20 = Utf8           ()V
#21 = Utf8           Code
#22 = Utf8           LineNumberTable
#23 = Utf8           LocalVariableTable
#24 = Utf8           this
#25 = Utf8           Lcom/stackoverflow/documentation/HelloWorldService;
#26 = Utf8           sayHello
#27 = Utf8           pvtMethod
#28 = Utf8           (Ljava/util/List;)[Ljava/lang/Object;
#29 = Utf8           strings
#30 = Utf8           Ljava/util/List;
#31 = Utf8           LocalVariableTypeTable
#32 = Utf8           Ljava/util/List;
#33 = Utf8           Signature
#34 = Utf8           (Ljava/util/List;)[Ljava/lang/Object;
#35 = Utf8           tryCatchResources
#36 = Utf8           (Ljava/lang/String;)Ljava/lang/String;
#37 = Utf8           bytes
#38 = Utf8           [B
#39 = Utf8           read
#40 = Utf8           I
#41 = Utf8           inputStream
#42 = Utf8           Ljava/io/InputStream;
#43 = Utf8           e
#44 = Utf8           Ljava/lang/Exception;
#45 = Utf8           filename
#46 = Utf8           Ljava/lang/String;
#47 = Utf8           StackMapTable
#48 = Class          #81          // com/stackoverflow/documentation/HelloWorldService
#49 = Class          #72          // java/lang/String
#50 = Class          #82          // java/io/InputStream
#51 = Class          #75          // java/lang/Throwable
#52 = Class          #38          // "[B"
#53 = Class          #83          // java/lang/Exception
#54 = Utf8           Exceptions
#55 = Utf8           stuff
#56 = Utf8           SourceFile
#57 = Utf8           HelloWorldService.java
#58 = Utf8           RuntimeVisibleAnnotations
#59 = Utf8           Lorg/springframework/stereotype/Service;
#60 = NameAndType    #19:#20     // ":"()
#61 = Class          #84          // java/lang/System
#62 = NameAndType    #85:#86     // out:Ljava/io/PrintStream;
#63 = Utf8           Hello, World!
#64 = Class          #87          // java/io/PrintStream
#65 = NameAndType    #88:#89     // println:(Ljava/lang/String;)V
#66 = Utf8           java/lang/Object
#67 = NameAndType    #90:#91     // getClass():Ljava/lang/Class;
#68 = Class          #92          // java/lang/Class
#69 = NameAndType    #93:#94     //
getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
#70 = Class          #82          // java/io/InputStream
#71 = NameAndType    #39:#95     // read:([B)I
#72 = Utf8           java/lang/String
#73 = NameAndType    #19:#96     // "":(BII)V
#74 = NameAndType    #97:#20     // close:()V
#75 = Utf8           java/lang/Throwable
#76 = NameAndType    #98:#99     // addSuppressed:(Ljava/lang/Throwable;)V
#77 = Utf8           java/io/IOException
#78 = Utf8           java/lang/RuntimeException

```

```

#79 = 类          #83      // java/lang/Exception
#80 = 名称和类型  #100:#20   // printStackTrace():V
#81 = Utf8        com/stackoverflow/documentation/HelloWorldService
#82 = Utf8        java/io/InputStream
#83 = Utf8        java/lang/Exception
#84 = Utf8        java/lang/System
#85 = Utf8        out
#86 = Utf8        Ljava/io/PrintStream;
#87 = Utf8        java/io/PrintStream
#88 = Utf8        println
#89 = Utf8        (Ljava/lang/String;)V
#90 = Utf8        getClass
#91 = Utf8        ()Ljava/lang/Class;
#92 = Utf8        java/lang/Class
#93 = Utf8        getResourceAsStream
#94 = Utf8        (Ljava/lang/String;)Ljava/io/InputStream;
#95 = Utf8        ([B)I
#96 = Utf8        ([BII)V
#97 = Utf8        close
#98 = Utf8        addSuppressed
#99 = Utf8        (Ljava/lang/Throwable;)V
#100 = Utf8       printStackTrace
{
public com.stackoverflow.documentation.HelloWorldService();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: invokespecial #1           // Method java/lang/Object."":()V
4: return
LineNumberTable:
line 10: 0
LocalVariableTable:
Start  Length  Slot  Name   Signature
0      5       0     this   Lcom/stackoverflow/documentation/HelloWorldService;

public void sayHello();
描述符: ()V
标志: ACC_PUBLIC
代码:
栈=2, 局部变量=1, 参数大小=1
0: getstatic    #2           // 字段 java/lang/System.out:Ljava/io/PrintStream;
3: ldc         #3           // 字符串 Hello, World!
5: invokevirtual #4          // 方法 java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
LineNumberTable:
第13行: 0
第14行: 8
局部变量表:
Start  Length  Slot  Name   Signature
0      9       0     this   Lcom/stackoverflow/documentation/HelloWorldService;

private java.lang.Object[] pvtMethod(java.util.List);
描述符: (Ljava/util/List;)[Ljava/lang/Object;
标志: ACC_PRIVATE
代码:
栈=4, 局部变量=2, 参数大小=2
0: iconst_1
1: anewarray   #5           // 类 java/lang/Object
4: dup
5: iconst_0
6: aload_1
7: aastore

```

```

#79 = Class        #83      // java/lang/Exception
#80 = NameAndType #100:#20   // printStackTrace():V
#81 = Utf8         com/stackoverflow/documentation/HelloWorldService
#82 = Utf8         java/io/InputStream
#83 = Utf8         java/lang/Exception
#84 = Utf8         java/lang/System
#85 = Utf8         out
#86 = Utf8         Ljava/io/PrintStream;
#87 = Utf8         java/io/PrintStream
#88 = Utf8         println
#89 = Utf8         (Ljava/lang/String;)V
#90 = Utf8         getClass
#91 = Utf8         ()Ljava/lang/Class;
#92 = Utf8         java/lang/Class
#93 = Utf8         getResourceAsStream
#94 = Utf8         (Ljava/lang/String;)Ljava/io/InputStream;
#95 = Utf8         ([B)I
#96 = Utf8         ([BII)V
#97 = Utf8         close
#98 = Utf8         addSuppressed
#99 = Utf8         (Ljava/lang/Throwable;)V
#100 = Utf8        printStackTrace
{
public com.stackoverflow.documentation.HelloWorldService();
descriptor: ()V
flags: ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: invokespecial #1           // Method java/lang/Object."":()V
4: return
LineNumberTable:
line 10: 0
LocalVariableTable:
Start  Length  Slot  Name   Signature
0      5       0     this   Lcom/stackoverflow/documentation/HelloWorldService;

public void sayHello();
descriptor: ()V
flags: ACC_PUBLIC
Code:
栈=2, 局部变量=1, 参数大小=1
0: getstatic    #2           // Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc         #3           // String Hello, World!
5: invokevirtual #4          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
LineNumberTable:
line 13: 0
line 14: 8
LocalVariableTable:
Start  Length  Slot  Name   Signature
0      9       0     this   Lcom/stackoverflow/documentation/HelloWorldService;

private java.lang.Object[] pvtMethod(java.util.List);
descriptor: (Ljava/util/List;)[Ljava/lang/Object;
flags: ACC_PRIVATE
Code:
stack=4, locals=2, args_size=2
0: iconst_1
1: anewarray   #5           // class java/lang/Object
4: dup
5: iconst_0
6: aload_1
7: aastore

```

```

8: areturn
LineNumberTable:
line 17: 0
LocalVariableTable:
Start Length Slot Name Signature
0 9 0 this Lcom/stackoverflow/documentation/HelloWorldService;
0 9 1 strings Ljava/util/List;
LocalVariableTypeTable:
Start Length Slot Name Signature
0 9 1 strings Ljava/util/List;
Signature: #34 // (Ljava/util/List;)[Ljava/lang/Object;

protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
descriptor: (Ljava/lang/String;)Ljava/lang/String;
flags: ACC_PROTECTED
Code:
stack=5, locals=10, args_size=2
0: aload_0
1: invokevirtual #6 // Method java/lang/Object.getClass:()Ljava/lang/Class;
4: aload_1
5: invokevirtual #7 // 方法
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
8: astore_2
9: aconst_null
10: astore_3
11: sipush 8192
14: newarray byte
16: astore 4
18: aload_2
19: aload 4
21: invokevirtual #8 // 方法 java/io/InputStream.read:([B)I
24: istore 5
26: new #9 // 类 java/lang/String
29: dup
30: aload 4
32: iconst_0
33: iload 5
35: invokespecial #10 // 方法 java/lang/String."":([BII)V
38: astore 6
40: aload_2
41: ifnull 70
44: aload_3
45: ifnull 66
48: aload_2
49: invokevirtual #11 // 方法 java/io/InputStream.close:()V
52: goto 70
55: astore 7
57: aload_3
58: aload 7
60: invokevirtual #13 // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
63: goto 70
66: aload_2
67: invokevirtual #11 // Method java/io/InputStream.close:()V
70: aload 6
72: areturn
73: astore 4
75: aload 4
77: astore_3
78: aload 4
80: athrow
81: astore 8
83: aload_2
84: ifnull 113

```

```

8: areturn
LineNumberTable:
line 17: 0
LocalVariableTable:
Start Length Slot Name Signature
0 9 0 this Lcom/stackoverflow/documentation/HelloWorldService;
0 9 1 strings Ljava/util/List;
LocalVariableTypeTable:
Start Length Slot Name Signature
0 9 1 strings Ljava/util/List;
Signature: #34 // (Ljava/util/List;)[Ljava/lang/Object;

protected java.lang.String tryCatchResources(java.lang.String) throws java.io.IOException;
descriptor: (Ljava/lang/String;)Ljava/lang/String;
flags: ACC_PROTECTED
Code:
stack=5, locals=10, args_size=2
0: aload_0
1: invokevirtual #6 // Method java/lang/Object.getClass:()Ljava/lang/Class;
4: aload_1
5: invokevirtual #7 // Method
java/lang/Class.getResourceAsStream:(Ljava/lang/String;)Ljava/io/InputStream;
8: astore_2
9: aconst_null
10: astore_3
11: sipush 8192
14: newarray byte
16: astore 4
18: aload_2
19: aload 4
21: invokevirtual #8 // Method java/io/InputStream.read:([B)I
24: istore 5
26: new #9 // class java/lang/String
29: dup
30: aload 4
32: iconst_0
33: iload 5
35: invokespecial #10 // Method java/lang/String."":([BII)V
38: astore 6
40: aload_2
41: ifnull 70
44: aload_3
45: ifnull 66
48: aload_2
49: invokevirtual #11 // Method java/io/InputStream.close:()V
52: goto 70
55: astore 7
57: aload_3
58: aload 7
60: invokevirtual #13 // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
63: goto 70
66: aload_2
67: invokevirtual #11 // Method java/io/InputStream.close:()V
70: aload 6
72: areturn
73: astore 4
75: aload 4
77: astore_3
78: aload 4
80: athrow
81: astore 8
83: aload_2
84: ifnull 113

```

```

87: aload_3
88: ifnull      109
91: aload_2
92: invokevirtual #11           // 方法 java/io/InputStream.close:()V
95: goto       113
98: astore      9
100: aload_3
101: aload      9
103: invokevirtual #13           // 方法
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
106: goto       113
109: aload_2
110: invokevirtual #11           // 方法 java/io/InputStream.close:()V
113: aload      8
115: athrow
116: astore_2
117: aload_2
118: invokevirtual #16           // 方法 java/lang/Exception.printStackTrace:()V
121: aload_2
122: athrow
异常表：
from   to   target 类型
48    52    55  类 java/lang/Throwable
11    40    73  类 java/lang/Throwable
11    40    81  任意
91    95    98  类 java/lang/Throwable
73    83    81  任意
0     70    116  类 java/io/IOException
0     70    116  类 java/lang/RuntimeException
73   116   116  类 java/io/IOException
73   116   116  类 java/lang/RuntimeException
行号表：
第21行: 0
第22行: 11
第23行: 18
第24行: 26
第25行: 40
第21行: 73
第25行: 81
第26行: 117
第27行: 121
局部变量表：
开始 长度 槽 名称 签名
18    55   4 字节 [B
26    47   5 read  I
9    107   2 inputStream Ljava/io/InputStream;
117   6    2 e    Ljava/lang/Exception;
0    123   0 this  Lcom/stackoverflow/documentation/HelloWorldService;
0    123   1 filename Ljava/lang/String;
StackMapTable: 条目数 = 9
frame_type = 255 /* full_frame */
offset_delta = 55
locals = [ 类 com/stackoverflow/documentation/HelloWorldService, 类 java/lang/String, 类
java/io/InputStream, 类 java/lang/Throwable, 类 "[B", int, 类 java/lang/String ]
stack = [ class java/lang/Throwable ]
frame_type = 10 /* same */
frame_type = 3 /* same */
frame_type = 255 /* full_frame */
offset_delta = 2
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String, class
java/io/InputStream, class java/lang/Throwable ]
stack = [ class java/lang/Throwable ]
frame_type = 71 /* same_locals_1_stack_item */
stack = [ class java/lang/Throwable ]

```

```

87: aload_3
88: ifnull      109
91: aload_2
92: invokevirtual #11           // Method java/io/InputStream.close:()V
95: goto       113
98: astore      9
100: aload_3
101: aload      9
103: invokevirtual #13           // Method
java/lang/Throwable.addSuppressed:(Ljava/lang/Throwable;)V
106: goto       113
109: aload_2
110: invokevirtual #11           // Method java/io/InputStream.close:()V
113: aload      8
115: athrow
116: astore_2
117: aload_2
118: invokevirtual #16           // Method java/lang/Exception.printStackTrace:()V
121: aload_2
122: athrow
Exception table:
from   to   target type
48    52    55  Class java/lang/Throwable
11    40    73  Class java/lang/Throwable
11    40    81  any
91    95    98  Class java/lang/Throwable
73    83    81  any
0     70    116  Class java/io/IOException
0     70    116  Class java/lang/RuntimeException
73   116   116  Class java/io/IOException
73   116   116  Class java/lang/RuntimeException
LineNumberTable:
line 21: 0
line 22: 11
line 23: 18
line 24: 26
line 25: 40
line 21: 73
line 25: 81
line 26: 117
line 27: 121
LocalVariableTable:
Start Length Slot Name   Signature
18    55   4 bytes [B
26    47   5 read  I
9    107   2 inputStream Ljava/io/InputStream;
117   6    2 e    Ljava/lang/Exception;
0    123   0 this  Lcom/stackoverflow/documentation/HelloWorldService;
0    123   1 filename Ljava/lang/String;
StackMapTable: number_of_entries = 9
frame_type = 255 /* full_frame */
offset_delta = 55
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String, class
java/io/InputStream, class java/lang/Throwable, class "[B", int, class java/lang/String ]
stack = [ class java/lang/Throwable ]
frame_type = 10 /* same */
frame_type = 3 /* same */
frame_type = 255 /* full_frame */
offset_delta = 2
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String, class
java/io/InputStream, class java/lang/Throwable ]
stack = [ class java/lang/Throwable ]
frame_type = 71 /* same_locals_1_stack_item */
stack = [ class java/lang/Throwable ]

```

```

frame_type = 255 /* full_frame */
offset_delta = 16
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String, class
java/io/InputStream, class java/lang/Throwable, top, top, top, top, class java/lang/Throwable ]
stack = [ class java/lang/Throwable ]
frame_type = 10 /* same */
frame_type = 3 /* same */
frame_type = 255 /* full_frame */
offset_delta = 2
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String ]
stack = [ class java/lang/Exception ]
异常:
抛出 java.io.IOException

```

```

void stuff();
描述符: ()V
标志:
代码:
栈=2, 局部变量=1, 参数大小=1
0: getstatic #2           // 字段 java/lang/System.out:Ljava/io/PrintStream;
3: ldc      #17          // 字符串 stuff
5: invokevirtual #4        // 方法 java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
LineNumberTable:
第32行: 0
第33行: 8
局部变量表:
Start Length Slot Name Signature
0      9      0  this  Lcom/stackoverflow/documentation/HelloWorldService;
}

```

源文件: "HelloWorldService.java"

运行时可见注解:

0: #59()

```

frame_type = 255 /* full_frame */
offset_delta = 16
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String, class
java/io/InputStream, class java/lang/Throwable, top, top, top, top, class java/lang/Throwable ]
stack = [ class java/lang/Throwable ]
frame_type = 10 /* same */
frame_type = 3 /* same */
frame_type = 255 /* full_frame */
offset_delta = 2
locals = [ class com/stackoverflow/documentation/HelloWorldService, class java/lang/String ]
stack = [ class java/lang/Exception ]
Exceptions:
throws java.io.IOException

```

```

void stuff();
descriptor: ()V
flags:
Code:
stack=2, locals=1, args_size=1
0: getstatic #2           // Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc      #17          // String stuff
5: invokevirtual #4        // Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
LineNumberTable:
line 32: 0
line 33: 8
LocalVariableTable:
Start Length Slot Name Signature
0      9      0  this  Lcom/stackoverflow/documentation/HelloWorldService;
}
SourceFile: "HelloWorldService.java"
RuntimeVisibleAnnotations:
0: #59()

```

第172章: JMX

JMX 技术提供了构建分布式、基于 Web、模块化和动态解决方案的工具，用于管理和监控设备、应用程序以及服务驱动的网络。该标准设计适用于改造遗留系统、实现新的管理和监控解决方案，并能够接入未来的解决方案。

第172.1节：使用平台MBean服务器的简单示例

假设我们有一个服务器，用于注册新用户并向他们发送问候信息。我们想要监控这个服务器并更改它的一些参数。

首先，我们需要一个包含监控和控制方法的接口

```
public interface UserCounterMBean {  
    long getSleepTime();  
  
    void setSleepTime(long sleepTime);  
  
    int getUserCount();  
  
    void setUserCount(int userCount);  
  
    String getGreetingString();  
  
    void setGreetingString(String greetingString);  
  
    void stop();  
}
```

以及一个简单的实现，让我们可以看到它的工作情况以及我们如何影响它

```
public class UserCounter implements UserCounterMBean, Runnable {  
    private AtomicLong sleepTime = new AtomicLong(10000);  
    private AtomicInteger userCount = new AtomicInteger(0);  
    private AtomicReference<String> greetingString = new AtomicReference<>("welcome");  
    private AtomicBoolean interrupted = new AtomicBoolean(false);  
  
    @Override  
    public long getSleepTime() {  
        return sleepTime.get();  
    }  
  
    @Override  
    public void setSleepTime(long sleepTime) {  
        this.sleepTime.set(sleepTime);  
    }  
  
    @Override  
    public int getUserCount() {  
        return userCount.get();  
    }  
  
    @Override  
    public void setUserCount(int userCount) {  
        this.userCount.set(userCount);  
    }  
}
```

Chapter 172: JMX

The JMX technology provides the tools for building distributed, Web-based, modular and dynamic solutions for managing and monitoring devices, applications, and service-driven networks. By design, this standard is suitable for adapting legacy systems, implementing new management and monitoring solutions, and plugging into those of the future.

Section 172.1: Simple example with Platform MBean Server

Let's say we have some server that registers new users and greets them with some message. And we want to monitor this server and change some of its parameters.

First, we need an interface with our monitoring and control methods

```
public interface UserCounterMBean {  
    long getSleepTime();  
  
    void setSleepTime(long sleepTime);  
  
    int getUserCount();  
  
    void setUserCount(int userCount);  
  
    String getGreetingString();  
  
    void setGreetingString(String greetingString);  
  
    void stop();  
}
```

And some simple implementation that will let us see how it's working and how we affect it

```
public class UserCounter implements UserCounterMBean, Runnable {  
    private AtomicLong sleepTime = new AtomicLong(10000);  
    private AtomicInteger userCount = new AtomicInteger(0);  
    private AtomicReference<String> greetingString = new AtomicReference<>("welcome");  
    private AtomicBoolean interrupted = new AtomicBoolean(false);  
  
    @Override  
    public long getSleepTime() {  
        return sleepTime.get();  
    }  
  
    @Override  
    public void setSleepTime(long sleepTime) {  
        this.sleepTime.set(sleepTime);  
    }  
  
    @Override  
    public int getUserCount() {  
        return userCount.get();  
    }  
  
    @Override  
    public void setUserCount(int userCount) {  
        this.userCount.set(userCount);  
    }  
}
```

```

@Override
public String getGreetingString() {
    return greetingString.get();
}

@Override
public void setGreetingString(String greetingString) {
    this.greetingString.set(greetingString);
}

@Override
public void stop() {
    this.interrupted.set(true);
}

@Override
public void run() {
    while (!interrupted.get()) {
        try {
            System.out.printf("User %d, %s%n", userCount.incrementAndGet(),
greetingString.get());
            Thread.sleep(sleepTime.get());
        } catch (InterruptedException ignored) {
        }
    }
}

```

对于本地或远程管理的简单示例，我们需要注册我们的MBean：

```

import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

public class Main {
    public static void main(String[] args) throws MalformedObjectNameException,
NotCompliantMBeanException, InstanceAlreadyExistsException, MBeanRegistrationException,
InterruptedException {
        final UserCounter userCounter = new UserCounter();
        final MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
        final ObjectName objectName = new ObjectName("ServerManager:type=UserCounter");
        mBeanServer.registerMBean(userCounter, objectName);

        final Thread thread = new Thread(userCounter);
        thread.start();
        thread.join();
    }
}

```

之后我们可以运行我们的应用程序，并通过 jConsole 连接它，jConsole 位于您的 \$JAVA_HOME/bin 目录中。首先，我们需要找到运行我们应用程序的本地 java 进程

```

@Override
public String getGreetingString() {
    return greetingString.get();
}

@Override
public void setGreetingString(String greetingString) {
    this.greetingString.set(greetingString);
}

@Override
public void stop() {
    this.interrupted.set(true);
}

@Override
public void run() {
    while (!interrupted.get()) {
        try {
            System.out.printf("User %d, %s%n", userCount.incrementAndGet(),
greetingString.get());
            Thread.sleep(sleepTime.get());
        } catch (InterruptedException ignored) {
        }
    }
}

```

For simple example with local or remote management, we need to register our MBean:

```

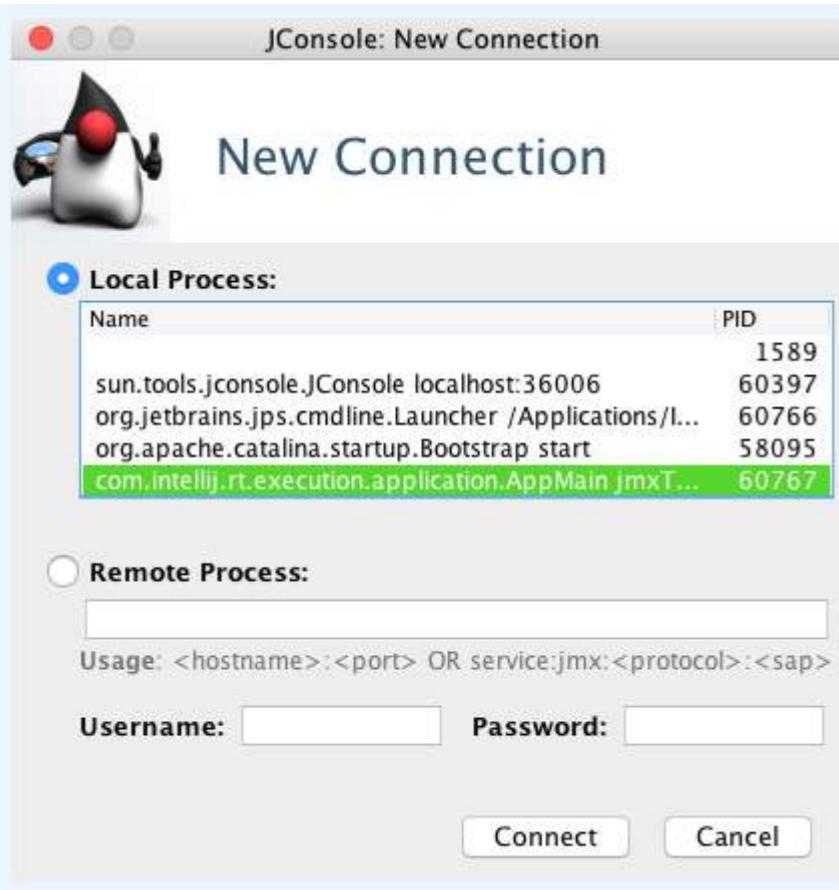
import javax.management.InstanceAlreadyExistsException;
import javax.management.MBeanRegistrationException;
import javax.management.MBeanServer;
import javax.management.MalformedObjectNameException;
import javax.management.NotCompliantMBeanException;
import javax.management.ObjectName;
import java.lang.management.ManagementFactory;

public class Main {
    public static void main(String[] args) throws MalformedObjectNameException,
NotCompliantMBeanException, InstanceAlreadyExistsException, MBeanRegistrationException,
InterruptedException {
        final UserCounter userCounter = new UserCounter();
        final MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
        final ObjectName objectName = new ObjectName("ServerManager:type=UserCounter");
        mBeanServer.registerMBean(userCounter, objectName);

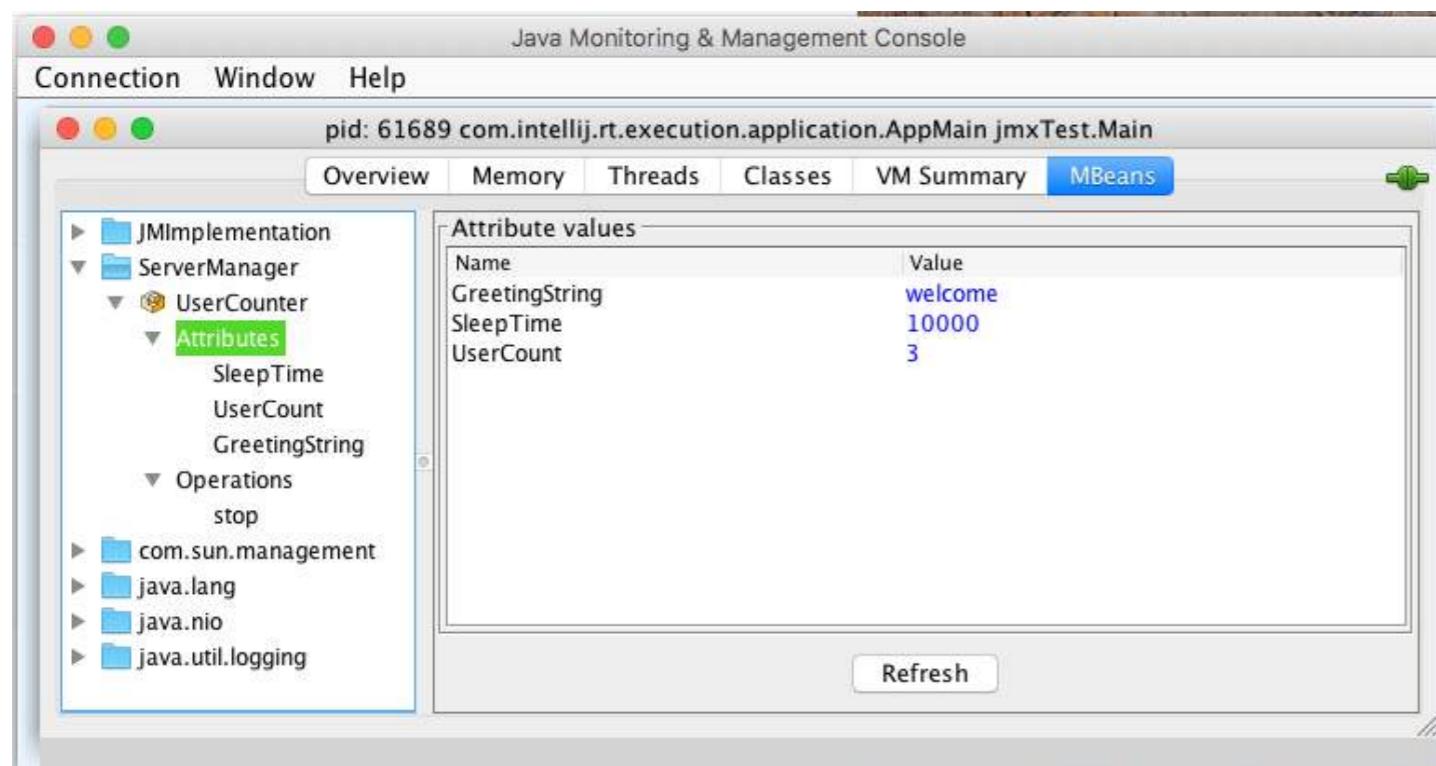
        final Thread thread = new Thread(userCounter);
        thread.start();
        thread.join();
    }
}

```

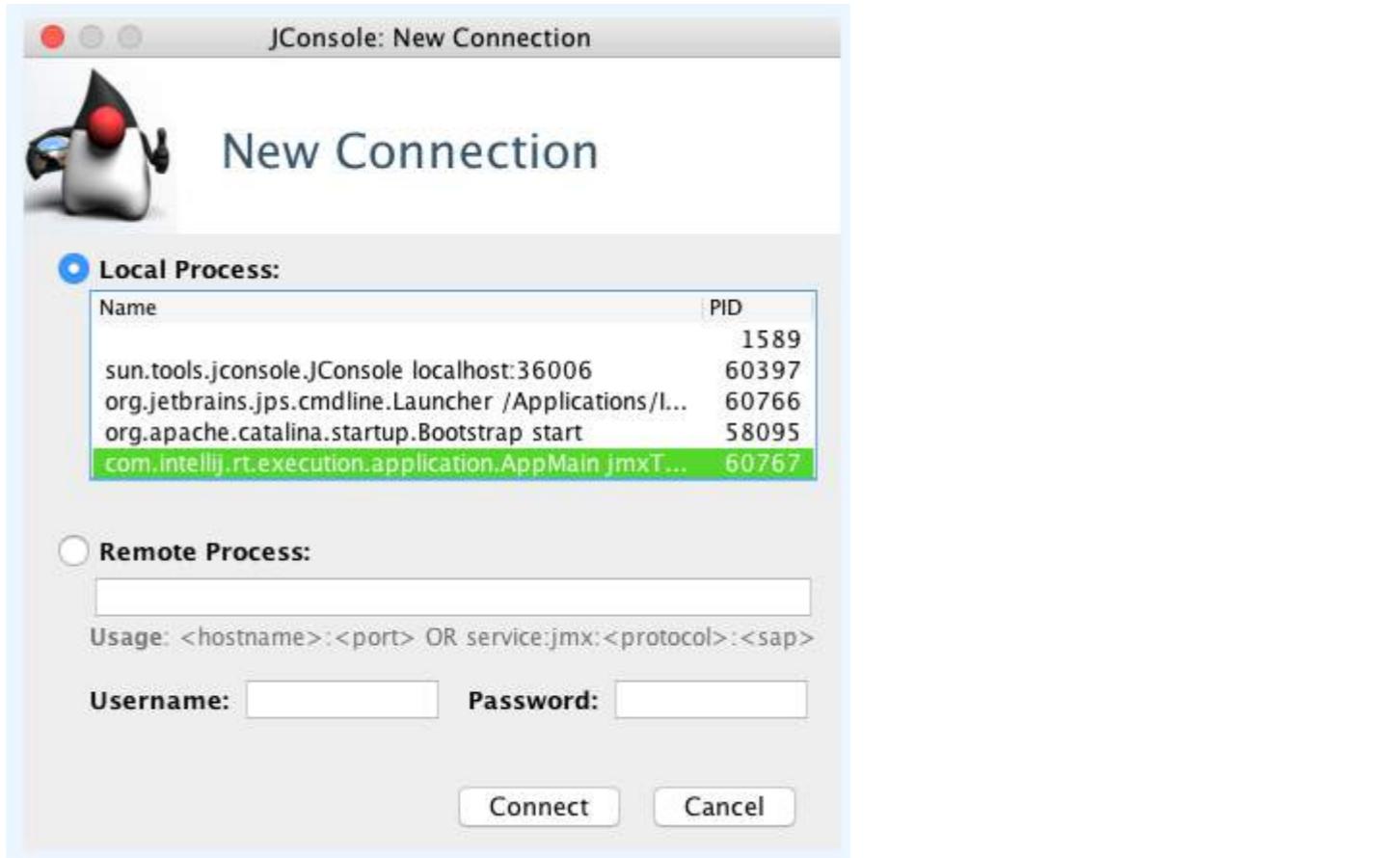
After that we can run our application and connect to it via jConsole, which can be found in your \$JAVA_HOME/bin directory. First, we need to find our local java process with our application



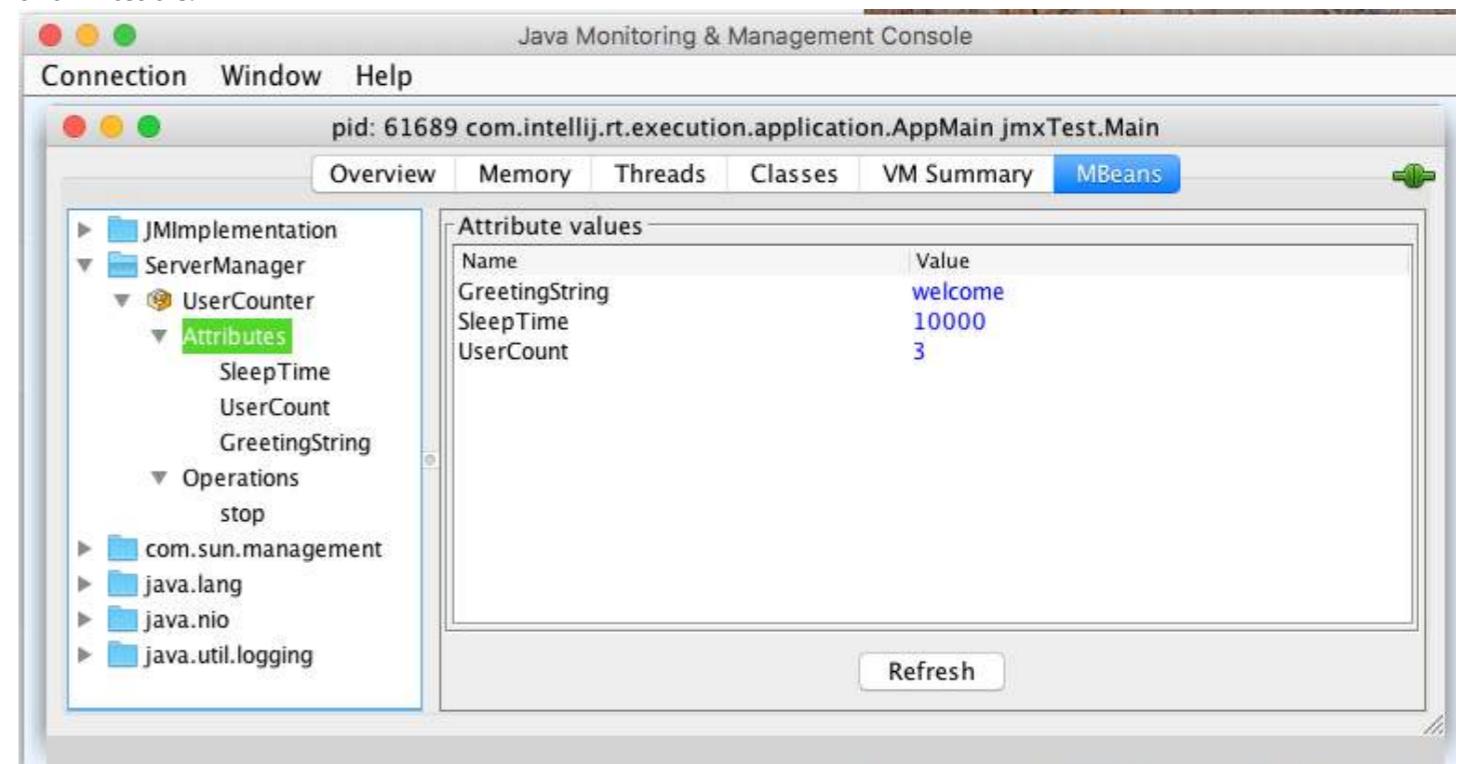
然后切换到 MBeans 选项卡，找到我们在主类中作为ObjectName使用的 MBean（在上面的示例中是ServerManager）。在Attributes部分，我们可以看到属性。如果你只指定了 get 方法，属性将是可读的但不可写的。如果你同时指定了 get 和 set 方法，属性将是可读且可写的。



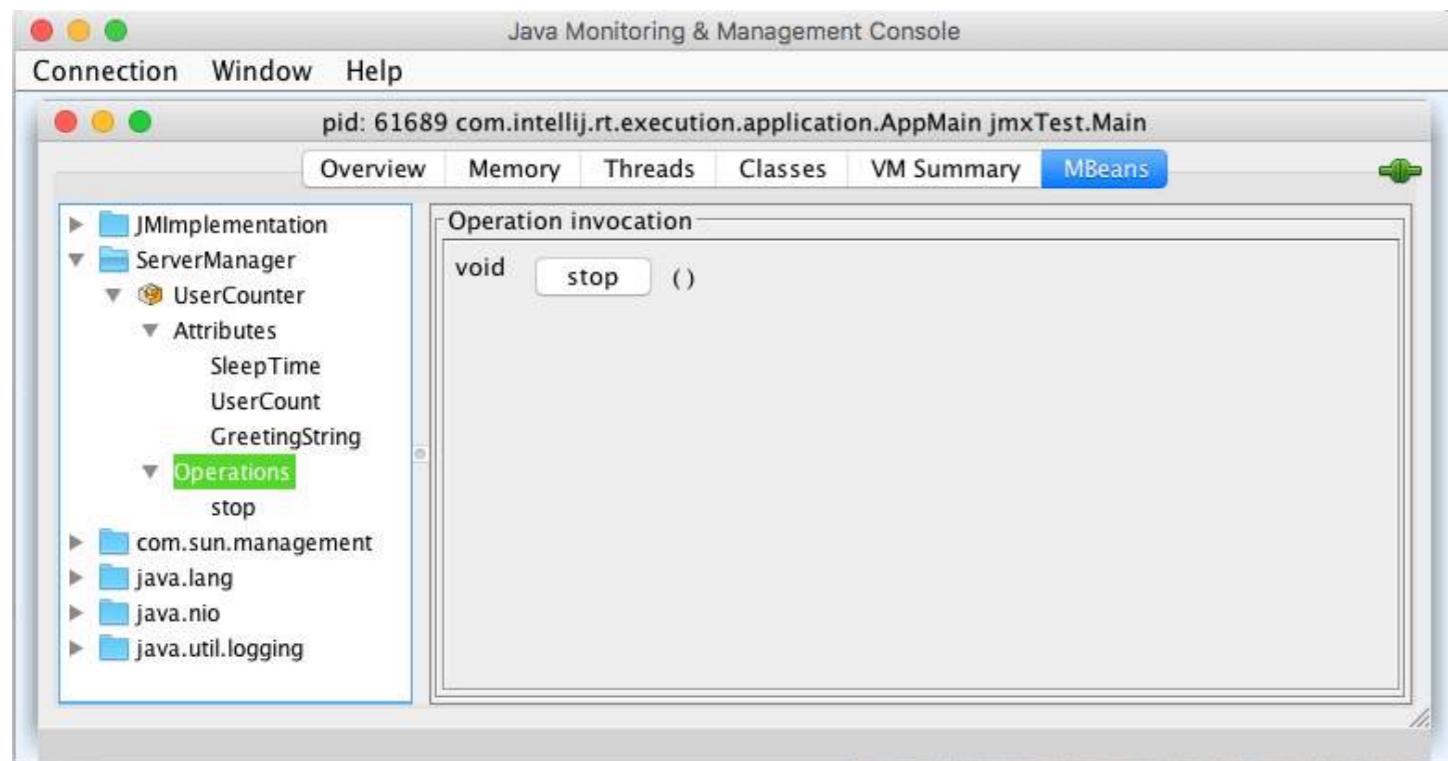
指定的方法可以在Operations部分调用。



then switch to MBeans tab and find that MBean that we used in our Main class as an ObjectName (in the example above it's ServerManager). In **Attributes** section we can see out attributes. If you specified get method only, attribute will be readable but not writeable. If you specified both get and set methods, attribute would be readable and writeable.



Specified methods can be invoked in Operations section.



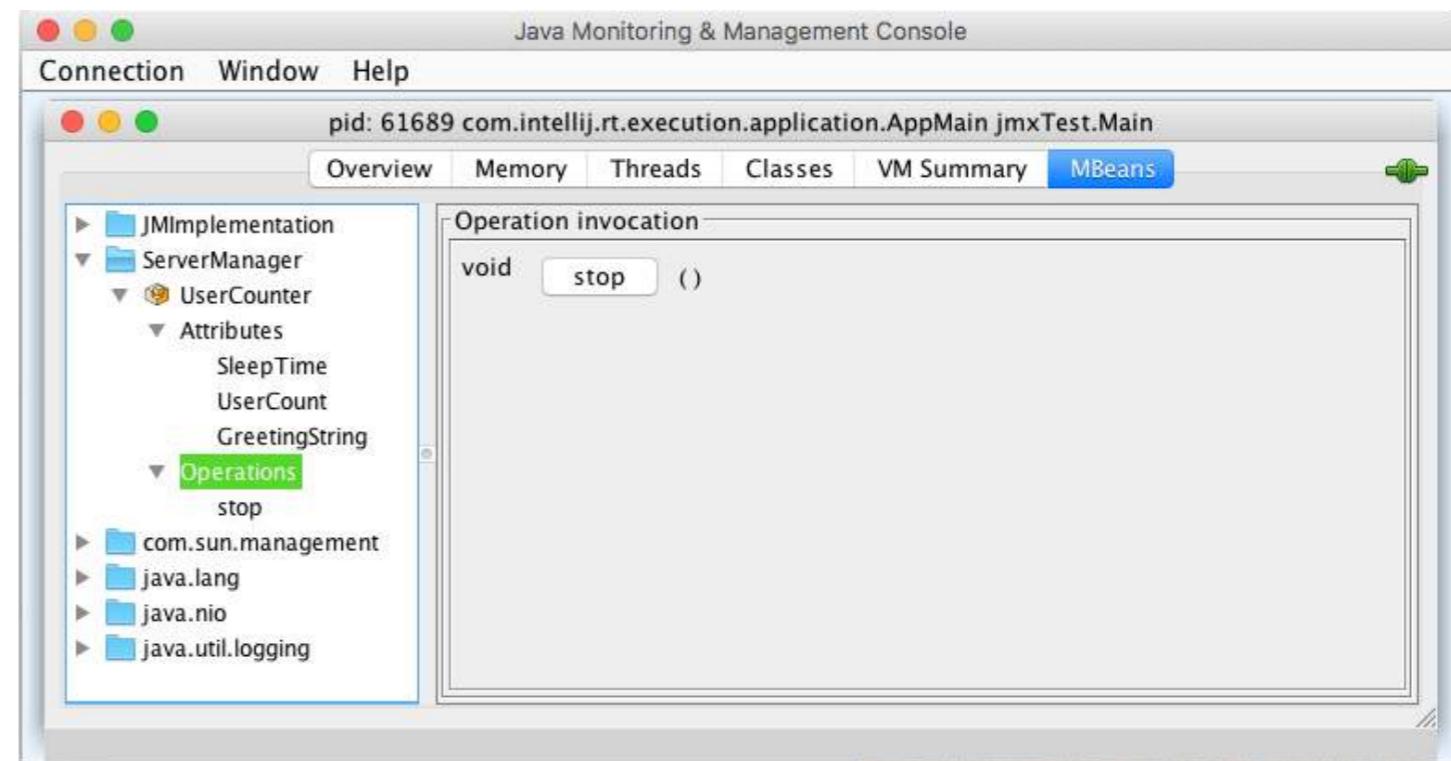
如果你想使用远程管理功能，需要额外的 JVM 参数，例如：

```
-Dcom.sun.management.jmxremote=true //默认值为 true  
-Dcom.sun.management.jmxremote.port=36006  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

这些参数可以在JMX 指南第 2 章中找到。之后，你将能够通过 jConsole 远程连接到你的应用程序，使用 jconsole host:port 或指定 host:port 连接，或者 service:jmx:rmi://jndi/rmi://hostName:portNum/jmxrmi 在 jConsole GUI 中。

有用的链接：

- [JMX 指南](#)
- [JMX 最佳实践](#)



If you want to be able to use remote management, you will need additional JVM parameters, like:

```
-Dcom.sun.management.jmxremote=true //true by default  
-Dcom.sun.management.jmxremote.port=36006  
-Dcom.sun.management.jmxremote.authenticate=false  
-Dcom.sun.management.jmxremote.ssl=false
```

These parameters can be found in [Chapter 2 of JMX guides](#). After that you will be able to connect to your application via jConsole remotely with `jconsole host:port` or with specifying `host:port` or `service:jmx:rmi://jndi/rmi://hostName:portNum/jmxrmi` in jConsole GUI.

Useful links:

- [JMX guides](#)
- [JMX Best practices](#)

第173章：Java虚拟机 (JVM)

第173.1节：基础知识

JVM 是一个抽象计算机或虚拟机，驻留在你的内存中。它具有平台无关的执行环境，将Java字节码解释为本地机器码。（Javac是Java编译器，将你的Java代码编译成字节码）

Java程序将在JVM内部运行，JVM再映射到底层的物理机器上。它是JDK中的一种编程工具。

（字节码是平台无关的代码，可在所有平台上运行，机器码是平台特定的代码，仅在特定平台如Windows或Linux上运行；这取决于执行环境。）

部分组件：

- 类加载器 - 将.class文件加载到内存中。
- 字节码验证器 - 检查代码中是否存在任何访问权限违规。
- 执行引擎 - 将字节码转换为可执行的机器码。
- JIT（即时编译） - JIT 是 JVM 的一部分，用于提升 JVM 的性能。它会在执行时动态编译或翻译 Java 字节码为本地机器码。

(已编辑)

Chapter 173: Java Virtual Machine (JVM)

Section 173.1: These are the basics

JVM is an **abstract computing machine** or **Virtual machine** that resides in your RAM. It has a platform-independent execution environment that interprets Java bytecode into native machine code. (Javac is Java Compiler which compiles your Java code into Bytecode)

Java program will be running inside the JVM which is then mapped onto the underlying physical machine. It is one of programming tool in JDK.

(*Byte* code is platform-independent code which is run on every platform and *Machine* code is platform-specific code which is run in only specific platform such as windows or linux; it depend on execution.)

Some of the components:

- Class Loader - load the .class file into RAM.
- Bytecode verifier - check whether there are any access restriction violations in your code.
- Execution engine - convert the byte code into executable machine code.
- JIT(just in time) - JIT is part of JVM which used to improves the performance of JVM. It will dynamically compile or translate java bytecode into native machine code during execution time.

(Edited)

第174章：XJC

参数	详情
模式文件	将 xsd 模式文件转换为 Java

XJC 是一个 Java SE 工具，用于将 XML 模式文件编译成带有完整注释的 Java 类。

它随 JDK 包一起分发，位于 /bin/xjc 路径下。

第174.1节：从简单的 XSD 文件生成 Java 代码

XSD 架构 (schema.xsd)

以下的 XML 模式 (xsd) 定义了一个带有属性name和reputation的用户列表。

```
<?xml version="1.0"?>

<xsschema version="1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:ns="http://www.stackoverflow.com/users"
    elementFormDefault="qualified"
    targetNamespace="http://www.stackoverflow.com/users">
<xselement name="users" type="ns:Users"/>

<xsccomplexType name="Users">
    <xsssequence>
        <xselement type="ns:User" name="user" minOccurs="0" maxOccurs="unbounded"/>
    </xsssequence>
</xsccomplexType>

<xsccomplexType name="User">
    <xssattribute name="name" use="required" type="xs:string"/>
    <xssattribute name="reputation" use="required">
        <xssimpleType>
            <xsrrestriction base="xs:int">
                <xss:minInclusive value="1"/>
            </xsrrestriction>
        </xssimpleType>
    </xssattribute>
</xsccomplexType>
</xsschema>
```

使用 xjc

这要求 xjc 工具 (JDK 二进制文件) 的路径在操作系统的路径变量中。

代码生成可以通过以下方式启动

```
xjc schema.xsd
```

这将在工作目录中生成 Java 文件。

结果文件

会有一些额外的注释，但基本上生成的java文件看起来是这样的：

```
package com.stackoverflow.users;
```

Chapter 174: XJC

Parameter	Details
schema file	The xsd schema file to convert to java

XJC is a Java SE tool that compiles an XML schema file into fully annotated Java classes.

It is distributed within the JDK package and is located at /bin/xjc path.

Section 174.1: Generating Java code from simple XSD file

XSD schema (schema.xsd)

The following xml schema (xsd) defines a list of users with attributes name and reputation.

```
<?xml version="1.0"?>

<xsschema version="1.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:ns="http://www.stackoverflow.com/users"
    elementFormDefault="qualified"
    targetNamespace="http://www.stackoverflow.com/users">
<xselement name="users" type="ns:Users"/>

<xsccomplexType name="Users">
    <xsssequence>
        <xselement type="ns:User" name="user" minOccurs="0" maxOccurs="unbounded"/>
    </xsssequence>
</xsccomplexType>

<xsccomplexType name="User">
    <xssattribute name="name" use="required" type="xs:string"/>
    <xssattribute name="reputation" use="required">
        <xssimpleType>
            <xsrrestriction base="xs:int">
                <xss:minInclusive value="1"/>
            </xsrrestriction>
        </xssimpleType>
    </xssattribute>
</xsccomplexType>
</xsschema>
```

Using xjc

This requires the path to the xjc tool (JDK binaries) to be in the OS path variable.

The code generation can be started using

```
xjc schema.xsd
```

This will generate java files in the working directory.

Result files

There will be some additional comments, but basically the java files generated look like this:

```
package com.stackoverflow.users;
```

```

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Users", propOrder = {
    "user"
})
public class Users {

    protected List<User> user;

    public List<User> getUser() {
        if (user == null) {
            user = new ArrayList<User>();
        }
        return this.user;
    }

}

package com.stackoverflow.users;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "User")
public class User {

    @XmlAttribute(name = "name", required = true)
    protected String name;
    @XmlAttribute(name = "reputation", required = true)
    protected int reputation;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public int getReputation() {
        return reputation;
    }

    public void setReputation(int value) {
        this.reputation = value;
    }

}

package com.stackoverflow.users;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;

```

```

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Users", propOrder = {
    "user"
})
public class Users {

    protected List<User> user;

    public List<User> getUser() {
        if (user == null) {
            user = new ArrayList<User>();
        }
        return this.user;
    }

}

package com.stackoverflow.users;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "User")
public class User {

    @XmlAttribute(name = "name", required = true)
    protected String name;
    @XmlAttribute(name = "reputation", required = true)
    protected int reputation;

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

    public int getReputation() {
        return reputation;
    }

    public void setReputation(int value) {
        this.reputation = value;
    }

}

package com.stackoverflow.users;

import javax.xml.bind.JAXBElement;
import javax.xml.bind.annotation.XmlElementDecl;
import javax.xml.bind.annotation.XmlRegistry;
import javax.xml.namespace.QName;

```

```

@XmlRegistry
public class ObjectFactory {

    private final static QName _Users_QNAME = new QName("http://www.stackoverflow.com/users",
"users");

    public ObjectFactory() {
    }

    public Users createUsers() {
        return new Users();
    }

    public User createUser() {
        return new User();
    }

    @XmlElementDecl(namespace = "http://www.stackoverflow.com/users", name = "users")
    public JAXBElement<Users> createUsers(Users value) {
        return new JAXBElement<Users>(_Users_QNAME, Users.class, null, value);
    }
}

```

package-info.java

```

@javax.xml.bind.annotation.XmlSchema(namespace = "http://www.stackoverflow.com/users",
elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package com.stackoverflow.users;

```

```

@XmlRegistry
public class ObjectFactory {

    private final static QName _Users_QNAME = new QName("http://www.stackoverflow.com/users",
"users");

    public ObjectFactory() {
    }

    public Users createUsers() {
        return new Users();
    }

    public User createUser() {
        return new User();
    }

    @XmlElementDecl(namespace = "http://www.stackoverflow.com/users", name = "users")
    public JAXBElement<Users> createUsers(Users value) {
        return new JAXBElement<Users>(_Users_QNAME, Users.class, null, value);
    }
}

```

package-info.java

```

@javax.xml.bind.annotation.XmlSchema(namespace = "http://www.stackoverflow.com/users",
elementFormDefault = javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
package com.stackoverflow.users;

```

第175章：JVM标志

第175.1节：-XXaggressive

-XXaggressive 是一组配置，使JVM尽快以高速运行并达到稳定状态。为了实现这一目标，JVM在启动时会使用更多的内部资源；然而，一旦达到目标，所需的自适应优化就会减少。我们建议您对独立运行的长时间、高内存消耗的应用程序使用此选项。

用法：

```
-XXaggressive : <param>
```

	描述
opt	提前安排自适应优化并启用新的优化，这些优化预计将在未来版本中成为默认设置。
memory	为内存密集型工作负载配置内存系统，并设置预期以启用大量内存资源以确保高吞吐量。如果可用，JRockit JV M也将使用大页。

第175.2节：-XXallocClearChunks

此选项允许您在TLA分配时清除引用和值的TLA，并预取下一个块。
当声明整数、引用或其他任何类型时，其默认值为0或null（取决于类型）。

在适当的时候，您需要清除这些引用和值以释放堆上的内存，以便Java
可以使用或重用它。您可以在对象分配时执行此操作，或者通过使用此选项，在请求新的
TLA时执行。

用法：

```
-XXallocClearChunks
```

```
-XXallocClearChunks=<true | false>
```

上述是一个布尔选项，通常建议在IA64系统上使用；最终是否使用取决于应用程序。如果您想设置清除块的大小，请将此选项与-XXallocClearChunkSize结合使用。如果使用此标志但未指定布尔值，默认值为true。

第175.3节：-XXallocClearChunkSize

与-XXallocClearChunkSize一起使用时，此选项设置要清除的块的大小。如果使用此标志但未指定值，默认值为512字节。

用法：

```
-XXallocClearChunks -XXallocClearChunkSize=<大小>[k|K][m|M][g|G]
```

第175.4节：-XXcallProfiling

此选项启用调用分析以进行代码优化。分析记录与应用程序相关的有用运行时统计信息，并且在许多情况下可以提
高性能，因为JVM随后可以根据这些统计信息采取行动。

Chapter 175: JVM Flags

Section 175.1: -XXaggressive

-XXaggressive is a collection of configurations that make the JVM perform at a high speed and reach a stable state as soon as possible. To achieve this goal, the JVM uses more internal resources at startup; however, it requires less adaptive optimization once the goal is reached. We recommend that you use this option for long-running, memory-intensive applications that work alone.

Usage:

```
-XXaggressive:<param>
```

	Description
opt	Schedules adaptive optimizations earlier and enables new optimizations, which are expected to be the default in future releases.
memory	Configures the memory system for memory-intensive workloads and sets an expectation to enable large amounts of memory resources to ensure high throughput. JRockit JVM will also use large pages, if available.

Section 175.2: -XXallocClearChunks

This option allows you to clear a TLA for references and values at TLA allocation time and pre-fetch the next chunk. When an integer, a reference, or anything else is declared, it has a default value of 0 or null (depending upon type). At the appropriate time, you will need to clear these references and values to free the memory on the heap so Java can use- or reuse- it. You can do either when the object is allocated or, by using this option, when you request a new TLA.

Usage:

```
-XXallocClearChunks
```

```
-XXallocClearChunks=<true | false>
```

The above is a boolean option and is generally recommended on IA64 systems; ultimately, its use depends upon the application. If you want to set the size of chunks cleared, combine this option with -XXallocClearChunkSize. If you use this flag but do not specify a boolean value, the default is **true**.

Section 175.3: -XXallocClearChunkSize

When used with -XXallocClearChunkSize, this option sets the size of the chunks to be cleared. If this flag is used but no value is specified, the default is 512 bytes.

Usage:

```
-XXallocClearChunks -XXallocClearChunkSize=<size>[k|K][m|M][g|G]
```

Section 175.4: -XXcallProfiling

This option enables the use of call profiling for code optimizations. Profiling records useful runtime statistics specific to the application and can—in many cases—increase performance because JVM can then act on those statistics.

注意：此选项在JRockit JVM R27.3.0及更高版本中受支持。未来版本中可能成为默认选项。

用法：

```
java -XXcallProfiling myApp
```

此选项默认禁用。您必须启用它才能使用。

第175.5节：-XXdisableFatSpin

此选项禁用Java中的fat锁自旋代码，允许尝试获取fat锁而阻塞的线程直接进入睡眠状态。

Java中的对象一旦有线程进入该对象的同步块，就成为一个锁。所有锁都会被持有（即保持锁定状态），直到被持锁线程释放。如果锁不会很快释放，它可以被膨胀为“fat锁”。“自旋”是指想要获取特定锁的线程不断检查该锁是否仍被占用，在紧密循环中进行检查。针对fat锁的自旋通常是有益的，尽管在某些情况下可能代价较高并影响性能。`-XXdisableFatSpin`允许您关闭针对fat锁的自旋，从而消除潜在的性能损失。

用法：

```
-XXdisableFatSpin
```

第175.6节：-XXdisableGCHeistics

此选项禁用垃圾回收器策略的更改。压缩启发式和新生代大小启发式不受此选项影响。默认情况下，垃圾回收启发式是启用的。

用法：

```
-XXdisableFatSpin
```

第175.7节：-XXdumpSize

此选项会生成转储文件，并允许您指定该文件的相对大小（即小、中或大）。

用法：

<大小>	描述
无	不生成转储文件。
小	在Windows上，会生成一个小型转储文件（在Linux上会生成完整的核心转储）。小型转储仅包含线程堆栈及其跟踪信息，几乎不包含其他内容。这是JRockit JVM 8.1（服务包1和2）以及7.0（服务包3及更高版本）的默认设置。
正常	在所有平台上生成正常转储。该转储文件包含除Java堆之外的所有内存。这是JRockit JVM 1.4.2及更高版本的默认值。
large	包括内存中的所有内容，包括Java堆。此选项使得 <code>-XXdumpSize</code> 等同于 <code>-XXdumpFullState</code> 。

Note: This option is supported with the JRockit JVM R27.3.0 and later version. It may become default in future versions.

Usage:

```
java -XXcallProfiling myApp
```

This option is disabled by default. You must enable it to use it.

Section 175.5: -XXdisableFatSpin

This option disables the fat lock spin code in Java, allowing threads that block trying to acquire a fat lock go to sleep directly.

Objects in Java become a lock as soon as any thread enters a synchronized block on that object. All locks are held (that is, stayed locked) until released by the locking thread. If the lock is not going to be released very fast, it can be inflated to a “fat lock.” “Spinning” occurs when a thread that wants a specific lock continuously checks that lock to see if it is still taken, spinning in a tight loop as it makes the check. Spinning against a fat lock is generally beneficial although, in some instances, it can be expensive and might affect performance. `-XXdisableFatSpin` allows you to turn off spinning against a fat lock and eliminate the potential performance hit.

Usage:

```
-XXdisableFatSpin
```

Section 175.6: -XXdisableGCHeistics

This option disables the garbage collector strategy changes. Compaction heuristics and nursery size heuristics are not affected by this option. By default, the garbage collection heuristics are enabled.

Usage:

```
-XXdisableFatSpin
```

Section 175.7: -XXdumpSize

This option causes a dump file to be generated and allows you to specify the relative size of that file (that is, small, medium, or large).

Usage:

<size>	Description
none	Does not generate a dump file.
small	On Windows, a small dump file is generated (on Linux a full core dump is generated). A small dump only include the thread stacks including their traces and very little else. This was the default in the JRockit JVM 8.1 with service packs 1 and 2, as well as 7.0 with service pack 3 and higher.
normal	Causes a normal dump to be generated on all platforms. This dump file includes all memory except the java heap. This is the default value for the JRockit JVM 1.4.2 and later.
large	Includes everything that is in memory, including the Java heap. This option makes <code>-XXdumpSize</code> equivalent to <code>-XXdumpFullState</code> .

第175.8节 : -XXexitOnOutOfMemory

此选项使JRockit JVM在首次发生内存溢出错误时退出。如果您更倾向于重启JRockit JVM实例而不是处理内存溢出错误，可以使用此选项。在启动时输入此命令，强制JRockit JVM在首次发生内存溢出错误时退出。

用法：

```
-XXexitOnOutOfMemory
```

Section 175.8: -XXexitOnOutOfMemory

This option makes JRockit JVM exit on the first occurrence of an out of memory error. It can be used if you prefer restarting an instance of JRockit JVM rather than handling out of memory errors. Enter this command at startup to force JRockit JVM to exit on the first occurrence of an out of memory error.

Usage:

```
-XXexitOnOutOfMemory
```

第176章：JVM工具接口

第176.1节：遍历从对象可达的对象 (Heap 1.0)

```
#include <vector>
#include <string>

#include "agent_util.hpp"
//该文件可在Java SE开发工具包8u101演示和示例中找到
//见 http://download.oracle.com/otn-pub/java/jdk/8u101-b13-demos/jdk-8u101-windows-x64-demos.zip
//jdk1.8.0_101.zip!\demo\jvmti\versionCheck\src\agent_util.h

/*
 * 用于jvmti->SetTag(object, <指向标签的指针>)的结构体 ;
 * http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#SetTag
 */
typedef struct Tag
{
    jlong referrer_tag;
    jlong size;
    char* classSignature;
    jint hashCode;
} Tag;

/*
 * 工具函数 : jlong -> Tag*
 */
static Tag* pointerToTag(jlong tag_ptr)
{
    if (tag_ptr == 0)
    {
        return new Tag();
    }
    return (Tag*)(ptrdiff_t)(void*)tag_ptr;
}

/*
 * 实用函数 : Tag* -> jlong
 */
static jlong tagToPointer(Tag* tag)
{
    return (jlong)(ptrdiff_t)(void*)tag;
}

/*
 * 堆 1.0 回调
 * http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#jvmtiObjectReferenceCallback
 */
static jvmtiIterationControl JNICALL heapObjectReferencesCallback(
    jvmtiObjectReferenceKind reference_kind,
    jlong class_tag,
    jlong size,
    jlong* tag_ptr,
    jlong referrer_tag,
    jint referrer_index,
    void* user_data)
{
```

Chapter 176: JVM Tool Interface

Section 176.1: Iterate over objects reachable from object (Heap 1.0)

```
#include <vector>
#include <string>

#include "agent_util.hpp"
//this file can be found in Java SE Development Kit 8u101 Demos and Samples
//see http://download.oracle.com/otn-pub/java/jdk/8u101-b13-demos/jdk-8u101-windows-x64-demos.zip
//jdk1.8.0_101.zip!\demo\jvmti\versionCheck\src\agent_util.h

/*
 * Struct used for jvmti->SetTag(object, <pointer to tag>);
 * http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#SetTag
 */
typedef struct Tag
{
    jlong referrer_tag;
    jlong size;
    char* classSignature;
    jint hashCode;
} Tag;

/*
 * Utility function: jlong -> Tag*
 */
static Tag* pointerToTag(jlong tag_ptr)
{
    if (tag_ptr == 0)
    {
        return new Tag();
    }
    return (Tag*)(ptrdiff_t)(void*)tag_ptr;
}

/*
 * Utility function: Tag* -> jlong
 */
static jlong tagToPointer(Tag* tag)
{
    return (jlong)(ptrdiff_t)(void*)tag;
}

/*
 * Heap 1.0 Callback
 * http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#jvmtiObjectReferenceCallback
 */
static jvmtiIterationControl JNICALL heapObjectReferencesCallback(
    jvmtiObjectReferenceKind reference_kind,
    jlong class_tag,
    jlong size,
    jlong* tag_ptr,
    jlong referrer_tag,
    jint referrer_index,
    void* user_data)
{
```

```

// 仅遍历引用字段
if (reference_kind != JVMTI_HEAP_REFERENCE_FIELD)
{
    return JVMTI_ITERATION_IGNORE;
}
auto tag_ptr_list = (std::vector<jlong>*)(ptrdiff_t)(void*)user_data;
// 创建并分配标签
auto t = pointerToTag(*tag_ptr);
t->referrer_tag = referrer_tag;
t->size = size;
*tag_ptr = tagToPointer(t);
// 收集标签
(*tag_ptr_list).push_back(*tag_ptr);

return JVMTI_ITERATION_CONTINUE;
}

/*
* 用于演示遍历从对象可达的对象的主函数
*
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject
*/
void iterateOverObjectHeapReferences(jvmtiEnv* jvmti, JNIEnv* env, jobject object)
{
std::vector<jlong> tag_ptr_list;

auto t = new Tag();
jvmti->SetTag(object, tagToPointer(t)); tag_ptr_list.push_back(tagToPointer(t));
stdout_message("调用回

调前标签列表大小: %d", tag_ptr_list.size());
/* 对每个可达对象引用调用回调函数
 * 详见
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject
 */
jvmti->IterateOverObjectsReachableFromObject(object, &heapObjectReferencesCallback,
(void*)&tag_ptr_list);
stdout_message("调用回调后标签列表大小: %d", tag_ptr_list.size());

if (tag_ptr_list.size() > 0)
{
jint found_count = 0;
jlong* tags = &tag_ptr_list[0];
jobject* found_objects;
jlong* found_tags;

/*
* 收集所有带标签的对象 (通过 *tag_ptr = 指向标签的指针)
 * 详见 http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectsWithTags
 */
jvmti->GetObjectsWithTags(tag_ptr_list.size(), tags, &found_count, &found_objects,
&found_tags);
stdout_message("找到 %d 个对象", found_count);

for (auto i = 0; i < found_count; ++i)
{
jobject found_object = found_objects[i];

char* classSignature;
 jclass found_object_class = env->GetObjectClass(found_object);

```

```

// iterate only over reference field
if (reference_kind != JVMTI_HEAP_REFERENCE_FIELD)
{
    return JVMTI_ITERATION_IGNORE;
}
auto tag_ptr_list = (std::vector<jlong>*)(ptrdiff_t)(void*)user_data;
// create and assign tag
auto t = pointerToTag(*tag_ptr);
t->referrer_tag = referrer_tag;
t->size = size;
*tag_ptr = tagToPointer(t);
// collect tag
(*tag_ptr_list).push_back(*tag_ptr);

return JVMTI_ITERATION_CONTINUE;
}

/*
* Main function for demonstration of Iterate Over Objects Reachable From Object
*
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject
*/
void iterateOverObjectHeapReferences(jvmtiEnv* jvmti, JNIEnv* env, jobject object)
{
std::vector<jlong> tag_ptr_list;

auto t = new Tag();
jvmti->SetTag(object, tagToPointer(t));
tag_ptr_list.push_back(tagToPointer(t));

stdout_message("tag list size before call callback: %d\n", tag_ptr_list.size());
/*
 * Call Callback for every reachable object reference
 * see
http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#IterateOverObjectsReachableFromObject
 */
jvmti->IterateOverObjectsReachableFromObject(object, &heabObjectReferencesCallback,
(void*)&tag_ptr_list);
stdout_message("tag list size after call callback: %d\n", tag_ptr_list.size());

if (tag_ptr_list.size() > 0)
{
jint found_count = 0;
jlong* tags = &tag_ptr_list[0];
jobject* found_objects;
jlong* found_tags;

/*
* collect all tagged object (via *tag_ptr = pointer to tag )
* see http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectsWithTags
*/
jvmti->GetObjectsWithTags(tag_ptr_list.size(), tags, &found_count, &found_objects,
&found_tags);
stdout_message("found %d objects\n", found_count);

for (auto i = 0; i < found_count; ++i)
{
jobject found_object = found_objects[i];

char* classSignature;
jclass found_object_class = env->GetObjectClass(found_object);

```

```

/*
* 获取 found_object_class 的字符串表示
* 参见 http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetClassSignature
*/
jvmti->GetClassSignature(found_object_class, &classSignature, nullptr);

jint hashCode;
/*
* 获取 found_object 的哈希码
* 参见 http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectHashCode
*/
jvmti->GetObjectHashCode(found_object, &hashCode);

//将所有内容保存到 Tag 中
Tag* t = pointerToTag(found_tags[i]);
t->classSignature = classSignature;
t->hashCode = hashCode;
}

//打印所有保存的信息
for (auto i = 0; i < found_count; ++i)
{
    auto t = pointerToTag(found_tags[i]);
    auto rt = pointerToTag(t->referrer_tag);

    if (t->referrer_tag != 0)
    {
        stdout_message("引用对象 %s%d --> 对象 %s%d (大小: %2d)",
                      rt->classSignature, rt-
                      >hashCode, t->classSignature, t->hashCode, t->size);
    }
}
}

```

第176.2节：获取JVMTI环境

在Agent_OnLoad方法内部：

```

jvmtiEnv* jvmti;
/* 获取JVMTI环境 */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);

```

第176.3节：Agent_OnLoad方法内部初始化示例

```

/* JVMTI_EVENT_VM_INIT的回调 */
static void JNICALL vm_init(jvmtiEnv* jvmti, JNIEnv* env, jthread thread)
{
jint runtime_version;
jvmti->GetVersionNumber(&runtime_version);    stdout_
message("JVMTI 版本: %d", runtime_version);

/* Agent_Load() 首先被调用，我们在这里为 VM_INIT 事件做准备。 */
JNIEXPORT jint JNICALL
Agent_Load(JavaVM* vm, char* options, void* reserved)
{
jint rc;
jvmtiEventCallbacks callbacks;

```

```

/*
* Get string representation of found_object_class
* see http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetClassSignature
*/
jvmti->GetClassSignature(found_object_class, &classSignature, nullptr);

jint hashCode;
/*
* Getting hash code for found_object
* see http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html#GetObjectHashCode
*/
jvmti->GetObjectHashCode(found_object, &hashCode);

//save all it in Tag
Tag* t = pointerToTag(found_tags[i]);
t->classSignature = classSignature;
t->hashCode = hashCode;
}

//print all saved information
for (auto i = 0; i < found_count; ++i)
{
    auto t = pointerToTag(found_tags[i]);
    auto rt = pointerToTag(t->referrer_tag);

    if (t->referrer_tag != 0)
    {
        std::cout << "referrer object " << rt->classSignature << " --> object " << rt->hashCode << " (size: " << t->classSignature << ", " << t->hashCode << ", " << t->size) " << std::endl;
    }
}
}

```

Section 176.2: Get JVMTI environment

Inside Agent_OnLoad method:

```

jvmtiEnv* jvmti;
/* Get JVMTI environment */
vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);

```

Section 176.3: Example of initialization inside of Agent_OnLoad method

```

/* Callback for JVMTI_EVENT_VM_INIT */
static void JNICALL vm_init(jvmtiEnv* jvmti, JNIEnv* env, jthread thread)
{
jint runtime_version;
jvmti->GetVersionNumber(&runtime_version);
stdout_message("JVMTI Version: %d\n", runtime_version);

/* Agent_Load() is called first, we prepare for a VM_INIT event here. */
JNIEXPORT jint JNICALL
Agent_Load(JavaVM* vm, char* options, void* reserved)
{
jint rc;
jvmtiEventCallbacks callbacks;

```

```

jvmtiCapabilities capabilities;
jvmtiEnv* jvmti;

/* 获取JVMTI环境 */
rc = vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
if (rc != JNI_OK)
{
    return -1;
}

/* 在获取 jvmtiEnv* 后，我们需要立即请求
* 该代理所需的功能。
*/
jvmti->GetCapabilities(&capabilities);
capabilities.can_tag_objects = 1;
jvmti->AddCapabilities(&capabilities);

/* 设置回调并启用事件通知 */
memset(&callbacks, 0, sizeof(callbacks));
callbacks.VMInit = &vm_init;

jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, nullptr);

return JNI_OK;
}

```

```

jvmtiCapabilities capabilities;
jvmtiEnv* jvmti;

/* Get JVMTI environment */
rc = vm->GetEnv(reinterpret_cast<void **>(&jvmti), JVMTI_VERSION);
if (rc != JNI_OK)
{
    return -1;
}

/* Immediately after getting the jvmtiEnv* we need to ask for the
* capabilities this agent will need.
*/
jvmti->GetCapabilities(&capabilities);
capabilities.can_tag_objects = 1;
jvmti->AddCapabilities(&capabilities);

/* Set callbacks and enable event notifications */
memset(&callbacks, 0, sizeof(callbacks));
callbacks.VMInit = &vm_init;

jvmti->SetEventCallbacks(&callbacks, sizeof(callbacks));
jvmti->SetEventNotificationMode(JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, nullptr);

return JNI_OK;
}

```

第177章：Java内存管理

第177.1节：设置堆、永久代和栈大小

当Java虚拟机启动时，它需要知道堆的大小以及线程栈的默认大小。这些可以通过java命令的命令行选项来指定。对于Java 8之前的版本，还可以指定堆的PermGen区域的大小。

请注意，PermGen在Java 8中被移除，如果尝试设置PermGen大小，该选项将被忽略（并显示警告信息）。

如果不显式指定堆和栈的大小，JVM将使用根据版本和平台特定方式计算的默认值。这可能导致您的应用程序使用的内存过少或过多。对于线程栈来说通常没问题，但对于使用大量内存的程序可能会有问题。

设置堆、PermGen和默认栈大小：

以下JVM选项设置堆大小：

- -Xms<大小> - 设置初始堆大小
- -Xmx<大小> - 设置最大堆大小
- -XX:PermSize<大小> - 设置初始PermGen大小
- -XX:MaxPermSize<大小> - 设置最大PermGen大小
- -Xss<大小> - 设置默认线程栈大小

参数<size>可以是字节数，也可以带有k、m或g后缀。后者分别表示以千字节、兆字节和千兆字节为单位的大小。

示例：

```
$ java -Xms512m -Xmx1024m JavaApp  
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp  
$ java -Xss512k JavaApp
```

查找默认大小：

可以使用-XX:+printFlagsFinal选项在启动JVM之前打印所有标志的值。该选项可用于打印堆和栈大小设置的默认值，如下所示：

- 适用于Linux、Unix、Solaris和Mac OSX

```
$ java -XX:+PrintFlagsFinal -version | grep -iE 'HeapSize|PermSize|ThreadStackSize'
```

- 适用于Windows：

```
java -XX:+PrintFlagsFinal -version | findstr /i "HeapSize PermSize ThreadStackSize"
```

上述命令的输出将类似于以下内容：

Chapter 177: Java Memory Management

Section 177.1: Setting the Heap, PermGen and Stack sizes

When a Java virtual machine starts, it needs to know how big to make the Heap, and the default size for thread stacks. These can be specified using command-line options on the java command. For versions of Java prior to Java 8, you can also specify the size of the PermGen region of the Heap.

Note that PermGen was removed in Java 8, and if you attempt to set the PermGen size the option will be ignored (with a warning message).

If you don't specify Heap and Stack sizes explicitly, the JVM will use defaults that are calculated in a version and platform specific way. This may result in your application using too little or too much memory. This is typically OK for thread stacks, but it can be problematic for a program that uses a lot of memory.

Setting the Heap, PermGen and default Stack sizes:

The following JVM options set the heap size:

- -Xms<size> - sets the initial heap size
- -Xmx<size> - sets the maximum heap size
- -XX:PermSize<size> - sets the initial PermGen size
- -XX:MaxPermSize<size> - sets the maximum PermGen size
- -Xss<size> - sets the default thread stack size

The <size> parameter can be a number of bytes, or can have a suffix of k, m or g. The latter specify the size in kilobytes, megabytes and gigabytes respectively.

Examples:

```
$ java -Xms512m -Xmx1024m JavaApp  
$ java -XX:PermSize=64m -XX:MaxPermSize=128m JavaApp  
$ java -Xss512k JavaApp
```

Finding the default sizes:

The -XX:+printFlagsFinal option can be used to print the values of all flags before starting the JVM. This can be used to print the defaults for the heap and stack size settings as follows:

- For Linux, Unix, Solaris and Mac OSX

```
$ java -XX:+PrintFlagsFinal -version | grep -iE 'HeapSize|PermSize|ThreadStackSize'
```

- For Windows:

```
java -XX:+PrintFlagsFinal -version | findstr /i "HeapSize PermSize ThreadStackSize"
```

The output of the above commands will resemble the following:

```

uintx 初始堆大小          := 20655360      {product}
uintx 最大堆大小          := 331350016    {product}
uintx 永久代大小          = 21757952       {pd product}
uintx 最大永久代大小      = 85983232       {pd product}
intx 线程栈大小          = 1024           {pd product}

```

大小以字节为单位。

第177.2节：垃圾回收

C++方法——new和delete

在像C++这样的语言中，应用程序负责管理动态分配内存所使用的内存。当使用new操作符在C++堆中创建对象时，需要对应使用delete操作符来销毁该对象：

- 如果程序忘记delete一个对象而只是“忘记”它，相关的内存就会丢失给应用程序。这种情况称为内存泄漏，如果内存泄漏过多，应用程序可能会使用越来越多的内存，最终崩溃。
- 另一方面，如果应用程序尝试对同一个对象执行两次delete，或者在对象被删除后继续使用该对象，应用程序可能会因内存损坏问题而崩溃。

在复杂的C++程序中，使用new和delete实现内存管理可能非常耗时。实际上，内存管理是常见的错误来源。

Java方法——垃圾回收

Java采取了不同的方法。Java没有显式的delete操作符，而是提供了一种称为垃圾回收的自动机制，用于回收不再需要的对象所占用的内存。Java运行时系统负责查找需要处理的对象。这个任务由一个称为垃圾收集器（GC）的组件执行。

在Java程序执行的任何时刻，我们都可以将所有存在的对象集合划分为两个不同的子集1：

- 可达对象在Java语言规范（JLS）中定义如下：

可达对象是指任何可以从任何活动线程的潜在后续计算中访问的对象。

实际上，这意味着存在一条引用链，从一个作用域内的局部变量或一个静态变量开始，某段代码可能通过这条链访问该对象。

- 不可达对象是指不可能被上述方式访问的对象。

任何不可达的对象都是有资格进行垃圾回收的。这并不意味着它们一定会被垃圾回收。事实上：

- 一个不可达对象不会在变为不可达时立即被回收1。
- 一个不可达对象可能永远不会被垃圾回收。

```

uintx InitialHeapSize          := 20655360      {product}
uintx MaxHeapSize              := 331350016    {product}
uintx PermSize                 = 21757952       {pd product}
uintx MaxPermSize              = 85983232       {pd product}
intx ThreadStackSize           = 1024           {pd product}

```

The sizes are given in bytes.

Section 177.2: Garbage collection

The C++ approach - new and delete

In a language like C++, the application program is responsible for managing the memory used by dynamically allocated memory. When an object is created in the C++ heap using the **new** operator, there needs to be a corresponding use of the **delete** operator to dispose of the object:

- If program forgets to delete an object and just "forgets" about it, the associated memory is lost to the application. The term for this situation is a *memory leak*, and it too much memory leaks an application is liable to use more and more memory, and eventually crash.
- On the other hand, if an application attempts to delete the same object twice, or use an object after it has been deleted, then the application is liable to crash due to problems with memory corruption

In a complicated C++ program, implementing memory management using **new** and **delete** can be time consuming. Indeed, memory management is a common source of bugs.

The Java approach - garbage collection

Java takes a different approach. Instead of an explicit **delete** operator, Java provides an automatic mechanism known as garbage collection to reclaim the memory used by objects that are no longer needed. The Java runtime system takes responsibility for finding the objects to be disposed of. This task is performed by a component called a *garbage collector*, or GC for short.

At any time during the execution of a Java program, we can divide the set of all existing objects into two distinct subsets1:

- Reachable objects are defined by the JLS as follows:

A reachable object is any object that can be accessed in any potential continuing computation from any live thread.

In practice, this means that there is a chain of references starting from an in-scope local variable or a **static** variable by which some code might be able to reach the object.

- Unreachable objects are objects that *cannot possibly* be reached as above.

Any objects that are unreachable are *eligible* for garbage collection. This does not mean that they *will* be garbage collected. In fact:

- An unreachable object *does not* get collected immediately on becoming unreachable1.
- An unreachable object *may not* ever be garbage collected.

Java语言规范给予JVM实现很大的自由度来决定何时回收不可达对象。它也（在实际中）允许JVM实现对不可达对象的检测采取保守策略。

Java语言规范（JLS）保证的唯一一件事是，任何可达的对象永远不会被垃圾回收。

当一个对象变得不可达时会发生什么

首先，当一个对象变得不可达时，并不会发生任何特定的事情。只有当垃圾回收器运行并且检测到该对象不可达时，才会发生事情。此外，垃圾回收运行时通常不会检测到所有不可达的对象。

当垃圾回收器检测到一个不可达对象时，可能会发生以下事件。

- 如果有任何引用（Reference）对象指向该对象，这些引用将在对象被删除之前被清除。
- 如果该对象是可终结的（finalizable），那么它将被终结。这发生在对象被删除之前。
- 该对象可以被删除，其占用的内存可以被回收。

注意，上述事件可以发生的顺序是明确的，但垃圾回收器并不要求在任何特定的时间范围内对任何特定对象执行最终删除。

可达对象和不可达对象的示例

考虑以下示例类：

```
// 简单“开放”链表中的一个节点。
public class Node {
    private static int counter = 0;

    public int nodeNumber = ++counter;
    public Node next;
}

public class ListTest {
    public static void main(String[] args) {
        test(); // M1
        System.out.println("Done"); // M2
    }

    private static void test() {
        Node n1 = new Node(); // T1
        Node n2 = new Node(); // T2
        Node n3 = new Node(); // T3
        n1.next = n2; // T4
        n2 = null; // T5
        n3 = null; // T6
    }
}
```

让我们来看看调用 `test()` 时会发生什么。语句 T1、T2 和 T3 创建了 `Node` 对象，这些对象分别通过变量 `n1`、`n2` 和 `n3` 可访问。语句 T4 将第二个 `Node` 对象的引用赋给第一个对象的 `next` 字段。完成后，第二个 `Node` 可以通过两条路径访问：

`n2 -> Node2`

The Java language Specification gives a lot of latitude to a JVM implementation to decide when to collect unreachable objects. It also (in practice) gives permission for a JVM implementation to be conservative in how it detects unreachable objects.

The one thing that the JLS guarantees is that no *reachable* objects will ever be garbage collected.

What happens when an object becomes unreachable

First of all, nothing specifically happens when an object *becomes* unreachable. Things only happen when the garbage collector runs *and* it detects that the object is unreachable. Furthermore, it is common for a GC run to not detect all unreachable objects.

When the GC detects an unreachable object, the following events can occur.

- If there are any `Reference` objects that refer to the object, those references will be cleared before the object is deleted.
- If the object is `finalizable`, then it will be finalized. This happens before the object is deleted.
- The object can be deleted, and the memory it occupies can be reclaimed.

Note that there is a clear sequence in which the above events *can* occur, but nothing requires the garbage collector to perform the final deletion of any specific object in any specific time-frame.

Examples of reachable and unreachable objects

Consider the following example classes:

```
// A node in simple "open" linked-list.
public class Node {
    private static int counter = 0;

    public int nodeNumber = ++counter;
    public Node next;
}

public class ListTest {
    public static void main(String[] args) {
        test(); // M1
        System.out.println("Done"); // M2
    }

    private static void test() {
        Node n1 = new Node(); // T1
        Node n2 = new Node(); // T2
        Node n3 = new Node(); // T3
        n1.next = n2; // T4
        n2 = null; // T5
        n3 = null; // T6
    }
}
```

Let us examine what happens when `test()` is called. Statements T1, T2 and T3 create `Node` objects, and the objects are all reachable via the `n1`, `n2` and `n3` variables respectively. Statement T4 assigns the reference to the 2nd `Node` object to the `next` field of the first one. When that is done, the 2nd `Node` is reachable via two paths:

`n2 -> Node2`

n1 -> Node1, Node1.next -> Node2

在语句T5中，我们将`null`赋值给n2。这打断了Node2的第一个可达链，但第二个链仍然保持完整，因此Node2仍然是可达的。

在语句 T6 中，我们将 `null` 赋值给了 n3。这打破了 Node3 唯一的可达链，使得 Node3 不可达。然而，Node1 和 Node2 仍然可以通过 n1 变量访问。

最后，当 `test()` 方法返回时，其局部变量 n1、n2 和 n3 超出作用域，因此无法被任何东西访问。这打破了 Node1 和 Node2 剩余的可达链，所有的 Node 对象都变得不可达并且 `eligible` 进行垃圾回收。

1 - 这是一个忽略了终结 (finalization) 和 Reference 类的简化说明。2 - 理论上，Java 实现可以这样做，但这样做的性能代价使其不切实际。

第177.3节：Java中的内存泄漏

在垃圾回收示例中，我们暗示Java解决了内存泄漏问题。实际上这并不正确。Java程序也可能发生内存泄漏，尽管泄漏的原因有所不同。

可达对象也会泄漏

考虑以下简单的栈实现。

```
public class NaiveStack {
    private Object[] stack = new Object[100];
    private int top = 0;

    public void push(Object obj) {
        if (top >= stack.length) {
            throw new StackException("stack overflow");
        }
        stack[top++] = obj;
    }

    public Object pop() {
        if (top <= 0) {
            throw new StackException("stack underflow");
        }
        return stack[--top];
    }

    public boolean isEmpty() {
        return top == 0;
    }
}
```

当你push一个对象然后立即pop它时，stack数组中仍然会有对该对象的引用。

栈实现的逻辑意味着该引用不能返回给API的客户端。如果一个对象已经被弹出，那么我们可以证明它不能“在任何活跃线程的任何潜在后续计算中被访问”。问题是当前一代JVM无法证明这一点。当前一代JVM在确定引用是否可达时不考虑程序的逻辑。（首先，这在实际操作中不可行。）但撇开可达性的真正含义不谈，我们显然处于这样一种情况：NaiveStack实现了“保留”了本应被回收的对象。这就是内存泄漏。

n1 -> Node1, Node1.next -> Node2

In statement T5, we assign `null` to n2. This breaks the first of the reachability chains for Node2, but the second one remains unbroken, so Node2 is still reachable.

In statement T6, we assign `null` to n3. This breaks the only reachability chain for Node3, which makes Node3 unreachable. However, Node1 and Node2 are both still reachable via the n1 variable.

Finally, when the `test()` method returns, its local variables n1, n2 and n3 go out of scope, and therefore cannot be accessed by anything. This breaks the remaining reachability chains for Node1 and Node2, and all of the Node objects are now unreachable and `eligible` for garbage collection.

1 - This is a simplification that ignores finalization, and Reference classes. 2 - Hypothetically, a Java implementation could do this, but the performance cost of doing this makes it impractical.

Section 177.3: Memory leaks in Java

In the Garbage collection example, we implied that Java solves the problem of memory leaks. This is not actually true. A Java program can leak memory, though the causes of the leaks are rather different.

Reachable objects can leak

Consider the following naive stack implementation.

```
public class NaiveStack {
    private Object[] stack = new Object[100];
    private int top = 0;

    public void push(Object obj) {
        if (top >= stack.length) {
            throw new StackException("stack overflow");
        }
        stack[top++] = obj;
    }

    public Object pop() {
        if (top <= 0) {
            throw new StackException("stack underflow");
        }
        return stack[--top];
    }

    public boolean isEmpty() {
        return top == 0;
    }
}
```

When you push an object and then immediately pop it, there will still be a reference to the object in the stack array.

The logic of the stack implementation means that that reference cannot be returned to a client of the API. If an object has been popped then we can prove that it cannot “be accessed in any potential continuing computation from any live thread”. The problem is that a current generation JVM cannot prove this. Current generation JVMs do not consider the logic of the program in determining whether references are reachable. (For a start, it is not practical.)

But setting aside the issue of what `reachability` really means, we clearly have a situation here where the NaiveStack implementation is “hanging onto” objects that ought to be reclaimed. That is a memory leak.

在这种情况下，解决方案很简单：

```
public Object pop() {
    if (top <= 0) {
        throw new StackException("stack underflow");
    }
    Object popped = stack[--top];
    stack[top] = null;           // 用 null 覆盖弹出的引用。
    return popped;
}
```

缓存可能导致内存泄漏

一种常见的提升服务性能的策略是缓存结果。其思路是将常见请求及其结果记录在称为缓存的内存数据结构中。然后，每次发出请求时，都会在缓存中查找该请求。如果查找成功，就返回对应的保存结果。

如果正确实现，这种策略非常有效。然而，如果实现不当，缓存可能会导致内存泄漏。考虑以下示例：

```
public class RequestHandler {
    private Map<Task, Result> cache = new HashMap<>();

    public Result doRequest(Task task) {
        Result result = cache.get(task);
        if (result == null) {
            result = doRequestProcessing(task);
            cache.put(task, result);
        }
        return result;
    }
}
```

这个代码的问题在于，虽然任何对 `doRequest` 的调用都可能向缓存中添加新条目，但没有任何机制来移除它们。如果服务持续接收不同的任务，那么缓存最终会耗尽所有可用内存。这是一种内存泄漏。

解决这个问题的一种方法是使用具有最大容量的缓存，当缓存超过最大容量时丢弃旧条目。（丢弃最近最少使用的条目是一个不错的策略。）另一种方法是使用 `WeakHashMap` 构建缓存，这样当堆内存开始变得过满时，JVM可以驱逐缓存条目。

第177.4节：终结

Java对象可以声明一个 `finalize` 方法。该方法在Java释放对象内存之前调用。它通常看起来像这样：

```
public class MyClass {

    // 类的方法

    @Override
    protected void finalize() throws Throwable {
        // 清理代码
    }
}
```

但是，关于Java终结的行为有一些重要的注意事项。

In this case, the solution is straightforward:

```
public Object pop() {
    if (top <= 0) {
        throw new StackException("stack underflow");
    }
    Object popped = stack[--top];
    stack[top] = null;           // Overwrite popped reference with null.
    return popped;
}
```

Caches can be memory leaks

A common strategy for improving service performance is to cache results. The idea is that you keep a record of common requests and their results in an in-memory data structure known as a cache. Then, each time a request is made, you lookup the request in the cache. If the lookup succeeds, you return the corresponding saved results.

This strategy can be very effective if implemented properly. However, if implemented incorrectly, a cache can be a memory leak. Consider the following example:

```
public class RequestHandler {
    private Map<Task, Result> cache = new HashMap<>();

    public Result doRequest(Task task) {
        Result result = cache.get(task);
        if (result == null) {
            result = doRequestProcessing(task);
            cache.put(task, result);
        }
        return result;
    }
}
```

The problem with this code is that while any call to `doRequest` could add a new entry to the cache, there is nothing to remove them. If the service is continually getting different tasks, then the cache will eventually consume all available memory. This is a form of memory leak.

One approach to solving this is to use a cache with a maximum size, and throw out old entries when the cache exceeds the maximum. (Throwing out the least recently used entry is a good strategy.) Another approach is to build the cache using `WeakHashMap` so that the JVM can evict cache entries if the heap starts getting too full.

Section 177.4: Finalization

A Java object may declare a `finalize` method. This method is called just before Java releases the memory for the object. It will typically look like this:

```
public class MyClass {

    // Methods for the class

    @Override
    protected void finalize() throws Throwable {
        // Cleanup code
    }
}
```

However, there are some important caveats on the behavior of Java finalization.

- Java并不保证 finalize()方法何时被调用。
- Java甚至不能保证在运行中的应用程序生命周期内某个时间会调用finalize()方法。
- 唯一可以保证的是该方法会在对象被删除之前调用.....如果对象被删除的话。

上述警告意味着依赖finalize方法来执行必须及时完成的清理（或其他）操作是个坏主意。过度依赖终结器可能导致存储泄漏、内存泄漏和其他问题。

简而言之，实际上很少有情况适合使用终结器作为解决方案。

终结器只运行一次

通常，对象在被终结后会被删除。然而，这并非总是如此。考虑以下示例1：

```
public class CaptainJack {
    public static CaptainJack notDeadYet = null;

    protected void finalize() {
        // 复活！
        notDeadYet = this;
    }
}
```

当CaptainJack的一个实例变得不可达且垃圾回收器尝试回收它时，finalize()方法会将该实例的引用赋值给notDead Yet变量。这将使该实例再次变得可达，垃圾回收器不会删除它。

问题：Captain Jack是不朽的吗？

答案：不。

关键是JVM在对象的生命周期内只会运行一次终结器。如果你将null赋值给notDeadYet，导致一个复活的实例再次变得不可达，垃圾收集器将不会对该对象调用finalize()。

1 - 参见 https://en.wikipedia.org/wiki/Jack_Harkness。

第177.5节：手动触发垃圾回收

你可以通过调用手动触发垃圾收集器

`System.gc();`

但是，Java并不保证调用返回时垃圾收集器已经运行。该方法只是向JVM（Java虚拟机）“建议”你希望它运行垃圾收集器，但并不强制执行。

通常认为尝试手动触发垃圾回收是不好的做法。JVM可以通过-XX:+DisableExplicitGC选项来禁用对System.gc()的调用。通过调用System.gc()触发垃圾回收可能会干扰JVM所使用的特定垃圾收集器实现的正常垃圾管理/对象晋升活动。

- Java makes no guarantees about when a finalize() method will be called.
- Java does not even guarantee that a finalize() method will be called some time during the running application's lifetime.
- The only thing that is guaranteed is that the method will be called before the object is deleted ... if the object is deleted.

The caveats above mean that it is a bad idea to rely on the finalize method to perform cleanup (or other) actions that must be performed in a timely fashion. Over reliance on finalization can lead to storage leaks, memory leaks and other problems.

In short, there are very few situations where finalization is actually a good solution.

Finalizers only run once

Normally, an object is deleted after it has been finalized. However, this doesn't happen all of the time. Consider the following example1:

```
public class CaptainJack {
    public static CaptainJack notDeadYet = null;

    protected void finalize() {
        // Resurrection!
        notDeadYet = this;
    }
}
```

When an instance of CaptainJack becomes unreachable and the garbage collector attempts to reclaim it, the finalize() method will assign a reference to the instance to the notDeadYet variable. That will make the instance reachable once more, and the garbage collector won't delete it.

Question: Is Captain Jack immortal?

Answer: No.

The catch is the JVM will only run a finalizer on an object once in its lifetime. If you assign `null` to notDeadYet causing a resurrected instance to be unreachable once more, the garbage collector won't call finalize() on the object.

1 - See https://en.wikipedia.org/wiki/Jack_Harkness.

Section 177.5: Manually triggering GC

You can manually trigger the Garbage Collector by calling

`System.gc();`

However, Java does not guarantee that the Garbage Collector has run when the call returns. This method simply “suggests” to the JVM (Java Virtual Machine) that you want it to run the garbage collector, but does not force it to do so.

It is generally considered a bad practice to attempt to manually trigger garbage collection. The JVM can be run with the `-XX:+DisableExplicitGC` option to disable calls to `System.gc()`. Triggering garbage collection by calling `System.gc()` can disrupt normal garbage management / object promotion activities of the specific garbage collector implementation in use by the JVM.

第178章：Java性能调优

第178.1节：基于证据的Java性能调优方法

唐纳德·克努斯（Donald Knuth）常被引用的一句话是：

“程序员浪费大量时间思考或担心程序中非关键部分的速度，而这些对效率的追求在调试和维护时实际上会产生强烈的负面影响。我们应该忘记那些微小的效率提升，大约97%的时间：过早优化是万恶之源。但我们也不应放弃那关键的3%的优化机会。”

来源

牢记这条睿智的建议，以下是优化程序的推荐步骤：

1. 首先，设计和编写程序或库时应注重简洁和正确性。起初，不要花太多精力在性能上。
2. 使程序达到可运行状态，并（理想情况下）为代码库的关键部分开发单元测试。
3. 开发一个应用级性能基准测试。该基准测试应涵盖应用的性能关键部分，并执行一系列典型的任务，这些任务反映应用在生产环境中的使用方式。
4. 测量性能。
5. 将测得的性能与应用所需的性能标准进行比较。（避免使用不切实际、无法达到或无法量化的标准，如“尽可能快”。）
6. 如果你已经满足了标准，停止。你的工作已经完成。（任何进一步的努力可能都是浪费时间。）
7. 在运行性能基准测试时，对应用程序进行性能分析。
8. 检查性能分析结果，找出最大的（未优化的）“性能热点”；即应用程序似乎花费最多时间的代码部分。
9. 分析热点代码部分，尝试理解为什么它成为瓶颈，并思考如何使其更快。
10. 将其作为拟议的代码更改来实现，进行测试和调试。
11. 重新运行基准测试，查看代码更改是否提升了性能：
 - 如果是，则返回步骤4。
 - 如果不是，则放弃该更改并返回步骤9。如果没有进展，选择另一个热点进行关注。

最终，你会达到一个点，应用程序要么足够快，要么你已经考虑了所有重要的热点。此时你需要停止这种方法。如果某段代码消耗了（比如）整体时间的1%，那么即使提升50%，整体应用程序的速度也只会提升0.5%。

显然，热点优化有一个临界点，超过这个点就是浪费精力。如果达到这个点，你需要

Chapter 178: Java Performance Tuning

Section 178.1: An evidence-based approach to Java performance tuning

Donald Knuth is often quoted as saying this:

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. *We should forget about small efficiencies, say about 97% of the time*: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

source

Bearing that sage advice in mind, here is the recommended procedure for optimizing programs:

1. First of all, design and code your program or library with a focus on simplicity and correctness. To start with, don't spend much effort on performance.
2. Get it to a working state, and (ideally) develop unit tests for the key parts of the codebase.
3. Develop an application level performance benchmark. The benchmark should cover the performance critical aspects of your application, and should perform a range of tasks that are typical of how the application will be used in production.
4. Measure the performance.
5. Compare the measured performance against your criteria for how fast the application needs to be. (Avoid unrealistic, unattainable or unquantifiable criteria such as "as fast as possible".)
6. If you have met the criteria, STOP. Your job is done. (Any further effort is probably a waste of time.)
7. Profile the application while it is running your performance benchmark.
8. Examine the profiling results and pick the biggest (unoptimized) "performance hotspots"; i.e. sections of the code where the application seems to be spending the most time.
9. Analyse the hotspot code section to try to understand why it is a bottleneck, and think of a way to make it faster.
10. Implement that as a proposed code change, test and debug.
11. Rerun the benchmark to see if the code change has improved the performance:
 - If Yes, then return to step 4.
 - If No, then abandon the change and return to step 9. If you are making no progress, pick a different hotspot for your attention.

Eventually you will get to a point where the application is either fast enough, or you have considered all of the significant hotspots. At this point you need to stop this approach. If a section of code is consuming (say) 1% of the overall time, then even a 50% improvement is only going to make the application 0.5% faster overall.

Clearly, there is a point beyond which hotspot optimization is a waste of effort. If you get to that point, you need to

采取更激进的方法。例如：

- 查看核心算法的算法复杂度。
- 如果应用程序花费大量时间进行垃圾回收，请寻找减少对象
创建速率的方法。
- 如果应用程序的关键部分是CPU密集型且单线程的，请寻找并行的机会。
- 如果应用程序已经是多线程的，请寻找并发瓶颈。

但尽可能依赖工具和测量，而非直觉，来指导你的优化工作。

第178.2节：减少字符串数量

在Java中，创建许多不必要的字符串实例太“容易”了。这个以及其他原因可能导致你的程序有大量字符串，垃圾回收器忙于清理它们。

你可能创建字符串实例的一些方式：

```
myString += "foo";
```

或者更糟的是，在循环或递归中：

```
for (int i = 0; i < N; i++) {  
    myString += "foo" + i;  
}
```

问题在于每个`+`都会创建一个新的字符串（通常如此，因为新编译器会优化某些情况）。可以使用`StringBuilder`或`StringBuffer`进行可能的优化：

```
StringBuffer sb = new StringBuffer(myString);  
for (int i = 0; i < N; i++) {  
    sb.append("foo").append(i);  
}  
myString = sb.toString();
```

如果你经常构建长字符串（例如SQL），请使用字符串构建API。

其他需要考虑的事项：

- 减少使用`replace`、`substring`等方法。
- 避免在频繁访问的代码中使用`String.toArray()`。
- 对于将被过滤的日志打印（例如根据日志级别），不应生成日志（应提前检查日志级别）。
- 如有必要，使用像`this`这样的库。
- 如果变量以非共享方式（跨线程）使用，`StringBuilder`更好。

第178.3节：一般方法

互联网上充满了关于Java程序性能提升的技巧。也许最重要的技巧是意识。这意味着：

- 识别可能的性能问题和瓶颈。
- 使用分析和测试工具。
- 了解良好实践和不良实践。

take a more radical approach. For example:

- Look at the algorithmic complexity of your core algorithms.
- If the application is spending a lot of time garbage collection, look for ways to reduce the rate of object creation.
- If key parts of the application are CPU intensive and single-threaded, look for opportunities for parallelism.
- If the application is already multi-threaded, look for concurrency bottlenecks.

But wherever possible, rely on tools and measurement rather than instinct to direct your optimization effort.

Section 178.2: Reducing amount of Strings

In Java, it's too "easy" to create many String instances which are not needed. That and other reasons might cause your program to have lots of Strings that the GC is busy cleaning up.

Some ways you might be creating String instances:

```
myString += "foo";
```

Or worse, in a loop or recursion:

```
for (int i = 0; i < N; i++) {  
    myString += "foo" + i;  
}
```

The problem is that each`+`creates a new String (usually, since new compilers optimize some cases). A possible optimization can be made using`StringBuilder`or`StringBuffer`:

```
StringBuffer sb = new StringBuffer(myString);  
for (int i = 0; i < N; i++) {  
    sb.append("foo").append(i);  
}  
myString = sb.toString();
```

If you build long Strings often (SQLs for example), use a String building API.

Other things to consider:

- Reduce usage of`replace`,`substring`etc.
- Avoid`String.toArray()`, especially in frequently accessed code.
- Log prints which are destined to be filtered (due to log level for example) should not be generated (log level should be checked in advance).
- Use libraries like`this`if necessary.
- `StringBuilder`is better if the variable is used in a non-shared manner (across threads).

Section 178.3: General approach

The internet is packed with tips for performance improvement of Java programs. Perhaps the number one tip is awareness. That means:

- Identify possible performance problems and bottlenecks.
- Use analyzing and testing tools.
- Know good practices and bad practices.

如果是关于新系统或模块，第一点应在设计阶段完成。如果是遗留代码，分析和测试工具就派上用场了。分析JVM性能的最基本工具是JVisualVM，它包含在JDK中。

第三点主要涉及经验和广泛的研究，当然还有本页和其他页面上出现的原始技巧，比如this。

The first point should be done during the design stage if speaking about a new system or module. If speaking about legacy code, analyzing and testing tools come into the picture. The most basic tool for analyzing your JVM performance is JVisualVM, which is included in the JDK.

The third point is mostly about experience and extensive research, and of course raw tips that will show up on this page and others, like [this](#).

第179章：基准测试

在Java中编写性能基准测试并不像在开始和结束时获取System.currentTimeMillis()并计算差值那么简单。要编写有效的性能基准测试，应使用合适的工具。

第179.1节：简单的JMH示例

编写合适基准测试的工具之一是JMH。假设我们想比较在HashSet与TreeSet中搜索元素的性能。

将JMH引入项目最简单的方法是使用maven和shade插件。你也可以查看JMH

examples中的pom.xml文件。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.0.0</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <finalName>/benchmarks</finalName>
            <transformers>
              <transformer
                implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                <mainClass>org.openjdk.jmh.Main</mainClass>
              </transformer>
            </transformers>
            <filters>
              <filter>
                <artifact>*.*</artifact>
                <excludes>
                  <exclude>META-INF/*.SF</exclude>
                  <exclude>META-INF/*.DSA</exclude>
                  <exclude>META-INF/*.RSA</exclude>
                </excludes>
              </filter>
            </filters>
            <configuration>
              <excludes>
                <exclude>META-INF/*.SF</exclude>
                <exclude>META-INF/*.DSA</exclude>
                <exclude>META-INF/*.RSA</exclude>
              </excludes>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
      <artifactId>jmh-core</artifactId>
      <version>1.18</version>
    </dependency>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
```

Chapter 179: Benchmarks

Writing performance benchmarks in java is not as simple as getting `System.currentTimeMillis()` in the beginning and in the end and calculating the difference. To write valid performance benchmarks, one should use proper tools.

Section 179.1: Simple JMH example

One of the tools for writing proper benchmark tests is [JMH](#). Let's say we want to compare performance of searching an element in [HashSet](#) vs [TreeSet](#).

The easiest way to get JHM into your project - is to use maven and [shade](#) plugin. Also you can see `pom.xml` from [JHM examples](#).

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.0.0</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <finalName>/benchmarks</finalName>
            <transformers>
              <transformer
                implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                <mainClass>org.openjdk.jmh.Main</mainClass>
              </transformer>
            </transformers>
            <filters>
              <filter>
                <artifact>*.*</artifact>
                <excludes>
                  <exclude>META-INF/*.SF</exclude>
                  <exclude>META-INF/*.DSA</exclude>
                  <exclude>META-INF/*.RSA</exclude>
                </excludes>
              </filter>
            </filters>
            <configuration>
              <excludes>
                <exclude>META-INF/*.SF</exclude>
                <exclude>META-INF/*.DSA</exclude>
                <exclude>META-INF/*.RSA</exclude>
              </excludes>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
      <artifactId>jmh-core</artifactId>
      <version>1.18</version>
    </dependency>
    <dependency>
      <groupId>org.openjdk.jmh</groupId>
```

```

<artifactId>jmh-generator-annprocess</artifactId>
<version>1.18</version>
</dependency>
</dependencies>

```

接下来你需要编写基准测试类本身：

```

package benchmark;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import java.util.TreeSet;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
public class CollectionFinderBenchmarkTest {
    private static final int SET_SIZE = 10000;

    private Set<String> hashSet;
    private Set<String> treeSet;

    private String stringToFind = "8888";

    @Setup
    public void setupCollections() {
        hashSet = new HashSet<>(SET_SIZE);
        treeSet = new TreeSet<>();

        for (int i = 0; i < SET_SIZE; i++) {
            final String value = String.valueOf(i);
            hashSet.add(value);
            treeSet.add(value);
        }
    }

    stringToFind = String.valueOf(new Random().nextInt(SET_SIZE));
}

    @Benchmark
    @BenchmarkMode(mode = Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testHashSet(Blackhole blackhole) {
        blackhole.consume(hashSet.contains(stringToFind));
    }

    @Benchmark
    @BenchmarkMode(mode = Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testTreeSet(Blackhole blackhole) {
        blackhole.consume(treeSet.contains(stringToFind));
    }
}

```

请记住这个blackhole.consume()，我们稍后会回到它。我们还需要一个用于运行基准测试的主类：

```
package benchmark;
```

```

<artifactId>jmh-generator-annprocess</artifactId>
<version>1.18</version>
</dependency>
</dependencies>

```

After this you need to write benchmark class itself:

```

package benchmark;

import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;

import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import java.util.TreeSet;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
public class CollectionFinderBenchmarkTest {
    private static final int SET_SIZE = 10000;

    private Set<String> hashSet;
    private Set<String> treeSet;

    private String stringToFind = "8888";

    @Setup
    public void setupCollections() {
        hashSet = new HashSet<>(SET_SIZE);
        treeSet = new TreeSet<>();

        for (int i = 0; i < SET_SIZE; i++) {
            final String value = String.valueOf(i);
            hashSet.add(value);
            treeSet.add(value);
        }
    }

    stringToFind = String.valueOf(new Random().nextInt(SET_SIZE));
}

    @Benchmark
    @BenchmarkMode(mode = Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testHashSet(Blackhole blackhole) {
        blackhole.consume(hashSet.contains(stringToFind));
    }

    @Benchmark
    @BenchmarkMode(mode = Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public void testTreeSet(Blackhole blackhole) {
        blackhole.consume(treeSet.contains(stringToFind));
    }
}

```

Please keep in mind this blackhole.consume()，we'll get back to it later. Also we need main class for running benchmark:

```
package benchmark;
```

```

import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

public class BenchmarkMain {
    public static void main(String[] args) throws RunnerException {
        final Options options = new OptionsBuilder()
.include(CollectionFinderBenchmarkTest.class.getSimpleName())
        .forks(1)
.build();

        new Runner(options).run();
    }
}

```

一切就绪。我们只需运行 `mvn package` (它将在你的 `/target` 文件夹中创建 `benchmarks.jar`) , 然后运行我们的基准测试 :

```
java -cp target/benchmarks.jar benchmark.BenchmarkMain
```

经过一些预热和计算迭代后, 我们将得到结果 :

```

# 运行完成。总时间: 00:01:21

Benchmark          模式 次数 分数   误差 单位
CollectionFinderBenchmarkTest.testHashSet 平均时间  20  9.940 ± 0.270 纳秒/操作
CollectionFinderBenchmarkTest.testTreeSet  平均时间  20  98.858 ± 13.743 纳秒/操作

```

关于那个 `blackhole.consume()`。如果你的计算没有改变应用程序的状态, java 很可能会忽略它。所以, 为了避免这种情况, 你可以让你的基准测试方法返回某个值, 或者使用 `Blackhole` 对象来消费它。

你可以在 [Aleksey Shipilëv 的博客](#)、[Jacob Jenkov 的博客](#)

以及 [java-performance](#) 博客中找到更多关于编写合适基准测试的信息 : 1, 2.

```

import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

public class BenchmarkMain {
    public static void main(String[] args) throws RunnerException {
        final Options options = new OptionsBuilder()
        .include(CollectionFinderBenchmarkTest.class.getSimpleName())
        .forks(1)
        .build();

        new Runner(options).run();
    }
}

```

And we're all set. We just need to run `mvn package` (it will create `benchmarks.jar` in your `/target` folder) and run our benchmark test:

```
java -cp target/benchmarks.jar benchmark.BenchmarkMain
```

And after some warmup and calculation iterations, we will have our results:

```

# Run complete. Total time: 00:01:21

Benchmark          Mode Cnt Score  Error Units
CollectionFinderBenchmarkTest.testHashSet avg  20  9.940 ± 0.270 ns/op
CollectionFinderBenchmarkTest.testTreeSet  avg  20  98.858 ± 13.743 ns/op

```

About that `blackhole.consume()`. If your calculations do not change the state of your application, java will most likely just ignore it. So, in order to avoid it, you can either make your benchmark methods return some value, or use `Blackhole` object to consume it.

You can find more information about writing proper benchmarks in [Aleksey Shipilëv's blog](#), in [Jacob Jenkov's blog](#) and in [java-performance](#) blog: 1, 2.

第180章：文件上传到 AWS

使用 spring rest API 上传文件到 AWS s3 存储桶。

第180.1节：上传文件到 s3 存储桶

这里我们将创建一个 rest API，从前端接收文件对象作为 multipart 参数，并使用 java rest API 上传到 S3 存储桶。

需求：用于上传文件的 S3 存储桶的密钥和访问密钥。

代码：DocumentController.java

```
@RestController
@RequestMapping("/api/v2")
公共类 DocumentController {

    私有静态 字符串 bucketName = "pharmerz-chat";
    // 私有静态字符串 keyName = "Pharmerz"+ UUID.randomUUID();

    @RequestMapping(value = "/upload", method = RequestMethod.POST, consumes =
    MediaType.MULTIPART_FORM_DATA)
    公共 URL uploadFileHandler(@RequestParam("name") 字符串 name,
                                @RequestParam("file") MultipartFile 文件) 抛出 IOException {
        **** 打印所有可能的@RequestParam参数 ****
        System.out.println("*****");
        System.out.println("file.getOriginalFilename() " + file.getOriginalFilename());
        System.out.println("file.getContentType() " + file.getContentType());
        System.out.println("file.getInputStream() " + file.getInputStream());
        System.out.println("file.toString() " + file.toString());
        System.out.println("file.getSize() " + file.getSize());
        System.out.println("name " + name);
        System.out.println("file.getBytes() " + file.getBytes());
        System.out.println("file.hashCode() " + file.hashCode());
        System.out.println("file.getClass() " + file.getClass());
        System.out.println("file.isEmpty() " + file.isEmpty());

        **** 传递给s3桶put Object的参数 ****
        InputStream is = file.getInputStream();
        String keyName = file.getOriginalFilename();

        // Aws的凭证
        AWSCredentials credentials = new BasicAWSCredentials("AKIA*****",
        "zr*****");

        ***** DocumentController.uploadfile(credentials); *****

        AmazonS3 s3client = new AmazonS3Client(credentials);
        try {
            System.out.println("从文件上传新对象到S3");
            //File file = new File(awsuploadfile);
            s3client.putObject(new PutObjectRequest(
                bucketName, keyName, is, new ObjectMetadata()));
        }
    }
}
```

Chapter 180: FileUpload to AWS

Upload File to AWS s3 bucket using spring rest API.

Section 180.1: Upload file to s3 bucket

Here we will create a rest API which will take file object as a multipart parameter from front end and upload it to S3 bucket using java rest API.

Requirement: secrete key and Access key for s3 bucket where you wanna upload your file.

code: DocumentController.java

```
@RestController
@RequestMapping("/api/v2")
public class DocumentController {

    private static String bucketName = "pharmerz-chat";
    // private static String keyName      = "Pharmerz"+ UUID.randomUUID();

    @RequestMapping(value = "/upload", method = RequestMethod.POST, consumes =
    MediaType.MULTIPART_FORM_DATA)
    public URL uploadFileHandler(@RequestParam("name") String name,
                                @RequestParam("file") MultipartFile file) throws IOException {
        **** Printing all the possible parameter from @RequestParam ****
        System.out.println("*****");
        System.out.println("file.getOriginalFilename() " + file.getOriginalFilename());
        System.out.println("file.getContentType() " + file.getContentType());
        System.out.println("file.getInputStream() " + file.getInputStream());
        System.out.println("file.toString() " + file.toString());
        System.out.println("file.getSize() " + file.getSize());
        System.out.println("name " + name);
        System.out.println("file.getBytes() " + file.getBytes());
        System.out.println("file.hashCode() " + file.hashCode());
        System.out.println("file.getClass() " + file.getClass());
        System.out.println("file.isEmpty() " + file.isEmpty());

        **** Parameters to be pass to s3 bucket put Object ****
        InputStream is = file.getInputStream();
        String keyName = file.getOriginalFilename();

        // Credentials for Aws
        AWSCredentials credentials = new BasicAWSCredentials("AKIA*****",
        "zr*****");

        ***** DocumentController.uploadfile(credentials); *****

        AmazonS3 s3client = new AmazonS3Client(credentials);
        try {
            System.out.println("Uploading a new object to S3 from a file\n");
            //File file = new File(awsuploadfile);
            s3client.putObject(new PutObjectRequest(
                bucketName, keyName, is, new ObjectMetadata()));
        }
    }
}
```

```

    URL url = s3client.generatePresignedUrl(bucketName, keyName,
Date.from(Instant.now().plus(5, ChronoUnit.MINUTES)));
    // URL url=s3client.generatePresignedUrl(bucketName,keyName,
Date.from(Instant.now().plus(5, ChronoUnit.))));
    System.out.println("*****");
    System.out.println(url);

    return url;

} catch (AmazonServiceException ase) {
    System.out.println("捕获了一个 AmazonServiceException, " +
        "这意味着您的请求已发送到 " +
        "Amazon S3, 但由于某种原因被拒绝并返回错误响应。" +
        ".");
    System.out.println("错误信息: " + ase.getMessage());
    System.out.println("HTTP 状态码: " + ase.getStatusCode());
    System.out.println("AWS 错误代码: " + ase.getErrorCode());
    System.out.println("错误类型: " + ase.getErrorType());
    System.out.println("请求 ID: " + ase.getRequestId());
} catch (AmazonClientException ace) {
    System.out.println("捕获了一个 AmazonClientException, " +
        "这意味着客户端在尝试 " +
        "与 S3 通信时遇到了内部错误, " +
        "例如无法访问网络。" +
        ".");
    System.out.println("错误信息: " + ace.getMessage());
}
}

return null;
}

```

前端功能

```

var form = new FormData();
form.append("file", "image.jpeg");

var settings = {
"async": true,
"crossDomain": true,
"url": "http://url/",
"method": "POST",
"headers": {
    "cache-control": "no-cache"
},
"processData": false,
"contentType": false,
"mimeType": "multipart/form-data",
"data": form
};

$.ajax(settings).done(function (response) {
    console.log(response);
});

```

```

    URL url = s3client.generatePresignedUrl(bucketName, keyName,
Date.from(Instant.now().plus(5, ChronoUnit.MINUTES)));
    // URL url=s3client.generatePresignedUrl(bucketName,keyName,
Date.from(Instant.now().plus(5, ChronoUnit.))));
    System.out.println("*****");
    System.out.println(url);

    return url;

} catch (AmazonServiceException ase) {
    System.out.println("Caught an AmazonServiceException, which " +
        "means your request made it " +
        "to Amazon S3, but was rejected with an error response" +
        " for some reason.");
    System.out.println("Error Message: " + ase.getMessage());
    System.out.println("HTTP Status Code: " + ase.getStatusCode());
    System.out.println("AWS Error Code: " + ase.getErrorCode());
    System.out.println("Error Type: " + ase.getErrorType());
    System.out.println("Request ID: " + ase.getRequestId());
} catch (AmazonClientException ace) {
    System.out.println("Caught an AmazonClientException, which " +
        "means the client encountered " +
        "an internal error while trying to " +
        "communicate with S3, " +
        "such as not being able to access the network.");
    System.out.println("Error Message: " + ace.getMessage());
}
}

return null;
}

```

Front end Function

```

var form = new FormData();
form.append("file", "image.jpeg");

var settings = {
"async": true,
"crossDomain": true,
"url": "http://url/",
"method": "POST",
"headers": {
    "cache-control": "no-cache"
},
"processData": false,
"contentType": false,
"mimeType": "multipart/form-data",
"data": form
};

$.ajax(settings).done(function (response) {
    console.log(response);
});

```

第181章：AppDynamics与TIBCO BusinessWorks的监控，便于集成

由于AppDynamics旨在提供一种衡量应用性能的方法，应用的开发速度和交付（部署）是使DevOps工作真正成功的关键因素。使用AppDynamics监控TIBCO BW应用通常简单且不耗时，但在部署大量应用时，快速监控是关键。本指南展示了如何在不修改每个应用的情况下，一步完成所有BW应用的监控部署。

第181.1节：Appdynamics中所有带宽应用的一步仪表化示例

1. 找到并打开您的 TIBCO BW bwengineтра文件，通常位于 TIBCO_HOME/bw/5.12/bin/bwengine.tra (Linux 环境)
2. 查找以下行：

***** 公共变量。仅修改这些。*****

3. 在该部分后添加以下行 tibco.deployment=%tibco.deployment%4. 到文件末尾添加（根

据需要用您自己的值替换问号，或删除不适用的标志）：java.extended.properties=-javaagent:/opt/appd/current/appagent/javaagent.jar -
Dappdynamics.http.proxyHost=? -Dappdynamics.http.proxyPort=? -Dappdynamics.agent.applicationName=?
-Dappdynamics.agent.tierName=? -Dappdynamics.agent.nodeName=%tibco.deployment% -
Dappdynamics.controller.ssl.enabled=? -Dappdynamics.controller.sslPort=? -Dappdynamics.agent.logs.dir=? -
Dappdynamics.agent.runtime.dir=? -Dappdynamics.controller.hostName=? -Dappdynamics.controller.port=?
-Dappdynamics.agent.accountName=? -Dappdynamics.agent.accountAccessKey=?

5. 保存文件并重新部署。现在所有应用程序应在部署时自动进行监控。

Chapter 181: AppDynamics and TIBCO BusinessWorks Instrumentation for Easy Integration

As AppDynamics aims to provide a way to measure application performance, speed of development, delivery (deployment) of applications is an essential factor in making DevOps efforts a true success. Monitoring a TIBCO BW application with AppD is generally simple and not time consuming but when deploying large sets of applications rapid instrumentation is key. This guide shows how to instrument all of your BW applications in a single step without modifying each application before deploying.

Section 181.1: Example of Instrumentation of all BW Applications in a Single Step for Appdynamics

1. Locate and open your TIBCO BW bwengine.tra file typically under TIBCO_HOME/bw/5.12/bin/bwengine.tra (Linux environment)
2. Look for the line that states:
***** Common variables. Modify these only. *****
3. Add the following line right after that section tibco.deployment=%tibco.deployment%
4. Go to the end of the file and add (replace ? with your own values as needed or remove the flag that does not apply): java.extended.properties=-javaagent:/opt/appd/current/appagent/javaagent.jar -
Dappdynamics.http.proxyHost=? -Dappdynamics.http.proxyPort=? -Dappdynamics.agent.applicationName=?
-Dappdynamics.agent.tierName=? -Dappdynamics.agent.nodeName=%tibco.deployment% -
Dappdynamics.controller.ssl.enabled=? -Dappdynamics.controller.sslPort=? -Dappdynamics.agent.logs.dir=? -
Dappdynamics.agent.runtime.dir=? -Dappdynamics.controller.hostName=? -Dappdynamics.controller.port=?
-Dappdynamics.agent.accountName=? -Dappdynamics.agent.accountAccessKey=?
5. Save file and redeploy. All your applications should now be instrumented automatically at deployment time.

附录 A：安装 Java（标准版）

本说明文档页面提供了在Windows、Linux和macOS计算机上安装Java标准版的说明。

第A.1节：在

Windows安装后设置%PATH%和%JAVA_HOME%

假设条件：

- 已安装Oracle JDK。
- JDK已安装到默认目录。

设置步骤

1. 打开Windows资源管理器。
2. 在左侧导航窗格中右键点击此电脑（旧版本Windows为计算机）。有一种在当前Windows版本中无需使用资源管理器的更快捷方式：只需按 **Win + 暂停**
3. 在新打开的控制面板窗口中，左键点击位于左上角的高级系统设置角落。这将打开系统属性窗口。



或者，在运行 (**Win + R**) 回车 **.**

4. 在系统属性的高级选项卡中，选择右下角的 **环境变量...** 按钮

5. 通过点击添加新系统变量 **新建...** 系统变量中的按钮，变量名为JAVA_HOME，其值为JDK安装目录的路径。输入这些值后，按下 **确定**。

Appendix A: Installing Java (Standard Edition)

This documentation page gives access to instructions for installing java standard edition on Windows, Linux, and macOS computers.

Section A.1: Setting %PATH% and %JAVA_HOME% after installing on Windows

Assumptions:

- An Oracle JDK has been installed.
- The JDK was installed to the default directory.

Setup steps

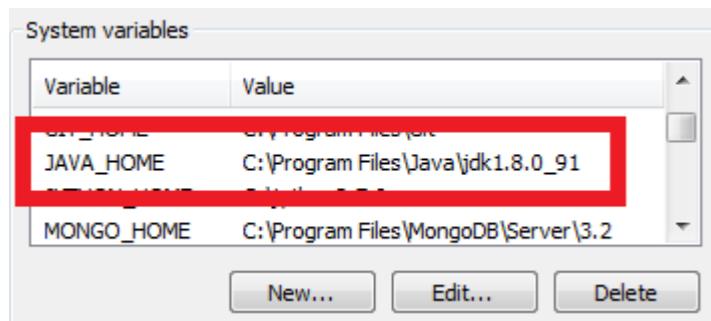
1. Open Windows Explorer.
2. In the navigation pane on the left right click on *This PC* (or *Computer* for older Windows versions). There is a shorter way without using the explorer in actual Windows versions: Just press **Win + Pause**
3. In the newly opened Control Panel window, left click *Advanced System Settings* which should be in the top left corner. This will open the *System Properties* window.



Alternatively, type **SystemPropertiesAdvanced** (case insensitive) in the Run (**Win + R**), and hit **Enter**.

4. In the *Advanced* tab of *System Properties* select the **Environment Variables...** button in the lower right corner of the window.

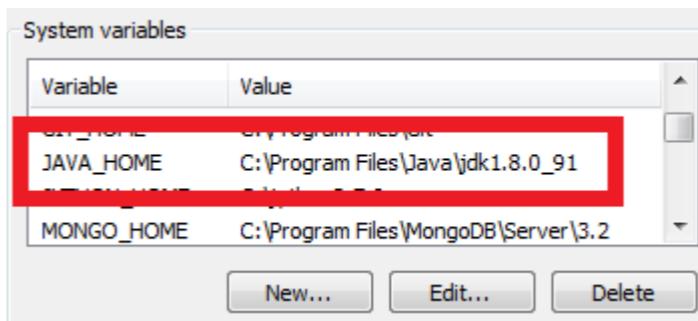
5. Add a **New System Variable** by clicking the **New...** button in *System Variables* with the name **JAVA_HOME** and whose value is the path to the directory where the JDK was installed. After entering these values, press **OK**.



6. 向下滚动系统变量列表，选择Path变量。

7. 注意：Windows依赖Path来查找重要程序。如果其中任何部分被删除，Windows可能无法正常运行。必须修改它以允许Windows运行JDK。考虑到这一点，选中Path变量后，点击“编辑...”按钮。添加将 %JAVA_HOME%\bin; 添加到 Path 变量的开头。

最好将其附加在行首，因为Oracle的软件会在Path中注册它们自己的Java版本——如果它出现在Oracle声明之后，将导致你的版本被忽略。



6. Scroll down the list of *System Variables* and select the Path variable.

7. **CAUTION:** Windows relies on Path to find important programs. If any or all of it is removed, Windows may not be able to function properly. It must be modified to allow Windows to run the JDK. With this in mind ,click the "Edit..." button with the Path variable selected. Add %JAVA_HOME%\bin; to the beginning of the Path variable.

It is better to append at the beginning of the line because Oracle's software used to register their own version of Java in Path - This will cause your version to be ignored if it occurs after Oracle's declaration.

检查你的操作

1. 通过点击开始，然后输入cmd并按Enter打开命令提示符。
2. 在提示符中输入javac -version。如果成功，将会显示JDK的版本在屏幕上。

注意：如果需要重试，请关闭提示符后再检查你的操作。这将强制Windows获取Path的新版本。

A.2节：在Linux上安装Java JDK

使用包管理器

OpenJDK或Oracle的JDK和/或JRE版本可以通过大多数主流Linux发行版的包管理器安装。（可用的选项取决于发行版。）

一般来说，步骤是打开终端窗口并运行下面显示的命令。（假设你有足够的权限以“root”用户身份运行命令.....这也是sudo命令的作用。如果没有，请联系你的系统管理员。）

推荐使用包管理器，因为它（通常）可以更方便地保持您的Java安装为最新版本。

apt-get, 基于Debian的Linux发行版 (Ubuntu等)

以下说明将安装Oracle Java 8：

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

Check your work

1. Open the command prompt by clicking Start then typing cmd and pressing Enter.
2. Enter javac -version into the prompt. If it was successful, then the version of the JDK will be printed to the screen.

Note: If you have to try again, close the prompt before checking your work. This will force windows to get the new version of Path.

Section A.2: Installing a Java JDK on Linux

Using the Package Manager

JDK and/or JRE releases for OpenJDK or Oracle can be installed using the package manager on most mainstream Linux distribution. (The choices that are available to you will depend on the distro.)

As a general rule, the procedure is to open terminal window and run the commands shown below. (It is assumed that you have sufficient access to run commands as the "root" user ... which is what the sudo command does. If you do not, then please talk to your system's administrators.)

Using the package manager is recommended because it (generally) makes it easier to keep your Java installation up to date.

apt-get, Debian based Linux distributions (Ubuntu, etc)

The following instructions will install Oracle Java 8:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

注意：要自动设置Java 8环境变量，您可以安装以下软件包：

```
$ sudo apt-get install oracle-java8-set-default
```

创建一个.deb文件

如果您更愿意从Oracle下载的.tar.gz文件自己创建.deb文件，请执行以下操作
(假设您已将.tar.gz下载到./<jdk>.tar.gz)：

```
$ sudo apt-get install java-package # 可能在默认仓库中不可用
$ make-jpkg ./<jdk>.tar.gz          # 不应以root身份运行
$ sudo dpkg -i *j2sdk*.deb
```

注意：这要求输入文件为".tar.gz"格式。

slackpkg, 基于Slackware的Linux发行版

```
sudo slapt-get install default-jdk
```

yum, RedHat、CentOS等

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

dnf, Fedora

在最近的Fedora版本中，yum 已被 dnf 取代。

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

在最近的Fedora版本中，没有用于安装Java 7及更早版本的软件包。

pacman, 基于Arch的Linux发行版

```
sudo pacman -S jdk8-openjdk
```

如果以root用户身份运行，则不需要使用 sudo。

Gentoo Linux

[Gentoo Java 指南](#)由Gentoo Java团队维护，并保持更新的wiki页面，包括所需的正确 portage软件包和USE标志。

在Redhat、CentOS、Fedora上安装Oracle JDK

从Oracle JDK或JRE tar.gz 文件安装JDK。

1. 从Oracle Java

[下 载站点 下载所需版本的Oracle归档文件 \("tar.gz"\)。](#)

2. 切换目录到你想放置安装的位置；

3. 解压归档文件；例如

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

Note: To automatically set up the Java 8 environment variables, you can install the following package:

```
$ sudo apt-get install oracle-java8-set-default
```

Creating a .deb file

If you prefer to create the .deb file yourself from the .tar.gz file downloaded from Oracle, do the following (assuming you've downloaded the .tar.gz to ./<jdk>.tar.gz):

```
$ sudo apt-get install java-package # might not be available in default repos
$ make-jpkg ./<jdk>.tar.gz          # should not be run as root
$ sudo dpkg -i *j2sdk*.deb
```

Note: This expects the input to be provided as a ".tar.gz" file.

slackpkg, Slackware based Linux distributions

```
sudo slapt-get install default-jdk
```

yum, RedHat, CentOS, etc

```
sudo yum install java-1.8.0-openjdk-devel.x86_64
```

dnf, Fedora

On recent Fedora releases, yum has been superseded by dnf.

```
sudo dnf install java-1.8.0-openjdk-devel.x86_64
```

In recent Fedora releases, there are no packages for installing Java 7 and earlier.

pacman, Arch based Linux distributions

```
sudo pacman -S jdk8-openjdk
```

Using sudo is not required if you're running as the root user.

Gentoo Linux

The [Gentoo Java guide](#) is maintained by the Gentoo Java team and keeps an updated wiki page including the correct portage packages and USE flags needed.

Installing Oracle JDks on Redhat, CentOS, Fedora

Installing JDK from an Oracle JDK or JRE tar.gz file.

1. Download the appropriate Oracle archive ("tar.gz") file for the desired release from the [Oracle Java downloads site](#).

2. Change directory to the place where you want to put the installation;

3. Decompress the archive file; e.g.

```
tar xzvf jdk-8u67-linux-x64.tar.gz
```

从 Oracle Java RPM 文件安装。

1. 从 Oracle Java 下载站点获取所需版本的 RPM 文件。
2. 使用 rpm 命令安装。例如：

```
$ sudo rpm -ivh jdk-8u67-linux-x64.rpm
```

A.3 节：在 macOS 上安装 Java JDK

Oracle Java 7 和 Java 8

macOS 版的 Java 7 和 Java 8 可从 Oracle 获取。该 Oracle 页面解答了许多关于 Mac 版 Java 的问题。请注意，Apple 出于安全原因已禁用 7u25 之前的 Java 7 版本。

一般来说，Oracle Java（版本 7 及以后）需要基于 Intel 的 Mac，运行 macOS 10.7.3 或更高版本。

Oracle Java 的安装

macOS 版的 Java 7 和 8 JDK 及 JRE 安装程序可从 Oracle'的网站下载：

- [Java 8 - Java SE 下载](#)
- [Java 7 - Oracle Java 存档](#)

下载相关软件包后，双击该软件包并按照正常安装流程进行安装。
JDK 应安装在以下位置：

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

其中对应已安装的版本。

命令行切换

安装 Java 后，安装的版本会自动设置为默认版本。要切换不同版本，请使用：

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #或者 1.7 或 1.8
```

以下函数可以添加到 `~/.bash_profile`（如果你使用默认的 Bash shell）以便使用：

```
function java_version {
    echo 'java -version';
}

function java_set {
    if [[ $1 == "6" ]]
    then
        export JAVA_HOME='/usr/libexec/java_home -v 1.6';
        echo "设置 Java 版本为 6..."
        echo "$JAVA_HOME"
    elif [[ $1 == "7" ]]
    then
        export JAVA_HOME='/usr/libexec/java_home -v 1.7';
        echo "设置 Java 版本为 7..."
        echo "$JAVA_HOME"
    elif [[ $1 == "8" ]]
    then
```

Installing from an Oracle Java RPM file.

1. Retrieve the required RPM file for the desired release from the [Oracle Java downloads site](#).
2. Install using the rpm command. For example:

```
$ sudo rpm -ivh jdk-8u67-linux-x64.rpm
```

Section A.3: Installing a Java JDK on macOS

Oracle Java 7 and Java 8

Java 7 and Java 8 for macOS are available from Oracle. This Oracle page answers a lot of questions about Java for Mac. Note that Java 7 prior to 7u25 have been disabled by Apple for security reasons.

In general, Oracle Java (Version 7 and later) requires an Intel-based Mac running macOS 10.7.3 or later.

Installation of Oracle Java

Java 7 & 8 JDK and JRE installers for macOS can be downloaded from Oracle's website:

- [Java 8 - Java SE Downloads](#)
- [Java 7 - Oracle Java Archive](#)

After downloading the relevant package, double click on the package and go through the normal installation process. A JDK should get installed here:

```
/Library/Java/JavaVirtualMachines/<version>.jdk/Contents/Home
```

where corresponds to the installed version.

Command-Line Switching

When Java is installed, the installed version is automatically set as the default. To switch between different, use:

```
export JAVA_HOME=/usr/libexec/java_home -v 1.6 #or 1.7 or 1.8
```

The following functions can be added to the `~/.bash_profile` (If you use the default Bash shell) for ease of use:

```
function java_version {
    echo 'java -version';
}

function java_set {
    if [[ $1 == "6" ]]
    then
        export JAVA_HOME='/usr/libexec/java_home -v 1.6';
        echo "Setting Java to version 6..."
        echo "$JAVA_HOME"
    elif [[ $1 == "7" ]]
    then
        export JAVA_HOME='/usr/libexec/java_home -v 1.7';
        echo "Setting Java to version 7..."
        echo "$JAVA_HOME"
    elif [[ $1 == "8" ]]
    then
```

```

export JAVA_HOME='/usr/libexec/java_home -v 1.8';
echo "设置 Java 版本为 8..."
echo "$JAVA_HOME"
fi
}

```

macOS 上的 Apple Java 6

在较旧版本的 macOS (10.11 El Capitan 及更早版本) 中, Apple 预装了 Java 6。如果已安装,可以在以下位置找到它 :

`/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home`

请注意, Java 6 已经过了生命周期终止期, 因此建议升级到更新的版本。关于重新安装 Apple Java 6 的更多信息, 请参见 Oracle 网站。

附录 A.4 : 在 Windows 上安装 Java JDK 或 JRE

Windows 平台仅提供 Oracle 的 JDK 和 JRE。安装过程非常简单 :

1. 访问 Oracle Java 下载页面 :
2. 点击 JDK 按钮、JRE 按钮或 Server JRE 按钮。请注意, 开发 Java 需要 JDK。关于 JDK 和 JRE 的区别, 请参见 [here](#)
3. 向下滚动到您想下载的版本。(一般来说, 推荐使用最新版本。)
4. 选择“接受许可协议”单选按钮。
5. 下载 Windows x86 (32 位) 或 Windows x64 (64 位) 安装程序。
6. 以您 Windows 版本的正常方式运行安装程序。

在 Windows 上使用命令提示符安装 Java 的另一种方法是使用 Chocolatey :

1. 从 <https://chocolatey.org/> 安装 Chocolatey
2. 打开一个命令提示符窗口, 例如按下 **Win + R** 然后在“运行”窗口中输入“cmd”, 接着按下回车。
3. 在你的命令提示符窗口中, 运行以下命令以下载并安装 Java 8 JDK :

C:\> choco install jdk8

使用便携版本快速启动

有些情况下, 你可能想在权限受限的系统 (如虚拟机) 上安装 JDK/JRE, 或者你想安装并使用多个版本或架构 (x64/x86) 的 JDK/JRE。直到下载安装程序 (.EXE) 之前, 步骤都是相同的。之后的步骤如下 (这些步骤适用于 JDK/JRE 7 及以上版本, 旧版本的文件夹和文件名称略有不同) :

1. 将文件移动到你希望 Java 二进制文件永久存放的合适位置。
2. 安装 7-Zip 或其便携版本, 如果你的权限有限。
3. 使用 7-Zip, 将 Java 安装程序 EXE 中的文件解压到该位置。
4. 在资源管理器中的文件夹内按住Shift并右键点击以打开命令提示符, 或导航到该文件夹从任何地方到那个位置。

```

export JAVA_HOME='/usr/libexec/java_home -v 1.8';
echo "Setting Java to version 8..."
echo "$JAVA_HOME"
fi
}

```

Apple Java 6 on macOS

On older versions of macOS (10.11 El Capitan and earlier), Apple's release of Java 6 comes pre-installed. If installed, it can be found at this location:

`/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home`

Note that Java 6 passed its end-of-life long ago, so upgrading to a newer version is recommended. There is more information on reinstalling Apple Java 6 on the Oracle website.

Section A.4: Installing a Java JDK or JRE on Windows

Only Oracle JDKs and JREs are available for Windows platforms. The installation procedure is straight-forward:

1. Visit the Oracle Java [Downloads page](#):
2. Click on either the JDK button, the JRE button or the Server JRE button. Note that to develop using Java you need JDK. To know the difference between JDK and JRE, see [here](#)
3. Scroll down to the version you want to download. (Generally speaking, the most recent one is recommended.)
4. Select the "Accept License Agreement" radio button.
5. Download the Windows x86 (32 bit) or Windows x64 (64 bit) installer.
6. Run the installer ... in the normal way for your version of Windows.

An alternate way to install Java on Windows using the command prompt is to use Chocolatey:

1. Install Chocolatey from <https://chocolatey.org/>
2. Open a cmd instance, for example hit **Win + R** and then type "cmd" in the "Run" window followed by an enter.
3. In your cmd instance, run the following command to download and install a Java 8 JDK:

C:\> choco install jdk8

Getting up and running with portable versions

There are instances where you might want to install JDK/JRE on a system with limited privileges like a VM or you might want to install and use multiple versions or architectures (x64/x86) of JDK/JRE. The steps remain same till the point you download the installer (.EXE). The steps after that are as follows (The steps are applicable for JDK/JRE 7 and above, for older versions they are slightly different in the names of folders and files):

1. Move the file to an appropriate location where you would want your Java binaries to reside permanently.
2. Install 7-Zip or its portable version if you have limited privileges.
3. With 7-Zip, extract the files from the Java installer EXE to the location.
4. Open up command prompt there by holding Shift and Right-Clicking in the folder in explorer or navigate to that location from anywhere.

5. 导航到新创建的文件夹。假设文件夹名称是jdk-7u25-windows-x64。所以输入cd jdk-7u25-windows-x64。然后按顺序输入以下命令：

```
cd .\src\JAVA_CAB10  
extrac32 111
```

6. 这将在该位置创建一个tools.zip文件。使用7-Zip解压tools.zip，以便其中的文件现在在同一目录下的tools中创建。

7. 现在在已打开的命令提示符中执行以下命令：

```
cd 工具  
for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
```

8. 等待命令完成。将tools目录下的内容复制到你希望放置二进制文件的位置。

这样，你就可以同时安装任何需要安装的JDK/JRE版本。

原始帖子：<http://stackoverflow.com/a/6571736/1448252>

附录A.5：使用alternatives在Linux上配置和切换Java版本

使用Alternatives

许多Linux发行版使用alternatives命令来切换不同版本的命令。你可以用它来切换机器上安装的不同版本的Java。

1. 在命令行中，将\$JDK设置为新安装的JDK的路径名；例如

```
$ JDK=/Data/jdk1.8.0_67
```

2. 使用 alternatives --install 将 Java SDK 中的命令添加到 alternatives：

```
$ sudo alternatives --install /usr/bin/java java $JDK/bin/java 2  
$ sudo alternatives --install /usr/bin/javac javac $JDK/bin/javac 2  
$ sudo alternatives --install /usr/bin/jar jar $JDK/bin/jar 2
```

等等。

现在你可以通过以下方式在不同版本的 Java 命令之间切换：

```
$ sudo alternatives --config javac  
有 1 个程序提供 'javac'。  
选择 命令  
-----  
*+ 1      /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.101-1.b14.fc23.x86_64/bin/javac
```

5. Navigate to the newly created folder. Let's say the folder name is jdk-7u25-windows-x64. So type `cd jdk-7u25-windows-x64`. Then type the following commands in order :

```
cd .\src\JAVA_CAB10  
extrac32 111
```

6. This will create a `tools.zip` file in that location. Extract the `tools.zip` with 7-Zip so that the files inside it are now created under `tools` in the same directory.

7. Now execute these commands on the already opened command prompt :

```
cd tools  
for /r %x in (*.pack) do .\bin\unpack200 -r "%x" "%~dx%~px%~nx.jar"
```

8. Wait for the command to complete. Copy the contents of `tools` to the location where you want your binaries to be.

This way, you can install any versions of JDK/JRE you need to be installed simultaneously.

Original post : <http://stackoverflow.com/a/6571736/1448252>

Section A.5: Configuring and switching Java versions on Linux using alternatives

Using Alternatives

Many Linux distributions use the `alternatives` command for switching between different versions of a command. You can use this for switching between different versions of Java installed on a machine.

1. In a command shell, set \$JDK to the pathname of a newly installed JDK; e.g.

```
$ JDK=/Data/jdk1.8.0_67
```

2. Use `alternatives --install` to add the commands in the Java SDK to alternatives:

```
$ sudo alternatives --install /usr/bin/java java $JDK/bin/java 2  
$ sudo alternatives --install /usr/bin/javac javac $JDK/bin/javac 2  
$ sudo alternatives --install /usr/bin/jar jar $JDK/bin/jar 2
```

And so on.

Now you can switch between different versions of a Java command as follows:

```
$ sudo alternatives --config javac  
There is 1 program that provides 'javac'.  
Selection    Command  
-----  
*+ 1          /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.101-1.b14.fc23.x86_64/bin/javac
```

按回车键保持当前选择[+]，或输入选择编号：2
\$

有关使用alternatives的更多信息，请参阅[alternatives\(8\)手册条目](#)。

基于Arch的安装

基于Arch Linux的安装附带命令archlinux-java用于切换Java版本。

列出已安装的环境

```
$ archlinux-java status
可用的Java环境：
java-7-openjdk (默认)
java-8-openjdk/jre
```

切换当前环境

```
# archlinux-java set <JAVA_ENV_NAME>
```

例如：

```
# archlinux-java 设置 java-8-openjdk/jre
```

更多信息可在Arch Linux Wiki中找到 [_____](#)

第 A.6 节：我需要什么进行 Java 开发

JDK 安装和文本编辑器是进行 Java 开发的最低要求。（最好有一个能进行 Java 语法高亮的文本编辑器，但没有也可以。）

然而，对于严肃的开发工作，建议您还使用以下工具：

- Java 集成开发环境（IDE），如 Eclipse、IntelliJ IDEA 或 NetBeans
- Java 构建工具，如 Ant、Gradle 或 Maven
- 用于管理代码库的版本控制系统（包括适当的备份和异地复制）
- 测试工具和持续集成（CI）工具

第 A.7 节：选择合适的 Java SE 版本

自 1995 年原始的 Java 1.0 版本发布以来，Java 已发布了许多版本。（请参阅[Java 版本历史](#)以获取摘要。）然而，大多数版本已过官方生命周期终止日期。这意味着供应商（通常是现在的 Oracle）已停止该版本的新开发，并且不再为任何漏洞或安全问题提供公开/免费的补丁。（私有补丁通常提供给拥有支持合同的个人/组织；请联系您供应商的销售办公室。）

一般来说，推荐使用的 Java SE 版本将是最新公开版本的最新更新。

目前，这意味着最新可用的 Java 8 版本。Java 9 计划于 2017 年公开发布。（Java 7 已经结束生命周期，最后一次公开发布是在 2015 年 4 月。不推荐使用 Java 7 及更早版本。）此建议适用于所有新的 Java 开发人员以及所有

学习 Java 的人。它也适用于那些仅想运行第三方提供的 Java 软件的人。一般来说，编写良好的 Java 代码可以在较新的 Java 版本上运行。（但请查看软件的发行说明，如果有疑问，请联系作者/供应商/厂商。）

Enter to keep the current selection[+], or type selection number: 2
\$

For more information on using alternatives, refer to the [alternatives\(8\)](#) manual entry.

Arch based installs

Arch Linux based installs come with the command archlinux-java to switch java versions.

Listing installed environments

```
$ archlinux-java status
Available Java environments:
java-7-openjdk (default)
java-8-openjdk/jre
```

Switching current environment

```
# archlinux-java set <JAVA_ENV_NAME>
```

Eg:

```
# archlinux-java set java-8-openjdk/jre
```

More information can be found on the [Arch Linux Wiki](#)

Section A.6: What do I need for Java Development

A JDK installation and a text editor are the bare minimum for Java development. (It is nice to have a text editor that can do Java syntax highlighting, but you can do without.)

However for serious development work it is recommended that you also use the following:

- A Java IDE such as Eclipse, IntelliJ IDEA or NetBeans
- A Java build tool such as Ant, Gradle or Maven
- A version control system for managing your code base (with appropriate backups, and off-site replication)
- Test tools and CI (continuous integration) tools

Section A.7: Selecting an appropriate Java SE release

There have been many releases of Java since the original Java 1.0 release in 1995. (Refer to [Java version history](#) for a summary.) However most releases have passed their official End Of Life dates. This means that the vendor (typically Oracle now) has ceased new development for the release, and no longer provides public / free patches for any bugs or security issues. (Private patch releases are typically available for people / organizations with a support contract; contact your vendor's sales office.)

In general, the recommended Java SE release for use will be the latest update for the latest public version.

Currently, this means the latest available Java 8 release. Java 9 is due for public release in 2017. (Java 7 has passed its End Of Life and the last public release was in April 2015. Java 7 and earlier releases are not recommended.)

This recommendation applies for all new Java development, and anyone learning Java. It also applies to people who just want to run Java software provided by a third-party. Generally speaking, well-written Java code will work on a newer release of Java. (But check the software's release notes, and contact the author / supplier / vendor if you have doubts.)

如果您正在处理较旧的 Java 代码库，建议确保您的代码能在最新的 Java 版本上运行。决定何时开始使用较新 Java 版本的功能更为困难，因为这会影响您支持那些无法或不愿升级其 Java 安装的客户的能力。

A.8 节：Java 版本发布和命名

Java 版本命名有些混乱。实际上有两套命名和编号系统，如下表所示：

JDK 版本	市场名称
jdk-1.0	JDK 1.0
jdk-1.1	JDK 1.1
jdk-1.2	J2SE 1.2
...	...
jdk-1.5	J2SE 1.5 重新命名为 Java SE 5
jdk-1.6	Java SE 6
jdk-1.7	Java SE 7
jdk-1.8	Java SE 8
jdk-91	Java SE 9 (尚未发布)

1 - 看起来甲骨文（Oracle）打算打破他们之前在Java版本字符串中使用“语义版本号”方案的惯例。是否会真正执行还有待观察。

营销名称中的“SE”指的是标准版（Standard Edition）。这是在大多数笔记本电脑、个人电脑和服务器（除Android外）上运行Java的基础版本。

Java还有另外两个官方版本：“Java ME”是微型版（Micro Edition），而“Java EE”是企业版（Enterprise Edition）。Android版本的Java与Java SE也有显著不同。Java ME、Java EE和Android Java不在本主题讨论范围内。

Java版本的完整版本号格式如下：

1.8.0_101-b13

这表示JDK 1.8.0，第101次更新，构建号13。甲骨文在发布说明中称之为：

Java™ SE 开发工具包 8，第101次更新 (JDK 8u101)

更新号很重要——Oracle 会定期为一个主要版本发布包含安全补丁、错误修复以及（在某些情况下）新功能的更新。构建号通常无关紧要。请注意，Java 8 和 Java 1.8 指的是同一事物；Java 8 只是 Java 1.8 的“市场”名称。

第 A.9 节：使用最新 tar 文件在 Linux 上安装 Oracle Java

按照以下步骤从最新的 tar 文件安装 Oracle JDK：

1. 从 [here](#) 下载最新的 tar 文件——当前最新版本是 Java SE 开发工具包 8u112。
2. 你需要 sudo 权限：

`sudo su`

If you are working on an older Java codebase, you would be advised to ensure that your code runs on the latest release of Java. Deciding when to start using the features of newer Java releases is more difficult, as this will impact your ability to support customers who are unable or unwilling to upgrade their Java installation.

Section A.8: Java release and version naming

Java release naming is a little confusing. There are actually two systems of naming and numbering, as shown in this table:

JDK version	Marketing name
jdk-1.0	JDK 1.0
jdk-1.1	JDK 1.1
jdk-1.2	J2SE 1.2
...	...
jdk-1.5	J2SE 1.5 rebranded Java SE 5
jdk-1.6	Java SE 6
jdk-1.7	Java SE 7
jdk-1.8	Java SE 8
jdk-91	Java SE 9 (not released yet)

1 - It appears that Oracle intends to break from their previous practice of using a "semantic version number" scheme in the Java version strings. It remains to be seen if they will follow through with this.

The "SE" in the marketing names refers to Standard Edition. This is the base release for running Java on most laptops, PCs and servers (apart from Android).

There are two other official editions of Java: "Java ME" is the Micro Edition, and "Java EE" is the Enterprise Edition. The Android flavor of Java is also significantly different from Java SE. Java ME, Java EE and Android Java are outside of the scope of this Topic.

The full version number for a Java release looks like this:

1.8.0_101-b13

This says JDK 1.8.0, Update 101, Build #13. Oracle refers to this in the release notes as:

Java™ SE Development Kit 8, Update 101 (JDK 8u101)

The update number is important -- Oracle regularly issue updates to a major release with security patches, bug fixes and (in some cases) new features. The build number is usually irrelevant. Note that Java 8 and Java 1.8 refer to the same thing; Java 8 is just the "marketing" name for Java 1.8.

Section A.9: Installing Oracle Java on Linux with latest tar file

Follow the below steps to install Oracle JDK from the latest tar file:

1. Download the latest tar file from [here](#) - Current latest is Java SE Development Kit 8u112.
2. You need sudo privileges:

`sudo su`

3. 创建一个用于安装 jdk 的目录：

```
mkdir /opt/jdk
```

4. 将下载的 tar 文件解压到该目录：

```
tar -zxf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
```

5. 验证文件是否已解压：

```
ls /opt/jdk
```

6. 设置 Oracle JDK 为默认 JVM：

```
update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
```

和

```
update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
```

7. 检查 Java 版本：

```
java -version
```

预期输出：

```
java version "1.8.0_111"
Java(TM) SE Runtime Environment (build 1.8.0_111-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)
```

第A.10节：Linux上的安装后检查和配置

安装Java SDK后，建议检查其是否已准备好使用。您可以使用普通用户账户运行以下两个命令来进行检查：

```
$ java -version
$ javac -version
```

这些命令会打印出JRE和JDK（分别）的版本信息，这些版本信息位于您的shell命令搜索路径中。请查找JDK / JRE的版本字符串。

- 如果上述任一命令失败，提示“command not found”，则说明JRE或JDK根本不在搜索路径中；请直接转到下面的直接配置PATH部分。
- 如果上述任一命令显示的版本字符串与您预期的不同，则说明您的搜索路径或“alternatives”系统需要调整；请转到检查Alternatives
- 如果显示了正确的版本字符串，说明您几乎完成了；请跳转到检查JAVA_HOME

直接配置PATH

如果当前搜索路径中没有java或javac，那么简单的解决方案是将其添加到您的搜索路径中

3. Create a dir for jdk install:

```
mkdir /opt/jdk
```

4. Extract downloaded tar into it:

```
tar -zxf jdk-8u5-linux-x64.tar.gz -C /opt/jdk
```

5. Verify if the files are extracted:

```
ls /opt/jdk
```

6. Setting Oracle JDK as the default JVM:

```
update-alternatives --install /usr/bin/java java /opt/jdk/jdk1.8.0_05/bin/java 100
```

and

```
update-alternatives --install /usr/bin/javac javac /opt/jdk/jdk1.8.0_05/bin/javac 100
```

7. Check Java version:

```
java -version
```

Expected output:

```
java version "1.8.0_111"
Java(TM) SE Runtime Environment (build 1.8.0_111-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)
```

Section A.10: Post-installation checking and configuration on Linux

After installing a Java SDK, it is advisable to check that it is ready to use. You can do this by running these two commands, using your normal user account:

```
$ java -version
$ javac -version
```

These commands print out the version information for the JRE and JDK (respectively) that are on your shell's command search path. Look for the JDK / JRE version string.

- If either of the above commands fails, saying "command not found", then the JRE or JDK is not on the search path at all; go to **Configuring PATH directly** below.
- If either of the above commands displays a different version string to what you were expecting, then either your search path or the "alternatives" system needs adjusting; go to **Checking Alternatives**
- If the correct version strings are displayed, you are nearly done; skip to **Checking JAVA_HOME**

Configuring PATH directly

If there is no java or javac on the search path at the moment, then the simple solution is to add it to your search

路径。

首先，找到你安装 Java 的位置；如果有疑问，请参见下面的Java 安装在哪里？。

接下来，假设你的命令行是bash，使用文本编辑器将以下内容添加到以下任一文件的末尾
~/.bash_profile 或 ~/.bashrc (如果你使用 Bash 作为你的 shell) 。

```
JAVA_HOME=<安装目录>
PATH=$JAVA_HOME/bin:$PATH
```

```
export JAVA_HOME
export PATH
```

... 将 <安装目录> 替换为你的 Java 安装目录的路径名。注意，上述内容假设安装目录包含一个 bin 目录，且 bin 目录中包含你要使用的 java 和 javac 命令。

接下来，执行你刚编辑的文件，使当前 shell 的环境变量得到更新。

```
$ source ~/.bash_profile
```

然后，重复检查 java 和 javac 的版本。如果仍有问题，使用 which java 和 which javac 来确认你是否正确更新了环境变量。

最后，注销并重新登录，使更新后的环境变量传播到所有 shell。现在应该完成了。

检查备选项

如果java-version或javac-version命令能执行但显示了意外的版本号，你需要检查这些命令来自哪里。使用which和ls-l命令来查明，方法如下：

```
$ ls -l `which java`
```

如果输出看起来像这样，：

```
lrwxrwxrwx. 1 root root 22 7月 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

那么就是使用了alternatives版本切换。你需要决定是继续使用它，还是直接通过设置PATH来覆盖它。

- 在Linux上使用alternatives配置和切换Java版本
- 参见上文“直接配置PATH”。

Java安装在哪里？

Java可以根据安装方式安装在不同的位置。

- Oracle的RPM包将Java安装在"/usr/java"目录下。
- 在 Fedora 上，默认位置是"/usr/lib/jvm"。
- 如果Java是通过ZIP或JAR文件手动安装的，安装位置可能在任何地方。

如果您难以找到安装目录，建议您使用find (或slocate) 来查找命令。例如：

path.

First, find where you installed Java; see **Where was Java installed?** below if you have doubts.

Next, assuming that bash is your command shell, use a text editor to add the following lines to the end of either
~/.bash_profile or ~/.bashrc (If you use Bash as your shell).

```
JAVA_HOME=<installation directory>
PATH=$JAVA_HOME/bin:$PATH
```

```
export JAVA_HOME
export PATH
```

... replacing <installation directory> with the pathname for your Java installation directory. Note that the above assumes that the installation directory contains a bin directory, and the bin directory contains the java and javac commands that you are trying to use.

Next, source the file that you just edited, so that the environment variables for your current shell are updated.

```
$ source ~/.bash_profile
```

Next, repeat the java and javac version checks. If there are still problems, use which java and which javac to verify that you have updates the environment variables correctly.

Finally, logout and login again so that the updated environment variables propagate to all of your shells. You should now be done.

Checking Alternatives

If java -version or javac -version worked but gave an unexpected version number, you need to check where the commands are coming from. Use which and ls -l to find this out as follows:

```
$ ls -l `which java`
```

If the output looks like this,：

```
lrwxrwxrwx. 1 root root 22 Jul 30 22:18 /usr/bin/java -> /etc/alternatives/java
```

then the alternatives version switching is being used. You needs to decide whether to continue using it, or simply override it by setting the PATH directly.

- Configuring and Switching Java versions on Linux using alternatives
- See "Configuring PATH directly" above.

Where was Java installed?

Java can be installed in a variety of places, depending on the installation method.

- The Oracle RPMs put the Java installation in "/usr/java".
- On Fedora, the default location is "/usr/lib/jvm".
- If Java was installed by hand from ZIP or JAR files, the installation could be anywhere.

If you are having difficultly finding the installation directory, We suggest that you use find (or slocate) to find the command. For example:

```
$ find / -name java -type f 2> /dev/null
```

这将为您提供系统上所有名为java的文件的路径名。 (将标准错误重定向到 "/dev/null" 可以抑制您无法访问的文件和目录的消息。)

```
$ find / -name java -type f 2> /dev/null
```

This gives you the pathnames for all files called java on your system. (The redirection of standard error to "/dev/null" suppresses messages about files and directories that you can't access.)

附录B：Java版本、版本号、发布和发行版

B.1节：Java SE JRE和Java SE JDK 发行版之间的区别

Sun / Oracle发布的Java SE有两种形式：JRE和JDK。简单来说，JRE支持运行Java应用程序，而JDK还支持Java开发。

Java运行时环境

Java运行环境或JRE发行版包含运行和管理Java应用程序所需的一套库和工具。典型的现代JRE中的工具包括：

- 用于在JVM（Java虚拟机）中运行Java程序的java命令。
- 用于运行Nashorn Javascript引擎的jjs命令。
- 用于操作Java密钥库的keytool命令。
- 用于编辑安全沙箱安全策略的policytool命令。
- 用于打包和解包用于网页部署的“pack200”文件的pack200和unpack200工具。
- 支持Java CORBA和RMI应用程序的orbd、rmid、rmiregistry和tnameserv命令。

“桌面JRE”安装程序包含适用于某些网页浏览器的Java插件。这在“服务器JRE”安装程序中被故意省略。

linux syscall read benchmarku

从Java 7更新6开始，JRE安装程序已包含JavaFX（版本2.2或更高）。

Java开发工具包

Java开发工具包或JDK发行版包含JRE工具，以及用于开发Java软件的额外工具。

附加工具通常包括：

- javac命令，用于将Java源代码（“.java”）编译成字节码文件（“.class”）。
- 用于创建JAR文件的工具，如jar和jarsigner
- 开发工具包括：
 - appletviewer用于运行applet
 - idljCORBA IDL到Java的编译器
 - javahJNI存根生成器
 - native2ascii用于Java源代码的字符集转换
 - schemagenJava到XML模式生成器（JAXB的一部分）
 - serialver生成Java对象序列化版本字符串。
 - 用于JAX-WS的wsgen和wsimport支持工具
- 诊断工具包括：
 - jdb基本的Java调试器
 - jmap和jhat用于转储和分析Java堆。
 - jstack用于获取线程堆栈转储。
 - javap用于查看“.class”文件。
- 应用管理和监控工具包括：
 - jconsole管理控制台，
 - jstat、jstatd、jinfo和jps用于应用监控

典型的Sun / Oracle JDK安装包还包括一个包含Java库源代码的ZIP文件。在Java 6之前，这是唯一公开可用的Java源代码。

Appendix B: Java Editions, Versions, Releases and Distributions

Section B.1: Differences between Java SE JRE or Java SE JDK distributions

Sun / Oracle releases of Java SE come in two forms: JRE and JDK. In simple terms, JREs support running Java applications, and JDKs also support Java development.

Java Runtime Environment

Java Runtime Environment or JRE distributions consist of the set of libraries and tools needed to run and manage Java applications. The tools in a typical modern JRE include:

- The java command for running a Java program in a JVM (Java Virtual Machine)
- The jjs command for running the Nashorn Javascript engine.
- The keytool command for manipulating Java keystores.
- The policytool command for editing security sandbox security policies.
- The pack200 and unpack200 tools for packing and unpacking "pack200" file for web deployment.
- The orbd, rmid, rmiregistry and tnameserv commands that support Java CORBA and RMI applications.

"Desktop JRE" installers include a Java plugin suitable for some web browser. This is deliberately left out of "Server JRE" installers.linux syscall read benchmarku

From Java 7 update 6 onwards, JRE installers have included JavaFX (version 2.2 or later).

Java Development Kit

A Java Development Kit or JDK distribution includes the JRE tools, and additional tools for developing Java software. The additional tools typically include:

- The javac command, which compiles Java source code（“.java”）to bytecode files（“.class”）。
- The tools for creating JAR files such as jar and jarsigner
- Development tools such as：
 - appletviewer for running applets
 - idlj the CORBA IDL to Java compiler
 - javah the JNI stub generator
 - native2ascii for character set conversion of Java source code
 - schemagen the Java to XML schema generator (part of JAXB)
 - serialver generate Java Object Serialization version string.
 - the wsgen and wsimport support tools for JAX-WS
- Diagnostic tools such as：
 - jdb the basic Java debugger
 - jmap and jhat for dumping and analysing a Java heap.
 - jstack for getting a thread stack dump.
 - javap for examining ".class" files.
- Application management and monitoring tools such as：
 - jconsole a management console,
 - jstat, jstatd, jinfo and jps for application monitoring

A typical Sun / Oracle JDK installation also includes a ZIP file with the source code of the Java libraries. Prior to Java 6, this was the only publicly available Java source code.

从 Java 6 开始，OpenJDK 的完整源代码可从 OpenJDK 网站下载。它通常不包含在 (Linux) JDK 包中，但作为单独的软件包提供。

B.2 节：Java SE 版本

Java SE 版本历史

下表提供了 Java SE 平台重要主要版本的时间线。

Java SE 版本1	代号	终止支持 (免费2)	发布日期
Java SE 10 (抢先体验) 无	未来	未来	2018-03-20 (预计)
Java SE 9	无	未来	2017-07-27
Java SE 8	无	未来	2014-03-18
Java SE 7	海豚	2015-04-14	2011-07-28
Java SE 6	野马	2013-04-16	2006-12-23
Java SE 5	虎 (Tiger)	2009-11-04	2004-10-04
Java SE 1.4.2	螳螂 (Mantis)	2003-06-26	Java SE 1.4.2 2009-11-04之前
11-04之前	霍珀 / 蚱蜢 (Hopper / Grasshopper)	2002-09-16	2009-11-04之前
Java SE 1.4	梅林 (Merlin)	2002-02-06	Java SE 1.3.1 2009-11-04之前
11-04之前	瓢虫	2009-11-04 之前	2001-05-17
Java SE 1.3	啄木鸟	2009-11-04 之前	2000-05-08
Java SE 1.2	游乐场	2009-11-04 之前	1998-12-08
Java SE 1.1	火花器	2009-11-04 之前	1997-02-19
Java SE 1.0	橡树	2009-11-04 之前	1996-01-21

脚注：

- 这些链接指向Oracle网站上各个版本发布文档的在线副本。许多较旧版本的文档已不再在线，但通常可以从Oracle Java档案中下载。
- 大多数历史版本的Java SE已过其官方“生命周期结束”日期。当Java版本达到此里程碑时，Oracle将停止为其提供免费更新。更新仍然可供拥有支持合同的客户使用。

来源：

- [JDK发布日期](#) 由加拿大Mind Products的Roedy Green提供

Java SE版本亮点

Java SE 版本	亮点
Java SE 8	Lambda表达式和受MapReduce启发的Streams。Nashorn Javascript引擎。类型注解和重复注解。无符号算术扩展。新的日期和时间API。静态链接的JNI库。JavaFX启动器。移除PermGen。
Java SE 7	字符串开关，try-with-resource，菱形符号 (<>)，数字字面量增强和异常处理/重新抛出改进。并发库增强。增强对本地文件系统的支持。Timsort排序算法。ECC加密算法。改进的二维图形 (GPU) 支持。可插拔注解。
Java SE 6	对JVM平台和Swing的显著性能提升。脚本语言API和Mozilla Rhino JavaScript引擎。JDBC 4.0。编译器API。JAXB 2.0。Web服务支持 (JAX-WS)。

From Java 6 onwards, the complete source code for OpenJDK is available for download from the OpenJDK site. It is typically not included in (Linux) JDK packages, but is available as a separate package.

Section B.2: Java SE Versions

Java SE Version History

The following table provides the timeline for the significant major versions of the Java SE platform.

Java SE Version1	Code Name	End-of-life (free2)	Release Date
Java SE 10 (Early Access) None	future	future	2018-03-20 (estimated)
Java SE 9 None	future	future	2017-07-27
Java SE 8 None	future	future	2014-03-18
Java SE 7 Dolphin	2015-04-14	2011-07-28	
Java SE 6 Mustang	2013-04-16	2006-12-23	
Java SE 5 Tiger	2009-11-04	2004-10-04	
Java SE 1.4.2 Mantis	prior to 2009-11-04	2003-06-26	
Java SE 1.4.1	Hopper / Grasshopper	prior to 2009-11-04	2002-09-16
Java SE 1.4	Merlin	prior to 2009-11-04	2002-02-06
Java SE 1.3.1	Ladybird	prior to 2009-11-04	2001-05-17
Java SE 1.3 Kestrel	prior to 2009-11-04	2000-05-08	
Java SE 1.2	Playground	prior to 2009-11-04	1998-12-08
Java SE 1.1	Sparkler	prior to 2009-11-04	1997-02-19
Java SE 1.0	Oak	prior to 2009-11-04	1996-01-21

Footnotes:

1. The links are to online copies of the respective releases documentation on Oracle's website. The documentation for many older releases no longer online, though it typically can be downloaded from the Oracle Java Archives.
2. Most historical versions of Java SE have passed their official "end of life" dates. When a Java version passes this milestone, Oracle stop providing free updates for it. Updates are still available to customers with support contracts.

Source:

- [JDK release dates](#) by Roedy Green of Canadian Mind Products

Java SE Version Highlights

Java SE Version	Highlights
Java SE 8	Lambda expressions and MapReduce-inspired Streams. The Nashorn Javascript engine. Annotations on types and repeating annotations. Unsigned arithmetic extensions. New Date and Time APIs. Statically linked JNI libraries. JavaFX launcher. Removal of PermGen.
Java SE 7	String switches, try-with-resource, the diamond (<>), numeric literal enhancements and exception handling / rethrowing improvements. Concurrency library enhancements. Enhanced support for native file systems. Timsort. ECC crypto algorithms. Improved 2D graphics (GPU) support. Pluggable annotations.
Java SE 6	Significant performance enhancements to JVM platform and Swing. Scripting language API and Mozilla Rhino Javascript engine. JDBC 4.0. Compiler API. JAXB 2.0. Web Services support (JAX-WS).

Java SE 5	泛型、注解、自动装箱、枚举类、可变参数、增强的for循环和静态导入。 Java内存模型规范。Swing和RMI增强。新增 java.util.concurrent.*包和Scanner类。	Generics, annotations, auto-boxing, enum classes, varargs, enhanced for loops and static imports. Specification of the Java Memory Model. Swing and RMI enhancements. Addition of java.util.concurrent.* package and Scanner.
Java SE 1.4	assert关键字。正则表达式类。异常链。NIO API——非阻塞I/O， Buffer和Channel。java.util.logging.* API。图像I/O API。集成的XML和XSLT (JAXP)。 集成的安全和加密 (JCE、JSSE、JAAS)。集成的Java Web Start。首选项API。	The assert keyword. Regular expression classes. Exception chaining. NIO APIs - non-blocking I/O, Buffer and Channel. java.util.logging.* API. Image I/O API. Integrated XML and XSLT (JAXP). Integrated security and cryptography (JCE, JSSE, JAAS). Integrated Java Web Start. Preferences API.
Java SE 1.3	Java SE 1.3 包含 HotSpot JVM, CORBA / RMI 集成。Java 命名和目录接口 (JNDI)。调试器 框架 (JPDA)。JavaSound API。代理 API。	HotSpot JVM included. CORBA / RMI integration. Java Naming and Directory Interface (JNDI). Debugger framework (JPDA). JavaSound API. Proxy API.
Java SE 1.2	引入了strictfp关键字。Swing API。Java 插件 (用于网页浏览器)。CORBA 互操作性。 集合框架。	The strictfp keyword. Swing APIs. The Java plugin (for web browsers). CORBA interoperability. Collections framework.
Java SE 1.1	内部类。反射。JDBC。RMI。Unicode / 字符流。国际化支持。 AWT 事件模型的全面改造。JavaBeans。	Inner classes. Reflection. JDBC. RMI. Unicode / character streams. Internationalization support. Overhaul of AWT event model. JavaBeans.

来源：

- 维基百科：[Java 版本历史](#)

B.3 节：Java EE、Java SE、Java ME 和 JavaFX 之间的区别

Java 技术既是一种编程语言，也是一个平台。Java 编程语言是一种高级的面向对象语言，具有特定的语法和风格。Java 平台是运行 Java 编程语言应用程序的特定环境。

存在多个 Java 平台。许多开发者，即使是长期的 Java 编程语言开发者，也不理解不同平台之间的关系。

Java 编程语言平台

Java 编程语言有四个平台：

- Java 平台，标准版 (Java SE)
- Java 平台，企业版 (Java EE)
- Java 平台，微型版 (Java ME)
- Java FX

所有 Java 平台都由 Java 虚拟机 (VM) 和应用程序编程接口 (API) 组成。Java 虚拟机是针对特定硬件和软件平台的程序，用于运行 Java 技术应用程序。API 是一组软件组件，您可以使用它们来创建其他软件组件或应用程序。每个 Java 平台都提供一个虚拟机和一个 API，这使得为该平台编写的应用程序能够在任何兼容系统上运行，并享有 Java 编程语言的所有优势：平台无关性、强大性、稳定性、易开发性和安全性。

Java SE

当大多数人想到 Java 编程语言时，他们想到的是 Java SE API。Java SE 的 API 提供了 Java 编程语言的核心功能。它定义了从 Java 编程语言的基本类型和对象到用于网络、安全、数据库访问、图形用户界面 (GUI) 开发和 XML 解析的高级类的所有内容。

除了核心 API 外，Java SE 平台还包括虚拟机、开发工具、部署技术以及 Java 技术应用中常用的其他类库和工具包。

Java SE 5	Generics, annotations, auto-boxing, enum classes, varargs, enhanced for loops and static imports. Specification of the Java Memory Model. Swing and RMI enhancements. Addition of java.util.concurrent.* package and Scanner.
Java SE 1.4	The assert keyword. Regular expression classes. Exception chaining. NIO APIs - non-blocking I/O, Buffer and Channel. java.util.logging.* API. Image I/O API. Integrated XML and XSLT (JAXP). Integrated security and cryptography (JCE, JSSE, JAAS). Integrated Java Web Start. Preferences API.
Java SE 1.3	HotSpot JVM included. CORBA / RMI integration. Java Naming and Directory Interface (JNDI). Debugger framework (JPDA). JavaSound API. Proxy API.
Java SE 1.2	The strictfp keyword. Swing APIs. The Java plugin (for web browsers). CORBA interoperability. Collections framework.
Java SE 1.1	Inner classes. Reflection. JDBC. RMI. Unicode / character streams. Internationalization support. Overhaul of AWT event model. JavaBeans.

Source:

- Wikipedia: [Java version history](#)

Section B.3: Differences between Java EE, Java SE, Java ME and JavaFX

Java technology is both a programming language and a platform. The Java programming language is a high-level object-oriented language that has a particular syntax and style. A Java platform is a particular environment in which Java programming language applications run.

There are several Java platforms. Many developers, even long-time Java programming language developers, do not understand how the different platforms relate to each other.

The Java Programming Language Platforms

There are four platforms of the Java programming language:

- Java Platform, Standard Edition (Java SE)
- Java Platform, Enterprise Edition (Java EE)
- Java Platform, Micro Edition (Java ME)
- Java FX

All Java platforms consist of a Java Virtual Machine (VM) and an application programming interface (API). The Java Virtual Machine is a program, for a particular hardware and software platform, that runs Java technology applications. An API is a collection of software components that you can use to create other software components or applications. Each Java platform provides a virtual machine and an API, and this allows applications written for that platform to run on any compatible system with all the advantages of the Java programming language: platform-independence, power, stability, ease-of-development, and security.

Java SE

When most people think of the Java programming language, they think of the Java SE API. Java SE's API provides the core functionality of the Java programming language. It defines everything from the basic types and objects of the Java programming language to high-level classes that are used for networking, security, database access, graphical user interface (GUI) development, and XML parsing.

In addition to the core API, the Java SE platform consists of a virtual machine, development tools, deployment technologies, and other class libraries and toolkits commonly used in Java technology applications.

Java EE

Java EE 平台构建于 Java SE 平台之上。Java EE 平台提供了用于开发和运行大规模、多层、可扩展、可靠且安全的网络应用程序的 API 和运行环境。

Java ME

Java ME 平台提供了一个 API 和一个小型虚拟机，用于在小型设备（如手机）上运行 Java 编程语言应用程序。该 API 是 Java SE API 的子集，并包含适用于小型设备应用开发的特殊类库。Java ME 应用程序通常是 Java EE 平台服务的客户端。

Java FX

Java FX 技术是一个用于创建丰富互联网应用的平台，这些应用使用 Java FX Script™ 编写。Java FX Script 是一种静态类型的声明式语言，编译成 Java 技术字节码，然后可以在 Java 虚拟机上运行。为 Java FX 平台编写的应用程序可以包含并链接 Java 编程语言类，也可以作为 Java EE 平台服务的客户端。

- 摘自Oracle 文档

Java EE

The Java EE platform is built on top of the Java SE platform. The Java EE platform provides an API and runtime environment for developing and running large-scale, multi-tiered, scalable, reliable, and secure network applications.

Java ME

The Java ME platform provides an API and a small-footprint virtual machine for running Java programming language applications on small devices, like mobile phones. The API is a subset of the Java SE API, along with special class libraries useful for small device application development. Java ME applications are often clients of Java EE platform services.

Java FX

Java FX technology is a platform for creating rich internet applications written in Java FX ScriptTM. Java FX Script is a statically-typed declarative language that is compiled to Java technology bytecode, which can then be run on a Java VM. Applications written for the Java FX platform can include and link to Java programming language classes, and may be clients of Java EE platform services.

- Taken from the [Oracle documentation](#)

附录 C：类路径

类路径列出了 Java 运行时应查找类和资源的位置。类路径也被 Java 编译器用来查找先前编译的和外部的依赖项。

C.1 节：指定类路径的不同方式

设置类路径有三种方式。

1. 可以使用CLASSPATH环境变量进行设置：

```
set CLASSPATH=...      # Windows 和 csh  
export CLASSPATH=...    # Unix ksh/bash
```

2. 可以在命令行中按如下方式设置

```
java -classpath ...  
javac -classpath ...
```

注意，-classpath（或-cp）选项优先于CLASSPATH环境变量。

3. 可执行 JAR 文件的类路径通过 MANIFEST.MF 中的 Class-Path 元素指定：

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

注意，这仅适用于如下方式执行 JAR 文件时：

```
java -jar some.jar ...
```

在这种执行模式下，-classpath 选项和 CLASSPATH 环境变量将被忽略，即使 JAR 文件没有 Class-Path 元素。

如果未指定类路径，则使用java-jar时，默认类路径为所选的JAR文件，否则为当前目录。

相关内容：

- <https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html>
- <http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html>

第C.2节：将目录中的所有JAR添加到类路径

如果想将目录中的所有JAR添加到类路径，可以使用类路径通配符语法简洁地实现；例如：

```
someFolder/*
```

这告诉JVM将 someFolder目录中的所有JAR和ZIP文件添加到类路径。此语法可用于-cp参数、CLASSPATH环境变量，或可执行JAR文件清单中的Class-Path属性。详见设置类路径：类路径通配符的示例和注意事项。

Appendix C: The Classpath

The classpath lists places where the Java runtime should look for classes and resources. The classpath is also used by the Java compiler to find previously compiled and external dependencies.

Section C.1: Different ways to specify the classpath

There are three ways to set the classpath.

1. It can be set using the CLASSPATH environment variable :

```
set CLASSPATH=...      # Windows 和 csh  
export CLASSPATH=...    # Unix ksh/bash
```

2. It can be set on the command line as follows

```
java -classpath ...  
javac -classpath ...
```

Note that the -classpath (or -cp) option takes precedence over the CLASSPATH environment variable.

3. The classpath for an executable JAR file is specified using the Class-Path element in MANIFEST.MF:

```
Class-Path: jar1-name jar2-name directory-name/jar3-name
```

Note that this only applies when the JAR file is executed like this:

```
java -jar some.jar ...
```

In this mode of execution, the -classpath option and the CLASSPATH environment variable will be ignored, even if the JAR file has no Class-Path element.

If no classpath is specified, then the default classpath is the selected JAR file when using java -jar, or the current directory otherwise.

Related:

- <https://docs.oracle.com/javase/tutorial/deployment/jar/downman.html>
- <http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html>

Section C.2: Adding all JARs in a directory to the classpath

If you want to add all the JARs in directory to the classpath, you can do this concisely using classpath wildcard syntax; for example:

```
someFolder/*
```

This tells the JVM to add all JAR and ZIP files in the someFolder directory to the classpath. This syntax can be used in a -cp argument, a CLASSPATH environment variable, or a Class-Path attribute in an executable JAR file's manifest file. See [Setting the Class Path: Class Path Wild Cards](#) for examples and caveats.

1. 类路径通配符首次在Java 6中引入。早期版本的Java不将“*”视为通配符。
2. 不能在“”前后添加其他字符；例如，“someFolder/.jar”不是通配符。
3. 通配符仅匹配后缀为“.jar”或“.JAR”的文件。ZIP文件会被忽略，后缀不同的JAR文件也会被忽略。
4. 通配符只匹配目录中的 JAR 文件，不包括其子目录中的文件。
5. 当一组 JAR 文件被通配符条目匹配时，它们在类路径上的相对顺序是不确定的。

第 C.3 节：从类路径加载资源

加载打包在 JAR 内的资源（图片、文本文件、属性文件、KeyStore 等）可能很有用。为此，我们可以使用Class和ClassLoader。

假设我们有如下项目结构：

```
program.jar
|
\-com
 \-project
 |
 |-file.txt
 \-Test.class
```

我们想从Test类访问file.txt的内容。可以通过类加载器来实现：

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```

通过使用类加载器，我们需要指定资源的完全限定路径（每个包）。

或者，我们也可以直接使用 Test 类对象

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

使用类对象时，路径是相对于类本身的。我们的 Test.class 位于 com.project 包中，与 file.txt 相同，因此我们根本不需要指定任何路径。

不过，我们也可以使用类对象的绝对路径，如下所示：

```
is = Test.class.getResourceAsStream("/com/project/file.txt");
```

C.4 节：类路径语法

类路径是一系列条目，这些条目可以是目录路径名、JAR 或 ZIP 文件路径名，或者 JAR / ZIP 通配符规范。

- 对于命令行中指定的类路径（例如 -classpath）或作为环境变量，条目必须用 ; (分号) 字符分隔（Windows 上），或用 : (冒号) 字符分隔（其他平台，如 Linux、UNIX、MacOSX 等）。
- 对于 JAR 文件的 MANIFEST.MF 中的 Class-Path 元素，使用单个空格分隔条目。

有时需要在类路径条目中嵌入空格

Notes:

1. Classpath wildcards were first introduced in Java 6. Earlier versions of Java do not treat “*” as a wildcard.
2. You cannot put other characters before or after the “”; e.g. “someFolder/.jar” is not a wildcard.
3. A wildcard matches only files with the suffix “.jar” or “.JAR”. ZIP files are ignored, as are JAR files with a different suffixes.
4. A wildcard matches only JAR files in the directory itself, not in its subdirectories.
5. When a group of JAR files is matched by a wildcard entry, their relative order on the classpath is not specified.

Section C.3: Load a resource from the classpath

It can be useful to load a resource (image, text file, properties, KeyStore, ...) that is packaged inside a JAR. For this purpose, we can use the **Class** and **ClassLoader**s.

Suppose we have the following project structure:

```
program.jar
|
\-com
 \-project
 |
 |-file.txt
 \-Test.class
```

And we want to access the contents of `file.txt` from the `Test` class. We can do so by asking the classloader:

```
InputStream is = Test.class.getClassLoader().getResourceAsStream("com/project/file.txt");
```

By using the classloader, we need to specify the fully qualified path of our resource (each package).

Or alternatively, we can ask the `Test` class object directly

```
InputStream is = Test.class.getResourceAsStream("file.txt");
```

Using the class object, the path is relative to the class itself. Our `Test.class` being in the `com.project` package, the same as `file.txt`, we do not need to specify any path at all.

We can, however, use absolute paths from the class object, like so:

```
is = Test.class.getResourceAsStream("/com/project/file.txt");
```

Section C.4: Classpath path syntax

The classpath is a sequence of entries which are directory pathnames, JAR or ZIP file pathnames, or JAR / ZIP wildcard specifications.

- For a classpath specified on the command line (e.g. -classpath) or as an environment variable, the entries must be separated with ; (semicolon) characters on Windows, or : (colon) characters on other platforms (Linux, UNIX, MacOSX and so on).
- For the Class-Path element in a JAR file's MANIFEST.MF, use a single space to separate the entries.

Sometimes it is necessary to embed a space in a classpath entry

- 当在命令行中指定类路径时，只需使用适当的shell引号即可。例如：

```
export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
```

(具体细节可能取决于您使用的命令shell。)

- 当类路径在JAR文件的“MANIFEST.MF”文件中指定时，必须使用URL编码。

```
Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
```

第C.5节：动态类路径

有时，仅仅添加文件夹中的所有JAR文件是不够的，例如当您有本地代码并且需要选择JAR文件的子集时。在这种情况下，您需要两个main()方法。第一个构建一个类加载器，然后使用该类加载器调用第二个main()方法。

下面是一个示例，它为您的平台选择正确的SWT本地JAR，添加您所有应用程序的JAR，然后调用真正的main()方法：创建跨平台Java SWT应用程序

第C.6节：将类名映射到路径名

标准的Java工具链（以及设计为与其互操作的第三方工具）对类名映射到表示它们的文件路径名和其他资源有特定规则。

映射规则如下

- 对于默认包中的类，路径名是简单的文件名。
- 对于命名包中的类，包名的各个组成部分映射为目录。
- 对于命名的嵌套类和内部类，文件名部分由类名通过\$字符连接组成。
- 对于匿名内部类，使用数字代替名称。

如下表所示：

类名	源文件路径名	类文件路径名
SomeClass	SomeClass.java	SomeClass.class
com.example.SomeClass	com/example/SomeClass.java	com/example/SomeClass.class
SomeClass.Inner	(in SomeClass.java)	SomeClass\$Inner.class
SomeClass 匿名内部类 (in SomeClass.java)		SomeClass\$1.class, SomeClass\$2.class, 等等

第C.7节：引导类路径

普通的Java类加载器会先在引导类路径中查找类，然后才检查扩展和应用程序类路径。默认情况下，引导类路径由“rt.jar”文件和一些其他由JRE安装提供的重要JAR文件组成。这些文件提供了标准Java SE类库中的所有类，以及各种“内部”实现类。

在正常情况下，你不需要关心这些。默认情况下，像java、javac等命令会使用相应版本的运行时库。

- When the classpath is specified on the command line, it is simply a matter of using the appropriate shell quoting. For example:

```
export CLASSPATH="/home/user/My JAR Files/foo.jar:second.jar"
```

(The details may depend on the command shell that you use.)

- When the classpath is specified in a JAR file's a "MANIFEST.MF" file, URL encoding must be used.

```
Class-Path: /home/user/My%20JAR%20Files/foo.jar second.jar
```

Section C.5: Dynamic Classpath

Sometimes, just adding all the JARs from a folder isn't enough, for example when you have native code and need to select a subset of JARs. In this case, you need two main() methods. The first one builds a classloader and then uses this classloader to call the second main().

Here is an example which selects the correct SWT native JAR for your platform, adds all your application's JARs and then invokes the real main() method: [Create cross platform Java SWT Application](#)

Section C.6: Mappingclassnames to pathnames

The standard Java toolchain (and 3rd-party tools designed to interoperate with them) have specific rules for mapping the names of classes to the pathnames of files and other resources that represent them.

The mappings are as follows

- For classes in the default package, the pathnames are simple filenames.
- For classes in a named package, the package name components map to directories.
- For named nested and inner classes, the filename component is formed by joining the class names with a \$ character.
- For anonymous inner classes, numbers are used in place of names.

This is illustrated in the following table:

Classname	Source pathname	Classfile pathname
SomeClass	SomeClass.java	SomeClass.class
com.example.SomeClass	com/example/SomeClass.java	com/example/SomeClass.class
SomeClass.Inner	(in SomeClass.java)	SomeClass\$Inner.class
SomeClass 匿名内部类 (in SomeClass.java)		SomeClass\$1.class, SomeClass\$2.class, etc

Section C.7: The bootstrap classpath

The normal Java classloaders look for classes first in the bootstrap classpath, before checking for extensions and the application classpath. By default, the bootstrap classpath consists of the "rt.jar" file and some other important JAR files that are supplied by the JRE installation. These provide all of the classes in the standard Java SE class library, along with various "internal" implementation classes.

Under normal circumstances, you don't need to concern yourself with this. By default, commands like java, javac and so on will use the appropriate versions of the runtime libraries.

极少情况下，有必要通过使用标准库中类的替代版本来覆盖Java运行时的正常行为。例如，您可能会遇到运行时库中的“致命”错误，无法通过正常方式解决。在这种情况下，可以创建一个包含修改后类的JAR文件，然后在启动JVM时将其添加到引导类路径中。

java命令提供了以下-X选项，用于修改引导类路径：

- `-Xbootclasspath:<path>` 用提供的路径替换当前的引导类路径。
- `-Xbootclasspath/a:<path>` 将提供的路径追加到当前的引导类路径。
- `-Xbootclasspath/p:<path>` 将提供的路径预置到当前的引导类路径。

请注意，当使用引导类路径选项替换或覆盖Java类（等等）时，您实际上是在修改Java。如果随后分发您的代码，可能会涉及许可方面的问题。（请参阅Java二进制许可的条款和条件……并咨询律师。）

附录C.8：类路径的含义：搜索机制

类路径的目的是告诉JVM在哪里查找类和其他资源。类路径的含义与搜索过程密切相关。

类路径是一种搜索路径，指定了一系列位置以查找资源。在标准类路径中，这些位置可以是主机文件系统中的目录、JAR文件或ZIP文件。在每种情况下，该位置都是将被搜索的命名空间的根。

在类路径上搜索类的标准流程如下：

1. 将类名映射到相对的类文件路径名RP。类名到类文件名的映射关系在其他地方有描述。
2. 对于类路径中的每个条目E：
 - 如果该条目是文件系统目录：
 - 将RP相对于E解析，得到绝对路径名AP。
 - 测试AP是否为一个存在的文件路径。
 - 如果是，从该文件加载类
 - 如果该条目是JAR或ZIP文件：
 - 在JAR / ZIP文件索引中查找RP。
 - 如果对应的JAR / ZIP文件条目存在，则从该条目加载类。

在类路径上搜索资源的过程取决于资源路径是绝对路径还是相对路径。对于绝对资源路径，过程如上。对于使用`Class.getResource`或`Class.getResourceAsStream`，类包的路径会在搜索前被添加。

（注意，这些都是标准Java类加载器实现的过程。自定义类加载器可能会以不同方式执行搜索。）

Very occasionally, it is necessary to override the normal behavior of the Java runtime by using an alternative version of a class in the standard libraries. For example, you might encounter a "show stopper" bug in the runtime libraries that you cannot work around by normal means. In such a situation, it is possible to create a JAR file containing the altered class and then add it to the bootstrap classpath which launching the JVM.

The java command provides the following -X options for modifying the bootstrap classpath:

- `-Xbootclasspath:<path>` replaces the current boot classpath with the path provided.
- `-Xbootclasspath/a:<path>` appends the provided path to the current boot classpath.
- `-Xbootclasspath/p:<path>` prepends the provided path to the current boot classpath.

Note that when use the bootclasspath options to replace or override a Java class (etcetera), you are technically modifying Java. There *may be* licensing implications if you then distribute your code. (Refer to the terms and conditions of the Java Binary License ... and consult a lawyer.)

Section C.8: What the classpath means: how searches work

The purpose of the classpath is to tell a JVM where to find classes and other resources. The meaning of the classpath and the search process are intertwined.

The classpath is a form of search path which specifies a sequence of *locations* to look for resources. In a standard classpath, these places are either, a directory in the host file system, a JAR file or a ZIP file. In each cases, the location is the root of a *namespace* that will be searched.

The standard procedure for searching for a class on the classpath is as follows:

1. Map the class name to a relative classfile pathname RP. The mapping for class names to class filenames is described elsewhere.
2. For each entry E in the classpath:
 - If the entry is a filesystem directory:
 - Resolve RP relative to E to give an absolute pathname AP.
 - Test if AP is a path for an existing file.
 - If yes, load the class from that file
 - If the entry is a JAR or ZIP file:
 - Lookup RP in the JAR / ZIP file index.
 - If the corresponding JAR / ZIP file entry exists, load the class from that entry.

The procedure for searching for a resource on the classpath depends on whether the resource path is absolute or relative. For an absolute resource path, the procedure is as above. For a relative resource path resolved using `Class.getResource` or `Class.getResourceAsStream`, the path for the classes package is prepended prior to searching.

（Note these are the procedures implemented by the standard Java classloaders. A custom classloader might perform the search differently.）

附录D：资源（在类路径上）

Java允许检索存储在JAR文件中与编译类一起的基于文件的资源。本节重点介绍加载这些资源并使其可供代码使用。

D.1节：加载默认配置

读取默认配置属性：

```
包com.example;

public class ExampleApplication {
    private Properties getDefaults() throws IOException {
        Properties defaults = new Properties();

        try (InputStream defaultsStream =
ExampleApplication.class.getResourceAsStream("config.properties")) {
            defaults.load(defaultsStream);
        }

        return defaults;
    }
}
```

D.2节：从资源加载图像

加载捆绑图像的方法：

```
包com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}
```

D.3节：使用类加载器查找和读取资源

Java中的资源加载包括以下步骤：

1. 查找将用于查找资源的Class或ClassLoader。
2. 查找资源。
3. 获取资源的字节流。
4. 读取并处理字节流。
5. 关闭字节流。

最后三个步骤通常通过将URL传递给库方法或构造函数来加载资源来完成。此时通常会使用getResource方法。也可以在应用程序代码中读取资源数据，此时通常会使用getResourceAsStream方法。

绝对路径和相对路径的资源

Appendix D: Resources (on classpath)

Java allows the retrieval of file-based resources stored inside of a JAR alongside compiled classes. This topic focuses on loading those resources and making them available to your code.

Section D.1: Loading default configuration

To read default configuration properties:

```
package com.example;

public class ExampleApplication {
    private Properties getDefaults() throws IOException {
        Properties defaults = new Properties();

        try (InputStream defaultsStream =
ExampleApplication.class.getResourceAsStream("config.properties")) {
            defaults.load(defaultsStream);
        }

        return defaults;
    }
}
```

Section D.2: Loading an image from a resource

To load a bundled image:

```
package com.example;

public class ExampleApplication {
    private Image getIcon() throws IOException {
        URL imageURL = ExampleApplication.class.getResource("icon.png");
        return ImageIO.read(imageURL);
    }
}
```

Section D.3: Finding and reading resources using a classloader

Resource loading in Java comprises the following steps:

1. Finding the Class or ClassLoader that will find the resource.
2. Finding the resource.
3. Obtaining the byte stream for the resource.
4. Reading and processing the byte stream.
5. Closing the byte stream.

The last three steps are typically accomplished by passing the URL to a library method or constructor to load the resource. You will typically use a getResource method in this case. It is also possible to read the resource data in application code. You will typically use getResourceAsStream in this case.

Absolute and relative resource paths

可以从类路径加载的资源由path表示。路径的语法类似于UNIX/Linux文件路径。它由用正斜杠 (/) 分隔的简单名称组成。一个相对路径以名称开头，而一个绝对路径以分隔符开头。

正如类路径示例所描述的，JVM的类路径通过叠加类路径中目录和JAR或ZIP文件的命名空间来定义一个命名空间。当解析绝对路径时，类加载器将初始的/解释为命名空间的根。相比之下，相对路径可能相对于命名空间中的任何“文件夹”进行解析。所使用的文件夹取决于你用来解析路径的对象。

获取Class或ClassLoader

资源可以使用Class对象或ClassLoader对象定位。Class对象可以解析相对路径，因此如果你有一个（类）相对资源，通常会使用其中之一。有多种方式可以获取Class对象。例如：

- 一个类字面量会给你Java源代码中任何可命名类的Class对象；例如 String.class会给你String类型的Class对象。
- Object.getClass() 将为您提供任何对象类型的 Class 对象；例如，"hello".getClass() 是获取 String 类型的 Class 的另一种方式。
- Class.forName(String)方法将（如有必要）动态加载一个类并返回其Class对象；例如：Class.forName("java.lang.String")。

通常通过调用Class对象的getClassLoader()方法来获取一个ClassLoader对象。也可以使用静态方法ClassLoader.getSystemClassLoader()来获取JVM的默认类加载器。

get方法

一旦你拥有了一个Class或ClassLoader实例，就可以使用以下方法之一来查找资源：

方法	描述
ClassLoader.getResource(path) ClassLoader.getResources(path)	返回一个URL，表示具有给定路径的资源的位置。
ClassLoader.getResources(path) Class.getResources(path)	返回一个Enumeration<URL>，提供可用于定位foo.bar资源的URL；见下文。
ClassLoader.getResourceAsStream(path) Class.getResourceAsStream(path)	返回一个InputStream，可以从中以字节序列读取 foo.bar 资源的内容。

注意事项：

- ClassLoader和Class版本方法的主要区别在于相对路径的解释方式不同。
 - Class方法会在对应类包的“文件夹”中解析相对路径。
 - ClassLoader方法将相对路径视为绝对路径；即在类路径命名空间的“根文件夹”中解析它们。
- 如果请求的资源（或资源集合）找不到，getResource和getResourceAsStream方法将返回null，getResources方法将返回一个空的Enumeration`。
- 返回的URL可以使用URL.toStream()进行解析。它们可能是file: URL或其他常规URL，但如果资源位于JAR文件中，则它们将是jar: URL，标识JAR文件及其中的特定资源。

Resources that can be loaded from the classpath are denoted by a path. The syntax of the path is similar to a UNIX / Linux file path. It consists of simple names separated by forward slash (/) characters. A relative path starts with a name, and an absolute path starts with a separator.

As the Classpath examples describe, a JVM's classpath defines a namespace by overlaying the namespaces of the directories and JAR or ZIP files in the classpath. When an absolute path is resolved, it the classloaders interpret the initial / as meaning the root of the namespace. By contrast, a relative path may be resolved relative to any "folder" in the namespace. The folder used will depend on the object that you use to resolve the path.

Obtaining a Class or Classloader

A resource can be located using either a **Class** object or a **ClassLoader** object. A **Class** object can resolve relative paths, so you will typically use one of these if you have a (class) relative resource. There are a variety of ways to obtain a **Class** object. For example:

- A *class literal* will give you the **Class** object for any class that you can name in Java source code; e.g. `String.class` gives you the **Class** object for the `String` type.
- The `Object.getClass()` will give you the **Class** object for the type od any object; e.g. `"hello".getClass()` is another way to get **Class** of the `String` type.
- The `Class.forName(String)` method will (if necessary) dynamically load a class and return its **Class** object; e.g. `Class.forName("java.lang.String")`.

A **ClassLoader** object is typically obtained by calling `getClassLoader()` on a **Class** object. It is also possible to get hold of the JVM's default classloader using the static `ClassLoader.getSystemClassLoader()` method.

The get methods

Once you have a **Class** or **ClassLoader** instance, you can find a resource, using one of the following methods:

Methods	Description
<code>ClassLoader.getResource(path)</code> <code>ClassLoader.getResources(path)</code>	Returns a URL which represents the location of the resource with the given path.
<code>ClassLoader.getResources(path)</code> <code>Class.getResources(path)</code>	Returns an Enumeration<URL> giving the URLs which can be used to locate the foo.bar resource; see below.
<code>ClassLoader.getResourceAsStream(path)</code> <code>Class.getResourceAsStream(path)</code>	Returns an <code>InputStream</code> from which you can read the contents of the foo.bar resource as a sequence of bytes.

Notes:

- The main difference between the **ClassLoader** and **Class** versions of the methods is in the way that relative paths are interpreted.
 - The **Class** methods resolve a relative path in the "folder" that corresponds to the classes package.
 - The **ClassLoader** methods treat relative paths as if they were absolute; i.e. the resolve them in the "root folder" of the classpath namespace.
- If the requested resource (or resources) cannot be found, the `getResource` and `getResourceAsStream` methods `return null`, and the `getResources` methods `return` an empty `Enumeration``.
- The URLs returned will be resolvable using `URL.toStream()`. They could be file: URLs or other conventional URLs, but if the resource resides in a JAR file, they will be jar: URLs that identify the JAR file and a specific resource within it.

- 如果代码使用`getResourceAsStream`方法（或`URL.toStream()`）获取`InputStream`，则负责关闭该流对象。未关闭流可能导致资源泄漏。

- If your code uses a `getResourceAsStream` method (or `URL.toStream()`) to obtain an `InputStream`, it is responsible for closing the stream object. Failure to close the stream could lead to a resource leak.

D.4节：从多个

JAR中加载同名资源

在类路径中，可能存在具有相同路径和名称的资源位于多个JAR文件中。常见情况是遵循某种约定的资源或作为打包规范一部分的资源。此类资源的示例有

- META-INF/MANIFEST.MF
- META-INF/beans.xml (CDI 规范)
- 包含实现提供者的 ServiceLoader 属性

要访问不同 jar 中的所有这些资源，必须使用 ClassLoader，它有一个用于此的函数。

返回的Enumeration可以方便地使用 Collections 函数转换为List。

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```

Section D.4: Loading same-name resource from multiple JARs

Resource with same path and name may exist in more than one JAR file on the classpath. Common cases are resources following a convention or that are part of a packaging specification. Examples for such resources are

- META-INF/MANIFEST.MF
- META-INF/beans.xml (CDI Spec)
- ServiceLoader properties containing implementation providers

To get access to *all* of these resources in different jars, one has to use a ClassLoader, which has a method for this.

The returned `Enumeration` can be conveniently converted to a `List` using a Collections function.

```
Enumeration<URL> resEnum = MyClass.class.getClassLoader().getResources("META-INF/MANIFEST.MF");
ArrayList<URL> resources = Collections.list(resEnum);
```

鸣谢

非常感谢所有来自 Stack Overflow Documentation 的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

100rabh	第18章和第79章
17slim	第28章、第40章、第72章和第109章
1d0m3n30	第9章、第35章、第45章、第47章和第106章
3442	第23章
3751_Creator	第1章
4castle	第2章和第57章
阿·博施曼	第42章和第67章
A.J. 布朗	第11章
亚伦·迪古拉	第47章和第184章
亚伦·弗兰克	第56章
阿斯蒙德·埃尔德胡塞特	第46章
阿卜杜勒·哈利克	第90章
阿比杰特	第65章
阿比谢克·贾因	第11、23和79章
阿布巴卡尔	第11、23、57、67和102章
acdjunior	第23和57章
无限期	第23、24、33、43和73章
亚当·拉特兹曼	第11章
阿迪尔·安萨里	第182章
阿多拉斯	第79章和第164章
阿德里安·克雷布斯	第11章、第23章、第54章、第69章、第74章和第111章
afzalex	第23章
agilob	第11章、第23章和第69章
agoeb	第54章
艾登·迪奥姆	第11章
艾米·博尔达	第22章和第57章
aioobe	第35章和第117章
ajablonksi	第182章
AJNeufeld	第74章
akgren_soar	第58章
阿基尔·S·K	第1、66、69、117和182章
alain.janinm	第16、19和138章
阿莱克·米耶茨科夫斯基	第20、33和78章
亚历克斯·A	第182章
亚历克斯·迈堡	第11和47章
亚历克斯·谢斯特罗夫	第11章和第20章
亚历克斯·T.	第132章
亚历山大·格里莫	第181章
阿列克谢·拉古诺夫	第83章
阿列克谢·谢梅纽克	第47章和第117章
阿列克西	第149章
阿隆·G。	第43章
阿尔珀·Firat 卡亚	第77章
alphaloop	第144章
altomnr	第24章和第182章
阿马尼·基卢曼加	第10、11、35、40、80和111章
阿米特·古贾拉蒂	第12、29、30、31和180章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

100rabh	Chapters 18 and 79
17slim	Chapters 28, 40, 72 and 109
1d0m3n30	Chapters 9, 35, 45, 47 and 106
3442	Chapter 23
3751_Creator	Chapter 1
4castle	Chapters 2 and 57
A Boschman	Chapters 42 and 67
A.J. Brown	Chapter 11
Aaron Digulla	Chapters 47 and 184
Aaron Franke	Chapter 56
Aasmund Eldhuset	Chapter 46
ABDUL KHALIQ	Chapter 90
Abhijeet	Chapter 65
Abhishek Jain	Chapters 11, 23 and 79
Abubakkar	Chapters 11, 23, 57, 67 and 102
acdjunior	Chapters 23 and 57
Ad Infinitum	Chapters 23, 24, 33, 43 and 73
Adam Ratzman	Chapter 11
Adeel Ansari	Chapter 182
Adowrath	Chapters 79 and 164
Adrian Krebs	Chapters 11, 23, 54, 69, 74 and 111
afzalex	Chapter 23
agilob	Chapters 11, 23 and 69
agoeb	Chapter 54
Aiden Deom	Chapter 11
Aimee Borda	Chapters 22 and 57
aioobe	Chapters 35 and 117
ajablonksi	Chapter 182
AJNeufeld	Chapter 74
akgren_soar	Chapter 58
Akhil S K	Chapters 1, 66, 69, 117 and 182
alain.janinm	Chapters 16, 19 and 138
Alek Mieczkowski	Chapters 20, 33 and 78
Alex A	Chapter 182
Alex Meiburg	Chapters 11 and 47
Alex Shesterov	Chapters 11 and 20
Alex T.	Chapter 132
Alexandre Grimaud	Chapter 181
Alexey Lagunov	Chapter 83
alexey semenyuk	Chapters 47 and 117
Alexiy	Chapter 149
Alon .G.	Chapter 43
Alper Firat Kaya	Chapter 77
alphaloop	Chapter 144
altomnr	Chapters 24 and 182
Amani Kilumanga	Chapters 10, 11, 35, 40, 80 and 111
Amit Gujarathi	Chapters 12, 29, 30, 31 and 180

阿米特·古普塔
阿纳托利·亚基姆丘克
安德烈亚斯
安德烈亚斯·费斯特
安德鲁
安德鲁·安蒂波夫
安德鲁·布鲁克
安德鲁·斯克利亚列夫斯基
安德烈·阿布拉莫夫
安德罗宾
安迪·托马斯
阿尼·梅农
阿尼尔
ankidaemon
安基特·卡蒂亚尔
安库尔·阿南德
Anony
anotherGatsby
安东尼·雷蒙德
安东·赫利尼斯蒂
安东尼奥
anuvab1911
ar4ers
阿拉什
北极领主
阿尔西
阿尔卡迪
阿尔皮特·潘迪
砷砷
亚瑟
亚萨夫
阿山佩雷拉
阿西亚特
阿西利亚斯
AstroCB
ata
Athari
augray
Aurasphere
奥斯汀
奥斯汀·戴
A_Arnold
巴特·库梅尔
巴蒂
bcosynot
ben75
巴维克·帕特尔
比尔博·巴金斯
比莱什·甘古利
二进制极客
Blubberguy22
bn.
鲍勃·里弗斯

第73章
第23章
第40、99和106章
第35章
第23、40、73和79章
第88章
第74章
第35章
第41、57、73、75、81、117和134章
第33章
第11章、第80章、第85章和第130章
第1章、第42章、第56章和第182章
第23章
第23章
第73章、第122章和第141章
第1章
第10、11、24、35、47、73和89章
第23章
第182章
第130章
第1章和第23章
第42章和第182章
第52章
第122章
第103章和第105章
第16章和第68章
第1章和第54章
第21章
第57章
第11章
第74章
第41章
第73和126章
第23章
第28章和第77章
第23和57章
第42章
第77章
第11章
第11章
第11章、第16章、第18章和第26章
第57章
第66章
第35章
第97章
第69章
第17章
第10章和第66章
第28章
第25章和第40章
第100章
第10、16、18、24、35、70和123章

Amit Gupta
Anatoly Yakimchuk
Andreas
Andreas Fester
Andrew
Andrew Antipov
Andrew Brooke
Andrew Sklyarevsky
Andrii Abramov
Androbin
Andy Thomas
Ani Menon
Anil
ankidaemon
Ankit Katiyar
Ankur Anand
Anony
anotherGatsby
Anthony Raymond
Anton Hlinisty
antonio
anuvab1911
ar4ers
Arash
ArcticLord
arcy
Arkadiy
arpit pandey
ArsenArsen
Arthur
Asaph
AshanPerera
Asiat
assylias
AstroCB
ata
Athari
augray
Aurasphere
Austin
Austin Day
A_Arnold
Bart Kummel
Batty
bcosynot
ben75
Bhavik Patel
Bilbo Baggins
Bilesh Ganguly
Binary Nerd
Blubberguy22
bn.
Bob Rivers

Chapter 73
Chapter 23
Chapters 40, 99 and 106
Chapter 35
Chapters 23, 40, 73 and 79
Chapter 88
Chapter 74
Chapter 35
Chapters 41, 57, 73, 75, 81, 117 and 134
Chapter 33
Chapters 11, 80, 85 and 130
Chapters 1, 42, 56 and 182
Chapter 23
Chapter 23
Chapters 73, 122 and 141
Chapter 1
Chapters 10, 11, 24, 35, 47, 73 and 89
Chapter 23
Chapter 182
Chapter 130
Chapters 1 and 23
Chapters 42 and 182
Chapter 52
Chapter 122
Chapters 103 and 105
Chapters 16 and 68
Chapters 1 and 54
Chapter 21
Chapter 57
Chapters 23, 71, 72, 77, 87, 93 and 110
Chapter 11
Chapter 74
Chapter 41
Chapters 73 and 126
Chapter 23
Chapters 28 and 77
Chapters 23 and 57
Chapter 42
Chapter 77
Chapter 11
Chapter 11
Chapter 11
Chapters 11, 16, 18 and 26
Chapter 57
Chapter 66
Chapter 35
Chapter 97
Chapter 69
Chapter 17
Chapters 10 and 66
Chapter 28
Chapters 25 and 40
Chapter 100
Chapters 10, 16, 18, 24, 35, 70 and 123

[鲍巴斯·佩特](#)
[博赫丹·科林尼](#)
[波希米亚的](#)
[博摩尔](#)
[博žo 斯托伊科维ć](#)
[bpoiss](#)
[布伦登·杜根](#)
[布雷特·凯尔](#)
[布赖恩·戈茨](#)
[布鲁诺DM](#)
[巴迪](#)
[布尔克哈德](#)
[bwegs](#)
[c.uent](#)
[c1phr](#)
[Cache Staheli](#)
[CaffeineToCode](#)
[Caleb Brinkman](#)
[Caner Balım](#)
[carloabelli](#)
[Carlton](#)
[CarManuel](#)
[卡特·布雷纳德](#)
[卡斯·埃利恩斯](#)
[卡塔利娜岛](#)
[cdm](#)
[ced](#)
[charlesreid1](#)
[查理·H](#)
[切蒂亚](#)
[奇拉格·帕尔马尔](#)
[克里斯·米德格利](#)
[克里斯蒂安](#)
[克里斯蒂安·威尔基](#)
[克里斯托夫·韦斯](#)
[克里斯托弗·施奈德](#)
[克劳迪娅](#)
[克劳迪奥](#)
[clinomaniac](#)
[code11](#)
[Codebender](#)
[编码员](#)
[咖啡忍者](#)
[咖啡馆编码员](#)
[冷火](#)
[compuhosny](#)
[Confiqure](#)
[君士坦丁](#)
[科西嘉](#)
[精工购物车](#)
[板球_007](#)
[网络科学家](#)
[C_LS](#)

第85章
第107章
第15、45和54章
第17、67和126章
第23章
第23和57章
第129章
第42章
第73章
第41章
第74章
第1、6、7、11、23、54、66、69、77和107章
第23章
第40章
第23章
第11章、第13章、第23章、第47章和第107章
第42章
第6章和第74章
第11章
第11章和第74章
第1章
第71章
第6章
第74章
第46章
第70章
第24、25、26、41和80章
第85章
第1章、第23章和第67章
第126章
第24章、第26章、第78章和第86章
第40章
第19章
第16章
第84章、第103章和第115章
第23章
第40章
第57章
第24章
第47章
第24、41和47章
第11章、第23章和第35章
第98章
第1章和第74章
第23章
第23章
第151章和第152章
第1章、第23章、第48章、第85章、第175章和第182章
第35章和第57章
第126章
第5章、第74章和第182章
第42章
第11章
第1章

[Bobas_Pett](#)
[Bohdan Korinnyi](#)
[Bohemian](#)
[bowmore](#)
[Božo Stojković](#)
[bpoiss](#)
[Brendon Dugan](#)
[Brett Kail](#)
[Brian Goetz](#)
[BrunoDM](#)
[Buddy](#)
[Burkhard](#)
[bwegs](#)
[c.uent](#)
[c1phr](#)
[Cache Staheli](#)
[CaffeineToCode](#)
[Caleb Brinkman](#)
[Caner Balım](#)
[carloabelli](#)
[Carlton](#)
[CarManuel](#)
[Carter Brainerd](#)
[Cas Eliëns](#)
[Catalina Island](#)
[cdm](#)
[ced](#)
[charlesreid1](#)
[Charlie H](#)
[Chetya](#)
[Chirag Parmar](#)
[Chris Midgley](#)
[Christian](#)
[Christian Wilkie](#)
[Christophe Weis](#)
[Christopher Schneider](#)
[Claudia](#)
[Claudio](#)
[clinomaniac](#)
[code11](#)
[Codebender](#)
[coder](#)
[Coffee Ninja](#)
[Coffeehouse Coder](#)
[Cold Fire](#)
[compuhosny](#)
[Confiqure](#)
[Constantine](#)
[corsiKa](#)
[CraftedCart](#)
[cricket_007](#)
[cyberscientist](#)
[C_LS](#)

[D D](#)
[丹尼尔](#)
[丹尼尔·凯弗](#)
[丹尼尔·林](#)
[丹尼尔·M.](#)
[丹尼尔·纽金特](#)
[丹尼尔·斯特拉多夫斯基](#)
[丹尼尔·怀尔德](#)
[达尼洛·吉马良斯](#)
[达留什](#)
[DarkV1](#)
[Datagrammar](#)
[戴夫·兰詹](#)
[大卫·格林伯格](#)
[大卫·索罗科](#)
[深度编码器](#)
[恶魔冷雾](#)
[恶魔魔像](#)
[desilijic](#)
[devguy](#)
[devnull69](#)
[DimaSan](#)
[dimo414](#)
[显示名称](#)
[德米特里·科托夫](#)
[dnup1092](#)
[杜如薇](#)
[DonyorM](#)
[dorukayhan](#)
[德拉肯](#)
[Drizzt321](#)
[呃](#)
[杜尔格帕尔·辛格](#)
[杜什科·约瓦诺夫斯基](#)
[杜什曼](#)
[DVarga](#)
[dwursteisen](#)
[迪兰](#)
[ebo](#)
[爱德华·维尔奇](#)
[Eilit](#)
[EJP](#)
[ekaerovets](#)
[以拉扎尔](#)
[埃米尔·谢尔查加](#)
[艾米莉·马布雷](#)
[无情的香蕉](#)
[埃姆雷·博拉特](#)
[伊纳穆尔·哈桑](#)
[工程师福阿德](#)
[engineercoding](#)
[Enigo](#)
[enrico.bacis](#)

[第22章](#)
[第10章](#)
[第23章](#)
[第48章和第111章](#)
[第35章、第46章、第57章、第73章、第80章、第112章和第177章](#)
[第81、114、130和139章](#)
[第11、23、24、26和57章](#)
[第116章](#)
[第35和163章](#)
[第23、43、54、57、82、91、102、126、128和138章](#)
[第1章、第11章、第23章和第86章](#)
[第95章](#)
[第48章](#)
[第54章](#)
[第87章](#)
[第1章](#)
[第43章](#)
[第26章](#)
[第176章](#)
[第79章](#)
[第93章](#)
[第26、33、50和126章](#)
[第69章、第120章和第131章](#)
[第113章和第170章](#)
[第28章](#)
[第10章和第11章](#)
[第6章和第10章](#)
[第54章、第57章和第93章](#)
[第11、20、76、87、133和134章](#)
[第73章](#)
[第43和103章](#)
[第23章](#)
[第23和28章](#)
[第43、57和130章](#)
[第111、114和173章](#)
[第11、23、47、54、57和79章](#)
[第57章](#)
[第41章](#)
[第74章](#)
[第120章](#)
[第23章](#)
[第66和145章](#)
[第97章](#)
[第42章](#)
[第23章、第42章、第115章和第182章](#)
[第28章和第185章](#)
[第66章和第120章](#)
[第1章和第23章](#)
[第56章](#)
[第23章](#)
[第33章](#)
[第13章、第24章、第28章、第42章和第77章](#)
[第1章、第11章、第23章和第57章](#)

[D D](#)
[Daniel](#)
[Daniel Käfer](#)
[Daniel LIn](#)
[Daniel M.](#)
[Daniel Nugent](#)
[Daniel Stradowski](#)
[Daniel Wild](#)
[Danilo Guimaraes](#)
[Dariusz](#)
[DarkV1](#)
[Datagrammar](#)
[Dave Ranjan](#)
[David Grinberg](#)
[David Soroko](#)
[DeepCoder](#)
[Demon Coldmist](#)
[demongolem](#)
[desilijic](#)
[devguy](#)
[devnull69](#)
[DimaSan](#)
[dimo414](#)
[Display Name](#)
[Dmitriy Kotov](#)
[dnup1092](#)
[Do Nhu Vy](#)
[DonyorM](#)
[dorukayhan](#)
[Draken](#)
[Drizzt321](#)
[Duh](#)
[Durgpal Singh](#)
[Dushko Jovanovski](#)
[Dushman](#)
[DVarga](#)
[dwursteisen](#)
[Dylan](#)
[ebo](#)
[Eduard Wirch](#)
[Eilit](#)
[EJP](#)
[ekaerovets](#)
[Elazar](#)
[Emil Sierżega](#)
[Emily Mabrey](#)
[emotionlessbananas](#)
[Emre Bolat](#)
[Enamul Hassan](#)
[Eng.Fouad](#)
[engineercoding](#)
[Enigo](#)
[enrico.bacis](#)

Enwired	第132章	Enwired	Chapter 132
Eran	第1章、第23章和第24章	Eran	Chapters 1, 23 and 24
erickson	第117章	erickson	Chapter 117
埃里克·米纳里尼	第23章	Erik Minarini	Chapter 23
埃尔坎·哈斯普拉特	第56章	Erkan Haspulat	Chapter 56
esin88	第22、172和179章	esin88	Chapters 22, 172 and 179
影响	第23章	Etki	Chapter 23
explv	第16、23、54和57章	explv	Chapters 16, 23, 54 and 57
F. 斯蒂芬·Q	第87和143章	F. Stephen Q	Chapters 87 and 143
法比安	第10、13、40、43、48、74、80、85、91、103、117、122、136和174章	fabian	Chapters 10, 13, 40, 43, 48, 74, 80, 85, 91, 103, 117, 122, 136 and 174
faraa	第47章	faraa	Chapter 47
FFY00	第137章	FFY00	Chapter 137
fgb	第13章	fgb	Chapter 13
fikovnik	第67章	fikovnik	Chapter 67
Fildor	第3、116、126和127章	Fildor	Chapters 3, 116, 126 and 127
菲利普·斯莫拉	第2章	Filip Smola	Chapter 2
飞行派怪兽	第40、77和122章	FlyingPiMonster	Chapters 40, 77 and 122
FMC	第69章	FMC	Chapter 69
foxt7ot	第150章	foxt7ot	Chapter 150
弗朗切斯科·门扎尼	第1、10和97章	Francesco Menzani	Chapters 1, 10 and 97
弗雷迪·科尔曼	第126章	Freddie Coleman	Chapter 126
弗里德里克	第132章	Friederike	Chapter 132
Functino	第1章和第23章	Functino	Chapters 1 and 23
futureelite7	第40章	futureelite7	Chapter 40
f_puras	第35章	f_puras	Chapter 35
加尔·德雷曼	第23、35、57、69、73、130、131、132和183章	Gal Dreiman	Chapters 23, 35, 57, 69, 73, 130, 131, 132 and 183
gar	第10和73章	gar	Chapters 10 and 73
garg10may	第1章	garg10may	Chapter 1
加雷斯·戈尔丁	第19章	Garrett Golding	Chapter 19
高塔姆·乔斯	第182章	Gautam Jose	Chapter 182
吉恩·马林	第23章、第35章和第178章	Gene Marin	Chapters 23, 35 and 178
geniushkg	第54章	geniushkg	Chapter 54
乔治·贝利	第6章和第41章	George Bailey	Chapters 6 and 41
杰拉尔德·穆克	第6章、第22章、第77章、第130章、第136章和第185章	Gerald Mücke	Chapters 6, 22, 77, 130, 136 and 185
GhostCat	第43章	GhostCat	Chapter 43
吉汉·查图兰加	第86章	Gihan Chathuranga	Chapter 86
姜头	第1章和第23章	GingerHead	Chapters 1 and 23
giucal	第117章	giucal	Chapter 117
glee8e	第93章	glee8e	Chapter 93
gontard	第57章	gontard	Chapter 57
GPI	第24、28、63、73、81、108、126、135和184章	GPI	Chapters 24, 28, 63, 73, 81, 108, 126, 135 and 184
GradAsso	第66章	GradAsso	Chapter 66
granmirupa	第23和25章	granmirupa	Chapters 23 and 25
格雷 (Gray)	第11章	Gray	Chapter 11
绿巨人 (GreenGiant)	第11章、第69章和第88章	GreenGiant	Chapters 11, 69 and 88
Grexis	第35章和第146章	Grexis	Chapters 35 and 146
格热戈日·奥莱兹基	第57章	Grzegorz Oledzki	Chapter 57
Gubbel	第58章	Gubbel	Chapter 58
吉列尔梅·托雷斯·卡斯特罗	第23章	Guilherme Torres Castro	Chapter 23
古斯塔沃·科埃略	第23章	Gustavo Coelho	Chapter 23
gwintrob	第67章	gwintrob	Chapter 67
吉蒂斯·特诺维马斯	第1章和第23章	Gytis Tenovimas	Chapters 1 and 23
hamena314	第11章和第85章	hamena314	Chapters 11 and 85

汉克·D	第57章和第73章	Hank D	Chapters 57 and 73
王革	第77章	Hay	Chapter 77
哈泽姆·法拉哈特	第81章	Hazem Farahat	Chapter 81
HCarrasko	第116章	HCarrasko	Chapter 116
hellrocker	第126章	hellrocker	Chapter 126
hexafraction	第47章、第69章、第73章、第126章和第144章	hexafraction	Chapters 47, 69, 73, 126 and 144
hirosh	第54章	hirosh	Chapter 54
Holger	第137章	Holger	Chapter 137
HON95	第11章	HON95	Chapter 11
HTNW	第82章和第130章	HTNW	Chapters 82 and 130
Hulk	第102章	Hulk	Chapter 102
hzpz	第67章	hzpz	Chapter 67
ldcmp	第45章	ldcmp	Chapter 45
iliketocode	第1章、第11章、第23章和第57章	iliketocode	Chapters 1, 11, 23 and 57
伊利亚	第11章、第23章、第82章、第93章、第97章、第118章和第126章	Ilya	Chapters 11, 23, 82, 93, 97, 118 and 126
Infuzion	第11章	Infuzion	Chapter 11
InitializeSahib	第97章	InitializeSahib	Chapter 97
inovaovao	第79章	inovaovao	Chapter 79
intboolstring	第23章、第46章、第74章和第79章	intboolstring	Chapters 23, 46, 74 and 79
因齐玛姆·塔里克 IT	第74章	Inzimam Tariq IT	Chapter 74
ipsi	第1章、第10章、第35章、第171章和第182章	ipsi	Chapters 1, 10, 35, 171 and 182
iqbal_cs	第140章	iqbal_cs	Chapter 140
Ironcache	第3章和第166章	Ironcache	Chapters 3 and 166
伊万·维尔吉列夫	第73章	Ivan Vergiliev	Chapter 73
J·阿特金	第10章、第28章、第43章、第57章、第67章、第73章和第89章	J Atkin	Chapters 10, 28, 43, 57, 58, 67, 73 and 89
杰雷米·博尔杜克	第23章	Jérémie Bolduc	Chapter 23
J. 皮查多	第150章	J. Pichardo	Chapter 150
J.D. 桑迪弗	第117章	J.D. Sandifer	Chapter 117
贾比尔	第11、16、24、28、69、72、86和91章	Jabir	Chapters 11, 16, 24, 28, 69, 72, 86 and 91
雅各布·G.	第61章	Jacob G.	Chapter 61
杰克D	第10章	JakeD	Chapter 10
詹姆斯·詹森	第77章	James Jensen	Chapter 77
詹姆斯·拉奇	第42、126和133章	james large	Chapters 42, 126 and 133
詹姆斯·奥斯瓦尔德	第79章	James Oswald	Chapter 79
詹姆斯·泰勒	第1章和第23章	James Taylor	Chapters 1 and 23
JamesENL	第42章和第53章	JamesENL	Chapters 42 and 53
扬·弗拉基米尔·莫斯特特	第25章、第47章和第79章	Jan Vladimir Mostert	Chapters 25, 47 and 79
雅诺什	第25章和第89章	janos	Chapters 25 and 89
贾里德·胡珀	第35章	Jared Hooper	Chapter 35
jatanp	第178章	jatanp	Chapter 178
贾廷·巴洛迪	第5章	Jatin Balodhi	Chapter 5
javac	第11章	javac	Chapter 11
JAVAC	第81章	JAVAC	Chapter 81
JavaHopper	第1章、第23章、第40章、第57章、第69章、第79章和第85章	JavaHopper	Chapters 1, 23, 40, 57, 69, 79 and 85
Javant	第23章和第85章	Javant	Chapters 23 and 85
哈维尔·迪亚兹	第28章	Javier Diaz	Chapter 28
jayantS	第56章	jayantS	Chapter 56
JD9999	第25章和第59章	JD9999	Chapters 25 and 59
让	第16章	Jean	Chapter 16
让·维托尔	第1章	Jean Vitor	Chapter 1
吉特	第153章	Jeet	Chapter 153
杰夫·科尔曼	第182章	Jeff Coleman	Chapter 182
杰弗里·博斯布姆	第23、28、52和54章	Jeffrey Bosboom	Chapters 23, 28, 52 and 54

杰弗里·林	第1和11章	Jeffrey Lin	Chapters 1 and 11
延斯·绍德	第1、23、47、48、58、69、74、88、126和127章	Jens Schauder	Chapters 1, 23, 47, 48, 58, 69, 74, 88, 126 and 127
杰罗恩·范德维尔德	第73章	Jeroen Vandevelde	Chapter 73
Jeutnarg	第23章	Jeutnarg	Chapter 23
吉姆·加里森	第23章	Jim Garrison	Chapter 23
吉滕德拉·瓦尔什尼	第23章	jitendra varshney	Chapter 23
jmattheis	第23章	jmattheis	Chapter 23
乔·C	第121章	Joe C	Chapter 121
约翰内斯	第23、35、79和126章	Johannes	Chapters 23, 35, 79 and 126
约翰·迪菲尼	第64章和第161章	John DiFini	Chapters 64 and 161
约翰·弗格斯	第1章	John Fergus	Chapter 1
约翰·纳什	第19章、第21章、第58章和第142章	John Nash	Chapters 19, 21, 58 and 142
约翰·斯莱格斯	第23章	John Slegers	Chapter 23
约翰·斯塔里奇	第135章	John Starich	Chapter 135
johnnyaug	第28章	johnnyaug	Chapter 28
jojodmo	第10章、第11章、第23章、第40章和第79章	jojodmo	Chapters 10, 11, 23, 40 and 79
乔恩·埃里克森	第57章	Jon Erickson	Chapter 57
JonasCz	第11章、第69章、第74章、第78章、第86章和第95章	JonasCz	Chapters 11, 69, 74, 78, 86 and 95
Jonathan	第1章、第23章、第28章、第46章、第57章、第79章、第84章、第88章和第125章	Jonathan	Chapters 1, 23, 28, 46, 57, 79, 84, 88 and 125
Jonathan Barbero	第96章和第119章	Jonathan Barbero	Chapters 96 and 119
Jonathan Lam	第23章和第182章	Jonathan Lam	Chapters 23 and 182
JonK	第88章	JonK	Chapter 88
jopasserat	第25章	jopasserat	Chapter 25
乔迪·卡斯蒂利亚	第11章	Jordi Castilla	Chapter 11
乔迪·贝拉克	第77章	Jordy Baylac	Chapter 77
乔雷尔·阿里	第71章	Jorel Ali	Chapter 71
约恩·弗尼	第4、11、42、43、47、54、57、75、79、104、126和135章	Jorn Vernee	Chapters 4, 11, 42, 43, 47, 54, 57, 75, 79, 104, 126 and 135
乔舒亚·卡莫迪	第2章	Joshua Carmody	Chapter 2
JStef	第23章	JStef	Chapter 23
裘德·尼罗山	第11、57、67和73章	Jude Niroshan	Chapters 11, 57, 67 and 73
判断与不判断	第13章和第73章	JudgingNotJudging	Chapters 13 and 73
尤尔根·D	第74章	juergen_d	Chapter 74
jwd630	第177章	jwd630	Chapter 177
K"	第184章	K"	Chapter 184
k3b	第35章	k3b	Chapter 35
kaartic	第1章	kaartic	Chapter 1
凯	第52、54、69和79章	Kai	Chapters 52, 54, 69 and 79
kajacx	第138章	kajacx	Chapter 138
kann	第70章	kann	Chapter 70
kaotikmynd	第80章	kaotikmynd	Chapter 80
Kapep	第11、43和57章	Kapep	Chapters 11, 43 and 57
KartikKannapur	第28章	KartikKannapur	Chapter 28
kasperjj	第97章	kasperjj	Chapter 97
Kaushal28	第20章、第26章和第86章	Kaushal28	Chapters 20, 26 and 86
考希克·NP	第11章	Kaushik NP	Chapter 11
kcoppock	第47章	kcoppock	Chapter 47
KdgDev	第10章	KdgDev	Chapter 10
凯尔文·凯尔纳	第159章	Kelvin Kellner	Chapter 159
肯·Y	第122章	Ken Y	Chapter 122
肯斯特	第11、25、28和63章	Kenster	Chapters 11, 25, 28 and 63
凯文·迪特拉格利亚	第54章	Kevin DiTraglia	Chapter 54
凯文·拉奥菲	第73章	Kevin Raoofi	Chapter 73
凯文·索恩	第6章、第23章、第40章和第69章	Kevin Thorne	Chapters 6, 23, 40 and 69

奇钦
[k i e d y s k t o s](#)
基尼奥利安
基普
[基兰·库马尔·马塔姆](#)
基里尔·索科洛夫
[基肖尔·图尔西亚尼](#)
克里斯蒂娜
[克日什托夫·克拉索ń](#)
[k s t a n d e l l](#)
[K u d z i e C h a s e](#)
[黑田](#)
[拉克兰·道丁](#)
[兰基马特](#)
劳雷尔
利亚奎
[学习曲线](#)
[Li357](#)
[利朱·托马斯](#)
[l l a m o s i t o p i a](#)
[洛里斯·塞库罗](#)
[卢安·尼科](#)
[卢卡斯·克努斯](#)
[M_M](#)
[马尔滕·博德韦斯](#)
[M a c 7 0](#)
[m a d x](#)
诚
[马肯](#)
[马拉夫](#)
马尔特
[马尼什·科塔里](#)
[马努蒂](#)
[曼努埃尔·斯皮戈隆](#)
[曼努埃尔·维达](#)
马克
[马克·格林](#)
[马克·斯图尔特](#)
[马克·伊斯里](#)
[马龙](#)
[马丁·弗兰克](#)
[马文](#)
[大师爆破手](#)
[马塔斯·瓦伊特凯维休斯](#)
[马特·克里普纳](#)
[m_a_t_e_u_s_c_b](#)
[马特塞曼](#)
马特
[马特·克拉克](#)
[马特·弗里克](#)
[马修·特劳特](#)
[马蒂亚斯·布劳恩](#)
[马修](#)

第87章
第95章
第22章
第58章
第27章、第30章、第36章、第37章、第39章、第49章、第53章、第58章、第59章和第92章
第89章
第164章
第45章
第25章
第11、42和79章
第24章
第18章
第86章
第56章
第79、80和86章
第77章
第111章
第106章
第23、128和169章
第23章
第19章，第23章和第56章
第23章，第54章，第89章和第103章
第102章
第14章
第20章，第35章，第86章和第140章
第40章
第1章、第35章和第103章
第23章、第70章和第74章
第23章
第11章
第23章、第32章、第73章、第88章、第93章和第126章
第19章和第128章
第109、156和168章
第11章
第18章
第1和47章
第73章
第5章
第46和71章
第24章
第24章
第11章和第23章
第48章、第57章和第87章
第23章
第126章
第77章
第88章
第1章、第23章和第47章
第11章、第16章、第58章和第86章
第20章、第43章和第52章
第73章
第11章
第94章

[Kichiin](#)
[kiedyskatos](#)
[Kineolyan](#)
[Kip](#)
[KIRAN KUMAR MATAM](#)
[Kirill Sokolov](#)
[Kishore Tulsiani](#)
[kristyna](#)
[Krzysztof Krasoń](#)
[kstandell](#)
[KudzieChase](#)
[Kuroda](#)
[Lachlan Dowding](#)
[Lankymart](#)
[Laurel](#)
[leaqui](#)
[Lernkurve](#)
[Li357](#)
[Liju Thomas](#)
[llamositopia](#)
[Loris Securo](#)
[Luan Nico](#)
[Lukas Knuth](#)
[M M](#)
[Maarten Bodewes](#)
[Mac70](#)
[madx](#)
[Makoto](#)
[Makyen](#)
[Malav](#)
[Malt](#)
[Manish Kothari](#)
[manouti](#)
[Manuel Spigolon](#)
[Manuel Vieda](#)
[Marc](#)
[Mark Green](#)
[Mark Stewart](#)
[Mark Yisri](#)
[Maroun](#)
[Martin Frank](#)
[Marvin](#)
[MasterBlaster](#)
[Matas Vaitkevicius](#)
[Matěj Kripner](#)
[mateuscb](#)
[Matsemann](#)
[Matt](#)
[Matt Clark](#)
[matt freake](#)
[Matthew Trout](#)
[Matthias Braun](#)
[Matthieu](#)

Chapter 87
Chapter 95
Chapter 22
Chapter 58
Chapters 27, 30, 36, 37, 39, 49, 53, 58, 59 and 92
Chapter 89
Chapter 164
Chapter 45
Chapter 25
Chapters 11, 42 and 79
Chapter 24
Chapter 18
Chapter 86
Chapter 56
Chapters 79, 80 and 86
Chapter 77
Chapter 111
Chapter 106
Chapters 23, 128 and 169
Chapter 23
Chapters 19, 23 and 56
Chapters 23, 54, 89 and 103
Chapter 102
Chapter 14
Chapters 20, 35, 86 and 140
Chapter 40
Chapters 1, 35 and 103
Chapters 23, 70 and 74
Chapter 23
Chapter 11
Chapters 23, 32, 73, 88, 93 and 126
Chapters 19 and 128
Chapters 109, 156 and 168
Chapter 11
Chapter 18
Chapters 1 and 47
Chapter 73
Chapter 5
Chapters 46 and 71
Chapter 24
Chapter 24
Chapters 11 and 23
Chapters 48, 57 and 87
Chapter 23
Chapter 126
Chapter 77
Chapter 88
Chapters 1, 23 and 47
Chapters 11, 16, 58 and 86
Chapters 20, 43 and 52
Chapter 73
Chapter 11
Chapter 94

[马克西姆·克雷希申](#)
[马克西姆·普列瓦科](#)
[马克西米利安·劳迈斯特](#)
[mayha](#)
[mayojava](#)
[MBorsch](#)
[Md. 纳西尔·乌丁·布伊扬](#)
[迈克尔](#)
[迈克尔·迈尔斯](#)
[迈克尔·皮费尔](#)
[迈克尔·冯·文克斯特恩](#)
[迈克尔·怀尔斯](#)
[michaelbahr](#)
[米哈乌·雷巴克](#)
[米奇助记符](#)
[MikeW](#)
[迈尔斯](#)
[米尔廷·米基奇](#)
[米穆尼](#)
[米米奇](#)
[Mine_Stone](#)
[米纳斯·卡迈尔](#)
[米罗斯拉夫·布拉迪奇](#)
[米奇·塔尔马奇](#)
[mnoronha](#)
[莫·阿什法克](#)
[穆罕默德·法德尔](#)
[姆鲁纳尔·帕格尼斯](#)
[姆什尼克](#)
[姆斯津博尔斯基](#)
[穆罕默德·雷法特](#)
[穆昆德](#)
[穆拉特·K.](#)
[穆雷尼克](#)
[武藤](#)
[尼古拉·亚申科](#)
[迈里迪姆](#)
[纳根](#)
[纳格什·拉基内帕利](#)
[纳姆舒布作家](#)
[纳雷什·库马尔](#)
[内森尼尔·福特](#)
[NatNgs](#)
[奈由纪](#)
[ncmathsadist](#)
[Nef10](#)
[neohope](#)
[nhahtdh](#)
[nicael](#)
[尼古拉斯·J·帕内拉](#)
[尼克·唐纳利](#)
[nickguletskii](#)
[Nicktar](#)

第23章
第11章和第23章
第23章
第11章
第10章和第85章
第10章
第20章和第126章
第11章和第58章
第35章和第103章
第23、45和126章
第7章
第67章
第28、69和87章
第135章
第35章
第79章
第11、16和17章
第23、42、69和122章
第23章
第23章
第93章
第23章
第88章
第1章和第23章
第1章
第28章
第23章
第107章和第116章
第47章、第50章、第54章和第81章
第19章
第23章和第54章
第1章
第126章
第57章和第59章
第57章
第44章
第69章
第23、35、40、42、46、55、95、111和122章
第6章
第16章和第88章
第57章
第28章
第47章和第126章
第22章、第23章、第42章和第89章
第73章
第35章
第145章
第80章
第23章
第107章
第2章
第126章
第16章、第42章和第85章

[Maxim Kreschishin](#)
[Maxim Plevako](#)
[Maximillian Laumeister](#)
[mayha](#)
[mayojava](#)
[MBorsch](#)
[Md. Nasir Uddin Bhuiyan](#)
[Michael](#)
[Michael Myers](#)
[Michael Piefel](#)
[Michael von Wenckstern](#)
[Michael Wiles](#)
[michaelbahr](#)
[Michał Rybak](#)
[Mick Mnemonic](#)
[MikeW](#)
[Miles](#)
[Miljen Mikic](#)
[Mimouni](#)
[Mimyck](#)
[Mine_Stone](#)
[Minhas Kamal](#)
[Miroslav Bradic](#)
[Mitch Talmadge](#)
[mnoronha](#)
[Mo.Ashfaq](#)
[Mohamed Fadhl](#)
[Mrunal Pagnis](#)
[Mshnik](#)
[mszymborski](#)
[Muhammed Refaat](#)
[Mukund](#)
[Murat K.](#)
[Murenik](#)
[Muto](#)
[Mykola Yashchenko](#)
[Myridium](#)
[NageN](#)
[Nagesh Lakinepally](#)
[NamshubWriter](#)
[Naresh Kumar](#)
[Nathaniel Ford](#)
[NatNgs](#)
[Nayuki](#)
[ncmathsadist](#)
[Nef10](#)
[neohope](#)
[nhahtdh](#)
[nicael](#)
[Nicholas J Panella](#)
[Nick Donnelly](#)
[nickguletskii](#)
[Nicktar](#)

Chapter 23
Chapters 11 and 23
Chapter 23
Chapter 11
Chapters 10 and 85
Chapter 10
Chapters 20 and 126
Chapters 11 and 58
Chapters 35 and 103
Chapters 23, 45 and 126
Chapter 7
Chapter 67
Chapters 28, 69 and 87
Chapter 135
Chapter 35
Chapter 79
Chapters 11, 16 and 17
Chapters 23, 42, 69 and 122
Chapter 23
Chapter 23
Chapter 93
Chapter 23
Chapter 88
Chapters 1 and 23
Chapter 1
Chapter 28
Chapter 23
Chapters 107 and 116
Chapters 47, 50, 54 and 81
Chapter 19
Chapters 23 and 54
Chapter 1
Chapter 126
Chapters 57 and 59
Chapter 57
Chapter 44
Chapter 69
Chapters 23, 35, 40, 42, 46, 55, 95, 111 and 122
Chapter 6
Chapters 16 and 88
Chapter 57
Chapter 28
Chapters 47 and 126
Chapters 22, 23, 42 and 89
Chapter 73
Chapter 35
Chapter 145
Chapter 80
Chapter 23
Chapter 107
Chapter 2
Chapter 126
Chapters 16, 42 and 85

尼希尔·R	第158章和第160章	Nikhil R	Chapters 158 and 160
尼基塔·库尔廷	第69章和第107章	Nikita Kurtin	Chapters 69 and 107
尼克拉斯·罗森克兰茨	第162章	Niklas Rosencrantz	Chapter 162
NikolaB	第11章	NikolaB	Chapter 11
Nishant123	第16章	Nishant123	Chapter 16
nishizawa23	第148章	nishizawa23	Chapter 148
Nithanim	第1章、第128章和第182章	Nithanim	Chapters 1, 128 and 182
niyasc	第23章	niyasc	Chapter 23
nobeh	第73章	nobeh	Chapter 73
Nolequen	第35、43和81章	Nolequen	Chapters 35, 43 and 81
noscreenname	第66和127章	noscreenname	Chapters 66 and 127
Nufail	第20章	Nufail	Chapter 20
努里·塔斯德米尔	第1、11、23、40和57章	Nuri Tasdemir	Chapters 1, 11, 23, 40 and 57
nyarasha	第1章	nyarasha	Chapter 1
奥克拉科克	第23章	Ocracoke	Chapter 23
OldCurmudgeon	第35章	OldCurmudgeon	Chapter 35
OldMcDonald	第54章	OldMcDonald	Chapter 54
奥列格·斯克利亚尔	第24、25、47和54章	Oleg Sklyar	Chapters 24, 25, 47 and 54
OliPro007	第35章	OliPro007	Chapter 35
奥马尔·阿亚拉	第114章	Omar Ayala	Chapter 114
奥努尔	第11、23、47、66和122章	Onur	Chapters 11, 23, 47, 66 and 122
orccrusher99	第23章	orccrusher99	Chapter 23
Ordiel	第157章	Ordiel	Chapter 157
奥尔托马拉·洛克尼	第40、43、47和57章	Ortomala Lokni	Chapters 40, 43, 47 and 57
ostrichofevil	第155章	ostrichofevil	Chapter 155
OverCoder	第35和177章	OverCoder	Chapters 35 and 177
P.J.Meisch	第11、13、35、69、100、115和130章	P.J.Meisch	Chapters 11, 13, 35, 69, 100, 115 and 130
巴勃罗	第24章	Pablo	Chapter 24
步伐	第42章	Pace	Chapter 42
padippist	第177章	padippist	Chapter 177
paisanco	第47章	paisanco	Chapter 47
熊猫	第23章	Panda	Chapter 23
ParkerHalo	第9章、第10章、第40章和第46章	ParkerHalo	Chapters 9, 10, 40 and 46
保罗·贝洛拉	第47章	Paul Bellora	Chapter 47
Pavneet_Singh	第1章	Pavneet Singh	Chapter 1
帕万	第57章和第111章	Pawan	Chapters 57 and 111
Pawel_Albecki	第23、24、35和47章	Pawel Albecki	Chapters 23, 24, 35 and 47
PcAF	第47章	PcAF	Chapter 47
彼得·拉德 (Peter Rader)	第132章	Peter Rader	Chapter 132
peterh	第72章	peterh	Chapter 72
佩特·弗里贝格 (Petter Friberg)	第3、11、24、35、42、47、56、57、69、73、82、103、117和162章	Petter Friberg	Chapters 3, 11, 24, 35, 42, 47, 56, 57, 69, 73, 82, 103, 117 and 162
phant0m	第11章	phant0m	Chapter 11
phatfingers	第28章	phatfingers	Chapter 28
philnate	第47、52、73、74和127章	philnate	Chapters 47, 52, 73, 74 and 127
海盗 杰克	第57章	Pirate Jack	Chapter 57
皮尤什·巴德里亚	第11、89和132章	Piyush Baderia	Chapters 11, 89 and 132
披萨青蛙	第6章	PizzaFrog	Chapter 6
波洛斯托	第24章	Polostor	Chapter 24
波普斯	第1章	Pops	Chapter 1
权力领主	第24和63章	Powerlord	Chapters 24 and 63
ppeterka	第11、16、23、33、57、69、70、80、91、105、107、122和135章	ppeterka	Chapters 11, 16, 23, 33, 57, 69, 70, 80, 91, 105, 107, 122 and 135
普拉萨德·雷迪	第24章	Prasad Reddy	Chapter 24
普雷姆·辛格·比斯特	第107章	Prem Singh Bist	Chapter 107

Přemysl Štastný	第11章	Přemysl Štastný	Chapter 11
化名帕特尔	第148章	Pseudonym Patel	Chapter 148
PSN	第23章	PSN	Chapter 23
PSo	第11和86章	PSo	Chapters 11 and 86
普詹·斯里瓦斯塔瓦	第73章	Pujan Srivastava	Chapter 73
QoP	第11章和第23章	QoP	Chapters 11 and 23
qxz	第40章	qxz	Chapter 40
拉德克·波斯托沃维奇	第10章和第69章	Radek Postołowicz	Chapters 10 and 69
Radiodef	第23章和第24章	Radiodef	Chapters 23 and 24
拉杜安·鲁菲德	第1章、第11章、第23章、第35章、第47章、第57章、第69章、第73章和第182章	Radouane ROUFID	Chapters 1, 11, 23, 35, 47, 57, 69, 73 and 182
拉斐尔·帕切科	第114章	Rafael Pacheco	Chapter 114
拉胡尔·泰亚吉	第40章	rahul tyagi	Chapter 40
拉贾迪利普·科利	第24章	rajadilipkolli	Chapter 24
拉杰什	第23章	Rajesh	Chapter 23
拉基蒂ć	第57章和第182章	Rakitić	Chapters 57 and 182
rakwaht	第170章	rakwaht	Chapter 170
拉尔夫·克莱伯霍夫	第16章和第41章	Ralf Kleberhoff	Chapters 16 and 41
拉姆	第1、16、23、28、42、43、48、70、74、78、80和91章	Ram	Chapters 1, 16, 23, 28, 42, 43, 48, 70, 74, 78, 80 and 91
拉面厨师	第1、11、23、33、40、69、73、79、86、106、126、151和165章	RamenChef	Chapters 1, 11, 23, 33, 40, 69, 73, 79, 86, 106, 126, 151 and 165
RAnders00	第10、11、60和77章	RAnders00	Chapters 10, 11, 60 and 77
拉文德拉·巴布 (Ravindra babu)	第54、103、111、126和127章	Ravindra babu	Chapters 54, 103, 111, 126 and 127
拉文德拉·HV (Ravindra HV)	第52和132章	Ravindra HV	Chapters 52 and 132
拉维特贾 (Raviteja)	第85章	Raviteja	Chapter 85
ravthiru	第28、57和82章	ravthiru	Chapters 28, 57 and 82
rd22	第24、33、35、47和126章	rd22	Chapters 24, 33, 35, 47 and 126
rdonuk	第24和69章	rdonuk	Chapters 24 and 69
Rednivrug	第22章	Rednivrug	Chapter 22
Redterd	第78章	Redterd	Chapter 78
伦斯·范德·海登	第116章	Rens van der Heijden	Chapter 116
reto	第57章	reto	Chapter 57
雷乌特·沙拉巴尼	第1、23、40和57章	Reut Sharabani	Chapters 1, 23, 40 and 57
richersoon	第52章	richersoon	Chapter 52
RobAu	第57章、第69章和第77章	RobAu	Chapters 57, 69 and 77
罗伯特·哥伦比亚	第10章、第23章和第42章	Robert Columbia	Chapters 10, 23 and 42
罗宾	第75章	Robin	Chapter 75
罗谢尔利	第11章	Rocherlee	Chapter 11
罗热里奥	第47章	Rogério	Chapter 47
rokonoid	第66章、第77章和第87章	rokonoid	Chapters 66, 77 and 87
rolve	第23章、第47章、第56章和第73章	rolve	Chapters 23, 47, 56 and 73
ronnyfm	第182章	ronnyfm	Chapter 182
罗农·德克斯	第35章	Ronon Dex	Chapter 35
RudolphEst	第132章	RudolphEst	Chapter 132
鲁斯兰·贝斯	第20章	Ruslan Bes	Chapter 20
保罗·V·拉特利奇	第47章	RutledgePaulV	Chapter 47
瑞安·科库佐	第48章	Ryan Cocuzzo	Chapter 48
瑞安·希尔伯特	第22章	Ryan Hilbert	Chapter 22
saagarjha	第56章和第89章	saagarjha	Chapters 56 and 89
萨钦·萨拉维	第1章和第131章	Sachin Sarawgi	Chapters 1 and 131
萨克利尔·巴洛尼姆	第73章	Saclyr Barlonium	Chapter 73
萨迪克·阿里	第71章	Sadiq Ali	Chapter 71
赛义夫	第80章	Saif	Chapter 80
萨姆克	第33章和第35章	Samk	Chapters 33 and 35
萨南德雷亚	第182章	Sanandrea	Chapter 182

桑迪普·查特吉	第182章	Sandeep Chatterjee	Chapter 182
sanjaykumar81	第117章	sanjaykumar81	Chapter 117
桑托什·拉马南	第74章和第117章	Santhosh Ramanan	Chapters 74 and 117
sargue	第6章和第50章	sargue	Chapters 6 and 50
Saša Šijak	第69章	Saša Šijak	Chapter 69
索拉布	第23章	Saurabh	Chapter 23
SaWo	第150章	SaWo	Chapter 150
scorpp	第131章	scorpp	Chapter 131
screab	第130章和第183章	Sergii Bishyr	Chapters 130 and 183
谢尔吉·比希尔	第23章、第57章和第73章	sevenforce	Chapters 23, 57 and 73
sevenforce	第11章、第23章、第57章和第74章	Shaan	Chapters 11, 23, 57 and 74
沙安	第86章	Shettyh	Chapter 86
谢蒂赫	第127章	shibli049	Chapter 127
shibli049	第142章	ShivBuyya	Chapter 142
ShivBuyya	第11章、第40章和第77章	shmosel	Chapters 11, 40 and 77
shmosel	第23章、第35章、第43章、第57章、第67章、第88章和第106章	Shoe	Chapters 23, 35, 43, 57, 67, 88 and 106
Shoe	第11章、第23章和第57章	Siguza	Chapters 11, 23 and 57
Siguza	第1和47章	Simon	Chapters 1 and 47
Simon	第17章	Simulant	Chapter 17
Simulant	第10章和第79章	Siva Sainath Reddy Bandi	Chapters 10 and 79
班迪·西瓦·赛纳斯·雷迪	第66章	SjB	Chapter 66
SjB	第24章	skia.heliou	Chapter 24
skia.heliou	第16章	Sky	Chapter 16
天空	第11章	Skylar Sutton	Chapter 11
斯凯拉·萨顿	第73章	smichel	Chapter 73
smichel	第101章	Smit	Chapter 101
斯密特	第148章	solidcell	Chapters 11 and 23
solidcell	第11章和第23章	someoneigna	Chapter 79
someoneigna	第79章	Somnath Musib	Chapters 26 and 85
索姆纳斯·穆西布	第26章和第85章	Spina	Chapters 35 and 57
斯皮纳	第35章和第57章	SRJ	Chapter 57
SRJ	第57章	stackptr	Chapters 1, 23 and 57
stackptr	第1章、第23章和第57章	Stefan Dollase	Chapter 57
斯特凡·多拉塞	第57章	stefanobaghino	Chapter 88
stefanobaghino	第88章	steffen	Chapter 135
斯特芬	第135章	Stephan	Chapter 124
斯特凡	第124章	Stephen C	Chapters 1, 5, 7, 8, 9, 10, 11, 13, 23, 25, 28, 33, 40, 42, 43, 45, 47, 50, 51, 54, 57, 59, 67, 69, 73, 77, 79, 81, 82, 85, 86, 88, 89, 95, 102, 103, 106, 126, 127, 130, 131, 132, 133, 134, 135, 137, 139, 146, 147, 148, 169, 177, 178, 182, 183, 184和185章
斯蒂芬·C	第1章、第23章、第47章、第69章和第73章	Stephen Leppik	Chapters 1, 23, 47, 69 and 73
斯蒂芬·莱皮克	第57章	Steve K	Chapter 57
史蒂夫·K	第54章、第69章、第77章、第86章和第107章	still_learning	Chapters 54, 69, 77, 86 and 107
still_learning	第107章	Stoyan Dekov	Chapter 107
斯托扬·德科夫	第54章和第126章	Sudhir Singh	Chapters 54 and 126
苏迪尔·辛格	第38章和第57章	Sugan	Chapters 38 and 57
苏甘	第3章	Sujith Niraikulathan	Chapter 3
苏吉特·尼赖库拉坦	第34章	Suketu Patel	Chapter 34
苏凯图·帕特尔	第127、139、146和147章	Suminda Sirinath S.	Chapters 127, 139, 146 and 147
苏明达·西里纳斯 S.		Dharmasena	
达尔马塞纳		sumit	Chapter 85
苏米特	第85章	svsav	Chapter 93
svsav	第93章		

[Shadowfa](#)
[taer](#)
[tainy](#)
[塔伦·马甘蒂](#)
[TDG](#)
[thatguy](#)
[戴帽子的家伙](#)
[迷失的心灵](#)
[幽灵玩家](#)
[提萨鲁·古鲁格](#)
[托马斯](#)
[托马斯·弗里奇](#)
[托马斯·格罗特](#)
[ThunderStruct](#)
[蒂姆](#)
[TMN](#)
[TNT](#)
[托比亚斯·弗里丁格](#)
[托马什·巴沃尔](#)
[tonirush](#)
[托尼](#)
[托尼·本布拉欣](#)
[托尔斯滕](#)
[托特·扎姆](#)
[tpunt](#)
[trashgod](#)
[特拉维斯·J](#)
[特里普塔·基鲁拉](#)
[Tunaki](#)
[TuringTux](#)
[泰勒·齐卡](#)
[tynn](#)
[Un3qual](#)
[Unihedron](#)
[通用电力](#)
[乌里·阿加西](#)
[user1121883](#)
[user1133275](#)
[user140547](#)
[user1803551](#)
[user187470](#)
[user2296600](#)
[user2314737](#)
[user2683146](#)
[user3105453](#)
[user6653173](#)
[Uux](#)
[azaif](#)
[vallismortis](#)
[瓦西里斯·瓦西拉托斯](#)
[瓦西里·弗拉索夫](#)
[瓦特萨尔苏拉](#)
[维德拉克](#)

[Shadowfa](#)
[taer](#)
[tainy](#)
[Tarun Maganti](#)
[TDG](#)
[thatguy](#)
[The Guy with The Hat](#)
[TheLostMind](#)
[ThePhantomGamer](#)
[Thisaru Guruge](#)
[Thomas](#)
[Thomas Fritsch](#)
[Thomas Gerot](#)
[ThunderStruct](#)
[Tim](#)
[TMN](#)
[TNT](#)
[Tobias Friedinger](#)
[Tomasz Bawor](#)
[tonirush](#)
[Tony](#)
[Tony BenBrahim](#)
[Torsten](#)
[Tot Zam](#)
[tpunt](#)
[trashgod](#)
[Travis J](#)
[Tripta Kiroula](#)
[Tunaki](#)
[TuringTux](#)
[Tyler Zika](#)
[tynn](#)
[Un3qual](#)
[Unihedron](#)
[Universal Electricity](#)
[Uri Agassi](#)
[user1121883](#)
[user1133275](#)
[user140547](#)
[user1803551](#)
[user187470](#)
[user2296600](#)
[user2314737](#)
[user2683146](#)
[user3105453](#)
[user6653173](#)
[Uux](#)
[azaif](#)
[vallismortis](#)
[Vasilis Vasilatos](#)
[Vasiliy Vlasov](#)
[VatsalSura](#)
[Veedrac](#)

[文](#)
[VGR](#)
[维亚切斯拉夫·韦德宁](#)
[维克多·G.](#)
[victorantunes](#)
[文](#)
[文斯·埃米格](#)
[vincentvanjoe](#)
[维诺德·库马尔·卡希亚普](#)
[维韦克·阿努普](#)
[弗拉德](#)
[弗拉基米尔·瓦盖采夫](#)
[Vogel612](#)
[vorburger](#)
[vsminkov](#)
[vsnyc](#)
[武科](#)
[vvtx](#)
[webo80](#)
[WillShackleford](#)
[Wilson](#)
[Wolfgang](#)
[xploreraj](#)
[xTrollxDudex](#)
[xwoker](#)
[yitzih](#)
[yiwei](#)
[Yogesh](#)
[Yohanes Khosiawan 许先汉](#)
[yuku](#)
[尤里·费多罗夫](#)
[扎卡里·大卫·桑德斯](#)
[泽·鲁比乌斯](#)
[锆石](#)
[Héb é](#)
[Łukasz Piaszczyk](#)
[ФХоце བ ພເປີເປົາ ຕ](#)
[□ R N](#)

[第23章](#)
[第11章、第72章、第114章和第185章](#)
[第28章和第62章](#)
[第23章和第35章](#)
[第104章](#)
[第1章](#)
[第79章](#)
[第73章](#)
[第16章](#)
[第18章](#)
[第47章和第126章](#)
[第89章](#)
[第8、20、23、28、41、43、58、80、104、127和162章](#)
[第123章](#)
[第107章](#)
[第57章](#)
[第54章](#)
[第11章](#)
[第24章和第73章](#)
[第102章和第164章](#)
[第1章、第11章、第23章、第47章、第57章和第69章](#)
[第73章](#)
[第24章、第73章和第88章](#)
[第126章和第139章](#)
[第19章和第74章](#)
[第123章](#)
[第69章](#)
[第73章](#)
[第4章](#)
[第11章和第23章](#)
[第23章和第107章](#)
[第1章](#)
[第57章和第182章](#)
[第79章和第134章](#)
[第78章、第120章和第152章](#)
[第19章](#)
[第11章、第17章、第21章、第23章、第25章、第43章、第78章和第103章](#)
[第1章和第25章](#)

[Ven](#)
[VGR](#)
[Viacheslav Vedenin](#)
[Victor G.](#)
[victorantunes](#)
[Vin](#)
[Vince Emigh](#)
[vincentvanjoe](#)
[Vinod Kumar Kashyap](#)
[Vivek Anoop](#)
[Vlad](#)
[Vladimir Vagaytsev](#)
[Vogel612](#)
[vorburger](#)
[vsminkov](#)
[vsnyc](#)
[Vucko](#)
[vvtx](#)
[webo80](#)
[WillShackleford](#)
[Wilson](#)
[Wolfgang](#)
[xploreraj](#)
[xTrollxDudex](#)
[xwoker](#)
[yitzih](#)
[yiwei](#)
[Yogesh](#)
[Yohanes Khosiawan 许先汉](#)
[yuku](#)
[Yury Fedorov](#)
[Zachary David Saunders](#)
[Ze Rubeus](#)
[Zircon](#)
[Héb é](#)
[Łukasz Piaszczyk](#)
[ФХоце བ ພເປີເປົາ ຕ](#)
[□ R □ □ □ N](#)

你可能也喜欢



You may also like

