面向对象编程

一、深刻认识面向对象

1.面向对象编程的优点：符合人类的思维习惯，使得编程更简单、更直观

2.对象的本质：一种特殊的数据结构

3.类（`class`）：

- 对象的设计图或者模板
- 通过类这张设计图，我们可以创建出无数个不同的对象，它们既有共性，也有差异
- 对象是用类`new`出来的，有了类我们才可以创建出对象

4.对象在计算机中的执行原理：

- 假设我们创建了一个`Student`类
- 然后我们再创建一个`Test`类，在里面执行`Student stu=new Student();`
- 这时候，计算机会在**堆内存**中开辟一块内存区域来存储这一个学生类对象`stu`
- **注意**：`stu`这个变量存储的是它的内存地址！因此`stu`变量也被称为**引用类型**的变量

5.类和对象的一些注意事项：

- 类名的首字母要大写，满足驼峰命名法
- 类中定义的变量称为**成员变量**（对象的属性），成员变量本身存在默认值，我们**不需要为其初始化赋值**
- 类中定义的方法称为**成员方法**（对象的行为）
- 一个代码文件中，可以写多个类，但只能有一个类用`public`**权限修饰符**修饰，且`public`修饰的类名必须成为代码文件的名称

6.面向对象编程的三大特征：

α.**封装**

- 用类设计对象处理某一个事物的数据时，应该把要处理的数据以及处理这些数据的方法，涉及到一个对象中去
- **设计规范**：合理隐藏、合理暴露，善于使用`public`、`private`等权限修饰符
- **代码示例**：

```
public class Student {
    //成员变量私有化，不让外部访问，可以理解为将汽车的内部零件隐藏起来
    private String name;
    private double chineseScore;
    private double mathScore;
    private double englishScore;

    //成员方法公开化，可以理解为将汽车的操作方法（比如方向盘、油门、刹车）公开出来
    public void printTotalScore(){
```

```
        System.out.println(name+"同学的总分是："+
(chineseScore+mathScore+englishScore));
    }

    public void printAverageScore(){
        System.out.println(name+"同学的各科平均分是："+
(chineseScore+mathScore+englishScore)/3.0);
    }
}
```

*b.***继承**

- `Java`中提供了一个关键字`extends`，使用该关键字可以让一个类和另一个类建立起父子关系

  `public class Son extends Father{}`

  `Son`类称为子类或派生类
- **特点**：子类继承父类的**非私有成员**（成员变量和成员方法）
- 子类对象实际上是由父类和子类这两张设计图共同创建出来的
- **使用继承的优点**：减少了重复代码的编写，提高了代码的复用性
- **注意事项**：
  - `Java`是**单继承**的，`Java`中的类不支持**多继承**（即一个儿子不能有多个父亲），但支持**多层继承**（即儿子有父亲，父亲也可以有他的父亲）
  - `Object`类是`Java`所有类的**祖宗类**，我们编写的每一个类，其实都继承了`Object`类或其子孙类
  - **方法重写**：
    - 当子类觉得父类中的某个方法不好用或者无法满足自己的需求时，子类可以重写该方法（要保证方法名称和参数列表一致），去覆盖父类的该方法
    - 建议加上`@Override`注解它可以让编译器帮助我们检查重写格式是否正确，同时增强代码可读性
    - 方法重写后，`Java`会遵循**就近原则**优先调用子类的方法
    - 子类重写父类方法时，访问权限必须>=父类该方法的权限（`public`>`protected`>缺省）
    - 重写方法的返回值类型必须保证一致，或者范围更小
    - **私有方法**和**类方法**不能被重写
    - 子类重写`Object`类的`toString()`方法，以便返回对象的内容而非对象的内存地址
  - **子类中访问其他成员的特点**：
    - 依照**就近原则**
    - 如果子类和父类中出现了重名的成员，由于子类会优先访问子类的成员，那该如何在子类中访问父类的成员呢？

      可以通过使用`super`关键字：`super.父类成员变量/父类成员方法`
  - **子类构造器的特点**：
    - 子类的全部构造器，都会优先调用父类的构造器，再调用自身的构造器
    - 默认情况下，子类全部构造器内部的第一行代码都是`super()`，即调用父类的无参构造器

- 若父类没有无参构造器，则我们必须子类构造器内部的第一行写上`super(...)`，即调用父类的某一有参构造器
- **补充知识**：`this(...)`调用兄弟构造器

```
public class Student{
    private String schoolName;
    private String name;

    //有参构造器A
    public Student(String name){
        //这里使用this关键字调用了有参构造器B，有了this就不能写super了！
        this(name,"Harbin Institute of Technology");
    }

    //有参构造器B
    public Student(String name,String schoolName){
        this.name=name;
        this.schoolName=schoolName;
    }
}
```

c.多态

- 多态是在继承(`extends`)/实现(`implements`)下的一种现象，表现为：对象多态、行为多态
- **代码示例**：

  （1）创建3个类：`People、Student、Teacher`，后面两个类都继承了`People`，并且都重写了`run()`方法

```
public class People {
    public void run(){
        System.out.println("人可以跑");
    }
}
```

```
public class Student extends People{
    @Override
    public void run() {
        System.out.println("学生跑得很快");
    }
}
```

```
public class Teacher extends People{
    @Override
    public void run() {
        System.out.println("老师跑得慢吞吞的");
    }
}
```

  （2）`main`方法中演示多态

```
People p1=new Student();
p1.run();

People p2=new Teacher();//对象多态
p2.run();//行为多态
```

（3）控制台输出结果

```
学生跑得很快
老师跑得慢吞吞的
```

- **多态的前提**：
  - 有继承(`extends`)/实现(`implements`)关系
  - 存在父类引用子类对象
  - 存在方法重写
- **注意事项**：多态是**对象和行为**的多态，`Java`中的成员变量不讨论多态
- 多态下会产生的**问题**：多态创建出的对象无法使用子类独有的功能

  解决方法：**强制类型转换**

  例如：

```
People p=new Teacher();
Teacher t=(Teacher) p;
//强转后就可以使用Teacher的独有功能
```

二、类`class`的五大成分

1.成员变量：

*a.***类变量**（有`static`修饰）

- 属于类的成员变量，与类一起加载一次，在内存中只有一份，可以被类和类的所有对象共享
- 调用格式：
  - 推荐：`类名.类变量`
  - 不推荐：`对象.类变量`

*b.***实例变量**（无`static`修饰）（也称为**对象变量**）

- 属于对象的成员变量，每个对象中都有一份，数据各不相同，只能由创建出来的对象访问
- 调用格式：`对象.实例变量`

2.构造器：

*a.***无参构造器**

*b.***有参构造器**

3.成员方法：

*a.***类方法**（有`static`修饰）

- 最常见的应用场景：做**工具类**
  - **工具类**：工具类中的方法都是类方法，每个类方法都用于完成一个特定功能，工具类是给全体开发人员使用的
  - 工具类没有创建对象的需求，实际工作中建议将工具类的**构造器私有化**
  - 为什么工具类中的方法都是类方法而不用实例方法？
    - 实例方法需要创建出对象以后才能调用，而此时对象的唯一作用就是调用方法，并无其他作用，反而会浪费内存
    - 类方法直接用类名就可以调用，无需创建对象，既方便又节省内存
  - **代码演示**：

```
public class XxxxUtil{//Util后缀表示某某工具类
    //建议将构造器私有化
    private XxxxUtil(){

    }

    public static void xxx(){
        ...
    }

    public static boolean xxxx(String message){
        ...
    }

    public static String xxxxx(int a){
        ...
    }

    ...
}
```

- 调用格式：
  - 推荐：`类名.类方法`
  - 不推荐：`对象.类方法`

*b.***实例方法**（无`static`修饰）

- 调用格式：`对象.实例方法`

*c.***注意事项**：

- 类方法中：可以直接访问类成员（包括类变量和其他类方法），但不可以直接访问实例成员（包括实例变量和实例方法）
- 实例方法中：既可以直接访问类成员，也可以直接访问实例成员
- 实例方法中可以出现 `this` 关键字，而类方法中不可以

4.代码块：

*a.***静态代码块**

- **格式**：

```
static{

}
```

- **特点**：类加载时自动执行，由于类只会加载一次，故静态代码块也只会加载一次
- **作用**：完成类的初始化操作，比如对类变量进行**初始化赋值**

*b.***实例代码块**

- **格式**：

```
{

}
```

- **特点**：每次创建一个新的对象时，都会执行一次实例代码块，注意是**先于构造器执行**
- **作用**：和构造器一样，都是用来完成对象的初始化操作，比如对实例变量进行**初始化赋值**

5.内部类：

- 如果一个类定义在另一个类的内部，那么这个类就叫作**内部类**
- **使用场景**：但一个类的内部包含了一个完整的事物，且这个事物没有必要单独设计时，我们就可以把这个事物设计成内部类
- **代码演示**：

```
public class Car{
    //内部类
    public class Engine{

    }
}
```

*a.***成员内部类**

- 类中的一个普通成员，类似于成员变量、成员方法
- `JDK 16`之后，成员内部类中支持定义静态成员
- ```
  public class Outer{
      //成员内部类
      public class Inner{

      }
  }
  ```

  `main`方法中创建对象的格式：

  ```
  Outer.Inner in=new Outer().new Inner();//必须使用2个new！！！
  ```

- 成员内部类中访问其他成员的特点：
  - 成员内部类中的实例方法中，可以直接访问外部类的实例成员和静态成员
  - 成员内部类中的实例方法中，可以通过`外部类名.this`拿到当前的外部类对象

*b.***静态内部类**

- 有`static`修饰的内部类，属于外部类自己持有
- ```
  public class Outer{
      //成员内部类
      public class Inner{

      }
  }
  ```

  `main`方法中创建对象的格式：

  ```
  Outer.Inner in=new Outer.Inner();//只需使用1个new！！！
  ```

- 静态内部类中访问其他成员的特点：

  可以直接访问外部类的静态成员，不能直接访问外部类的实例成员

*c.***局部内部类**

- 定义在**方法/代码块/构造器**中的内部类

- ```
  public class Outer{
      //构造器
      public Outer(){
          //局部内部类A
          class A{

          }
      }

      //代码块
      static{
          //局部内部类B
          class B{

          }
      }

      //方法
      public void calculate(){
          //局部内部类C
          class C{

          }
      }
  }
  ```

*d.***匿名内部类（重点）**

- 一种特殊的局部内部类
- **匿名**：程序员不需要为这个类起名
- **代码演示**：

  假设我们预先定义了一个`Animal`类

  在`main`方法中，我们来创建一个`Animal`类的匿名内部类对象

  ```
  new Animal(){
      //类体（一般是方法重写）
      @Override
      public void cry(){
          System.out.println("小狗汪汪叫");
      }
  }
  ```

- **特点**：匿名内部类本质上就是一个子类，并会立即创建出一个子类对象
- **作用**：更方便地创建出一个子类对象
- **常见使用场景**：作为一个参数传递给方法

  ```

三、抽象类`abstract class`

1.简介：

- 在`Java`中有一个关键字叫：`abstract`，可以用来修饰类和成员方法
- `abstract`修饰类，这个类就是抽象类；修饰成员方法，这个方法就是抽象方法

2.注意事项：

- 抽象类中不一定有抽象方法，但有抽象方法的类一定是抽象类
- 类该有的五大成员，抽象类都可以有

3.特点：

- 抽象类不能创建对象，仅仅作为一种特殊的父类，让子类继承并重写抽象类的全部抽象方法（必须重写）

4.应用场景和好处：

- 用抽象类可以把子类中相同的代码，包括方法签名（方法名称、参数列表、返回值类型）都抽象到父类中，这样能更好地支持多态，提高代码的灵活性
- 当我们不知道系统未来具体的业务实现时，我们可以预先定义一个抽象类，将来让子类去继承，以方便系统的扩展

5.模板方法设计模式：

*a.***作用**：解决方法中存在重复代码的问题

```
public class A {
    ...

    public void sing(){
        代码一样

        代码不同

        代码一样
    }
    ...
}
```

```
public class B {
    ...

    public void sing(){
        代码一样

        代码不同

        代码一样
    }
    ...
}
```

*b.***写法**：

（1）定义一个抽象类

（2）在这个抽象类内部定义2个方法

```
+ **模板方法**：把相同的代码放进去
+ **抽象方法**：具体事现交由子类完成
```

- 建议使用`final`关键字修饰模板方法，原因如下：
  - 模板方法是给对象直接使用的，不能被子类重写
  - 一旦子类重写了模板方法，模板方法就失效了

c.**代码演示**：

（1）先定义一个抽象类

```
public abstract class People{
    /**模板方法设计模式*/
    //1.定义一个模板方法出来
    public final void write(){
        System.out.println("\t\t\t\t\t《我的父亲》");
        System.out.println("\t\t我的父亲是一个伟大的人，我很敬爱他。");

        //2.模板方法并不清楚正文部分究竟应该怎么写，但是它知道子类肯定要写正文部分
        System.out.println("\t\t"+writeMainPart());

        System.out.println("\t\t能拥有这样的父亲是我这辈子最大的幸福。");
    }

    //3.定义一个抽象方法来写正文，具体的实现交给子类来完成
    public abstract String writeMainPart();
}
```

（2）再定义2个子类继承抽象类，并重写抽象方法

```
public class Student extends People{
    @Override
    public String writeMainPart(){
        return "我爸爸很牛逼，一天抽十包烟。";
    }
}
```

```
public class Teacher extends People{
    @Override
    public String writeMainPart(){
        return "是我的父亲指引我走上了成为一名人民教师的道路。";
    }
}
```

（3）创建子类对象，调用模板方法完成需求

```
public class Test {
    public static void main(String[] args) {
        //目标：搞清楚模板方法设计模式是什么
        /*场景：
            学生和老师都需要写一篇作文：《我的父亲》
            第一段是一样的
            正文部分自由发挥
            最后一段也是一样的
         */
```

```
        Teacher teacher=new Teacher();
        teacher.write();
        System.out.println("====================================");
        Student student=new Student();
        student.write();
    }
}
```

（4）控制台输出内容

```
                    《我的父亲》
        我的父亲是一个伟大的人，我很敬爱他。
        是我的父亲指引我走上了成为一名人民教师的道路。
        能拥有这样的父亲是我这辈子最大的幸福。
====================================
                    《我的父亲》
        我的父亲是一个伟大的人，我很敬爱他。
        我爸爸很牛逼，一天抽十包烟。
        能拥有这样的父亲是我这辈子最大的幸福。
```

四、接口`interface`

1.概述：

- 接口可以理解为一种特殊的类，接口内部都是由全局常量和抽象方法所组成
- 接口用于弥补`Java`无法实现多继承这一缺陷
- 实际开发中，接口更多的作用是**制定标准**，然后由不同类去具体实现这些标准

```
public interface 接口名{
    //成员变量（常量）
    //成员方法（抽象方法）
}
```

2.注意事项：

- 接口不能创建对象
- 接口是用来被类实现（`implements`）的，实现接口的类叫作**实现类**
- 一个类可以实现多个接口，注意必须重写完全部接口的全部抽象方法，否则这个类必须定义成抽象类，而不能是实现类
- 一个类实现多个接口，如果多个接口中存在方法签名冲突，则此时不支持多实现
- 一个类继承了父类，又实现了接口，若父类和接口有同名的默认方法，则实现类会优先调用父类方法
- 一个类实现多个接口，这多个接口又存在同名的默认方法，只要这个类重写该方法就可以不产生冲突

```
public/protected/缺省/private class 实现类名 implements 接口1,接口2,接口3,...{
    ...
}
```

3.接口的多继承：

- 一个接口可以同时继承多个接口，便于实现类去实现
- 一个接口继承多个接口，若这多个接口中存在方法签名冲突，则此时不支持多继承

```
public interface C extends A,B{
    ...
}
```

4.接口新增方法(After `JDK 1.8`)：

- 默认方法：使用`default`修饰，由实现类的对象调用
- 静态方法：使用`static`修饰，必须用当前接口名调用
- 这2种新增方法都默认被`public`修饰
- 私有方法（After `JDK 1.9`）：使用`private`修饰，只能在接口内部被调用