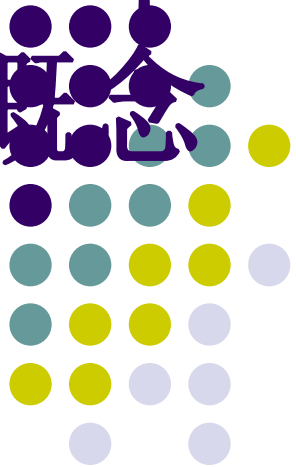
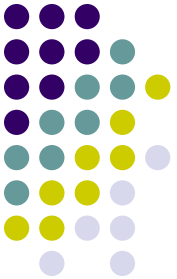
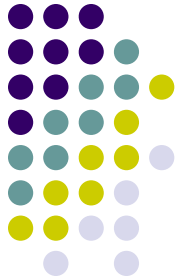
JAVA语言程序设计

第二章
类与对象的基本概念



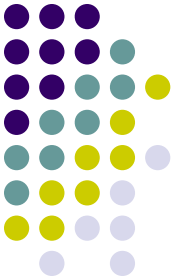


目录



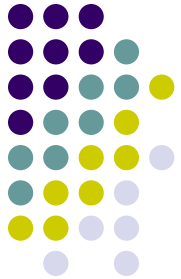
2.1 面向对象的程序设计方法概述

- 面向对象的程序设计
  - 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



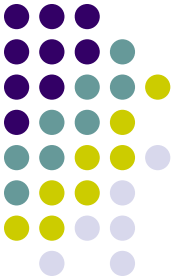
2.1 面向对象的程序设计方法概述(续)

- 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



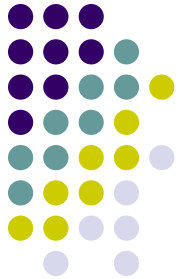
2.1 面向对象的程序设计方法概述(续)

- 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，



2.1 面向对象的程序设计方法概述(续)

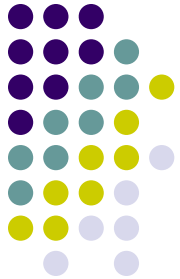
- 结构化程序设计
  - 通常由若干个程序模块组成，每个程序模块都可以是子程序或函数
  - 数据和功能分离，代码难于维护和复用
- 面向对象程序设计
  - 基本组成单位是类
  - 程序在运行时由类生成对象，对象是面向对象程序的核心
  - 对象之间通过发送消息进行通信，互相协作完成相应功能



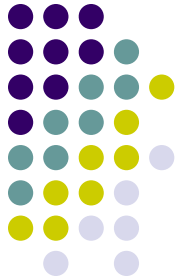
2.1.1 抽象

- 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
  - 计算机软件开发中所使用的抽象有
    - 过程抽象
    - 数据抽象

2.1.1 抽象(续)
——过程抽象



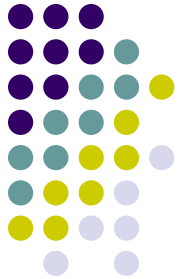
- 过程抽象
  - 将整个系统的功能划分为若干部分，强调功能完成的过程和步骤，而隐藏其具体的实现
  - 任何一个明确定义的功能操作都可被看作单个的实体，尽管这个操作实际上可能由一系列更低级的操作来完成
  - 基于过程抽象的两个标准程序设计技术
    - 过程分解
    - 递归技术



2.1.1 抽象(续)

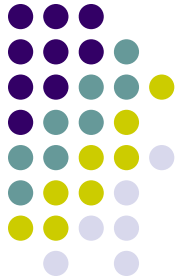
——数据抽象

- 数据抽象
  - 将需要处理的数据和这些数据上的操作结合在一起，抽象成不同的抽象数据类型
  - 每个抽象数据类型既包含了数据，也包含了针对这些数据的操作
  - 相对于过程抽象，数据抽象是更为合理的抽象方法



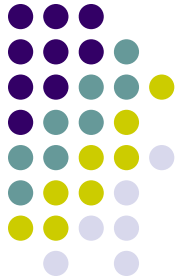
2.1.1 抽象(续)

- 例：
  - 钟表
    - 数据(属性)
      - int Hour; int Minute; int Second;
    - 方法(行为)
      - SetTime(); ShowTime();



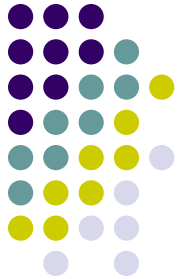
2.1.1 抽象(续)

- 例：
  - 人
    - 数据(属性)
      - char *name; char *gender; int age; int id;
    - 方法(行为)
      - 生物行为
        Eat(), Step(),…
      - 社会行为
        Work(), Study(),…



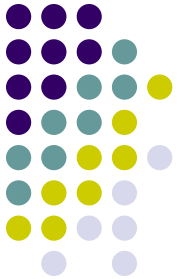
2.1.2 封装

- 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



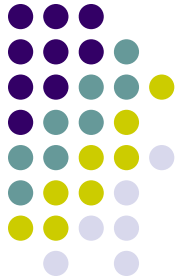
2.1.2 封装(续)

- 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。



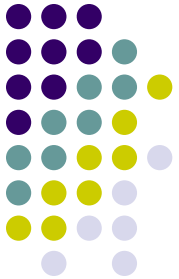
2.1.2 封装(续)

- 封装的意义
  - 在面向对象的程序设计中，类封装了数据及对数据的操作，是程序中的最小模块
  - 禁止了外界直接操作类中的数据，模块与模块之间只能通过严格控制的接口进行交互，这使得模块之间的偶合度大大降低
  - 保证了模块具有较好的独立性，程序维护和修改较为容易



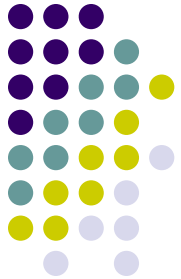
2.1.3 继承

- 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



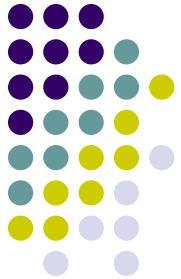
2.1.3 继承(续)

- 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



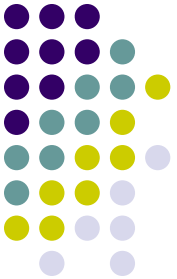
2.1.4 多态

- 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(*子类*)可以响应同名(*父类*)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



2.2 类与对象

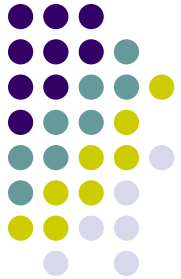
- 类与对象
  - 在程序中，对象是通过一种抽象数据类型来描述的，这种抽象数据类型称为类(Class)
  - 一个类是对一类对象的描述。类是构造对象的模板
  - 对象是类的具体实例



2.2.1 类的声明

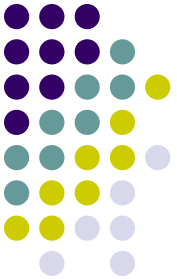
- 声明形式

```
[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;
}
```



2.2.1 类的声明(续)

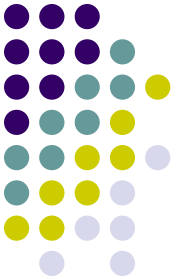
- 关键字
  - class
    - 表明其后声明的是一个类。
  - extends
    - 如果所声明的类是从某一父类派生而来，那么，父类的名字应写在extends之后
  - implements
    - 如果所声明的类要实现某些接口，那么，接口的名字应写在implements之后



2.2.1 类的声明(续)

- 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
  - 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

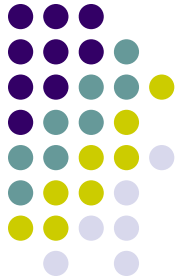
2.2.1 类的声明(续)
——例2_1



- 钟表类

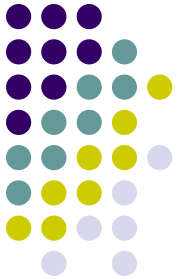
```
public class  Clock
{  // 成员变量
    int hour ;
    int minute ;
    int second ;

   // 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);
   }
}
```



2.2.2 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



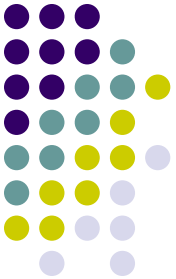
2.2.2 对象的声明与引用(续)

- 对象的声明
  - 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



2.2.2 对象的声明与引用(续)

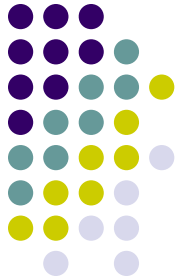
- 对象的创建
  - 生成实例的格式：

    new <类名>()

    例如： aclock=new Clock()

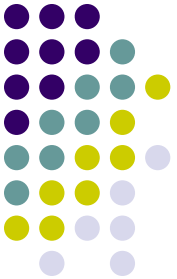
    其作用是：
    - 在内存中为此对象分配内存空间
    - 返回对象的引用(reference，相当于对象的存储地址)
  - 引用变量可以被赋以空值

    例如：aclock=null;



2.2.3 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)
  - 分为实例变量和类变量

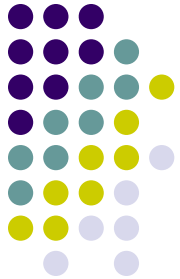


2.2.3 数据成员(续)

- 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], … ;

- 格式说明
  - 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



补充说明：

transient和volatile两个关键字一个用于对象序列化，一个用于线程同步 。

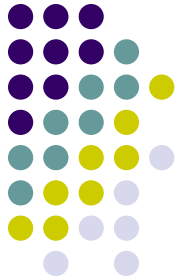
transient是类型修饰符，只能用来修饰字段。在对象序列化的过程中，标记为transient的变量不会被序列化

volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

使用建议：在两个或者更多的线程访问的成员变量上使用volatile。当要访问的变量已在synchronized代码块中，或者为常量时，不必使用。

2.2.3 数据成员(续)

——实例变量

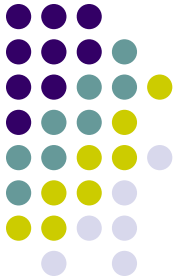


- 实例变量
  - 没有static修饰的变量称为实例变量(Instance Variables)
  - 用来存储所有实例所需要的属性信息，不同实例的属性值可能会不同
  - 可通过下面的表达式访问实例属性的值

    <实例名>.<实例变量名>

2.2.3 数据成员(续)

——例2_2



- 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 {
        int 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);
    }
}
```

2.2.3 数据成员(续)

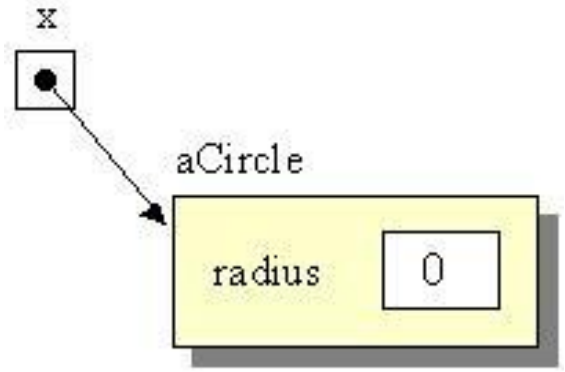
——例2_2运行结果

- 编译后运行结果如下：

  Circle@26b249

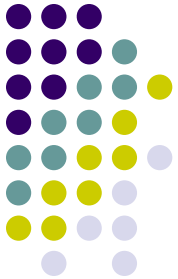
  radius =0

- 解释
  - @之后的数值为x所指的对象的存储地址
  - x的值及对象的状态如图



2.2.3 数据成员(续)

——例2_3



- 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 + "   " + y);
            }
 }
```

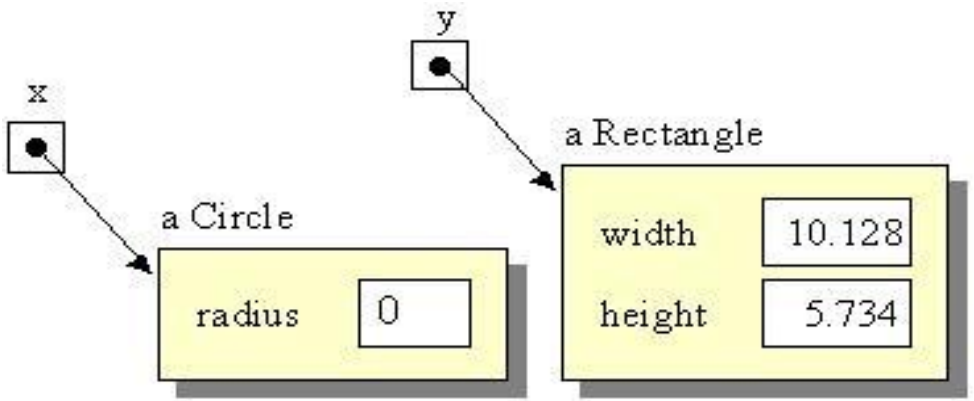
2.2.3 数据成员(续)

——例2_3运行结果

- 编译后运行结果如下：

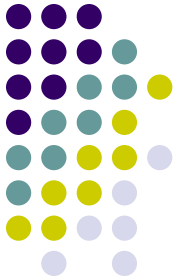
  Circle@82f0db　Rectangle@92d342

- 解释
  - Circle及Rectangle类对象的状态如图



2.2.3 数据成员(续)

——例2_3修改



- 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);
   }
}
```

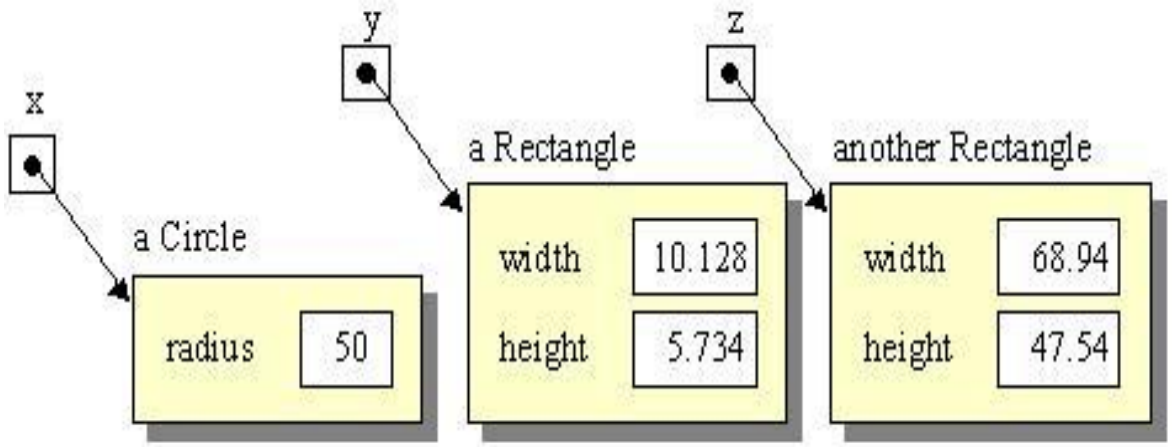
2.2.3 数据成员(续)

——修改后运行结果

- 编译后运行结果如下：

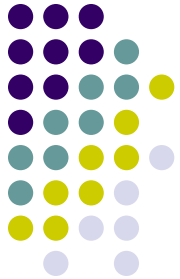
  50 10.128 68.94

- 解释
  - Circle及Rectangle类对象的状态如图



2.2.3 数据成员(续)

——例2_4

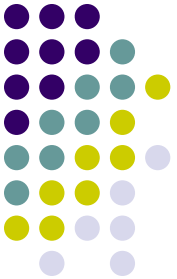


- 地址簿程序
  - 一个人的地址通常包括以下信息：

    姓名，省份，城市，街道，门牌号，邮政编码
  - 采用过程化的程序设计方法，使用简单变量存储，则存储两个人地址的代码如下

2.2.3 数据成员(续)

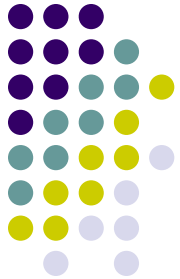
——例2_4



```
public static void main(String args[]) {
   String name1,name2;
   int   gateNumber1, gateNumber2;
   String streetName1,streetName2;
   String city1,city2;
   String province1,province2;
   String postalCode1,postalCode2;
   name1 = " Zhang Li";
   gateNumber1 = 15;
   streetName1 = " Tsinghua East  Road";
   city1 = " Beijing";
   province1 = “Beijing";
   postalCode1 = " 100084";
   name2 = " Li Hong";
   gateNumber2 = 2;
   streetName2 = " BeiNong";
   city2 = " Beijing";
   province2 = " Beijing";
   postalCode2 = " 102206";
  //...do something interesting
}
```

2.2.3 数据成员(续)

——例2_4

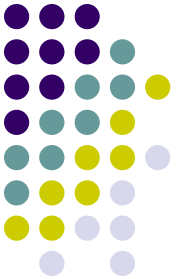


- 采用面向对象的程序设计方法，则需要首先声明Address类如下

```
public class Address {
   String name;
   int gateNumber;
   String streetName;
   String city;
   String province;
   String postalCode;
   //方法成员略
}
```

2.2.3 数据成员(续)

——例2_4

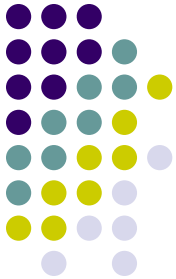


- 主方法改写如下

```
public static void main(String args[]) {
  Address address1 = new Address(), address2 = new ddress();
  address1.name = “Zhang Li";
  address1.streetNumber = 15;
  address1.streetName = “Tsinghua East Road";
  address1.city = “Beijing";
  address1.province = “Beijing";
  address1.postalCode = “100084";
  address2.name = “Li Hong";
  address2.streetNumber = 2;
  address2.streetName = “BeiNong";
  address2.city = “Beijing";
  address2.province = “Beijing";
  address2.postalCode = “102206";
  //...do something interesting
}
```

2.2.3 数据成员(续)

——类变量

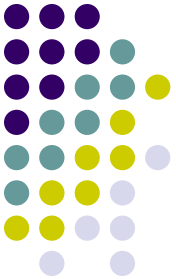


- 类变量
  - 也称为静态变量，声明时需加static修饰符
  - 不管类的对象有多少，类变量只存在一份，在整个类中只有一个值
  - 类初始化的同时就被赋值
  - 适用情况
    - 类中所有对象都相同的属性
    - 经常需要共享的数据
    - 系统中用到的一些常量值
  - 引用格式

    <类名 | 实例名>.<类变量名>

2.2.3 数据成员(续)

——例2_5



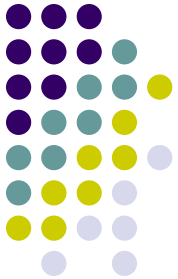
- 对于一个圆类的所有对象，计算圆的面积时，都需用到π的值，可在Circle类的声明中增加一个类属性PI

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

当我们生成Circle类的实例时，在每一个实例中并没有存储PI的值，PI的值存储在类中

2.2.3 数据成员(续)

——例2_5运行结果



- 对类变量进行测试

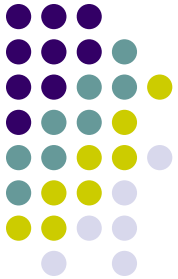
```
public class ClassVariableTester {
   public static void main(String args[]) {
     Circle x = new Circle();
     System.out.println(x.PI);
     System.out.println(Circle.PI);
     Circle.PI = 3.14;
     System.out.println(x.PI);
     System.out.println(Circle.PI);
   }
}
```

- 测试结果

```
3.14159265
3.14159265
3.14
3.14
```

2.2.3 数据成员(续)

——Point.java

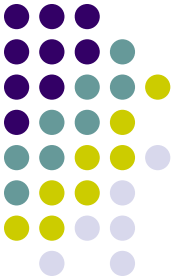


- 声明一个Point类，有两个私有变量保存点坐标，一个类变量保存已有点的个数

```
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++;}
}
```

2.2.3 数据成员(续)

——Point类测试

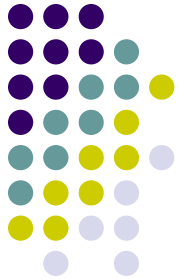


- 测试类ex2_4

```
class ex2_4
{
    public static void main(String[] args)
   {
        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);
    }
}
```

2.2.3 数据成员(续)

——Point类测试结果



- 测试结果

  1

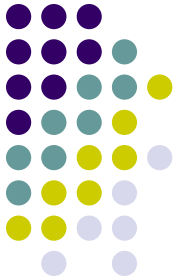
  2

  true

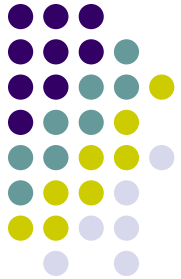
  2

2.2.3 数据成员(续)

——final修饰符



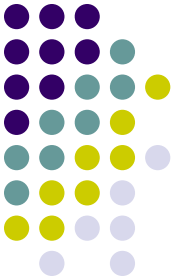
- 实例变量和类变量都可被声明为final
- final实例变量必须在每个构造方法结束之前赋初值，以保证使用之前会被初始化
- final类变量必须在声明的同时初始化



2.2.4 方法成员

- 方法成员(成员方法)
  - 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
  - 分为实例方法和类方法

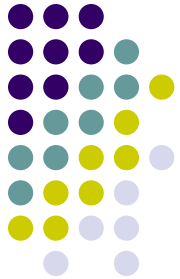
2.2.4 方法成员(续)
——声明格式



- 声明格式

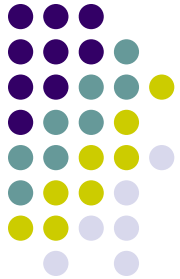
```
[public | protected | private]
[static][ final][abstract] [native] [synchronized]
返回类型  方法名([参数列表]) [throws exceptionList]
{
        方法体
}
```

2.2.4 方法成员(续)
——格式说明



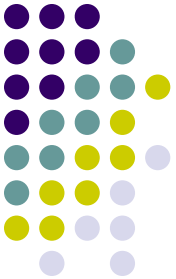
- 格式说明
  - 方法修饰
    - public、protected、private 为存取控制符
    - static指明方法是一个类方法
    - final指明方法是一个终结方法
    - abstract指明方法是一个抽象方法
    - native用来集成java代码和其它语言的代码
    - synchronized用来控制多个并发线程对共享数据的访问

2.2.4 方法成员(续)
——格式说明



- 格式说明(续)
  - 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
  - throws exceptionList
    - 用来处理异常

2.2.4 方法成员(续)
——方法调用

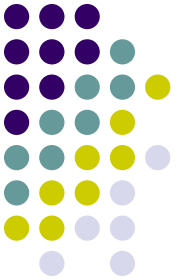


- 方法调用
  - 给对象发消息意味着调用对象的某个方法
    - 从对象中取得信息
    - 修改对象的状态或进行某种操作
    - 进行计算及取得结果等
  - 调用格式

    <对象名>.<方法名>（［参数列表］）

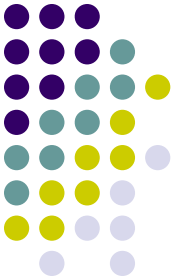
    称点操作符“.”前面的<对象名>为消息的接收者(receiver)
  - 参数传递
    - 值传递：参数类型为基本数据类型时
    - 引用传递：参数类型为对象类型或数组时

2.2.4 方法成员(续)
——实例方法



- 实例方法
  - 表示特定对象的行为
  - 声明时前面不加static修饰符
  - 使用时需要发送给一个类实例

2.2.4 方法成员(续)
——例2_6



- 在Circle类中声明计算周长的方法

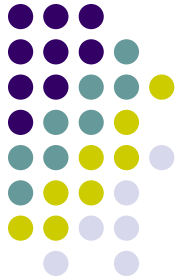
```
public class Circle {
    static double PI = 3.14159265;
    int radius;
      public double circumference() {
        return 2 * PI * radius;
    }
}
```

  - 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

2.2.4 方法成员(续)
——例2_6

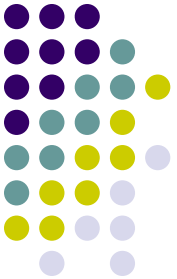


- 方法调用测试

```
public class CircumferenceTester {
    public static void main(String args[]) {
        Circle c1 = new Circle();
        c1.radius = 50;
        Circle c2 = new Circle();
        c2.radius = 10;
         double circum1 = c1.circumference();
        double circum2 = c2.circumference();
        System.out.println("Circle 1 has circumference " + circum1);
        System.out.println("Circle 2 has circumference " + circum2);
    }
}
```

2.2.4 方法成员(续)

——例2_6运行结果



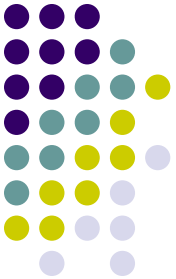
- 运行结果

  Circle 1 has circumference 314.159265

  Circle 2 has circumference 62.831853

- 说明
  - 在使用实例方法时，需要将其发送给一个实例对象（也称给对象发送一条消息），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

2.2.4 方法成员(续)
——例2_7



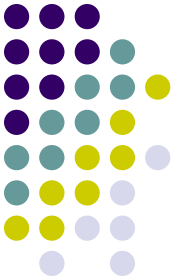
- 在Circle类及Rectangle类中声明计算面积的方法area()

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

2.2.4 方法成员(续)

——例2_7

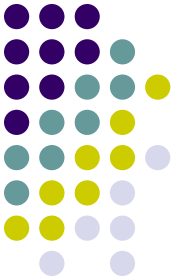


- 声明测试类，对Circle类及Rectangle类的area()方法进行测试

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

2.2.4 方法成员(续)

——例2_7运行结果



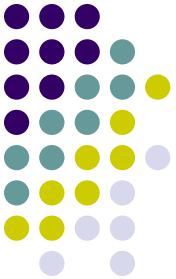
- 运行结果

  Circle has area 7853.981625

  Rectangle has area 600.0

- 说明
  - 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

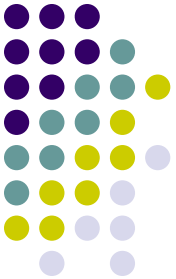
2.2.4 方法成员(续)
——例2_8



- 带参数的方法举例：在Circle类中增加方法改变圆的半径

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

2.2.4 方法成员(续)
——例2_8

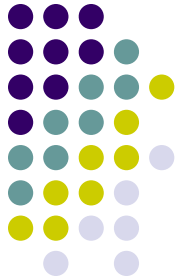


- 测试类

```
public class EnlargeTester {
   public static void main(String args[]) {
      Circle c1 = new Circle();
      c1.radius = 50;
      Circle c2 = new Circle();
      c2.radius = 10;
      System.out.println("Circle 1 的周长: " + c1.circumference());
      System.out.println("Circle 2 的周长: " + c2.circumference());
      c2.enlarge(4);
      System.out.println("Circle 2 扩大后的周长: " + c2.circumference());
    }
}
```

2.2.4 方法成员(续)

——例2_8运行结果



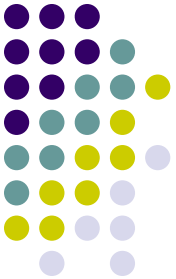
- 运行结果

  Circle 1 的周长: 314.159265

  Circle 2 的周长: 62.831853

  Circle 2 扩大后的周长: 251.327412

2.2.4 方法成员(续)
——例2_9

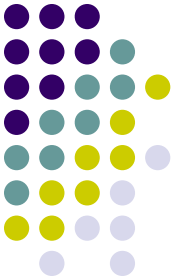


- 以对象作为参数的方法举例：在Circle类中增加fitsInside方法判断一个圆是否在一个长方形内，需要以Rectangle类的对象作为参数

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

2.2.4 方法成员(续)

——例2_9

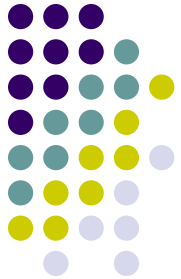


测试类

```
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));
  }
}
```

2.2.4 方法成员(续)

——例2_9运行结果

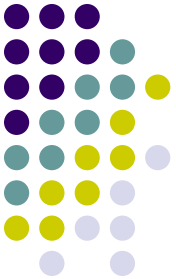


- 运行结果

  Circle 1 fits inside Rectangle: true

  Circle 2 fits inside Rectangle: false

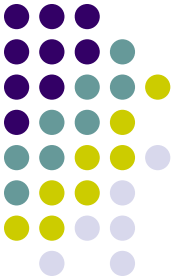
2.2.4 方法成员(续)
——类方法



- 类方法
  - 也称为静态方法，表示类中对象的共有行为
  - 声明时前面需加static修饰符
  - 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

2.2.4 方法成员(续)

——例2_10



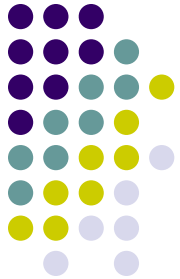
- 将摄氏温度(centigrade)转换成华氏温度(fahrenheit)
  - 转换公式为 fahrenheit = centigrade * 9 / 5 + 32
  - 除了摄氏温度值及公式中需要的常量值，此功能不依赖于具体的类实例的属性值，因此可声明为类方法
  - 转换方法centigradeToFahrenheit放在类Converter中

```
public class Converter {
  public static int centigradeToFahrenheit(int cent)
  { return (cent * 9 / 5 + 32);
  }
 }
```

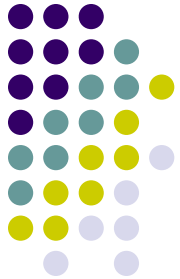
  - 方法调用

```
Converter.centigradeToFahrenheit(40)
```

2.2.5 类的组织——包的概念

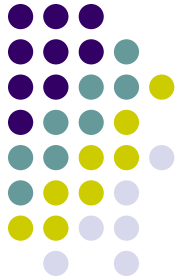


- 包
  - 是一组类的集合
    - 一个包可以包含若干个类文件，还可包含若干个包
  - 包的作用
    - 将相关的源代码文件组织在一起
    - 类名的空间管理，利用包来划分名字空间，便可以避免类名冲突
    - 提供包一级的封装及存取权限



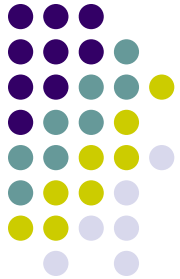
2.2.5 类的组织——包的概念

- 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
    - 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



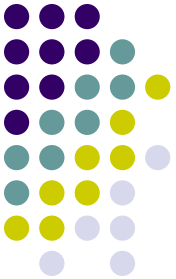
2.2.5 类的组织——包的概念(续)

- 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



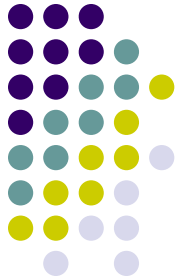
2.2.5 类的组织——包的概念(续)

- 包的声明
  - **命名的包（Named Packages）**
    - 例如： package Mypackage;
  - **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



2.2.5 类的组织——包的概念(续)

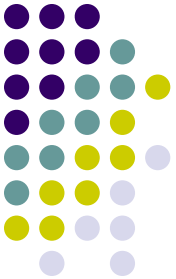
- 包与目录
  - Java使用文件系统来存储包和类
  - 包名就是文件夹名，即目录名
  - 目录名并不一定是包名
  - 用javac编译源程序时，如遇到当前目录(包)中没有声明的类，就会以环境变量classpath为相对查找路径，按照包名的结构来查找。因此，要指定搜寻包的路径，需设置环境变量classpath



2.2.5 类的组织——包的概念(续)

- 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
  - Java编译器为所有程序自动引入包java.lang
  - import语句的格式

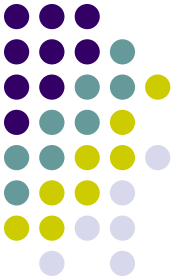
    import package1[.package2…]. (classname |*);
    - 其中package1[.package2…]表明包的层次，它对应于文件目录
    - classname则指明所要引入的类名
    - 如果要引入一个包中的所有类，则可以使用星号（*）来代替类名



2.2.6 类的访问控制

- 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

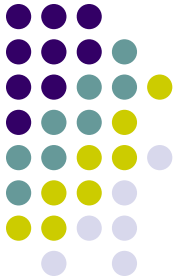
类型	无修饰	public
同一包中的类	yes	yes
不同包中的类	no	yes



2.2.6 类成员的访问控制

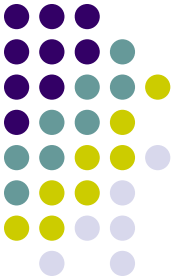
- 类成员的访问控制
  - 公有(public)
    - 可以被其他任何对象访问(前提是对类成员所在的类有访问权限)
  - 保护(protected)
    - 只可被同一类及其子类的实例对象访问
  - 私有(private)
    - 只能被这个类本身访问，在类外不可见
  - 默认(default)
    - 仅允许同一个包内的访问；又被称为“包（package)访问权限”

2.2.6 类成员的访问控制(续)



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

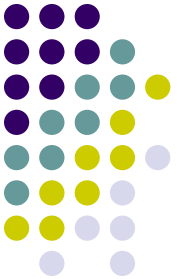
2.2.6 类成员的访问控制(续)
——例2_11



- 对例2-6中的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;
        }
 }
```

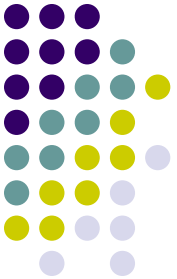
2.2.6 类成员的访问控制(续)
——例2_11



- 再编译CircumferenceTester.java

```
public class CircumferenceTester {
    public static void main(String args[]) {
        Circle c1 = new Circle();
        c1.radius = 50;
        Circle c2 = new Circle();
        c2.radius = 10;
         double circum1 = c1.circumference();
        double circum2 = c2.circumference();
        System.out.println("Circle 1 has circumference " + circum1);
        System.out.println("Circle 2 has circumference " + circum2);
    }
}
```

2.2.6 类成员的访问控制(续)
——例2_11编译



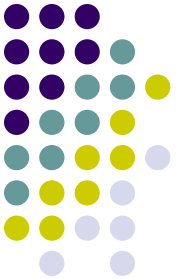
- 编译时会提示出错

  在编译语句“c1.radius = 50;”及“c2.radius =10;”时会提示存在语法错误

  “radius has private access in Circle”

- 说明
  - 由于在Circle类声明中变量radius被声明为private，因此在其它类中不能直接对radius进行存取
  - 如果要允许其它类访问radius的值，就需要在Circle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

2.2.6 类成员的访问控制(续)
——get方法



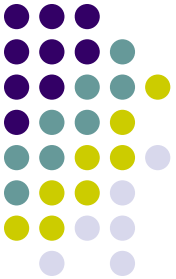
- get方法
  - 功能是取得属性变量的值
  - get方法名以“get”开头，后面是实例变量的名字
  - 一般具有以下格式：

```
public <fieldType> get<FieldName>() {
    return <fieldName>;
}
```

  - 对于实例变量radius，声明其get方法如下：

```
public int getRadius(){
   return radius;
}
```

2.2.6 类成员的访问控制(续)
——set方法



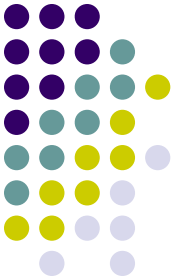
- set方法
  - 功能是修改属性变量的值
  - set方法名以“set”开头，后面是实例变量的名字
  - 一般具有以下格式

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

  - 声明实例变量radius的set方法如下：

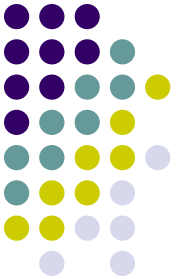
```
public void setRadius(int r){
   radius = r;
}
```

2.2.6 类成员的访问控制(续)
——set方法



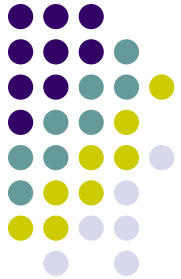
- 关键字this的使用
  - 如果形式参数名与实例变量名相同，则需要在实例变量名之前加this关键字，否则系统会将实例变量当成形式参数。
  - 在上面的set方法中，如果形式参数为radius，则需要在成员变量radius之前加上关键字this。代码如下：

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



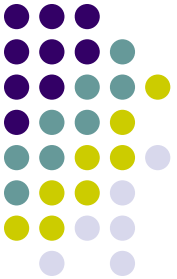
2.3 对象初始化和回收

- 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



2.3.1 构造方法

- 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

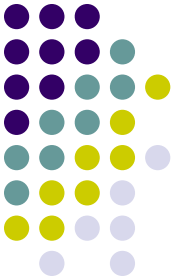


2.3.1 构造方法(续)

- 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

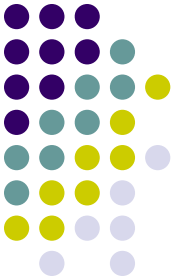
2.3.1 构造方法(续)

——默认构造方法



- 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

2.3.1 构造方法(续)
——例2_12（使用默认构造方法）

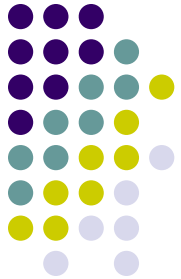


- 声明一个银行帐号类及测试代码

```
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);
    }
}
```

2.3.1 构造方法(续)

——例2_12运行结果



- 运行结果

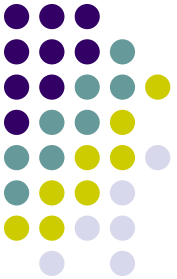
  ownerName=null

  accountNumber=0

  balance=0.0

2.3.1 构造方法(续)

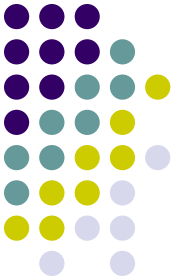
——自定义构造方法



- 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

2.3.1 构造方法(续)

——例2_13



- 为BankAccount声明一个有三个参数的构造方法

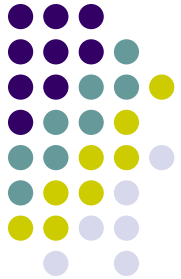
```
public BankAccount(String initName, int initAccountNumber, float
    initBalance) {
        ownerName = initName;
        accountNumber = initAccountNumber;
        balance = initBalance;
 }
```

- 假设一个新帐号的初始余额可以为0，则可增加一个带有两个参数的构造方法

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

2.3.1 构造方法(续)

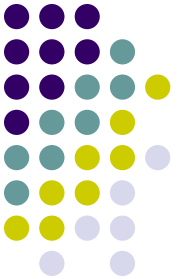
——自定义无参构造方法



- 自定义无参的构造方法
  - 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
  - 在声明构造方法时，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

2.3.1 构造方法(续)

——Bush.java



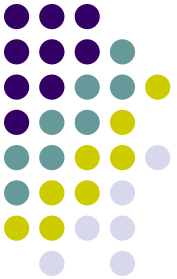
- 构建一个Bush类，有两个有参数的构造方法

```
class Bush {
    Bush(int i) {}
     Bush(double d) {}
 }
```

  - 如果写：new Bush();
    编译器将要告诉你找不到对应的构造方法
- 说明
  - 用户在进行类声明时，如果没有声明任何构造方法，系统会赋给此类一个默认（无参）的构造方法。但是，只要用户声明了构造方法，即使没有声明无参的构造方法，系统也不再赋默认的构造方法

2.3.1 构造方法(续)

——例2_14

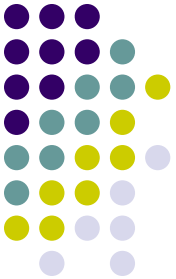


- 在例2_13基础上再声明一个无参的构造方法

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

2.3.1 构造方法(续)

——Tree.java

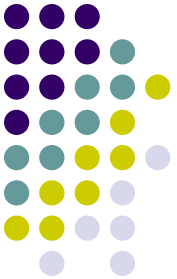


- 创建一个拥有两个构造方法的Tree类，一个有参，一个无参。

```
import java.util.*;
class Tree
{
  int height;
  Tree()   { prt("Planting a seedling"); height = 0;  }
  Tree(int i) {
       prt("Creating new Tree that is "+ i + " feet tall");
      height = i;
  }
  void info()   { prt("Tree is " + height + " feet tall");  }
  void info(String s)  { prt(s + ": Tree is " + height + " feet tall"); }
  static void prt(String s)  { System.out.println(s);   }
}
```

2.3.1 构造方法(续)

——Overloading.java

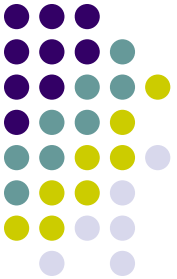


- 测试Tree类

```
public class Overloading
{
  public static void main(String[] args)
  {
    for(int i = 0; i < 5; i++)
    {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    new Tree();
  }
}
```

2.3.1 构造方法(续)

——Overloading测试结果

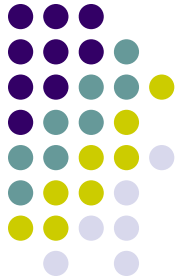


- 测试结果

Creating new Tree that is 0 feet tall
Tree is 0 feet tall
overloaded method: Tree is 0 feet tall
Creating new Tree that is 1 feet tall
Tree is 1 feet tall
overloaded method: Tree is 1 feet tall
Creating new Tree that is 2 feet tall
Tree is 2 feet tall
overloaded method: Tree is 2 feet tall
Creating new Tree that is 3 feet tall
Tree is 3 feet tall
overloaded method: Tree is 3 feet tall
Creating new Tree that is 4 feet tall
Tree is 4 feet tall
overloaded method: Tree is 4 feet tall
Planting a seedling

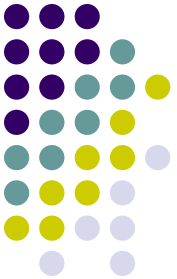
2.3.1 构造方法(续)

——this关键字的使用



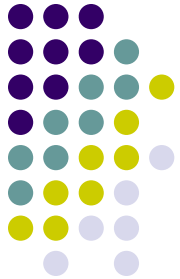
- this关键字的使用
  - 可以使用this关键字在一个构造方法中调用另外的构造方法
  - 代码更简洁，维护起来也更容易
  - 通常用参数个数比较少的构造方法调用参数个数最多的构造方法

2.3.1 构造方法(续)
——BankAccount.java



- 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;
 }
```

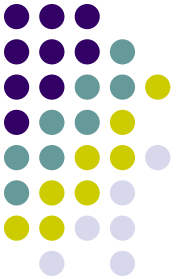


2.3.2 内存回收技术

- 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

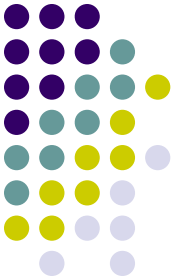
2.3.2 内存回收技术(续)

——垃圾收集器



- 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

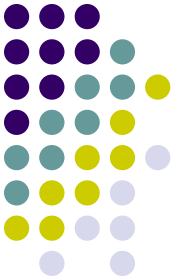
2.3.2 内存回收技术(续)
——finalize()方法



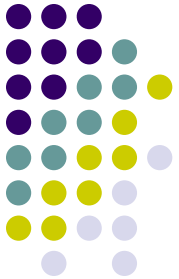
- finalize()方法
  - 在类java.lang.Object中声明，因此 Java中的每一个类都有该方法
  - 用于释放系统资源，如关闭打开的文件或socket等
  - 声明格式

    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

2.3.2 内存回收技术(续)
——同C和C++的区别

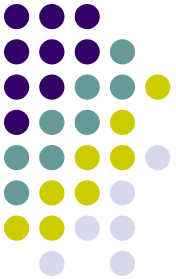


- 同C和C++的区别
  - C语言中通过free来释放内存
  - C++中则通过delete来释放内存
  - 在C和C++中，如果程序员忘记释放内存，则容易造成内存泄漏甚至导致内存耗尽
  - 在Java中不会发生内存泄漏情况，但对于其它资源，则有产生泄漏的可能性



2.4 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
  - 声明toString()方法
  - 声明存取款方法
  - 使用DecimalFormat类
  - 声明类方法生成特殊的实例
  - 声明类变量



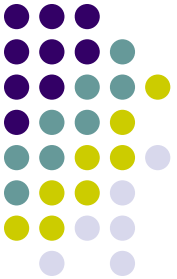
2.4.1 声明BankAccount类

- 包括状态、构造方法、get方法及set方法

应用举例

```
public class BankAccount{
    private String ownerName;
    private int accountNumber;
    private float balance;
    public BankAccount() {
      this("", 0, 0);
    }
    public BankAccount(String initName, int initAccNum, float initBal) {
      ownerName = initName;
      accountNumber = initAccNum;
      balance = initBal;
    }
```

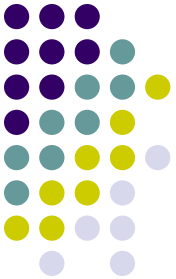
2.4.1 声明BankAccount类(续)
——BankAccount.java



应用举例

```
    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;
    }
}
```

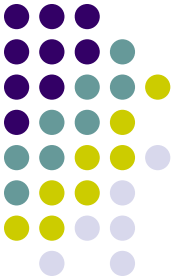
2.4.1 声明BankAccount类
——AccountTester.java



- 声明测试类AccountTester

应用举例

```
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());
    }
}
```



2.4.1 声明BankAccount类(续)
——AccountTester测试结果

应用举例

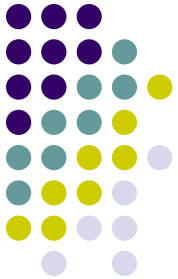
- 测试结果

Here is the account: BankAccount@372a1a

Account name: ZhangLi

Account number: 100023

Balance: $100.0



2.4.2 声明toString()方法

应用举例

- 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：

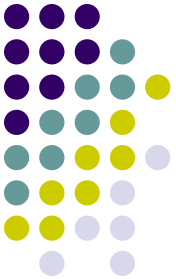
```
getClass().getName() + '@' +
Integer.toHexString(hashCode())
```

    - 下面的两行代码等价

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

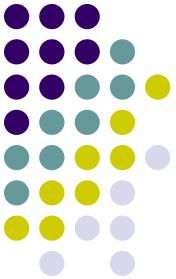
  - 如果需要特殊的转换功能，则需要自己重写toString()方法

2.4.2 声明toString()方法(续)
——几点说明



- toString()方法的几点说明
  - 必须被声明为public
  - 返回类型为String
  - 方法的名称必须为toString，且没有参数
  - 在方法体中不要使用输出方法System.out.println()

2.4.2 声明toString()方法(续)
——修改BankAccount类

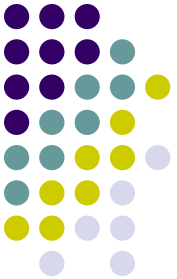


- 为BankAccount类添加自己的toString()方法

应用举例

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

2.4.2 声明toString()方法(续)
——测试结果



- 对BankAccount类进行重新编译并运行测试类BankAccountTester，结果如下

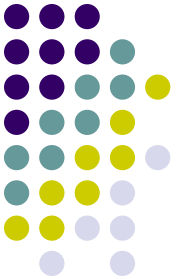
应用举例

Here is the account: Account #100023 with balance $100.0

Account name: ZhangLi

Account number: 100023

Balance: $100.0



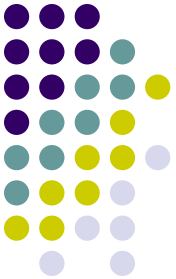
2.4.3 声明存取款方法

- 给BankAccount类增加存款及取款方法

应用举例

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

2.4.3 声明存取款方法(续)
——修改AccountTester.java

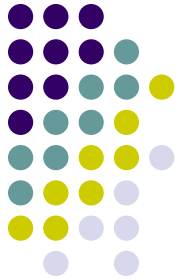


应用举例

```
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(anAccount);
        System.out.println();
        anAccount = new BankAccount("WangFang", 100024,0);
        System.out.println(anAccount);
        anAccount.deposit(225.67f);
        anAccount.deposit(300.00f);
        System.out.println(anAccount);
        anAccount.withdraw(400.17f);
        System.out.println(anAccount);
   }
}
```

2.4.3 声明存取款方法(续)

——测试结果



- 测试结果

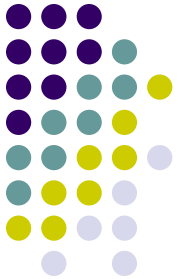
  Account #100023 with balance $100.0

应用举例

Account #100024 with balance $0.0

Account #100024 with balance $525.67

Account #100024 with balance $125.49997

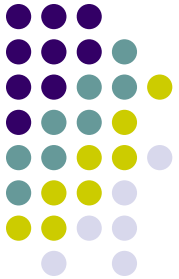


2.4.4 使用DecimalFormat类

- DecimalFormat类
  - 在java.text包中
  - 在toString()方法中使用DecimalFormat类的实例方法format对数据进行格式化
  - 修改后的toString()方法如下

应用举例

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



2.4.4 使用DecimalFormat类(续)
——测试结果

- 对BankAccount类重新进行编译，再运行BankTester类，运行结果如下

应用举例

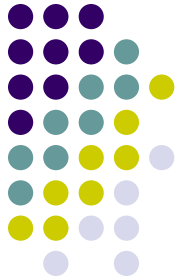
Account #100023 with balance $100.00

Account #100024 with balance $0.00
Account #100024 with balance $525.67
Account #100024 with balance $125.50

2.4.5 声明类方法生成特殊的实例

- 声明类方法返回特殊的BankAccount实例，作用是生成特定的几种账号样例
  - Example1()方法

```
public static BankAccount example1() {
    BankAccount ba = new BankAccount();
    ba.setOwnerName("LiHong");
    ba.setAccountNumber(554000);
     ba.deposit(1000);
     return ba;
}
```

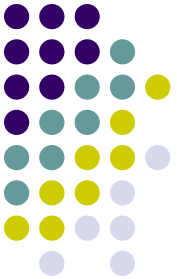


2.4.5 声明类方法生成特殊的实例(续)

——example2()方法

- Example2()

```
public static BankAccount example2() {
        BankAccount ba = new BankAccount();
        ba.setOwnerName("ZhaoWei");
        ba.setAccountNumber(554001);
        ba.deposit(1000);
        ba.deposit(2000);
        return ba;
 }
```

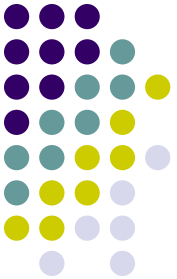


2.4.5 声明类方法生成特殊的实例(续)

——emptyAccountExample()方法

- emptyAccountExample()

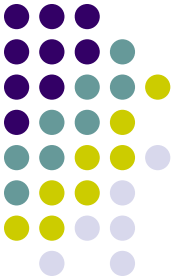
```
public static BankAccount emptyAccountExample()
{
        BankAccount ba = new BankAccount();
        ba.setOwnerName("HeLi");
        ba.setAccountNumber(554002);
        return ba;
  }
```



2.4.6 声明类变量

- 修改BankAccount类
  - 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

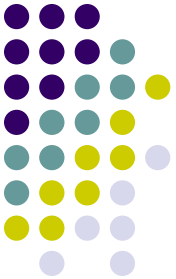
2.4.6 声明类变量
——BankAccount2.java



```
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;
  }
```

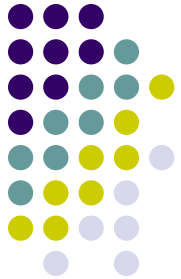
2.4.6 声明类变量

——BankAccount2.java



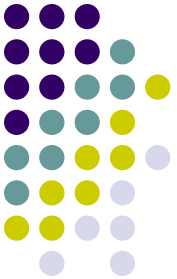
```
public static BankAccount2 example1() {
    BankAccount2 ba = new BankAccount2();
    ba.setOwnerName("LiHong");
    ba.deposit(1000);
    return ba;
  }
  public static BankAccount2 example2() {
    BankAccount2 ba = new BankAccount2();
    ba.setOwnerName("ZhaoWei");
    ba.deposit(1000);
    ba.deposit(2000);
    return ba;
   }
  public static BankAccount2 emptyAccountExample() {
    BankAccount2 ba = new BankAccount2();
    ba.setOwnerName("HeLi");
    return ba;
  }
```

2.4.6 声明类变量
——BankAccount2.java



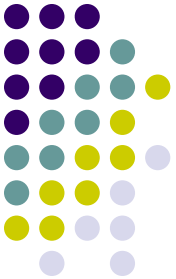
```
public int getAccountNumber() {
  return accountNumber;
}
public String getOwnerName() {
  return ownerName;
}
public float getBalance() {
  return balance;
}
public void setOwnerName(String aName) {
  ownerName = aName;
}
```

2.4.6 声明类变量
——BankAccount2.java



```
public String toString() {
   return("Account #"+
            new java.text.DecimalFormat("000000").format(accountNumber) +
           " with balance " +
           new java.text.DecimalFormat("$0.00").format(balance));
  }
  public float deposit(float anAmount) {
    balance += anAmount;
    return balance;
  }
  public float withdraw(float anAmount) {
   if (anAmount <= balance)
     balance -= anAmount;
    return anAmount;
  }
}
```

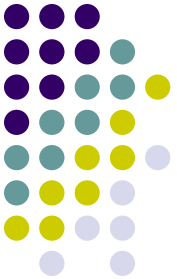
2.4.6 声明类变量
——AccountTester2.java



```
public class AccountTester2 {
   public static void main(String args[]) {
      BankAccount2 bobsAccount, marysAccount, biffsAccount;
      bobsAccount = BankAccount2.example1();
      marysAccount = BankAccount2.example1();
      biffsAccount = BankAccount2.example2();
      marysAccount.setOwnerName("Mary");
      marysAccount.deposit(250);
      System.out.println(bobsAccount);
      System.out.println(marysAccount);
      System.out.println(biffsAccount);
   }
}
```

2.4.6 声明类变量

——AccountTester2测试结果

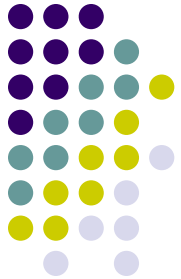


- 测试结果

Account #000001 with balance $1000.00

Account #000002 with balance $1250.00

Account #000003 with balance $3000.00



2.5 本章小结

- 本章内容
  - 面向对象程序设计的基本概念和思想
  - Java语言类与对象的基本概念和语法，包括类的声明、类成员的访问，以及对象的构造、初始化和回收
- 本章要求
  - 理解类和对象的概念
  - 熟练使用类及其成员的访问控制方法
  - 熟练掌握各种构造方法
  - 了解java的垃圾回收机制