**重写(Override)**

重写是子类对父类的允许访问的方法的实现过程进行重新编写, 返回值和形参都不能改变。即外壳不变，核心重写！

重写的好处在于子类可以根据需要，定义特定于自己的行为。 也就是说子类能够根据需要实现父类的方法。

重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如： 父类的一个方法申明了一个检查异常 IOException，但是在重写这个方法的时候不能抛出 Exception 异常，因为 Exception 是 IOException 的父类，抛出 IOException 异常或者 IOException 的子类异常。

在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：

TestDog.java 文件代码：

class Animal{

public void move(){

System.out.println("动物可以移动");

}

}

class Dog extends Animal{

public void move(){

System.out.println("狗可以跑和走");

}

}

public class TestDog{

public static void main(String args[]){

Animal a = new Animal(); // Animal 对象

Animal b = new Dog(); // Dog 对象

a.move();// 执行 Animal 类的方法

b.move();//执行 Dog 类的方法

}

}

以上实例编译运行结果如下：

动物可以移动

狗可以跑和走

在上面的例子中可以看到，尽管 b 属于 Animal 类型，但是它运行的是 Dog 类的 move方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，Java 虚拟机(JVM)指定对象的类型并且运行该对象的方法。

因此在上面的例子中，之所以能编译成功，是因为 Animal 类中存在 move 方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

TestDog.java 文件代码：

class Animal{

public void move(){

System.out.println("动物可以移动");

}

}

class Dog extends Animal{

public void move(){

System.out.println("狗可以跑和走");

}

public void bark(){

System.out.println("狗可以吠叫");

}

}

public class TestDog{

public static void main(String args[]){

Animal a = new Animal(); // Animal 对象

Animal b = new Dog(); // Dog 对象

a.move();// 执行 Animal 类的方法

b.move();//执行 Dog 类的方法

b.bark();

}

}

以上实例编译运行结果如下：

TestDog.java:30: cannot find symbol

symbol : method bark()

location: class Animal

b.bark();

^

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。