面向对象-day03

今日学习内容：

- 接口的定义
- 接口和接口之间的继承关系
- 接口和类之间的实现关系
- 多态的操作
- 多态对象调用方法问题
- 引用类型的类型转换
- 多态的好处

今日学习目标：

- 掌握接口定义的语法
- 掌握接口和接口之间的继承关系
- 重点掌握接口和类之间的实现关系
- 掌握多态对象的创建和使用
- 重点掌握多态对象调用方法的执行过程
- 了解引用数据类型的类型转换
- 掌握多态的好处及USB案例

多态思想

1.1、接口

1.1.1、接口概述（了解）

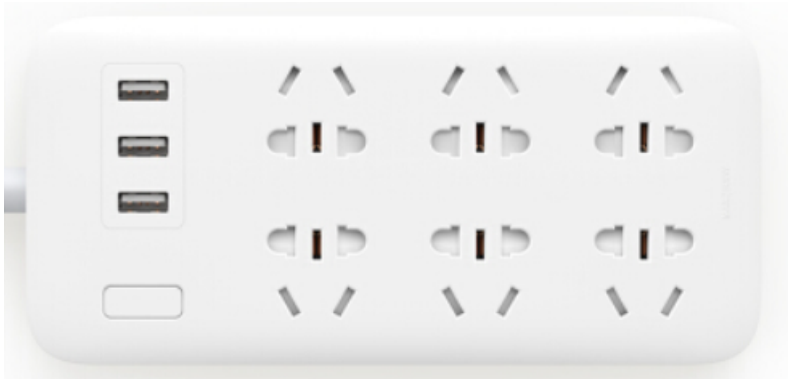


图1-1 GB 209912008 标准插座

我们要完成一个工程，需要一个插座

思考1：去市场买个回来！=> 市场上有公牛、小米... => 小米和公牛认识吗？什么原因导致公牛和小米的插座我都可以用

接口是一种约定的规范，是多个抽象方法的集合。仅仅只是定义了应该有哪些功能，本身不实现功能，至于每个功能具体怎么实现，就交给实现类完成。

接口中的方法是抽象方法，并不提供功能实现，**体现了规范和实现相分离的思想**，也体现了组件之间低耦合的思想。

所谓耦合度，表示组件之间的依赖关系。依赖关系越多，耦合性越强，同时表明组件的独立性越差，在开发中往往提倡降低耦合性，可提高其组件独立性，举一个低耦合的例子。

电脑的显卡分为集成显卡和独立显卡：

- 集成显卡：显卡和主板焊死在一起，显卡坏了，只能换主板
- 独立显卡：显卡和主板相分离，显卡插到主板上即可，显卡坏了，只换显卡，不用换主板

接口也体现的是这种低耦合思想（在开发过程中，如果想要解耦，一定要想到接口），接口仅仅提供方法的定义，却不提供方法的代码实现。那么得专门提供类并去实现接口，再覆盖接口中的方法，最后实现方法的功能，在多态案例中再说明。

1.1.2、接口定义和多继承性（重点掌握）

接口可以认为是一种特殊的类，但是定义类的时候使用class关键字，定义接口使用interface关键字。

```
public interface  接口名{
    //抽象方法1();
    //抽象方法2();
    //抽象方法2();
}

// 接口命名规范:大写的驼峰命名法。
```

接口表示具有某些功能的事物，接口名使用名词，有人也习惯以**I**打头如IWalkable.java。

接口定义代码：

```
public interface IWalkable {
    void walk();
}
```

接口中的方法都是公共的抽象方法，等价于：

```
public interface IWalkable {
    public abstract void walk();
}
```

拓展 （lambda 讲解）：从Java8开始, Java支持在接口中定义有实现的方法, 如:

```
public interface IWalkable {
    public abstract void walk();//抽象方法

    default void defaultMethod(){
        System.out.println("有默认实现的方法，属于对象");
    }
    static void defaultMethod(){
        System.out.println("有默认实现的方法，属于类");
    }
}
```

在java中，接口也可以继承，一个接口可以继承多个接口，也就是说一个接口可以同时继承多个接口，如两栖动物可以行走也可以拥有。

可行走规范：

```
public interface IWalkable {
    void walk();
}
```

可游泳规范：

```
public interface ISwimable {
    void swim();
}
```

两栖动物规范，即可以游泳，又可以行走。

```
public interface IAmphibiable  extends IWalkable,ISwimable{

}
```

此时子接口能继承所有父接口的方法。

1.1.3、接口实现类（重点掌握）

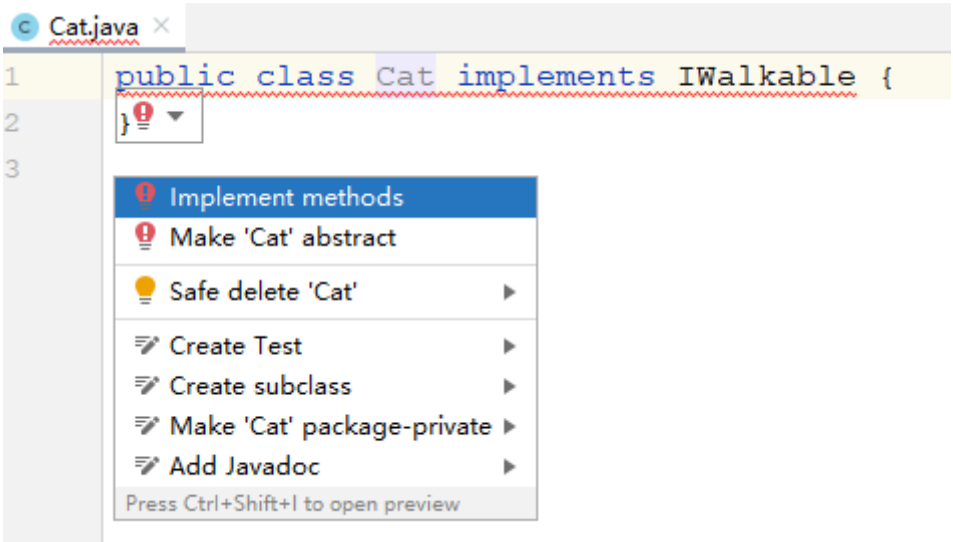
[1]. 因为接口中的方法是抽象的，没有方法体，所以接口是不能创建对象的，此时必须定义一个类去**实现**接口，并覆盖接口中的方法，这个类称之为**实现类**，实现类和接口之间的关系称之为实现关系（implements）。

```
public class 类名 implements 接口名{
    // 覆盖接口中抽象方法
}
```

实现类实现接口后，必须实现接口中的所有抽象方法，完成功能代码，此时接口和实现类之间的关系：

- 接口：定义多个抽象方法，仅仅定义有哪些功能，却不提供实现。
- 实现类：实现接口，实现接口中抽象方法，完成功能具体的实现。

如果实现类没有全部实现接口中的方法，要么报错，要么把实现类设置为抽象类（下图）。



需求：定义一个猫类（Cat）实现IWalkable接口，并创建对象调用方法。

```
public class Cat implements IWalkable{

    public void walk() {
        System.out.println("走猫步...");
    }
}
```

根据方法覆盖原则：子类方法的访问修饰符必须大于等于父类方法的访问修饰符，接口中的方法都是public修饰的，所以实现类中的方法只能使用public修饰。

需求：定义一个人类，实现ISwimable,IWalkable接口

```
public class Person implements ISwimable, IWalkable {
    @Override
    public void walk() {
        System.out.println("人类步行...");
    }

    @Override
    public void swim() {
        System.out.println("人类游泳...");
    }
}
```

总结：实现类实现了接口，一定具备接口中定义的能力。

[2]、实现类可以继承父类，但可以同时实现一个或多个接口，继承在前，实现在后。

定义一个青蛙类（Frog）继承于动物类（Animal），同时实现于会走路（IWalkable），会游泳（ISwimable）的接口，语法如下（记住定义语法即可）：

```
public class Frog extends Animal implements ISwimable,IWalkable{

    public void walk() {
        System.out.println("跳啊跳...");
    }

    public void swim() {
        System.out.println("游啊游..");
    }
}
```

[3]、接口是一种引用数据类型，可以用来声明变量，并接收(引用)所有该接口的实现类对象。如果要创建实现类对象，语法如下：

```
接口 变量 = new 实现类();
```

测试类：

```
public class CatDemo {
    public static void main(String[] args) {
        IWalkable walkable = new Cat();
        walkable.walk();

        // walkable 能不能接收其他的实现类呢？
        walkable = ?
    }
}
```

1.1.4、接口总结

- 接口表示一种规约（规范、标准），它里面定义了一些列抽象方法（功能），它可以被多个类实现。
- 实现类实现接口，必须实现接口中的所有抽象方法。我们经常说：**接口约定了实现类应该具备的能力。**

```
// Person 类作为IWalkable的实现类
=> IWalkable接口约定了walk()的规范，Person必须实现接口中的所有抽象方法
=> 实现类具有接口中定义的功能
=> 接口约定了实现类应该具备的功能
public class Person implements IWalkable{
    @Override
    public void walk() {
        System.out.println("人类走路...");
    }
}
```

- 面向接口编程（先熟悉名称）

1.2、多态（掌握）

1.2.1、多态概念

多态时面向对象三大特征：封装、继承、多态。

在继承关系，是一种 `is A` 的关系，也即`什么是什么`,也就说子类是父类的一种特殊情况，有如下代码：

例如：如果Student 继承于Person，我们就可以说，学生 is a 人类

```
public class Animal{}
public class Dog extends Animal{}   // Dog is a Animal
public class Cat extends Animal{}   // Cat is a Animal
```

那么我们可以认为狗和猫都是一种特殊的动物，那么可以使用动物类型来表示狗或猫。

```
Dog     d   =   new Dog();      //创建一只狗对象,赋给子类类型变量
Animal  a   =   new Cat();      //创建一只猫对象,赋给父类类型变量
```

此时对象（a）具有两种类型:

- **编译时类型：声明对象变量的类型——>Animal**
- **运行时类型：对象的真实类型 ——>new Dog**

当编译类型和运行类型不一致的时候，此时多态就产生了：

注意：编译类型必须是运行类型的父类或接口。

所谓**多态**，简单地理解，表示一个对象具有多种形态。

说的具体点就是同一引用类型变量调用同一方法时，由于引用实例不同，方法产生的结果不同。

```
Animal  a = null;
a   =  new    Dog();    //  a此时表示Dog类型的形态
a   =  new    Cat();    //  a此时表示Cat类型的形态
```

多态的前提，可以是继承关系（类和类），也可以是实现关系（接口和实现类），在开发中，一般都指接口和实现类之间的关系。

一言以蔽之：父类引用变量指向于子类对象，调用方法时实际调用的是子类的方法。

我家有一种动物，你猜它的叫声是怎么样的，猜不到，因为这个动物有多种形态。

- 如果该动物是狗，叫声是：旺旺旺...
- 如果该动物是猫，叫声是：妙妙妙...

多态操作有两种定义格式和操作语法：

- 操作继承关系（开发中不是很多）：

```
父类 变量名 = new 子类();
变量名.方法();
```

- 操作实现关系（开发中最频繁）：

```
接口 变量名 = new 实现类();
变量名.方法();
```

1.2.2、操作继承关系（掌握）

```
父类 变量名 = new 子类();
变量名.方法();
```

Animal类：

```
public class Animal {
    public void shut() {
        System.out.println("Animal...shout...");
    }
}
```

Cat类：

```
public class Cat extends Animal{
    public void shut() {
        System.out.println("妙妙妙...");
    }
}
```

Dog类：

```
public class Dog extends Animal{
    public void shut() {
        System.out.println("旺旺旺...");
    }
}
```

测试类：

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

        Animal animal = null;

        // 创建Cat对象
        animal = new Cat();
        animal.shut();

        // 创建Dog对象
        animal = new Dog();
        animal.shut();
    }
}
```

运行结果：

```
妙妙妙...
旺旺旺...
```

结论：父类引用变量指向于子类对象，调用方法时实际调用的是子类的方法。

1.2.3、操作实现关系（重点掌握）

```
接口 变量名 = new 实现类();
变量名.方法();
```

ISwimable 接口：

```
public interface ISwimable {
    void swim();
}
```

Fish类：

```
public class Fish implements ISwimable{
    public void swim() {
        System.out.println("游啊游...");
    }
}
```

测试类：

```
public class FishDemo {
    public static void main(String[] args) {
        // 创建Fish对象
        ISwimable fish = new Fish();
        fish = new Dog();//Dog也是实现了ISwimable接口的
        Fish f = new Fish();

        fish.swim();
    }
}
```

运行结果：

```
游啊游...
```

结论：接口引用变量指向实现类对象，调用方法时实际调用的是实现类实现接口的方法。

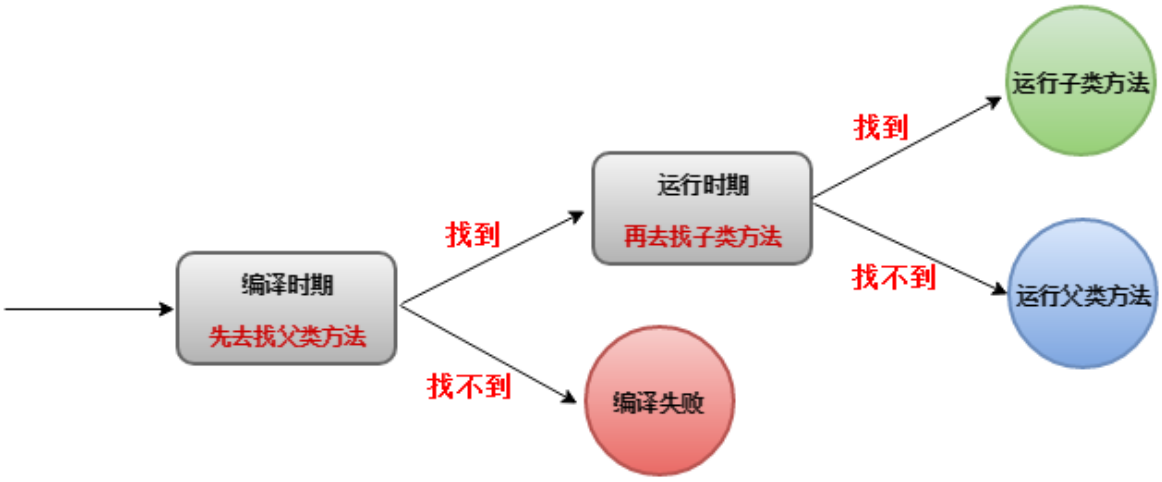
1.2.4、多态时方法调用问题（重点掌握）

把子类对象赋给父类变量，此时调用方法：

```
Animal animal = new Cat();
animal.shut();
```

那么 animal 对象调用的shut方法，是来自于Animal中还是Cat中？判断规则如下：

一张图，看懂到底调用的是哪一个类中的方法！



文字解释，先判断shout方法是否在父类Animal类中：

- 找不到：编译报错
- 找　到：再看shut方法是否在子类Cat类中：
- 找不到：运行父类方法
- 找　到：运行子类方法（这个才是真正的多态方法调用）

1.2.5、多态小结

```
所谓多态，就是一个对象的多种形态，这个对象一般都比较抽象，我们说一个动物 Animal a = null;
a = new Dog();
此时如果通过编译时类型a调用一个父类的一个方法时，例如：a.shut() 如果子类重写/实现了这个方法时，
运行时，a.shut()执行的结果就是被子类重写/被实现类实现的shut()方法，此时我们说动物a对shut方法
呈现多态了。
如果
a = new Cat() ;
a.shut()

总结：当通过同一引用类型，由于引用的实例不同，对同一行为产生的结果不同。
```

1.3 多态中的类型转换(了解)

1.3.1、类型转换

自动类型转换：把子类对象赋给父类变量（多态）

```
Animal a = new Dog();
Object  obj = new Dog();    //Object是所有类的根类
```

强制类型转换：把父类类型对象赋给子类类型变量。

```
子类类型 变量 = (子类类型)父类对象;
```

（前提：该对象的真实类型应该是子类类型），**在实际开发过程中，如果需要调用子类特有的方法时**，一定要进行强制类型转换。

```
Animal a = new Dog();
Dog d = (Dog) a;    //正确
Cat c = (Cat) a;    //错误
```

1.3.2、instanceOf

instanceof 运算符：判断该对象是否是某一个类/接口的实例，在开发中运用不是很多

```
语法格式：
boolean b = 对象A   instanceof  类B;  //判断 A对象是否是 B类的实例？如果是,返回true
```

代码如下：

```
Animal a = new Dog();
System.out.println(a instanceof Animal);    //true
System.out.println(a instanceof Dog);       //true
System.out.println(a instanceof Cat);       //false
```

1.4 多态的好处 - USB案例（掌握）

需求：模拟在主板上安装鼠标、键盘等，比较没有规范和有规范的区别。

没有统一规范：

鼠标类：

```
public class Mouse {
    //鼠标工作的方法
    public void work1() {
        System.out.println("鼠标在移动");
    }
}
```

键盘类：

```
public class Keyboard {
    //键盘工作的方法
    public void work2() {
        System.out.println("鼠标在移动");
    }
}
```

主板类：

```
public class MotherBoard {
    //在主板上安装鼠标对象
    public void plugin(Mouse m) {
        m.work1();  //调用鼠标工作的方法
    }
    //在主板上安装键盘对象
    public void plugin(Keyboard k) {
        k.work2();  //调用键盘工作的方法
    }
}
```

上述代码是没有统一规范的，我们能发现其中的问题：

- 不同设备中工作的方法名称是不一致的
- 每次需要安装新设备，都需要在主板类上新增一个方法　（这个问题严重）

有统一规范：

USB规范接口：

```
//定义一种规范，用来约束所有的USB设备应该具有的功能
public interface IUSB {
    //USB设备工作的方法
    void swapData();
}
```

在Mouse和Keyboard类遵循于USB规范——工作的方法名称也就相同了。

```
public class Mouse implements IUSB{

    public void swapData() {
        System.out.println("鼠标在移动");
    }
}

public class Keyboard implements IUSB{

    public void swapData() {
        System.out.println("用键盘打字");
    }
}
```

主板类，在安装方法plugin上也体现出了多态的特征：

```
public class MotherBoard {
   //IUSB类型可以接受实现类对象
    public void plugin(IUSB usb) {
        usb.swapData();
    }
}
```

面向接口编程，体现的就是多态，其好处：把实现类对象赋给接口类型变量，屏蔽了不同实现类之间的实现差异，从而可以做到通用编程。

测试类，无论是否使用多态，测试代码相同：

```
public class USBDemo {
    public static void main(String[] args) {
        // 创建主板对象
        MotherBoard board = new MotherBoard();
        // 创建鼠标对象
        Mouse mouse = new Mouse();
        // 创建键盘对象
        Keyboard keyboard = new Keyboard();

        //在主板上安装鼠标
        board.plugin(mouse);
        //在主板上安装键盘
        board.plugin(keyboard);
    }
}
```

请问：使用USB接口后，哪里出现多态了？