Java面向对象(高级)

Java面向对象(中级)

类变量(静态变量)

类变量也叫静态变量或者静态属性，是该类的所有对象共享的变量，任何一个该类的对象去访问该变量的时候得到的都是相同的值，同样任何一个该类的对象去修改它的时候修改的也是同一个变量
定义格式：

```
访问修饰符 static 数据类型 变量名;
static 访问修饰符 数据类型 变量名;
```

访问方式类变量可以使用类名或者对象实例名访问

```
类名.类变量名/实例对象名.类变量名
```

使用情况当我们需要让某个类的所有对象都共享一个属性的时候，就可以考虑使用类变量
访问修饰类变量的访问修饰符的访问权限和范围和普通属性是一样的
类变量是同一个类中所有变量共享，在类加载的时候就生成了,即使没有创建对象实例，类变量也会创建存在，类变量的·生命周期跟随类的加载开始随着类的消亡而销毁

类方法(静态方法)

类方法是该类所有对象共享的方法
定义格式：

```
访问修饰符 static 数据类型 变量名;
static 访问修饰符 数据类型 变量名;
```

访问方式类方法可以使用类名或者对象实例名访问

```
类名.类方法名/实例对象名.类方法名
```

类方法中不允许使用和对象有关的关键字例如super和this、
静态方法只能访问静态的变量或者静态的方法
普通成员方法可以访问静态的变量和方法

main方法

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

1. main方法是java虚拟机调用
2. java虚拟机需要调用main()方法，所以该方法的访问权限必须是公共的public
3. java虚拟机在运行main()方法的时候不需要创建对象，所以main方法必须是静态方法static
4. 该方法接受String类型的数组参数，该数组中保存执行java命令的时候传递给运行的类的参数
5. 在main方法中可以直接使用main方法中的静态属性或方法，在使用非静态属性和方法的时候需要首先创建实例对象再使用

代码块

代码化块又叫做初始化块，属于类中的成员(是类中的一部分)，类似于方法，将逻辑语句封装在方法体中，通过{}包围起来；
但是与方法不同，没有方法名，没有返回，没有参数，并且不需要通过对象或者类显示调用，而是在加载类或者创建对象的时候隐式调用
基本格式：

```
[修饰符]{
代码;
};
```

代码块细节

1. 修饰符可选，要写的话只能写static
2. 代码块分为两类，静态代码块(使用static修饰)和普通代码块
3. 其中包含的代码可以是任意逻辑语句
4. 代码块大括号最后可以写分号，也可以不写
5. 代码块相当于另外一种格式的构造器，在每个构造器中重复使用的语句可以直接写在代码块中
6. 代码块的调用顺序优先于构造器
7. static代码块也叫静态代码块，作用是对类进行初始化，并且它随着类的加载而进行，并且只会执行一次，如果是普通代码块则每创建(new)一个新的对象就会执行一次
8. **类什么时候会被加载**：
   创建对象实例的时候
   创建子类对象实例的时候，父类也会加载，而且父类先加载子类后加载
   使用子类的静态属性或者方法的时候
9. 创建一个对象在一个类中的调用顺序是：
   调用静态代码块和静态属性初始化的时候，静态代码块和静态属性初始化的优先级一样，如果两者同时存在则按照定义的顺序调用
   调用普通代码块与普通属性初始化的过程与静态类似

如果四者同时存在则静态优先于普通，因为首先需要进行类的初始化才能创建新的对象实例

10. 构造器前面其实隐藏了super()和调用普通代码块，静态的相关代码块，属性初始化在类加载的时候就执行完毕、
11. 调用顺序

- 父类的静态代码和静态属性
- 子类的 静态代码块和静态属性
- 父类的普通代码块和普通属性初始化
- 父类的构造方法
- 子类的普通代码和普通属性初始化
- 子类的构造方法

单例模式

单例模式就是采取一定的方法保证在整个软件系统中，对某个类中只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。

设计步骤：

1. 构造器私有化-> 防止直接new一个对象
2. 类的内部创建对象
3. 向外暴露一个静态的公共方法，返回内部创建的对象
4. 饿汉式：使用对象的某个属性的时候该对象自动创建，可能造成创建了对象但是对象并没有使用，存在浪费空间的问题
   懒汉式：当用户使用getInstance方法的时候才会返回对象,再次调用的时候返回的仍然是上次创建的对象，从而保证单例。具有多线程风险存在

final关键字

1. **final关键字的使用场景**

- 当不希望类被继承时，可以使用`final`修饰。
- 当不希望父类的某个方法被子类重写时，可以使用`final`修饰。
- 当不希望类的某个属性被修改时，可以使用`final`修饰。

2.**** final修饰的属性（常量）

- 被`final`修饰的属性称为**常量**，必须赋初值且以后不能再修改。
- 赋值位置有以下几种：
  - 定义属性时直接赋值。
  - 在构造器中赋值。

- 在代码块中赋值。

示例如下：

```
public class Example {
    final int num1 = 10;           // 定义时直接赋值
    final int num2;

    {
        num2 = 20;                 // 在代码块中赋值
    }

    final int num3;

    public Example() {
        num3 = 30;                 // 在构造器中赋值
    }
}
```

- 若`final`修饰的是静态变量，只能在定义时或静态代码块中赋值：

```
public class Example {
    static final int num1 = 100;  // 定义时 赋值
    static final int num2;

    static {
        num2 = 200;                // 在静态代码块中赋值
    }
}
```

3. final修饰类的特点（final类）

- 使用`final`修饰的类称为**final类**，不能被继承，但可以实例化对象。
- 若某个类是`final`类，则其方法无需再用`final`修饰（因为类已经无法被继承）。
- Java中所有的包装类（如`Integer`、`Double`、`Float`、`Boolean`、`String`）都是`final`类，不可以被继承。
- final修饰的方法子类不能重写，但是仍可以继承该方法

4. final的其他注意事项

- 构造器（构造方法）不能用`final`修饰。
- 如果变量同时被`static`和`final`修饰，使用该属性不会导致类加载（因为该属性属于编译期常量）。

```
public class Test {
    public static final int CONST = 10;
```

```
    public static void main(String[] args) {
        System.out.println(Test.CONST);  // 不会触发Test类的加载
    }
}
```

抽象类abstract

1. 抽象类的概念

- 当父类的某些方法需要声明，但实现方式又不确定时，可将这些方法声明为**抽象方法**。
- **抽象方法**就是没有方法体的方法，仅提供方法签名。
- 当一个类中存在抽象方法时，这个类必须声明为**抽象类**。
- 抽象类中可以没有抽象方法，但含有抽象方法的类一定是抽象类。
- 抽象类不能实例化，只能通过继承由子类具体实现。

2. 抽象方法的定义和特点

- **抽象方法没有方法体**，不能包含任何具体操作。
- 抽象方法的定义格式：

```
public abstract void methodName();
```

- `abstract`关键字只能修饰**类和方法**，不能修饰属性、构造方法等其他成员。
- 抽象类仍然是一个类，可以拥有类的全部成员（构造方法、普通方法、属性等）。
- 抽象类更多地体现的是设计上的价值，规定了一种统一的格式和结构。

3. 抽象方法不能使用以下修饰符：

- `private`：抽象方法需要子类继承实现，`private`限制子类访问。
- `final`：抽象方法需要被重写，而`final`禁止重写。
- `static`：抽象方法属于实例方法，不能声明为静态。

4. 抽象类的继承规则

- 如果一个类继承了抽象类，它必须实现抽象类中的所有抽象方法。
- 如果子类不完全实现父类的抽象方法，那么子类也必须声明为抽象类。

接口interface

1. 基本介绍
    - 接口就是给出一些没有实现的方法，封装到一起，当某个类需要使用的时候，根据具体情况将这些方法重写出来

- 在jdk8以后，接口中可以存在静态方法，默认方法，也就是说接口中可以有方法的具体实现，这里需要使用default或者static关键字修饰
- 定义格式

```
interface 接口名{
//属性
//方法
}
class 类名 implements 接口{
//类独有属性
//类独有方法
//必须实现的接口方法
}
```

2. 使用细节
   - 接口不能被实例化
   - 接口中所有的方法都默认为public方法且不可以修改，因此不需要使用public修饰，对于接口中的抽象方法可以不使用abstract来修饰
   - 一个普通类连接接口则必须实现接口中所有的方法(alt+enter快捷键)，并且方法的访问权限必须是public，因为不可以将一个更弱的访问修饰符代替更强的访问修饰符。
   - 如果连接接口的类不实现接口中的方法则需要标明abstract，传给它的子类完成实现
   - 一个类可以同时实现多个接口
   - 接口中的属性只能是final属性，并且是public static final修饰符,所以接口中的属性必须初始化，并且初始化后无法被改变
   - 接口属性访问格式：接口名.属性名
   - 接口不能继承其他的类，但是可以继承多个别的接口
   - 接口的修饰符只能是public和默认
3. 接口的多态
   如果子接口继承了父接口，一个类实现了子接口，则该类实现了父接口
   接口实例化方式：

- 传统方式，显示实现类：通过定义一个具体的类来实现接口，然后实例化实现该类
- 匿名内部类，直接使用new 接口(){匿名类中的方法(必须包含对接口中未实现方法的重写)}来实现创建匿名类的对象

接口VS继承

如果使用继承，子类会自动获得父类所拥有的属性和方法
如果子类需要拓展功能，可以通过接口来实现，实现接口可以是对java单继承的解决、
继承的价值在于：解决代码的复用性和可维护性
接口的价值在于：设计各种方法，让其他类取实现这些方法，更加灵活
接口相对于继承更加灵活。继承满足"is a"的关系，接口满足"like a"的关系

内部类

一个类中又完整地嵌套了另一个类的结构，被嵌套的类被称为内部类，被嵌套的其他类被称为外部类

1. 内部类有四种：
   - 定义在外部类局部变量上(例如方法内)
     - 局部内部类(有类名)
     - 匿名内部类(没有类名)
   - 定义在外部类的成员变量上
     - 成员内部类(没有使用static修饰)
     - 静态内部类(使用static修饰)
2. 局部内部类
   局部内部类定义在外部类的局部位置，比如在方法中，有类名
   - 内部类可以直接访问外部类的所有成员包含私有成员
   - 内部类不可以添加访问修饰符，因为它的地位相当于一个局部变量，局部变量不需要访问修饰符，但是可以用final修饰，因为final可以修饰局部变量
   - 局部类的作用范围仅仅在定义它的方法或者代码块中
   - 外部类的方法中可以创建内部对象
   - 其他外部类不能访问本外部类的内部类，因为内部类实际上是一个局部变量
   - 如果外部类和内部类成员重名的时候，默认遵循就近原则，如果访问外部类的成员则可使用(外部类.this.成员);外部类.this实际上就是外部类的对象，谁调用了外部类的这个方法就是谁
3. 匿名内部类
   匿名内部类是定义在外部类的局部位置并且没有类名的类
   匿名内部类其实系统中给它分配名字了，只是没有显示出来，其实格式是：所在类名$i 系统在底层进行分配。匿名内部类使用一次就不能使用了

```
接口名/类名 对象名 = new 接口名/类名() {
    @Override
    方法体;
};
也可以直接:
new 接口名/类名() {
    @Override
    方法体;
}.方法;
```

   - 可以直接访问外部类的所有成员，包括私有的
   - 不能添加访问修饰符，因为它的地位相当于一个局部变量
   - 作用域仅仅在定义它的方法和代码块中
   - 如果外部类和内部类成员重名的时候，默认遵循就近原则，如果访问外部类的成员则可使用(外部类.this.成员);

4. 成员内部类
   在类中方法与代码块之外定义，并且不使用static修饰
   - 可以使用外部类的成员包含私有的
   - 访问修饰符进行修饰，因为它本身就是一个成员
   - 作用域和外部类的成员一样，作用域为整个类体
   - 外部类使用成员内部类，先创建后访问
   - 外部其他类访问成员内部类的三种方式：

```
- 将内部类看做外部类的成员通过点进行访问
  外部类.外部类中的内部类 内部类对象实例 =
  外部类对象实例.new 内部类();
- 在外部类中定义一个返回内部类的方法 get内部类的方法:
  外部类.外部类中的内部类 内部类对象实例=
  外部类对象实例.get内部类方法();
- 连续进行两次new
  new 外部类().new 内部类();
```

- 如果外部类和内部类成员重名的时候，默认遵循就近原则，如果访问外部类的成员则可使用(外部类.this.成员);

5. 静态内部类
   放在外部类的成员变量，使用static修饰
   - 可以访问外部所有的静态成员，但是不能直接访问非静态成员
   - 可以添加访问修饰符，因为它的地位是一个成员
   - 作用域是整个类体
   - 外部类访问内部类，创建对象再访问
   - 外部其他类访问静态内部类
   `- 外部类名.内部类名 内部类实例= new 外部类名.内部类名(); - 编写一个静态方法返回一个静态内部类的对象实例 外部类名.内部类名 内部类实例= 外部类.get内部类方法();`
   根据java的多态属性，接口如果使用匿名内部类实例化，该对象仍然可以被当做一个接口进行传参，但是接口本身是不能实例化的

注解Annotation

使用注解Annotation的时候需要再它前面加@符号，应该将其当做一个修饰符来使用，用于修饰它支持的程序元素
@interface是注解类并不是接口
常用基本注释：

- @Override 限定于某个方法，是重写父类方法，该注释只能用于方法
  - @Override注释在子类重写父类对应的方法上面(表示提示重写)，如果写了该注释则编译器会检查该方法是否真正实现了方法的重写

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

其中@Target是修饰注解的注解，称为元注解

- @Deprecated 用于表示某个程序元素(类，方法等)已过时，不推荐使用，但是仍然可以使用，使用的时候该元素会被划线表示不推荐使用，方便做出一个过渡

```
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE,
PARAMETER, TYPE})
```

可以修饰的元素上述所示

- @SuppressWarning 抑制编译器警告

```
@SuppressWarnings({"all"})
```

写入相关的字符串实现不显示对应操作显示出的警告
作用元素：

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
```

作用范围：写在哪个元素上面就作用于哪个方法上

- 元注释用于修饰其他的注释
  - Retention指定注释的作用范围，通常为SOURCE，CLASS，RUNTIME
  - Target指定注释可以在哪些地方使用
  - Documented指定该注释是否会在javadoc中体现
  - Inherited子类会继承父类注释