



接口： Interface

接口是一个特殊的类，与 class 是平级的。

特点：

1. 它是一个类， 是一个抽象类，是一个特殊，最特殊的抽象类
2. 接口中的每一个属性都是公开，静态，常量（public static final）
3. 接口中的每一个方法都是公开抽象方法（public abstract）

要实现一个接口，就是要实现接口中所定义的所有方法！（抽象方法）

关键字： implements

由于接口的特点 3，接口中的每一个方法都是公开抽象方法（public abstract），所以在定义接口方法时可以省略掉这两个关键字，但是，在实现接口的类中，修饰符 public 一定不能省，

一定要得加上，否则编译会报错。（不满足多态）

接口存在多态，它能定义一个变量指向实现此接口的类的对象，所抽象类类似，它也只能做为编译时类型，不能做为运行时类型。

例如：

```
Interface IA {
```

```
    Int id = 5 ; //相当于： public static final int id = 5;
```

```
    Void show(); //相当于： public abstract void show();
```

```
}
```

```
Class IAImp1 implements IA {
```

```
    Public void show() {
```

```
        System.out.println(“第一个接口程式”);
```

```
    }
```

```
}
```

现在，在主方法中： 就可以使用： IA a = new IAImp1();

```
    a.show(); //调的是运行时类型中的
```

```
    show 方法
```

思考： 抽象与接口的不同？



1. 接口与接口之间可以支持多继承。(接口是一种特殊的类);
2. 一个类可以实现多个接口, 但只能继承一个类。
3. 一个类在继承另一个类后, 还可以实现多个接口。 规则是: 先写继承类, 再写实现的接口, 多个接口之间用 ‘,’ 号分开。

二. instanceof 操作符。

用法: 对象 instanceof 类

注: 此 ‘对象’ 指的是运行时类型, 也就是它真正的类型。

如:

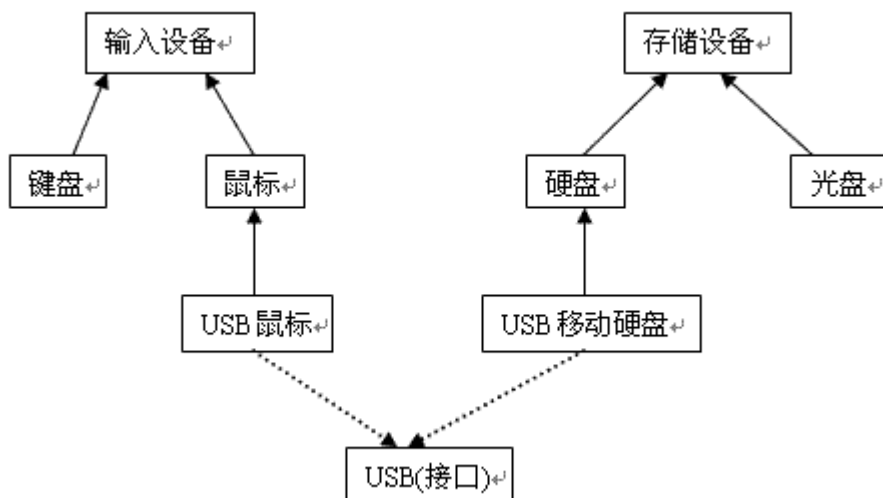
```
Object o = new String("abc");
if(o instanceof String) { //此处, o 运行时类型是 String, 编译时类型
是 Object.
    String s = (String)o;
    System.out.println(s);
}
```

Instanceof 操作符一般是用来 保证类型强转换不会出错的一个很好的判断。
如:

```
Class Animal { }
Class Cat extends Animal { }
Class Dog extends Animal { }
....
Animal a = new Dog();
Animal b = new Cat();
System.out.println( a instanceof Animal ); // true
System.out.println( b instanceof Cat ); // true;
System.out.println( a instanceof Cat ); // false
```

思考 2: 为什么要用接口?

1. 多继承, 没有增加类之间关系复杂程度, 同时, 还保存了 JAVA 的树形继承结构。
(注: 其实接口并不能完全地支持多重继承, 这个我们后面会再讲到这种情况)
2. 用接口可以实现混合类型。



如上图所示，两个不相干的设备，通过 **USB 接口**，可以把它们联系起来。我们来看一个例子： 假设系统中有一个方法叫 **USB 适配器**，它就以 **USB 接口** 作为参数，它可以让任何的 **USB 设备**（实现了 **USB 接口** 的类）接入到系统，要求 **USB 接口** 定义一方法，叫传输，这样就能保证只要是 **USB 设备**，就能接入到系统。

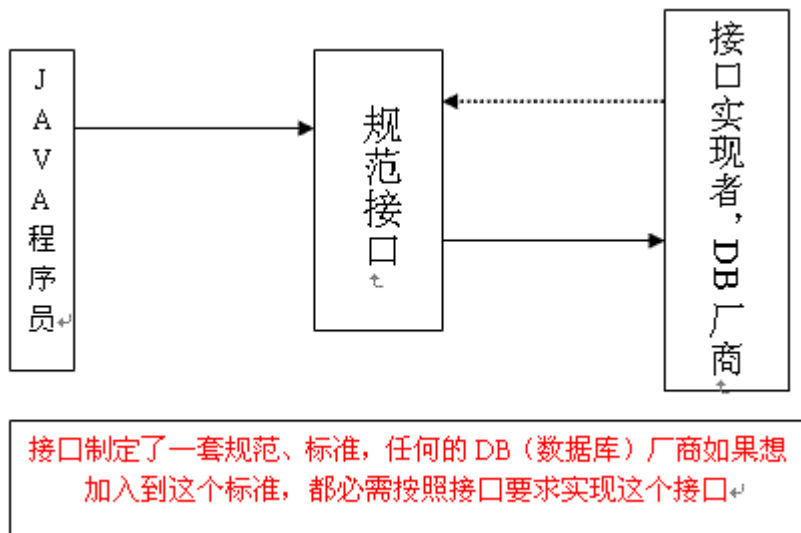
Coding:

```
Interface USB {  
    Void translate(); //定义一传输方法  
}  
Class USBMouse implements USB {  
    Public void translate() {  
        System.out.println(“开始使用鼠标”);  
    }  
}  
Class USBDisk implements USB {  
    Public void translate() {  
        System.out.println(“开始传输数据”);  
    }  
}  
  
Class System {  
    Public static void main(String[] args) {  
        USB u1 = new USBMouse();  
        USB u2 = new USBDisk();  
        usbAdapter(u1);  
        usbAdapter(u2);  
    }  
    Public static void usbAdapter(USB u) {
```



```
        u.translate();  
    }  
}
```

3. 接口使得标准有了一个制订者。（接口就是标准制订者）



如上图所示,规范: 定义一个 Driver 接口, 由 SUN 公司完成, 假如它里面有一个 connect() 方法。

但是这个方法 SUN 公司自己无法去实现呀, 所以只能由那些愿意实现此 Driver 接口的 DB 厂商去实现,

这样既保证了 DB 厂商商业机密不会被卸漏, 对于 JAVA 程序员来说, 也不必去了解 connect() 方法及它是如何实现的。

这样就将 ‘方法的实现者、 方法的使用者’ 分开了。这其实就是解耦合的要求, 接口可以做到这一点, 所以说接口是种解耦工具。

Coding:

```
Interface Driver {  
    void connect() ;  
}  
Class OracleDriver implements Driver {  
    Public void connect() {  
        System.out.println("oracle supply");  
    }  
}  
Class MySQLDriver implements Driver {  
    Public void connect() { System.out.println("mysql  
supply"); }  
}
```

而对于我们的 JAVA 程序员来说, 只以简单地使用: Driver d = new OracleDriver(); d = new MySQLDriver(); 来创建一个 Driver 对象, 然后只是很简单地调用: d.connect(); 就可以了。



因为根据多态的原则，d 会找到它自己真正的类型的 connect() 方法并进行调用，JAVA 程序员不用关心细节。

以上三点就是为什么要使用接口的重要原因，它大大提高了程序的复用性，可插入性，而且极大地降低了对对象之间的耦合度。

最后，记住使用接口的原则：

1. 尽量地针对接口去编程，（尽量地使用接口作为参数，返回值等）
2. 接口隔离原则，尽量地把大接口拆成小接口，这样更适合于‘定制’

三. Object 类。

Object 类在 JAVA 中，是所有类的根类，任一自定义的类，如没有显示地继承于任一类，则默认的父亲类就是 Object 类，所以，Object 是整个树状结构的根。

既，也就是说，Object 可以作为任一类型的对象的编译时类型。

现在，让我们来认识并掌握 Object 类中最常用的三个方法：

1. finalize() 方法 在确定一个对象要销毁时，在销毁此对象之前 JVM(JAVA 虚拟机)会自动调用此方法。

a) 有一点要注意，因为我们不知道对象何时会被销毁，所以此方法根本不知道何时会被调用，所以最好不要依赖此方法，一般不推荐使用。

2. toString() 方法： 返回一个字符串

在使用 System.out.println(o) 打印一个对象时，会自动地调用 o.toString() 方法来把一个对象转变成字符串，然后打印出来。注：此处 o 是指一个对象。

所以，JAVA 推荐，在你自定义的类中，应该‘覆盖’此方法，以便能正确地打印出自定义的对象。

```
如： class Employee {  
    Private String name;  
    Private int age;  
    Private double salary;  
    ...  
    Public String toString() {  
        Return name+"'" +age+"'" +salary;  
    }  
}
```

... ..

```
Employee e = new Employee("yejf",28, 15000.0);  
System.out.println(e); // 相当于 System.out.println(e.toString());
```

3. Equals(Object o) 方法：判断当前对象是否与此对象 O 相等。

a) 注：此方法中 Object 中的实现只是很简单地用 == 来比较两对象的地址而已。 Boolean equals(Object o) { return this == o; }

b) 所以，在我们自己定义的类中，应该覆盖此方法。

c) 写一个好的 equals(Object o) 方法的原则如下：



- i. 自反性: `o.equals(o)` 要永远为 `true`
- ii. 对称性: 如果 `o1.equals(o2)` 为 `true`, 则 `o2.equals(o1)` 也一定为 `true`;
- iii. 传递性: 如果 `o1.equals(o2)` 为 `true`, `o2.equals(o3)` 为 `true`, 则 `o1.equals(o3)` 也一定为 `true`.

下面我们根据以上三点原则来写一个标准的 `equals(Object o)` 方法

```
Class Student {  
    Private String name;  
    Private int age;  
    ...  
    Public Boolean equals(Object o) {  
        If(this == o) return true; //自反性判断, 同时也可以提高效率, 避免不必要的比较。  
        If(o == null) return false; //任何值与 null 比较, 一定不相等  
        If(this.getClass() != o.getClass() ) return false; //类型不同, 就没什么好比较的了  
        Student s = (Student)o;  
        If(this.age == s.age && this.name.equals(s.name) ) return true;  
        Else return false;  
    }  
}
```

注: 一般来说, 自定义的类型都应该覆盖 `toString()` 和 `equals(Object o)` 这两个方法.

作者: 叶加飞 (Steven Ye)
mailto: leton.ye@gmail.com