第4章

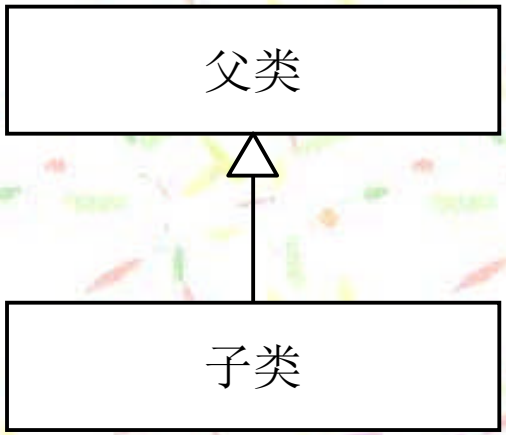
继承与多态

面向对象程序设计

4.1 继承与多态的实现技术

继承

继承是指一个类的定义可以基于另外一个已经存在的类，即子类基于父类，从而实现父类代码的重用。两个类之间的这种继承关系可以用UML图形符号表示：



面向对象程序设计

父类与子类相比较，涵盖了更加共性的内容，更加具有一般性，而子类所添加的内容更加具有个性，是一般性之外的特殊内容，因此，这种类的继承关系充分地反映了类之间的“一般-特殊”关系。

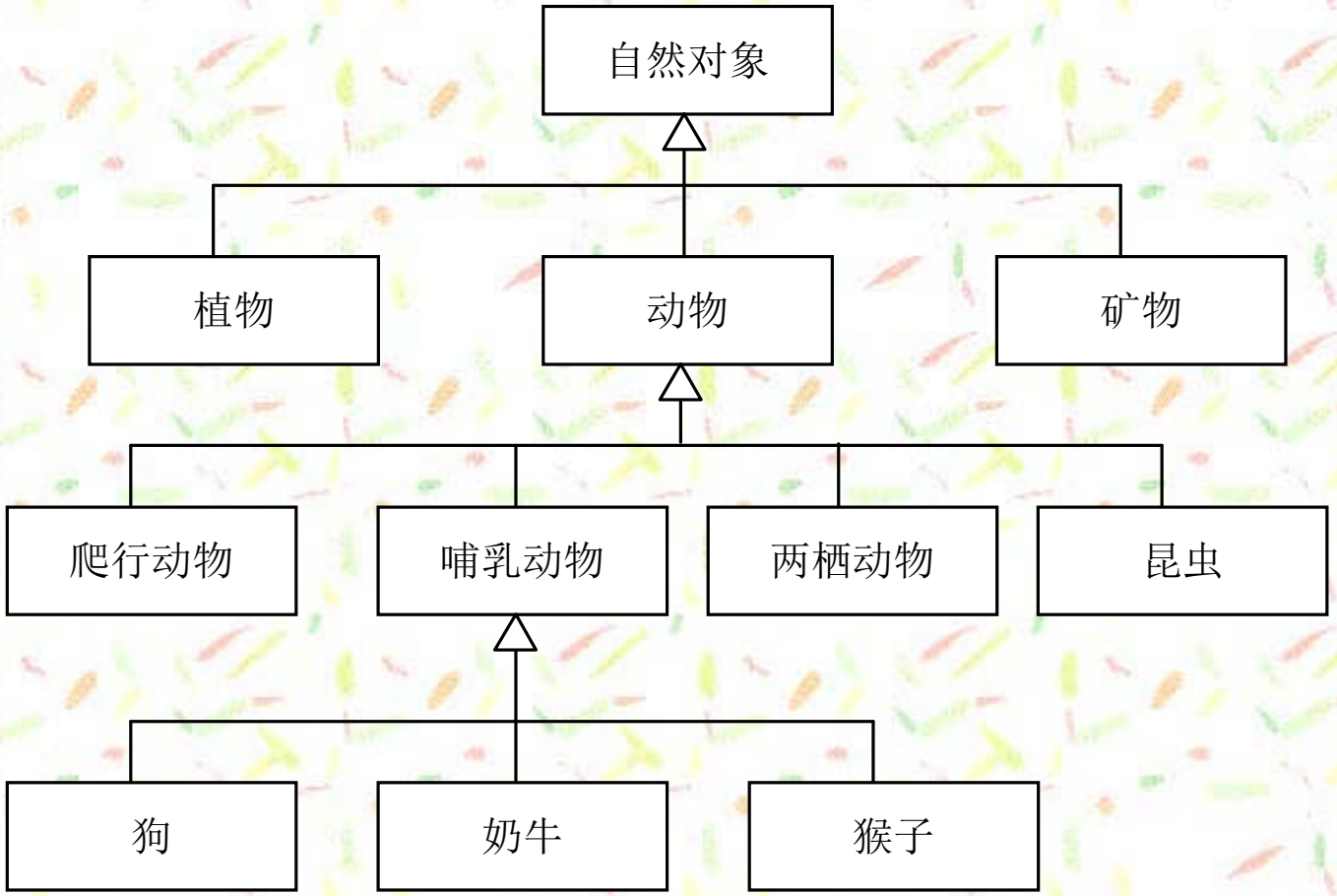
类的继承具有传递性，即子类还可以再派生子类，最终形成一个类层次结构。

在Java语言中，通过定义子类支持继承性。不仅如此，Java还提供了抽象类和接口，以便使类层次得到更高级别的抽象。

多态性

多态性是面向对象程序设计的又一个核心概念，它有助于增加软件系统的可扩展性、自然性和可维护性。所谓多态是指不同的类对象收到同一个消息可以产生完全不同的响应效果的现象。利用多态机制，用户可以发送一个通用的消息给各个类对象，而实现细节由接收对象自行决定，这样，同一个消息可能会导致调用不同的方法。

类层次结构举例



4.2 类的继承

4.2.1 定义子类

子类是通过在定义类时利用关键字extends指出父类实现的，其语法格式为：

```
[Modifier] class ClassName extends SuperClassName
{
    //ClassBody
}
```

Modifier 是类定义修饰符，ClassName 是子类的名称，extends是指出父类的关键字，SuperClassName是直接父类的名称，ClassBody是所有子类成员的定义。

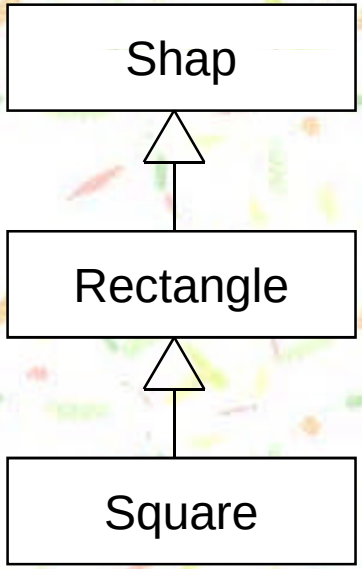
面向对象程序设计

最简单的Applet应用程序的类定义

```
import java.applet.*;
public JavaApplet extends Applet
{
    public void paint(Graphics g)
{
        g.drawString("This is a Applet.",50,50);
    }
}
```

一个有关几何图元处理的例子

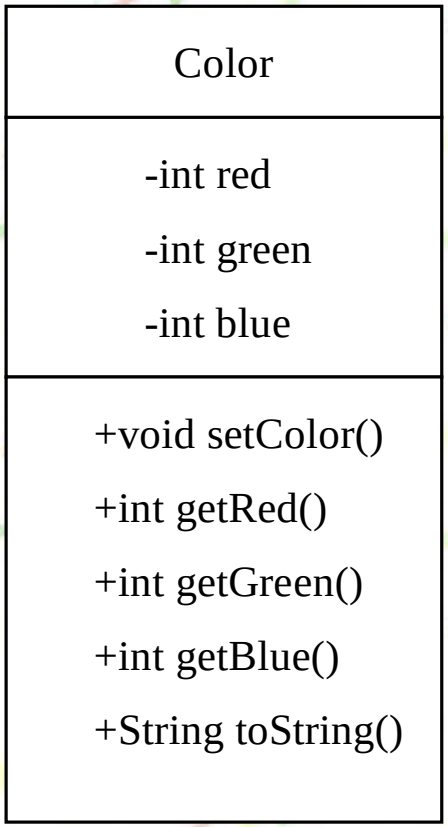
任何一个几何图元都有颜色和原点这两个基本属性。矩形是一种图元，它还有长（long）、宽（width）两个属性，正方形是一种特殊的矩形，它的特殊性在于长和宽相等。这三个类之间的关系可以用图所示的UML类图描述：

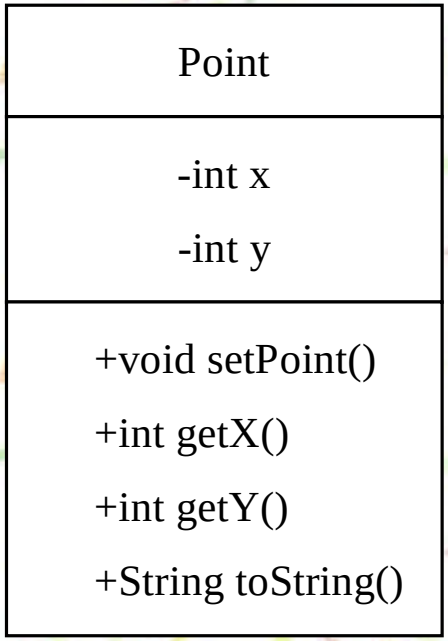


面向对象程序设计

Shape类

设定两个属性：一个是几何图元的颜色因此，我们要定义一个Color类另一个是几何图元的原点，由x和y惟一确定，为此，需要定义一个Point类。





Color类定义

```
public class Color        //Color类定义
{
    private int red;      //红色
    private int green;    //绿色
    private int blue;     //蓝色
    public Color(){red=0;green=0;blue=0;}    //构造方法
      public Color(int red,int green,int blue)     //构造方法
      {
            if (red<0||red>255) this.red=0;
            else this.red=red;
            if (green<0||green>255) this.green=0;
            else this.green=green;
            if (blue<0||blue>255) this.blue=0;
            else this.blue=blue;
      }
```

```
    public void setColor(int red,int green,int blue)    //设置颜色
    {
       if (red<0||red>255) this.red=0;
       else this.red=red;
       if (green<0||green>255) this.green=0;
       else this.green=green;
       if (blue<0||blue>255) this.blue=0;
       else this.blue=blue;
    }
    public int getRed(){return red;}        //获取红色
    public int getGreen(){return green;}    //获取绿色
    public int getBlue(){return blue;}      //获取蓝色
    public String toString()    //将颜色信息转换成字符串描述形式
    {
      return "Red: "+red+"  ,Green: "+green"+"   ,Blue: "+blue;
    }
}
```

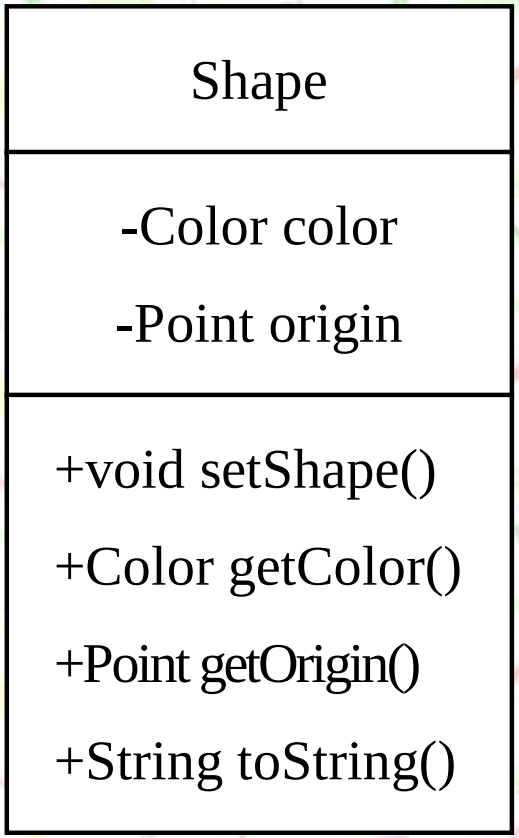
面向对象程序设计

Point类定义

```
public class Point          //Point类定义
{
   private int x,y;          //x和y坐标点
   public Point(){x=0;y=0;}        //构造方法
   public Point(int x,int y) { this.x=x;this.y=y;}      //构造方法
   public Point(Point point)       //构造方法
   {
      x=point.x;
      y=point.y;
   }
   public int getX(){return x;}   //获取x
   public int getY(){return y;}    //获取y
   public void setPoint(int x,int y)     //设置坐标点
   {
      this.x=x<0?0:x;
      this.y=y<0?0:y;
   }
    public String toString() //将坐标信息转换成字符串描述形式
    {
       return "("+x+","+y+")";
       }
}
```

Shape类的定义

Shape类含有两个成员变量，一个是几何图元的颜色color，另一个是几何图元的原点origin



面向对象程序设计

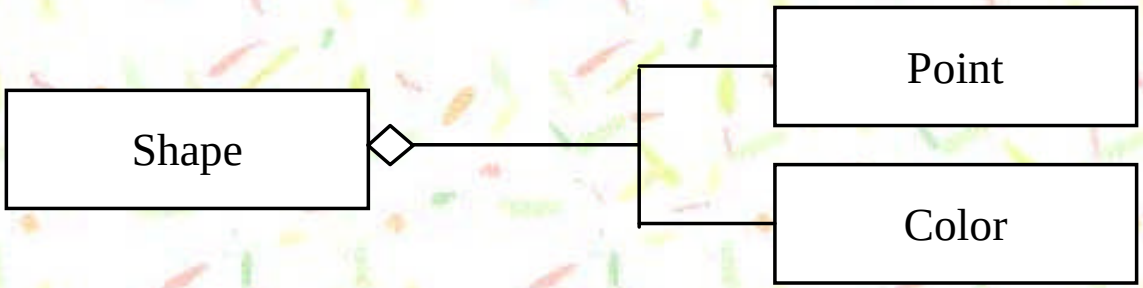
Shape类定义

```
public class Shape   //Shape类定义
{
  private Color color;      //颜色属性
  private Point origin;      //原点属性
  public Shape()      //构造方法
  {
     color=new Color();
     origin=new Point();
  }
  public Shape(Color c,Point o)                    //构造方法
  {
     color=c;  origin=o;
  }
  public void setShape(Color c,Point o)//设置几何图形的颜色和原点
  {
     color=c;  origin=o;
  }
```

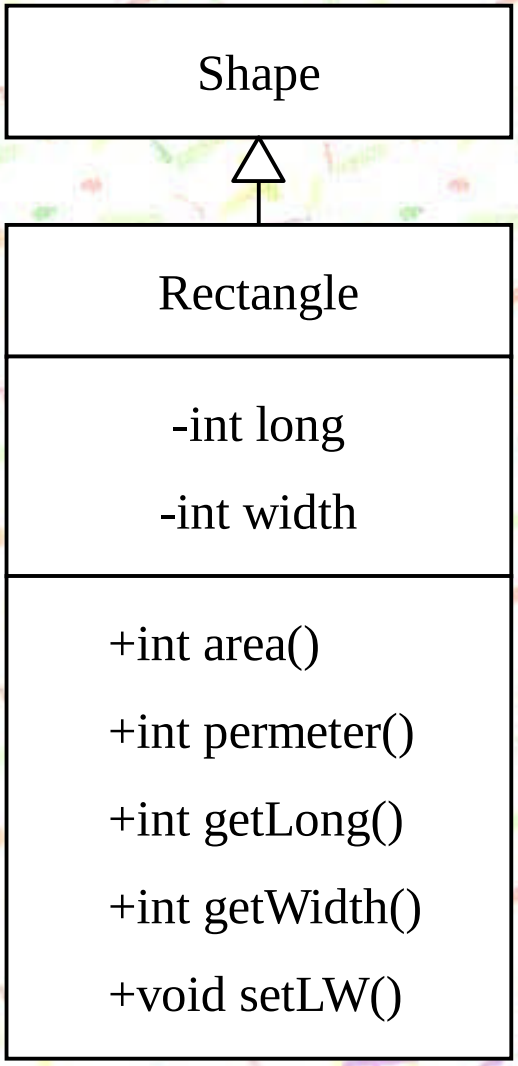
面向对象程序设计

```
    public Color getColor(){return color;}   //获取颜色
    public Point getOrigin(){return origin;}   //获取原点
    public String toString()   //将几何图形的信息转换成字符串描述形式
    {
        return color.toString()+"\n"+origin.toString();
    }
}
```

Shape类由Color和Point类组合而成，属于“整体-部分”关系，若用UML类图描述应如图:



Rectangle类的UML类图描述



面向对象程序设计

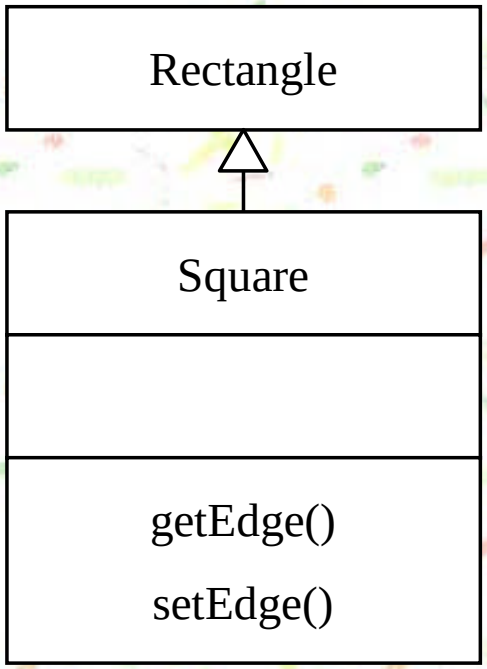
Rectangle类定义

```
public class Rectangle extends Shape      //Rectangle类定义
{
   private int long;       //长
   private int width;      //宽
   public Rectangle()     //构造方法
   {
      super();    //调用父类的构造方法
      long=0;
      width=0;
   }
   public Rectangle(Color c,Point o,int l,int w)    //构造方法
   {
      super(c,o);
      long=l;
      width=w;
   }
   public int getLong(){return long;}      //获取长度
   public int getWidth(){return width;}    //获取宽度
   public void setLW(int l,int w){long=l;width=w;}      //设置长、宽
}
```


面向对象程序设计

Square类

Square类不需要增加新的成员变量，只重新定义一些适用于正方形的成员方法或更改一些成员方法的接口形式以便更加适用于正方形即可。



Square类定义

```
public class Square extends Rectangle      //Square类定义
{
  public Square() { super();}
  public Square(Color c,Point o,int e) {  super(c,o,e,e);  }
  public int getEdge() { return getLong(); }        //获取边长
  public void setEdge(int e)  {   setLW(e,e);   }   //设置边长
}
```

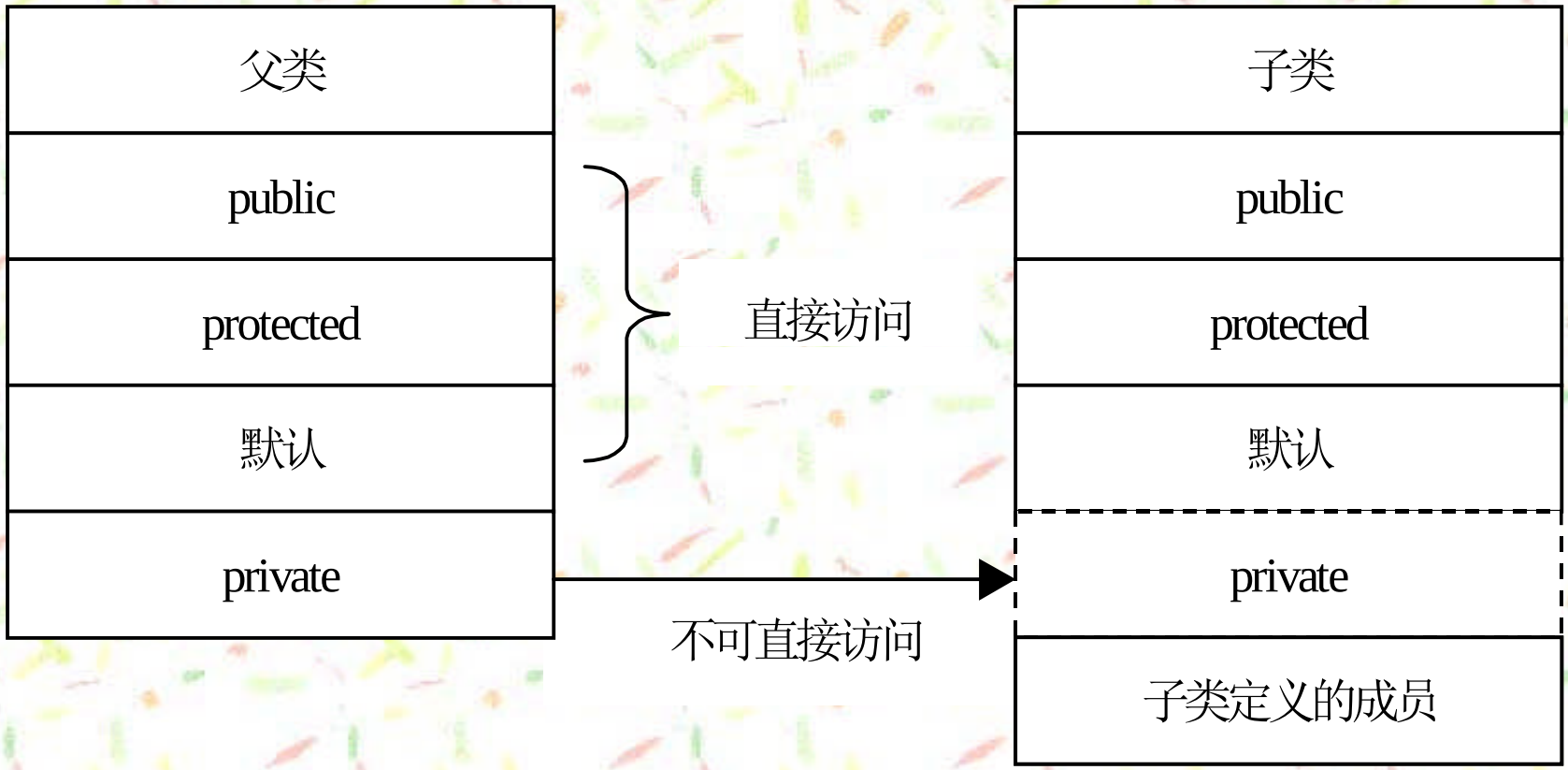
面向对象程序设计

在Java语言中，子类将继承父类的成员，但子类对象对父类成员的可访问性却由访问属性控制。

如果子类与父类在同一个包中，子类可以直接访问父类具有public、proteced和默认访问属性的成员。

如果子类和父类不在同一个包中，子类只能够直接访问父类具有public、proteced访问属性的成员，而具有private和默认访问属性的成员需要通过具有public或protected访问属性的成员方法实现访问目的。

在同一个包中子类访问父类成员的规则



面向对象程序设计

在不同包中子类访问父类成员的规则

面向对象程序设计

4.2.2 子类的构造方法

在Java语言中，子类不负责调用父类中带参数的构造方法。若要在创建子类对象时希望对从父类继承的成员变量初始化，就要在子类的构造方法中利用super()调用父类的构造方法，并且必须将它作为子类构造方法中的第一条语句。如果第一条语句不是调用父类构造方法的语句，系统将自动地插入一条调用父类默认构造方法的语句。由于默认的构造方法不带参数，所以，如果父类定义了带参数的构造方法，而没有定义不带参数的构造方法，将会出现编译错误。

简单的例子

```
public class Animal          //动物类
{
  private String type;          //动物种类
  public Animal(String  type)    //构造方法
  {
     this.type=new String(type);
  }
  public String toString()    //将Animal类对象的内容转换成字符串的
   描述形式
  {
    return "This is a "+type;
  }
}
```

在Animal类的基础上，定义一个Dog类

```
public class Dog extends Animal      //狗类
{
   private String name;
   private String breed;
public Dog(String name)
   {
     super("Dog");       //调用父类的构造方法
     this.name=name;
     breed="Unknow";
   }
   public Dog(String name,String breed)
   {
      super("Dog");            //调用父类的构造方法
      this.name=name;
      this.breed=breed;
   }
}
```

在Dog类中，定义了两个构造方法。为了初始化type属性，需要在构造方法中，调用父类的构造方法，由于它们传递的参数都是“Dog”，所以，不管调用哪个构造方法，type都将被初始化为“Dog”。但是，如果在这两个构造方法中，第一条语句不是调用父类的构造方法，将会出现编译错误，这是因为，此时系统会自动地在第一条语句的位置添加调用父类的默认构造方法的语句，但在Animal类中，不存在不带参数的构造方法。

4.2.3 super

super是Java语言的关键字，用来表示直接父类的引用。前面已经看到，若在子类中调用父类的构造方法，就需要借助于这个关键字。另外，如果在子类中，希望使用父类中的那些被子类覆盖的成员，也需要利用super实现。

4.2.4 通用父类Object

Java语言将所有的类都作为Object类的子类。首先，一个Object类型的变量可以用来引用任何类的对象。当在程序中，处理未知类型的对象时这个功能显得尤为重要；其次，可以将成员方法的参数设置为Object类型，以便方法能够接收参数传递进来的任何类型的对象；最后，在Object类中提供了所有对象都应该具有的行为方法，这样可以更好地统一这些成员方法的接口形式。

Object类中7个常用的public成员方法

成员方法	描述
toString()	该成员方法以String类对象的形式返回当前对象的字符串描述。
equals()	该成员方法通过参数带入一个对象，并将它与当前对象进行比较。
getClass()	该成员方法返回一个Class类对象，该对象内部包含了一些能够标识当前对象的信息。
hashCode()	该成员方法计算一个对象的哈希码，并将其返回。
notify()	该成员方法可以唤醒一个与当前对象关联的线程。
notifyAll()	该成员方法可以唤醒与当前对象关联的所有线程。
wait()	该成员方法将导致线程等待一个指定的时间间隔或等待另一个线程调用当前对象的notify()或notifyAll()方法。

4.3 类成员的隐藏与重载

在子类继承父类成员的同时，子类自己还可以定义一些新的成员。当子类中定义的新成员变量与父类中某个成员变量的名字相同时，子类会将父类相应的成员变量隐藏起来。当子类中定义的成员方法与父类中某个成员方法的签名完全一样时，子类同样将父类的相应成员方法隐藏起来，这种现象被称为覆盖。倘若只是子类中定义的成员方法与父类中某个成员方法的名字相同，则称为重载。

4.3.1 成员变量的继承与隐藏

子类将继承父类中除私有访问属性的所有成员变量，除此之外，子类还可以自行定义一些成员变量，这些新的成员变量有些用来扩展父类的描述细节，有些用来将父类中的某个成员变量隐藏起来，使之更加适于描述特定的对象类型。在程序设计中，这种子类隐藏父类成员变量的形式使用的并不多。如果不是必要，建议不要这样设计成员变量，以便降低程序的可读性，增加系统的资源开销。

4.3.2成员方法的继承、重载与覆盖

子类将继承父类除私有访问属性的所有成员方法，除此之外，子类还可以自行定义一些成员方法，其中主要包括下列几种形式：

- 在父类中没有的、全新的成员方法。这些成员方法将用来扩展父类的接口形式，增加子类对象的操作功能。
- 子类中定义与父类具有相同签名的成员方法。这些成员方法起到了覆盖父类相应成员方法的作用，因此又称为成员方法的覆盖。
- 子类中定义的某个成员方法只是与父类中的某个成员方法的名字相同，称为成员方法的重载。

4.4 多态性的实现

多态性是指不同类的对象调用同一个签名的成员方法，却执行不同的代码段的现象。若要实现多态性，需要具备下面两个条件。

- 多态性作用于子类，它是依赖于类层次结构中的一项新功能。在Java语言中，提供了一个指向父类对象的引用可以被用来指向它的任何子类对象的能力，这是实现多态性的先决条件。
- 若得到多态性的操作，相应的成员方法必须同时包含在父类和子类中，且对应的成员方法签名完全一样，子类中该方法的访问属性不能严于父类中该方法的访问属性。如果这个成员方法在父类中不存在，就不能使用父类型对象引用调用它。

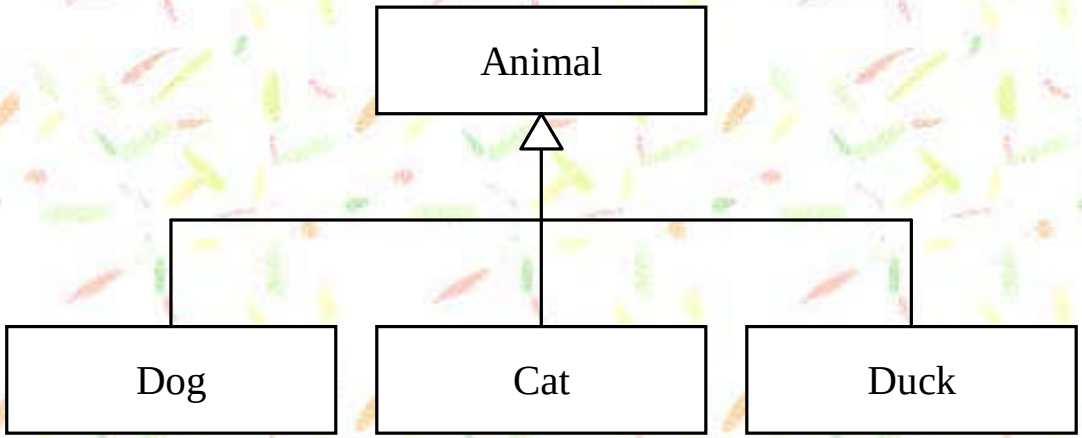
面向对象程序设计

实现多态性需要的基本步骤

- 定义一个父类的引用
- 让该引用指向其子类对象
- 使用该对象调用成员方法

实现多态性的例子

Animal、Dog、Cat、Duck类关系的UML类图



面向对象程序设计

Animal类的定义：

```
import java.util.Random;
class Animal       //动物类
{
   protected String type;      //种类
   protected String name;      //名称
   protected String breed;     //品种
   public Animal(String type,String name,String breed)
   {
     this.type=new String(type);
     this.name=new String(name);
     this.breed=new String(breed);
   }
   public String toString()
   {
     return "This is a "+type+"\nIt's "+name+"   the  "+breed;
    }
    public void sound(){}
}
```

Dog类，Cat类Duck类定义：

```
class Dog extends Animal     //Dog类
{
   public Dog(String name)  { super("Dog",name,"Unknow");   }
   public Dog(String name,String breed) { super("Dog",name,breed);  }
   public void sound() { System.out.println("Woof   Woof");  } //发出狗的叫声
}
class Cat extends Animal      //Cat类
{
   public Cat(String name) { super("Cat",name,"Unknow");   }
   public Cat(String name,String breed) { super("Cat",name,breed);   }
   public void sound()  { System.out.println("Miiaooww");  } //发出猫的叫声
}
class Duck extends Animal     //Duck类
{
   public Duck(String name)  {  super("Duck",name,"Unknow");   }
   public Duck(String name,String breed) { super("Duck",name,breed);  }
   public void sound() { System.out.println("Quack quackquack"); } //发出鸭子的
     叫声
 }
```

面向对象程序设计

Animal是一个具有抽象意义的类。这是因为在现实世界中，每个动物都一定属于某个特定的种类，因此，这个类中的**sound()**成员方法的方法体为空，即本身没有任何特定的操作，只是为了实现多态性而设置。在**Dog**、**Cat**和**Duck**中，覆盖了**Animal**类中的这个成员方法，它们根据具体动物的特点发出不同的叫声。

测试类的定义：

```
public class TryPolymorphism      //测试类
{
    public static void main(String[] args)
    {
        Animal[] theAnimals={    //创建包含Dog、Cat和Duck对象的数组
          new Dog("Rover","Poodle"),
          new Cat("Max","Abyssinian"),
          new Duck("Daffy","Aylesbury")
        };
       Animal petChoice;                  //声明父类的引用
       Random select=new Random();        //创建随机数对象
       for (int i=0;i<5;i++)
       {
          petChoice=theAnimals[select.nextInt(theAnimals.length)];
          System.out.println("\nYour Choice:\n"+petChoice);
          petChoice.sound();
        }
    }
}
```

每次执行sound()成员方法时都将根据petChoice当前引用的对象类型调用相应的代码段，这就是典型的多态性效果。

4.5 抽象类

概念是对类进行高层抽象的结果，可实例化的类是抽象概念的具体表现。在Java语言中，用抽象类表示概念性的事物，用类表示可实例化的实体类别，例如，可以用“学生”类描述“学生”这个抽象的概念，用“小学生”、“中学生”、“大学生”…… 描述具体的学生类别，这些类可以被实例化，是“学

Java语言中抽象类的声明

在Java语言中，抽象类就是用abstract修饰符声明的类。其格式如下所示：

```
abstract class className……
{
  //成员变量和成员方法
}
```

其中的成员方法既可以是抽象方法，也可以是一般的成员方法。所谓抽象方法是指在类定义中，只被声明原型，而不定义方法体的成员方法。其具体定义格式为：

```
abstract returnType methodName(parameterList)
```

其中，abstract是声明抽象方法的关键字，returnType是返回类型，methodName是抽象方法的名称，parameterList是参数列表。由于抽象方法是不完整的成员方法，因此，它只能包含在不能够被实例化的抽象类中。

面向对象程序设计

使用抽象类时，需要注意下面几点：

- 任何包含抽象方法的类都必须被定义成抽象类
- 由于抽象类不是一个完整的类，因此不能够被实例化
- 抽象类主要用来派生子类，且在子类中必须覆盖抽象类中的所有抽象方法，以便完善它们的定义
- **static、private和final**修饰符不能应用于抽象方法和抽象类中。

面向对象程序设计

4.6 接口

4.6.1 接口的声明

声明接口的基本格式为：

```
[public] interface interfacename
{
        常量声明;
        抽象方法声明;
}
```

public为接口的访问属性。在接口中，只允许出现常量和成员方法的定义，常量必须为public static final，因此可以省略，成员方法必须是抽象方法，所以也可以省略abstract修饰符。

面向对象程序设计

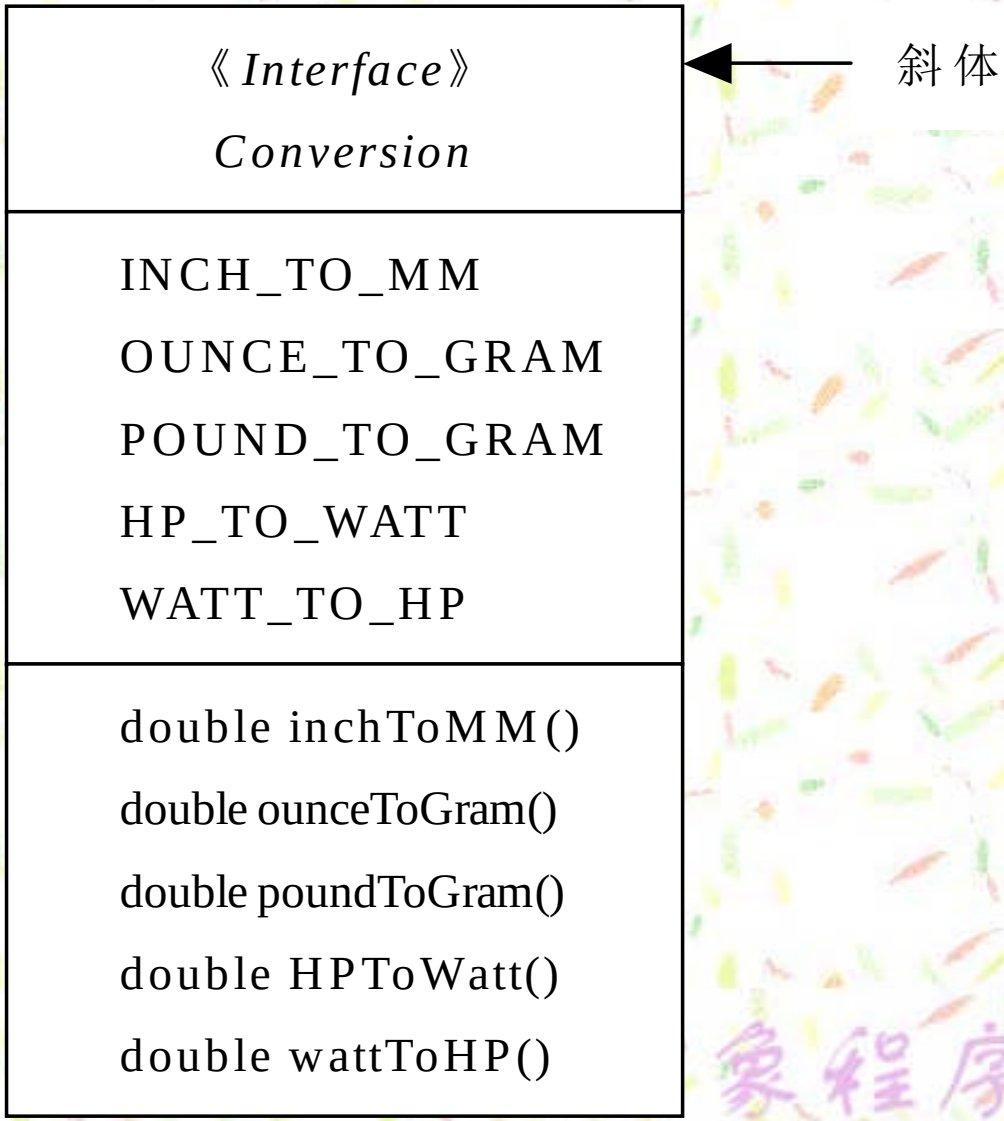
接口实例

```
public interface Conversions      //各种计量单位转换接口
{
   double INCH_TO_MM=25.4;      //1英寸=25.4毫米
   double OUNCE_TO_GRAM=28.349523125;   //1盎司=28.349523125克
   double POUND_TO_GRAM=453.5924;     //1磅=453.5924克
   double HP_TO_WATT=745.7;      //1马力=745.7瓦特
   double WATT_TO_HP=1.0/HP_TO_WATT;    //1瓦特=1.0/745.7马力

   double inchToMM(double inches);     //英寸转换成毫米
   double ounceToGram(double ounces);     //盎司转换成克
   double poundToGram(double pounds);     //磅转换成克
   double HPToWatt(double hp);           //马力转换成瓦特
   double wattToHP(double watts);         //瓦特转换马力
}
```

面向对象程序设计

Conversions接口的UML图形表示形式



4.6.2 接口的实现

接口是一种用来声明常量和操作行为格式的特殊抽象类，在接口中声明的所有成员方法都是抽象方法，因此，接口不能实例化，需要构造一个类，并在该类中覆盖接口中的所有方法，以便将其完善，我们将此称为某个类实现接口。

实现Conversions接口的类定义

```
public class MyClass implements Conversions //实现Conversions接口
{
   public double inchToMM(double inches)
   {   return inches*INCH_TO_MM;   }
   public  double ounceToGram(double ounces)
   {  return ounces*OUNCE_TO_GRAM;  }
   public  double poundToGram(double pounds)
   {  return inches*POUND_TO_GRAM;  }
   public  double HPToWatt(double hp)
   {  return inches* HP_TO_WATT;  }
   public  double wattToHP(double watts)
   {  return inches* WATT _TO_ HP;  }
}
```

implements Conversions说明MyClass类将实现Conversions接口，因此，在MyClass类定义中必须实现该接口中的所有抽象方法。这样一来，就可以通过MyClass类细化接口中的所有操作。有了MyClass类之后，就可以创建MyClass类的对象。

例如：

```
MyClass MyObject=new MyClass();
System.out.println("1 pound is "+poundToGram(1));
```

如果在类定义时，指明实现某个接口，但在类中并没有覆盖接口中的所有抽象方法，则这个类就必须被声明成抽象类:

```
public abstract class MyClass implements Conversions
{
   public double inchToMM(double inches)
   {  return inches*INCH_TO_MM;  }
   public  double ounceToGram(double ounces)
   {  return ounces*OUNCE_TO_GRAM;  }
}
```

由于这个类是抽象类，所以不能创建该类的对象，必须再定义一个MyClass子类，并在这个子类中实现上述接口中的其余方法。

接口的主要应用

接口是一种特殊形式的抽象类，它主要用来组织公用的常量，并统一操作行为的格式，通常它被应用在两个主要方面：

- 是用接口包装常量。我们可以将各式各样的常量放在接口中，让每个使用这些常量的类对象实现这个接口，从而达到享有这些常量的目的
- 用接口实现多态性。将成员方法从类中分离出来组成一个接口，随后由每个类实现这个接口

4.7 包

包是类和接口的集合。将所有的类和接口按功能分别放置在不同的包中主要有两点好处：一是便于将若干个已存在的类或接口整体地添加到程序代码中；二是避免出现类名冲突的现象。Java语言规定，在一个包中不允许有相同名称的类文件，但对于在不同包中的类文件没有这种限制，这是因为加载每个类时，必须指明该类所在的包名，以此区别不同包中的类。

4.7.1 创建包

包的概念是通过创建目录实现的。创建一个包就是用包的名称在文件系统下创建一个目录。在创建的目录下，既可以存放类文件或接口文件，也可以包含子目录，这些子目录是该包中的子包。创建一个包且将类文件放入其中的语法格式为：

package packageName;

包的实例

若将类文件放入userPackage中，可以这样写：

```
package userPackage;
```

一条包语句必须是文件中的第一条语句。如果在一个类文件中，包含了这样一条语句，系统就会自动地在指定路径下寻找这个包，即目录名。若不存在，立即创建，并将该文件放入这个包中。如果希望将在一个文件中定义的类或接口放在不同的包中，就只能将它们分别放在不同的文件中，并利用包语句指定不同的包。

指定子包

指定一个包中的子包，就需要将每个包按层次顺序书写成一个包序列，包之间用逗号分隔。例如，若希望将一个类文件放入userPackage1的子包userPackage2的子包 userPackage3中，就应该将包

语句写成：

```
package userPackage1. userPackage2. userPackage3;
```

4.7.2 加载包

对于在包中具有public访问属性的类或接口，可以通过导入语句（import）将其添加到程序代码中，并通过类名或接口名引用这些类或接口。导入语句的基本格式为import后跟包名序列及类名。

import userPackage.*;

userPackage是包名，.*代表将包中的所有类和接口都加载进来。

import userPackage1. userPackage2. userPackage3.*;

这条语句表示将userPackage1包中的子包userPackage2的子包 userPackage3中的所有类和接口加载进来。

JAVA标准包1

包名	描述
java.lang	包含Java语言所使用的基础类。
java.io	包含支持输入/输出操作的所有标准类。
java.awt	包含支持Java的图形用户接口的标准类。
javax.swing	提供支持‘Swing’的GUI组件的类。
javax.swing.border	支持生成Swing组件的边框类。
javax.swing.event	支持Swing组件的事件处理类。

JAVA标准包2

java.awt.geom	包含用二维图元绘画的类。
java.awt.image	包含支持图象处理的类。
java.applet	包含编写Applet程序的类。
java.util	包括支持管理数据集合、访问数据和时间信息，以及分析字符串的类。
java.util.zip	包含支持建立. Jar（Java Archive）文件的类。
java.awt.event	包括支持事件处理的类。
java.sql	包含支持使用标准SQL对数据库访问的类。