类与接口（5）：Java 多态、方法重写、隐藏

一、Java 多态性

面向对象的三大特性：封装、继承、多态。

多态的类型，分为以下两种：

- 编译时多态： 指的是方法重载。编译时多态是在编译时确定调用处选择那个重载方法，所以也叫静态多态，算不上真正的多态。所以，一般说的多态都是运行时的多态。
- 运行时多态： 由于方法重写，所以想要确定引用变量所调用的方法的入口，必须根据运行时的引用变量所指向的实例对象来确定。从而使得同一个引用变量调用同一个方法，但不同的实例对象表现出不同的行为。再简单点来说，就是在运行时，可以通过指向基类的指针，来调用实现子类中的方法。

下面讲的多态都是指运行时的多态。

多态的定义： 指允许不同类的对象对同一消息做出不同的响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）；

上面的定义参考了网上的说法，定义中的不同类，都是由同一个基类扩展而来。

多态的好处：

- 可替换性（substitutability）。多态对已存在代码具有可替换性。例如，draw 函数对圆 Circle 类工作，对其他任何圆形几何体，如圆环，也同样工作。
- 可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。
- 接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。如图 8.3 所示。图中超类 Shape 规定了两个实现多态的接口方法，computeArea()以及 computeVolume()。子类，如 Circle 和 Sphere 为了实现多态，完善或者覆盖这两个接口方法。
- 灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。
- 简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

java 的多态性三要素：

- 继承；
- 重写；
- 父类的引用指向子类的引用

多态性的实现： 依靠动态绑定；

绑定： 将一个方法调用与方法主体关联起来。

前期绑定：在程序执行前绑定 ,由编译器和链接程序完成 ,C 语言的函数调用便是前期绑定。
动态绑定： 也称**后期绑定**。在运行时，根据具体的对象类型进行方法调用绑定。除了 static 方法、final 方法（private 方法也是 final 方法），其他方法都是动态绑定；

二、方法重写与隐藏

方法重写：就是子类中覆盖了从父类继承下来的方法(不是所有的父类方法都可以覆盖），从而使得通过父类的引用来调用重写后的方法。也就是说，父类类型与子类类型只保留重写后的方法。

隐藏： 覆盖是相对重写而言的。当子类中出现与父类相同的方法时，重写是子类与父类只保持一份，但方法隐藏则是子类与父类各自独立保持一份，也就两份。从父类继承下来的成员中，除了部分方法是可以重写外，其余成员都是隐藏，如变量、内部类、静态方法等。
注意： final 方法既不能重写，也不能隐藏。

```
class ParentClass{
    public int a = 5;
    protected final String name = "parentClass";
    public final void finalMethod() {//final 方法，子类既不能重写，也不能隐藏
        System.out.println("final 方法");
    }

    public static void monday() {//静态方法
        System.out.println("父类 ParentClass 的 monday()方法");
    }

    public void count() {//可继承的成员方法
        System.out.println("父类 ParentClass 的 count()方法");
    }

    class InnerClass{//内部类
        public InnerClass() {
            System.out.println("父类 ParentClass 的内部类");
        }
    }
}

class ChildClass extends ParentClass{
    public int a = 5;
    protected final String name = "ChildClass";
```

```
    /*//编译不通过
     * public final void finalMethod() {
        System.out.println("final 方法");
    }*/

    public static void monday() {//静态方法
        System.out.println("子类 ChildClass 的 monday()方法");
    }

    public void count() {//可继承的成员方法
        System.out.println("子类 ChildClass 的 count()方法");
    }

    class InnerClass{//内部类
        public InnerClass() {
            System.out.println("子类 ChildClass 的内部类");
        }
    }
}

public class MyTest {
    public static void main(String[] args) {
        ChildClass child = new ChildClass2();
        ParentClass parent = child; //类型上转
        System.out.println("---------------变量的隐藏测试-----------------");
        child.a = 10;
        System.out.println("parent.a： "+parent.a);
        System.out.println("child.a： "+child.a);
        System.out.println("\n---------------静态方法的隐藏测试-----------------");
        parent.monday();
        child.monday();
        System.out.println("\n---------------方法的重写测试-----------------");
        parent.count();
        child.count();
        System.out.println("\n---------------内部类的隐藏测试-----------------");
        ParentClass.InnerClass pa = parent.new InnerClass();
        ChildClass.InnerClass ch = child.new InnerClass();
    }
}
```

运行结果：

```
--------------变量的隐藏测试----------------
   parent.a： 5
   child.a： 10

--------------静态方法的隐藏测试----------------
   父类 ParentClass 的 monday()方法
   子类 ChildClass 的 monday()方法

--------------方法的重写测试----------------
   子类 ChildClass 的 count()方法
   子类 ChildClass 的 count()方法

-------------内部类的隐藏测试----------------
   父类 ParentClass 的内部类
   子类 ChildClass 的内部类
```

上面的例子中，只有 count()方法是被重写了，父类类型与子类类型只保持重写后的方法，而其他成员都是隐藏，父类类型保持一份，子类类型也保持一份。

方法重写的条件

- 重写的方法是子类从父类继承下来的实例方法，不能是静态方法
- 子类重写后的方法的返回类型必须是原父类方法的返回类型的可替换类型
- 子类重写后的方法的访问权限不能比原父类方法的访问权限低；
- 子类重写后的方不能比父类方法抛出更多的异常；
- 当重写泛型方法时，先进行类型擦除。再按照上面的 4 个小点，重写类型擦除后的方法;

可替换类型补充：

- 对于返回类型是基本类型、void，重写方法的返回类型必须是一样；
- 对于返回类型是引用类型，返回类型可替换成该类型的子类型;

```
class ParentClass{//父类
    public int count() {//
        return 0;
    }

    Object method() {
        return "aa";
    }

    <T extends ParentClass> T getValue(T t) {//泛型方法
       System.out.println();
```

```
        return t;
    }
}


class ChildClass extends ParentClass{//子类
    public int count() {//重写 count()方法，由于返回类型是基本类型，不能变，必须是一致
        return 0;
    }


    public String method() {//重写 method()：访问权限增大，返回类型是 Object 的子类 String
        return "aa";
    }


    ChildClass getValue(ParentClass ch) {//重写泛型方法 getValue()
        return null;
    }
}
```

解析一下此例子中的泛型方法重写，父类中的泛型方法 getValue()进行类型擦除后，是：

```
ParentClass getValue(ParentClass t){
    return null;
}
```

所以，子类 ChildClass 的方法重写是合理的。