

Lesson6

面向对象的三大特称之三：多态 (polymorphism)

主讲老师：申雪萍



面向对象编程的三大特性：封装、继承、多态

- 封装

- 类：是一个封装体

- 通过权限修饰符，使得类的封装更合理，更健壮

- Package：类空间的划分单位

- Software System / Hardware System

- 预留接口

代码一级封装

系统一级封装

面向对象编程有三大特性：封装、继承、多态

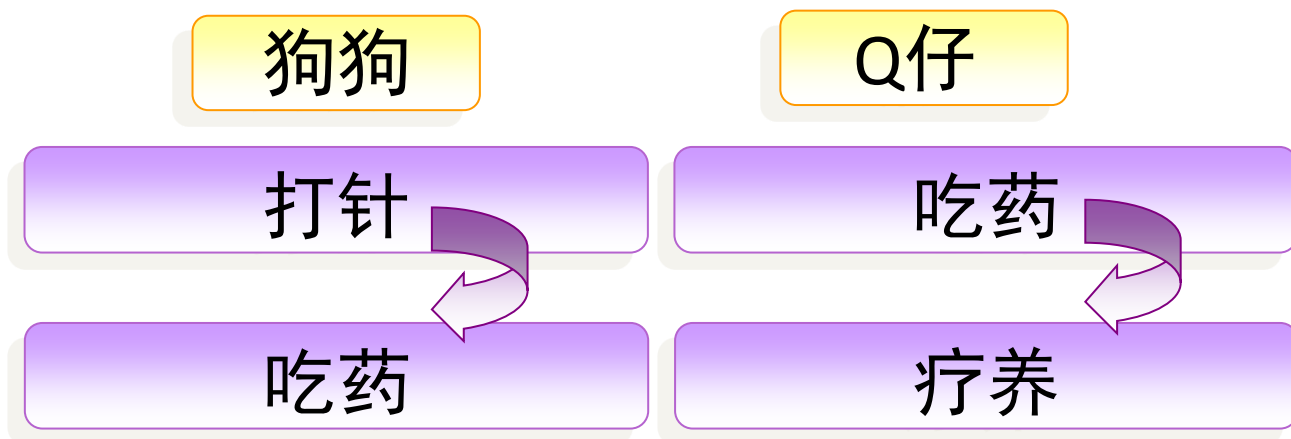
- 两个类若存在IS-A的关系就可以使用继承。继承是为了重用父类代码，解决了代码冗余。
- 更为重要的是：继承为多态的实现做了铺垫

主要内容

- 多态的必要性
- 静多态和动多态
- 方法重载，方法覆盖
- 多态的优点及运行机制
- 抽象方法
- 抽象类
- 接口的必要性（将接口用作API）
- 定义接口
- 实现接口
- 将接口用作类型、接口回调（使用接口）
- 接口的进化（通过接口的继承完成）
- 面向接口的编程
- 案例分析

为什么使用多态？

- 宠物生病了，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?

测试方法

- 编码实现

主人类

```
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");  
    }  
}
```

... ..

```
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 extends Pet {  
    public void toHospital() {  
        this.setHealth(60);  
        System.out.println("打针、吃药");  
    }  
}
```

Penguin类

```
public class Penguin extends Pet {  
    public void toHospital() {  
        this.setHealth(70);  
        System.out.println("吃药、疗养");  
    }  
}
```

主人类

```
public class Master {  
    public void Cure(Pet pet) {  
        if (pet.getHealth() < 50)  
            pet.toHospital()  
    }  
}
```

父类引用子类对象

测试方法

```
... ..  
Pet pet = new Dog();  
Master master = new Master();  
master.Cure(pet);  
Pet pet = new Penguin();  
master.Cure(pet);  
... ..
```


为什么使用多态？

- 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）

代码可扩展性、可维护性变好了！

动多态：运行时根据所引用的具体实例来确定调用父类方法还是子类方法

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

```
public class Master {  
    public void Cure(Pet pet) {  
        if (pet.getHealth() < 50)  
            pet.toHospital();  
    }  
}
```

```
... ..  
Pet pet = new Dog();  
Master master = new Master();  
master.Cure(pet);  
Pet pet = new Penguin();  
master.Cure(pet);  
... ..
```

为什么会出现多态？

- Java 中的引用变量有两个类型：
 - 一个是编译时的类型，一个是运行时的类型，编译时的类型由声明该变量时使用的类型决定，运行时的类型由实际赋给该变量的对象决定。
- 如果编译时的类型与运行时的类型不一致就会出现所谓的多态？

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”，具体指在程序中使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：包括隐藏、方法的重载
 - 动态多态性：在编译时不能确定所要调用的方法，只有在运行时才能确定所要调用的方法，又称为运行时的多态性

静态多态（静多态）（深刻理解一下静多态）

- 静态多态：即在编译时决定调用哪个方法，也称为编译时多态，也称为静态联编，也称为静绑定；
- 静态多态一般是指方法重载，方法隐藏；
- 只要构成了方法重载，就可以认为形成了静态多态的条件；因此，静态多态与是否发生继承没有必然联系。

- 方法重载：Java允许在一个类中定义多个同名的方法，但这些方法的参数列表必须不同
 - 方法名相同，参数个数、参数类型及参数顺序至少有一个不同
- 重载的目的：一般是为了创建一组要完成相似任务的成员方法。
- 构造方法和静态成员方法都是可以重载，静态成员方法重载后的方法也可以是非静态成员方法。

方法重载

- 注意1：对于**方法重载**，返回值类型与访问权限修饰符可以相同也可以不同，上述两项不能当做判断是否重载的条件。
- 注意2：如果一个类中有两个同名方法，其参数列表完全一样，仅仅返回值类型不同，则编译时会产生错误

com.buaa.classEx.MethodOverload

```
public class MethodOverload {  
    public static void main(String args[]) {  
        int a = 51, b = -98, c = 8, d = 191;  
        double u = 25.1, v = -29.8, x = 3.1, y = 89.98;  
        System.out.println("51、-98、8、191四数的最大值是: " + max(a, b, c, d));  
        System.out.println("25.1、-29.8、3.1、89.98四数的最大值是: " + max(u, v, x, y));  
    }  
}
```

```
static int max(int a, int b, int c, int d) {
```

```
    int x, y;  
    x = a > b ? a : b;  
    y = c > d ? c : d;  
    return x > y ? x : y;  
}
```

```
static double max(int a, int b, int c, int d) {  
    double x, y;  
    x = a > b ? a : b;  
    y = c > d ? c : d;  
    return x > y ? x : y;  
}
```

```
/*  
 * 如果一个类中有两个同名方法，其参数列表完全一样，仅仅返回值类型不同，则编译时会产生错误  
 */  
/*  
static double max(int a, int b, int c, int d) {  
    int x, y;  
    x = a > b ? a : b;  
    y = c > d ? c : d;  
    return x > y ? x : y;  
}  
*/
```


- 方法覆盖是子类的成员方法重写了父类的成员方法，重写的目的很大程度上是为了实现多态；
- 动态多态：即在运行时才能确定调用哪个方法，也称为运行时多态，也称为动态联编，也称为动绑定；
- Java中，实现多态有3个条件：继承、覆盖、向上转型，缺一不可。
 - “覆盖(override)方法、抽象方法和接口” 和动态联编关系紧密

Java形成动态多态必须具备以下条件:

- Java形成动态多态必须具备以下条件:

① 必须要有继承的情况存在;

② 在继承中必须要有方法覆盖;

③ 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。**

... ..

```
Pet pet = new Dog();
```

```
Master master = new Master();
```

```
master.Cure(pet);
```

```
Pet pet = new Penguin();
```

```
master.Cure(pet);
```

... ..

- **方法覆盖**：方法名、参数个数、参数类型及参数顺序必须一致；
- 若父类方法定义时有异常抛出，则子类覆盖父类该方法时，该方法也不能有更多的异常抛出，否则编译时会产生错误

方法覆盖（续）

- 子类方法不能缩小父类方法的访问权限：
 - a) 一个package方法可以被重写为package、protected和public的；
 - b) 一个protected方法可以被重写为protected和public的；
 - c) 一个public方法只可以被重写为public的；
- ② 私有方法、静态方法不能被覆盖，如果在子类出现了同签名的方法，就是方法隐藏；
- ③ 父类中，被final修饰的方法是最终方法，不允许覆盖。

```
class Sup {
    public int x, y;

    Sup(int a, int b) {
        x = a;
        y = b;
    }

    public void display() {
        int z;
        z = x + y;
        System.out.println("add=" + z);
    }
}

class Sub extends Sup {
    Sub(int a, int b) {
        super(a, b);
    }

    public void display() {
        int z;
        z = x * y;
        System.out.println("product=" + z);
    }
}
```

//display()在编译时不能被系统识别，而是在运行时才被系统识别，
//也称为运行时多态，也称为动态联编，也称为动绑定。

```
public class ResultDemo extends Sub
{
    ResultDemo(int x,int y)
    {
        super(x,y);
    }

    public static void main(String args[ ])
    {
        Sup num1=new Sup(7,14);
        Sub num2=new Sub(7,14);
        ResultDemo num3=new ResultDemo(7,14);
        num1.display( );
        num2.display( );
        num3.display( );
        num1=num2;
        num1.display();
        num1=num3;
        num1.display();
    }
}
```

```
add=21
product=98
product=98
product=98
product=98
```

```
package com.buaa.test;

public class Door {
    public Door() {
        super();
        System.out.println("Door...");
    }
}
```

```
package com.buaa.test;

public class WoodDoor extends Door {
    public WoodDoor() {
        super();
        System.out.println("Wood Door...");
    }
}
```

```
package com.buaa.test;
```

```
public class Room {
```

```
    public static Door getDoor() {  
        return new Door();  
    }
```

```
}  
package com.buaa.test;
```

```
public class Bedroom extends Room{
```

```
    public static Door getDoor() {  
        return new WoodDoor();  
    }
```

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

```
        Room m=new Bedroom();  
        System.out.println(m.getDoor());  
        System.out.println(Bedroom.getDoor());  
    }
```



```
Door...  
com.buaa.test.Door@15db9742  
Door...  
Wood Door...  
com.buaa.test.WoodDoor@6d06d69c
```

私有方法、静态方法不能被覆盖，如果在子类出现了同签名的方法，那是方法隐藏；

```
package com.buaa.test;
```

```
public class SuperClass {
```

```
    private void print() {  
        System.out.println("A");  
    }
```

```
    public static void main(String[] args) {  
        SuperClass a = new SubClass();  
        a.print();  
        SubClass b = new SubClass();  
        b.print();  
        new SubClass().print();  
    }
```

```
}
```

```
class SubClass extends SuperClass {
```

```
    public void print() {  
        System.out.println("B");  
    }
```

```
}
```

A
B
B

```
class Base {  
    int x = 1;  
    static int y = 2;  
    int z = 3;  
  
    int method() {  
        return x;  
    }  
    public static void test() {}  
}
```

```
class Subclass extends Base {  
    int x = 4;  
    int y = 5;  
    static int z = 6;  
    int method() {  
        return x;  
    }  
    public static void test(int x) {} //重载  
    //静态方法不能被重写为非静态方法，否则编译出错  
    public void test() {}  
}
```

```
package com.buaa.test;
public class Owner {
    private void f1() {
        System.out.println("Father f1()");
    }
    void f2() {
        System.out.println("Father f2()");
    }
    public static void main(String[] args) {
        Owner jack = new Son();
        jack.f1();
        jack.f2();
    }
}
class Son extends Owner {
    public void f1() {
        System.out.println("Son f1()");
    }
    protected void f2() {
        System.out.println("Son f2()");
    }
}
```

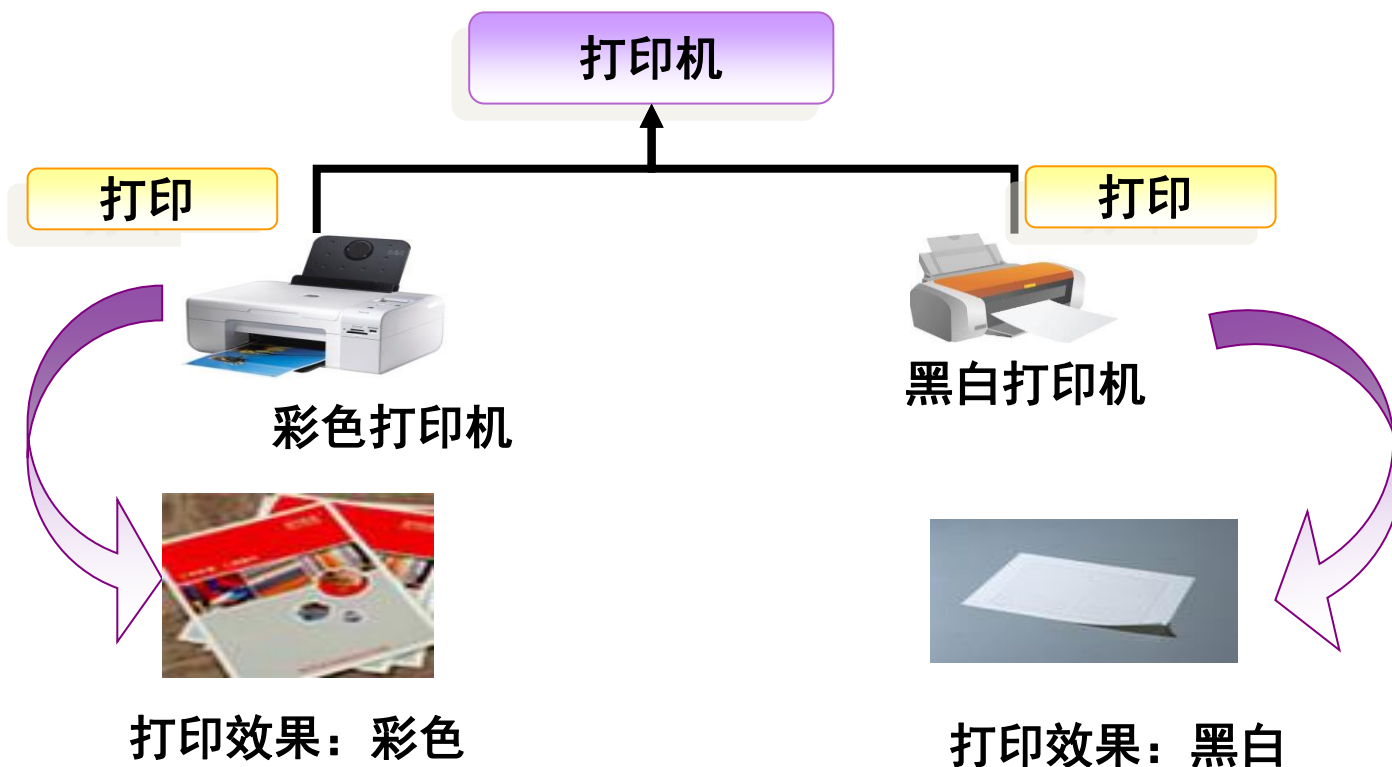
Father f1()
Son f2()

多态的实现：通过继承实现

- Java中，一个类只能有一个直接父类，不能多继承。
- Java中，一个父类可以有多个子类，而在子类里可以覆盖父类的方法。
- 当用父类的变量去引用不同的子类，在调用这个相同的方法的时候得到的结果和表现形式就不一样了，这就是多态，相同的消息（也就是调用相同的方法）会有不同的结果

案例解析（1）：问题需求

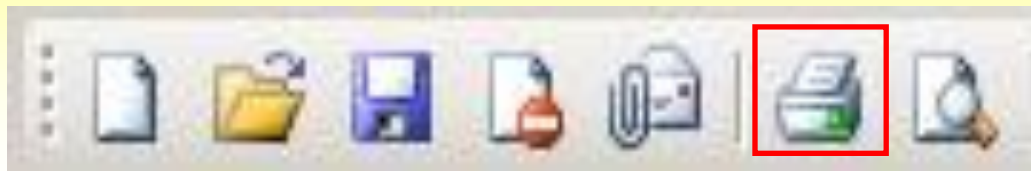
- 用多态实现打印机
 - 打印机分为黑白打印机和彩色打印机
 - 不同类型的打印机打印效果不同



分析：实现多态的流程

计算机可以连接各种打印机

无论连接何种打印机打印方法都相同



根据连接打印机不同，效果也不同

使用多态实现思路

- 编写父类
- 编写子类，子类重写（覆盖）父类方法
- 运行时，使用父类的类型，子类的对象

编码实现

继承是子类使用父类的方法，而多态则是父类使用子类的方法。（把父类当做子类来用）

```
class Printer(){  
    print(String str){}  
}
```

```
class ColorPrinter extends Printer {  
    print(String str) {  
        System.out.println("输出彩色的"+str);  
    }  
}
```

子类

```
class BlackPrinter extends Printer {  
    print(String str) {  
        System.out.println("输出黑白的"+str);  
    }  
}
```

```
public static void main(String[] args) {  
    Printer p = new ColorPrinter();  
    p.print();  
    p = new BlackPrinter();  
    p.print();  
}
```

运行

实现多态的三个要素：

1. 继承
2. 方法覆盖
3. 使用父类类型

- 实现运行时多态技术的条件
 - 有一个继承层次关系；
 - 在子类中重写父类的方法, 构成方法覆盖；
 - 通过父类的引用对子类对象进行调用。

采用多态技术的优点

- 引进多态技术之后，尽管子类的对象千差万别，但都可以采用 父类引用.方法名([参数]) 统一 方式来调用，在程序运行时能根据子对象的不同得到不同的结果。
- 应用程序不必为每一个派生类（子类）编写功能调用，只需要对抽象基类进行处理即可。这种“**以不变应万变**”的形式可以规范、简化程序设计，符合软件工程的“**一个接口，多种方法**”思想，可以**大大提高程序的可复用性**。
- 派生类的功能可以被基类的引用变量引用，这叫**向后兼容**，可以提高程序的**可扩充性和可维护性**。

- Java多态机制是基于“**方法绑定 (binding)**”，就是建立**method call**（方法调用）和**method body**（方法本体）的关联。
- 如果绑定动作发生于程序执行前（由编译器和连接器完成），称为“**先期绑定**”或者**早绑定**。
 - 对于面向过程的语言它们没有其他选择，一定是先期绑定。比如C编译器只有一种**method call**，就是先期绑定。（C++有先期联编和后期联编）

- 在编译阶段能够确定方法在内存什么位置的机制就叫静态绑定机制
- 所有私有方法、静态方法、构造器及final修饰方法都是采用静态绑定机制。在编译器阶段就已经指明了调用方法在常量池中的符号引用，JVM运行的时候只需要进行一次常量池解析即可

- 当有动多态的情况时，解决方案便是所谓的后期绑定（late binding）即晚绑定：绑定动作将在执行期根据对象类型而进行。
- 后期绑定也被称为执行期绑定（run-time binding）或动态绑定（dynamic binding）。

多态的运行机制（续）

- 对于Java当中的方法而言，`final`，`static`，`private`修饰的方法和构造方法是前期绑定。
- Java中，所有的`private`方法都被隐式的指定为`final`的。
- 将方法声明为`final`类型的一是为了防止方法被覆盖，二是为了有效的关闭java中的动态绑定。或者说，这么做便是告诉编译器：动态绑定是不需要的。于是编译器可以产生效率较佳的程序代码。

主要内容

- 多态的必要性
- 静多态和动多态
- 方法重载，方法覆盖
- 多态的优点及运行机制
- 抽象方法
- 抽象类
- 接口的必要性（将接口用作API）
- 定义接口
- 实现接口
- 将接口用作类型、接口回调（使用接口）
- 接口的进化（通过接口的继承完成）
- 面向接口的编程
- 案例分析

为什么使用抽象方法？

- 如果父类的方法没机会被访问调用，或者没有办法给出明确的定义。以下代码有什么问题？

```
public abstract class Pet {  
    public void print() {  
        //...  
    }  
}
```

每个子类的print（）方法实现不同，父类print（）方法的实现是多余的。

为什么使用抽象方法？

- 可以使用抽象方法来优化
 - 抽象方法**没有方法体**
 - 抽象方法**必须在抽象类**里
 - 抽象方法**必须在子类中被实现**，除非子类是抽象类

```
public abstract void print();
```

没有方法体

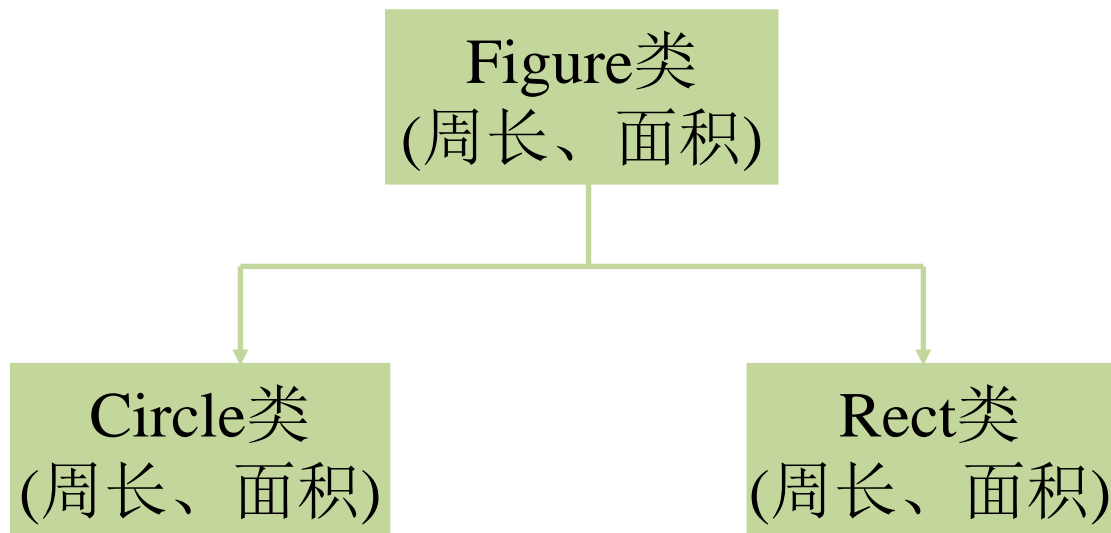
抽象类和抽象方法

- 一. 关键字`abstract` 可用来修饰方法和类，表示“尚未实现”的含义
- 二. 抽象类: 如果一个类用`abstract`修饰，则它是一个抽象类。
- 三. 抽象方法: 如果一个方法用`abstract`修饰，则它是一个抽象方法；抽象方法是没有方法体的，以一个紧跟在声明后的“;”结束。

注意：无方法体与方法体为空是两个不同的概念。

案例解析（2）：抽象类(abstract)

- 抽象是面向对象的一种重要方法,通过抽象我们能够设计一个更普通、更通用的类。
 - 例如：从许许多多学生中，抽象出**Student**类，从学生、工人、农民、...抽象出**Person**类。
 - 下面，我们来分析一个例子：



本案例重点

- 理解抽象类（重点）
- 理解抽象方法（重点）
- **理解抽象类引用（难点）**
- 复习覆盖
- 复习多态三要素

案例解析（1）：com.buaa.classEx.Figure

```
abstract class Figure { //抽象类,一般作为其它类的超类
    protected double x;

    Figure() {
    }

    Figure(double x1) {
        x = x1;
    }

    abstract public double area(); //抽象方法

    public String toString() {
        return "x=" + x;
    }
}
```

```
class Circle extends Figure {
```

```
    public Circle(double x1) {  
        super(x1);  
    }
```

```
    public double area() { //具体方法  
        return 3.1415926 * x * x;  
    }
```

```
    public String toString() { //方法覆盖  
        return "圆: \t" + super.toString() + "\tarea=" + area();  
    }
```

```
}
```

```
class Rectangle extends Figure {
```

```
    protected double y;  
    public Rectangle() {  
    };
```

```
    public Rectangle(double a, double b) {  
        super(a);  
        y = b;  
    }
```

```
    public double area() { //具体方法  
        return x * y;  
    }
```

```
    public String toString() { //方法覆盖  
        return "长方形: \t" + super.toString() +  
            "y=" + y + "\tarea=" + area();  
    }
```

```
}
```


抽象类引用

```
public class TestFigure {  
    public static void main(String args[]) {  
        Rectangle R1 = new Rectangle(10.0, 20.0);  
        Figure C1 = new Circle(10.0);  
        Figure F1 = new Rectangle(30.0, 30.0);  
        System.out.println(R1.toString());  
        System.out.println(C1.toString());  
        System.out.println(F1.toString());  
    }  
}
```

多态的存在有三个必要的条件:

- 1、要有继承（两个类之间存在继承关系，子类继承父类）
- 2、要有重写（在子类里面重写从父类继承下来的方法）
- 3、父类引用指向子类对象

长方形:	x=10.0y=20.0	area=200.0
圆:	x=10.0	area=314.15926
长方形:	x=30.0y=30.0	area=900.0

抽象类引用

- 虽然不能实例化抽象类，但可以创建它的引用。
- Java支持多态性，允许通过**父类引用**来引用子类的对象。

案例解析（3）：优化继承那一节中的Animal类

```
abstract class Animal {  
    public abstract void eat();  
    public abstract void sleep();  
}
```

```
class Giraffe extends Animal {  
    public void run() {  
        System.out.println("长颈鹿四条腿走路");  
    }  
    //覆盖  
    public void eat() {  
        System.out.println("长颈鹿在愉快的吃草");  
    }  
    //覆盖  
    public void sleep() {  
        System.out.println("长颈鹿站着睡觉");  
    }  
}
```

```
class Lion extends Animal {  
    //重写相应的方法  
    public void eat() {  
        System.out.println("狮子在吃肉");  
    }  
    //重写相应的方法  
    public void sleep() {  
        System.out.println("狮子在躺着睡觉");  
    }  
}
```

```
class Mouse extends Animal {  
    //重写相应的方法  
    public void eat() {  
        System.out.println("老鼠在吃肉");  
    }  
    //重写相应的方法  
    public void sleep() {  
        System.out.println("老鼠在躺着睡觉");  
    }  
    //添加新的方法  
    public void bore() {  
        System.out.println("老鼠在愉快地钻洞");  
    }  
}
```

```
public class AnimalTest {  
    public static void main(String[] args) {  
        Animal aMouse = new Mouse();  
        Animal aGiraffe = new Giraffe();  
        Animal aLion = new Lion();  
        aMouse.eat();  
        aMouse.sleep();  
        aLion.eat();  
        aLion.sleep();  
        aGiraffe.eat();  
        aGiraffe.sleep();  
        System.out.println("-----");  
        Animal[] aArray = new Animal[3];  
        aArray[0] = aMouse;  
        aArray[1] = aGiraffe;  
        aArray[2] = aLion;  
        for (Animal i : aArray) {  
            i.eat();  
            i.sleep();  
        }  
    }  
}
```

多态的存在有三个必要的条件:

- 1、要有继承 (两个类之间存在继承关系, 子类继承父类)
- 2、要有重写 (在子类里面重写从父类继承下来的方法)
- 3、父类引用指向子类对象

注意事项(1)

- 一. 如果一个类继承自某个抽象父类，而没有具体实现抽象父类中的抽象方法，则必须定义为抽象类。
- 二. **抽象类是不能实例化的，但可以创建它的引用。**
它的作用是提供一个恰当的父亲。因此一般作为其它类的超类, 与final类正好相反。
- 三. 如果一个类里有抽象的方法，则这个类就必须声明成抽象的。但一个抽象类中却可以没有抽象方法。

注意事项 (2)

- 抽象方法不能被`private`、`final`或`static`修饰。为什么？
 - ① 抽象方法必须被子类所覆盖，如果说明为`private`，则外部无法访问，覆盖也无从谈起。
 - ② 若说明为`static`，即使不创建对象也能访问：`类名.方法名()`，这要求给出方法体，但与抽象方法的定义相矛盾。
 - ③ `Final`和`abstract`含义矛盾
- 当类实现了一个接口，但并没有实现该接口的所有方法时，该类必须声明为抽象类，否则出错；

案例分析（4）

```
public abstract class Animal {  
    public String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public abstract void enjoy();  
}
```

```
public abstract class Cat extends Animal {  
    public String eyeColor;  
  
    public Cat(String n, String c) {  
        super(n);  
        this.eyeColor = c;  
    }  
}
```

如果一个类继承自某个抽象父类，而没有具体实现抽象父类中的抽象方法，则必须定义为抽象类。

```
public class BlueEyeCat extends Cat {  
    public BlueEyeCat(String n, String c) {  
        super(n, c);  
    }  
  
    @Override  
    public void enjoy() {  
        System.out.println("蓝眼猫叫...");  
    }  
}
```

```
public class Dog extends Animal {  
    public String furColor;  
  
    public Dog(String n, String c) {  
        super(n);  
        this.furColor = c;  
    }  
  
    @Override  
    public void enjoy() {  
        System.out.println("狗叫....");  
    }  
}
```

```
public class Demo {  
    public static void main(String[] args) {  
  
        Animal a;  
        a = new Dog("DOG", "black");  
        a.enjoy();  
        a = new BlueEyeCat("CAT", "blue");  
        a.enjoy();  
    }  
}
```

抽象类与具体类的比较

抽象类	具体类
用于划分具体类	用于表示真实世界的对象
不能实例化	可以实例化
定义了未提供实现的抽象方法	不定义未提供实现的抽象方法
为自己的部分方法提供实现	为所有的方法提供实现

案例解析（5）：

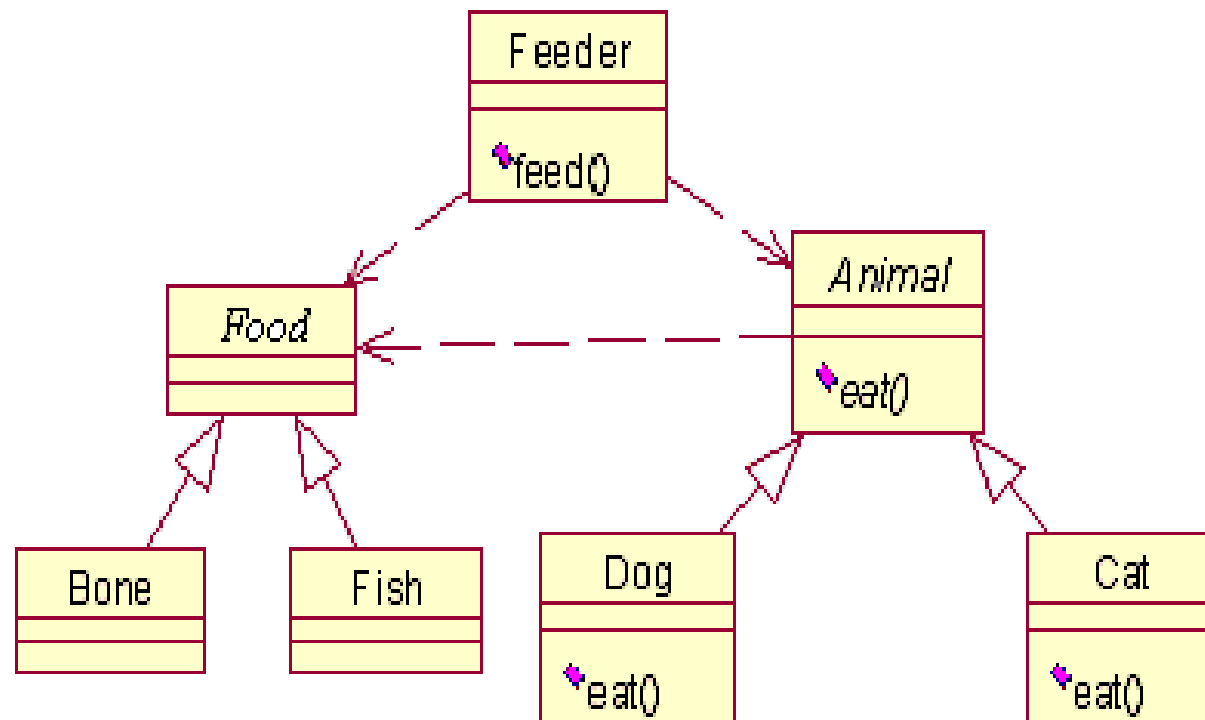
- 编写程序模拟动物园里饲养员给各种动物喂养各种不同食物的过程。

分析:

- 在这个动物园里，涉及的对象有
 - 饲养员
 - 各种不同动物
 - 以及各种不同的食物。
- 当饲养员给动物喂食时，动物发出欢快的叫声。
- 很容易抽象出3个类Feeder、Animal和Food。
- 假设只考虑猫和狗，则由Animal类派生出Cat类和Dog类
- 由Food类可以进一步派生出其子类Bone、Fish。因为他们之间存在着明显的is-a关系。

用到的知识点

- 继承
- 多态
- 抽象类



示例代码

```
package com.buaa.adstractAnimal;  
public abstract class Food {  
    public abstract String getName();  
}
```


```
package com.buaa.adstractAnimal;  
public abstract class Animal {  
    private String name;  
  
    public Animal(String name) {  
        this.name = name;  
    }  
  
    public abstract void eat(Food food);  
    public String getName() {  
        return this.name;  
    }  
}
```

Cat类和Food类属于依赖关系

```
package com.buaa.adstractAnimal;
public class Cat extends Animal {


    public Cat(String name) {
        super(name);
    }

    @Override
    public void eat(Food food) {
        System.out.println(
            "小猫" + this.getName()
            + "正在吃着" + food.getName()
        );
    }
}
```



Dog类和Food类属于依赖关系

```
public class Dog extends Animal {  
  
    public Dog(String name) {  
        super(name);  
    }  
  
    @Override  
    public void eat(Food food) {  
        System.out.println(  
            "小狗" + this.getName()  
            + "正在啃着" + food.getName()  
        );  
    }  
}
```




```
package com.buaa.adstractAnimal;  
public class Fish extends Food{  
    @Override  
    public String getName() {  
        return "鱼";  
    }  
}
```

示例代码

```
package com.buaa.adstractAnimal;  
public class Feeder {
```

```
    private String name;  
    public Feeder(String name) {  
        this.name = name;  
    }
```



```
        public void feed(Animal animal, Food food) {  
            System.out.println(  
                "饲养员" + this.name  
                + "喂养动物" + animal.getName()  
            );  
            animal.eat(food);  
        }
```

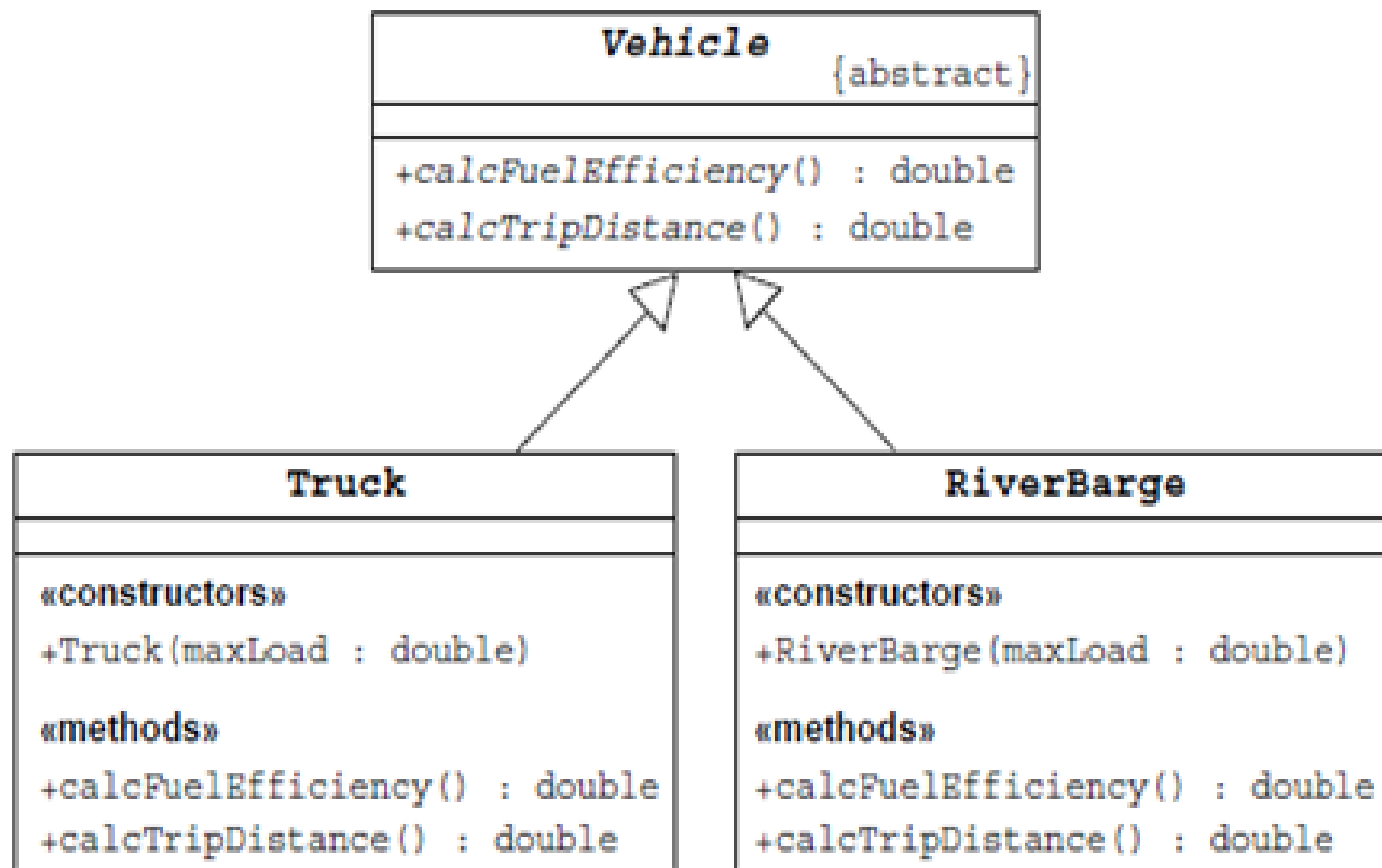
```
    }
```

```
package com.buaa.adstractAnimal;
public class Demo {
    public static void main(String[] args) {
        Feeder feeder = new Feeder("李大壮");
        Animal a;
        Food food;

        a = new Dog("阿柴");
        food = new Bone();
        feeder.feed(a, food);

        a = new Cat("喵喵");
        food = new Fish();
        feeder.feed(a, food);
    }
}
```


案例解析（6）：继承，抽象类

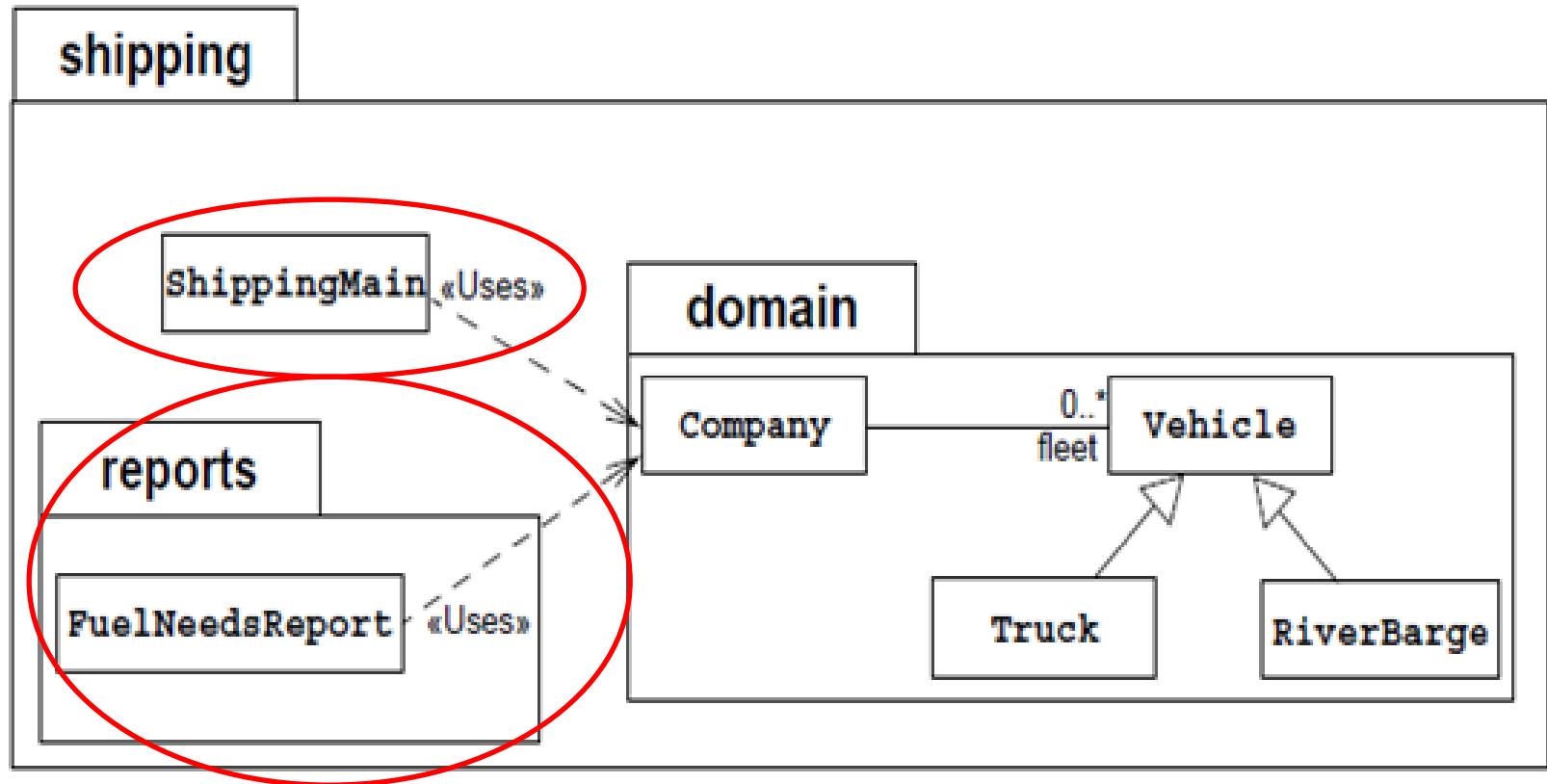


```
public abstract class Vehicle {  
    public abstract double calcFuelEfficiency();  
    public abstract double calcTripDistance();  
}
```


```
public class Truck extends Vehicle {  
    public Truck(double maxLoad) {...}  
    public double calcFuelEfficiency() {  
        /* calculate the fuel consumption of a truck at a given load */  
    }  
    public double calcTripDistance() {  
        /* calculate the distance of this trip on highway */  
    }  
}
```

```
public class RiverBarge extends Vehicle {  
    public RiverBarge(double maxLoad) {...}  
    public double calcFuelEfficiency() {  
        /* calculate the fuel efficiency of a river barge */  
    }  
    public double calcTripDistance() {  
        /* calculate the distance of this trip along the river-ways */  
    }  
}
```

案例分析(7): has a



```
public class FuelNeedsReport {  
    private Company company;  
  
    public FuelNeedsReport(Company company) {  
        this.company = company;  
    }  
  
    public void generateText(PrintStream output) {  
        Vehicle1 v;  
        double fuel;  
        double total_fuel = 0.0;  
  
        for ( int i = 0; i < company.getFleetSize(); i++ ) {  
            v = company.getVehicle(i);  
        }  
    }  
}
```



```
// Calculate the fuel needed for this trip
```

```
fuel = v.calcTripDistance() / v.calcFuelEfficiency();
```

```
output.println("Vehicle " + v.getName() + " needs "  
               + fuel + " liters of fuel.");
```

```
total_fuel += fuel;
```

```
}
```

```
output.println("Total fuel needs is " + total_fuel + " liters.");
```

```
}
```

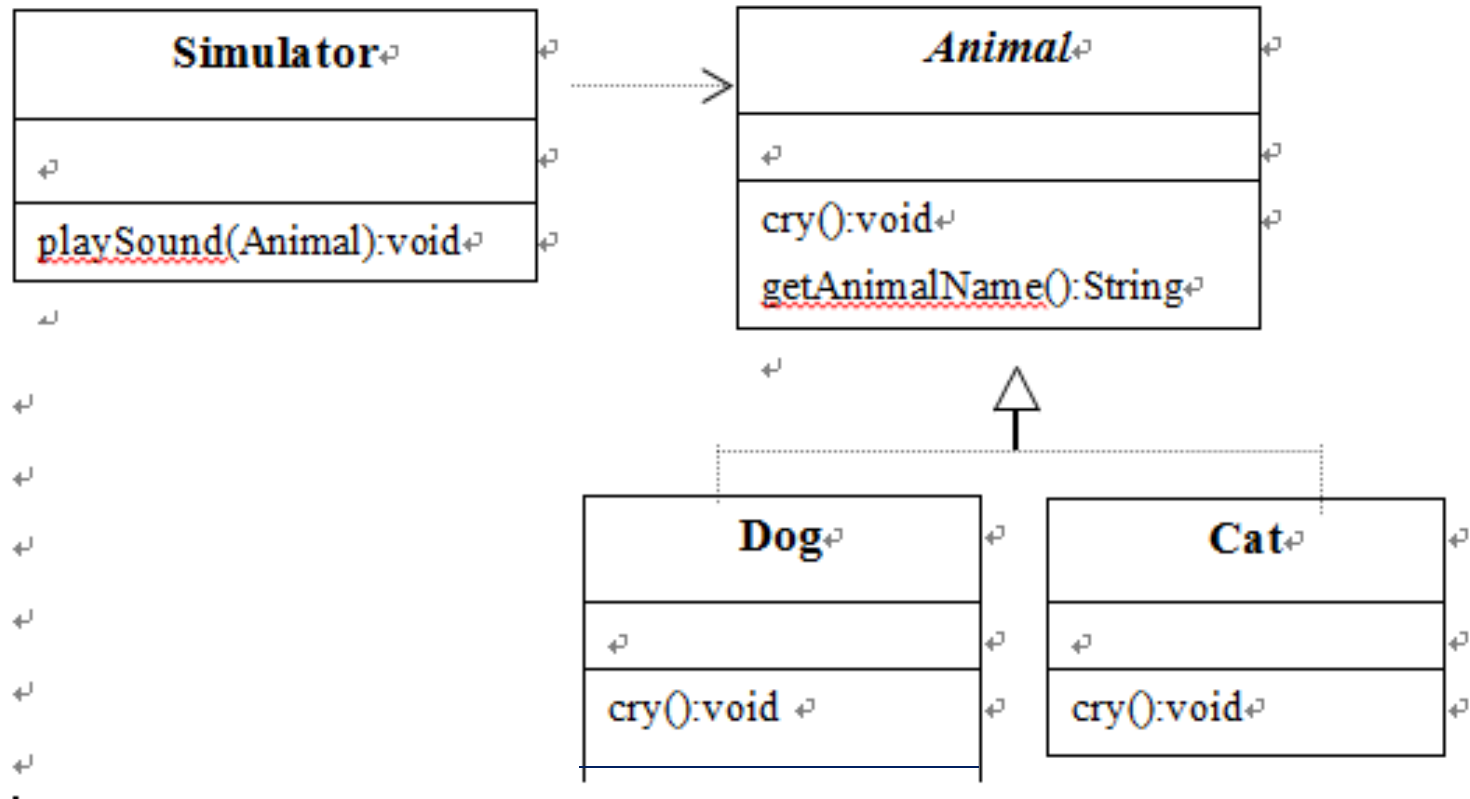
```
}
```

```
public class ShippingMain {  
    public static void main(String[] args) {  
        Company c = new Company();  
  
        // populate the company with a fleet of vehicles  
        c.addVehicle( new Truck(10000.0) );  
        c.addVehicle( new Truck(15000.0) );  
        c.addVehicle( new RiverBarge(500000.0) );  
        c.addVehicle( new Truck(9500.0) );  
        c.addVehicle( new RiverBarge(750000.0) );  
  
        FuelNeedsReport report = new FuelNeedsReport(c);  
        report.generateText(System.out);  
    }  
}
```


案例7

- 设计一个动物声音“模拟器”，希望模拟器可以模拟许多动物的叫声。要求如下：
- 编写抽象类Animal
 - Animal抽象类有2个抽象方法cry()和getAnimalName()，即要求各种具体的动物给出自己的叫声和种类名称。
- 编写Animal类的子类：Dog，Cat类
- 编写模拟器类Simulator
 - 该类有一个playSound(Animal animal)方法，该方法的参数是Animal类型。即参数animal可以调用Animal的子类重写的cry()方法播放具体动物的声音、调用子类重写的getAnimalName()方法显示动物种类的名称。
- 编写主类Application（用户程序）

案例7



代码解析

```
public abstract class Animal {  
    public abstract void cry();  
    public abstract String getAnimalName();  
}
```

```
public class Cat extends Animal {  
    public void cry() {  
        System.out.println("喵喵...喵喵");  
    }  
    public String getAnimalName() {  
        return "猫";  
    }  
}
```

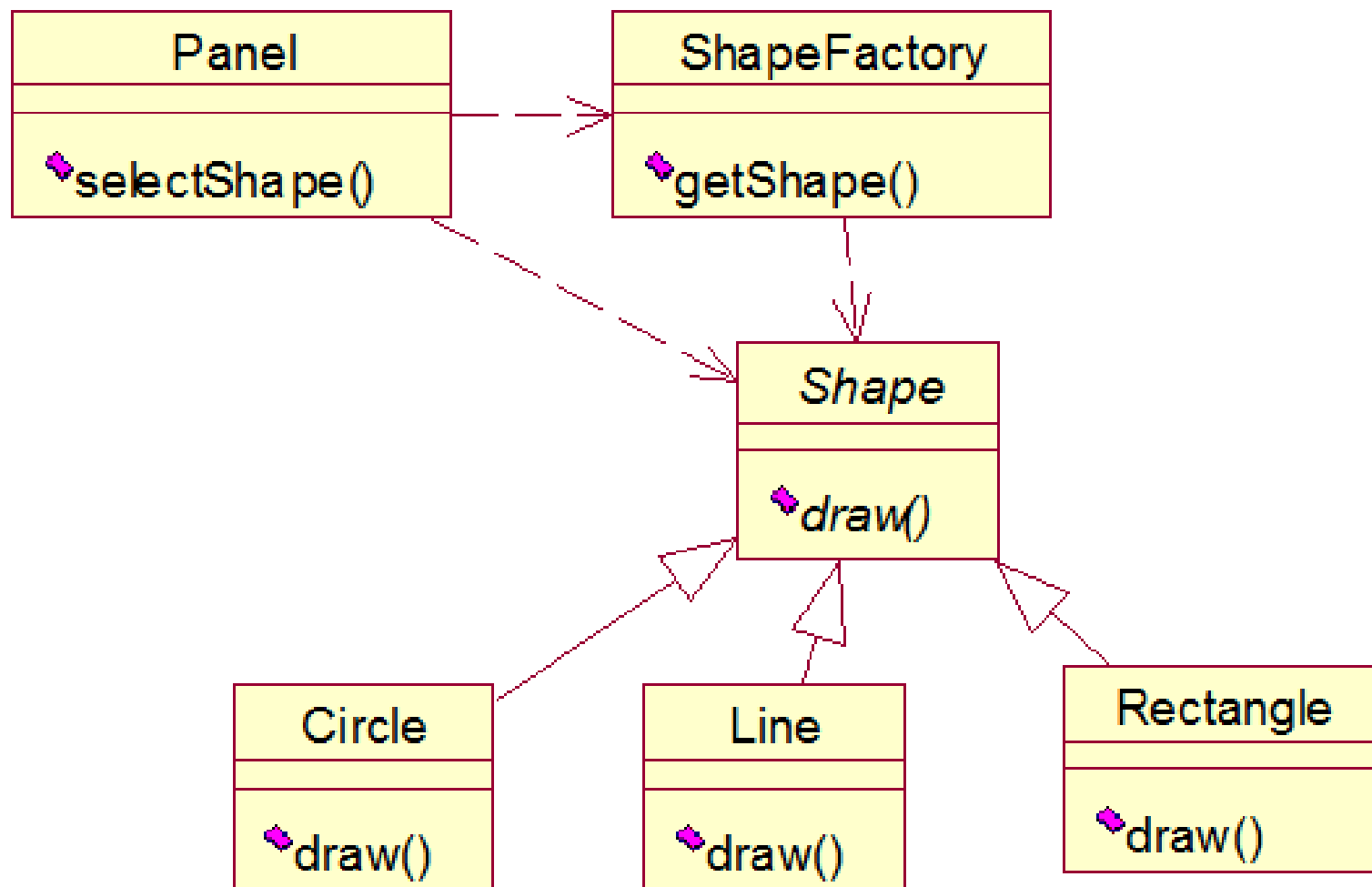
代码解析

```
public class Dog extends Animal {  
    public void cry() {  
        System.out.println("汪汪...汪汪");  
    }  
    public String getAnimalName() {  
        return "狗";  
    }  
}
```

```
public class Simulator {  
    public void playSound(Animal animal) {  
        System.out.print("现在播放" +  
            animal.getAnimalName() + "类的声音:");  
        animal.cry();  
    }  
}
```

```
public class Application {  
    public static void main(String args[]) {  
        Simulator simulator = new Simulator();  
        simulator.playSound(new Dog());  
        simulator.playSound(new Cat());  
    }  
}
```

案例8（工厂模式）（解耦）



```
abstract public class Shape { // 抽象类
    abstract void draw(); // 抽象方法
}

class Circle extends Shape { // 继承Shape类
    public void draw() {
        System.out.println("draw a circle");// 模拟画圆的行为
    }
}

class Line extends Shape { // 继承Shape类
    public void draw() {
        System.out.println("draw a line");// 模拟画直线的行为
    }
}

class Rectangle extends Shape { // 继承Shape类
    public void draw() {
        System.out.println("draw a rectangle");// 模拟画长方形的行为
    }
}
```

```
import java.util.HashMap;
import java.util.Map;
public class ShapeFactory {
    /** 定义形状类型常量 */
    public static final int SHAPE_TYPE_CIRCLE = 1;
    public static final int SHAPE_TYPE_RECTANGLE = 2;
    public static final int SHAPE_TYPE_LINE = 3;
```

```
private static Map<Integer,String> shapes
    = new HashMap<Integer,String>();
```

```
static {
    // 静态代码块，当Java虚拟机加载ShapeFactory类的代码时，就会执行这段代码
    // 建立形状类型和形状类名的对应关系
    shapes.put(new Integer(SHAPE_TYPE_CIRCLE), "Circle");
    shapes.put(new Integer(SHAPE_TYPE_RECTANGLE), "Rectangle");
    shapes.put(new Integer(SHAPE_TYPE_LINE), "Line");
}
```

*/** 构造具体的Shape对象，这是一个静态方法 */*

```
public static Shape getShape(int type){
```

```
    try {
```

```
        //获得与形状类型匹配的形状类名
```

```
        String className = shapes.get(new Integer(type));
```

```
        //运用Java反射机制构造形状对象
```

```
        return (Shape)Class.forName(className).newInstance();
```

```
    } catch (Exception e) {
```

```
        return null;
```

```
    }
```

```
}
```

```
}
```



```
import java.io.*;
public class Panel {
    public void selectShape() throws Exception {
        System.out.println("请输入形状类型: ");
        // 从控制台读取用户输入形状类型
        BufferedReader input = new BufferedReader(new InputStreamReader(
            System.in));
        int shapeType = Integer.parseInt(input.readLine());
        // 获得形状实例
        Shape shape = ShapeFactory.getShape(shapeType);
        if (shape == null) {
            System.out.println("输入的形状类型不存在");
        } else {
            shape.draw(); // 画形状
        }
    }

    /** 这是整个软件程序的入口方法 */
    public static void main(String[] args) throws Exception {
        new Panel().selectShape();
    }
}
```

小结：多态的优点：

- 借助于继承和动多态，派生类的功能可以被基类的引用变量引用，这叫**向后兼容**，从而可以提高程序的可扩充性和可维护性, 程序的调用界面清楚，可读性好，并解决代码冗余。
- 课后希望同学们总结一下工厂模式的使用

- 多态的必要性
- 静多态和动多态
- 方法重载，方法覆盖
- 多态的优点及运行机制
- 抽象方法
- 抽象类