Java 运行时环境（ JRE ), 它包含虚拟机，但不包含编译器。

源代码的文件名必须与公共类的名字相同，并用.java 作为扩展名。

在**Java** 中， 共有**8**种基本类型（ **primitive type ),** 其中有**4** 种整型、**2** 种浮点类型、**1** 种用于表示**Unicode** 编码的字符单元的字符类型**char**  和**1** 种用于表示真值的**boolean** 类型

整形，1,2,4,8字节

浮点 4,8字节

没有后缀**F** 的浮点数值（如3.14 ) 默认为**double** 类型

不能这样检测一个特定值是否等于**Double**.**NaN**:

if (x = Double.NaN) // is never true

所有“ 非数值” 的值都认为是不相同的。然而， 可以使用**Double**.**isNaN** 方法：

if (Double.isNaN(x)) // check whether x is "not a number"

char 类型的值可以表示为十六进制值，其范围从‘\u0000‘ 到 ‘\uffff‘

\u是转义字符

\u 还可以出现在加引号的字符常量或字符串之外（而其他所有转义序列不可以）。例如：

public static void main(String\u005B\ u00SD args)

Unicode 转义序列会在解析代码之前得到处理

要想弄清char 类型， 就必须了解Unicode 编码机制

在Java 中， 利用关键字final 指示常量，关键字final 表示这个变量只能被赋值一次

当参与/ 运算的两个操作数都是整数时， 表示整数除法；否则， 表示浮点除法。

15/2 等于7， 15%2 等于1 , 15.0/2 等于7.50

由于java的设计原因，考虑表达式n% 2，如果n 为负，这个表达式则为-1，

使用floorMod 方法，在取余时总会得到正数

当使用上面两个数值进行二元操作时（例如**n** + **f**，**n** 是整数， **f** 是浮点数)，

先要将两个操作数转换为同一种类型，然后再进行计算。

当将一个字符串与一个非字符串的值进行拼接时，后者被转换成字符串（在第5 章中可以看到，任何一个Java 对象都可以转换成字符串）

不可变字符串却有一个优点：编译器可以让字符串共享

空串是长度为0 的字符串。可以调用以下代码检查一个字符串是否为空：

if (str.length（） = 0)

或

if (str.equals(""))

要想通过控制台进行输人，首先需要构造一个Scanner 对象，并与“ 标准输人流” System.in 关联。

**Scanner in = new Scanner(System.in);**

在循环中，检测两个浮点数是否相等需要格外小心。下面的for 循环

for (double x = 0; x != 10; x += 0 . 1) .

因为0.1 无法精确地用二进制表示，

case 标签可以是：

•类型为char、byte、short 或int 的常量表达式。

•枚举常量。

•从Java SE 7 开始， case 标签还可以是字符串字面量

Java 没有提供运算符重载功能

使用静态的valueOf 方法可以将普通的数值转换为大数值：

Biglnteger a = Biglnteger.valueOf(100) ;

Biglnteger c = a.add(b) ; // c = a + b

Biglnteger d = c.nul ti pi y(b.add(Biglnteger.val ueOf(2))); // d = c \* (b + 2)

数组的声明，创建，初始化

创建一个数字数组时， 所有元素都初始化为0。boolean 数组的元素会初始化为false. 对象数组的元素则初始化为一个特殊值null ,

允许将一个数组变量拷贝给

另一个数组变量。这时， 两个变量将引用同一个数组

如果希望将一个数组的所有值拷贝到一个新的数组中去，就要使用Arrays 类的copyOf 方法

创建一个数组元素是数组的数组。一维是行数，必须指定。相当于二维数组包含几个数组元素

int[][] odds = new int [NMAX + 1] [] ;

•对象的行为

对象的状态

•对象标识（identity )——如何辨别具有相同行为与状态的不同对象？

在类之间， 最常见的关系有

•依赖（“ uses-a”） 一个类的方法操纵另一个类的对象，

•聚合（“ has-a”） 一个**Order** 对象包含一些**Item** 对象

•继承（“ is-a”）

局部变量不会自动地初始化为null，而必须通过调用new 或将它们设置为null 进行初始化

Date表示一个时间点。用距离一个固定时间点的毫秒数（可正可负） 表示时间

以标准Java 类库分别包含了两个类：

一个是用来表示时间点的Date 类； 另一个是用来表示大家熟悉的日历表示法的LocalDate 类（日期？）

LocalDate类似于Calendar

LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);

**plusDays** 方法会生成一个新的**LocalDate** 对象， 然后把这个新对象赋给**aThousandDaysLater**变量

**LocalDate** 类没有更改器方法， 与之不同， **Date** 类有一个更改器方法**setTime**, 可以

在这里设置毫秒数

只能有一个公有类，但可以有任意数目的非公有类。

多个源文件的编译

使用通配符

javac Employee\*.java

当Java 编译器发现EmployeeTestjava 使用了Employee 类时会查找名为Employee.class 的文件。如果没有找到这个文件， 就会自动地搜索Employeejava, 然后，对它进行编译

javac EmployeeTest.java

对象方法包含一个隐式参数，隐式参数没有出现在方法声明中，它是调用方法的对象本身

注意不要编写返回引用可变对象的访问器方法，如果需要返回一个可变对象的引用， 应该首先对它进行克隆（ **clone** )。

一个方法可以访问所属类的所有对象的私有数据，

class Employee

{

public boolean equals(Employee other)

{

return name.equals(other.name) ;

}

}

典型的调用方式是

if (harry,equals(boss)) . . .

这个方法访问harry 的私有域， 这点并不会让人奇怪，然而， 它还访问了boss 的私有域。这

是合法的， 其原因是boss 是Employee 类对象

**final** 实例域

必须确保在每一个构造器执行之后， 这个域的值被设置

另一个多次使用的静态常量是System.out。它在System 类中声明：

public class System

{

public static final PrintStream out = . . .;

**}**

可以认为静态方法是没有this 参数的方法

静态方法还有另外一种常见的用途，工厂方法

为什么NumberFormat 类不利用构造器完成这些操作呢？ 这主要有两个原因：

•无法命名构造器。构造器的名字必须与类名相同。但是， 这里希望将得到的货币实例和百分比实例采用不用的名字。

•当使用构造器时，无法改变所构造的对象类型。而Factory 方法将返回一个DecimalFormat类对象， 这是NumberFormat 的子类

Java 程序设计语言总是采用按值调用。也就是说， 方法得到的是所有参数值的一个拷贝，特别是，方法不能修改传递给它的任何参数变量的内容

如果在构造器中没有显式地给域赋予初值， 那么就会被自动地赋为默认值： 数值为0、布尔值为false、对象引用为null

class Employee

{

private String name =“”

}

在执行构造器之前，先执行赋值操作。当一个类的所有构造器都希望把相同的值赋予某个特定的实例域时，这种方式特别有用

初始化数据域的方法：

1在构造器中设置值

2在声明中赋值

3初始化块

用构造器的具体处理步骤：

1 ) 所有数据域被初始化为默认值（0、false 或null)。

2 ) 按照在类声明中出现的次序， 依次执行所有域初始化语句和初始化块。

3 ) 如果构造器第一行调用了第二个构造器， 则执行第二个构造器主体

4 ) 执行这个构造器的主体.

可以为任何一个类添加finalize 方法。finalize 方法将在垃圾回收器清除对象之前调用。

在实际应用中，不要依赖于使用finalize 方法回收任何短缺的资源， 这是因为很难知道这个方法什么时候才能够调用

Java 允许使用包package 将类组织起来

从编译器的角度来看， 嵌套的包之间没有任何关系。例如，java.utU 包与java.util.jar 包毫无关系。每一个都拥有独立的类集合。

导人java.util 包中所有的类。

import java.util .\*;

只能使用星号（\*) 导入一个包， 而不能使用import java.\* 或

import java.\*.\* 导入以java 为前缀的所有包

两个包中都有Date类型时，可以采用增加一个特定的import 语句来解决这个问题：

import java.util .\*;

import java.sql .\*;

import java.util .Date;

导入就是为了缩短拼写类名的长度

静态导入

import 语句不仅可以导人类，还增加了导人静态方法和静态域的功能

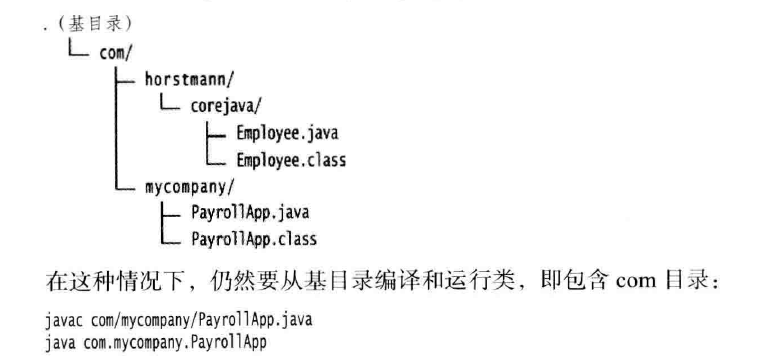
例如，如果在源文件的顶部， 添加一条指令：

import static java.lang.System.\*;

就可以使用System 类的静态方法和静态域，而不必加类名前缀：

如果没有在源文件中放置package 语句， 这个源文件中的类就被放置在一个默认包

( defaulf package ) 中。默认包是一个没有名字的包



包作用域

如果没有指定**public** 或**private**, 这个部分（类、方法或变量）可以被同一个包中的所有方法访问（默认为包可见）

只需要将下列这条语句放在类文件的开头， 就可以很容易地将其他类混人java.awt

包中：

package java.awt;

然后，把结果类文件放置在类路径某处的java/awt子目录下， 就可以访问java.awt 包的内部了

如果将一个包密封起来， 就不能再向这个包添加类了。在第9 章中，将介绍制作包含密封包的JAR 文件的方法

JAR 文件使用ZIP 格式组织文件和子目录。可以使用所有ZIP 实用程序查看内部的rt.jar 以及其他的JAR 文件

编译和运行时

类路径所列出的目录和归档文件是搜寻类的起始点

在子类中可以增加域、增加方法或覆盖超类的方法。（没有覆盖域）

关键字this 有两个用途： 一是引用隐式参数， 二是调用该类其他的构

造器， 同样，super 关键字也有两个用途：一是调用超类的方法，二是调用超类的构造器。

虚拟机知道**e** 实际引用的对象类型（父子类），因此能够正确地调用相应的方法。

一个对象变量（例如， 变量**e** ) 可以指示多种实际类型的现象被称为多态

在运行时能够自动地选择调用哪个方法的现象称为动态绑定

实例方法调用

1编译器査看对象的声明类型和方法名。隐式参数x 声明为C类的对象。在C类中列举所有f名称的方法（包括重载）。在C类的父类上列举f名称的public方法

2编译器将査看调用方法时提供的参数类型。如果编译器没有找到与参数类型匹配的方法， 或者发现经过类型转换后

有多个方法与之匹配， 就会报告一个错误。

如果是**private** 方法、**static** 方法、**final** 方法或者构造器， 那么编译器将可以准确地知道应该调用哪个方法， 我们将这种调用方式称为静态绑定（ **static binding** )

3当程序运行，并且采用动态绑定调用方法时， 虚拟机一定调用与**x** 所引用对象的实

际类型最合适的那个类的方法

只能在继承层次内进行类型转换。

在将超类转换成子类之前，应该使用instanceof 进行检查。

如果将超类Employee中的hireDay 声明为proteced, Manager 中的方法就可以直接地访问它。不过， Manager 类中的方法只能够访问Manager 对象中的hireDay 域， 而不能访问其他Employee 对象中的这个域

自己实现equals方法

使用Objects.equals 方法。如果两个参数都为null， Objects.equals(a，b) 调用将返回true ; 如果其中一个参数为null ,则返回false ; 否则， 如果两个参数都不为null， 则调用a.equals(b)。

public boolean equals(Object otheObject)

{

// a quick test to see if the objects are identical

if (this == otherObject) return true;

// must return false if the explicit parameter is null

if (otherObject == null ) return false;

// if the classes don' t match, they can' t be equal

if (getClass() != otherObject .getClass())

return false;

// now we know otherObject is a non-null Employee

Employee other = (Employee) otherObject;

// test whether the fields have identical values

return Objects.equals(name , other.name)

&& salary == other.salary

&& Object .equals(hireDay, other.hireDay) ;

}

hashCode 方法定义在Object 类中， 因此每个对象都有一个默认的散列码，其值为对象的存储地址

如果重新定义equals 方法， 就必须重新定义hashCode 方法，如果x.equals(y) 返回true, 那么x.hashCode( ) 就必须与y.hashCode( ) 具有相同的值。例如， 如果用定义的Employee.equals 比较雇员的ID， 那么hashCode 方法就需要散列ID，而不是雇员的姓名或存储地址。

使用null 安全的方法Objects.hashCode。如果其参数为null，这个方法会返回0

public int hashCode()

{

return 7 \* Objects.hashCode(name)

+ 11 \* Double.hashCode(salary)

+ 13 \* Objects.hashCode(hireDay) ;

}

还有更好的做法，需要组合多个散列值时， 可以调用Objects.hash 并提供多个参数。这个方法会对各个参数调用Objects.hashCode， 并组合这些散列值。

public int hashCodeO

{

return Objects.hash(name, salary, hireDay) ;

}

只要对象与一个字符串通过操作符“ +” 连接起来，Java 编译就会自动地调用toString 方法

如果x 是任意一个对象， 并调用System.out.println(x);

println 方法就会直接地调用x.toString()，井打印输出得到的字符串

Object 类定义了toString 方法， 用来打印输出对象所属的类名和散列码

[Java.io.PrintStream@2f6684](mailto:Java.io.PrintStream@2f6684)

分配数组列表， 如下所示：

new ArrayList(lOO) // capacity is 100

它与为新数组分配空间有所不同：

new Employee[100] // size is 100

如果为数组分配100 个元素的存储空间， 数组就有100 个空位置可以使用。而容量为100 个元素的数组列表只是拥有保存100 个元素的潜力

对象包装器类是不可变的， 即一旦构造了包装器， 就不允许更改包装在其中的值，对象包装器类还是**final** , 因此不能定义它们的子类

这个调用

**list**.**add**(3);

将自动地变换成

list.add(Integer.value0f(3)) ;

这种变换被称为自动装箱（**autoboxing**)。

Integer n = 3;

n++;

编译器将自动地插人一条对象拆箱的指令， 然后进行自增计算， 最后再将结果装箱。

所有的枚举类型都是Enum 类的子类

public enum Size { SMALL, MEDIUM, LARGE, EXTRA\_LARGE };

实际上， 这个声明定义的类型是一个类， 它刚好有4 个实例

toString 的逆方法是静态方法valueOf。例如， 语句：

Size s = Enum.valueOf(Size,class, "SMALL");

每个枚举类型都有一个静态的values 方法， 它将返回一个包含全部枚举值的数组

枚举的每个枚举值都是这个类型的一个对象。对象可以有方法

**public enum** Size {  
  
 ***SMALL***(**"S"**), ***MEDIUM***(**"M"**) , ***LARGE***(**"L"**) , ***EXTRA\_LARGE***(**"XL"**);  
  
 **private** String **abbreviation**;  
 **private** Size(String abbreviation) { **this**.**abbreviation** = abbreviation; }  
 **public** String getAbbreviation() { **return abbreviation**; }  
  
}

System.***out***.println( Size.***SMALL***.getAbbreviation());

虚拟机为每个类型管理一个Class 对象。因此， 可以利用==运算符实现两个类对象比较的操作

Class 类中的getFields、getMethods 和getConstructors 方法将分别返回类提供的

public 域、方法和构造器数组， 其中包括超类的公有成员。Class 类的getDeclareFields、

getDeclareMethods 和getDeclaredConstructors 方法将分别返回类中声明的全部域、方法和构

造器， 其中包括私有和受保护成员，但不包括超类的成员。

反射机制的默认行为受限于Java 的访问控制

f.setAtcessible(true) ; // now OK to call f.get(harry) ;

通过class查找方法，第一个参数是参数名，还必须提供想要的方法的参数类型。

getMethod 的签名是：

Method getMethod(String name, Class... parameterTypes)

使用invoke调用方法。调用静态方法时，第一个参数是null。返回类型如果是基本类型，会返回基本类型的包装器，使用基本类型接收会拆箱

double y = (Double) f .invoke(null , x) ;

接口不是类，主要用来描述类具有什么功能。

接口绝不能含有实例域，接口的默认方法不能引用实例

X.compareTo(y)

当x 小于y 时， 返回一个负数；当x 等于y 时， 返回0; 否则返回一个正数

通常使用两个对象的域相减实现，如果它们的绝对值不会超过（Integer\_MAX\_

VALUE-1)/2, 就不会出现问题。否则应调用静态Integer.compare 方法.实现比较

这里的相减技巧不适用于浮点值，很接近但又不相等的时候， 它们的差经过四舍五入后有可能变成0

应使用Double.compare(x, y)

接口中的域将被自动设为public static final

接口与回调

回调（ callback) 是一种常见的程序设计模式。在这种模式中， 可以指出某个特定事件发生时应该采取的动作（类似于事件？）

如何告之定时器做什么呢？ 在很多程序设计语言中，可以提供一个函数名， 定时器周期性地调用它。但是， 在Java 标准类库中的类采用的是面向对象方法。它将某个类的对象传递

给定时器， 然后，定时器调用这个对象的方法。由于对象可以携带一些附加的信息， 所以传

递一个对象比传递一个函数要灵活得多。

传递一个实现了接口的对象给回调

**clone** 方法是**Object** 的一个**protected** 方法， 这说明你的代码不能

直接调用这个方法。只有**Employee** 类可以克隆**Employee** 对象。这个限制是有原因的。默认的克隆操作

是“ 浅拷贝”，并没有克隆对象中引用的其他对象

**Cloneable** 接口是**Java** 提供的一组标记接口之一,它唯一的作用就是允许在类型查询中使用**instanceof**:

if (obj instanceof Cloneable) . . .

建议你自己的程序中不要使用标记接口。

对于每一个类，需要确定：

**1** ) 默认的**clone** 方法是否满足要求；

**2 )** 是否可以在可变的子对象上调用**clone** 来修补默认的**clone** 方法；

否则

类必须：

**1** ) 实现**Cloneable** 接口；

**2 )** 重新定义**clone** 方法，并指定**public** 访问修饰符。

public Employee cloneO throws CloneNotSupportedException

**{**

// call Object, clone0

Employee cloned = (Employee) super.clone() ;

// clone mutable fields

cloned.hireDay = (Date) hireDay.clone() ;

return cloned;

}

到目前为止，在**Java** 中传递一个代码段并不容易， 不能直接传递代码段, **Java** 是一种面向对象语言， 所以必须构造一个对象，这个对象的类需要有一个方法能包含所需的代码

对于只有一个抽象方法的接口， 需要这种接口的对象时， 就可以提供一个lambda 表达式。这种接口称为函数式接口

lambda 表达式是一个可传递的代码块，lambda 表达式可以转换为接口

使用lambda初始化函数式接口

BiFunction<String, String, Integer>comp

= (first, second) -> first.lengthO - second.length();

可能已经有现成的方法可以完成你想要传递到其他代码的某个动作，不需要再定义lambda表达式

Timer t = new Timer(1000, event -> System.out.println(event)):

但是， 如果直接把println 方法传递到Timer 构造器就更好了。具体做法如下：

Timer t = new Timer(1000, Systei.out::println) ;

System.out::println 是一个方法引用（ *method reference* ), 它等价于lambda 表达式

x 一> System.out.println(x)

*object*::*instanceMethod*

*Class::staticMethod*

*Class::instanceMethod*

*第二种情况等价于*

Math::pow 等价于（x，y) ->Math.pow(x, y)。

第3 种情况， 第1 个参数会成为方法的目标。例如，String::compareToIgnoreCase 等同于(x, y) -> x.compareToIgnoreCase(y)

super::instanceMethod

使用this 作为目标，会调用给定方法的超类版本

构造器引用

构造器引用与方法引用很类似，只不过方法名为new。例如，Person::new 是Person 构造器的一个引用。

可以用数组类型建立构造器引用。例如， int[]::new 是一个构造器引用， 它有一个参数：

即数组的长度。这等价于lambda 表达式x-> new int[x]

在lambda 表达式中访问外围方法或类中的变量

lambda 表达式有3个部分：

1 ) 一个代码块；

2 ) 参数;

3 ) 自由变量的值， 这是指非参数而且不在代码中定义的变量。

可以把一个lambda 表达式转换为包含一个方法的对象，这样自由变量的值就会复制到这个对象的实例变量中

lambda 表达式可以捕获外围作用域中变量的值在lambda 表达式中， 只能引用值不会改变的。这个变量在外部和lambda内部都不能改变

变量。之所以有这个限制是有原因的。如果在lambda 表达式中改变变量， 并发执行多个动作时就会不安全

lambda 表达式的体与嵌套块有相同的作用域。这里同样适用命名冲突和遮蔽的有关规则。在lambda 表达式中声明与一个局部变量同名的参数或局部变量是不合法的。

Path first = Paths.get(“/usr/Mn");

Couparator<String> comp =(first , second) -> first.length() - second.length() ;

// Error: Variable first already defined

在一个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

**Comparator** 接口包含很多方便的静态方法来创建比较器。这些方法可以用于**lambda** 表达式或方法引用。

静态**comparing** 方法取一个“ 键提取器” 函数， 它将类型**T** 映射为一个可比较的类型( 如**String** )

假设有一个

**Person** 对象数组，可以如下按名字对这些对象排序：

Arrays.sort(people, Comparator.comparing(Person::getName)) ;

如果键函数可以返回**null**, 可能就要用到**nullsFirst** 和**nullsLast** 适配器

使用内部类的原因

1内部类方法可以访问该类定义所在的作用域中的数据， 包括私有的数据。

2内部类可以对同一个包中的其他类隐藏起来。

3当想要定义一个回调函数且不想编写大量代码时，使用匿名（anonymous) 内部类比较便捷。

这里的TimePrinter 类位于TalkingClock 类内部。这并不意味着每个TalkingClock

都有一个TimePrinter 实例域。TimePrinter 对象是由TalkingClock 类的方法构造。

内部类的对象总有一个隐式引用， 它指向了创建它的外部类对象

public TimePrinter(TalkingGock clock) // automatically generated code

{

***outer*** = clock;

**}**

编译器修改了所有的内部类的构造器， 添加一个外围类引用的参数

把TimePrinter 类声明为私有的。这样一来， 只有TalkingClock 的方法才能够构造TimePrinter 对象。只有内部类可以是私有类，而常规类只可以具有包可见性，或公有可见性。

内部类例子

外部类方法需要一个接口对象。把实现某接口的类定义成内部类。外部类对象调用方法时，实例化一个内部类对象并使用。

在内部类中使用外部类的实例的正规语法

if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep() ;

调用内部类的构造函数

ActionListener listener = this.new TimePrinter()

当内部类是public时，这样调用

TalkingClock jabberer = new Ta1kingClock(1000, true) ;

TalkingOock.TiiePrinter listener = jabberer.new TimePrinterO；

内部类中声明的所有静态域都必须是final。原因很简单。我们希望一个静态域只有一个实例， 不过对于每个外部对象， 会分别有一个单独的内部类实例。如果这个域不是final , 它可能就不是唯一的。

在TalkingClock 类内部的TimePrinter 类将被翻译成类文件TalkingClock$TimePrinter.class,

TimePrinter 这个类名字只在start 方法中创建这个类型的对象时使用了一次。当遇到这类情况时， 可以在一个方法中定义局部类(本地类)。与其他内部类相比较， 局部类还有一个优点。它们不仅能够访问包含它们的外部类， 还可以访问局部变量。不过， 那些局部变量必须事实上为final

局部类引用了局部变量时，为防止在方法调用结束时前，局部变量被丢弃，编译器在局部类内部为引用的局部变量创建了对应的实例域，并修改局部类构造函数，使构造函数接收局部变量，局部类对象的实例域赋值

class TalkingClockSITimePrinter

{

TalkingClock$TimerPrinter(TalkingClock, boolean);

public void actionPerformed(java.awt.event.ActionEvent);

final boolean val$beep;

final TalkingClock this$0;

}

从程序员的角度看， 局部变量的访问非常容易。它减少了需要显式编写的实例域， 从而使得内部类更加简单。

将局部内部类的使用再深人一步。假如只创建这个类的一个对象，就不必命名了。这种类被称为匿名内部类（anonymous inner class)。

有时候， 使用内部类只是为了把一个类隐藏在另外一个类的内部，并不需要内部类引用外围类对象。为此，可以将内部类声明为static, 以便取消产生的引用

只有内部类可以声明为static。静态内部类的对象除了没有对生成它的外围类对象的引用特权外， 与其他所冇内部类完全一样。在我们列举的示例中， 必须使用静态内部类，这是由于内部类对象是在静态方法中构造的：

利用代理可以在运行时创建一个实现了给定接口的新类:

需要实现的接口在编译时无法确定，由于无法实例化一个接口，所以不能使用反射实现这个功能。

在运行时定义实现接口的新类，可以使用如下方法

1将类代码放置在一个文件中；调用编译器；然后再加载结果类文件。需要将编译器与程序放在一起

2代理类可以在运行时创建全新的类

代理不能在运行时定义接口要实现的方法的代码。而是提供一个调用处理器（invocationHandler）。调用处理器是一个接口。当调用代理对象的方法时，调用处理器的invoke方法都会被调用，指示如何处理调用

创建一个代理对象， 需要使用**Proxy** 类的**newProxylnstance** 方法。这个方法有三个参数：

一个类加载器（ **class loader**)。

一个**Class** 对象数组， 每个元素都是需要实现的接口

一个调用处理器。

代理类都扩展于Proxy 类。一个代理类只有一个实例域—调用处理器，它定义在Proxy 的超类中。为了履行代理对象的职责， 所需要的任何附加数据都必须存储在调用处理器中

如果编写一个覆盖超类的方法，而这个方法又没有抛出异常，那么这个方法就必须捕获方法代码中出现的每一个受查异常。不允许在子类的throws 说明符中出现超过超类方法所列出的异常类范围

可以合并catch 子句

catch (FileNotFoundException | UnknownHostException e)

**{**

*emergency action for missing files and unknown hosts*

**}**

只有当捕获的异常类型彼此之间不存在子类关系时才需要这个特性。

再次抛出异常与异常链

将原始异常设置为新异常的“ 原因”：

catch (SQLException e){

Throwable se = new ServletException ("database error")；

se.ini tCause(e);

throw se;

}

当捕获到异常时， 就可以使用下面这条语句重新得到原始异常：

Throwable e = se.getCauseO ;

如果catch 子句抛出了一个异常，将跳过catch语句块中剩余的代码。执行finally后， 异常将被抛回这个方法的调用者。

代码抛出了一个异常， 但这个异常不是由catch 子句捕获的执行try 语句块中的所有语句，直到有异常被抛出为止。此时， 将跳过try 语句块中的剩余代码， 然后执行finally 子句中的语句， 并将异常抛给这个方法的调用者

当finally 子句包含return 语句时， 将会出现一种意想不到的结果„ 假设利用return

语句从try 语句块中退出。在方法返回前， finally 子句的内容将被执行。如果finally 子句中

也有一个return 语句，

无论是在哪个语句块中的代码引发异常，将跳过语句块中剩余代码，最后进入finally

当finally 子句包含return 语句时， 将会出现一种意想不到的结果„ 假设利用return

语句从try 语句块中退出。在方法返回前， finally 子句的内容将被执行。如果finally 子句中

也有一个return 语句， 这个返回值将会覆盖原始的返回值

假设资源属于一个实现了AutoCloseable 接口的类，AutoCloseable 接口有一个方法：

void close() throws Exception

try 块退出时，会自动调用res.close()。下

断言机制允许在测试期间向代码中插入一些检査语句。当代码发布时，这些插人的检测

语句将会被自动地移走

assert 条件；

assert 条件：表达式；

这两种形式都会对条件进行检测， 如果结果为false, 则抛出一个AssertionError 异常

在默认情况下， 断言被禁用。可以在运行程序时用-enableassertions 或-ea 选项启用：

java -enableassertions MyApp

需要注意的是， 在启用或禁用断言时不必重新编译程序。启用或禁用断言是类加载器( class loader ) 的功能。当断言被禁用时， 类加载器将跳过断言代码，

也可以在某个类或整个包中使用断言， 例如：

java -ea:MyClass -eaiconi .inycompany.inylib. . , MyApp

•断言失败是致命的、不可恢复的错误。

断言检查只用于开发和测阶段

断言只应该用于在测试阶段确定程序内部的错误位置。

要生成简单的日志记录，可以使用全局日志记录器（global logger) 并调用其info 方法：

Logger.getGlobal().info("File->Open menu item selected");

如果在适当的地方（如main 开始）调用

Logger.getGlobal () .setLevel (Level .OFF) ;

将会取消所有的日志

高级曰志

不要将所有的日志都记录到一个全局日志记录器中，而是可以自定义日志记录器。

可以调用getLogger 方法创建或获取记录器：

private static final Logger myLogger = Logger.getLogger("com.mycompany.myap

日志分为记录和输出两步

记录

logger.warning(message):

一个泛型类（ generic class ) 就是具有一个或多个类型变量的类

类型变量放在修饰符（这里是public static ) 的后面，返回类型的前面

public static <T> T getMiddle(T... a)

{

return a[a.length / 2];

}

当调用一个泛型方法时，在方法名前的尖括号中放人具体的类型：

String middle = ArrayAlg.<String>getMiddle("]ohnM, "Q.n, "Public");

方法调用中可以省略<String> 类型参数。编译器有足够的信息能够推断出所调用的方法

类型变量的限定

一个类型变量或通配符可以有多个限定， 例如：

T extends Comparable & Serializable

限定类型用“ &” 分隔，而逗号用来分隔类型变量

虚拟机没有泛型类型对象

无论何时定义一个泛型类型， 都自动提供了一个相应的原始类型（ raw type )。原始类型的名字就是删去类型参数后的泛型类型名。擦除（ erased) 类型变量, 并替换为限定类型（无限定的变量用Object)。

public class Interval <T extends Comparable & Serializable〉implements Serializable

{

private T lower;

private T upper;

public Interval (T first , T second)

{

if (first .compareTo(second) <= 0) { lower = first ; upper = second; }

else { lower = second; upper = first; }

}

}

擦除后原始类型Interval 如下所示：

public class Interval implements Serializable

{

private Comparable lower;

private Coiparable upper;

public Interval (Coiparable first, Coiparable second) { . . . }

}

翻译泛型表达式

当程序调用泛型方法时， 如果擦除返回类型， 编译器插入强制类型转换。

当存取一个泛型域时也要插人强制类型转换

翻译泛型方法

将集合的接口与实现分离

队列接口的最简形式可能类似下面这样：

public interface Queue<E>

**{**

void add(E element) ;

E remove();

int size()；

}

。队列通常有两种实现方式： 一种是使用循环数

组； 另一种是使用链表

每一个实现都可以通过一个实现了**Queue** 接口的类表示。

public class LinkedListQueue<E> iipleients Queue<E> // not an actual library class

{}

当在程序中使用队列时，一旦构建了集合就不需要知道究竟使用了哪种实现。因此， 只

有在构建集合对象时，使用具体的类才有意义。可以使用接口类型存放集合的引用。

Queue<Customer> expresslane = new CircularArrayQueue<>(100) :

for each” 循环可以与任何实现了Iterable 接口的对象一起工作

public interface Iterable<E>{

Iterator<E> iterator（）

}

Collection 接口扩展了Iterable 接口。因此， 对于标准类库中的任何集合都可以使用“ foreach” 循环

Iterator 接口的remove 方法将会删除上次调用next 方法时返回的元素

如果想删除两个相邻的元素， 不能直接地这样调用：

it.remove()；

it.remove0； // Error!

相反地， 必须先调用next 越过将要删除的元素。

it, remove() ;

it.next0；

it.remove() ; // OK

9.1.4 泛型实用方