



JAVA语言程序设计

---

## 第四章 类的重用

本课件以清华大学郑莉老师的课件为基础进行改编



# 目录

---

- 4.1 类的继承
- 4.2 Object类（自学）
- 4.3 终结类与终结方法
- 4.4 抽象类
- 4.5 类的组合
- 4.6 包的应用
- 4.7 本章小结



## 4.1 类的继承

---

- 类的继承

- 一种由已有的类创建新类的机制，是面向对象程序设计的基石之一
- 通过继承，可以根据已有类来定义新类，新类拥有已有类的所有功能
- **Java**只支持类的单继承，每个子类只能有一个直接父类



## 4.1.1 继承的概念

---

### 类的继承

- 继承
  - 父类是所有子类的公共属性及方法的集合，子类则是父类的特殊化
  - 只需指明新类与现有类的不同，即增加自己的属性和方法即可
  - 继承介于类之间，而不是对象之间



# 4.1.1 继承的概念

## ——基类和派生类

类的继承

- 基类(base class)
  - 也称超类(superclass)
  - 是被直接或间接继承的类
- 派生类(derived-class)
  - 也称子类(subclass)
  - 继承其他类而得到的类
  - 继承所有祖先的状态和行为
  - 派生类可以增加变量和方法
  - 派生类也可以覆盖(override)继承的方法

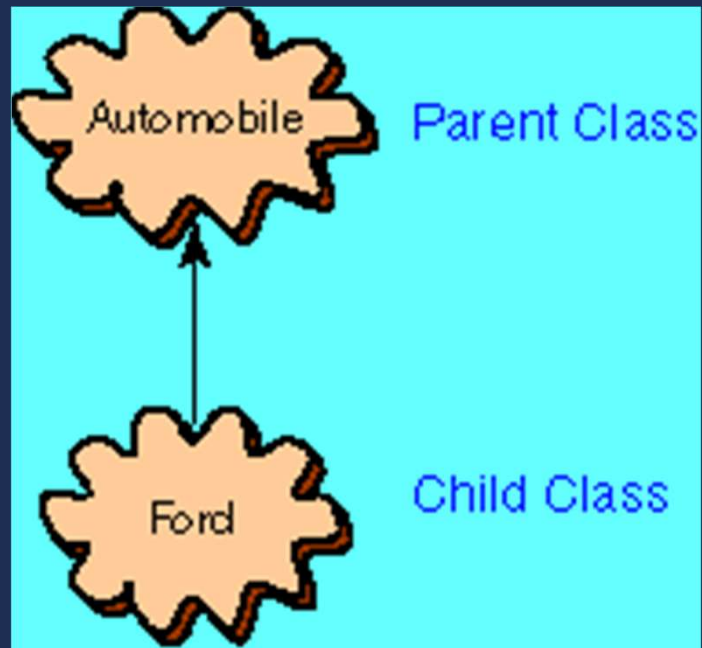


## 4.1.1 继承的概念

——is\_a关系

- 子类对象与父类对象存在“IS A”(或“is kind of”)的关系

类的继承



## 4.1.1 继承的概念

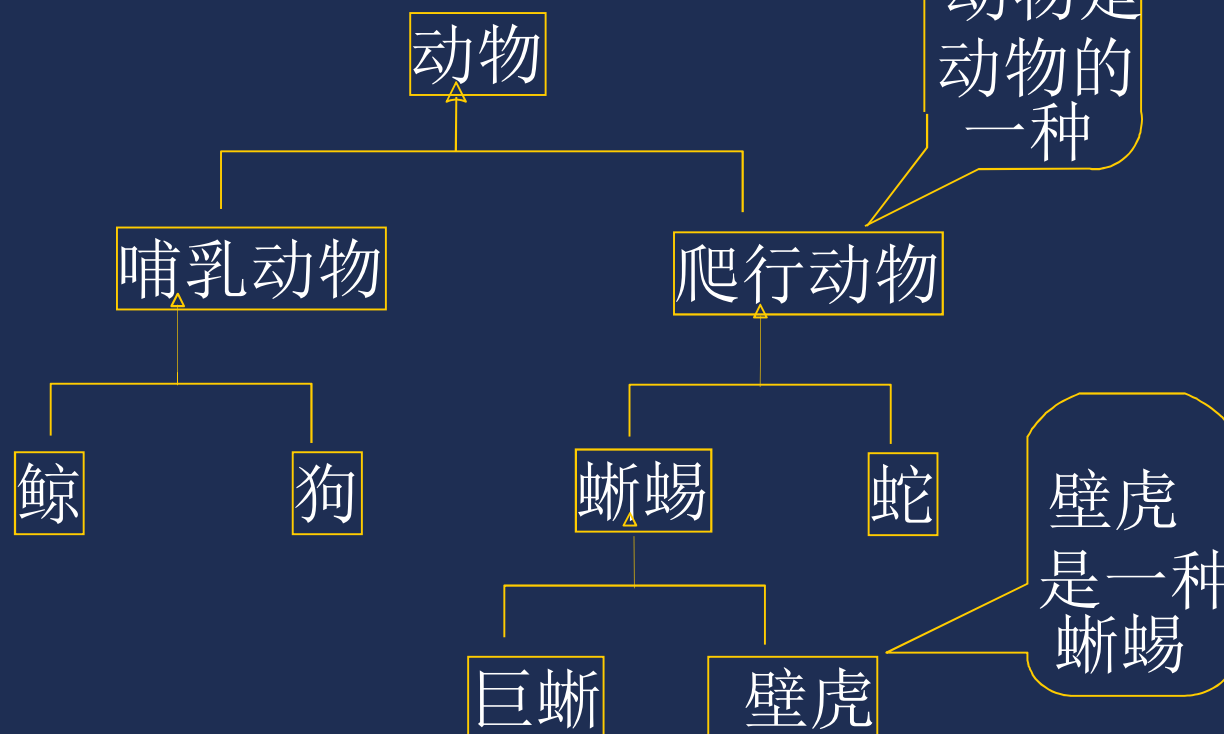
——图4\_1

### ● 动物类层次举例

类的继承

一般

具体

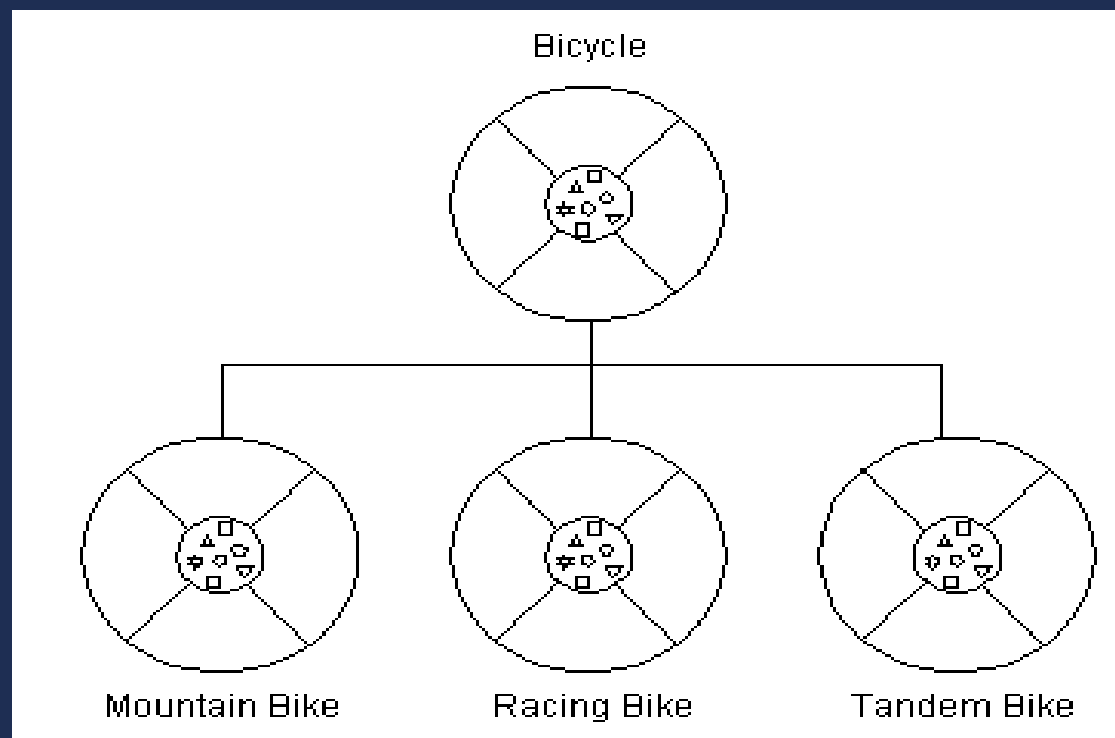


## 4.1.1 继承的概念

——又一个例子

- 举例

类的继承





## 4.1.1 继承的概念

### ——派生类对象

- 派生类产生的对象
  - 从外部来看，它应该包括
    - 与基类相同的接口
    - 也许具有更多的方法和数据成员
  - 其内包含着一个基类类型的子对象



## 4.1.1 继承的概念

### ——继承的意义

类的继承

- 继承的意义
  - 采用继承的机制来组织、设计系统中的类，可以提高程序的抽象程度，使之更接近于人类的思维方式
  - 采用继承编写的程序结构清晰，节省了编程时间，因为代码的编写量减少，因此也降低了维护的工作量



## 4.1.2 继承的语法

### 类的继承

- 继承的语法

- 语法形式

```
class childClass extends parentClass
{
    //类体
}
```

- 说明

- 关键字**extends**说明要声明的类需要继承父类的属性和行为
    - **parentClass**是被继承的父类名称



## 4.1.2 继承的语法

### ——派生类成员

#### 类的继承

- 派生类成员
  - 继承基类中声明为public 或protected 的成员
  - 如果基类和派生类在同一个包中，派生类可以继承基类没有访问权限标志符的(缺省的)成员
  - 如果派生类某成员名和基类某成员名同名，派生类同名变量成员隐藏基类成员，派生类同名方法成员覆盖基类成员



# 继承实例

class 继承

类名

属性

操作

**Employee**

- name: String  
- number: String

+ Employee(String, String)  
+ getNumber(): String  
+ setNumber(String): void

**PartTimeEmployee**

- hourlyPay: double

+ calculateWeeklyPay(int): double  
+ getHourlyPay(): String  
+ PartTimeEmployee(double, String, String)  
+ setHourlyPay(double): void

Employee:雇员

PartTimeEmployee:兼职雇员

兼职雇员是一类雇员，除了继承雇员全部的属性和操作，还具有自己特有的属性和操作。



# Employee 定义

---

```
public class Employee
{
    private String number;
    private String name;

    public Employee(String numberIn, String nameIn)
    {
        number = numberIn;
        name = nameIn;
    }
}
```



# Employee 定义

---

```
public void setName(String nameIn)
{
    name = nameIn;
}
public String getNumber()
{
    return number;
}
public String getName()
{
    return name;
}
}
```




# PartTimeEmployee 定义

---

```
public class PartTimeEmployee extends Employee
{
    private double hourlyPay;

    public PartTimeEmployee(String numberIn, String nameIn, double
hourlyPayIn)
    {
        super(numberIn, nameIn);//调用父类构造方法
        hourlyPay = hourlyPayIn;
    }

    public void setHourlyPay(double hourlyPayIn)
    {
        hourlyPay = hourlyPayIn;
    }
}
```





# PartTimeEmployee 定义

---

```
public double getHourlyPay()  
{  
    return hourlyPay;  
}
```

```
public double calculateWeeklyPay(int noOfHoursIn)  
{  
    return hourlyPay * noOfHoursIn;  
}
```

```
}
```



## 4.1.2 继承的语法

### ——公有(及保护)成员的继承

类的继承

- 公有(及保护)成员的继承
  - 子类的对象可以使用其父类中声明为公有(及保护)的属性和方法, 就如同在其自己的类中声明一样



---

```
public class EmployeeTest {  
    public static void main(String[] args) {  
        Employee emp=new  
Employee("1001","Jack");  
        PartTimeEmployee pEmp=new  
PartTimeEmployee("1002","Susan",20.5);  
  
        System.out.println("hello"+emp.getName());  
        //子类可以直接使用父类中定义的可见性为public的方法  
        System.out.println("hello"+pEmp.getName());  
        System.out.println(pEmp.getName()+"your  
weeklyPay is:"+  
pEmp.calculateWeeklyPay(20));  
    }  
}
```

## 4.1.2 继承的语法

### ——私有属性及行为的继承

类的继承

- 私有成员的继承
  - 子类不能直接访问从父类中继承的私有属性及方法，但可以使用公有（及保护）方法进行访问



## 4.1.2 继承的语法(续)

### ——例4\_3

类的继承

```
public class B {  
    public int a = 10;  
    private int b = 20;  
    protected int c = 30;  
    public int getB() { return b; }  
}
```

```
public class A extends B {  
    public int d;  
    public void tryVariables() {  
        System.out.println(a);  
        System.out.println(b);  
        System.out.println(getB());  
        System.out.println(c);  
    }  
}
```

//允许

//不允许, 编译出错

//允许

//允许



- 子类无法直接访问父类中的私有属性，如何解决
  - 1. 可以将父类中的属性声明为public，但这样彻底违反了面向对象的封装原则
  - 2. 可以将父类中的属性声明为protected，父类中protected的属性可以在子类中直接访问。但1) 必须提前遇见这个类将来会有子类；2) 该方法减弱了类对信息的封装性。因为，Java中，protected的属性可以被同一包中的所有类访问（protected具有子类可见性和包可见性）
  - 3. 保持父类属性为private，但预先详细筹划提供哪些get和set方法。



## 4.1.3 隐藏和覆盖

---

类的继承

- 隐藏和覆盖
  - 是指子类对从父类继承来的属性变量及方法可以重新加以定义



## 4.1.3 隐藏和覆盖(续)

### ——属性的隐藏

```
class Parent {  
    Number aNumber;  
}
```

```
class Child extends Parent {  
    Float aNumber;  
}
```

类的继承

- 属性的隐藏
  - 子类对从父类继承来的属性变量重新加以定义，则从父类继承的属性将被隐藏
  - 子类拥有了两个相同名字的变量，一个继承自父类，另一个由自己声明
  - 当子类执行继承自父类的操作时，处理的是继承自父类的变量，而当子类执行它自己声明的方法时，所操作的就是它自己声明的变量，而把继承自父类的变量“隐藏”起来了





## 4.1.3 隐藏和覆盖(续)

### ——访问被隐藏的父亲属性

类的继承

- 如何访问被隐藏的父亲属性
  - 调用从父类继承的方法，则操作的是从父类继承的属性
  - 使用`super.`属性



```
class A1
{
    int x = 2;
    public void setx(int i)
    {
        x = i;
    }
    void printa()
    {
        System.out.println(x);
    }
}
```

```
class B1 extends A1
{
    int x=100;
    void printb()
    {
        super.x = super.x +10 ;
        System.out.println("super.x= " + super.x);
    }
}
```

```
public class Test2
{
    public static void main(String[] args)
    {
        A1 a1 = new A1();
        a1.setx(4);
        a1.printa();

        B1 b1 = new B1();
        b1.printb();
        b1.printa();

        b1.setx(6); // 将继承来的x值设置为6
        b1.printb();
        b1.printa();
        a1.printa();
    }
}
```

```
4
super.x= 12    x= 100
12
super.x= 16    x= 100
16
4
```

```

class A1
{
    static int x = 2;
    public void setx(int i)
    {
        x = i;
    }
    void printa()
    {
        System.out.println(x);
    }
}

```

```

class B1 extends A1
{
    int x=100;
    void printb()
    {
        super.x = super.x + 10 ;
        System.out.println("super.x= " + super.x);
    }
}

```

```

public class Test2
{
    public static void main(String[] args)
    {
        A1 a1 = new A1();
        a1.setx(4);
        a1.printa();

        B1 b1 = new B1();
        b1.printb();
        b1.printa();

        b1.setx(6); // 将继承来的x值设置为6
        b1.printb();
        b1.printa();
        a1.printa();
    }
}

```

```

4
super.x= 14   x= 100
14
super.x= 16   x= 100
16
16

```

## 4.1.3 隐藏和覆盖(续)

### ——例4\_4运行结果

类的继承

- 子类不能继承父类中的静态属性，但可以对父类中的静态属性进行操作。如在上面的例子中，将“**int x = 2;**”改为“**static int x = 2;**”，再编译及运行程序，会得到下面的结果

```
4
super.x= 14 x= 100
14
super.x= 16 x= 100
16
16
```

- 在上面的结果中，第一行及最后一行都是语句“**a1.printa();**”输出的结果，显然类 B 中的**printb()**方法修改的是类 A 中的静态属性**x**

## 4.1.3 隐藏和覆盖(续)

### ——方法覆盖

#### 类的继承

- 方法覆盖

- 如果子类不需使用从父类继承来的方法的功能，则可以声明自己的同名方法，称为方法覆盖
- 覆盖方法的方法名称，参数的个数及类型必须和被覆盖的方法一模一样，且返回类型必须要兼容（子类方法的返回类型和父类方法的返回类型相同，或者是父类方法返回类型的子类）
- 覆盖方法的访问权限可以比被覆盖的宽松，但是不能更为严格
- 只需在方法名前面使用不同的类名或不同类的对象名即可区分覆盖方法和被覆盖方法



## 4.1.3 隐藏和覆盖(续)

### ——方法覆盖的应用场合

类的继承

- 方法覆盖的应用场合
  - 子类中实现与父类相同的功能，但采用不同的算法或公式
  - 在名字相同的方法中，要做比父类更多的事情
  - 在子类中需要取消从父类继承的方法



## 4.1.3 隐藏和覆盖(续)

### ——方法覆盖的注意事项

类的继承

- 必须覆盖的方法
  - 派生类必须覆盖基类中的抽象的方法，否则派生类自身也成为抽象类。
- 不能覆盖的方法
  - 基类中声明为**final**的终结方法



## 4.1.3 隐藏和覆盖(续)

### ——Super关键字

类的继承

```
class Superclass {  
    Number aNumber;  
}
```

```
class Subbie extends Superclass{  
    Float aNumber;  
}
```

- 从Subbie中访问Superclass的成员aNumber  
    `super.aNumber`
- 调用被覆盖的方法  
    `super.overriddenMethodName();`





## 4.1.4 有继承时的构造方法

### 类的继承

- 有继承时的构造方法遵循以下的原则
  - 子类不能从父类继承构造方法
  - 好的程序设计方法是在子类的构造方法中调用某一个父类构造方法，调用语句必须出现在子类构造方法的第一行，可使用`super`关键字
  - 如子类构造方法的声明中没有明确调用父类构造方法，则系统在执行子类的构造方法时会自动调用父类的缺省构造方法（即无参的构造方法），这也是为什么我们前面说要么不定义构造方法，要么至少定义两个构造方法，其中一个是无参的构造方法。
  - 如果父类的构造方法可见性是`private`，则子类构造方法中无法调用父类的这个构造方法

## 4.1.4 有继承时的构造方法(续)

### ——例4\_5

```
class Person
{
    protected String name, phoneNumber, address;
    public Person()
    {
        this("", "", "");
    }
    public Person(String aName, String aPhoneNumber, String anAddress)
    {
        name=aName;
        phoneNumber=aPhoneNumber;
        address=anAddress;
    }
}
```



## 4.1.4 有继承

```
class Person
{
    protected String name, phoneNumber, address;
    public Person()
    {
        this("", "", "");
    }
    public Person(String aName, String aPhoneNumber, String anAddress)
    {
        name=aName;
        phoneNumber=aPhoneNumber;
        address=anAddress;
    }
}
```

```
class Employee extends Person
{
    protected int employeeNumber;
    protected String workPhoneNumber;
    public Employee()
    {
        //此处隐含调用构造方法 Person()
        this(0, "");
    }
    public Employee(int aNumber, String aPhoneNumber)
    {
        //此处隐含调用构造方法 Person()
        employeeNumber=aNumber;
        workPhoneNumber = aPhoneNumber;
    }
}
```



```

class Person
{
    protected String name, phoneNumber, address;
    public Person()
    {
        this("", "", "");
    }
    public Person(String aName, String aPhoneNumber, String anAddress)
    {
        name=aName;
        phoneNumber=aPhoneNumber;
        address=anAddress;
    }
}

```

```

class Employee extends Person
{
    protected int employeeNumber;
    protected String workPhoneNumber;
    public Employee()
    {
        //此处隐含调用构造方法 Person()
        this(0, "");
    }
    public Employee(int aNumber, String aPhoneNumber)
    {
        //此处隐含调用构造方法 Person()
        employeeNumber=aNumber;
        workPhoneNumber = aPhoneNumber;
    }
}

```

```

class Professor extends Employee
{
    protected String research;
    public Professor()
    {
        super(); //调用超类的构造方法
        research = "";
    }
    public Professor(int aNumber, String aPhoneNumber, String aResearch)
    {
        super(aNumber, aPhoneNumber); //调用超类的构造方法
        research = aResearch;
    }
}

```



## 4.1.4 有继承时的

试试看：运行结果？

输出结果

Art constructor

Drawing constructor

Cartoon constructor

类的继承

```
class Art
{
    Art() {System.out.println("Art constructor");}
}

class Drawing extends Art
{
    Drawing() {System.out.println("Drawing constructor");}
}

public class Cartoon extends Drawing
{
    Cartoon() {System.out.println("Cartoon constructor");}
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```



## 4.1.4 有继承时

### ——有

类的继承

```
class Game
{
    Game(int i){
        System.out.println("Game constructor");
    }
}

class BoardGame extends Game
{
    BoardGame(int i){
        super(i);
        System.out.println("BoardGame constructor");
    }
}

public class Chess extends BoardGame
{
    Chess(){
        super(11);
        System.out.println("Chess constructor");
    }
    public static void main(String[] args){
        Chess x = new Chess();
    }
}
```

输出结果

Game constructor

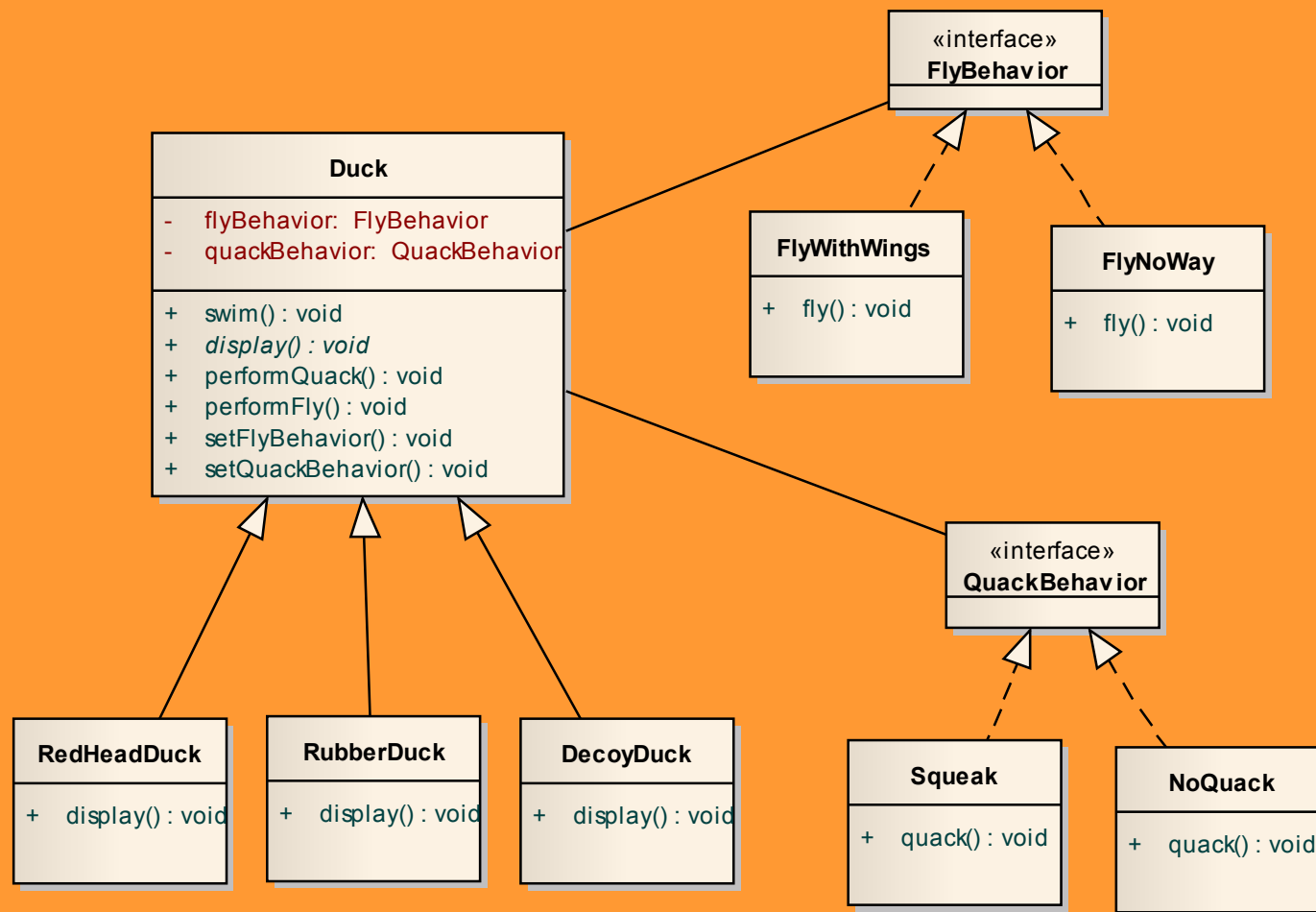
BoardGame constructor

Chess constructor



## 4.1.5 应用举例

class duck

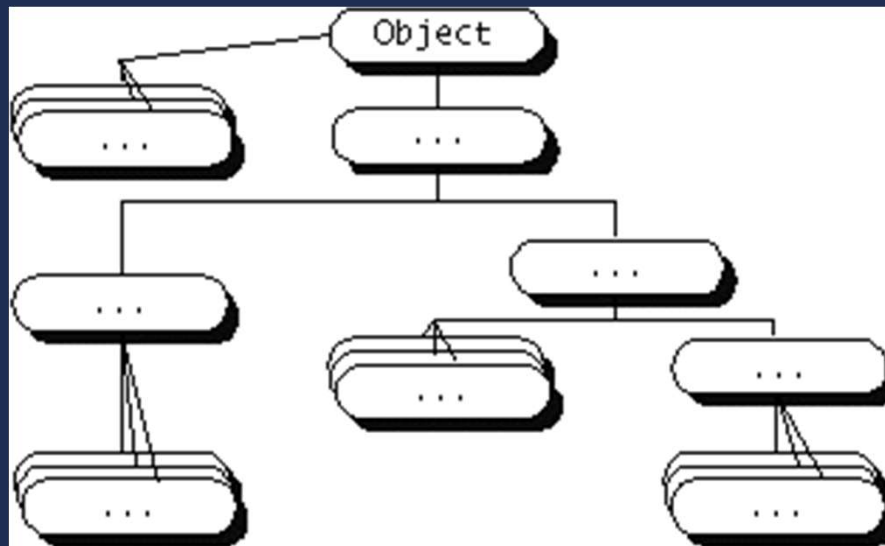


Ducks.rar

## 4.2 Object 类

- Object类

- Java程序中所有类的直接或间接父类，类库中所有类的父类，处在类层次最高点
- 包含了所有Java类的公共属性，其构造方法是Object( )





## 4.2 Object类(续)

### ——包含的主要方法

#### Object 类

- **Object**类定义了所有对象必须具有的状态和行为，较主要的方法如下
  - **public final Class getClass()**
    - 获取当前对象所属的类信息，返回Class对象
  - **public String toString()**
    - 返回当前对象本身的有关信息，按字符串对象返回
  - **public boolean equals(Object obj)**
    - 比较两个对象是否是同一对象，是则返回true
  - **protected Object clone( )**
    - 生成当前对象的一个拷贝，并返回这个复制对象
  - **Public int hashCode()**
    - 返回该对象的哈希代码值
  - **protected void finalize() throws Throwable**
    - 定义回收当前对象时所需完成的资源释放工作
- 你的类不可以覆盖终结方法，即有**final**修饰的方法



## 4.2 Object类(续)

### ——相等和同一

#### Object类

- 相等和同一的概念
  - 两个对象具有相同的类型，及相同的属性值，则称二者相等(equal)
  - 如果两个引用变量指向的是同一个对象，则称这两个变量(对象)同一(identical)
  - 两个对象同一，则肯定相等
  - 两个对象相等，不一定同一
  - 等号“==”判断的是这两个对象是否同一

## 4.2 Object类(续)

### ——例4\_7

#### Object类

- 判断两个对象是否同一

```
public class Exam4_7{  
    public static void main(String args[]){  
        BankAccount a = new BankAccount("Bob", 123456, 100.00f);  
        BankAccount b = new BankAccount("Bob", 123456, 100.00f);  
        if (a == b)  
            System.out.println("YES");  
        else  
            System.out.println("NO");  
    }  
}
```

- BankAccount**类在2.5.1中声明，此程序运行的结果为“NO”，原因是使用等号“==”判断的是两个对象是否同一，显然a和b是两个对象

## 4.2 Object类(续)

### ——例4\_7

#### Object类

- 修改刚才的程序

```
public class Exam4_7{
    public static void main(String args[]){
        BankAccount a = new BankAccount("Bob", 123456, 100.00f);
        BankAccount b = a;
        if (a == b)
            System.out.println("YES");
        else
            System.out.println("NO");
    }
}
```

- 将**a**所指对象的地址赋给**b**，因此**a**与**b**指向的是同一个对象，**a**与**b**同一。输出结果为“YES”

## 4.2 Object类(续)

### ——equals方法

#### Object 类

- equals 方法
  - 由于Object是类层次结构中的树根节点，因此所有其他类都继承了equals()方法
  - Object类中的 equals() 方法的定义如下，可见，也是判断两个对象是否同一

```
public boolean equals(Object x) {  
    return this == x;  
}
```



## 4.2 Object类(续)

### ——例4\_8

- Object类中equals方法的使用举例

```
public class EqualsTest{  
    public static void main(String args[]){  
        BankAccount a = new BankAccount("Bob", 123456, 100.00f);  
        BankAccount b = new BankAccount("Bob", 123456, 100.00f);  
        if (a.equals(b))  
            System.out.println("YES");  
        else  
            System.out.println("NO");  
    }  
}
```

- 由于不是同一对象，运行结果仍然是“NO”



## 4.2 Object类(续)

### ——equals方法 的重写

Object类

- equals方法的重写
  - 要判断两个对象各个属性域的值是否相同，则不能使用从Object类继承来的equals方法，而需要在类声明中对equals方法进行重写
  - String类中已经重写了Object类的Equals方法，可以判别两个字符串是否内容相同



## 4.2 Object类(续)

### ——equals方法 的重写

- 在BankAccount类中增加equals方法，由于是对Object类中的equals方法进行重写，因此方法定义头必须与Object类中的equals方法完全相同

Object类

```
public boolean equals(Object x) {  
    if (this.getClass() != x.getClass())  
        return false;  
    BankAccount b = (BankAccount) x;  
    return  
        ((this.getOwnerName().equals(b.getOwnerName()))  
        &&(this.getAccountNumber() == b.getAccountNumber())  
        &&(this.getBalance() == b.getBalance()));  
}
```





## 4.2 Object类(续)

### ——例4\_9

#### Object 类

- equals方法的应用举例

```
public class Apple {  
    private String color;  
    private boolean ripe;  
    public Apple(String aColor, boolean isRipe) {  
        color = aColor;  
        ripe = isRipe;  
    }  
    public void setColor(String aColor) { color = aColor; }  
    public void setRipe(boolean isRipe) { ripe = isRipe; }  
    public String getColor() { return color; }  
    public boolean getRipe() { return ripe; }  
    public String toString() {  
        if (ripe) return("A ripe " + color + " apple");  
        else return("A not so ripe " + color + " apple");  
    }  
}
```



## ● 运行结果

- A ripe red apple is equal to A ripe red apple: true
- a is identical to b: false
- A ripe red apple is equal to A ripe red apple: true
- a is identical to c: true

```
public boolean equals(Object obj) {  
    if (obj instanceof Apple) {  
        Apple a = (Apple) obj;  
        return (color.equals(a.getColor()) && (ripe == a.getRipe()));  
    }  
    return false;  
}  
}  
}  
public class AppleTester {  
    public static void main(String args[]) {  
        Apple a = new Apple("red", true);  
        Apple b = new Apple("red", true);  
        System.out.println(a + " is equal to " + b + ": " + a.equals(b));  
        System.out.println("a is identical to b: " + (a == b));  
        Apple c = a;  
        System.out.println(a + " is equal to " + c + ": " + a.equals(c));  
        System.out.println("a is identical to c: " + (a == c));  
    }  
}
```

## 4.2 Object类(续)

### ——Clone方法

#### Object类

- Clone方法
  - 根据已存在的对象构造一个新的对象
  - 在根类Object 中被定义为protected, 所以需要覆盖为public
  - 实现Cloneable 接口, 赋予一个对象被克隆的能力(cloneability)

```
class MyObject implements Cloneable
{ //...
}
```



## 4.2 Object类(续)

### ——finalize方法

---

Object  
类

- **finalize方法**

- 在对象被垃圾回收器回收之前，系统自动调用对象的**finalize**方法
- 如果要覆盖**finalize**方法，覆盖方法的最后必须调用**super.finalize**



## 4.2 Object类(续)

### ——getClass方法

#### Object类

- **getClass**方法
  - **final** 方法，返回一个**Class**对象，用来代表对象隶属的类
  - 通过**Class** 对象，你可以查询**Class**对象的各种信息：比如它的名字，它的基类，它所实现接口的名字等。

```
void PrintClassName(Object obj) {  
    System.out.println("The Object's class is " +  
        obj.getClass().getName());  
}
```



## 4.2 Object类(续)

### ——notify、notifyAll、wait方法

---

Object  
类

- notify、notifyAll、wait方法
  - final方法，不能覆盖
  - 这三个方法主要用在多线程程序中



## 4.3 终结类与终结方法

---

- 终结类与终结方法
  - 被final修饰符修饰的类和方法
  - 终结类不能被继承
  - 终结方法不能被当前类的子类重写



## 4.3.1 终结类

### 终结类与终结方法

- 终结类的特点
  - 不能有派生类
- 终结类存在的理由
  - 安全: 黑客用来搅乱系统的一个手法是建立一个类的派生类, 然后用他们的类代替原来的类
  - 设计: 你认为你的类是最好的或从概念上你的类不应该有任何派生类





## 4.3.1 终结类(续)

### ——一个例子

终  
结  
类  
与  
终  
结  
方  
法

- 声明**ChessAlgorithm** 类为**final** 类

```
final class ChessAlgorithm { ... }
```

- 如果写下如下程序:

```
class BetterChessAlgorithm extends ChessAlgorithm  
{ ... }
```

编译器将显示一个错误

```
Chess.java:6: Can't subclass final classes: class  
ChessAlgorithm
```

```
class BetterChessAlgorithm extends ChessAlgorithm {
```

^

1 error



## 4.3.2 终结方法

### 终结类与终结方法

- 终结方法的特点
  - 不能被派生类覆盖
- 终结方法存在的理由
  - 对于一些比较重要且不希望子类进行更改的方法，可以声明为终结方法。可防止子类对父类关键方法的错误重写，增加了代码的安全性和正确性
  - 提高运行效率。通常，当java运行环境（如java解释器）运行方法时，它将首先在当前类中查找该方法，接下来在其超类中查找，并一直沿类层次向上查找，直到找到该方法为止



## 4.3.1 终结方法(续)

### ——例4\_10

终  
结  
类  
与  
终  
结  
方  
法

- **final** 方法举例

```
class Parent
{
    public Parent() { } //构造方法
    final int getPI() { return Math.PI; } //最终方法
}
```

- 说明

- **getPI()**是用**final**修饰符声明的终结方法，不能在子类中对该方法进行覆盖，因而如下声明是错的

```
Class Child extends Parent
{
    public Child() { } //构造方法
    int getPI() { return 3.14; } //重写父类中的终结方法，不允许
}
```

## 4.4 抽象类

---

- 抽象类
  - 代表一个抽象概念的类
  - 没有具体实例对象的类，不能使用new方法进行实例化
  - 类前需加修饰符abstract
  - 可包含常规类能够包含的任何东西，例如构造方法，非抽象方法
  - 也可包含抽象方法，这种方法只有方法的声明，而没有方法的实现



## 4.4 抽象类(续)

### ——存在意义

#### 抽象类

- 存在意义
  - 抽象类是类层次中较高层次的概括，抽象类的作用是让其他类来继承它的抽象化的特征
  - 抽象类中可以包括被它的所有子类共享的公共行为和公共属性
  - 在程序中不能用抽象类作为模板来创建对象；
  - 在用户生成实例时强迫用户生成更具体的实例，保证代码的安全性



## 4.4 抽象类(续)

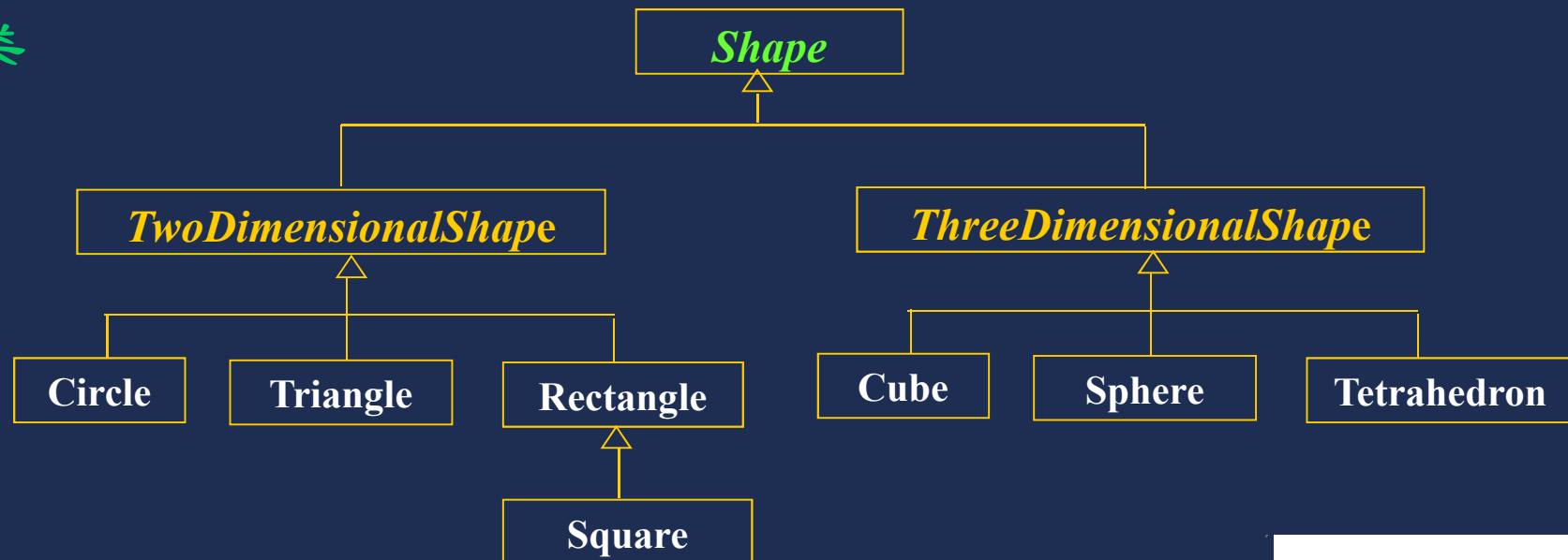
### ——几何形状的例子

- 将所有图形的公共属性及方法抽象到抽象类**Shape**。再将**2D**及**3D**对象的特性分别抽取出来，形成两个抽象类

**TwoDimensionalShape**及**ThreeDimensionalShape**

- 2D图形包括Circles、Triangles、Rectangles和Squares
- 3D图形包括Cube、Sphere、或Tetrahedron
- 在UML中，抽象类的类名为斜体，以与具体类相区别

抽象类

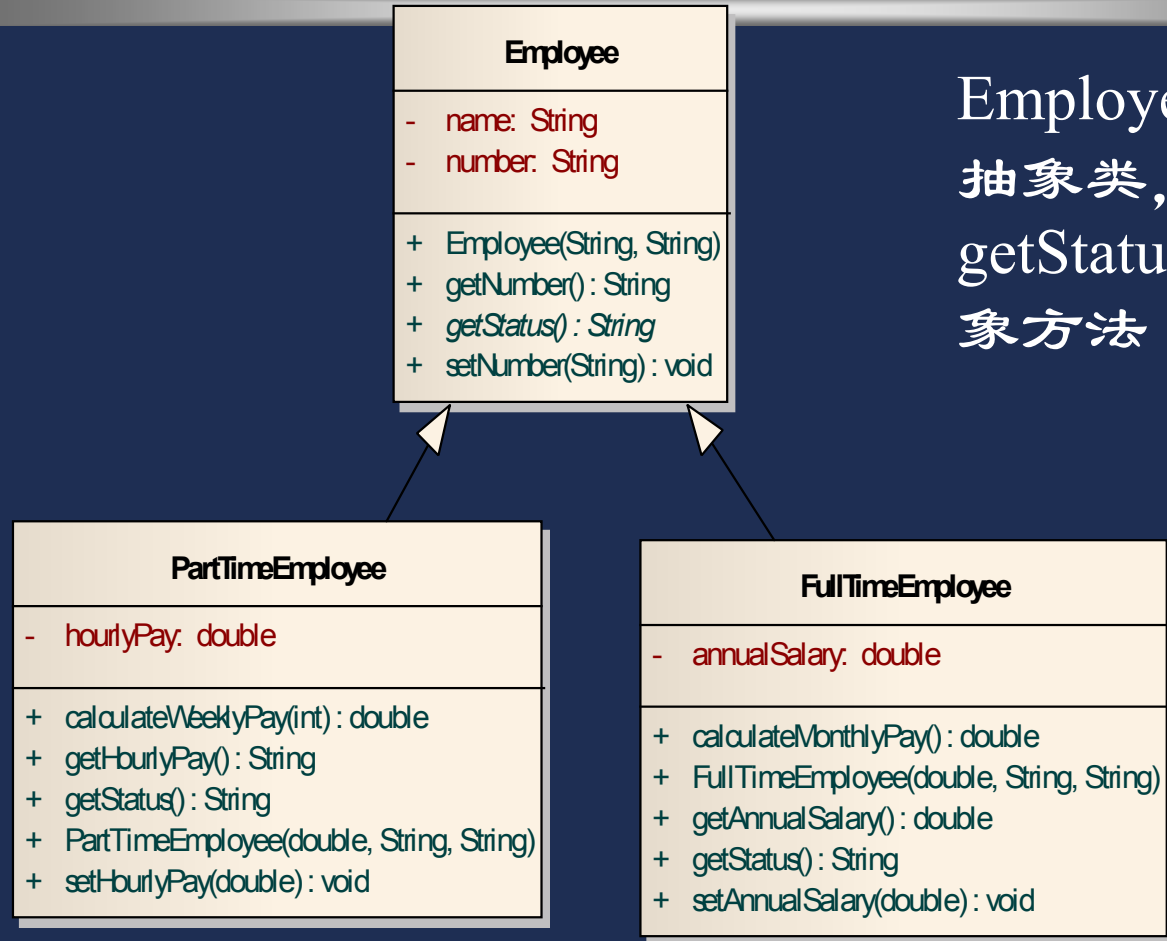


## 4.4 抽象类(续)

### 雇员例子

class 继承

抽象类



Employee是  
抽象类，  
getStatus是抽  
象方法



## 4.4.1 抽象类的声明

---

- 抽象类声明的语法形式为

```
abstract class Employee{
```

```
    ...
```

```
}
```

抽象类

如果写：

```
new Employee();
```

编译器将显示错误（抽象类不能用来实例化对象）





## 4.4.2 抽象方法

- 抽象方法

- 声明的语法形式为

```
public abstract <returnType> <methodName>(...);
```

- 仅有方法头，而没有方法体和操作实现
- 具体实现由当前类的不同子类在它们各自的类声明中完成
- 抽象类可以包含抽象方法

抽象类



# Employee 定义

---

```
public abstract class Employee
{
    private String number;
    private String name;

    public Employee(String numberIn, String nameIn)
    {
        number = numberIn;
        name = nameIn;
    }
    public void setName(String nameIn)
    {
        name = nameIn;
    }
}
```



# Employee 定义

---

```
public String getNumber()  
{  
    return number;  
}  
public String getName()  
{  
    return name;  
}  
abstract public String getStatus(); // an abstract method  
}
```



# PartTimeEmployee 定义

---

```
public class PartTimeEmployee extends Employee
{
    private double hourlyPay;

    public PartTimeEmployee(String numberIn, String nameIn, double
hourlyPayIn)
    {
        super(numberIn, nameIn);//调用父类构造方法
        hourlyPay = hourlyPayIn;
    }

    public void setHourlyPay(double hourlyPayIn)
    {
        hourlyPay = hourlyPayIn;
    }
}
```



# PartTimeEmployee 定义

```
public double getHourlyPay()
{
    return hourlyPay;
}

public double calculateWeeklyPay(int noOfHoursIn)
{
    return hourlyPay * noOfHoursIn;
}
// the abstract method is overridden
public String getStatus()
{
    return "Part-Time";
}
}
```



# FullTimeEmployee定义

---

```
public class FullTimeEmployee extends Employee
{
    private double annualSalary;

    public FullTimeEmployee(String numberIn,
                            String nameIn, double salaryIn)
    {
        super(numberIn,nameIn);
        annualSalary = salaryIn;
    }
    public void setAnnualSalary(double salaryIn)
    {
        annualSalary = salaryIn;
    }
}
```



# FullTimeEmployee定义

---

```
public double getAnnualSalary()
{
    return annualSalary;
}
public double calculateMonthlyPay()
{
    return annualSalary/12;
}
// the abstract method is overridden
public String getStatus()
{
    return "Full-Time";
}
}
```



## 4.4.2 抽象方法(续)

### ——需注意的问题

- 需注意的问题

- 一个抽象类的子类如果不是抽象类，则它必须为父类中的所有抽象方法书写方法体，即重写父类中的所有抽象方法
- 只有抽象类才能具有抽象方法，即如果一个类中含有抽象方法，则必须将这个类声明为抽象类
- 除了抽象方法，抽象类中还可以包括非抽象方法

抽象类





## 4.4.2 抽象方法(续)

### ——抽象方法的优点

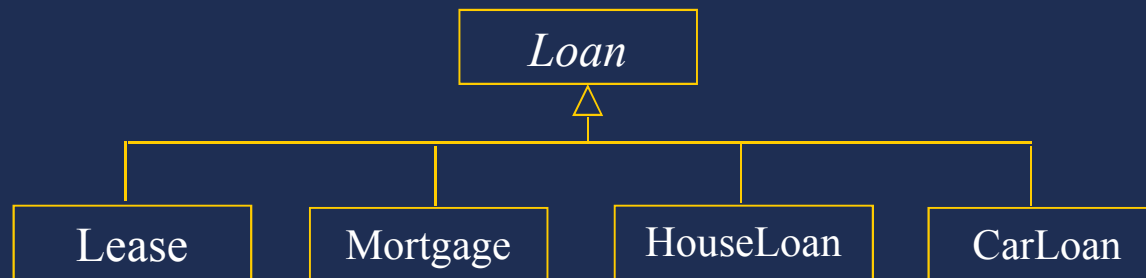
抽象类

- 抽象方法的优点
  - 隐藏具体的细节信息，使调用该方法的程序不必过分关注该类和它的各子类的内部状况。所有的子类使用的都是相同的方法头，而方法头里实际包含了调用该方法的程序语句需要了解的全部信息
  - 强迫子类完成指定的行为，抽象类的所有非抽象子类都必须完成其父类中声明的抽象方法，抽象类通常声明抽象方法规定其子类需要用到的“标准”行为



## 4.4.2 抽象方法(续)

### ——例4\_11



#### 抽象类

- 贷款(Loan)分为许多种类, 如租借(Lease)、抵押(Mortgage)、房屋贷款(HouseLoan)、汽车贷款(CarLoan)等
- 将Loan声明为抽象类, 并指定所有的子类对象都应具有的行为, 如计算月还款值(**calculateMonthlyPayment**), 还款(**makePayment**), 取得客户信息(**getClientInfo**), 其中前两个因贷款种类不同计算方法也不同, 可声明为抽象方法, Loan的所有子类都必须对这两个抽象方法进行重写

```
public abstract class Loan {
    public abstract float calculateMonthlyPayment();
    public abstract void makePayment(float amount);
    public Client getClientInfo() { }
}
```

## 4.5 类的组合

---

- 类的组合

- 面向对象编程的一个重要思想就是用软件对象来模仿现实世界的对象
  - 现实世界中，大多数对象由更小的对象组成
  - 与现实世界的对象一样，软件中的对象也常常是由更小的对象组成
- **Java**的类中可以有其他类的对象作为成员，这便是类的组合



## 4.5.1 组合的语法

### 类的组合

- 组合的语法很简单，只要把已存在类的对象放到新类中即可
- 可以使用“**has a**”语句来描述这种关系
- 例如，考虑**Kitchen**类提供烹饪和冷藏食品的功能，很自然的说“my kitchen '**has a**' cooker/refrigerator”。所以，可简单的把对象**myCooker**和**myRefrigerator**放在类**Kitchen**中。格式如下

```
class Cooker{ // 类的语句 }  
class Refrigerator{ // 类的语句}  
class Kitchen{  
    Cooker myCooker;  
    Refrigerator myRefrigerator;  
}
```



## 4.5.1 组合的语法(续)

### ——例4\_12

类的  
组  
合

- 一条直线是由两个点组成的

```
public class Point //点类
{
    private int x, y; //coordinate
    public Point(int x, int y) { this.x = x; this.y = y;}
    public int GetX() { return x; }
    public int GetY() { return y; }
}
```

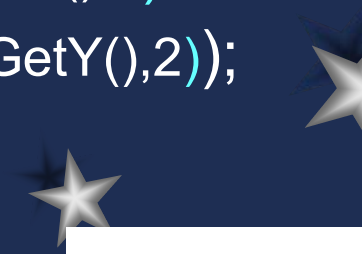


## 4.5.1 组合的语法(续)

### ——例4\_12

类的组合

```
class Line //线类
{
    private Point p1,p2; // 两 endpoints
    Line(Point a, Point b) {
        p1 = new Point(a.GetX(),a.GetY()); //可否写成p1=a?
        p2 = new Point(b.GetX(),b.GetY());
    }
    public double Length() {
        return Math.sqrt(Math.pow(p2.GetX()-p1.GetX(),2)
            + Math.pow(p2.GetY()-p1.GetY(),2));
    }
}
```



## 4.5.2 组合与继承的比较

### 类的组合

- 组合与继承的比较
  - “包含”关系用合成来表达
    - 如果新类内部想利用一个现有类的特性，而不想使用它的接口，通常应选择组合，我们需在新类里嵌入现有类的`private`对象
    - 如果想让类用户直接访问新类的组合成分，需要将成员对象的属性变为`public`
  - “属于”关系用继承来表达
    - 取得一个现成的类，并制作它的一个特殊版本。通常，这意味着我们准备使用一个常规用途的类，并根据特定需求对其进行定制

## 4.5.2 组合与 继承的比较(续)

### ——Car的例子

类的组合

- **car**（汽车）对象是一个很好的例子，由于汽车的装配是故障分析时需要考虑的一项因素，所以有助于客户程序员理解如何使用类，而且类创建者的编程复杂程度也会大幅度降低





**class Engine**

```
{    //发动机类
    public void start() {}
    public void rev() {}
    public void stop() {}
}
```

**class Window**

```
{    //车窗类
    public void rollup() {}
    public void rolldown() {}
}
```

**class Wheel**

```
{    //车轮类
    public void inflate(int psi) {}
}
```

**class Door**

```
{    //车门类
    public Window window = new Window();
    public void open() {}
    public void close() {}
}
```

**public class Car**

```
{
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door left = new Door(), right = new Door();
    public Car()
    {
        for(int i = 0; i < 4; i++)
            wheel[i] = new Wheel();
    }
    public static void main(String[] args)
    {
        Car car = new Car();
        car.left.window.rollup();
        Car.wheel[0].inflate(72);
    }
}
```



## 4.5.3 组合与继承的结合

---

- 许多时候都要求将合成与继承两种技术结合起来使用，创建一个更复杂的类

类的  
组  
合



## 4.5.3 组合与 继承的结合(续)


### ——例4\_13

类的  
组  
合

- 组合与继承举例

```
class Plate { //声明盘子
    public Plate(int i) {
        System.out.println("Plate constructor");
    }
}

class DinnerPlate extends Plate { //声明餐盘为盘子的子类
    public DinnerPlate(int i) {
        super(i);
        System.out.println("DinnerPlate constructor");
    }
}
```



## 4.5.3 组合与 继承的结合(续)

### ——例4\_13

类的  
组  
合

```
class Utensil { //声明器具
    Utensil(int i) {
        System.out.println("Utensil constructor");
    }
}

class Spoon extends Utensil { //声明勺子为器具的子类
    public Spoon(int i) {
        super(i);
        System.out.println("Spoon constructor");
    }
}
```



## 4.5.3 组合与 继承的结合(续)

### ——例4\_13

类的  
组  
合

```
class Fork extends Utensil { //声明餐叉为器具的子类
    public Fork(int i) {
        super(i);
        System.out.println("Fork constructor");
    }
}

class Knife extends Utensil { //声明餐刀为器具的子类
    public Knife(int i) {
        super(i);
        System.out.println("Knife constructor");
    }
}
```



## 4.5.3 组合与 继承

### 类的组合

```
class Custom { // 声明做某事的习惯
    public Custom(int i) { System.out.pr
}

public class PlaceSetting extends Cust
    Spoon sp; Fork frk; Knife kn;
    DinnerPlate pl;
    public PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
        frk = new Fork(i + 3);
        kn = new Knife(i + 4);
        pl = new DinnerPlate(i + 5);
        System.out.println("PlaceSetting constructor");
    }
    public static void main(String[] args) {
        PlaceSetting x = new PlaceSetting(9);
    }
}
```

### ● 运行结果

Custom constructor  
Utensil constructor  
Spoon constructor  
Utensil constructor  
Fork constructor  
Utensil constructor  
Knife constructor  
Plate constructor  
DinnerPlate constructor  
PlaceSetting constructor



## 4.6 包的应用

---

- 包

- 为了解决类名冲突，Java提供包来管理类名空间
- Java利用包来组织相关的类，并控制访问权限
- 同一包中的类在缺省情况下可以互相访问，通常把需要在一起工作的类放在一个包里



## 4.6.1 Java 基础类库简介

- Java基础类库

- Java提供了用于语言开发的类库，称为Java基础类库(JFC, Java Foundational Class)，也称应用程序编程接口(API, Application Programming Interface)，分别放在不同的包中

- Java提供的包主要有

java.lang, java.io, java.math, java.util

java.applet, java.awt, java.awt.datatransfer

java.awt.event, java.awt.image, java.beans

java.net, java.rmi, java.security, java.sql等

包的  
应用





# 4.6.1 Java基础类库简介(续)

## ——语言包

包的  
应用

- 语言包(java.lang)
  - 语言包java.lang提供了Java语言最基础的类，包括
    - Object类
    - 数据类型包裹类(the Data Type Wrapper)
    - 字符串类(String、StringBuffer)
    - 数学类(Math)
    - 系统和运行时类(System、Runtime)
    - 类操作类(Class, ClassLoader)



# 4.6.1 Java基础类库简介(续)

## ——数据类型包裹类

包的  
应用

- 数据包裹类
  - 对应Java的每一个基本数据类型(primitive data type)都有一个数据包裹类
  - 每个包裹类都只有一个类型为对应的基本数据类型的属性域

基本数据类型	数据包裹类
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double



## 4.6.1 Java基础类库简介(续)

### ——生成数据类型包裹类对象的方法

包  
的  
应  
用

- 生成数据类型包裹类对象的方法
  - 从基本数据类型的变量或常量生成包裹类对象  

```
double x = 1.2;  
Double a = new Double(x);  
Double b = new Double(-5.25);
```
  - 从字符串生成包裹类对象  

```
Double c = new Double("-2.34"); //c中存储的是-2.34  
Integer i = new Integer("1234"); //i中存储的是1234
```
  - 已知字符串，可使用valueOf方法将其转换成包裹类对象：  

```
Integer.valueOf("125");  
Double.valueOf("5.15");
```



## 4.6.1 Java基础类库简介(续)

### ——得到基本数据类型数据的方法

- 得到基本数据类型数据的方法

- 每一个包裹类都提供相应的方法将包裹类对象转换回基本数据类型的数据

`anIntegerObject.intValue()` // 返回 `int` 类型的数据

`aCharacterObject.charValue()` // 返回 `char` 类型的数据

- `Integer`、`Float`、`Double`、`Long`、`Byte` 及 `Short` 类提供了特殊的方法能够将字符串类型的对象直接转换成对应的 `int`、`float`、`double`、`long`、`byte` 或 `short` 类型的数据

`Integer.parseInt("234")` // 返回 `int` 类型的数据

`Float.parseFloat("234.78")` // 返回 `float` 类型的数据



# 4.6.1 Java基础类库简介(续)

## ——常量字符串类String

包的  
应用

- **String类**
  - 该类字符串对象的值和长度都不变化
  - 称为常量字符串
- **生成String类对象的方法**
  - 可以这样生成一个常量字符串

```
String aString;  
aString = "This is a string"
```
  - 调用构造方法生成字符串对象

```
new String();  
new String(String value);  
new String(char[] value);  
new String(char[] value, int offset, int count);  
new String(StringBuffer buffer);
```



# 4.6.1 Java基础类库简介(续)

## ——String类的常用方法1

包的应用

名称	解释
<b>int length( )</b>	返回字符串中字符的个数
<b>char charAt(int index)</b>	返回序号index处的字符
<b>int indexOf(String s)</b>	在接收者字符串中进行查找，如果包含子字符串s，则返回匹配的第一个字符的位置序号， 否则返回-1
<b>String substring(int begin, int end)</b>	返回接收者对象中下标从begin开始到end-1的子字符串
<b>String concat(String s)</b>	返回接收者字符串与参数字符串s进行连接后的字符串

# 4.6.1 Java基础类库简介(续)

## ——String类的常用方法2

包的  
应用

名称	解释
<b>String replace(char oldChar, char newChar);</b>	将接收者字符串的oldChar替换为newChar
<b>int compareTo(String s);</b>	将接收者对象与参数对象进行比较
<b>boolean equals(String s);</b>	接收者对象与参数对象的值进行比较
<b>String trim( );</b>	将接收者字符串两端的空字符串都去掉
<b>String toLowerCase()</b>	将接收者字符串中的字符都转为小写
<b>String toUpperCase()</b>	将接收者字符串中的字符都转为大写

# 4.6.1 Java基础类库简介(续)

## ——变量字符串类StringBuffer

包的  
应用

- **StringBuffer类**
  - 其对象是可以修改的字符串
    - 字符的个数称为对象的长度(length)
    - 分配的存储空间称为对象的容量(capacity)
  - 与String类的对象相比，执行效率要低一些
  - 该类的方法不能被用于String类的对象





## 4.6.1 Java基础类库简介(续)

### ——生成StringBuffer类的对象

包的  
应用

- 生成StringBuffer类的对象
  - new StringBuffer();
    - 生成容量为16的空字符串对象
  - new StringBuffer(int size);
    - 生成容量为size的空字符串对象
  - new StringBuffer(String aString);
    - 生成aString的一个备份，容量为其长度 +16



## 4.6.1 Java基础类库简介(续)

### ——StringBuffer类的常用方法1

包的应用

名称	解释
<b>int length ( )</b>	返回字符串对象的长度
<b>int capacity( )</b>	返回字符串对象的容量
<b>void ensureCapacity(int size)</b>	设置字符串对象的容量
<b>void setLength(int len)</b>	设置字符串对象的长度。如果len的值小于当前字符串的长度，则尾部被截掉
<b>char charAt(int index)</b>	返回index处的字符

## 4.6.1 Java基础类库简介(续)

### ——StringBuffer类的常用方法2

包的应用

名称	解释
<b>void setCharAt(int index, char c)</b>	将index处的字符设置为c
<b>void getChars(int start, int end, char [ ] charArray, int newStart)</b>	将接收者对象中从start位置到end-1位置的字符拷贝到字符数组charArray中, 从位置newStart开始存放
<b>StringBuffer reverse( )</b>	返回将接收者字符串逆转后的字符串
<b>StringBuffer insert(int index, Object ob)</b>	将ob插入到index位置
<b>StringBuffer append(Object ob)</b>	将ob连接到接收者字符串的末尾

## 4.6.1 Java基础类库简介(续)

### ——例4\_14

- 已知一个字符串，返回将字符串中的非字母字符都删除后的字符串

```
public class StringEditor {  
    public static String removeNonLetters(String original) {  
        StringBuffer aBuffer = new StringBuffer(original.length());  
        char aCharacter;  
        for (int i=0; i<original.length(); i++) {  
            aCharacter = original.charAt(i);  
            if (Character.isLetter(aCharacter))  
                aBuffer.append(new Character(aCharacter));  
        }  
        return new String(aBuffer);  
    }  
}
```

包的  
应用



## 4.6.1 Java基础类库简介(续)

### ——例4\_14

包的  
应用

```
public class StringEditorTester {  
    public static void main(String args[]) {  
        String original = "Hello123, My Name is Mark,  
                           234I think you are my classmate?!!";  
        System.out.println(  
            StringEditor.removeNonLetters(original));  
    }  
}
```

- 运行结果

HelloMyNameisMarkIthinkyouaremyclassmate



# 4.6.1 Java基础类库简介(续)

## ——数学类(Math)

包的  
应用

- 数学类
  - 提供一组常量和数学函数，包括
    - E和PI常数
    - 求绝对值的abs方法
    - 计算三角函数的sin方法和cos方法
    - 求最小值、最大值的min方法和max方法
    - 求随机数的random方法等
  - 其中所有的变量和方法都是静态的(static)
  - 是终结类(final)，不能从中派生其他的新类



# 4.6.1 Java基础类库简介(续)

## ——系统和运行时类System、Runtime

包  
的  
应  
用

- **System类**

- 访问系统资源

- arraycopy() 复制一个数组
- exit() 结束当前运行的程序
- currentTimeMillis() 返回以毫秒为单位的当前时间

- 访问标准输入输出流

- System.in 标准输入，表示键盘
- System.out 标准输出，表示显示器

- **Runtime类**

- 可直接访问运行时资源

- totalMemory() 返回系统内存总量
- freeMemory() 返回内存的剩余空间



## 4.6.1 Java基础类库简介(续)

### ——类操作类 (Class、ClassLoader)

包的  
应用

- **Class**类

- 提供运行时信息，如名字、类型以及父类
- **Object**类中的**getClass**方法返回当前对象所在的类，返回类型是**Class**
- 它的**getName**方法返回一个类的名称，返回值是**String**
- 它的**getSuperclass**方法可以获得当前对象的父类

- **ClassLoader**类

- 提供把类装入运行时环境的方法





## 4.6.1 Java基础类库简介(续)

——例4\_15

包  
的  
应  
用

- **Class**类应用举例。

```
public class ClassTest {  
    public static void main(String args[]) {  
        BankAccount anAccount = new BankAccount();  
        Class aClass = anAccount.getClass();  
        System.out.println(aClass);  
        System.out.println(aClass.getName());  
    }  
}
```

- 运行结果

```
class BankAccount  
BankAccount
```



# 4.6.1 Java基础类库简介(续)

## ——实用包

### 包的 应用

- 实用包(**java.util**)——实现各种不同实用功能
  - 日期类：描述日期和时间
    - Date
    - Calendar
    - GregorianCalendar
  - 集合类
    - Collection（无序集合）、Set（不重复集合）
    - List（有序不重复集合）、Enumeration（枚举）
    - LinkedList（链表）、Vector（向量）
    - Stack（栈）、Hashtable（散列表）、TreeSet（树）
  - StringTokenizer类
    - 允许以某种分隔标准将字符串分隔成单独的子字符串



# 4.6.1 Java基础类库简介(续)

## ——Date类

包的  
应用

- Date类
  - 构造方法
    - Date() 获得系统当前日期和时间值。
    - Date(long date) 以date创建日期对象，date表示从GMT（格林威治）时间1970-1-1 00:00:00开始至某时刻的毫秒数
  - 常用方法
    - getTime()  
返回一个长整型表示时间，单位为毫秒（millisecond）
    - after(Date d)  
返回接收者表示的日期是否在给定的日期之后
    - before(Date d)  
返回接收者表示的日期是否在给定的日期之前



## 4.6.1 Java基础类库简介(续)

### ——Calendar类

包的  
应用

- Calendar类
  - 一个抽象的基础类，支持将Date对象转换成一系列单个的日期整型数据集，如YEAR、MONTH、DAY、HOUR等常量
  - 它派生的GregorianCalendar类实现标准的Gregorian日历
  - 由于Calendar是抽象类，不能用new方法生成Calendar的实例对象，可以使用getInstance()方法创建一个GregorianCalendar类的对象



## 4.6.1 Java基础类库简介(续)

### ——Calendar类中的常量

包的  
应用

- Calendar类中声明的常量
  - Calendar.SUNDAY
  - Calendar.MONDAY
  - Calendar.TUESDAY
  - Calendar.SATURDAY
  - Calendar.JANUARY
  - Calendar.FEBRUARY
  - Calendar.AM
  - ...



## 4.6.1 Java基础类库简介(续)

### ——Calendar类中的方法

- Calendar类中的方法

- isLeapYear(int year) 返回给定的年份是否是闰年
- get(int field) 取得特定Calendar对象的信息
  - aCalendar.get(java.util.Calendar.YEAR);
  - aCalendar.get(java.util.Calendar.MONTH);
  - aCalendar.get(java.util.Calendar.DAY\_OF\_WEEK);
  - aCalendar.get(java.util.Calendar.MINUTE);
  - ...
- set(int field, int value) 给日期域设定特定的值
  - aCalendar.set(Calendar.MONTH, Calendar.JANUARY);
  - aCalendar.set(1999, Calendar.AUGUST, 15);
  - ...

包  
的  
应  
用



## 4.6.1 Java基础类库简介(续)

### ——GregorianCalendar类

- **Java.util.GregorianCalendar**类

- 用于查询及操作日期
- 构造方法

- new `GregorianCalendar()` // 当前日期
- new `GregorianCalendar(1999, 11, 31)` // 特定日期
- new `GregorianCalendar(1968, 0, 8, 11, 55)` // 日期和时间

- **getTime()**方法 返回**Date**对象，显示日历

- `System.out.println(new GregorianCalendar().getTime());`
- `System.out.println(new GregorianCalendar(1999, 11, 31).getTime());`
- `System.out.println(new GregorianCalendar(1968, 0, 8, 11, 55).getTime());`



## 4.6.1 Java基础类库简介(续)

### ——StringTokenizer类

包的  
应用

- **StringTokenizer类**
  - 允许以某种分隔标准将字符串分隔成单独的子字符串，如可以将单词从语句中分离出来
  - 术语分隔符（**delimiter**）是指用于分隔单词（也称为标记，**tokens**）的字符
  - 常用方法
    - **int countTokens()** 返回单词的个数
    - **String nextToken()** 返回下一个单词
    - **boolean hasMoreTokens()** 是否还有单词





## 4.6.1 Java基础类库简介(续)

### ——生成StringTokenizer类对象的方法

包  
的  
应  
用

- 生成StringTokenizer类的对象的方法
  - new StringTokenizer(String aString);
    - 指定了将被处理的字符串，没有指定分隔符（delimiter），这种情况下缺省的分隔符为空格
  - new StringTokenizer(String aString, String delimiters);
    - 除了指定将被处理的字符串，还指定了分隔符字符串，如分隔符字符串可以为“,:;|\_()”
  - new StringTokenizer(String aString, String delimiters, boolean returnDelimiters);
    - 第三个参数如果为true，则分隔符本身也作为标记返回

```
public class test {  
    public static void main(String[] args) {  
        StringTokenizer s = new  
StringTokenizer("hello  
    boys and girls,welcome", " |,");  
        System.out.println(s.countTokens());  
        while(s.hasMoreTokens())  
            System.out.println(s.nextToken());  
    }  
}
```

输出:

hello  
boys  
and  
girls  
welcome



## 4.6.1 Java基础类库简介(续)

### ——文本包

包的  
应用

- 提供各种文本或日期格式
- 包含
  - Format类
  - DateFormat类
  - SimpleDateFormat类
    - 使用已定义的格式对日期对象进行格式化
    - 构造方法 以一指定格式的字符串作为参数  
`new java.text.SimpleDateFormat(formatString);`
    - `format(Date d)` 将此种格式应用于给定的日期  
`aSimpleDateFormat.format(aDate);`



## 4.6.1 Java基础类库简介(续)

### ——一个例子

- 各种格式字符串及其在日期**2004年4月26**日下午**12:08**的应用结果

包的  
应用

格式字符串	结果字符串
无	" Mon Apr 26 12:08:52 EDT 2004"
"yyyy/MM/dd"	"2004/04/26"
"yy/MM/dd"	"04/04/26"
"MM/dd"	"04/26"
"MMM dd,yyyy"	"Apr 26, 2004"
"MMMM dd,yyyy"	"April 26, 2004"
"EEE. MMMM dd,yyyy"	"Mon. April 26, 2004"
"EEEE, MMMM dd,yyyy"	"Monday, April 26, 2004"
"h:mm a"	"12:08 PM"
"MMMM dd, yyyy (hh:mma)"	"April 26, 2004 (12:08PM)"

## 4.6.2 自定义包

### 包的应用

- 自定义包
  - 包是一组类的集合，利用包来管理类，可实现类的共享与复用
  - 同一包中的类在缺省情况下可以互相访问，通常把需要在一起工作的类放在一个包里
  - 在实际使用中，用户可以将自己的类组织成包结构

## 4.6.2 自定义包(续)

### ——包的声明

#### 包的 应用

- 包的声明

- 包名

- 通常全部用小写字母
    - 且每个包的名称必须是“独一无二”的，为避免包名冲突，可将机构的Internet域名反序，作为包名的前导  
例如：cn.edu.tsinghua.computer.class0301

- 声明语句

package mypackage;

- 说明当前文件中声明的所有类都属于包mypackage
    - 此文件中的每一个类名前都有前缀mypackage，即实际类名应该是mypackage.ClassName，因此不同包中的相同类名不会冲突



## 4.6.2 自定义包(续)

### ——包的声明

#### 包的 应用

- Package语句

- Java源文件的第一条语句， 前面只能有注释或空行
- 一个文件中最多只能有一条
- 如果源文件中没有，则文件中声明的所有类属于一个默认的无名包
- 包声明的语句的完整格式如下：
  - package pkg1[.pkg2[.pkg3...]];
  - Java编译器把包对应于文件系统的目录结构
  - 用点来指明目录的层次



## 4.6.2 自定义包(续)

### ——编译和生成包

包的  
应用

- 编译和生成包
  - 如果在程序Test.java中已声明了包mypackage
    - 编译时采用方式 `javac -d destpath Test.java`
      - 则编译器会自动在destpath目录下建立子目录mypackage，并将生成的.class文件都放到destpath/mypackage下。
    - 不使用d选择符
      - 在当前目录(源程序文件所在的目录)下建立子目录mypackage





## 4.6.2 自定义包(续)

### ——包的使用

包的  
应用

- 假设已定义并生成了下面的包

```
package mypackage;  
public class MyClass {  
    // ...  
}
```

- 如果其他人想使用**MyClass**类

- 使用import语句引入

```
import mypackage.*;  
// ...
```

```
MyClass m = new MyClass();
```

- 不使用import语句，则需要使用全名

```
mypackage.MyClass m = new mypackage.MyClass();
```




## 4.6.2 自定义包(续)

### ——举例

```
package PackageA;
import PackageB.*;
public class A1
{
    public void FunA()
    {
        B1 obj=new B1();
        A2 two=new A2();
    }
}
class A2
{
    public void FunA2()
    {
        A1 obj1=new A1();
        B obj2=new B();
    }
}
```


```
package PackageB;
import PackageA.*;
public class B1
{
    public void FunB()
    {
        A1 obj=new A1();
        B2 two=new B2();
    }
}
class B2
{
    public void FunB2()
    {
        B1 obj1=new B2();
        A1 obj2=new A1();
    }
}
```



## 4.6.2 自定义包(续)

### ——举例

```
import PackageA.*;
import PackageB.*;
public class PackageTest
{
    public static void main(String[ ] args)
    {
        A1 Myobj=new A1();
        System.out.println("An object of class A1
has been created!");
    }
}
```



## 4.6.3 JAR文件

包的  
应用

- JAR文件
  - Java 的一种文档格式
- JAR文件格式
  - 非常类似 ZIP 文件，唯一区别就是在 JAR 文件的内容中包含了
    - META-INF/MANIFEST.MF 文件
      - 在生成 JAR 文件的时候自动创建
    - 部署描述符
      - 用来指示工具如何处理特定的JAR文件



## 4.6.3 JAR文件(续)

### ——JAR文件的功能

#### 包的应用

- JAR文件的功能
  - 压缩和发布文件
  - 相比于ZIP文件，具有如下优势和功能
    - 安全性：可以对 JAR 文件内容加上数字化签名
    - 减少下载时间：浏览器可以在一个 HTTP 事务中下载 JAR文件包含的所有类文件和相关资源
    - 传输平台扩展：Java 扩展框架提供了向 Java 核心平台添加功能的方法，这些扩展是用 JAR 文件打包的
    - 包密封：存储在 JAR 文件中的包可以选择进行密封，以增强版本一致性和安全性
    - 包版本控制：一个 JAR 文件可以包含有关它所包含的文件的数据，如厂商和版本信息
    - 可移植性：处理 JAR 文件的机制是 Java 平台核心 API 的标准部分，因此具有很好的可移植性



## 4.6.3 JAR文件(续)

### ——META-INF目录

#### 包的应用

- META-INF 目录
  - 大多数 JAR 文件包含一个 META-INF 目录
  - 存储包和扩展的配置数据，如安全性和版本信息
  - Java 2 平台识别并解释 该 目录中的下述文件和目录，以便配置应用程序
    - MANIFEST.MF
      - 定义了与扩展和包相关的数据
    - INDEX.LIST
      - 由 jar 工具的新选项 -i 生成，它包含在应用程序或者扩展中定义的包的位置信息。它是 JarIndex 实现的一部分，并由类装载器用于加速类装载过程
    - xxx.SF
      - JAR 文件的签名文件，xxx 标识了签名者
    - xxx.DSA
      - 与签名文件相关联的签名程序块文件，它存储了用于签名 JAR 文件的公共签名

## 4.6.3 JAR文件(续)

### ——jar工具

包的  
应用

- jar 工具
  - 为了用 JAR 文件执行基本的任务，要使用JDK提供的 jar 工具(Java Archive Tool)
  - 随 JDK 安装，在 JDK 安装目录下的 bin 目录中
    - Windows 下文件名为 jar.exe
    - Linux 下文件名为 jar
  - 它的运行需要用到 JDK 安装目录下 lib 目录中的 tools.jar 文件
  - 用 jar 命令调用



## 4.6.3 JAR文件(续)

### ——jar命令格式

- jar命令格式

jar {ctxu} [vfmOM] [jar-文件] [manifest-文件] [-C 目录] 文件名

- 解释

- {ctxu}: jar命令的子命令，每次只能包含其中之一

- -c 创建新的 JAR 文件包
- -t 列出 JAR 文件包的内容列表
- -x 展开 JAR 文件包的指定文件或者所有文件
- -u 更新已存在的 JAR 文件包 (添加文件到 JAR 文件包中)

- [vfmOM]: jar 命令的选项参数，可任选一个或不选

- -v 生成详细报告并打印到标准输出
- -f 指定 JAR 文件名，通常这个参数是必须的
- -m 指定需要包含的 MANIFEST 清单文件
- -O 只存储，不压缩，这样产生的 JAR 文件包会比不用该参数产生的体积大，但速度更快
- -M 不产生所有项的清单 (MANIFEST) 文件，此参数会忽略 -m 参数

包的  
应用



## 4.6.3 JAR文件(续)

### ——jar命令格式

- jar命令格式

jar {ctxu} [vfmOM] [jar-文件] [manifest-文件] [-C 目录] 文件名...

- 解释

包的  
应用

- [jar-文件]

- 需要生成、查看、更新或者解开的 JAR 文件包
- -f 参数的附属参数

- [manifest-文件]

- MANIFEST 清单文件
- -m 参数的附属参数

- [-C 目录]

- 表示转到指定目录下去执行这个 jar 命令的操作

- “文件名 ...”

- 指定要添加到 JAR 文件包中的文件/目录



## 4.6.3 JAR文件(续)

### ——常用jar工具

包的  
应用

命令	功能
<code>jar cf jar-file input-file..</code>	用一个单独的文件创建 JAR 文件
<code>jar cf jar-file dir-name</code>	用一个目录创建 JAR 文件
<code>jar cf0 jar-file dir-name</code>	创建一个未压缩的 JAR 文件
<code>jar uf jar-file input-file...</code>	更新一个 JAR 文件
<code>jar tf jar-file</code>	查看一个 JAR 文件的内容
<code>jar xf jar-file</code>	提取一个 JAR 文件的内容
<code>jar xf jar-file archived-file...</code>	从JAR 文件中提取特定的文件
<code>java -jar app.jar</code>	运行一个打包为可执行 JAR 文件的应用程序

## 4.6.3 JAR文件(续)

### ——可执行的JAR文件包

包  
的  
应  
用

- 可执行的 JAR文件包
  - 一个可执行的 jar 文件是一个自包含的 Java 应用程序，它存储在特别配置的 JAR 文件中
  - 可以由 JVM 直接执行它而无需事先提取文件或者设置类路径，不用知道它的主要入口点
  - 有助于方便发布和执行 Java 应用程序



## 4.6.3 JAR文件(续)

### ——创建和运行可执行JAR

包的  
应用

- 创建一个可执行 JAR
  - 首先将所有应用程序代码放到一个目录中
  - 假设应用程序中的主类是 `com.mycompany.myapp.Sample`  
在某个位置创建一个名为 `manifest` 的文件，并在其中加入以下一行  
`Main-Class:com.mycompany.myapp.Sample`
  - 然后，创建可执行的 JAR 文件包：  
`jar cmf manifest ExecutableJar.jar application-dir`
- 直接从文件启动这个应用程序  
`java -jar ExecutableJar.jar`



## 4.7 本章小结

- 本章内容

- 介绍了Java语言类的重用机制，形式可以是组合或继承
- Object类的主要方法
- 终结类和终结方法的特点和语法
- 抽象类和抽象方法的特点和语法
- Java基础类库的一些重要的类
- JAR文件和jar命令

- 本章要求

- 理解组合和继承的区别，能够知道何时使用那种方法
- 了解终结类、终结方法、抽象类、抽象方法的概念
- 熟练掌握本章提到的Java基础类库中的一些常见类
- 初步了解JAR文件的概念，jar命令的格式