动态绑定和多态

- 动态绑定是指在执行期间（非编译期间）判断所引用对象的实际类型，根据其实际类型调用其相应的成员方法
    - 非private方法不存在继承，即不存在覆盖
- 对象直接绑定（对象.属性的方式）的成员变量由对象的申明类型决定（静态绑定）
- 动态绑定的三个必要条件：
    - 要用继承
    - 要用重写
    - 父类引用指向子类对象

实例一：

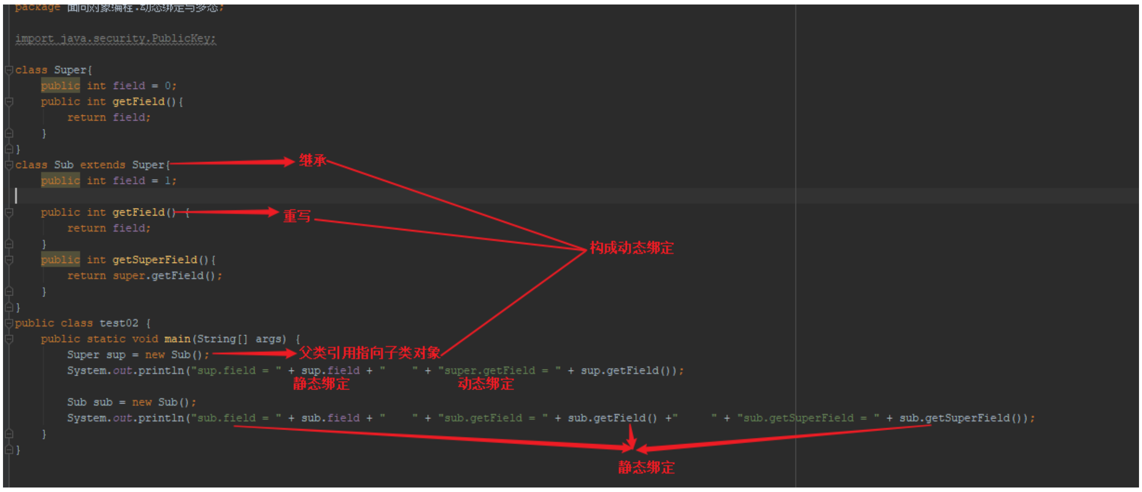
```
package 面向对象编程.动态绑定与多态;


class Animal{
    private String name;
    Animal(String name){
        this.name = name;
    }
    public void fun(){
        System.out.println("动物叫声");
    }
}
class Cat extends Animal{
    private String eyesColor;
    Cat(String name,String eyesColor){
        super(name);
        this.eyesColor = eyesColor;
    }
    public void fun(){
        System.out.println("猫叫声");
    }
}
class Dog extends Animal{
    private String furColor;
    Dog(String name,String furColor){
        super(name);
        this.furColor = furColor;
    }
    public void fun(){
        System.out.println("狗叫声");
    }
}
class Lady{
    private String name;
    private Animal pet;
    Lady(String name,Animal pet){
        this.name = name;
        this.pet = pet;
    }
    public void myPetFun(){
        pet.fun();
    }
}
public class test01 {
    public static void main(String[] args) {
        Cat cat = new Cat("cat","blue");
        Dog dog = new Dog("dog","white");

        Lady lady = new Lady("汤家平",cat);
        lady.myPetFun();

        Lady lady1 = new Lady("张三",dog);
        lady1.myPetFun();
    }
}

结果:
    猫叫声
    狗叫声
```

实例二:

- 动态绑定和静态绑定的区别

```
package 面向对象编程.动态绑定与多态;

import java.security.PublicKey;

class Super{
    public int field = 0;
    public int getField(){
        return field;
    }
}
class Sub extends Super{
    public int field = 1;

    public int getField() {
        return field;
    }
    public int getSuperField(){
        return super.getField();
    }
}
public class test02 {
    public static void main(String[] args) {
        Super sup = new Sub();
        System.out.println("sup.field = " + sup.field + "    " + "super.getField = " + sup.getField());

        Sub sub = new Sub();
        System.out.println("sub.field = " + sub.field + "    " + "sub.getField = " + sub.getField() +"     " + "sub.getSuperField
    }
}

结果:
    sup.field = 0    super.getField = 1
    sub.field = 1    sub.getField = 1     sub.getSuperField = 0
```

分析:



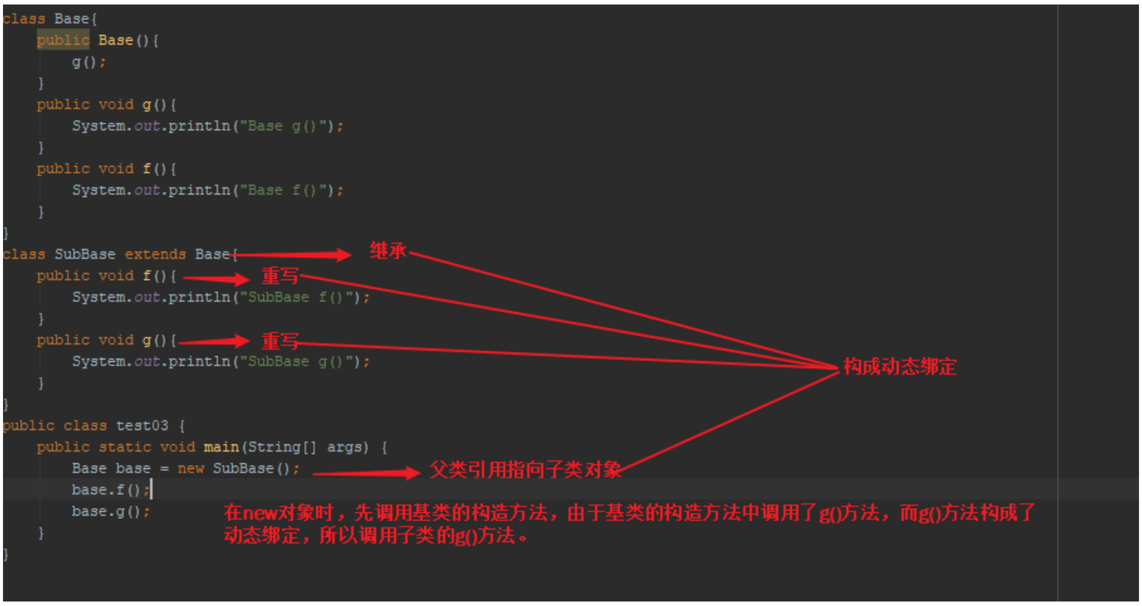
```
package 面向对象编程.动态绑定与多态;

import java.security.PublicKey;

class Super{
    public int field = 0;
    public int getField(){
        return field;
    }
}
class Sub extends Super{
    public int field = 1;

    public int getField() {
        return field;
    }
    public int getSuperField(){
        return super.getField();
    }
}
public class test02 {
    public static void main(String[] args) {
        Super sup = new Sub();
        System.out.println("sup.field = " + sup.field + "    " + "super.getField = " + sup.getField());

        Sub sub = new Sub();
        System.out.println("sub.field = " + sub.field + "    " + "sub.getField = " + sub.getField() +"     " + "sub.getSuperField = " + sub.getSuperField());
    }
}
```

实例三：

- 构造方法中也存在动态绑定

```
package 面向对象编程.动态绑定与多态;

class Base{
    public Base(){
        g();
    }
    public void g(){
        System.out.println("Base g()");
    }
    public void f(){
        System.out.println("Base f()");
    }
}
class SubBase extends Base{
    public void f(){
        System.out.println("SubBase f()");
    }
    public void g(){
        System.out.println("SubBase g()");
    }
}
public class test03 {
    public static void main(String[] args) {
        Base base = new SubBase();
        base.f();
        base.g();
    }
}

结果：
    SubBase g()
    SubBase f()
    SubBase g()
```

分析：



```java
class Base{
    public Base(){
        g();
    }
    public void g(){
        System.out.println("Base g()");
    }
    public void f(){
        System.out.println("Base f()");
    }
}
class SubBase extends Base{
    public void f(){
        System.out.println("SubBase f()");
    }
    public void g(){
        System.out.println("SubBase g()");
    }
}
public class test03 {
    public static void main(String[] args) {
        Base base = new SubBase();
        base.f();
        base.g();
    }
}
```

在new对象时，先调用基类的构造方法，由于基类的构造方法中调用了g()方法，而g()方法构成了动态绑定，所以调用子类的g()方法。