



第八章 多态

第8章 多态

- ◆ 多态实例
- ◆ 动态绑定
- ◆ 对象的多态性
- ◆ 多态应用实例

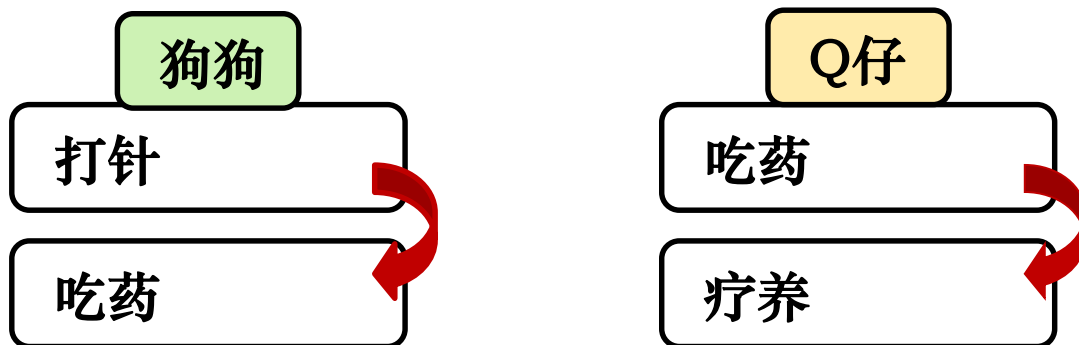
多态

- ◆ **多态**是不同事务具有不同表现形式的能力，即不同的对象面对同一个行为呈现出不同的执行效果。
- ◆ **继承**允许对象视为它自己本身或者父类类型加以处理。同样的代码就可以毫无差别地运行在不同类型之上。
- ◆ **多态调用**允许一种类型表现出其他相似类型之间的区别，只要他们都是从同一个父类继承而来的。

为什么使用多态?



- 宠物 (Pet) 生病了，需要主人给宠物看病
 - 不同宠物看病过程不一样



- 不同宠物恢复后体力值不一样

为什么使用多态?

■ 编码实现

主人類

```
public class Master {  
    public void Cure(Dog dog) {  
        if (dog.getHealth() < 50) {  
            dog.setHealth(60);  
            System.out.println("打针、吃药");  
        }  
    }  
    public void Cure(Penguin penguin){  
        if (penguin.getHealth() < 50)  
            penguin.setHealth(70);  
        System.out.println("吃药、疗养");  
    }  
}
```

测试方法

```
... ..  
Master master = new Master();  
master.Cure(dog);  
master.Cure(penguin);  
... ..
```

为什么使用多态?



- 如果又需要给XXX看病，怎么办？
 - 添加XXX类，继承Pet类
 - 修改Master类，添加给XXX看病的方法

频繁修改代码，代码可扩展性、可维护性差

使用多态优化设计

为什么使用多态?

■ 使用多态优化后的代码

Dog类

```
public class Dog {  
    public void setHealth(int health) {  
        this.setHealth(health);  
        System.out.println("吃药、疗养");  
    }  
}
```

Penguin类

```
public class Penguin {  
    public void setHealth(int health) {  
        this.setHealth(health);  
        System.out.println("吃药、疗养");  
    }  
}
```

又要给XXX看病时，只需：

1. 编写XXX类继承Pet类（旧方案也需要）
2. 创建XXX类对象（旧方案也需要）
3. 其他代码不变（不用修改Master类）

```
pet pet) {  
    if (pet.getHealth() < 50)  
        pet.Cure();  
}
```

3

```
Master master = new Master();  
master.Cure(pet);  
... ..
```

4

什么是多态?

■ 生活中的多态



你能列举出一个多态的生活示例吗?

同一种事物，由于条件不同，产生的结果也不同

■ 程序中的多态

同一个引用类型，使用不同的实例而执行不同操作

父类引用，子类对象

多态性(polymorphism)

- 概念：是面向对象程序设计的另一个**重要特征**，其基本含义是“**拥有多种形态**”，具体指在程序中**用相同的名称来表示不同的含义**。例如：用同一方法名来表示不同的操作。
- 类型：有两种
 - **静态多态性**：包括变量的隐藏、方法的重载
 - **动态多态性**：在编译时不能确定调用方法，只有在运行时才能确定调用方法，又称为运行时的多态性。

静态多态

- **静态多态**也称为**编译时多态**，即在编译时决定调用哪个方法；
- 静态多态一般是指**方法重载**；
- 只要构成了**方法重载**，就可以认为形成了静态多态的条件；
- 静态多态与是否发生继承**没有**必然联系。

动态多态

- 动态多态也称为**运行时多态**，即在运行时才能确定调用哪个方法；
- 形成动态多态必须具体以下条件：
 - 必须要有**继承**的情况存在；
 - 在继承中必须要有**方法覆盖**；
 - 必须由**父类的引用**指向**派生类的实例**，并且通过**父类的引用调用被覆盖的方法**；
- 由上述条件可以看出，**继承是实现动态多态的首要前提。**

下面主要介绍动态多态性

//多态性的例子

```
class Animal {  
    public void roar(){  
        System.out.println("动物:...");  
    }  
}  
  
class Cat extends Animal {  
    public void roar(){  
        System.out.println("猫:喵,喵,喵,...");  
    }  
}  
  
class Dog extends Animal {  
    public void roar(){  
        System.out.println("狗:汪,汪,汪,...");  
    }  
}
```

//多态性的例子(续)

```
public class AnimalTest {  
    public static void main(String args[]){  
        Animal am=new Animal();  
        am.roar();  
        am=new Dog();  
        am.roar();  
        am=new Cat();  
        am.roar();  
    }  
}
```

程序运行结果:
动物:...
狗:汪,汪,汪,...
猫:喵,喵,喵,...

根据对象的赋值规则，可以把子类对象赋给父类对象名，这是允许。当用同一形式：

父类对象名.roar()

去调用时，能够根据子类对象的不同，得到不同的结果，这就是多态性。

8.1 多态实例

- ◆ 独立的一个类，当**没有继承**关系时，它的**句柄**只能引用该类的**对象**。
- ◆ **句柄**是**生命为某类**的**变量**，此变量为**引用类型**。

8.1.1 句柄引用对象实例

- ◆ 独立的一个类，当没有继承关系时，它的句柄只能引用该类的对象。
- ◆ 句柄是生命为某类的变量，此变量为引用类型。

8.1.1 句柄引用对象实例

【例8.1】 独立的Pencil类

// Pencil.java

```
public class Pencil {  
    private String type;  
    public Pencil() {  
    }  
    public Pencil(String type) {  
        this.type = type;  
    }  
    public String getType() {  
        return this.type;  
    }  
    public void setType(String type) {  
        this.type = type;  
    }  
}
```



```
public void write(String s) {  
    System.out.print("使用铅笔书写: " + s);  
}  
public String toString() {  
    String r;  
    r = "一支铅笔";  
    if (this.type != null && this.type.length() != 0) {  
        r += ", 其类型是 " + this.type;  
    }  
    return r;  
}  
}
```

测试类TestPencil.java。

在测试类中可以声明**Pencil句柄**，创建对象，并使用对象的方法。

```
public class TestPencil{  
    public static void main(String[] args) {  
        Pencil pen = null;  
        //声明Pencil类的句柄  
        p.write("现在开始写今天的家庭作业!");  
        //调用Pencil的方法  
    }  
}
```

8.1.2 父类句柄引用子类对象

◆ 当类之间存在继承时，**某类的句柄**除了可以应用**该类的对象**之外，还可以引用该类的**所有子类对象**。

```
Pencil pen = null;
```

```
pen = new RubberPencil();
```

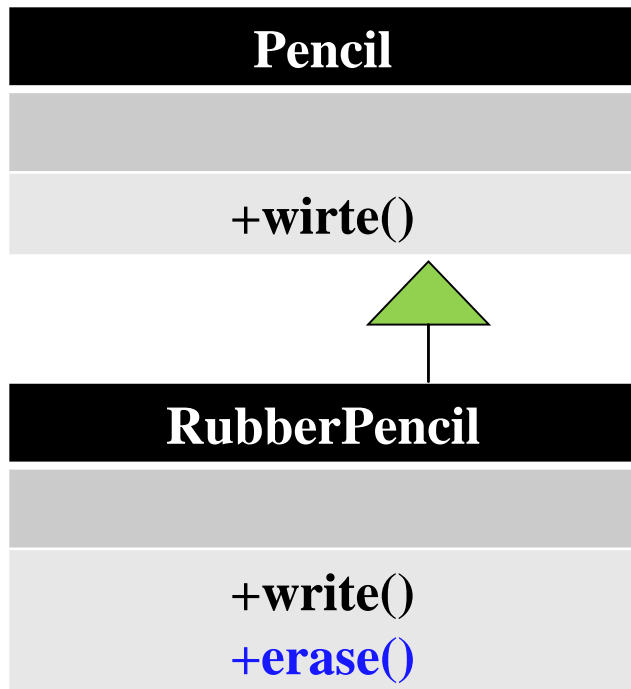
```
pen.write (“开始做家庭作业！ 第一题的答案是29\n”);
```

```
pen.write (“第二题的答案是62\n! ”);
```

8.1.2 父类句柄引用子类对象

- ◆ 父类的句柄可以引用子类的对象，是由于继承关系保证的。
- ◆ 父类和子类的关系是一般和特殊的关系。
- ◆ 声明为父类的句柄可以指向它的一种特殊子类。
- ◆ 父类的句柄引用子类对象，不可以使用子类有而父类没有的成员，包括成员变量和方法。

```
Pencil pen = null;  
pen = new RubberPencil();  
System.out.println();  
pen.write (“开始做家庭作业！ 第一题的答案是29\n”);  
pen.write (“第二题的答案是62\n! ”);  
pen.erase(“62”);
```



使用父类的句柄引用子类对象职能访问父类已经定义的成员，不能访问子类中存在但父类没有的成员。

8.2 动态绑定

- ◆ 将方法调用和方法体连接到一起成为绑定，可将绑定分为编译时绑定和动态绑定。
- ◆ 如果在程序运行之前进行绑定，称为编译时绑定或早期绑定。
- ◆ 如果在程序运行期间进行绑定，称为动态绑定、后期绑定或运行时绑定。
- ◆ Java中的多态性是依靠动态绑定实现的。

8.2 动态绑定

【例8.1】 独立的Pencil类

// Pencil.java

```
public class Pencil {  
    private String type;  
    public Pencil() {  
    }  
    public Pencil(String type) {  
        this.type = type;  
    }  
    public String getType() {  
        return this.type;  
    }  
    public void setType(String type) {  
        this.type = type;  
    }  
}
```

```
public void write(String s) {  
    System.out.print("使用铅笔书写: " + s);  
}  
public String toString() {  
    String r;  
    r = "一支铅笔";  
    if (this.type != null && this.type.length() != 0) {  
        r += ", 其类型是 " + this.type;  
    }  
    return r;  
}  
}
```


Eraser接口

```
public interface Eraser {  
    void erase(String s);  
}
```

RubberPencil类

```
public class RubberPencil extends Pencil implements Eraser {  
    public RubberPencil() {  
    }  
    public RubberPencil(String type) {  
        super(type);  
    }  
}
```

// 重写write () 方法

```
public void write(String s) {  
    System.out.print("使用橡皮铅笔书写: " + s);  
}
```

//实现Erase接口中的erase()方法

//实现Erase接口中的erase()方法

```
public void erase(String s) {  
    System.out.print("使用橡皮铅笔擦除: " + s);  
}
```

// 重写toString()

```
public String toString() {  
    String r;  
    r = "一支橡皮铅笔";  
    if (getType() != null && getType().length() != 0) {  
        r += ", 其类型是 " + getType();  
    }  
    return r;  
}
```

```
public static void main(String[] args) {  
    Pencil rp = new RubberPencil("2B");  
    // 用父类句柄调用子类对象的方法  
    rp.write("现在开始写今天的家庭作业! \n");  
}  
}
```

运行结果:

使用橡皮铅笔书写: 现在开始写今天的家庭作业!

8.2 动态绑定

- ◆ 句柄引用对象时，如果没有继承，某类的句柄只能引用该类的对象。
- ◆ 存在继承关系后，某类的句柄可以引用该类的对象，还可以引用其子类的对象，子类包括直接子类和间接子类。
- ◆ 父类的句柄引用子类对象时，不能访问父类没有定义而子类定义的成员。如果父类的方法在子类中进行了重写，父类句柄调用重写的方法时将运行子类的方法体。

8.3 对象的多态性

从左到右是从数据占位空间小到占位空间大的转换，可以由编译器自动转换。从右到左则不能自动转换，必须使用强制类型转换。

■ 隐式类型转换

低----->高

byte,short,char—> int —> long—> float —> double

8.3 对象的多态性

例如：

```
long lg;
```

```
float f = 3.14f;
```

```
// 注意，此处必须加上 f，否则产生编译错误
```

```
// 因为任何带小数点的数值默认都是 double 类型
```

```
lg = (long) f;
```

```
int i, j;
```

```
double r;
```

```
i = 29;
```

```
j = 6;
```

```
r = i / j;
```

```
r = ((double) i) / j;
```

造型也是一种类型转换，只不过是把对象从一个类转换为另一个类。类可以是具体类、抽象类、接口。

隐式转换：

```
Pencil pen = new RubberPencil();  
pen.write("开始做作业!");
```

以上是把对象`pen` 从`RubberPencil`类型转换为`Pencil`类型。由于二者是继承关系。`RubberPencil`子类实例自动地就是`Pencil`父类的实例。

显式转换：

```
Object obj;  
RubberPencil rb= (RubberPencil)obj;
```

instanceof 运算符

instanceof用于判断对象是不是某个类的实例。

<引用变量> instanceof <类名称/接口名称>

```
public class Demo{  
    public static void main (String[] args){  
        Object obj = “this is String”;  
        System.out.println(obj instanceof Object);  
    }  
}
```


8.4 多态应用实例

- ◆ **封装**可以隐藏实现细节，使得代码模块化。
- ◆ **继承**可以扩展已经存在的类。
- ◆ **多态**是为了实现接口重用。
 - 使用多态，应用程序不用逐一调用子类，只需要处理抽象父类，提高了程序的可复用性。
 - 子类的功能可以被父类的方法或引用变量调用，提高了程序的可扩充性和可维护性。

8.4 多态应用实例

// USB接口定义

```
public interface USB{  
    public void work();  
    public void insert();  
}
```

// MP3类实现了USB接口

```
public class Mp3 implements USB{  
    private String brand;  
    public Mp3(){  
        brand="unknown";  
    }  
    public Mp3(String brand){  
        this.brand=brand;  
    }  
    public void work(){  
        System.out.println(brand + " mp3 is working...");  
    }  
}
```

```
public void insert(){
    System.out.println(brand + " mp3 is inserting...");
}
}
```

// Printer类实现了USB接口

```
public class Printer implements USB{
    private String brand;
    public Printer(){
        brand="unknown";
    }
    public Printer(String brand){
        this.brand=brand;
    }
    public void work(){
        System.out.println(brand + " Printer is working...");
    }
    public void insert(){
        System.out.println(brand + " Printer is inserting...");
    }
}
```

//定义电脑类，使用USB接口

```
public class Computer{
```

```
    // USB接口作为输入参数
```

```
    void useUSB (USB usb){
```

```
        usb.work();
```

```
        usb.insert();
```

```
    }
```

```
}
```

// 测试类

```
public class TestComputer{
```

```
    public static void main(String args[]){
```

```
        Mp3 mp3=new Mp3('philipse');
```

```
        Printer printer=new Printer('hp');
```

```
        Computer IBM=new Computer();
```

```
        IBM.useUSB(mp3);    //使用MP3
```

```
        IBM.useUSB(printer);    //使用打印机
```

```
    }
```

```
}
```

8.4.2 父类作为方法返回类型实例

// 笔工厂，根据硬币投掷情况，生成铅笔或橡皮铅笔

```
class PencilFactory {  
    public static Pencil getPencil()    // 返回类型为父类Pencil  
    {  
        Pencil pen = null;  
        if (Coin.throwOut() == Coin.ZHENG_MIAN) {  
            pen = new Pencil();  
        }  
        else {  
            pen = new RubberPencil();  
        }  
        return pen;  
    }  
}
```

【例8.5】 随机选择铅笔或橡皮铅笔写作业

//铅笔类

```
class Pencil {  
    private String type;  
    public Pencil() {  
    }  
    public Pencil(String type) {  
        this.type = type;  
    }  
    public String getType() {  
        return this.type;  
    }  
    public void setType(String type) {  
        this.type = type;  
    }  
}
```

【例8.5】 随机选择铅笔或橡皮铅笔写作业

//铅笔类


```
public void write(String s) {  
    System.out.print("使用铅笔书写: " + s);  
}  
  
public String toString() {  
    String r;  
    r = "一支铅笔";  
    if (this.type != null && this.type.length() != 0) {  
        r += ", 其类型是 " + this.type;  
    }  
    return r;  
}  
}
```

// 橡皮接口

```
interface Eraser {  
    void erase(String s);  
}
```

// 橡皮铅笔类，继承铅笔类，实现橡皮接口

```
class RubberPencil extends Pencil implements Eraser {  
    public RubberPencil() {  
    }  
    public RubberPencil(String type) {  
        super(type);  
    }  
    public void write(String s) {  
        System.out.print("使用橡皮铅笔书写: " + s);  
    }  
    public void erase(String s) {  
        System.out.print("使用橡皮铅笔擦除: " + s);  
    }  
}
```

```
public String toString() {  
    String r;  
    r = "一支橡皮铅笔";  
    if (getType() != null && getType().length() != 0) {  
        r += ", 其类型是 " + getType();  
    }  
    return r;  
}  
}
```

// 投硬币，随机产生正面和反面

```
class Coin {  
    public static final int ZHENG_MIAN = 0;  
    public static final int FAN_MIAN = 1;  
  
    public static int throwOut() {  
        int i = (int) (Math.random() * 100);  
        if (i % 2 == 0) {  
            return ZHENG_MIAN;  
        }  
        else {  
            return FAN_MIAN;  
        }  
    }  
}
```

// 笔工厂，根据硬币投掷情况，生成铅笔或橡皮铅笔

```
class PencilFactory {  
    public static Pencil getPencil()    // 返回类型为父类Pencil  
    {  
        Pencil pen = null;  
        if (Coin.throwOut() == Coin.ZHENG_MIAN) {  
            pen = new Pencil();  
        }  
        else {  
            pen = new RubberPencil();  
        }  
        return pen;  
    }  
}
```

// 测试类，调用笔工厂，生成笔，完成作业

```
public class PencilSelectorWithFactory {  
    public static void main(String[] args) {  
        Pencil pen = null;  
        pen = PencilFactory.getPencil();  
        System.out.println();  
        pen.write(" 开始做家庭作业! \n");  
        RubberPencil rp = null;  
        if (pen instanceof RubberPencil) {  
            rp = (RubberPencil) pen;  
            rp.write(" 第一题的答案是 36\n");  
            rp.erase(" 第一题的答案是 36\n");  
        }  
    }  
}
```



运行结果是：

使用铅笔书写： 开始做家庭作业！

也可能是：

使用橡皮铅笔书写： 开始做家庭作业！

使用橡皮铅笔书写： 第一题的答案是 36

使用橡皮铅笔擦除： 第一题的答案是 36

8.4.3 面向接口编程

面向接口编程是面向对象编程体系中的思想精髓之一。接口是抽象方法和常量的集合体。那么，接口存在的意义是什么。可以从以下两个视角考虑：

(1) 接口是一组规则的集合，它规定了实现接口的类或接口必须拥有的一组规则。体现了自然界“如果你是……则必须能……”的理念。

(2) 接口是在一定粒度视图上同类事物的抽象表示。注意这里强调了在一定粒度视图上，因为“同类事物”这个概念是相对的。