

sih3.Java基本程序设计结构

static void

main()方法正常退出退出代码为0， 如果希望终止程序时返回其他代码-System.exit()

自动生成文档的注释/** */

数据类型

强类型语言， 每个变量必须声明一种类型， 8中基本类型（4整型2浮点1Unicode字符编码单元1真值boolean）

Java整型范围与运行机器无关， 解决移植性问题（C,C++需要针对不同处理器32位/16位）

类型	存储需求	备注
int	4	0b可以写二进制数.
short	2	可以为数字字面量加下划线， 1_000_000， 易读性， 无用， 编译器会去除下划线。
long	8	Java的一切都是有符号的， 没有无符号形式的类型。
byte	1	
float	4	需要存储大量数据/需要单精度数据的库时使用， 有一个后缀F/f。
double	8	字面量默认double类型， 可添加D或d;浮点数值不适用于无法接受舍入误差的金融计算（二进制系统无法精确地表示分数）， 此时使用BigDecimal类。可以用十六进制表示浮点数值,0x1.0p-3， 在十六进制中使用p表示指数， 指数基数是2 特殊浮点， 常量Double.NaN, Double.POSITIVE_INFINITY
char	2	Unicode字符可以用一个/两个char值描述， 转义序列\u(如\r是\u000d) Unicode会在解析代码之前得到处理， 小心注释中的u（后后面必须跟四个十六进制数字不然会发生语法错误）
boolean	1	java中只有false和true能表示值， 整型值和布尔值之间不能相互转换

Unicode编码在设计时认为两个字节的代码宽度足以对世界上各种语言的所有字符进行编码。

然后超了， 解决方法：码点（code point)是与字符对应的代码值， 如U+0041表示（\u0041-A),Unicode分成17个代码级别。char类型描述了UTF-16编码中的一个代码单元， **建议不要用这个类型。**

枚举类型

enum enumName{A,B,C};

enumName s = enumName.A/null， 限定变量取值在有限范围之中。

变量

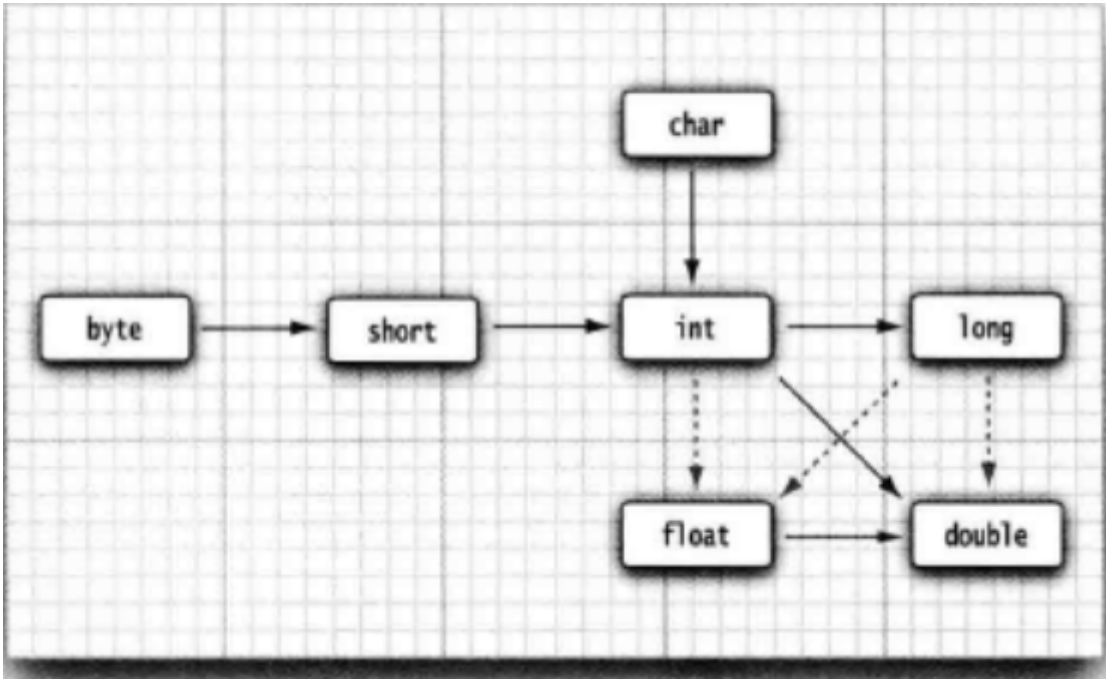
_,\$(合法但不要， 只用在编译器或其他工具生成的名字中)

声明后必须显式赋值初始化， 不区分变量的声明与定义。

static 类变量，被同一个类的对象使用

数值类型之间的转换。虚线表示可能有精度损失

boolean转数值可以使用b?1:0



运算符

当整数/整数时表示整数除法，否则表示浮点除法。

strictfp关键字标记方法使用严格的浮点计算来生成可再生的结果。

数值类型进行二元操作是会将两个操作数转换成同一种类型然后在进行计算

- 有double,转double->有float转float->有long装long->否则两个操作数都转成int

位运算：>>>用0填充高位，>>用符号位填充高位，右操作数要完成模32的运算（若左操作数是long，模64）1<<35 = 1<<3

结合性从右向左的运算符（一元运算，?:，+=）

优先级

运 算 符	结 合 性
[].()(方法调用)	从左向右
! ~ ++ -- +(一元运算)-(一元运算)()(强制类型转换)new	从右向左
* / %	从左向右
+ -	从左向右
<< >> >>>	从左向右
< <= > >= instanceof	从左向右
= = !=	从左向右
&	从左向右
^	从左向右
	从左向右
&&	从左向右
	从左向右
?:	从右向左
= += -= *= /= %= &= = ^= <<= >>= >>>=	从右向左

方法

静态方法-处理的不是对象？

静态导入-import static java.lang.Math.*; , 然后Math.pow就可以直接写成pow()。

字符串

字符串是不可变的，但是变量可以被修改，应用另一个字符串。

不可变的优点：编译器可以让字符串共享。

常用方法	简介
子串 substring(0,3)	第二个参数是不想赋值的第一个位置。0,1,2位置的字符。 长度 为b-a=3.
拼接	+ 定界符分割后连接用静态join String all =String.join(" ", "A","B","C"); ->"A B C"
相等	==不要用来检测字符串是否相等。只有字符串常量是共享的，+ 或substring等操作产生的结果不一定共享。 使用equals () 检查
检查	if(str!=null&&str.length()!=0)
码点	代码单元数量 (char) length(), 码点数量) 实际长度, codePointCound(0,greeting.length());

```
char charAt(int index); int indexOf(String str);int lastIndexOf;  
int compareTo(String other)返回self-other  
boolean equals(Object other);equalsIgnoreCase;startsWith;endsWith;  
String replace(charSequence(可以用String或StringBuilder) oldString, newString);String substring(int  
beginIndex, int endIndex);  
String toLowerCase();trim()删除头尾部空格
```

CharSequence是一种接口类型，可以传入String，StringBuilder类型的对象。

StringBuilder类

是可以修改的，添加内容时使用builder.append(ch),需要构建字符串时toString()方法。

```
void setCharAt(int i, char c)  
StringBuilder insert(int offset, String str);  
StringBuilder delete(int startIndex, int endIndex)  
String toString()
```

输入输出

输入

Scanner对象，标准输入流System.in关联，读取单词（空白符分界）next()，读取类型nextInt()，读取一行（行中可以由空格）nextLine；boolean hasNext(),hasNextInt()

输入可见，不适用于在控制台读取密码，可以用Console类读取密码。Console
cons=System.console(); char[] passwd =cons.readPassword("Password");

输出

表 3-5 用于 printf 的转换符

转换符	类 型	举 例	转换符	类 型	举 例
d	十进制整数	159	s	字符串	Hello
x	十六进制整数	9f	c	字符	H
o	八进制整数	237	b	布尔	True
f	定点浮点数	15.9	h	散列码	42628b2
e	指数浮点数	1.59e+01	tx 或 Tx	日 期 时 间 (T 强制大写)	已经过时，应当改为使用 java.time 类，参见卷 II 第 6 章
g	通用浮点数	—	%	百分号	%
a	十六进制浮点数	0x1.fccdp3	n	与平台有关 的行分隔符	—

表 3-6 用于 printf 的标志

标 志	目 的	举 例
+	打印正数和负数的符号	+3333.33
空格	在正数之前添加空格	3333.33
0	数字前面补 0	003333.33
-	左对齐	3333.33
(将负数括在括号内	(3333.33)
,	添加分组分隔符	3,333.33
# (对于 f 格式)	包含小数点	3,333.
# (对于 x 或 0 格式)	添加前缀 0x 或 0	0xcafe
\$	给定被格式化的参数索引。例如，%1\$d, %1\$x 将以十进制和十六进制格式打印第 1 个参数	159 9F
<	格式化前面说明的数值。例如，%d%<x 以十进制和十六进制打印同一个数值	159 9F

可以使用静态String.format方法创建格式化字符串而不打印（即把相应输出内容保存在返回的字符串中）

文件输入与输出

File对象构造Scanner对象，文件名中如果包含反斜杠符号需要再加一个额外的反斜杠。读取文本文件时需要知道字符编码，省略时会使用运行Java程序及其的默认编码。

```
Scanner in = new Scanner(Paths.get("myfile.txt"), "UTF-8");

PrintWriter out = new PrintWriter("myfile.txt","UTF-8") 使用print,println,printf等命令输出。
```

应该告知编译器：已经知道有可能出现“输入/输出”异常，利用throws子句标记。

```
public static throws IOException
```

控制语句

嵌套块中不能声明同名变量，无法通过编译，C++中允许内层定义覆盖外层定义变量，Java中不允许。

大数值

java.math.BigInteger、BigDecimal
add(),multiple()方法。
Java只重载了+运算符而没有重载其他的运算符。

数组

length是属性不是方法。
for-each循环处理数组中元素

for(int element:collection){};collection必须是数组或者实现了Iterable接口的类对象。读作 (for each element in collection)

打印数组可以使用println(Arrays.toString(a)) "[1,2,3]"

匿名数组重新初始化数组

smallArray=new int[]{1,2,3};,Java中允许数组长度为0, 返回结果是数组而结果为空时可以创建一个长度为0的数组。

数组拷贝

- =: 两个变量引用同一个数组; `int [] a =b;`
- copyOf () 把一个数组的所有值拷贝到新的数组中, `int [] copied=Arrays.copyOf(original,original.length*2/1/3均可)`

java的数组声明类似于C++的数组指针而不是数组本身, `int [] a =new int[100];` 更像是 `int *a = new int [100]`而不是 `int a[100]`

`String args[]`接收从"java 程序名 参数"中参数起的字符串。

数组排序-`Arrays.sort(a)`

`Math.random()`返回0-1的一个随机数;

Arrays方法

- `static void fill(type[] a , type v);`
- `static int binarySearch(type[] a, type v)`
- `static boolean equals(type[] a , type[] b);`

多维数组

for each循环处理二维数组时按照行 (一维数组) 处理, 遍历需要嵌套:

```
1  for (double [] row:a)
2      for(double value: row)
3          do;
4
5  快速打印二维数组数据元素可以调用Arrays.deepToString(a),类似matlab的a
```

Java本质上只有一维数组, 多维数组时数组的数组, `int a[3][4]`, a数组是一个包含3个元素的数组, 每个元素是一个由4个元素组成的数组。所以可以单独地存取数组的某一行, 设定不同长度: `odds[n]=new int[n+1];`

4.对象和类







静态, 包, 类路径, 自定义类, 预定一类, 方法参数, 文档注释。

对象

行为 (可调用的方法), 状态 (当前特征的信息), 标识 (身份);

类之间的关系-依赖 (uses-a), 聚合 (has-a), 继承 (is-a) ,

表 4-1 表达类关系的 UML 符号

关 系	UML 连接符
继承	
接口实现	
依赖	
聚合	
关联	
直接关联	

一个对象变量仅仅引用一个对象而没有实际包含一个对象，值是对存储在另一个地方的一个对象的引用。

局部变量不会自动初始化为null,必须通过new或者设置为null进行初始化。

如何理解：

```
1 | Date birthday;//Java
2 | Date * birthday;//C++
3 | 所以必须通过new进行初始化。
```

[?]静态工厂方法 (factory method) 代表你调用构造器

```
LocalDate newday = LocalDate.now();
```

更改器方法-mutator method——调用对象本身的状态会改变

访问器方法-accessor method 只访问对象而不修改对象。

C++中带有const后缀的是访问器方法，默认是更改器方法

java.time.LocalDate 8

- static Localtime now()
- static LocalTime of(int year, int month, int day)
- int getYear(),getMonthValue(),getDayOfMonth()
- DayOfWeek getDayOfWeek(1表示星期一， 7表示星期日)
- LocalDate plusDays(int n)

用户自定义类

workhorse class主力类，有自己的实例域和实例方法但没有main方法。

类定义形式： field->constructor->method.

在一个源文件中，只能由一个公有类，但可以有任意数目的非公有类。

javac Main.java会在发现里面使用了别的类时查找这个类对应的文件并编译，可以认为**内置了make功能**

Java对象是在堆中构造的，构造器总是伴随着new操作符一起使用

不要在构造器中定义与实例域重名的局部变量（实例域就是对象的属性）

隐式参数this，是出现在方法名前的对象。（方法调用目标/接收者）

C++中的内联。是否设置为内联方法是Java虚拟机的任务，编译器会监视那些简洁、经常被调用、没有被重载以及可优化的方法。

如果一个域是public的，破坏这个值的捣乱者有可能出现在任何地方。

- 私有数据域
- 公有域访问器和域更改器（可移植性错误检查）

不要编写返回引用可变对象的访问器方法，因为可以通过返回的这个对象去更改这个域。如果需要返回可以使用克隆的方法，返回存放在另一个位置上的对象副本。即**如果需要返回一个可变数据域的拷贝时使用clone。**

```
1 class Employee
2 {
3     private Date hireDay;
4     public Date getHireDay()
5     {
6         return hireDay;//bad.
7         return (Date)hireDay.clone();//great.
8     }
9 }
```

访问权限

一个方法可以访问所属类的所有对象的私有数据，包括不是调用方法本身的对象。

private方法-类的设计者可以确信他不会被外部的其他类操作调用，可以快乐地删除。

final实例域-构件对象时必须初始化并且之后不能再更改。

对于可变类，有点迷惑，`private final StringBuilder evaluations` 只表示存储在evaluations中的对象引用不会再指示其他StringBuilder对象，不过这个对象可以更改。

immutable类-不可变类，如果类中每个方法都不会改变其对象，这种类就是不可变的类。

static静态域，又名**类域**-每个类中只有一个这样的域，属于类而不属于任何独立的对象，所有对象共享一个static域。

static final 静态常量-如设个PI，可以public反正都不能改。

可以修改final变量的值-你可以通过不使用Java语言实现的本地方法绕过Java语言的存取控制机制。

static静态方法-不能向对象实施操作，可以认为是无this参数的方法，但是可以访问自身类中的静态域

```
1 Math.pow(x,a)//计算时不使用任何Math对象
2 //可以使用Math对象调用静态方法，但是这个结果和这个对象毫无联系，所以不建议这样做来混淆，建议直接使用类名
```

factory method-静态方法构造对象。

```
1 NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
2 NumberFormat percentFormatter = NumberFormat.getPercentInstance();
3 double x=0.1;
4 System.out.println(currencyFormatter.format(x));//0.1
5 System.out.println(percentFormatter.format(x));//10%
```

使用场合

- 无法命名构造器
- 使用构造器时无法改变所构造的对象类型

main方法

每一个类可以有一个main方法，可以进行单元测试；

java employee会执行这个main;

java Application不会执行这个main(如果employee属于application)

方法参数

call by value方法接收调用者的值，Java只使用值传递，但是可以传递**基本数据类型**（本身不会改变）和**对象引用**（可以改变引用的对象，对象引用和其他的拷贝同时引用同一个对象），参数被初始化为拷贝进行整个方法。

一个方法不能让对象参数引用一个新的对象。

C++有 call by value/reference(&)

对象构造

重载-不同参数，相同名字（signature=方法名+参数类型），编译器匹配调用方法的值类型选择对应的方法（overloading resolution），不匹配时产生编译时错误。

Remark：必须明确地初始化方法中的局部变量，但是类中的域可以自动初始化为默认值（0，false，null）

仅当类没有提供任何构造器时，系统才会提供一个默认的构造器

C++有特殊的初始化器列表语法，但是Java中不需要，因为对象没有子对象，只有指向其他对象的指针。

调用另一个构造器

```
1 public employee(double s)
2 {
3     //call employee(String,double)
4     this("Employee "+NextId,s);
5     nextId++;
6 }
```

初始化数据域的方法：

- 构造器中设置
- 声明中赋值
- 初始化块，就是类中一个单纯的{}，没什么用。会先运行初始化块然后在运行构造器的主体部分。

处理构造器调用步骤：

- 初始化所有数据域为默认值
- 按照类声明次序依次执行初始化语句和初始化块
- 如**构造器第一行调用了第二个构造器**，执行第二个构造器主体
- 执行这个构造器的主体

```
1 static
2 {
3     Random generator = new Random();
4     nextId = generator.nextInt(10000); //（返回0-n-1之间的随机数）
5 } //赋予一个小于10000的随机整数，第一次加载类时执行静态域初始化
```

对象析构

不支持析构器，可以添加finalize方法在垃圾回收器清楚对象之前调用，对象用完时可以用close来完成相应清理工作。

包

组织类，保证类名的唯一性，类似命名空间。

一个类可以使用所属包中的所有类，以及其他包中的公有类。可以

- `java.time.LocalDate.now()`
- `import java.util.*; LocalDate today = LocalDate.now();`

使用import java.util.*对代码大小没有影响，明确所导入的类会使代码读者更准确地知道加载了哪些类。

两个包中有类冲突的话可以指明用方法1究竟用的是哪个类。

C++的#include需要将外部特性的声明加载进来，它无法查看任何文件的内部，除了在头文件中明确包含的文件。Java编译器可以查看其他文件的内部，只要告诉它到哪里去看。

静态导入

`import static java.lang.System.*;`可以在不加类名前缀的情况下使用System类的静态方法和静态域：`out.println("Bye");//System.out`

将类放入包中把包的名字放在源文件开头

`package com.horstmann.corejava;`如果没有放置package语句，源文件中的类被放置在默认包（default package）

你自己要把文件放到匹配的子目录 `com/horstmann/corejava` 中，这样编译器就会把类文件放在相同的目录结构里。

包作用域

- public
- private
- 缺省：同一个包中所有方法访问

类路径JAR文件使用ZIP格式组织文件和子目录

文档注释

JDK的javadoc可以由源文件生成HTML文档。

`/**`开始的注释可以用来生成文档。

javadoc 从特性中抽取信息，注释应放在所描述特性的前面。

包、公有类与接口、公有的和受保护的构造器及方法、公有的和受保护的域。

```
1  /**
2  @标记
3  自由格式文本（第一句式概要性句子，javadoc会抽成概要页）
4  可以使用HTML修饰符。
```

- 方法注释
 - @param 可以占据多行，所有@param标记必须放在一起
 - @return
 - @throws
- 通用注释（类文档注释）
 - @author
 - @version
 - @since对引入特性的版本描述
 - @deprecated 不再使用
 - @see 引用，`@see com.horstmann.corejava.employee#raisesSalary(double)` #分割类名与方法名。

抽取注释- `javadoc -d docDirectory nameOfPackage...`

类设计技巧

- 保证数据私有，不要破坏封装性。数据的表示形式可能改变但使用方式却不会经常发生变化
- 数据初始化，不要依赖默认值而应该显式初始化所有数据
- 不要在类中食用过多的基本类型，可以用别的类（包含这些基本类型）来代替它们以方便理解并处理变化。
- 将职责过多的类进行分解。
- 优先使用不可变的类-如果多个线程试图同时更新一个对象会发生并发更改，结果不可预料；不可变的类可以安全地在多个线程间共享对象。

5.继承

基于已存在的类构造一个新类，继承这些类的方法和域。

反射 (?)

1.类、超类、子类

```
public class Manager extends Employee{}
```

是否应该设计为继承关系：“is-a”关系，B属于A；**置换法则**=程序中出现超类对象的地方可以用子类对象置换。

所有都是公有继承，C++中有私有继承和保护继承；

子类比超类拥有更加丰富的功能，超类集合包含所有子类集合。子类可以**增加域、方法或者覆盖超类的方法**。

覆盖方法-@override，**子类方法不能低于超类方法的可见性**，如果超类是Public，子类也一定是public。

【?】权限问题

Warning：子类的方法不能够直接的访问超类的私有域，需要借助超类中的公有借口

子类构造器必须在第一条语句中使用super调用超类构造器/缺省系统默认调用超类默认构造器（无参数）

```
1 public double getSalary()
2 {
3     return super.getSalary()+bonus;
4     //salary+bonus x
5     //super.getSalary读取超类的salary
6 }
```

super与this：

- super是一个指示编译器调用超类方法的特殊关键字，但是this是一个对象的引用（隐参数）
- super: 1.调用超类方法 2.调用超类构造器
- this:1.调用隐式参数 2.调用该类其他构造器

多态-一个对象变量可以指示多种实际类型的现象（如Employee类型可以引用Manager对象，虚拟机知道它实际引用的对象类型可以正确调用对应方法），**动态绑定**-运行时自动地选择调用那个方法。如果不希望有虚拟特征，可以将方法标记为**final**

继承层次 (inheritance hierarchy)是由一个公共超类派生出的所有类的集合。Java不支持多继承但是可以祖孙继承，只是横向上不能多继承。

超类变量可以引用子类对象/超类对象，调用方法时根据变量具体引用的对象类型使用对应的方法，子类变量不能引用超类，因为在调用子类专用方法时可能会出现错误。

```
1 //子类数组的引用可以直接转换为超类数组的引用而不需要采取强制类型转换。
2 Manager [] managers = new Manager[10];
3 Employee [] staff = managers;
4 //但是我们需要知道staff和manager引用的是同一个对象（经理对象），如果试图存储一个
  Employee类型的引用会引发异常：
5 staff[0](Manager)=new Employee("Harry");//x
6
7 //【?】为什么
8 Manager boss = new Manager();
9 Employee [] slaves= new Employee [3];
10 slaves[0]=boss;
11 boss.getSalary//调用Manager的方法
12 slaves[0].getSalary//调用employee的方法
```

方法调用x.f(param)，X是类C的一个对象

- 获取所有可能被调用的候选方法，如所有C类中名为f的方法，和其超类中名为f且为public的方法（超类私有方法不可访问）
- **重载解析**查看调用方法时提供的参数类型，完全匹配>类型转换（多种可能）
返回类型不属于签名，覆盖方法时需要保证返回类型兼容（可协变的返回类型）。
- 静态绑定-private/static/final/构造器，编译器知道应该调用哪个方法。否则采用动态绑定
- 动态绑定-虚拟机调用与x所引用对象的**实际类型**最适合的那个类的方法。X实际类型是D，C的子类，调用D的方法f，若不存在，在D的超类中寻找，以此类推。

final关键字-阻止继承

不允许扩展。子类无法覆盖。**final类中所有方法自动成为final方法。**

final类中的方法自动成为final，但不包括域。

目的：确保不会在子类中改变语义。

内联-如果方法没有被覆盖且很短，编译器会优化处理其为内联。所以有些程序员会为了避免动态绑定带来的系统开销而使用final关键词。

强制类型转换

唯一原因：在暂时忽视对象的实际类型之后使用对象的全部功能。

检查能否成功转换

```
1  //✓
2  if(staff[1] instanceof Manager)
3  {
4      boss = (Manager)staff[1];
5  }
6  //X[?]
7  Manager boss = (Manager)staff[1]
```

- 只能在继承层次内进行类型转换
- 在将超类转换为子类前，使用instanceof进行检查
- 尽量少用。
- 类似C++的dynamic_cast,但是转换失败时会抛出异常而不是生成null对象。

抽象类

祖先类进行高级抽象，把抽象层次高的方法放置在位于继承关系较高层次的通用超类中。

使用abstract关键词不需要实现这个方法。

包含抽象方法的类本身必须被声明为抽象的。抽象类可以有具体的数据和方法，类即使不包含抽象方法也可以声明为抽象的。

- 定义全部抽象方法-子类非抽象
- 未全部实现-子类也是抽象

抽象类不能被实例化，但是可以定义一个抽象类的对象变量，引用非抽象子类的对象。

由于不能生成抽象类的实例，所有抽象类变量永远不会被调用抽象类的方法，而是调用非抽象子类的方法。

受保护访问

protected-允许子类访问超类的方法/域。

【？】【我必要查一下三个访问权限的事情】

控制可见性的访问修饰符

- private-仅对本类可见
- public-对所有类可见
- protected-对本包和所有子类可见
- 默认缺省-对本包可见

2.Object类

所有类的始祖，每个类都是由Object类扩展而来的，需要熟悉这个类提供的所有服务。

C++中每个指针都可以转换成void * 指针

- equals方法，如果两个对象的状态相等（同一个引用），就认定这两个对象是相等的。但是对于多数类来说这种判断没什么意义，我们希望通过属性的比较来比较两个对象是否相等。

只有两个对象属于同一个类时才有可能相等。

Objects.equals(a,b)

- null&null->true
- null&非null->false
- 都不是null->a.equals(b)

```
1
2 public class Employee
3 {
4     ..
5     public boolean equals(Object otherObject)
6     {
7         //a quick test
8         if(this == otherObject) return true;
9         //null
10        if(otherObject == null) return false;
11        //class must match
12        if(getClass() != otherObject.getClass()) return false;
13        //non-null Employee
14        Employee other = (Employee) otherObject;
15        //test the identical values we care about
16        return id.equals(other.id);
17    }
18 }
19
20 public class Manager extends Employee
21 {
22     public boolean equals(Object otherObject)
23     {
24         //检查了类不相同
25         if (!super.equals(otherObject)) return false;
26         //然后检查子类特有的值（子类中的实例域）
27         Manager other = (Manager) otherObject;
28         return bonus == other.bonus;
29     }
30 }
```

Java语言规范要求equals方法具有下面的特性

- 自反性: x.equals(x)->true
- 对称性: y.equals(x)->true, x.equals(y)
- 传递性: x.equals(y), y.equals(z)->x.equals(z)也返回true
- 一致性: 反复调用应该返回同样的结果
- 对于任意非空引用x, x.equals(null)应该返回false

超类和子类比较时：如果子类能够拥有自己的相等概念，对称性需求将强制采用getClass进行检测。如果由超类决定相等的概念，使用instanceof进行检测，这样可以在不同子类的对象之间进行相等的比较。

注：集合是特殊的例子，**final** AbstractSet.equals

打了@Override标记，如果你没有覆盖的话会看到一个错误报告。

- hashCode-由对象导出的一个整型值，默认的散列码为对象的存储地址。但是类可以自己定义出散列码，如String类由内容导出。

如果重新定义equals方法就必须重新定义hashCode方法-equals和hashCode的定义必须一致。

```
1 return name.hashCode()+new Double(salary).hashCode();
2 =>
3 return Objects.hashCode(name)+Double.hashCode(salary);
4 Objects.hashCode()如果参数为null会返回0。（null安全）
```

- toString方法

绝大多数类-类的名字，方括号括起来的域值。

只要对象与一个字符串通过操作符“+”连接，Java就会自动调用toString以获得字符串描述。

x.toString()=>""+x

打印数组-Arrays.toString(arr). 多维-Arrays.deepToString.

调试信息（代替print）：`Logger.global.info("Current position = "+ position)`

为自定义的每一个类加一个toString方法。

3.泛型数组列表

运行时确定数组大小，但是一旦确定后不容易改变。->使用ArrayList类可以在添加/删除元素时自动调节数组容量。

ArrayList

一个采用类型参数（type parameter）的泛型类（generic class）

菱形语法-`ArrayList<Employee>=new ArrayList<>()`

Java老版本中，Vector用来实现动态数组，ArrayList类更有效。

.add()方法添加元素到数组列表中。如果内部数组已满，数组列表自动创建一个更大的数组并拷贝。

.ensureCapacity(N)分配一个预估的大小。在N之前不需要扩充。

.size()返回当前元素数量-等价于array.length

.trimToSize()调整存储区域大小为当前元素数量所需要的存储空间数目。

.set()只能替换数组中已经存在的元素内容。增加了访问元素语法的复杂程度-staff.set(i,harry)（等价于C的a[i]=harry;）

.get()获取数组列表的元素

```
1 //将数组元素拷贝到数组中
2 x[] a = new x[list.size()];
3 list.toArray(a);
```

.remove(n)移除元素，将之后所有元素都向前移动一个位置。

如果数组元素存储比较多，经常需要在中间插入/删除->考虑链表。

`for(Employee e:staff)` for-each循环遍历数组列表。

【？】Java不尽如人意的参数化类型的限制-有点没明白

4.对象包装器与自动装箱

Wrapper-与基本类型对应的类 (Integer、Long、Float、Double、Short、Byte、(前六个派生于公共超类Number)、Character、Boolean)

ArrayList<>中的参数类型不允许是基本类型，执行效率比较低。小型集合时可以用这个。

自动装箱

`list.add(3)` 会被自动转换为 `list.add(Integer.valueOf(3))`;

自动拆箱

`int n = list.get(i)` -> 翻译成 `list.get(i).intValue()`;

在两个包装器对象比较时，使用equals而不要用==，因为可能会产生混淆（比较指向的存储地址而不是值）。

可能会抛出NullPointerException异常（包装器类引用可能为null）

装箱和拆箱是编译器认可的，生成字节码时调用。

包装器对象可以放一些基本方法，如将**数字字符串转为整型**：`int x=Integer.parseInt(s)`，静态方法，与Integer对象无关。

Integer对象是不可变的，包含在包装器中的内容不会改变，不能使用这些包装器类创建修改数值参数的方法。

- static Integer valueOf(String s)
用s初始化一个Integer对象并返回
- static int parseInt(String s)返回s表示的整型数值。
- static String toString(int i)

5.参数数量可变的方法

省略号...报名这个方法可以接收任意数量的对象。

```
public PrintStream printf(String fmt, Object ... args){}
```

6.枚举类

```
1 public enum Size{SMALL,MEDIUM,LARGE}
2 //比较时只需要使用==而不用调用equals
3 public enum Size{
4     SMALL("S"),MEDIUM("M"),LARGE("L");
5
6     private String abbreviation;
7     private Size(String abbreviation){this.abbreviation = abbreviation;}
8     public getAbbreviation(){return abbreviation;}
9 }
10 //继承了Enum类。
```

Size.SMALL.toString()返回字符串，逆方法valueOf- `Size s = Enum.valueOf(Size.class, "SMALL")`;

Size.values()返回一个包含全部枚举值（类型是Size）的数组。

Size.MEDIUM.ordinal()返回enum中声明枚举常量的位置，从0开始计数。

`int compareTo(E other) = this-other`（枚举常量出现的位置次序之差）

7.反射

能够分析类能力的程序成为反射。

8.继承的设计技巧

1. 将公共操作和域放在超类
2. 不要使用protected机制
3. 使用继承实现“is-a”关系
4. 除非所有继承的方法都有意义，否则不要使用继承。
5. 覆盖方法时不要改变预期的行为-置换原则
6. 使用多态而非类型信息。
7. 不要过多的使用反射。

六、接口和Lambda表达式

高级技术：

- 接口：描述类有什么功能而不给出每个功能的具体实现，一个类可以实现多个接口，并在需要接口的地方随时使用实现了相应接口的对象。
- lambda表达式：简介地表示使用回调或变量行为的代码。
- 内部类-定义在另一个类的内部，其中的方法可以访问包含他们的外部类的域。主要用于设计具有相互协作关系的累计和。
- 代理：实现任意接口的对象，一种非常专业的构造工具。

1.接口

接口是对类的一组需求描述，类要遵从接口描述的统一格式进行定义。

如果类遵从某个特定接口，那么就履行这项服务。Arrays类中的sort方法承诺可以对对象数组进行排序，但要求满足下列前提：对象所属的类必须实现了Comparable接口。

接口中的所有方法**自动地属于public**，但是在**实现接口的时候必须把方法声明为public**。

接口不能含有实例域。可以定义常量。

让类实现接口：-将类声明为实现给定的接口（implements），对接口中所有方法进行定义

对Object参数进行类型转换总是让人感觉不太顺眼，现在已经可以写成泛型<>了。

Q：为什么不直接在Employee类中提供compareTo方法而必须实现Comparable接口？

A：因为Java是一种强类型的语言，调用方法时会检查方法是否存在，sort方法中有compareTo（）方法故编译器必须确认他一定有compareTo方法。实现接口可以保证这一点。

对于任意的x和y，实现必须能够保证 $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$ ，即如果调换参数，结果的符号也应该调换。

如果Manager覆盖了compareTo就必须要有经理和雇员比较的准备而不是把雇员转成经理（置换原则）

此时解决方法为-1.判断类，抛出类不同的错误。2.在超类中生成final方法，提供对两个不同子类对象比较的方法。

接口特性

1. 不可构造对象，可以
 - 声明接口变量，必须引用实现了接口的类对象。 `Comparable x=new Employee(..);`
 - 检查一个对象是否实现接口 `if(anObject instanceof Comparable)`
2. 接口也可以被扩展，允许存在多条从具有较高通用性的接口到较高专用型的接口的链。
3. 可以包含常量，域被自动设为 `public static final` 类型，不包含实例域/静态方法。
4. 每个类只有一个超类，但可以由很多借口。

接口取代抽象类：不支持多继承，提供多重继承的大多数好处，避免多重继承的复杂性和低效性。

默认方法：

default修饰符可以为接口方法提供了一个默认实现，`default int compareTo(T other){}`，在类实现接口时程序员只需要为真正关心的事件覆盖相应的监听器。**默认方法可以调用任何其他方法。**

接口演化（interface evolution），如果你在一个老的接口上提供一个新的方法，你不把他设置为默认方法的话，会导致原来的依靠这个接口的类编译失败；只有添加默认方法才能保证源代码兼容。

默认方法冲突解决规则：

- 超类优先，如果超类提供具体方法，同名且具有相同参数类型的默认方法会被忽略。
- 接口冲突，如果两个接口提供了一样参数类型的默认方法，必须覆盖这个方法来解决冲突。**让程序猿自己来解决二义性。**
- 超类和接口的方法冲突，只考虑超类方法，接口的默认方法会被忽略。**类优先的原则**

Java设计者更强调一致性！类优先的原则可以保证兼容性，接口新加的默认方法不会干扰之前能正常工作的代码。

2.接口与回调

回调

指出某个特定事件发生时采取的行动

- ActionListener接口

```
void actionPerformed(ActionEvent event)
```

```
1 class TimePrinter implements ActionListener
2 {
3     public void actionPerformed(ActionEvent event)
4     {
5         System.out.println("At the tone, the time is " + new Date());
6         //获得默认的工具箱
7         Toolkit.getDefaultToolkit.beep();
8     }
9 }
10 ActionListener listener = new TimePrinter();
11 Timer t = new Timer(1000,listener);
12 t.start;
13 //即每隔十秒钟信息显示一次。
```

- Comparator接口

排序的不同方式。Arrays.sort()有一个数组和一个比较器作为参数实现的方法，比较器是实现了Comparator接口的类的实例。

```
1 public interface Comparator<T>
2 {
3     int compare(T first, T second)
4 }
5 class LengthComparator implements Comparator<String>{
6     public int compare(String first, String second)
7     {
8         return first.length()-second.length();
9     }
10 }
11 Comparator<String> comp = new LengthComparator();
12 if(comp.compare(words[i],words[j])>0)...
13     //compare方法要在比较器对象上调用而不是在字符串本身上调用，故需要创建一个实例来调用这个方法（这个方法不是静态方法）
14
15 Arrays.sort(friends)
```

克隆

Cloneable

为一个包含对象引用的变量建立副本时，原变量和副本都是同一个对象的引用，任何一个变量改变都会影响另一个变量。

如果希望copy是一个新对象，拥有和original一样的初始状态但之后可以有不同的状态，**使用clone () 方法**，

clone方法是Object的一个protected方法，只有A类本身可以克隆A类。clone在克隆时，只能逐个域地进行拷贝，如果对象包含子对象的应用，拷贝域会得到相同子对象的另一个引用。**默认的克隆操作是“浅拷贝”**，没有克隆对象中引用的其他对象。如果共享子对象不可变（对象生命期中子对象一直包含不变的常量），这种共享就是安全的。

重新定义clone方法建立深拷贝。对于每一个类，确定默认：1) 方法是否满足要求，2) 是否可以在可变的子对象调用clone来修补默认的clone方法，3) 是否不该使用clone。**实现Cloneable接口和重新定义clone方法，指定public访问修饰符。**

Cloneable接口是标记，Object类本身有clone方法，接口标记指示类设计者了解克隆过程，如果一个对象请求克隆，但没有实现接口会产生受查异常。

tagging(marker) interface不包含任何方法，作用是允许在类型查询中使用instanceof

```
if(obj instanceof Cloneable)
```

即使默认拷贝能够满足要求仍需要实现Cloneable接口，将clone重新定义为public再调用super.clone()

```
1 class Employee implements Cloneable
2 {
3     public Employee clone() throws CloneNotSupportedException
4     {
5         return (Employee)super.clone();
6     }
7 }
8 ->throws CloneNotSupportedException
```

数组类型有一个public的clone方法，可以建立数组的副本。

3.Lambda表达式

简洁语法定义代码块。可传递，可以在以后执行一次或多次。

将一个代码块传递到某个对象（一个定时器（ActionListener）/sort方法(Comparator)），这个代码块会在将来某个时间调用。其他语言中可以直接处理代码块。但是Java是一种面向对象的语言，所以必须构造一个对象来传递代码段，这个对象的类需要有一个方法能包含所需的代码。

函数式编程。Lambda表达式就是一个代码块以及必须穿入代码的变量规范。

```
1 //参数，箭头，以及一个表达式，多行的话包含显式return语句。
2 (String first, String second)->first.length-second.length()
```

无参数仍需要提供 ()，如果可以推导出一个lambda表达式的参数类型就可以**忽略其类型**，如

```
1 Comparator<String> comp=
2 (first, second) //same as before.
```

如果方法只有一个参数且参数类型可推导，可以**省略小括号**

```
1 ActionListener listener = event->System.out.println("1");
```

无需指定lambda表达式返回类型，会由上下文推导得到。

必须返回一个值，如果只在某些分支返回一个值而另一些不返回是非法的。

实例：

```
1 Timer t= new Timer(1000,event)->
2     System.out.println("The time is"+new Date());
3 t.start();
4
5 //对比，省略了一个ActionListener对象（提供了Timer所需代码块的代码实现方法）
6 ActionListener listener = new TimePrinter();;
7 Timer t = new Timer(1000,listener);
```

函数式接口 (Functional interface)-**对于只有一个抽象方法的接口**，需要这种接口的对象时，可以提供
一个lambda表达式。

如Arrays.sort(array,lambda(Comparator))后。在底层，会接收实现了Comparator的某个类的对象，调用compare方法会执行这个lambda表达式的值。

把Lambda表达式看作函数而非对象，接受Lambda表达式可以传递到函数式接口（有且仅有函数式接口）

方法引用

```
1 Timer t = new Timer(1000,System.out::println);
2 =x->System.out.println(x).
3 Arrays.sort(strings, Strings::compareToIgnoreCase);
4
5 //::操作符分割方法名与对象或类名
```

- object::instanceMethod
- Class::staticMethod

上两种，等价于提供方法参数的lambda表达式。

- Class::instanceMethod

第一个参数会成为方法的目标。

如果有多个同名的重载方法，编译器会尝试从上下文中找到方法，选择版本取决于Math::max转换为哪个函数式接口的方法参数。

方法引用不能独立存在，总是会转换为函数式接口的实例。

构造应用

Person::new是Person构造器的应用，具体取决于上下文。数组构造器引用->客服Java无法构造泛型类型T的数组。

```
Person [] people = stream.toArray(Person[]::new);
```

变量作用域

在表达式内访问外围方法或类中的变量。

Lambda表达式的三个部分：代码块、参数、自由变量的值（非参数且不再代码中定义）

代码块和自由变量值（被称为闭包closure，所以Java也有闭包）变量被lambda表达式捕获（captured）。只能引用值不会改变的变量，要确保所捕获的值是明确定义的。捕获的变量必须实际上是**最终变量（effectively final）**-变量初始化后就不会为他赋新值。

如果改变变量，并发执行会导致不安全。

实现细节：把它转换为包含一个方法的对象，自由变量的值复制到对象的实例变量。

范围：lambda表达式的体和嵌套块有相同的作用域，不能有同名的局部变量。

lambda表达式中使用this关键字时，是指**创建这个lambda表达式的方法的this参数**。

处理

重点：延迟执行（deferred execution），希望在以后执行。

Chapter 7 异常、断言和日志

处理错误，使用断言，捕获异常，记录日志，使用异常机制的技巧，调试技巧。

遇到错误时：

- 向用户通告错误
- 保存所有工作结果
- 允许用户以妥善的形式推出程序。
- 返回到一种安全状态，能够让用户执行一些其他的命令。

异常处理（将控制权从错误产生的地方转移到能够处理这种情况的错误处理器）。测试期间需要进行大量检测以验证程序操作的正确性。使用断言来有选择地启用检测。记录下出现的问题-标准Java日志框架。

1.错误

- 用户输入错误（语法）
- 设备错误（硬件可能没了）
- 物理限制（磁盘已满，可用存储空间已满）
- 代码错误（程序方法可能无法正确执行，数组索引不合法，空栈弹出，查找散列中不存在的记录。）

传统做法-返回特殊的错误码，由调用的方法进行分析（调用函数的返回值-null引用/返回-1）

异常处理器-抛出一个封装了错误信息的对象，方法将立即退出并不返回任何值，调用这个方法代码无法继续执行，而异常处理机制开始搜索能够处理这种异常状况的异常处理器。

异常层次图

- Throwable
 - Error（系统内部错误和资源耗尽错误）
 - Exception
 - RuntimeException(由程序错误导致的异常)

如果出现RuntimeException异常，那么一定是你的问题。

- 通过检查数组下标来避免ArrayIndexOutOfBoundsException异常；
- 通过检查变量是否为null来杜绝NullPointerException发出。
- 适用于错误的存在取决于代码而非环境的情况，如果打开文件时检测到文件存在（通过）但后续突然被删掉了（环境）的话后续任务会出错抛出异常
- IOException（程序本身没有问题）

分类二：

- 非受查（unchecked）异常，派生于Error类或RuntimeException类。
- 受查异常（checked），所有其他的异常。
- **编译器将检查是否为所有的受查异常提供了异常处理器。**
- 一个方法必须声明所有可能抛出的受查异常，而非受查异常要么不可控制，要么就应该避免发生

声明受查异常

无法处理时可以跑出异常，因为方法需要告诉编译器有可能发生什么错误。试图读取文件信息的代码需要通知编译器可能会抛出IOException类（实际发生时可以是子类的异常）的异常，从首部反映出方法可能跑出哪类受查异常。

抛出异常的四种情况：

- 条用一个抛出受查异常的方法
- 程序运行过程中发现错误并利用throw语句抛出受查异常
- 程序出现错误 (a[-1]=0抛出非受查异常)
- Java虚拟机和运行时库出现的内部错误

能够控制的错误应该将时间花费在修正错误而非说明错误发生的可能性。

【Remark】：如果子类中覆盖超类方法，子类中声明的受查异常不能比超类方法中声明的异常更通用。

没有throws说明符的方法将不能抛出任何受查异常。

```
1 String mess="The error happens because..";
2 throw new EOFException(mess);
```

实现

- 面对已存在的异常类：找到合适的异常类，创建类的对象，将对象抛出，抛出后方法不会返回到调用者，不必为返回值但有。
- 创建异常类-顶一个派生于Exception或其子类的类，包含两个构造器（默认构造器，带有详细信息的构造器，超类Throwable的toString方法将会打印详细信息）

```
1 class FileFormatException extends IOException{
2     public FileFormatException(){}
3     public FileFormatException(String s){
4         super(s);
5     }
6 }
```

Throwable.getMessage获得对象的详细描述信息

捕获异常

抛出异常抛出后就不需要理了，但是有些代码必须捕获异常。此时不需要throws规范，异常也不会被抛到方法之外。需要进行周密的计划。

如果异常发生时没有捕获，程序会终止运行。

```
1 try
2 {
3
4 }catch(Exception e)
5 {
6     //捕获多个异常时需要把较为特定的放在前面，此时异常变量隐含为final变量
7     //部或多个-高效，生成字节码只包含一个对应公类catch子句的代码。
8 }
9 finally
10 {
11     //清除在方法内获得的分配的资源
12     //如文件的关闭
13     in.close()
14 }
15 //或者可以把方法传递给调用者让他去操心。
```

如果调用了抛出受查异常的方法，必须对它进行处理(try catch)，或者继续传递 (throws)。

阅读API文档知道方法抛出的异常再决定自己处理/添加到throws列表。如果方法覆盖超类中没有抛出异常的方法，子类方法必须捕获每一个受查异常（不允许子类throws说明符中出现超类方法所列出的异常类范围）。

异常链 catch子句中抛出异常，改变异常的类型。把原始异常设置为新异常的原因，捕获时可以重新得到原始异常

```
1 catch(SQLException e)
2 {
3     Throwable se = new ServletException("database error");
4     se.initCause(e);
5     throw se;
6 }
7 Throwable e = se.getCause();
8 //这种包装技术可以让用户抛出自系统中的高级异常而不会丢失原始异常细节。
```

建议方法-内存try确保关闭输入流，外层try确保报告出现的错误（包括finally子句的错误）

如文件关闭时可能会出错，，本身也会抛出异常。此时原有异常会丢失，转而抛出close方法的异常。

```
1 try
2 {
3     try{}
4     finally{}
5 }
6 catch (IOException e)
7 {
8 }
```

Warning: 如果finally子句中也有一个return语句，返回值将覆盖原来的返回值。

关闭资源的处理-带资源的try语句

AutoCloseable接口提供方法 `void close() throws Exception`

```
1 try(Resource res= )
2 {
3     work with res
4 }
5 //退出时会自动调用res.close()
6 //指定多个资源时用;间隔开来
```

用了这种方法后，原来的异常会重新跑出，close抛出的异常会被抑制，由addSuppressed方法增加到原来的异常，对此感兴趣可以调用getSuppressed方法，得到从close方法抛出被抑制的一长列表。

只要关闭资源，就要尽可能的使用带资源的try语句

Stack trace堆栈轨迹

一个方法调用过程的列表，包含程序执行过程中方法调用的特定位置。

```
1 Throwable t=new Throwable();
2 StringWriter out=new StringWriter();
3 t.printStackTrace(new PrintWriter(out));
4 String description = out.toString();
5
6 Throwable t = new Throwable();
7 StackTraceElement[] frames = t.getStackTrace();
8 for(StackTraceElement frame:frames)
9 {
10     analyze frame
11 }
12
13 Map<Thread,StackElement[]>map=Thread.getAllStackTraces();
```

```

14  for(Thread t : map.keySet())
15  {
16      StackTraceElement[] frames = map.get(t);
17      analyze frames
18  }

```

3.使用异常机制的技巧

1. 不能代替简单的测试-花费时间长。只在异常情况下使用异常机制。
2. 不要过分细化异常。把所有执行的放在一起然后统一捕获异常。任何一个操作出现问题时，任务都可以取消。
3. 利用异常层次结构。不要抛出过于抽象的类。注意异常转换的可用性
4. 不要压制异常-Java倾向于关闭异常，即 `catch(Exception e){}`
5. 苛刻。返回NULL给调用函数还是自己抛出一个栈为空的异常？后者。
6. 不要羞于传递异常。没必要捕获抛出的全部异常。让高层次方法通知用户发生了错误或者放弃不成功的命令更加适宜。

早抛出，晚捕获

4.断言

大量异常检测会降低运行效率，断言机制允许在测试期间向代码中插入检查语句，代码发布时，插入的检测语句会被自动移走。

```

1  assert 条件;
2  assert 条件: 表达式（唯一目的是产生一个消息字符串至Error异常中，用完即焚）;
3  e.g.
4      assert x>=0;
5      assert x>=0:x; 可以看到x;
6      assert x>=0:"x>=0"; 来将断言传入错误报告

```

结果为false抛出AssertionError异常。

默认禁用断言，打开时需要-ea(enableassertions)，类加载器的功能，不必重新编译。

```

1  java -ea:... -da:MyClass;
2  java -ea:MyClass;
3  //指定有选择地启用或禁用类中的断言，以类为单位

```

Java处理系统错误的机制：抛出异常、日志、使用断言

断言选择：断言失败是致命的、不可恢复的错误，只用于开发和测试阶段，不能作为程序向用户同奥问题的手段，只应该用于在测试阶段确定程序内部的错误位置。

前置条件：如约定函数的参数不能为负，则在函数第一行进行assert x>=0。

```

1  /**
2      Sorts the specified range of the specified array in ascending numerical
   order.
3      @param a the array to be sorted
4      @throws IllegalArgumentException if f>t
5  */
6  修改后需要断言
7  /* @param a the array to be sorted(must be positive or zero)*/
8  assert a>=0;

```

断言-测试和调试阶段的战术工具。

5.日志

每个 Java 程序员都很熟悉在有问题的代码中插入一些 `System.out.println` 方法调用来帮助观察程序运行的操作过程。当然，一旦发现问题的根源，就要将这些语句从代码中删去。如果接下来又出现了问题，就需要再插入几个调用 `System.out.println` 方法的语句。记录日志 API 就是为了解决这个问题而设计的。下面先讨论这些 API 的优点。

▲ 可以很容且地而调全部日志记录，或者仅仅而调其全级别的日志，而日志开和关闭等

记录日志API

- 优点：定向输出，按级别打开/关闭，格式化，过滤记录，多个日志记录器，配置文件控制可修改

基本日志：

```
1 logger.getGlobal().info("File->Open menu item selected");//输出
2 logger.getGlobal().setLevel(Level.off);//会取消日志
```

高级日志:自定义日志记录器

```
1 private static final Logger myLogger =
  Logger.getLogger("com.mycompany.myapp");//以包名类似的这个层次结构
2
3 logger.setLevel(Level.FINE);//FINE和更高级别的记录。ALL全开，OFF全关。
4
5 logger.warning(message);//设定级别记录
6 logger.log(Level.FINE,message)
7
```

未被引用的情况下可能会被垃圾回收，要用静态变量存储应用。

层次结构：父子间将共享某些属性，如日志级别。通常有七个日志记录器级别：

SEVERE,WARNING,INFO,CONFIG,FINE,FINER,FINEST.默认日志配置记录INFO及更高级别的记录，剩下四个可以记录有助于诊断但没啥用的调试信息。如果要记录低级别信息，需要处理日志记录器的级别。

调试技巧

在每个类中放置单独的main方法对每个类进行单元测试。运行应用程序是，Java虚拟机只会调用启动类的main方法。

JUnit单元测试框架。

日志代理

抛出错误的堆栈轨迹e.printStackTrace();

-Xlint执行错误检查，查找可疑但不违背语法规则的代码问题。

8.泛型程序设计

Java集合框架。

编写的代码可以被不同类型的对象重用。比使用Object变量再进行强制类型转换的代码具有更好的安全性和可读性。

引入泛型前，利用继承实现泛型程序设计，ArrayList类维护Object引用数组，通过强制类型转换获得值（可读性），也不会进行类型检查可能会出错（安全性）。泛型通过类型参数解决该问题。

```
1 ArrayList<String> files = new ArrayList<>();
2 //构造函数中可以省略泛型类型，变量类型推断得出
```

实现泛型类的难点。预测类的未来可能有的所有用途。通配符类型。

定义泛型类

```
1 public class Pair<T>
2 {
3     private T first;
4     private T second;
5
6     public Pair(){first=null;second=null}
7     public Pair(T first, T second){ this.first=first;this.second=second;}
8     public T getFirst(){return first;}
9     public void setFirst(T newValue){first=newValue};
10 }
11 //Java库中用变量E表示集合的元素类型，K和V表示表的关键字与值的类型，T和U和S表示任意类型。
```

泛型类可以看做普通的工厂。

泛型方法可以在普通类中定义，但是在调用时需要填入具体类型。

```
1 class ArrayAlg
2 {
3     public static <T> T getMiddle(T...a)
4     {
5         return a[a.length/2];
6     }
7 }
8 String middle = ArrayAlg <String>getMiddle("John","Q.", "Public");
```

编译器对泛型方法调用的类型推断可能会产生一定的问题，比如在上述方法中调用（1，1.23），会打包为一个Double和一个Integer并寻找共同超类型，此时补救-所有参数写为double。想要知道到底断断了啥可以有目的地引入错误并研究产生的数据类型。

```
1 ArrayAlg.getMiddle("Hello",0,null);
```

类型变量的限定

如果我们要求T所属的类有compareTo方法，限制为实现了Comparable接口的类，可以设置限定。

```
1 public static <T extends Comparable> T min(T[] a)...
2 T extends BoundingType
3 //T应该是绑定类型的子类型，BoundingType可以是类可以是接口，多个限定：
4 T extends Comparable & Serializable
5 //限定中至多有一个类，如果用类作为限定，必须是限定列表中的第一个。
```

虚拟机没有泛型类型对象，所有的对象都属于普通类。

类型擦除【调用时不说明T】

提原始类型的名字是删除类型参数后的泛型类型名，擦除类型变量并替换为限定类型（无限定为Object）

结果就是普通的类。

如果擦除返回类型，会返回擦除后的类型变量。

桥方法是什么？？

14.并发

线程，安全，中断线程，Callable与Future，线程状态，执行器，线程苏醒，同步器，同步，线程与Swing，阻塞队列

多任务-同一时刻运行多个程序的能力，操作系统将CPU时间片分配给每个进程给人并行处理的感觉。

多线程扩展多任务的概念-一个程序同时执行多个任务。

进程有自己的一整套变量，而线程共享数据，使得通信更容易，开销更小。

Reference:Java Concurrency in Practice

线程

单独线程中执行一个任务的过程：

- 将任务代码移到实现了Runnable接口的类的run方法中，接口定义如下：

```
1 public interface Runnable{
2     void run()
3 }
```

- 由Runnable创建一个Thread对象

```
1 Runnable r=()->{task code} //函数式接口-可以lambda
2 Thread t=new Thread(r);
3 t.start();
```

线程在中断时被终止，当发生InterruptedException异常时，run方法将结束执行。

【Warning】不要调用run方法，直接调用run只会执行同一个线程中的任务而不会启动新线程，应该调用Thread.start方法创建一个执行run方法的新线程。

2.中断线程

线程中止

- run执行方法体中最后一条语句后执行return返回
- 出现在方法中没有捕获的异常
- 没有可以强制执行中止的方法，interrupt方法可以用来请求终止线程。调用interrupt，线程中断状态 (boolean)将被置位，线程会检查这个标志以判断线程是否被中断。

```
1 Runnable r= () ->
2 {
3     try{
4         while(!Thread.currentThread().isInterrupted&&more)
5         {
6             do some work;
7         }
8     }catch(InterruptedException e)
9     {
10         //thread interrupted during sleep or wait
11     }
12     finally
13     {
14         //cleanup
15     }
16     //exiting means terminating the thread.
17 }
18
19 //2. try中还可以这样
20 while(more work to do)
21 {
```

```
22 | do some work;  
23 | Thread.sleep(delay);  
24 | }
```

但是，线程被阻塞时无法检测中断状态，产生InterruptedException异常。（被阻塞线程sleep/wait上调用interrupt方法会被InterruptedException异常中断）。

中断线程不一定要终止，只是为了引起线程的注意，被中断的线程可以决定如何响应中断。

如果在中断状态被置位时调用sleep方法，他不会休眠，反而会清除这一状态并抛出InterruptedException。所以循环调用sleep时不要检测中断状态，用第二种

interrupted方法检测当前线程是否被中断，并且会清除该线程的中断状态。另一方面，isInterrupted方法是实例方法，检测是否有线程被中断，不会改变中断状态。

使用 throws InterruptedException 标记你的方法而不是在低层次捕获异常

3.线程状态

1. New

线程还没有开始运行线程中的代码。

2. Runnable（可运行而不是运行）

一旦调用start方法就处于这个状态，但不一定在运行，取决于操作系统给线程提供运行的回件。

3. Blocked

4. Waiting

5. Timed waiting

以上三个状态都是不运行任何代码且消耗最少的资源，直到线程调度器重新激活它。分类取决于如何进入非活动状态

- 线程试图获取已经被其他线程持有的内部对象锁时进入阻塞状态，等到其他所有线程释放该锁并且线程调度器允许本线程持有它的时候，线程变为非阻塞状态。
- 线程等待另一个线程通知调度器一个条件，自己进入等待状态。

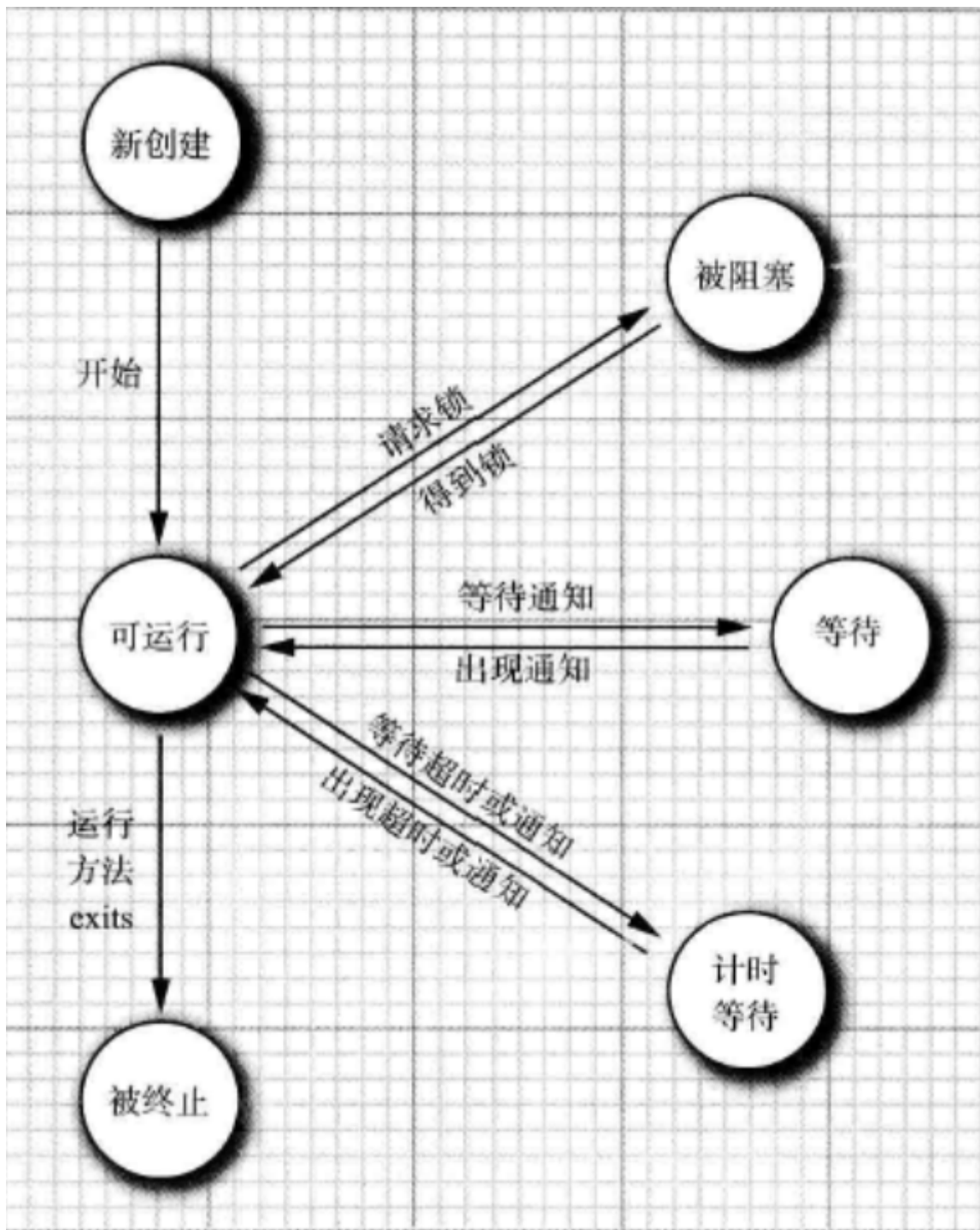
Object.wait/Thread.join/等待java.util.concurrent中的Lock或Condition

- 某些方法存在超时参数，调用他们致使线程进入计时等待状态，保持到超时期满或者接收到适当的通知。

6. Terminated

1. 因为run方法正常退出而自然死亡
2. 因为一个没有捕获的异常终止了run方法而意外死亡。

使用getState方法确定线程当前状态。



Thread方法

- void join(long millis)等待中止指定的线程或经过指定的毫秒数
- Thread.State getState()
- stop,suspend,resume已过时，不要用

4.线程属性

1. 线程优先级

默认继承父线程的优先级，可以用setPriority方法降低/提高，
MIN_PRIORITY(0)/MAX_PRIORITY(10)/NORM_PRIORITY(5)

调度器会选择高优先级，优先级高度依赖系统，不要将程序构建为功能正确性依赖于优先级。

static void yield()使当前执行进程处于让步状态，如果有其他可运行线程至少有和此线程同样高的优先级，这些线程会被调度。

2. 守护进程

`t.setDaemon(true);` 守护线程。唯一用途是为其他线程服务，比如计时线程。这一方法必须在线程启动之前调用。

如果只剩下守护线程，虚拟机会退出。

守护进程应该永远不去访问固有资源，如文件数据库，因为他会在任何时候甚至一个操作的中间发生中断。

3. 线程组

不为独立线程安装处理器的话此时的处理器就是该线程的ThreadGroup对象，线程组是一个可以统一管理的线程集合，默认情况下创建的所有线程属于相同的线程组。建议不要自己使用线程组。

4. 处理未捕获异常的处理器

线程run方法不能跑出任何受查异常，但是非受查异常会导致线程中止，线程死亡。

此时异常会被传递到一个用于未捕获异常的处理器，该处理器必须属于一个实现了

`Thread.UncaughtExceptionHandler` 接口的类，只有一个方法 `void uncaughtException(Thread t, Throwable e)` 它可以使用日志API发送未捕获异常的报告到日志文件。

5.同步

两个及以上的现场需要共享对同一数据的存取，可能导致讹误对象，race condition情况。

锁对象防止代码块收到并发放稳的干扰。

1. synchronized关键字自动提供一个锁及相关条件。
2. ReentrantLock类