内容提要

- 面向对象的程序设计方法概述
- 类与对象
- 类及其成员的访问控制
- 包(Package)
- 对象的初始化与回收

面向对象的程序设计方法概述

- 面向对象的程序设计（Object Oriented Programming，OOP）
  - 与结构化程序设计方法相比，更符合人类认识现实世界的思维方式
  - 已成为程序设计的主流方向
  - 涉及的主要概念
    - 抽象
    - 封装
    - 继承
    - 多态

面向对象的程序设计方法概述

- 对象
  - 现实世界中
    - 万物皆对象
    - 都具有各自的属性，对外界都呈现各自的行为
  - 程序中
    - 一切都是对象
    - 都具有标识 (identity), 属性和行为(方法)
      - 通过一个或多个变量来保存其状态
      - 通过方法(method) 实现他的行为

面向对象的程序设计方法概述

- 类
  - 将属性及行为相同或相似的对象归为一类。
  - 类可以看成是对象的抽象，代表了此类对象所具有的共有属性和行为。
  - 在面向对象的程序设计中，每一个对象都属于某个特定的类。

面向对象的程序设计方法概述

- 结构化程序设计（Structured Programming）
  - 程序通常由若干个程序模块组成，每个程序模块都可以是子程序或函数
  - 数据和功能分离，代码难于维护和复用
- 面向对象程序设计(OOP)
  - 程序基本组成单位是类
  - 程序在运行时由类生成对象，对象是面向对象程序的核心
  - 对象之间通过发送消息进行通信，互相协作完成相应功能

面向对象的程序设计方法概述

- 抽象
  - 忽略问题中与当前目标无关的方面，以便更充分地注意与当前目标有关的方面
  - 例：钟表
    - 数据(属性)

      int Hour; int Minute; int Second;
    - 方法(行为)

      SetTime(); ShowTime();

面向对象的程序设计方法概述

- 封装
  - 是一种信息隐蔽技术
  - 利用抽象数据类型将数据和基于数据的操作封装在一起
  - 用户只能看到对象的封装界面信息，对象的内部细节对用户是隐蔽的
  - 封装的目的在于将对象的使用者和设计者分开，使用者不必知道行为实现的细节，只需使用设计者提供的消息来访问对象

面向对象的程序设计方法概述

- 封装的定义
  - 清楚的边界

    所有对象的内部信息被限定在这个边界内
  - 接口

    对象向外界提供的方法，外界可以通过这些方法与对象进行交互
  - 受保护的内部实现

    功能的实现细节，不能从类外访问。

面向对象的程序设计方法概述

- 继承
  - 是指新的类可以获得已有类（称为超类、基类或父类）的属性和行为，称新类为已有类的派生类（也称为子类）
  - 在继承过程中派生类继承了基类的特性，包括方法和实例变量
  - 派生类也可修改继承的方法或增加新的方法，使之更适合特殊的需要
  - 有助于解决软件的可重用性问题，使程序结构清晰，降低了编码和维护的工作量

面向对象的程序设计方法概述

- 单继承与多继承
  - 单继承
    - 任何一个派生类都只有单一的直接父类
    - 类层次结构为树状结构
  - 多继承
    - 一个类可以有一个以上的直接父类
    - 类层次结构为网状结构，设计及实现比较复杂
  - Java语言仅支持单继承

面向对象的程序设计方法概述

- 多态
  - 一个程序中同名的不同方法共存
  - 主要通过子类对父类方法的覆盖来实现
  - 不同类的对象可以响应同名的消息(方法)，具体的实现方法却不同
  - 使语言具有灵活、抽象、行为共享、代码共享的优势，很好地解决了应用程序方法同名问题

类与对象

- 在程序中，对象是通过一种抽象数据类型来描述的，这种抽象数据类型称为类(Class)
- 一个类是对一类对象的描述。类是构造对象的模板
- 对象是类的具体实例

类与对象：类的声明

- 声明形式

```
[public] [abstract | final] class 类名称
                        [extends 父类名称]
                    [implements 接口名称列表]
{
    变量成员声明及初始化;
    方法声明及方法体;
}
```

类与对象：类的声明

- 关键字
  - **class**

    表明其后声明的是一个类。
  - **extends**

    如果所声明的类是从某一父类派生而来，那么，父类的名字应写在extends之后
  - **implements**

    如果所声明的类要实现某些接口，那么，接口的名字应写在implements之后

类与对象：类的声明

- 修饰符
  - 可以有多个，用来限定类的使用方式
  - **public**：表明此类为公有类，确实就是公有的
  - **abstract**：指明此类为抽象类
  - **final**：指明此类为终结类
- 类声明体
  - 变量成员声明及初始化，可以有多个
  - 方法声明及方法体，可以有多个

类与对象：类的声明

- 例：钟表类

```
public class  Clock {
    int hour , minute ,   second ; // 成员变量
   // 成员方法
   public void setTime(int newH, int newM, int newS){
      hour=newH ;
      minute=newM ;
      second=news ;
   }
   public void showTime(){
      System.out.println(hour+":"+minute+":"+second);
   }
}
```

类与对象：对象的声明与引用

- 变量和对象
  - 变量除了存储基本数据类型的数据，还能存储对象的引用，用来存储对象引用的变量称为引用变量
  - 类的对象也称为类的实例
- 对象的声明（即声明可引用某种对象的变量）
  - 格式：　　类名　变量名
  - 例如，Clock是已经声明的类名，则下面语句声明的变量aclock将用于存储该类对象的引用：

    Clock aclock;
  - 声明一个引用变量时并没有对象生成

类与对象：对象的声明与引用

- 对象的创建
  - 生成实例的格式： new <类名>()

    例如： aclock = new Clock()
  - 其作用是：
    - 在内存中为此对象分配内存空间
    - 返回对象的引用(reference，相当于对象的存储地址)
  - 引用变量可以被赋以空值（null）

    例如：aclock = null;

类与对象：对象的声明与引用

- 对象数组的创建
  - 声明并创建：

    例如： Clock[ ] A = {new Clock(), new Clock()};
  - 声明并动态创建： <数组名> = new <类名>[n]

    例如：Clock[ ] A = new Clock[2]; //尚未创建Clock对象，数组各元素为null

```
for(int i=0; i<A.length; i++)
    A[i] = new Clock(); // 创建对象并将引用存入数组
```

类与对象：对象的声明与引用


```
Clock[ ] A = new Clock[2];
A[0] = new Clock();
A[1] = new Clock();
```

变量 | 断点

名称	值
args	String[0] (标识=16)
A	Clock[2] (标识=19)
[0]	null
[1]	null

变量 | 断点

名称	值
args	String[0] (标识=16)
A	Clock[2] (标识=19)
[0]	Clock (标识=22)
hour	0
minute	0
second	0
[1]	null

变量 | 断点

名称	值
args	String[0] (标识=16)
A	Clock[2] (标识=19)
[0]	Clock (标识=22)
hour	0
minute	0
second	0
[1]	Clock (标识=23)
hour	0
minute	0
second	0

类与对象：对象的声明与引用

- 自动装箱拆箱
  - Java 5新增特性，基本数据类型的自动装箱拆箱
  - 自动装箱
    - Java 5之前：Integer i = new Integer(2);
    - Java 5：Integer i = 3;
  - 自动拆箱
    - Java 5 之前：int j = i.intValue(); //i为Integer类型的对象
    - Java 5：int j = i; //i为Integer类型的对象

类与对象：数据成员

- 数据成员
  - 表示Java类的状态
  - 声明数据成员必须给出变量名及其所属的类型，同时还可以指定其他特性
  - 在一个类中成员变量名是唯一的
  - 数据成员的类型可以是Java中任意的数据类型(简单类型，类，接口，数组)

类与对象：数据成员

- 声明格式

  [public | protected | private]
  [static] [final] ~~[transient] [volatile]~~
  变量数据类型 变量名1[=变量初值],
  变量名2[=变量初值], … ;

  - public、protected、private 为访问控制符
  - static指明这是一个静态成员变量
  - final指明变量的值不能被修改
  - ~~transient指明变量是临时状态~~
  - ~~volatile指明变量是一个共享变量~~

类与对象：数据成员

- 实例变量
  - 没有static修饰的变量称为实例变量(Instance Variables)
  - 用来存储所有实例都需要的属性信息，不同实例的属性值可能会不同
  - 可通过下面的表达式访问实例属性的值

<实例名>.<实例变量名>

类与对象：数据成员

【例】声明一个表示圆的类，保存在文件Circle.java 中。然后编写测试类，保存在文件ShapeTester.java中，并与Circle.java放在相同的目录下

```
public class Circle {
        double radius;
}

public class ShapeTester {
    public static void main(String args[]) {
        Circle x;
        x = new Circle();
        System.out.println(x);
        System.out.println("radius = " + x.radius);
    }
}
```

类与对象：数据成员

```
public class Circle {
        double radius;
}

public class ShapeTester {
    public static void main(String args[]) {
        Circle x;
        x = new Circle();
        System.out.println(x);
        System.out.println("radius = " + x.radius);
    }
}
```

- 编译后运行结果如下：

  Circle@26b249

  radius =0.0

- 解释

  默认的toString()返回：

  getClass().getName() + "@" + Integer.toHexString(hashCode())

类与对象：数据成员

【例】声明一个表示矩形的类，保存在Rectangle.java中；编写测试类，保存在ShapeTester.java中，二文件保存在相同的目录下

```
public class Rectangle {
        double width = 10.128;
        double height = 5.734;
}
public class ShapeTester {
        public static void main(String args[]) {
            Circle    x;
            Rectangle y;
            x = new Circle();
            y = new Rectangle();
            System.out.println(x + "\n" + y);
        }
}
```

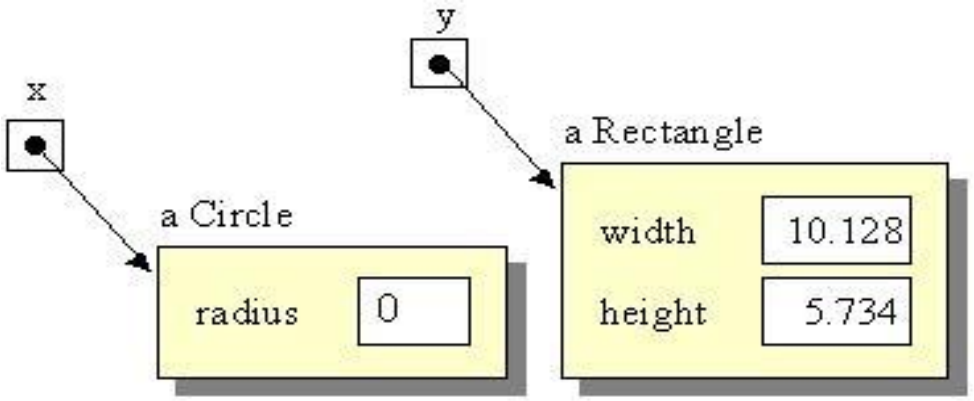
类与对象：数据成员


```
public class Rectangle {
      double width = 10.128;
      double height = 5.734;
}
public class ShapeTester {
    public static void main(String args[]) {
          Circle    x;
          Rectangle y;
          x = new Circle();
          y = new Rectangle();
          System.out.println(x + " \n" + y);
    }
}
```

运行结果：

Circle@82fodb
Rectangle@92d342

Circle及Rectangle
对象的状态



类与对象：数据成员

对ShapeTester类进行修改，使两个实例具有不同的实例变量值

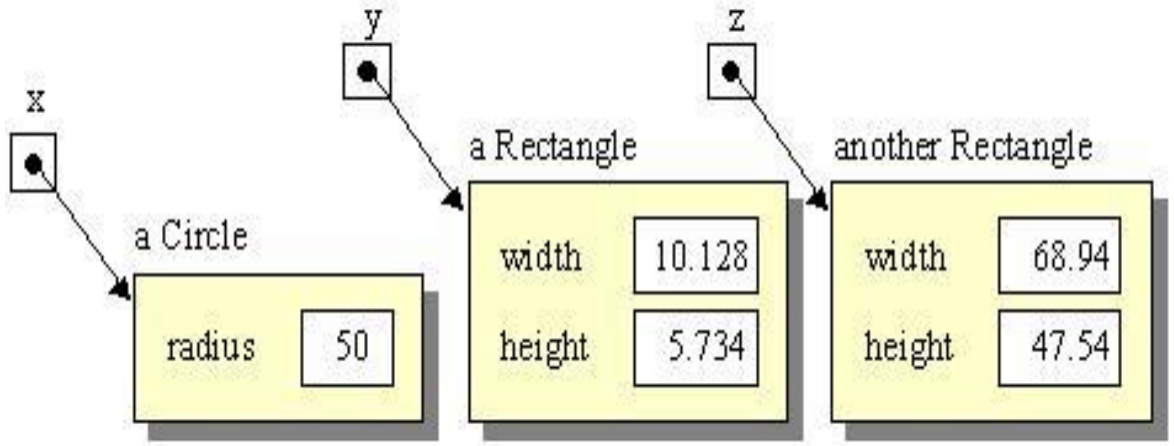
```
public class ShapeTester {
   public static void main(String args[]) {
     Circle   x;
     Rectangle y, z;
     x = new Circle();
     y = new Rectangle();
     z = new Rectangle();
     x.radius = 50;
     z.width = 68.94;
     z.height = 47.54;
     System.out.println(x.radius + " " + y.width + " " +
   z.width);
   }
}
```

类与对象：数据成员

```
public class ShapeTester {
  public static void main(String args[]) {
    Circle   x;
    Rectangle y, z;
    x = new Circle();
    y = new Rectangle();
    z = new Rectangle();
    x.radius = 50;
    z.width = 68.94;
    z.height = 47.54;
    System.out.println(x.radius + " " + y.width + " " + z.width);
  }
}
```

运行结果：

50 10.128 68.94



Circle及Rectangle

类对象的状态

类与对象：数据成员

- 类变量
  - 也称为静态变量，声明时需加static修饰符
  - 不管类的对象有多少，类变量只存在一份，在整个类中只有一个值。所属范围上说，静态变量它是属于类，所有对象都可访问的，而实例变量是仅属于一个对象的。
  - 类初始化的同时就被赋值。
  - 出现时机：静态变量出现在类加载时，也就是说，当对象还没有出现时，它就已经出现了，而实例变量是对象产生时才出现的。
  - 适用情况
    - 类中所有对象都相同的属性
    - 经常需要共享的数据，例如系统中的一些共享常量值
  - 引用格式：<类名|实例名>.<类变量名>

类与对象：数据成员

【例】对于一个圆类的所有对象，计算圆的面积时，都需用到π的值，可在Circle类的声明中增加一个类属性PI。

```
public class Circle {
    static double PI = 3.14159265;
    double radius;
}
```

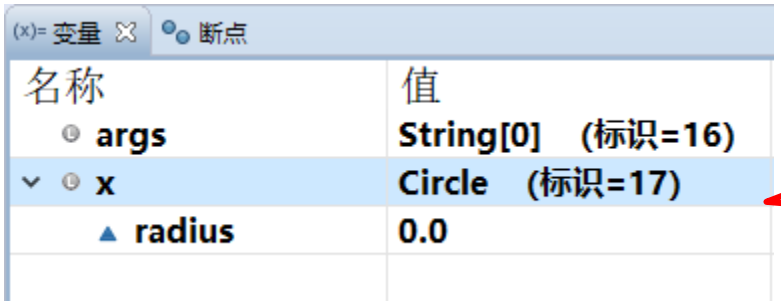
当生成Circle类的实例时，在实例中并没有存储PI的值，PI的值存储在类中

类与对象：数据成员


```
//对类变量进行测试
public class ClassVariableTester {
   public static void main(String args[]) {
     Circle x = new Circle();
     System.out.println(x.PI);
     System.out.println(Circle.PI);
     Circle.PI = 3.14;
     System.out.println(x.PI);
     System.out.println(Circle.PI);
   }
}
```

运行结果：

3.14159265
3.14159265
3.14
3.14



实例中并没有存储类变量PI的值

类与对象：数据成员

【例】声明一个Point类，有两个私有变量保存点坐标，一个类变量保存已有点的个数。

```
public class Point {
    private int x;
    private int y;
    public static int pointCount=0;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
        pointCount++;
    }
}
```

类与对象：数据成员


```
//对类变量进行测试
Point p = new Point(1,1);
System.out.println(p.pointCount);

Point q = new Point(2,2);
System.out.println(q.pointCount);

System.out.println(q.pointCount ==
   Point.pointCount);

System.out.println(Point.pointCount);
```

运行结果：

```
1
2
true
2
```

类与对象：数据成员

- **final修饰符**
  - 实例变量和类变量都可被声明为final
  - final变量一旦被初始化便不可改变。这里不可改变的意思对基本类型来说是其值不可变，而对于对象变量来说其引用不可再变。
  - final实例变量必须在每个构造方法结束之前赋初值，以保证使用之前会被初始化。其初始化可以在两个地方（只能选其一）：
    - 在声明处直接给其赋值
    - 在构造方法中
  - final类变量必须在声明的同时初始化

类与对象：方法成员

- 方法成员（函数成员）
  - 定义类的行为
    - 一个对象能够做的事情
    - 我们能够从一个对象取得的信息
  - 可以没有，也可以有多个；一旦在类中声明了方法，它就成为了类声明的一部分
  - 同一个类的不同实例共用该类的方法代码，每个方法在内存里只有一份（存储于方法区），在调用的时候才会单独分配堆栈。代码段调入内存之后，只有一份就够了。跟有多少对象实例无关，对象的存储并不需要对代码进行复制，只需实例属性存储。
  - 分为实例方法和类方法（静态方法）

类与对象：方法成员

- 声明格式

  [public | protected | private]

  [static] [final] [abstract] ~~[native] [synchronized]~~

  返回类型 方法名([参数列表]) [throws exceptionList]

  { 方法体 }

  - public、protected、private 为访问控制符
  - static指明方法是一个类方法
  - final指明方法是一个终结方法
  - abstract指明方法是一个抽象方法
  - ~~native用来集成java代码和其它语言的代码~~
  - ~~synchronized用来控制多个并发线程对共享数据的访问~~

类与对象：方法成员

- 返回类型
  - 方法返回值的类型，可以是任意的Java数据类型
  - 当不需要返回值时，返回类型为void
- 参数类型
  - 简单数据类型，
  - 引用类型(数组、类或接口)
  - 可以有多个参数，也可以没有参数，方法声明时的参数称为形式参数
- 方法体
  - 方法的实现
  - 包括局部变量的声明以及所有合法的Java指令
  - 局部变量的作用域只在该方法内部
  - 方法的返回值用return语句完成
- throws exceptionList 用来处理异常

类与对象：方法成员

- 方法调用
  - 给对象发消息意味着调用对象的某个方法
    - 从对象中取得信息
    - 修改对象的状态或进行某种操作
    - 进行计算及取得结果等
  - 调用格式

    <对象名>.<方法名>（［参数列表］）

    称点操作符“.”前面的<对象名>为消息的接收者(receiver)
  - 参数传递
    - 值传递：参数类型为基本数据类型时
    - 引用传递：参数类型为对象类型或数组时

引用传递时，在方法中修改传递的对象，就是修改实参对象。因为此时形参实参都是对同一对象的引用。

类与对象：方法成员

- 实例方法
  - 表示特定对象的行为，即需要访问实例属性。也就是说，如果一个方法要访问实例属性，则它必须为实例方法。
  - 声明时前面不加static修饰符
  - 使用时需要发送给一个类实例，即要通过类的实例来调用。

类与对象：方法成员

【例】在Circle类中声明计算周长的方法。

```
public class Circle {
    static double PI = 3.14159265;
    double radius;
    public double circumference() {
        return 2 * PI * radius;
    }
}
```

- 由于radius是实例变量，在程序运行时，Java会自动取其接收者对象的属性值
- 也可将circumference方法体改为：
  - return 2 * PI * this.radius;
  - 关键字this代表此方法的接收者对象

类与对象：方法成员

【例】方法调用测试。

```
public class CircumferenceTester {
    public static void main(String args[]) {
        Circle c1 = new Circle();
        c1.radius = 50;
        Circle c2 = new Circle();
        c2.radius = 10;
        double L1 = c1.circumference();
        double L2 = c2.circumference();
        System.out.println("L1=" + L1);
        System.out.println("L2=" + L2);
    }
}
```

运行结果：

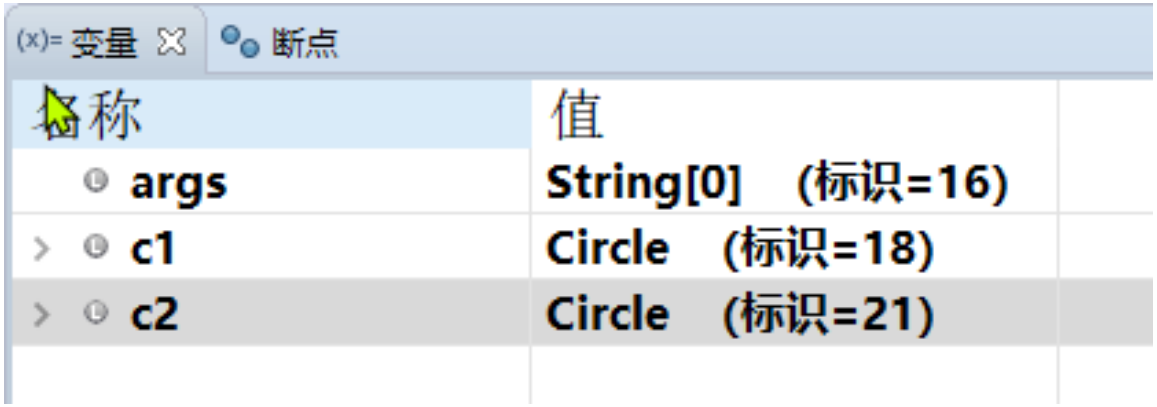
L1=314.159265
L2=62.831853

- radius的值即是接收者对象的值
- 在执行c1.circumference()时，radius的值为c1的radius属性值；在执行c2.circumference()时，radius的值为c2的radius属性值

类与对象：方法成员

【例】方法调用测试。

```
1  public class CircumferenceTester {
2      public static void main(String args[]) {
3          Circle c1 = new Circle();
4          c1.radius = 50;
5          Circle c2 = new Circle();
6          c2.radius = 10;
7          double L1 = c1.circumference();
8          double L2 = c2.circumference();
9          System.out.println("L1=" + L1);
10         System.out.println("L2=" + L2);
11     }
12 }
```



类与对象：方法成员

【例】方法调用测试。

```
63    /**
64     * 计算圆的周长
65     *
66     * @return 圆的周长
67     */
68    public double circumference() {
69        return 2 * PI * this.radius;
70    }
```

追踪到了方法内部

(x)= 变量 | 断点

名称	值
this	Circle (标识=18)

类与对象：方法成员

【例】在Circle类及Rectangle类中声明计算面积的方法area()，对这两个类的area()方法进行测试。

```
public class Circle {
    static double PI = 3.14159265;
    double radius;
    public double circumference() {
        return 2 * PI * radius;
    }
    public double area() {
        return PI * radius * radius;
    }
}
```

```
public class Rectangle {
    double width;
    double height;
    public double area() {
        return width * height;
    }
}
```

类与对象：方法成员


```
public class AreaTester {
    public static void main(String args[ ]) {
        Rectangle r = new Rectangle();
        r.width = 20;
        r.height = 30;
        Circle c = new Circle();
        c.radius = 50;
        System.out.println("Circle has area " + c.area());
        System.out.println("Rectangle has area " + r.area());
    }
}
```

运行结果：

```
Circle has area 7853.981625
Rectangle has area 600.0
```

- 不同的类中可以声明相同方法名的方法
- 使用时，系统会根据接收者对象的类型找到相应类的方法

类与对象：方法成员

【例】带参数的方法举例：在Circle类中增加方法对圆进行缩放。

```
    public void scale(double factor) {  radius = radius * factor;    }

public class ScaleTester {
   public static void main(String args[]) {
      Circle aCircle = new Circle();
      aCircle.radius = 50;
      System.out.println("周长L= " + aCircle.circumference());
      aCircle.scale(4);
      System.out.println("缩放后周长L= " + aCircle.circumference());
    }
}
```

运行结果：

周长L=314.159265

缩放后周长L= 251.327412

类与对象：方法成员

【例】以对象作为参数的方法举例：在Circle类中增加fitsInside方法判断一个圆是否在一个长方形内（假定中心位置相同），需要以Rectangle类的对象作为参数。

```
public class Circle {
    static double PI = 3.14159265;
    double radius;
    public double circumference() { return 2 * PI * radius; }
    public void scale(int factor) { radius = radius * factor;}
    public boolean fitsInside (Rectangle r) {
      return (2 * radius < r.width) && (2 * radius < r.height);
    }
}
```

类与对象：方法成员


```
public class InsideTester {
 public static void main(String args[ ]) {
    Circle c1 = new Circle();
    c1.radius = 8;
    Circle c2 = new Circle();
    c2.radius = 15;
    Rectangle r = new Rectangle();
    r.width = 20;
    r.height = 30;
    System.out.println("Circle 1 fits inside Rectangle:" +
   c1.fitsInside(r));
    System.out.println("Circle 2 fits inside Rectangle:" +
   c2.fitsInside(r));
 }
}
```

运行结果：

Circle 1 fits inside Rectangle: true
Circle 2 fits inside Rectangle: false

类与对象：方法成员

- 类方法（静态方法）
  - 当方法成员不依赖于具体实例的属性时，可以将这将其声明为静态方法，它表示类中对象的共有行为
  - 例如，单位换算、数据类型转换、数学运算等功能可以定义成静态方法。
  - 声明时前面需加static修饰符
  - 静态方法可以访问静态属性，但不能访问实例属性
  - 不能被声明为抽象的
  - 类方法可以在不建立对象的情况下用类名直接调用，也可用类实例调用：
    - <类名>.<类方法>
    - <类实例>.<类方法>

类与对象：方法成员

【例】将摄氏温度(centigrade)转换成华氏温度(fahrenheit)。

```
//转换方法centigradeToFahrenheit放在类Converter中
public class Converter {
    public static int centigradeToFahrenheit(int cent)
    {
        return (cent * 9 / 5 + 32);
    }
}
```

- 方法调用：

① Converter.centigradeToFahrenheit(40)

② Converter cToF=new Converter();
cToF .centigradeToFahrenheit(40); // 也可，但不必

类与对象：方法成员

- 对象数组作为方法的参数传递

【例】计算圆数组中所有圆的面积之和

```
public static double sumOfArea(Circle[ ] circles) {
        double s=0;
        for (int i = 0; i < circles.length; i++) {
            s += circles[i].area();
        }
        circles[0].radius=2; //注意这会修改circles数组的对象，因为对象是引用传递的!
        return s;
    }
```

类与对象：方法成员

```
public static void main(String[] args) {
        Circle[] cs=new Circle[3]; //动态创建数组。注意，此时并没有创建Circle对象，仅仅是创建了一个可以存储对象引用的数组！

        // 创建Circle对象，并将对象引用存储到数组
        for(int i=0;i<cs.length;i++)
            cs[i]=new Circle();

        cs[0].radius=1;
        cs[1].radius=8;
        cs[2].radius=5;
        System.out.println( sumOfArea(cs) );
        System.out.println(cs[0].radius);
    }
```

运行结果：

282.7433385000005

2.0

类与对象：方法成员

- 从Java 5开始，可以在方法的参数中使用可变长参数
  - 可变长参数使用省略号表示，其实质是数组
  - 例如，“String … s”表示“String[ ] s”
  - 对于可变长参数的方法，传递给可变长参数的实际参数可以是多个对象，也可以是一个对象或者是没有对象。例如：

```
class A{
    public void func(String…s){
        for(String x: s)  // 也可以使用s.length属性和s[i]
            System.out.println(x);
    }
}
```

  - a.func(); // 可以不传递参数
  - a.func (“Apple”); // 可以传递一个参数
  - a.func(“Apple”,“Banana”,“Orange”); // 可以传递多个参数
  - a.func(names); // 还可以传递一个数组
  - 注意，若是数组作为参数，则参数不能缺失

类与对象：方法成员

【例】使用可变长参数。

```
static double maxArea(Circle c, Rectangle... varRec) {
        double s=c.area();
        Rectangle[ ] rec = varRec;
        for (int i=0;i<rec.length;i++) {  if (rec[i].area()>s)  s=rec[i];    }
        return s;
}
public static void main(String[ ] args) {
        Circle c = new Circle();
        c. radius=10;
        Rectangle r1 = new Rectangle();
        r1.width = 40;      r1.height = 50;
        Rectangle r2 = new Rectangle();
        r2.width = 60;      r2.height = 9;
        System.out.println("maxArea(c, r1, r2)=" + maxArea(c, r1, r2) );
        System.out.println("maxArea(c, r1)=" + maxArea(c, r1) );
        System.out.println("maxArea(c)=" + maxArea(c) );
}
```

运行结果：

```
maxArea(c, r1, r2)=540.0
maxArea(c, r1)=314.159265
maxArea(c)=314.159265
```

类与对象：方法重载

- 一个类中名字相同的多个方法
- 这些方法的参数必须不同，Java可通过参数列表的不同来辨别重载的方法
  - 或者参数个数不同
  - 或者参数类型不同
  - 或者个数和类型都相同，但顺序不同
- 返回值可以相同，也可以不同
- 重载的价值在于它允许通过使用一个方法名来访问多个方法
- Java API类库中大量使用方法重载
  - 构造方法基本上都有重载，例如Color类有7个构造方法
  - 一般方法，例如System.out.println()参数可为各种数据类型

类与对象：方法重载

【例】通过方法重载分别接收一个或几个不同数据类型的数据。

```
class MethodOverloading {
  public void receive(int i){
        System.out.println("Receive one int parameter. ");
        System.out.println("i="+i);
  }
  public void receive(double d){
        System.out.println("Receive one double parameter. ");
        System.out.println("d="+d);
  }
  public void receive(String s){
        System.out.println("Receive one String parameter. ");
        System.out.println("s="+s);
  }
```

类与对象：方法重载

【例】通过方法重载分别接收一个或几个不同数据类型的数据。

```
class MethodOverloading {
   public void receive(int i){ System.out.println(i); }
   public void receive(double d){ System.out.println(d); }
   public void receive(String s){ System.out.println(s); }
   public void receive(int i,int j){
         System.out.println("i=" + i + "  j=" + j);
   }
   public void receive(int i,double d){
         System.out.println("i=" + i + "  d=" + d);
   }
   public void receive(double d,int i){
         System.out.println("d=" + d + "  i=" +i);
   }
}
```

包(Package)

- **包的概念**

计算机操作系统使用文件夹或者目录来存放相关或者同类的文档，在Java编程语言中，提供了一个包的概念来组织相关的类。包在物理上就是一个文件夹，逻辑上代表一个分类概念。

- 是一组类的集合
- 一个包可以包含若干个类文件，还可包含若干个包
- 包的作用
  - 将相关的源代码文件组织在一起
  - 类名的空间管理，利用包来划分名字空间，便可以避免类名冲突
  - 提供包一级的封装及存取权限

包(Package)

- 包的命名
  - 每个包的名称必须是“独一无二”的
  - Java中包名使用小写字母表示
  - 命名方式建议
    - 将机构的Internet域名反序，作为包名的前导，例如：cn.edu.upc.math.class20140812
    - 若包名中有任何不可用于标识符的字符，用下划线替代
    - 若包名中的任何部分与关键字冲突，后缀下划线
    - 若包名中的任何部分以数字或其他不能用作标识符起始的字符开头，前缀下划线

包(Package)

- 编译单元与类空间
  - 一个Java源代码文件称为一个编译单元，由三部分组成
    - 所属包的声明（省略，则属于默认包）
    - import（引入）包的声明，用于导入外部的类
    - 类和接口的声明
  - 一个编译单元中只能有一个public类，该类名与文件名相同，编译单元中的其他类往往是public类的辅助类，经过编译，每个类都会产一个class文件
  - 利用包来划分名字空间，便可以避免类名冲突

包(Package)

- 包的声明
  - 用Package语句声明命名的包（Named Packages）

    package pkg1[.pkg2[.pkg3…]];
  - 例如： package cn.edu.upc;
    - Java源文件的第一条语句，前面只能有注释或空行; 一个文件中最多只能声明一个包
    - 当前文件中声明的所有类都属于包cn.edu.upc
    - 此文件中的每一个类名前都有前缀cn.edu.upc，即实际类名应该是cn.edu.upc.ClassName，因此不同包中的相同类名不会冲突
  - 默认包（未命名的包）

    不含有包声明的编译单元是默认包的一部分

包(Package)

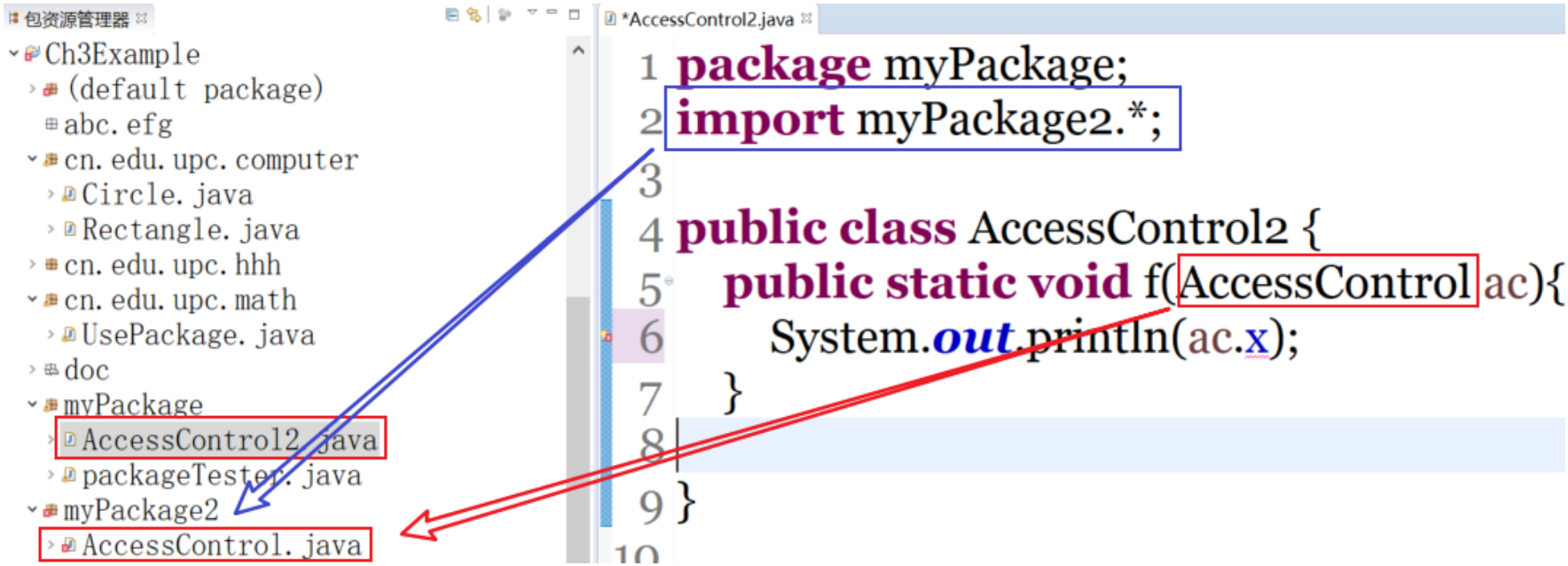
- 引入包
  - 为了使用其它包中所提供的类，需要使用import语句引入所需要的类

```
import package1[.package2…]. (classname |*);
```

  - 其中package1[.package2…]表明包的层次，它对应于文件目录
  - classname则指明所要引入的类名
  - 如果要引入一个包中的所有类，则可以使用星号（*）来代替类名

包(Package)

- 引入包
  - 导入自定义的另一个包中的类



注意：Java 4开始，不允许在另一个包中导入默认包中的类。

包(Package)

- 引入包
  - Java编译器为所有程序自动引入包java.lang
  - 用javac编译源程序时，如遇到当前目录(包)中没有声明的类，就会以环境变量classpath为相对查找路径，按照包名的结构来查找。因此，要指定搜寻包的路径，需设置环境变量classpath

包(Package)

- 静态引入
  - Java 5 新特性
    - 在Java 5之前，通过类名使用类的静态成员。例如，**Math.PI**，**Math.sin(double)**
    - 如果在程序中需要多次使用静态成员，则每次使用都需要加上类名
  - 静态引入分为两种：单一引入和全体引入
    - 单一引入是指引入某一个指定的静态成员，例如：
      import static java.lang.Math.PI;
    - 全体引入是指引入类中所有的静态成员，例如：
      import static java.lang.Math.*;

不推荐

包(Package)

【例】包声明与引入示例：获取指定文件的大小

```
package cn.edu.upc.math; // 声明包
import java.io.*; // 引入包

public class UsePackage {
   public static void main(String[] args) {
         String fName="cn/edu/upc/math/UsePackage.java";
         try {
                File file = new File(fName); // 创建文件对象
                System.out.println(file.length()+"Bytes");
                //System.out.println(file.getAbsolutePath());
         } catch (NullPointerException e) {
                System.out.println(e.getMessage());
         }
   }
}
```

包(Package)

【例】包声明与引入示例：获取指定文件的大小

```
package cn.edu.upc.math; // 声明包
import java.io.*; // 引入包

public class UsePackage {…}
```

注意：在命令行中运行命名包中的程序时，需要在类名前面加上包名，例如：

编译　D:\...\Ch3Example>javac cn/edu/upc/math/UsePackage.java

运行　D:\...\Ch3Example>java cn.edu.upc.math.UsePackage

如果不加包名运行：java UsePackage，则出现如下错误：

错误: 找不到或无法加载主类 UsePackage

类及其成员的访问控制

- 类的访问控制
  - 类的访问控制只有public（公共类）及无修饰符（缺省类）两种
  - 访问权限符与访问能力之间的关系如表

类型	无修饰	public
同一包中的类	yes	yes
不同包中的类	no	yes

类及其成员的访问控制

- 类的访问控制

```
package myPackage2;

public class AccessControl {
    private int x;
    public static void main(String[] args) {
        AccessControl ac=new AccessControl();
        System.out.println(ac.x);
    }
}
```

```
package myPackage;
import myPackage2.*;

public class AccessControl2 {
    public static void f(AccessControl ac){
        System.out.println(ac.x);
    }
}
```

同一包、不同包中的类均可访问

类及其成员的访问控制

● 类的访问控制

```
package myPackage2;

class AccessControl {
   private int x;
   public static void main(String[] args) {
       AccessControl ac=new AccessControl();
       System.out.println(ac.x);
   }
}
```

```
package myPackage;
import myPackage2.*;

public class AccessControl2 {
    public static void f(AccessControl ac){
       System.out.println(ac.x);
    }
}
```

仅限同一包中类可以访问

AccessControl
在此处不可见

类及其成员的访问控制

- 类成员的访问控制
  - 公有(public)
    - 可以被其他任何对象访问(前提是对类成员所在的类有访问权限)
  - 保护(protected)
    - 只可被同一包中的类及其子类（同一包或不同包）的实例对象访问
  - 私有(private)
    - 只能被这个类本身访问，在类外不可见
  - 默认(default)
    - 仅允许同一个包内的访问；又被称为“包(package)访问权限”

类及其成员的访问控制

- 类成员的访问控制

范围 \ 类型	private	无修饰	protected	public
同一类	yes	yes	yes	yes
同一包中的子类	no	yes	yes	yes
同一包中的非子类	no	yes	yes	yes
不同包中的子类	no	no	yes	yes
不同包中的非子类	no	no	no	yes

类及其成员的访问控制

- 类成员的访问控制

```
package myPackage2;

public class AccessControl {
    private int x;
    public static void main(String[] args) {
        AccessControl ac=new AccessControl();
        System.out.println(ac.x);
    }
}

class AccessControl2 {
    public static void f(AccessControl ac){
        System.out.println(ac.x);
    }
}
```

私有的x仅在本类中可以访问

x在此处不可见

类及其成员的访问控制

- 类成员的访问控制

```
package myPackage2;

public class AccessControl {
    private int x;
    public static void main(String[] args) {
        AccessControl ac=new AccessControl();
        System.out.println(ac.x);
    }
}
```

```
package myPackage2;

public class AccessControl2 {
    public static void f(AccessControl ac){
        System.out.println(ac.x);
    }
}
```

私有的x仅在本类中可以访问

x在此处不可见

类及其成员的访问控制

- 类成员的访问控制

```
package myPackage2;

public class AccessControl {
    private int x;
    public static void main(String[] args) {
        AccessControl ac=new AccessControl();
        System.out.println(ac.x);
    }
}
```

```
package myPackage;
import myPackage2.*;

public class AccessControl2 {
    public static void f(AccessControl ac){
        System.out.println(ac.x);
    }
}
```

私有的x仅在本类中可以访问

x在此处不可见

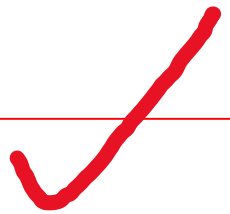
类及其成员的访问控制

- 类成员的访问控制

```
package myPackage2;

public class AccessControl {
    int x;
    public static void main(String[] args) {
        AccessControl ac=new AccessControl();
        System.out.println(ac.x);
    }
}

class AccessControl2 {
    public static void f(AccessControl ac){
        System.out.println(ac.x);
    }
}
```



默认的x限于在同一个包访问

类及其成员的访问控制

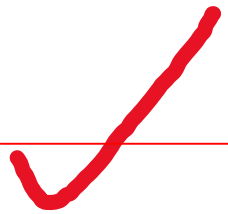
- 类成员的访问控制

```
package myPackage2;

public class AccessControl {
    int x;
    public static void main(String[] args) {
        AccessControl ac=new AccessControl();
        System.out.println(ac.x);
    }
}
```

```
package myPackage2;

public class AccessControl2 {
    public static void f(AccessControl ac){
        System.out.println(ac.x);
    }
}
```



默认的x限于在同一个包访问

类及其成员的访问控制

- 类成员的访问控制

```
package myPackage2;

public class AccessControl {
    int x;
    public static void main(String[] args) {
        AccessControl ac=new AccessControl();
        System.out.println(ac.x);
    }
}
```

```
package myPackage;
import myPackage2.*;

public class AccessControl2 {
    public static void f(AccessControl ac){
        System.out.println(ac.x);
    }
}
```

默认的x限于在同一个包访问

x在此处不可见

类及其成员的访问控制

【例】对Circle类声明进行修改，给实例变量加上private修饰符

```
public class Circle {
        static double PI = 3.14159265;
        private int radius;
        public double circumference() {
            return 2 * PI * radius;
        }
 }
```

编译时会提示出错

在编译语句“c1.radius = 50;”时会提示存在语法错误“radius has private access in Circle”

```
Circle c1 = new Circle();
c1.radius = 50;
```

类及其成员的访问控制

- 类成员的访问控制
  - 如果要允许其它类访问私有数据成员的值，就需要在类中声明相应的公有方法。通常有两类典型的方法用于访问属性值，get方法及set方法
  - 一般具有以下格式：

```
public <fieldType> get<FieldName>() {
    return <fieldName>;
}
public void set<FieldName>(<fieldType> <paramName>) {
    <fieldName> = <paramName>;
}
```

类及其成员的访问控制

- 类成员的访问控制
  - 例如：//声明实例变量radius的get和set方法

```
public double getRadius() {  return radius; }
public void setRadius(double r) {  radius= r; }
```

  - 如果形式参数名与实例变量名相同，则需要在实例变量名之前加this关键字，否则系统会将实例变量当成形式参数。

例如：

```
public void setRadius(double radius) {
        this.radius= radius;
    }
```

Eclipse支持自动生成set、get、toString等方法

对象初始化和回收

- **实例对象初始化**

  系统在生成对象时，会为对象分配内存空间，并自动调用构造方法对实例变量进行初始化

- **对象回收**

  对象不再使用时，系统会调用垃圾回收程序将其占用的内存回收

对象初始化

- 构造方法
  - 一种和类同名的特殊方法
  - 用来初始化对象
  - Java中的每个类都有构造方法，用来初始化该类的一个新的对象
  - 没有定义构造方法的类，系统自动提供默认的构造方法

对象初始化

- 构造方法的特点
  - 方法名与类名相同
  - 没有返回类型，修饰符void也不能有
  - 通常被声明为公有的(public)
  - 可以有任意多个参数
  - 主要作用是完成对象的初始化工作
  - 不能在程序中显式的调用
  - 在生成一个对象时，系统会自动调用该类的构造方法为新生成的对象初始化

对象初始化

- 系统提供的默认构造方法
  - 如果在类的声明中没有声明构造方法，则Java编译器会提供一个默认的构造方法
  - 默认的构造方法没有参数，其方法体为空
  - 使用默认的构造方法初始化对象时，如果在类声明中没有给实例变量赋初值，则对象的属性值为零或空

对象初始化

【例】默认构造方法示例：银行帐号类及测试代码。

```
public class BankAccount{
   String    ownerName;
    int          accountNumber;
    float       balance;
}
public class BankTester{
     public static void main(String args[]){
       BankAccount myAccount = new BankAccount();
       System.out.println("ownerName=" +
                                    myAccount.ownerName);
       System.out.println("accountNumber=" +
                                     myAccount.accountNumber);
      System.out.println("balance=" +
                                     myAccount.balance);
   }
}
```

运行结果：

```
ownerName=null
accountNumber=0
balance=0.0
```

对象初始化

- 自定义构造方法与方法重载
  - 可在生成对象时给构造方法传送初始值，使用希望的值给对象初始化。
  - 构造方法可以被重载，构造方法的重载和方法的重载一致。
  - 一个类中有两个及以上同名的方法，但参数表不同，这种情况就被称为方法重载。在方法调用时，Java可以通过参数列表的不同来辨别应调用哪一个方法。

对象初始化

- 例如：为BankAccount声明一个有3个参数的构造方法

```
public BankAccount(String initName, int initAccountNumber,
   float initBalance) {
       ownerName = initName;
       accountNumber = initAccountNumber;
       balance = initBalance;
 }
```

- 假设一个新帐号的初始余额可以为0，则可增加一个带有2个参数的构造方法

```
public BankAccount(String initName, int initAccountNumber) {
       ownerName = initName;
       accountNumber = initAccountNumber;
       balance = 0.0f;
 }
```

new BankAccount("Tom",105594,200)
new BankAccount("Tom",105578)

对象初始化

- 自定义无参构造方法
  - case 1: 希望给实例变量特定的默认值时
  - case 2: 无参的构造方法对其子类的声明很重要。如果在一个类中不存在无参的构造方法，则要求其子类声明时必须声明构造方法，否则在子类对象的初始化时会出错
  - 关于构造方法，好的声明习惯是
    - 不声明构造方法
    - 如果声明，至少声明一个无参构造方法
  - 给BankAccount类再声明一个无参的构造方法:

```
public BankAccount() {
        ownerName = "";
        accountNumber = 999999;
        balance = 0.0f;
    }
```

对象初始化

- this关键字在构造方法的使用
  - 可以使用this关键字在一个构造方法中调用另外的构造方法
  - 代码更简洁，维护起来也更容易
  - 通常用参数个数比较少的构造方法调用参数个数最多的构造方法

对象初始化

【例】使用this关键字，修改BankAccout类中无参数和二参数的构造方法

```
public BankAccount() {
        this("", 999999, 0.0f);
}
public BankAccount(String initName, int
   initAccountNumber) {
        this(initName, initAccountNumber, 0.0f);
}
public BankAccount(String initName, int
   initAccountNumber,  float initBalance) {
        ownerName = initName;
        accountNumber = initAccountNumber;
        balance = initBalance;
 }
```

内存回收技术

- 当一个对象在程序中不再被使用时，就成为一个无用对象
  - 当前的代码段不属于对象的作用域
  - 把对象的引用赋值为空
- Java运行时系统通过垃圾收集器周期性地释放无用对象所使用的内存
- Java运行时系统会在对对象进行自动垃圾回收前，自动调用对象的finalize()方法

内存回收技术

- 垃圾收集器
  - 自动扫描对象的动态内存区，对不再使用的对象做上标记以进行垃圾回收
  - 作为一个线程运行
    - 通常在系统空闲时异步地执行
    - 当系统的内存用尽或程序中调用System.gc()要求进行垃圾收集时，与系统同步运行

内存回收技术

- finalize()方法
  - 在类java.lang.Object中声明，因此 Java中的每一个类都有该方法
  - 用于释放系统资源，如关闭打开的文件或socket等
  - 声明格式
  - protected void finalize() throws throwable
  - 如果一个类需要释放除内存以外的资源，则需在类中重写finalize()方法

枚举类型

- Java 5的特色，可以取代Java 5之前的版本中使用的常量
- 需要一个有限集合，而且集合中的数据为特定的值时，可以使用枚举类型
- 格式:

```
[public] enum 枚举类型名 [implements 接口名称列表]{
    枚举值;
    变量成员声明及初始化;
    (构造、set/get、一般)方法声明及方法体;
}
```

枚举类型

【例】声明一个表示考试成绩不同等级的枚举类型

```
enum Score { EXCELLENT, QUALIFIED, FAILED; } //Score.java
public class ScoreTester { // ScoreTester.java
    public static void main(String[] args) {
        giveScore(Score.EXCELLENT);
        System.out.println(Score.EXCELLENT);
        System.out.println(Score.EXCELLENT.ordinal()); //返回枚举常量的序数（它在枚举声明中的位置，其中初始常量序数为零）。
    }
    public static void giveScore(Score s){
        switch(s){ // case中枚举元素前不加枚举类型名称
            case EXCELLENT:
                System.out.println("Excellent"); break;
            case QUALIFIED:
                System.out.println("Qualified"); break;
            case FAILED:
                System.out.println("Failed");    break;
        }
    }
}
```

运行结果：

```
Excellent
EXCELLENT
0
```

枚举类型

【例】修改枚举类型Score，添加变量成员、构造方法

```
enum Score {
  // 枚举元素（它们实质上是该枚举类型的final static 对象成员）
   EXCELLENT("优秀", 1),  QUALIFIED ("合格", 2),  FAILED ("未通过", 1);
   // 变量成员
   private String name;
   private int value;
  //构造方法
   private Score(String name, int value){
      this.name=name;
      this.value=value;
      System.out.println("Constructor is called.");
   }
  //get方法
  public String getName(){ return name; }
  public int getValue(){ return value; }
   //覆盖toString方法
  @Override
  public String toString(){ return name+"_"+value;}
}
```

枚举构造方法仅用于初始化各枚举元素（对象），这些元素是枚举的final static成员，构造方法仅需在枚举的内部使用，因此Java规定枚举的构造方法为private或不加修饰符。

枚举类型

```
public class ScoreTester {
    public static void main(String[] args) {
        Score  s1=Score.EXCELLENT;
        System.out.println(s1);
        System.out.println("ordinal=" + s1.ordinal() +
                           ", value=" + s1.getValue());

        Score  s2=Score. QUALIFIED;
        System.out.println(s2);
        System.out.println("ordinal=" + s2.ordinal() +
                           "value=" + s2.getValue());
}
```

运行结果:

```
Constructor is called.
Constructor is called.
Constructor is called.
优秀_1
ordinal=0, value=1
合格_2
ordinal=1, value=2
```

应用举例

- 对银行帐户类BankAccount进行一系列修改和测试
  - 声明BankAccount类
  - 覆盖toString()方法
  - 声明存取款方法
  - 使用DecimalFormat类
  - 声明类变量（即类的静态数据成员）

应用举例

- 声明BankAccount类：包括状态、构造方法、get方法及set方法

```
public class BankAccount{
   private String ownerName;
   private int accountNumber;
   private float balance;

   public BankAccount() {  this("", 0, 0); }
    public BankAccount(String initName, int initAccNum)
    {  this(initName, initAccNum, 0); }

  public BankAccount(String initName, int initAccNum, float initBal)
   {
     ownerName = initName;
     accountNumber = initAccNum;
     balance = initBal;
   }
```

应用举例

- 声明BankAccount类：包括状态、构造方法、get方法及set方法

```
public String getOwnerName()  { return ownerName; }
public int getAccountNumber() { return accountNumber; }
public float getBalance()  { return balance; }

public void setOwnerName(String newName) {
    ownerName = newName;
}
public void setAccountNumber(int newNum) {
    accountNumber = newNum;
}
public void setBalance(float newBalance) {
    balance = newBalance;
}
}
```

应用举例

- 声明测试类AccountTester

```
public class AccountTester {
    public static void main(String args[]) {
        BankAccount    anAccount;
        anAccount = new BankAccount("ZhangLi", 100023,0);
        anAccount.setBalance(anAccount.getBalance() + 100);
        System.out.println("Here is the account: " + anAccount);
        System.out.println("Account name: "+
                                    anAccount.getOwnerName());
        System.out.println("Account number: "+
                                   anAccount.getAccountNumber());
        System.out.println("Balance: $" + anAccount.getBalance());
    }
}
```

运行结果:

```
Here is the account: BankAccount@1db9742
Account name: Obama
Account number: 807169
Balance: $100.0
```

应用举例

- 覆盖toString()方法
  - 将对象的内容转换为字符串
  - Java的所有类都有一个默认的toString()方法，其方法体如下：

```
getClass().getName() + '@' +
    Integer.toHexString(hashCode())
```

  - 下面的两行代码等价

```
System.out.println(anAccount);
System.out.println(anAccount.toString());
```

  - 如果需要特殊的转换功能，则需要自己重写toString()方法

应用举例

- 覆盖toString()方法
  - 方法原型必须是： public String toString()
  - 为BankAccount类添加自己的toString()方法

```
public String toString() {
   return("Account #" + accountNumber + " with balance $" +
    balance);
}
```

运行结果：

```
Here is the account: Account #807169 with balance $100.0
Account name: Obama
Account number: 807169
Balance: $100.0
```

应用举例

- 给BankAccount类增加存款及取款方法

```
//存款
public float deposit(float anAmount) {
   balance += anAmount;
   return balance;  // 返回余额
 }

// 取款
public float withdraw(float anAmount) {
   balance -= anAmount;
   return balance;
}
```

应用举例

- 在AccountTester类中增加测试代码

```
anAccount= new BankAccount("Putin", 807170,200);
System.out.println(anAccount);
anAccount.deposit(500.49f);
System.out.println(anAccount);
anAccount.withdraw(300.07f);
System.out.println(anAccount);
```

运行结果:

....

Account #807170 with balance $200.0

Account #807170 with balance $700.49

Account #807170 with balance $400.41998

应用举例

- 使用DecimalFormat类格式化数值
  - 在java.text包中
  - 其实例方法format用于对数据进行格式化
  - 修改后的toString()方法如下

```
public String toString() {
    return("Account #" + accountNumber + " with balance " +
     new java.text.DecimalFormat("$0.00").format(balance));
}
```

试一试：String.format("$%1$.2f", balance)

运行结果：....

Account #807170 with balance $200.00
Account #807170 with balance $700.49
Account #807170 with balance $400.42

应用举例

- 修改BankAccount类：声明类变量（即类的静态变量）
  - 目的：实现银行账号的自动维护
  - 增加类变量LAST_ACCOUNT_NUMBER，初始值为0，当生成一个新的BankAccount对象时，其帐号为LAST_ACCOUNT_NUMBER的值累加1
  - 自动产生对象的accountNumber，且不允许直接修改其值
  - 修改构造方法，取消帐号参数
  - 取消setAccountNumber方法
  - 取消setBalance方法

应用举例

- 修改BankAccount类 （BankAccount2.java）

```
public class BankAccount2 {
  private static int LAST_ACCOUNT_NUMBER = 0;
  private int accountNumber;
  private String ownerName;
  private float balance;
  public BankAccount2() {  this("", 0);  }
  public BankAccount2(String initName) { this(initName, 0); }
  public BankAccount2(String initName, float initBal) {
    ownerName = initName;
    accountNumber = ++LAST_ACCOUNT_NUMBER;
    balance = initBal;
  }
```

应用举例

- 修改BankAccount类 （BankAccount2.java）

```
public int getAccountNumber() {
    return accountNumber;
  }
  public String getOwnerName() {
    return ownerName;
  }
  public float getBalance() {
    return balance;
  }
  public void setOwnerName(String aName) {
    ownerName = aName;
  }
```

应用举例

```
public String toString() {    return("Account #"+  new
  java.text.DecimalFormat("000000").format(accountNumber) +  "
  with balance " +  String.format("%1$.2f", balance));
 }
```

试一试：String.format("$%1$06 d", accountNumber)

```
 public float deposit(float anAmount) {
     balance += anAmount;
     return balance;
 }
 public boolean withdraw(float anAmount) {
          if(anAmount<=balance){
                    balance -= anAmount;
                    return true;
          }
          else // 余额不足
                    return false;
   }
}
```

应用举例

- 修改BankTester类 （BankTester2.java）

```
public class BankTester2 {
   public static void main(String args[]) {
      BankAccount2 anAccount= new BankAccount2("Obama", 100);
       System.out.println(anAccount);

       anAccount= new BankAccount2("Putin", 200);
       System.out.println(anAccount);
       anAccount.deposit(500.49f);
       System.out.println(anAccount);
       boolean optFlag=anAccount.withdraw(300.07f);
       System.out.println(optFlag? "Withdraw successfully" :
                                    "Insufficient balance");
       System.out.println(anAccount);
       optFlag=anAccount.withdraw(850.0f);
       System.out.println(optFlag? "Withdraw successfully" :
                                    "Insufficient balance");
       System.out.println(anAccount);
   }
}
```

```
BankAccount2 anAccount= new BankAccount2("Obama", 100);
System.out.println(anAccount);

anAccount= new BankAccount2("Putin", 200);
System.out.println(anAccount);
anAccount.deposit(500.49f);
System.out.println(anAccount);
boolean optFlag=anAccount.withdraw(300.07f);
System.out.println(optFlag? "Withdraw successfully" : "Insufficient balance");
System.out.println(anAccount);
optFlag=anAccount.withdraw(850.0f);
System.out.println(optFlag? "Withdraw successfully" : "Insufficient balance");
System.out.println(anAccount);
```

运行结果：....

```
Account #000001 with balance $100.00
Account #000002 with balance $200.00
Account #000002 with balance $700.49
Withdraw successfully
Account #000002 with balance $400.42
Insufficient balance
Account #000002 with balance $400.42
```

习 题

1. 参照BankAccount2类声明一个Student类及其测试类StudentTester。

   ① Student类数据成员包括学号、姓名、性别、成绩。学号（从2209050101开始）自动维护且不许外部修改，设计适当的构造方法、数据成员的set/get方法，重写toString方法用于输出学生信息。

   ② 在StudentTester类中创建一个Student类型的数组，并引用动态创建的若干个Student实例。对学生信息进行设置修改和访问输出。

   ③ 在StudentTester类中定义一个静态方法printStudents，以表格形式输出学生数组。创建好数组或修改元素后可以调用该方法输出数组，以观察数据变化。

   ④ 这2个类在包cn.edu.upc.sci中声明。

习 题

备注：

（1）性别可以定义一个枚举类型，例如：

```
enum Gender{
        Male, //默认name属性为Male,序号属性ordinal为0
        Female//默认name属性为Female,序号属性ordinal为1
}
```

然后，可以这样定义变量：

```
private sex Gender; // 属性：性别，枚举类型
```

在代码中可以直接使用Gender.Male和Gender.Female

（2）测试数据直接使用字面量，不必从键盘输入，例如：

```
new Student("张三",Gender.Male,98)
```

（3）请同学们在代码中适当写一些注释，特别是类的属性和方法的简要说明。

谢谢大家!