



JAVA语言程序设计

第二章

类与对象的基本概念

本课件以清华大学郑莉老师的课件为基础进行改编



目录

2.2 类与对象

2.3 对象初始化和回收

2.4 应用举例

2.6 本章小节



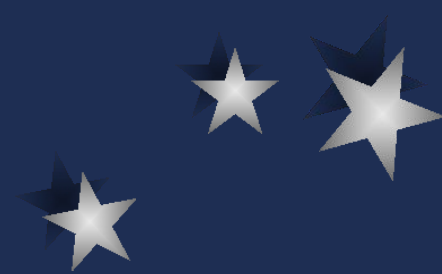
2.2 类与对象

- 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



● 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，其他类可以使用此类

abstract: 指明此类为抽象类

final: 指明此类为终结类，不能有子类

那么，接口的名字应写在**implements**之后

[public] [abstract | final] class 类名称

[extends 父类名称]

[implements 接口名称列表]

{

数据成员变量声明及初始化;

方法声明及方法体;

}



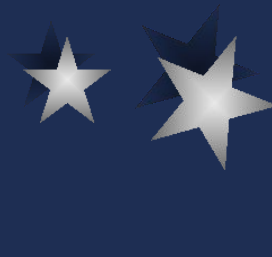
2.2.1 类的声明(续)

——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

创建Clock对象

```
Clock aclock = new clock();
```

3步走:

1.声明一个引用变量： 类名 变量名;

```
Clock aclock = new clock();
```

2.创建对象： new <classname>()

```
Clock aclock = new clock();
```

3.连接对象和引用

```
Clock aclock = new clock();
```



Clock



Clock对象



Clock对象

2.2.2 对象的声明与引用



- 引用变量：存储对象的引用（reference，相当于对象的存储地址）



- 注意：声明一个引用变量时并没有对象生成
- 引用变量可以被赋以空值。如：`clock=null;`

总结一下：对象与对象的引用

- 尽管将一切都“看作”对象，但实际是通过一个指向对象的“句柄”（**Handle**）或“引用”来操纵对象
- 可将这一情形想象成用遥控器（引用）操纵电视机（对象）

只要握住这个遥控器，就相当于掌握了与电视机连接的通道。

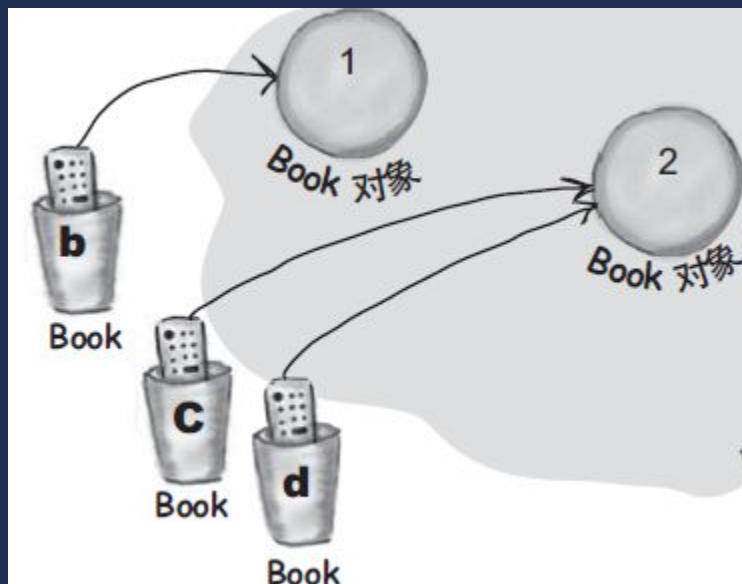
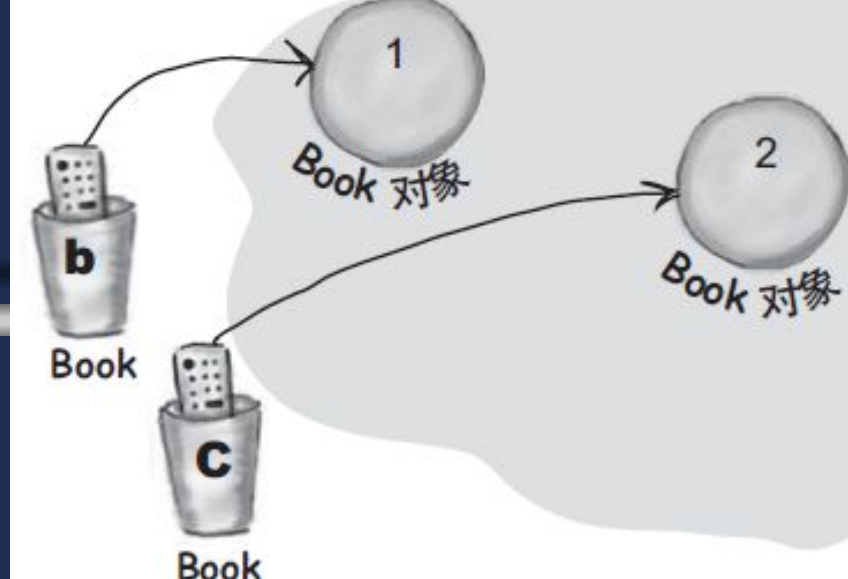
但一旦需要“换频道”或者“关小声音”，我们实际操纵的是遥控器（引用），再由遥控器自己操纵电视机（对象）。

如果要在房间里四处走走，并想保持对电视机的控制，那么手上拿着的是遥控器，而非电视机。

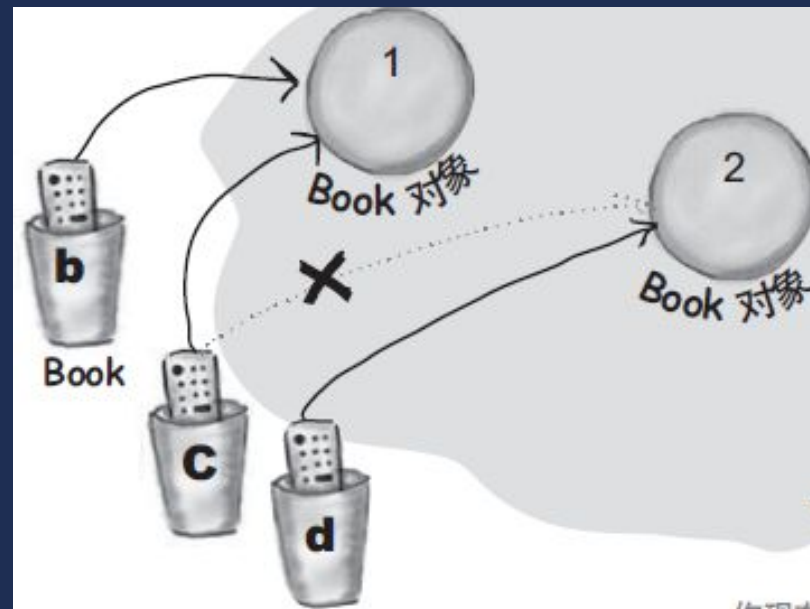
- 引用和对象可以单独存在，但要记得建立两者的关联


```
Book b = new Book();  
Book c = new Book();
```

```
Book d = c;
```



```
c = b;
```



2.2.3 数据成员



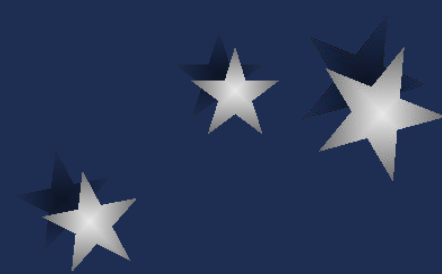
- 数据成员

表示Java类的状态

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（这是关键！）



2.2.3 数据成员(续)

声明格式

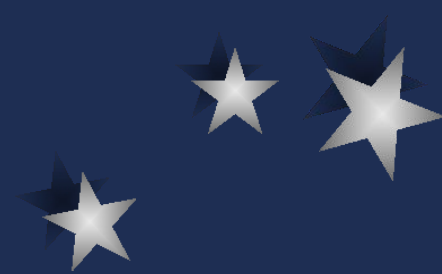
[public | protected | private]

[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



2.2.3 数据成员(续)

- 当一个对象被创建时会对其中各种类型的成员变量自动进行初始化赋值

成员变量类型	初始值
byte	0
short	0
int	0
long	0
float	0.0F
double	0.0D
char	'\u0000' (空格)
boolean	false
All reference type	null

2.2.3 数据成员(续)

——实例变量

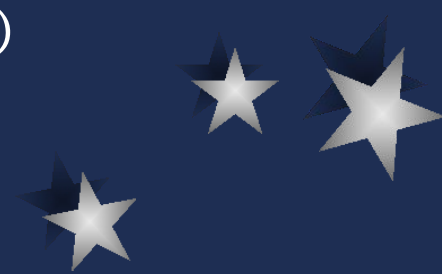
- 实例变量（属性）

没有**static**修饰的变量（属性）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
（当然，这不是一种好方式！）

<实例名>.<实例变量名>



- 可以将一个类的声明放在一个单独的文件中，也可以将多个类的声明放在一个文件中（在这种情况下，最多只能有一个类声明为公有类）。**Java**源文件名必须根据文件中的公有类名来命名，并且要区分大小写。

- 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：

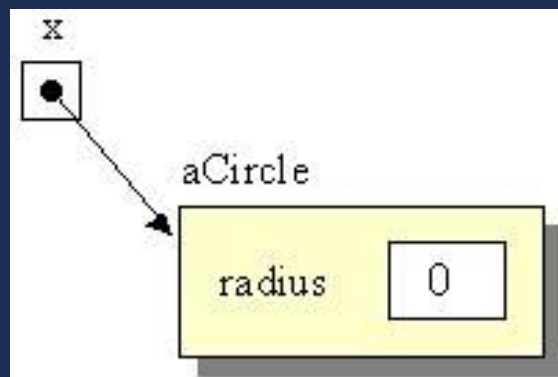
Circle@26b249

radius =0

- 解释

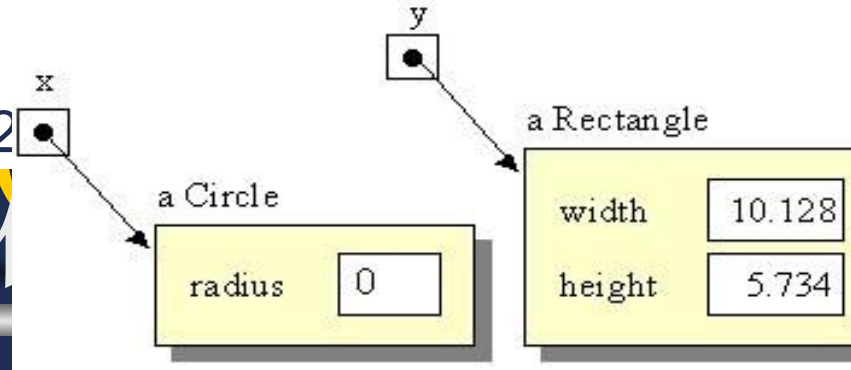
@之后的数值为x所指的对象的存储地址

x的值及对象的状态如图



- 编译后运行结果如下：

Circle@82f0db Rectangle@92d342

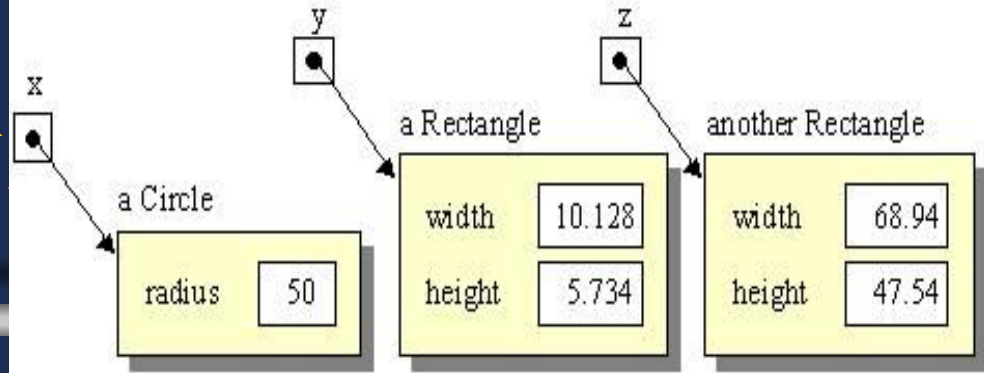


- 声明一个表示矩形的类，保存在**Rectangle.java**中；编写测试类，保存在**ShapeTester.java**中，二文件保存在相同的目录下

```
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);  
    }  
}
```


- 编译后运行结果如下：

50 10.128 68.94



- 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;  
        y.width = 10.128;  
        y.height = 5.734;  
        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)

——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)

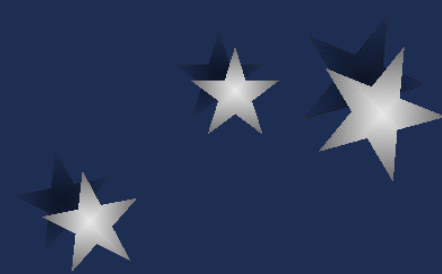
——例2_5运行结果

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)

——final修饰符

- 实例变量和类变量都可被声明为**final**，表示这个变量一旦被初始化便不可改变
- 其初始化可以在两个地方

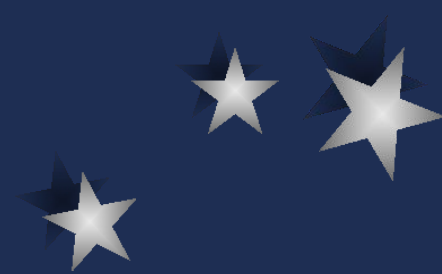
- 1.在**final**变量定义时直接给其赋值。

- 2.在构造方法中赋值。

这两个地方只能选其一，不能同时既在定义时给了值，又在构造方法中给另外的值。

final实例变量可在1或2中初始化

final类变量必须在1中初始化



2.2.4 方法成员



- 方法成员

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
分为实例方法和类方法



- **static** 指明方法是一个类方法
- **返回类型**：方法返回值的类型，可以是任意的**Java**数据类型。
- **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。

式参数。

声明格式

[public | protected | private]

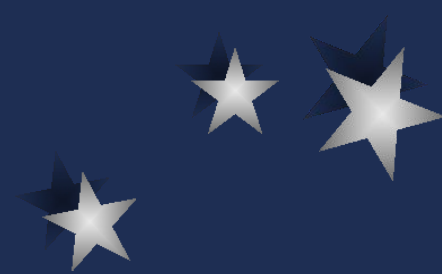
[static][final][abstract] [native] [synchronized]

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

{

方法体

}



2.2.4 方法成员(续)

——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)

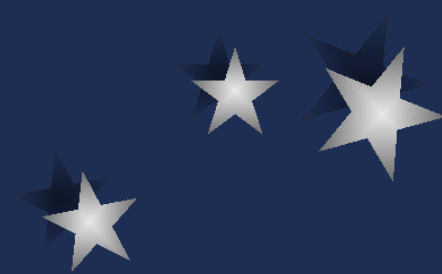
——实例方法

- 实例方法

表示特定对象的行为

声明时前面不加**static**修饰符

使用时需要将其发送给一个实例对象
(也称给对象发送一条消息)



2.2.4 方法成员(续)

——例2_6

- 在Circle类中声明计算周长的方法

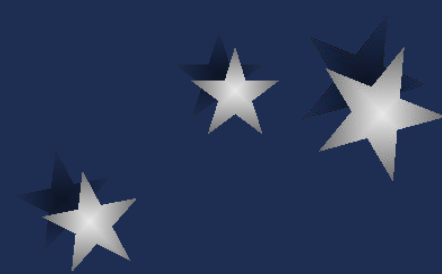
```
public class Circle {  
    static double PI = 3.14159265;  
    int radius;  
    public double circumference() {  
        return 2 * PI * radius;  
    }  
}
```

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)

运行结果

Circle 1 has circumference 314.159265

Circle 2 has circumference 62.831853

- 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_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;  
    }  
}
```



- 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

- 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_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)

——类方法

- 类方法

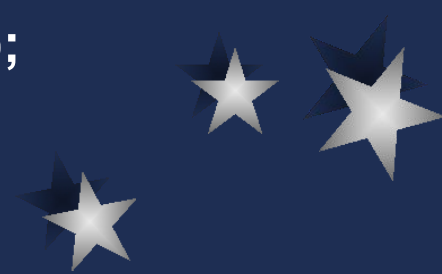
也称为静态方法，表示类中对象的共有行为

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

```
import java.util.*;
public class EasyScanner
{
    public static int nextInt()
    {
        Scanner sc = new Scanner(System.in);
        double d = sc.nextDouble();
        return d;
    }
    public static double nextDouble()
    {
        Scanner sc = new Scanner(System.in);
        double d = sc.nextDouble();
        return d;
    }
}
```

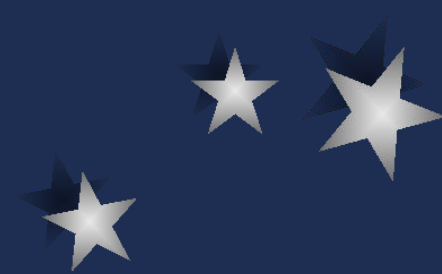


```
public static boolean nextBoolean()
```

```
{  
    Scanner sc = new Scanner(System.in);  
    boolean d = sc.nextBoolean();  
    return d;  
}
```

```
public static String nextString()
```

```
{  
    Scanner sc = new Scanner(System.in);  
    String s = sc.nextLine();  
    return s;  
}
```



```
public static char nextChar()
{
    Scanner sc = new Scanner(System.in);
    char c = sc.next().charAt(0);
    return c;
}
}
```



-
- 注意：对象的属性具有默认值，但方法中定义的局部变量没有默认值！如果未初始化就使用，编译器会报错！



想一想

- 注意：

类方法不能访问实例变量（编译器不知道访问的是哪个实例变量）；

类方法不能调用实例方法（因为实例方法往往会访问实例变量）

- 思考：

实例方法能否对该类中的类变量进行读写？



总结

- 类的定义（属性和方法）
- 对象的创建
- 对象的访问（引用变量）
- 实例变量（属性）和类变量（属性）的区别、用途
- 实例方法和类方法（静态方法）的区别、用途



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(不同包中的类可以同名)
- 提供包一级的封装及存取权限（一个类的属性和操作只允许本包中的类进行访问）

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)

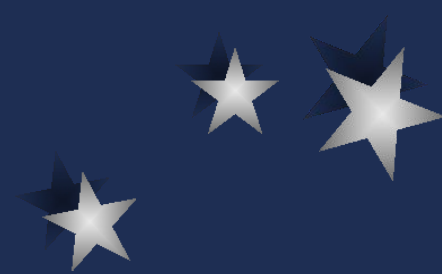
- 包的声明

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
- 如 `import java.io.*;` // 引入包中的所有类
- `import java.io. FileWriter ;` // 引入包中的一个类

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

当使用**public**修饰符时，表示所有其他的类都可以访问此类；

当没有修饰符时，表示只有与此类处于同一包中的其他类可以访问此类。

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): 包可见、子类可见

- 只可被同一类及其子类的实例对象、以及同一包中类的实例对象访问

私有(private)

- 只能被这个类本身访问, 在类外不可见

默认(default): 包可见

- 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 类的访问控制



● 面向对象的封装原则：

只对外暴露必须暴露的部分，目的是减少对象间的耦合！

考虑Circle存在的问题：radius属性完全暴露在外，任人宰割！

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

Circle x ;
x = new Circle();
x.radius = 0; //显然0是不合理的，但无可奈何...

下述代码能通过编译么？

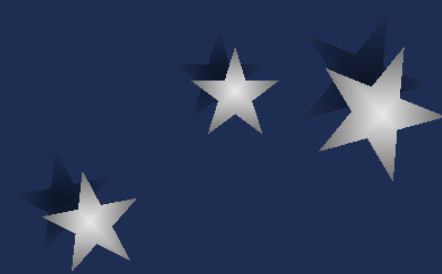
```
Circle c1 = new Circle();  
c1.radius = 50;
```

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”

- 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编译

- 又要保护数据，又要允许数据被访问，咋办？
- 办法就是：
 - 将实例变量标记为private
 - 提供公有的get和set方法对实例变量进行读写



2.2.6 类成员的访问控制(续)

——get方法

- **get方法**

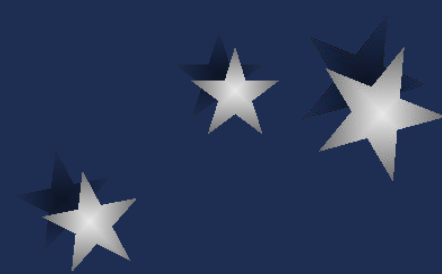
功能是取得属性变量的值

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**

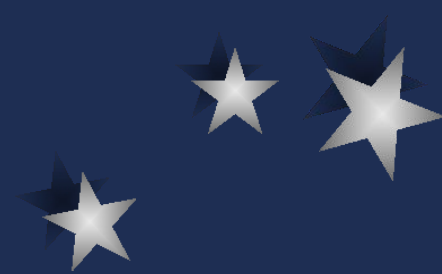
功能是修改属性变量的值

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)

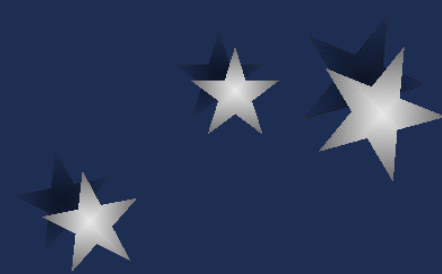
——set方法

- 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;  
}
```

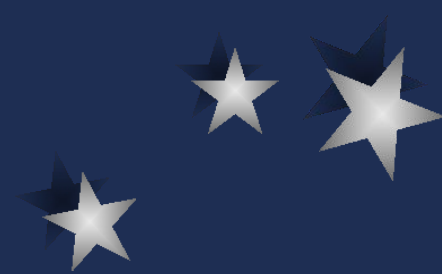


- 使用**get**和**set**方法的好处:

强迫其他的程序一定要通过**set**，如此就能够检查参数合理性，并判断是否执行。

在**get**中也可以进行一些处理，如对浮点值进行四舍五入后再对外提供。

```
public void setRadius(int radius){  
    if(radius>0)  
        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

对象在堆中分配内存

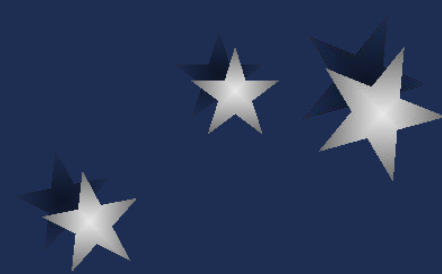
对象的实例变量保存在所属的对象上，位于堆上

对象方法中的局部变量和参数，在栈中分配内存

2.3 对象初始化和回收

- 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

在新建对象时被调用，用来初始化对象的实例变量

Java中的每个类都有构造方法

如果程序员没有定义构造方法，系统会自动提供默认的构造方法

一旦程序员定义了构造方法，则系统将不再提供默认构造方法。

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**public**（特殊情况下声明为**private**）

- 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（**null**）



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，返回值可以不同，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)

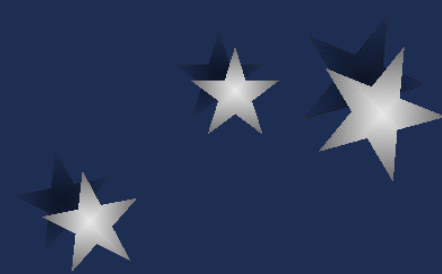
——例2_13

- 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)

——Bush.java

- 构建一个Bush类，有两个有参数的构造函数

```
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

- 解释

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)

——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（原因在类继承中会讲）

在声明构造方法时，好的声明习惯是

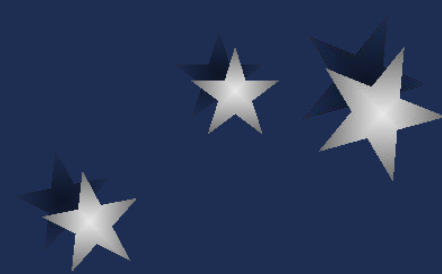
- 不声明构造方法（编译器提供默认构造方法）
- 如果声明(此时编译器不再提供默认构造方法)，至少声明两个构造方法，其中一个是无参构造方法

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)

——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同一个类的另外的构造方法，使代码更简洁。

this()调用只能出现在构造函数中，且必须是第一行语句！

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BankAccount类中无参数和二参数的构造函数

```
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

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。

当一个对象的最后一个引用消失时，对象就会变成可回收的。



2.3.2 内存回收技术

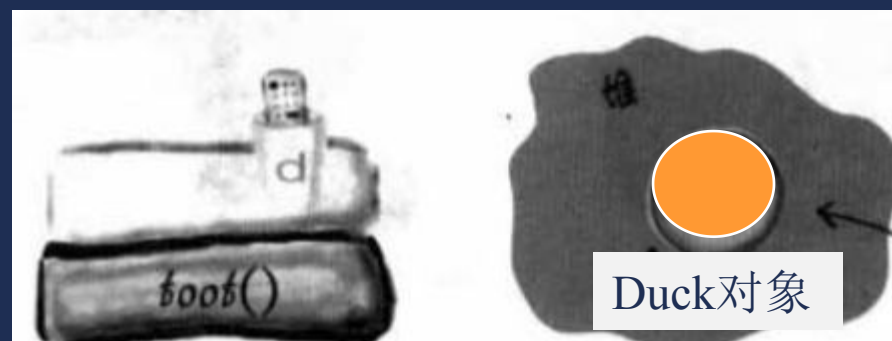
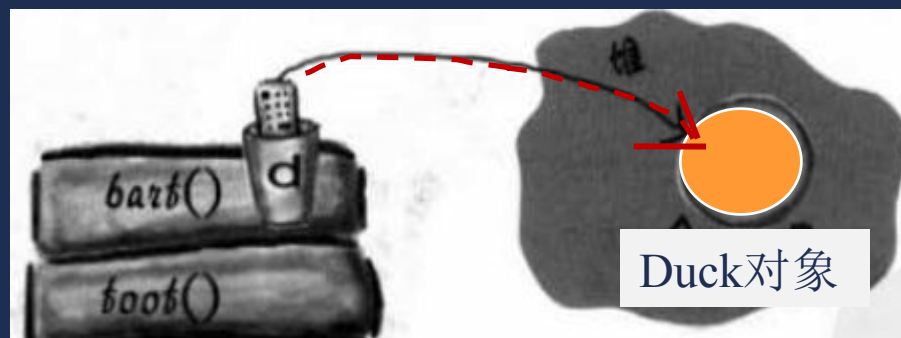
-释放对象的引用

有三种方法可以释放对象的引用：

- 1) 引用永久性的离开它的范围



```
public class StackRef{
    public void foof(){
        barf();
    }
    public void barf(){
        Duck d = new Duck();
    }
}
```



Barf执行完毕，d也就被释放了

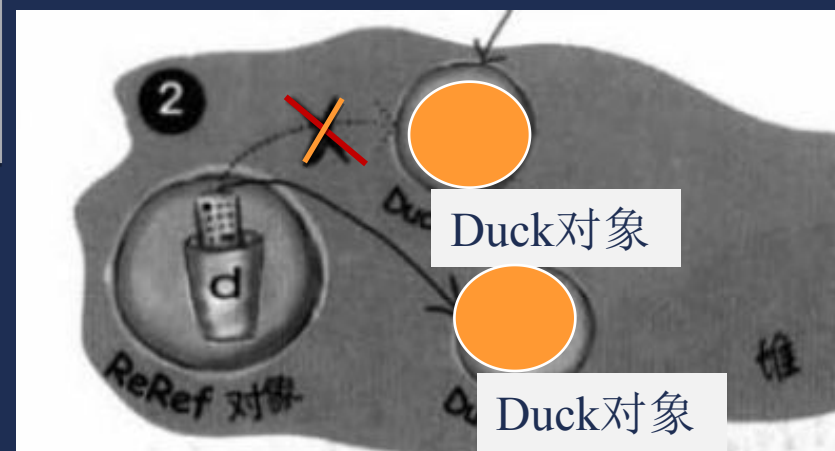
2.3.2 内存回收技术

-释放对象的引用

● 2) 引用被赋值到其他对象上

```
public class ReRef{
    Duck d = new Duck();

    public void go(){
        d = new Duck();
    }
}
```



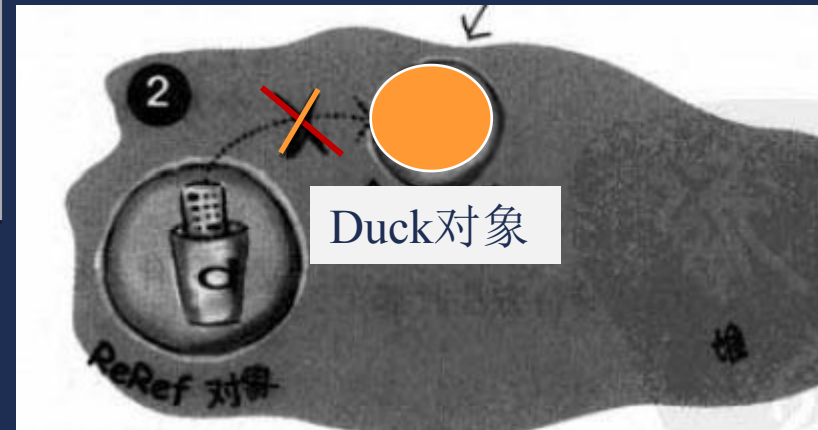
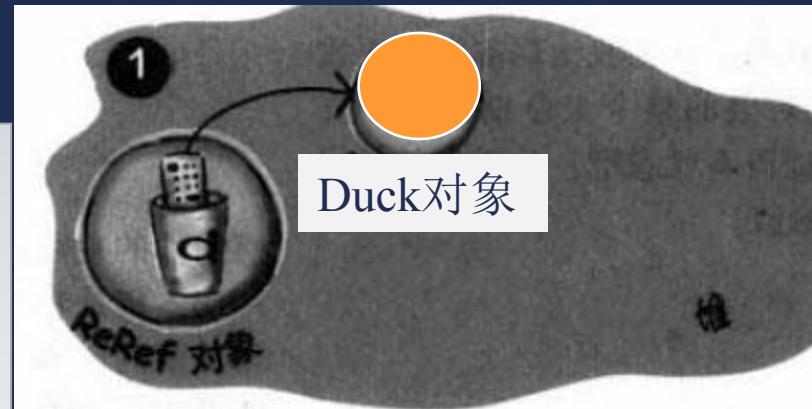
2.3.2 内存回收技术

-释放对象的引用

● 3) 直接将引用设定为null

```
public class ReRef{
    Duck d = new Duck();

    public void go(){
        d = null;
    }
}
```



2.3.2 内存回收技术

- **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。该方法在 `java.lang.Object` 中实现，所有的Java类都继承该类，因此也就继承了这个方法，也无需重写这个方法。



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

对象
初始
化和
回收

- **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

用于释放内存资源、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

对象初始化和回收

- 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，则容易造成内存泄露甚至导致内存耗尽

在Java中不会发生内存泄露情况，但对于其它资源，则有产生泄露的可能性

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

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

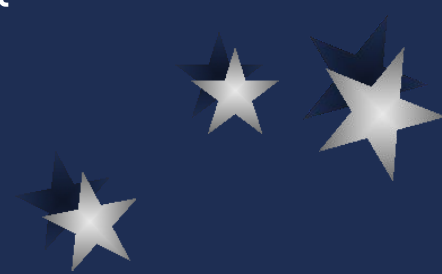
```
public class BankAccount{
    private String ownerName;
    private int accountNumber;
    private float balance;
    public BankAccount() {
        this("", 0, 0);
    }
    public BankAccount(String initName, int initAccNum, float initBal) {
        ownerName = initName;
        accountNumber =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



2.4.1 声明BankAccount类(续)

——BankAccount.java

```
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 账户

- 测试结果

Here is the account: BankAccount@372a1a

Account name: ZhangLi

Account number: 100023

Balance: \$100.0

- 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());
    }
}
```

2.4.2 声明toString()方法

- **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());
```

hashCode()方法通过运用对象的内容执行一个哈希函数来生成一个int值。

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

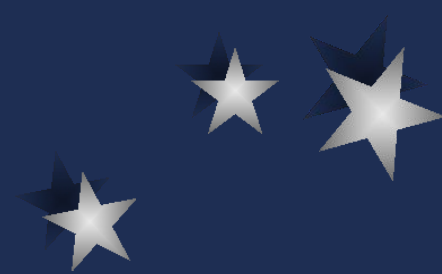
`System.out.println()`



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

- 为BankAccount类添加自己的toString()方法

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



2.4.2 声明toString()方法(续)

——测试结果

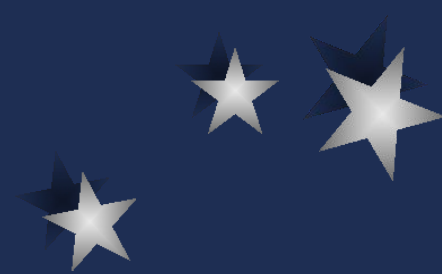
- 对**BankAccount**类进行重新编译并运行测试类**BankAccountTester**，结果如下

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

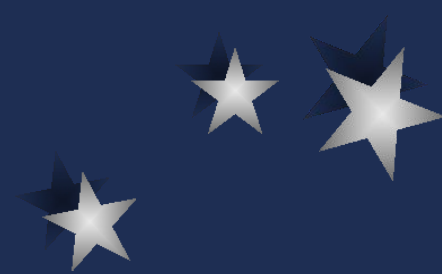
- 给BankAccount类增加存款及取款方法

//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

● 测试结果

Account #100023 with balance \$100.0
 Account #100024 with balance \$0.0
 Account #100024 with balance \$525.67
 Account #100024 with balance \$125.49997

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

2.4.6 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

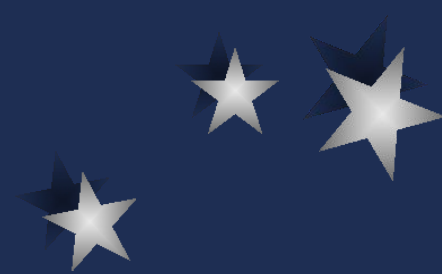
LAST_ACCOUNT_NUMBER的值累加1

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

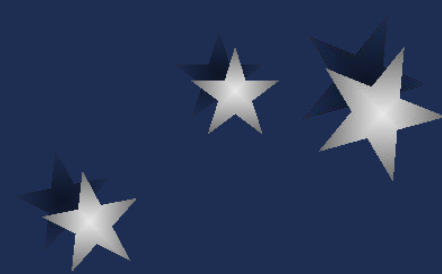
```
public int getAccountNumber() {  
    return accountNumber;  
}  
public String getOwnerName() {  
    return ownerName;  
}  
public float getBalance() {  
    return balance;  
}  
public void setOwnerName(String aName) {  
    ownerName = aName;  
}
```



2.4.6 声明类变量

——BankAccount2.java

```
public String toString() {  
    return("Account #" + accountNumber + " with balance " + balance);  
}  
public float deposit(float anAmount) {  
    balance += anAmount;  
    return balance;  
}  
public float withdraw(float anAmount) {  
    if (anAmount <= balance)  
        balance -= anAmount;  
    return anAmount;  
}  
}
```



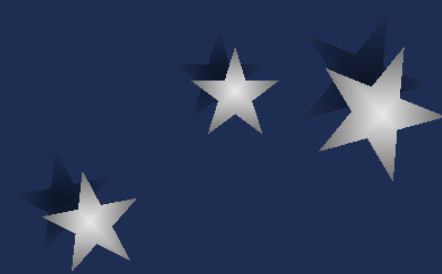
2.4.6

- 测试结果

Account #000001 with balance \$1000.00

Account #000002 with balance \$1250.00

```
public class AccountTester2 {
    public static void main(String args[]) {
        BankAccount2 bobsAccount, marysAccount, biffsAccount;
        bobsAccount = new BankAccount2 ("Bob", 1000);
        marysAccount = new BankAccount2 ("mary", 1000);
        marysAccount.setOwnerName("Mary");
        marysAccount.deposit(250);
        System.out.println(bobsAccount);
        System.out.println(marysAccount);
    }
}
```



2.6 本章小结

- 本章内容

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

理解类属性、类方法的特点和应用场合

理解抽象方法的特点和应用目的

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

