面向过程&面向对象

语言的进化发展跟生物的进化发展其实是一回事，都是"物以类聚"。相近的感光细胞聚到一起变成了我们的眼睛，相近的嗅觉细胞聚到一起变成了我们的鼻子。

语句多了，我们将完成同样功能的相近的语句，聚到了一块儿，便于我们使用。于是，方法出现了！

变量多了，我们将功能相近的变量组在一起，聚到一起归类，便于我们调用。于是，结构体出现了！

再后来，方法多了，变量多了！结构体不够用了！我们就将功能相近的变量和方法聚到了一起，于是类和对象出现了！

寥寥数语，就深刻的展示了语言的进化历史！其实，都非常自然，"物以类聚"。希望大家能记住这句话。

企业的发展也是"物以类聚"的过程，完成市场推广的人员聚到一起形成了市场部。完成技术开发的人员聚到一起形成了开发部！

面向过程的思维模式

面向过程的思维模式是简单的线性思维，思考问题首先陷入第一步做什么、第二步做什么的细节中。这种思维模式适合处理简单的事情，比如：上厕所。

如果面对复杂的事情，这种思维模式会陷入令人发疯的状态！比如：如何造神舟十号！

面向对象的思维模式

面向对象的思维模式说白了就是分类思维模式。思考问题首先会解决问题需要哪些分类，然后对这些分类进行单独思考。最后，才对某个分类下的细节进行面向过程的思索。

这样就可以形成很好的协作分工。比如：设计师分了10个类，然后将10个类交给了10个人分别进行详细设计和编码！

显然，面向对象适合处理复杂的问题，适合处理需要多人协作的问题！

如果一个问题需要多人协作一起解决，那么你一定要用面向对象的方式来思考！

对于描述复杂的事物，为了从宏观上把握、从整体上合理分析，我们需要使用面向对象的思路来分析整个系统。但是，具体到微观操作，仍然需要面向过程的思路去处理。

OOP详解

1、什么是面向对象

Java的编程语言是面向对象的，采用这种语言进行编程称为面向对象编程(Object-Oriented Programming, OOP)。

面向对象编程的本质就是：以类的方式组织代码，以对象的组织(封装)数据。

抽象(abstract)

```
1 忽略一个主题中与当前目标无关的那些方面，以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题，而只是选择其中的一部分，暂时不用关注细节。
2 例如:要设计一个学生成绩管理系统，那么对于学生，只关心他的班级、学号、成绩等，而不用去关心他的身高、体重这些信息。 抽象是什么?就是将多个物体共同点归纳出来，就是抽出像的部分！
```

封装(Encapsulation)

封装是面向对象的特征之一，是对象和类概念的主要特性。封装是把过程和数据包围起来，对数据的访问只能通过指定的方式。

在定义一个对象的特性的时候，有必要决定这些特性的可见性，即哪些特性对外部是可见的，哪些特性用于表示内部状态。

通常，应禁止直接访问一个对象中数据的实际表示，而应通过操作接口来访问，这称为信息隐藏。

信息隐藏是用户对封装性的认识，封装则为信息隐藏提供支持。

封装保证了模块具有较好的独立性，使得程序维护修改较为容易。对应用程序的修改仅限于类的内部，因而可以将应用程序修改带来的影响减少到最低限度。

继承(inheritance)

```
1  继承是一种联结类的层次模型，并且允许和支持类的重用，它提供了一种明确表述共性的方法。
2    新类继承了原始类后,新类就继承了原始类的特性，新类称为原始类的派生类(子类)，而原始类称为新类的基类(父类)。
3        派生类(子类)可以从它的基类(父类)那里继承方法和实例变量，并且派生类(子类)中可以修改或增加新的方法使之更适合特殊的需要继承性很好的解决了软件的可重用性问题。比如说，所有的Windows应用程序都有一个窗口，它们可以看作都是从一个窗口类派生出来的。但是有的应用程序用于文字处理，有的应用程序用于绘图，这是由于派生出了不同的子类，各个子类添加了不同的特性。
```

多态(polymorphism)

```
1  多态性是指允许不同类的对象对同一消息作出响应。
2      多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好的解决了应用程序函数同名问题。
```

相同类域的不同对象,调用相同方法,表现出不同的结果

从认识论角度考虑是先有对象后有类。对象，是具体的事物。类，是抽象的，是对对象的抽象。

从代码运行角度考虑是先有类后有对象。类是对象的模板。

2、类与对象的关系

类是一种抽象的数据类型,它是对某一类事物整体描述/定义,但是并不能代表某一个具体的事物.

```
1  例如：我们生活中所说的词语：动物、植物、手机、电脑等等。这些也都是抽象的概念,而不是指的某一个具体的东西。
```

例如: Person类、Pet类、Car类等，这些类都是用来描述/定义某一类具体的事物应该具备的特点和行为

对象是抽象概念的具体实例

```
1  例如：张三就是人的一个具体实例,张三家里的旺财就是狗的一个具体实例。能够体现出特点,展现出功能的是具体的实例,而不是一个抽象的概念.
```

【示例】

```
1  Student s = new Student(1L,"tom",20);
2  s.study();
3
4  Car c = new Car(1,"BWM",500000);
5  c.run();
```

对象s就是Student类的一个实例,对象c就是Car类的一个具体实例,能够使用的是具体实例,而不是类。类只是给对象的创建提供了一个参考的模板而已.
但是在java中,没有类就没有对象,然而类又是根据具体的功能需求,进行实际的分析,最终抽象出来的.

3、对象和引用的关系

引用 "指向" 对象
使用类类型、数组类型、接口类型声明出的变量,都可以指向对象,这种变量就是引用类型变量,简称引用。
在程序中,创建出对象后,直接使用并不方便,所以一般会用一个引用类型的变量去接收这个对象,这个就是所说的引用指向对象.

总结:对象和引用的关系,就如电视机和遥控器,风筝和线的关系一样。

方法回顾及加深

方法一定是定义在类中的,属于类的成员。

1、方法的定义

```
格式：    修饰符  返回类型  方法名(参数列表)异常抛出类型{...}
```

1. 修饰符

```
public、static、abstract、final等等都是修饰符，一个方法可以有多个修饰符。例如程序入口main方法，就使用了public static这个俩个修饰符
注:如果一个方法或者属性有多个修饰符,这多个修饰符是没有先后顺序的
```

2. 返回类型

```
方法执行完如果有要返回的数据,那么就要声明返回数据的类型,如果没有返回的数据,那么返回类型就必须写void.
只有构造方法(构造器)不写任何返回类型也不写void
```

【示例】

```
public String sayHello(){
    return "hello";
}
public int max(int a,int b){
    return a>b?a:b;
}
public void print(String msg){
    System.out.println(msg);
}
```

思考：声明返回类型的方法中一定要出现return语句,那么没有返回类型(void)的方法中,能不能出现return语句?
注：break和return的区别

狂神社群笔记资料，禁止外传，本人QQ：24736743

```
return 语句的作用
(1) return 从当前的方法中退出,返回到该调用的方法的语句处,继续执行。
(2) return 返回一个值给调用该方法的语句，返回值的数据类型必须与方法的声明中的返回值的类型一致。
(3) return后面也可以不带参数，不带参数就是返回空，其实主要目的就是用于想中断函数执行，返回调用函数处。

break语句的作用
(1)break在循环体内，强行结束循环的执行，也就是结束整个循环过程，不在判断执行循环的条件是否成立，直接转向循环语句下面的语句。
(2)当break出现在循环体中的switch语句体内时，其作用只是跳出该switch语句体。
```

3. 方法名

```
遵守java中标示符的命名规则即可.
```

4. 参数列表

```
根据需求定义,方法可以是无参的,也可以有一个参数,也可以有多个参数
```

5. 异常抛出类型

```
如果方法中的代码在执行过程中,可能会出现一些异常情况,那么就可以在方法上把这些异常声明并抛出,也可以同时声明抛出多个异常,使用逗号隔开即可。
```

【示例】

```
public void readFile(String file)throws IOException{

}

public void readFile(String file)throws IOException,ClassNotFoundException{

}
```

2、方法调用

在类中定义了方法,这个方法中的代码并不会执行,当这个方法被调用的时候,方法中的代码才会被一行一行顺序执行。

1. 非静态方法

```
没有使用static修饰符修饰的方法,就是非静态方法.
调用这种方法的时候,是"一定"要使用对象的。因为非静态方法是属于对象的。(非静态属性也是一样的)
```

【例子】

```
public class Student{
    public void say(){}
}

main:
Student s = new Student();
s.say();
```

2. 静态方法

```
使用static修饰符修饰的方法,就是静态方法.
调用这种方法的时候,"可以"使用对象调用,也"可以"使用类来调用,但是推荐使用类进行调用,因为静态方法是属于类的。(静态属性也是一样的)
```

【例子】

```
public class Student{
    public static void say(){}
}

main:
Student.say();
```

3. 类中方法之间的调用

假设同一个类中有俩个方法,a方法和b方法，a和b都是非静态方法，相互之间可以直接调用。

```
public void a(){
    b();
}
public void b(){

}
```

a和b都是静态方法,相互之间可以直接调用.

```
public static void a(){
    b();
}
public static void b(){

}
```

a静态方法,b是非静态方法，a方法中不能直接调用b方法,但是b方法中可以直接调用a方法. 静态方法不能调用非静态方法!

```
public static void a(){
    //b();报错
}
public void b(){
    a();
}
```

另外：在同一个类中,静态方法内不能直接访问到类中的非静态属性.

总结：类中方法中的调用，两个方法都是静态或者非静态都可以互相调用，当一个方法是静态，一个方法是非静态的时候，非静态方法可以调用静态方法，反之不能。

3、调用方法时的传参

1. 形参和实参

【例子】

狂神社群笔记资料，禁止外传，本人QQ：24736743

```
// a = x;
public void test(int a){
    //..
}
main:
int x = 1;
t.test(x);
```

参数列表中的a是方法test的形参(形式上的参数)
调用方法时的x是方法test的实参(实际上的参数)

注意：形参的名字和实参的名字都只是一个变量的名字,是可以随便写的,我们并不关心这个名字,而是关心变量的类型以及变量接收的值。

2. 值传递和引用传递
调用方法进行传参时,分为值传递和引用传递两种。

如果参数的类型是基本数据类型,那么就是值传递。

如果参数的类型是引用数据类型,那么就是引用传递。

值传递是实参把自己变量本身存的简单数值赋值给形参.
引用传递是实参把自己变量本身存的对象内存地址值赋值给形参.

所以值传递和引用传递本质上是一回事,只不过传递的东西的意义不同而已.

【示例：值传递】

```
public class Test{
    public static void changeNum(int a){
        a = 10;
    }

    public static void main(String[] args){
        int a = 1;
        System.out.println("before: a = "+a);  //1
        changeNum(a);
        System.out.println("after: a = "+a);   //1
    }
}
```

【示例：引用传递】

```
public class Demo03 {

    public static void changeName(Student s){
        s.name = "tom";
    }

    public static void main(String[] args){
        Student s = new Student();
        System.out.println("before: name = "+s.name); //null
        changeName(s);
        System.out.println("after: name = "+s.name); //tom
    }
}

class Student{
    String name;
```

```
}

```

4、this关键字

在类中,可以使用this关键字表示一些特殊的作用。

1、this在类中的作用

【区别成员变量和局部变量】

```
public class Student{
    private String name;
    public void setName(String name){
        //this.name表示类中的属性name
        this.name = name;
    }
}
```

【调用类中的其他方法】

```
public class Student{
    private String name;
    public void setName(String name){
        this.name = name;
    }
    public void print(){
        //表示调用当前类中的setName方法
        this.setName("tom");
    }
}
```

注：默认情况下,setName("tom")和this.setName("tom")的效果是一样的.

【调用类中的其他构造器】

```
public class Student{
    private String name;
    public Student(){
        //调用一个参数的构造器,并且参数的类型是String
        this("tom");
    }
    public Student(String name){
        this.name = name;
    }
}
```

注:this的这种用法,只能在构造器中使用.普通的方法是不能用的.并且这局调用的代码只能出现在构造器中的第一句.

【示例】

```
public class Student{
    private String name;
    //编译报错,因为this("tom")不是构造器中的第一句代码.
    public Student(){
        System.out.println("hello");
        this("tom");
    }
    public Student(String name){
        this.name = name;
    }
}
```

2、this关键字在类中的意义

this在类中表示当前类将来创建出的对象.

【例子】

```
public class Student{
    private String name;
    public Student(){
        System.out.println("this = "+this);
    }
    public static void main(String[] args){
        Student s = new Student();
        System.out.println("s = "+s);
    }
}
```

运行后看结果可知,this和s打印的结果是一样的,那么其实也就是变量s是从对象的外部执行对象,而this是在对象的内部执行对象本身.
这样也就能理解为什么this.name代表的是成员变量,this.setName("tom")代表的是调用成员方法,因为这俩句代码从本质上讲,和在对象外部使用变量s来调用是一样的,s.name和s.setName("tom")。

【this和s打印出来的内存地址是一样的,使用==比较的结果为true。】

```
public class Student{
    public Student getStudent(){
        return this;
    }

    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = s1.getStudent();
        System.out.println(s1 == s2);//true
    }
}
```

【 类中的this是和s1相等还是和s2相等呢?】

```java
public class Student{
    private String name;
    public void test(){
        System.out.println(this);
    }
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student();
        s1.test();
        s2.test();
    }
}
```

注:这句话是要这么来描述的,s1对象中的this和s1相等,s2对象中的this和s2相等,因为类是模板,模板中写的this并不是只有一个,每个对象中都有一个属于自己的this,就是每个对象中都一个属于自己的name属性一样.

创建与初始化对象

使用new关键字创建对象

使用new关键字创建的时候，除了分配内存空间之外，还会给 创建好的对象 进行默认的初始化 以 及对类中构造器的调用。

```
那么对main方法中的以下代码:
    Student s = new Student();

1)为对象分配内存空间,将对象的实例变量自动初始化默认值为0/false/null。(实例变量的隐式赋值)

2)如果代码中实例变量有显式赋值,那么就将之前的默认值覆盖掉。(之后可以通过例子看到这个现象)
    例如:显式赋值
    private String name = "tom";

3)调用构造器

4)把对象内存地址值赋值给变量。(=号赋值操作)
```

构造器

类中的构造器也称为构造方法，是在进行创建对象的时候必须要调用的。并且构造器有以下俩个特点:

1. 必须和类的名字相同
2. 必须没有返回类型,也不能写void

构造器的作用:

1. 使用new创建对象的时候必须使用类的构造器
2. 构造器中的代码执行后,可以给对象中的属性初始化赋值

【演示】

狂神社群笔记资料，禁止外传，本人QQ：24736743

```
public class Student{
    private String name;
    public Student(){
        name = "tom";
    }
}
```

构造器重载

除了无参构造器之外,很多时候我们还会使用有参构造器,在创建对象时候可以给属性赋值.
【例子】

```
public class Student{
    private String name;

    public Student(){
        name = "tom";
    }

    public Student(String name){
        this.name = name;
    }
}
```

构造器之间的调用

使用this关键字,在一个构造器中可以调用另一个构造器的代码。

注意:this的这种用法不会产生新的对象,只是调用了构造器中的代码而已.一般情况下只有使用new关键字才会创建新对象。

【演示】

```
public class Student{
    private String name;
    public Student(){
        this();
    }
    public Student(String name){
        this.name = name;
    }
}
```

默认构造器

在java中,即使我们在编写类的时候没有写构造器,那么在编译之后也会自动的添加一个无参构造器,这个无参构造器也被称为默认的构造器。

【示例】

```
public class Student{

}

main:
//编译通过,因为有无参构造器
Student s = new Student();
```

但是,如果我们手动的编写了一个构造器,那么编译后就不会添加任何构造器了

【示例】

```java
public class Student{
    private String name;
    public Student(String name){
        this.name = name;
    }
}

main:
//编译报错,因为没有无参构造器
Student s = new Student();
```

内存分析

JAVA程序运行的内存分析

栈 stack：

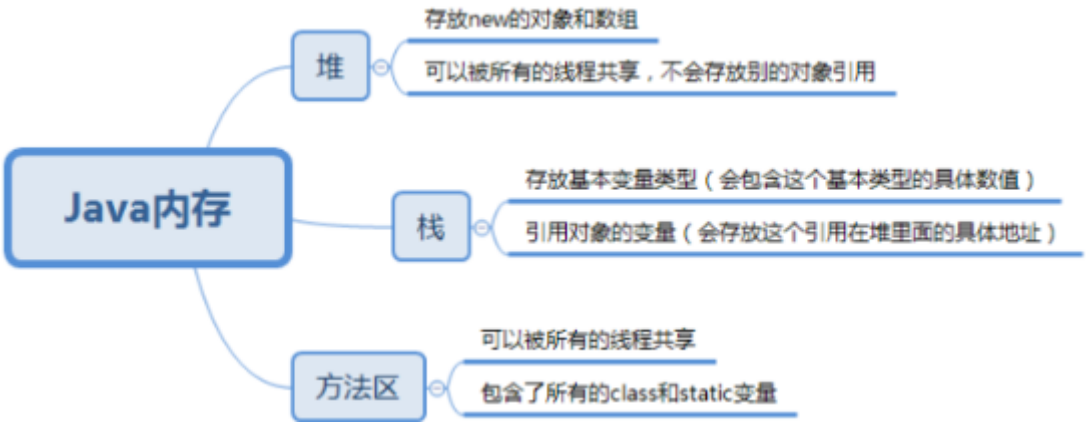
1. 每个线程私有，不能实现线程间的共享！
2. 局部变量放置于栈中。
3. 栈是由系统自动分配，速度快！栈是一个连续的内存空间！

堆 heap：

1. 放置new出来的对象！
2. 堆是一个不连续的内存空间，分配灵活，速度慢！

方法区(也是堆)：

1. 被所有线程共享！
2. 用来存放程序中永远是不变或唯一的内容。（类代码信息、静态变量、字符串常量）



注意：本次内存分析，我们的主要目的是让大家了解基本的内存概念。类加载器、Class对象这些更加详细的内容，我们将在后面专门讲反射的课程里面讲。

引用类型的概念

1. java中，除了基本数据类型之外的其他类型称之为引用类型。
2. java中的对象是通过引用来操作的。(引用：reference)说白了，引用指的就是对象的地址！

属性（field，或者叫成员变量）

1. 属性用于定义该类或该类对象包含的数据或者说静态属性。
2. 属性作用范围是整个类体。
3. 属性的默认初始化：

狂神社群笔记资料，禁止外传，本人QQ：24736743

在定义成员变量时可以对其初始化，如果不对其初始化，Java使用默认的值对其初始化。(数值：0,0.0 char:u0000, boolean:false, 所有引用类型:null)

4.属性定义格式：

```
[修饰符]  属性类型  属性名 = [默认值]
```

类的方法

方法是类和对象动态行为特征的抽象。方法很类似于面向过程中的函数。面向过程中，函数是最基本单位，整个程序有一个个函数调用组成；面向对象中，整个程序的基本单位是类，方法是从属于类或对象的。

方法定义格式：

```
[修饰符]  方法返回值类型  方法名(形参列表) {
    // n条语句
}
```

java对象的创建和使用

- 必须使用 new 关键字创建对象。

Person person= new Person ();

- 使用对象（引用）. 成员变量来引用对象的成员变量。

person.age

- 使用对象（引用）. 方法（参数列表）来调用对象的方法。

    1. setAge(23)

类中就是：//静态的数据 //动态的行为

学习完类与对象终于认识到什么是类，什么是对象了。接下来要看的就是java的三大特征：继承、封装、多态。

封装

我要看电视，只需要按一下开关和换台就可以了。有必要了解电视机内部的结构吗？有必要碰碰显像管吗？

制造厂家为了方便我们使用电视，把复杂的内部细节全部封装起来，只给我们暴露简单的接口，比如：电源开关。需要让用户知道的暴露出来，不需要让用户了解的全部隐藏起来。这就是封装。

白话：该露的露，该藏的藏

专业：我们程序设计要追求“高内聚，低耦合”。高内聚就是类的内部数据操作细节自己完成，不允许外部干涉；低耦合：仅暴露少量的方法给外部使用。

封装（数据的隐藏）

在定义一个对象的特性的时候，有必要决定这些特性的可见性，即哪些特性对外部是可见的，哪些特性用于表示内部状态。

通常，应禁止直接访问一个对象中数据的实际表示，而应通过操作接口来访问，这称为信息隐藏。

1、封装的步骤

1. 使用private 修饰需要封装的成员变量。

狂神社群笔记资料，禁止外传，本人QQ：24736743

2. 提供一个公开的方法设置或者访问私有的属性
   设置 通过set方法，命名格式： set属性名（）；属性的首字母要大写

访问 通过get方法，命名格式： get属性名（）；属性的首字母要大写

【演示】

```
//对象能在类的外部"直接"访问
public class Student{
    public String name;
    public void println(){
        System.out.println(this.name);
    }
}
public class Test{
    public static void main(String[] args){
        Student s = new Student();
        s.name = "tom";
    }
}
```

在类中一般不会把数据直接暴露在外部的,而使用private(私有)关键字把数据隐藏起来

【演示】

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

public class Test{
    public static void main(String[] args){
        Student s = new Student();
        //编译报错,在类的外部不能直接访问类中的私有成员
        s.name = "tom";
    }
}
```

如果在类的外部需要访问这些私有属性,那么可以在类中提供对于的get和set方法,以便让用户在类的外部可以间接的访问到私有属性

【示例】

```
//set负责给属性赋值
//get负责返回属性的值
public class Student{
    private String name;
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }
}

public class Test{
    public static void main(String[] args){
        Student s = new Student();
        s.setName("tom");
        System.out.println(s.getName());
```

```
    }
}
```

2、作用和意义

1. 提高程序的安全性，保护数据。
2. 隐藏代码的实现细节
3. 统一用户的调用接口
4. 提高系统的可维护性
5. 便于调用者调用。

良好的封装，便于修改内部代码，提高可维护性。

良好的封装，可进行数据完整性检测，保证数据的有效性。

3、方法重载

类中有多个方法,有着相同的方法名,但是方法的参数各不相同,这种情况被称为方法的重载。方法的重载可以提供方法调用的灵活性。

思考：HelloWorld中的System.out.println()方法，为什么可以把不同类型的参数传给这个方法?

【演示：查看println方法的重载】

例如:

```
public class Test{
    public void test(String str){

    }
    public void test(int a){
    }
}
```

方法重载必须满足以下条件

1. 方法名必须相同
2. 参数列表必须不同(参数的类型、个数、顺序的不同)

```
public void test(Strig str){}
public void test(int a){}

public void test(Strig str,double d){}
public void test(Strig str){}

public void test(Strig str,double d){}
public void test(double d,Strig str){}
```

3. 方法的返回值可以不同，也可以相同。

在java中,判断一个类中的俩个方法是否相同,主要参考俩个方面:方法名字和参数列表

继承

现实世界中的继承无处不在。比如：

动物：哺乳动物、爬行动物

哺乳动物：灵长目、鲸目等。

狂神社群笔记资料，禁止外传，本人QQ：24736743

继承的本质是对某一批类的抽象，从而实现对现实世界更好的建模。

为什么需要继承？继承的作用？

第一好处：继承的本质在于抽象。类是对对象的抽象，继承是对某一批类的抽象。

第二好处：为了提高代码的复用性。

extands的意思是“扩展”。子类是父类的扩展。

【注】JAVA中类只有单继承，没有多继承！ 接口可以多继承！

1、继承

1. 继承是类和类之间的一种关系。除此之外,类和类之间的关系还有依赖、组合、聚合等。
2. 继承关系的俩个类，一个为子类(派生类),一个为父类(基类)。子类继承父类,使用关键字extends来表示。

```
public class student extends Person{
}
```

3. 子类和父类之间,从意义上讲应该具有"is a"的关系.

```
student is a person
dog is a animal
```

4. 类和类之间的继承是单继承

```
一个子类只能"直接"继承一个父类,就像是一个人只能有一个亲生父亲
一个父类可以被多子类继承,就像一个父亲可以有多个孩子

注:java中接口和接口之间,有可以继承,并且是多继承。
```

5. 父类中的属性和方法可以被子类继承

子类中继承了父类中的属性和方法后,在子类中能不能直接使用这些属性和方法,是和这些属性和方法原有的修饰符(public protected default private)相关的。
例如：
父类中的属性和方法使用public修饰,在子类中继承后"可以直接"使用
父类中的属性和方法使用private修饰,在子类中继承后"不可以直接"使用
注：具体细则在修饰符部分详细说明

父类中的构造器是不能被子类继承的,但是子类的构造器中,会隐式的调用父类中的无参构造器(默认使用super关键字)。
注:具体细节在super关键字部分详细说明

2、Object类

java中的每一个类都是"直接" 或者 "间接"的继承了Object类.所以每一个对象都和Object类有"is a"的关系。从API文档中,可以看到任何一个类最上层的父类都是Object。(Object类本身除外)AnyClass is a Object。

```
System.out.println(任何对象 instanceof Object);
//输出结果:true
//注:任何对象也包含数组对象

例如:
//编译后,Person类会默认继承Object
public class Person{}

//Student是间接的继承了Object
public class Student extends Person{}
```

在Object类中,提供了一些方法被子类继承,那么就意味着,在java中,任何一个对象都可以调用这些被继承过来的方法。(因为Object是所以类的父类)
例如:toString方法、equals方法、getClass方法等
注:Object类中的每一个方法之后都会使用到.

3、Super关键字

子类继承父类之后,在子类中可以使用this来表示访问或调用子类中的属性或方法,使用super就表示访问或调用父类中的属性和方法。

1. **super的使用**

【访问父类中的属性】

```
public class Person{
    protected String name = "zs";
}
public class Student extends Person{
    private String name = "lisi";
    public void tes(String name)t{
        System.out.println(name);
        System.out.println(this.name);
        System.out.println(super.name);
    }
}
```

【调用父类中的方法】

```
public class Person{
    public void print(){
        System.out.println("Person");
    }
}
public class Student extends Person{
    public void print(){
        System.out.println("Student");
    }
    public void test(){
        print();
        this.print();
        super.print();
    }
}
```

【调用父类中的构造器】

```
public class Person{

}
public class Student extends Person{
    //编译通过,子类构造器中会隐式的调用父类的无参构造器
    //super();
    public Student(){
    }
}
```

父类没有无参构造

```
public class Person{
    protected String name;
    public Person(String name){
        this.name = name;
    }
}
public class Student extends Person{
    //编译报错,子类构造器中会隐式的调用父类的无参构造器,但是父类中没有无参构造器
    //super();
    public Student(){

    }
}
```

【显式的调用父类的有参构造器】

```
public class Person{
    protected String name;
    public Person(String name){
        this.name = name;
    }
}
public class Student extends Person{
    //编译通过,子类构造器中显式的调用父类的有参构造器
    public Student(){
        super("tom");
    }
}
```

注：不管是显式还是隐式的父类的构造器,super语句一定要出现在子类构造器中第一行代码。所以this和super不可能同时使用它们调用构造器的功能,因为它们都要出现在第一行代码位置。

【例子】

```
public class Person{
    protected String name;
    public Person(String name){
        this.name = name;
    }
}
public class Student extends Person{
    //编译报错,super调用构造器的语句不是第一行代码
    public Student(){
        System.out.println("Student");
        super("tom");
    }
}
```

【例子】

```
public class Person{
    protected String name;
    public Person(String name){
        this.name = name;
    }
}
//编译通过
public class Student extends Person{
    private int age;
    public Student(){
        this(20);
    }
    public Student(int age){
        super("tom");
        this.age = age;
    }
}
```

【super使用的注意的地方】

1. 用super调用父类构造方法，必须是构造方法中的第一个语句。
2. super只能出现在子类的方法或者构造方法中。
3. super 和 this 不能够同时调用构造方法。（因为this也是在构造方法的第一个语句）

【super 和 this 的区别】

1. 代表的事物不一样:
   this：代表所属方法的调用者对象。

   ```
       super：代表父类对象的引用空间。
   ```

2. 使用前提不一致:
   this：在非继承的条件下也可以使用。

   ```
       super：只能在继承的条件下才能使用。
   ```

3. 调用构造方法:
   this：调用本类的构造方法。

   ```
       super：调用的父类的构造方法
   ```

4、方法重写

方法的重写（override）

1. 方法重写只存在于子类和父类(包括直接父类和间接父类)之间。在同一个类中方法只能被重载，不能被重写.
2. 静态方法不能重写
    1. 父类的静态方法不能被子类重写为非静态方法 //编译出错
    2. 父类的非静态方法不能被子类重写为静态方法； //编译出错
    3. 子类可以定义与父类的静态方法同名的静态方法(但是这个不是覆盖)

【例子】

```
A类继承B类 A和B中都一个相同的静态方法test

B a = new A();
a.test();//调用到的是B类中的静态方法test

A a = new A();
a.test();//调用到的是A类中的静态方法test

可以看出静态方法的调用只和变量声明的类型相关
这个和非静态方法的重写之后的效果完全不同
```

1. 私有方法不能被子类重写，子类继承父类后,是不能直接访问父类中的私有方法的,那么就更谈不上重写了。
   【例子】

```
public class Person{
    private void run(){}
}
//编译通过,但这不是重写,只是俩个类中分别有自己的私有方法
public class Student extends Person{
    private void run(){}
}
```

1. 重写的语法
    1. 方法名必须相同
    2. 参数列表必须相同
    3. 访问控制修饰符可以被扩大,但是不能被缩小： public protected default private
    4. 抛出异常类型的范围可以被缩小,但是不能被扩大
       ClassNotFoundException ---> Exception
    5. 返回类型可以相同,也可以不同,如果不同的话,子类重写后的方法返回类型必须是父类方法返回类型的子类型
       例如：父类方法的返回类型是Person,子类重写后的返回类可以是Person也可以是Person的子类型

注：一般情况下,重写的方法会和父类中的方法的声明完全保持一致,只有方法的实现不同。(也就是大括号中代码不一样)

```
public class Person{
    public void run(){}

    protected Object test()throws Exception{
        return null;
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

```
    }
}
//编译通过,子类继承父类,重写了run和test方法.
public class Student extends Person{
    public void run(){}

    public String test(){
        return "";
    }
}
```

为什么要重写?

子类继承父类,继承了父类中的方法,但是父类中的方法并不一定能满足子类中的功能需要,所以子类中需要把方法进行重写。

1. 总结:

方法重写的时候，必须存在继承关系。

方法重写的时候，方法名和形式参数 必须跟父类是一致的。

方法重写的时候，子类的权限修饰符必须要大于或者等于父类的权限修饰符。(private < protected < public，friendly < public)

方法重写的时候，子类的返回值类型必须小于或者等于父类的返回值类型。(子类 < 父类) 数据类型没有明确的上下级关系

方法重写的时候，子类的异常类型要小于或者等于父类的异常类型。

多态

1、认识多态

多态性是OOP中的一个重要特性，主要是用来实现动态联编的，换句话说，就是程序的最终状态只有在执行过程中才被决定而非在编译期间就决定了。这对于大型系统来说能提高系统的灵活性和扩展性。

多态可以让我们不用关心某个对象到底是什么具体类型，就可以使用该对象的某些方法，从而实现更加灵活的编程，提高系统的可扩展性。

允许不同类的对象对同一消息做出响应。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。
相同类域的不同对象,调用相同的方法,执行结果是不同的

1. 一个对象的实际类型是确定的

```
    例如: new Student(); new Person();等
```

2. 可以指向对象的引用的类型有很多

```
    一个对象的实现类型虽然是确定的,但是这个对象所属的类型可能有很多种。
```

例如: Student继承了Person类

```
Student s1 = new Student();
Person s2  = new Student();
Object s3  = new Student();
```

因为Person和Object都是Student的父类型

狂神社群笔记资料，禁止外传，本人QQ：24736743

注:一个对象的实际类型是确定,但是可以指向这个对象的引用的类型,却是可以是这对象实际类型的任意父类型。

1. 一个父类引用可以指向它的任何一个子类对象
   例如:

```
Object o = new AnyClass();
Person p = null;
p = new Student();
p = new Teacher();
p = new Person();
```

2. 多态中的方法调用

```
public class Person{
    public void run(){}
}

public class Student extends Person{

}
```

3. 调用到的run方法,是Student从Person继承过来的run方法

```
main:
Person p = new Student();
p.run();
```

例如:

```
public class Person{
    public void run(){}
}
public class Student extends Person{
    public void run(){
        //重写run方法
    }
}

//调用到的run方法,是Student中重写的run方法
main:
Person p = new Student();
p.run();
```

注：子类继承父类,调用a方法，如果a方法在子类中没有重写,那么就是调用的是子类继承父类的a方法,如果重写了,那么调用的就是重写之后的方法。

子类中独有方法的调用

```
public class Person{
    public void run(){}
}
public class Student extends Person{
    public void test(){
    }
}
main:
Person p = new Student();
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

```
//调用到继承的run方法
p.run();

//编译报错,因为编译器检查变量p的类型是Person,但是在Person类中并没有发现test方法,所以编译报错.
p.test();
```

注:一个变量x,调用一个方法test,编译器是否能让其编译通过,主要是看声明变量x的类型中有没有定义test方法,如果有则编译通过,如果没有则编译报错.而不是看x所指向的对象中有没有test方法.

原理：编译看左边，运行不一定看右边。

```
编译看左边的意思：java 编译器在编译的时候会检测引用类型中含有指定的成员，如果没有就会报错。子类的成员是特有的，父类的没有的，所以他是找不到的。
```

子类引用和父类引用指向对象的区别

```
Student s = new Student();
Person p = new Student();
```

变量s能调用的方法是Student中有的方法(包括继承过来的),变量p能调用的方法是Person中有的方法(包括继承过来的)。

但是变量p是父类型的,p不仅可以指向Student对象,还可以指向Teacher类型对象等,但是变量s只能指Studnet类型对象,及Student子类型对象。变量p能指向对象的范围是比变量s大的。
Object类型的变量o,能指向所有对象,它的范围最大,但是使用变量o能调用到的方法也是最少的,只能调用到Object中的声明的方法,因为变量o声明的类型就是Object.
注：java中的方法调用,是运行时动态和对象绑定的,不到运行的时候,是不知道到底哪个方法被调用的。

2、重写、重载和多态的关系

重载是编译时多态

```
调用重载的方法,在编译期间就要确定调用的方法是谁,如果不能确定则编译报错
```

重写是运行时多态

```
调用重写的方法,在运行期间才能确定这个方法到底是哪个对象中的。这个取决于调用方法的引用,在运行期间所指向的对象是谁,这个引用指向哪个对象那么调用的就是哪个对象中的方法。(java中的方法调用,是运行时动态和对象绑定的)
```

3、多态的注意事项

1. 多态是方法的多态，属性没有多态性。
2. 编写程序时，如果想调用运行时类型的方法，只能进行类型转换。不然通不过编译器的检查。但是如果两个没有关联的类进行强制转换，会报：ClassCastException。 比如：本来是狗，我把它转成猫。就会报这个异常。
3. 多态的存在要有3个必要条件：要有继承，要有方法重写，父类引用指向子类对象

4、多态存在的条件

1. 有继承关系
2. 子类重写父类方法
3. 父类引用指向子类对象

狂神社群笔记资料，禁止外传，本人QQ：24736743

补充一下第二点，既然多态存在必须要有“子类重写父类方法”这一条件，那么以下三种类型的方法是没有办法表现出多态特性的（因为不能被重写）：

1. static方法，因为被static修饰的方法是属于类的，而不是属于实例的
2. final方法，因为被final修饰的方法无法被子类重写
3. private方法和protected方法，前者是因为被private修饰的方法对子类不可见，后者是因为尽管被protected修饰的方法可以被子类见到，也可以被子类重写，但是它是无法被外部所引用的，一个不能被外部引用的方法，怎么能谈多态呢

5、方法绑定(method binding)

执行调用方法时，系统根据相关信息，能够执行内存地址中代表该方法的代码。分为静态绑定和动态绑定。

静态绑定：

在编译期完成，可以提高代码执行速度。

动态绑定：

通过对象调用的方法，采用动态绑定机制。这虽然让我们编程灵活，但是降低了代码的执行速度。这也是JAVA比C/C++速度慢的主要因素之一。JAVA中除了final类、final方、static方法，所有方法都是JVM在运行期才进行动态绑定的。

多态：如果编译时类型和运行时类型不一致，就会造成多态。

6、instanceof和类型转换

1. instanceof

```
public class Person{
    public void run(){}
}
public class Student extends Person{
}
public class Teacher extends Person{
}
```

```
main:
Object o = new Student();
System.out.println(o instanceof Student);//true
System.out.println(o instanceof Person);//true
System.out.println(o instanceof Object);//true
System.out.println(o instanceof Teacher);//false
System.out.println(o instanceof String);//false
\--------------------------

Person o = new Student();
System.out.println(o instanceof Student);//true
System.out.println(o instanceof Person);//true
System.out.println(o instanceof Object);//true
System.out.println(o instanceof Teacher);//false
//编译报错
System.out.println(o instanceof String);
\--------------------------
    Student o = new Student();
System.out.println(o instanceof Student);//true
System.out.println(o instanceof Person);//true
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

```
System.out.println(o instanceof Object);//true
//编译报错
System.out.println(o instanceof Teacher);
//编译报错
System.out.println(o instanceof String);
```

【分析1】

```
System.out.println(x instanceof Y);
该代码能否编译通过,主要是看声明变量x的类型和Y是否存在子父类的关系.有"子父类关"系就编译通过,
没有子父类关系就是编译报错.
之后学习到的接口类型和这个是有点区别的。
```

【分析2】

```
System.out.println(x instanceof Y);
输出结果是true还是false,主要是看变量x所指向的对象实际类型是不是Y类型的"子类型".
```

```
main:
Object o = new Person();
System.out.println(o instanceof Student);//false
System.out.println(o instanceof Person);//true
System.out.println(o instanceof Object);//true
System.out.println(o instanceof Teacher);//false
System.out.println(o instanceof String);//false
```

2. 类型转换

```
public class Person{
    public void run(){}
}
public class Student extends Person{
    public void go(){}
}
public class Teacher extends Person{
}
```

【为什么要类型转换】

```
//编译报错,因为p声明的类型Person中没有go方法
Person p = new Student();
p.go();

//需要把变量p的类型进行转换
Person  p = new Student();
Student s = (Student)p;
s.go();
或者
//注意这种形式前面必须要俩个小括号
((Student)p).go();
```

【类型转换中的问题】

```
//编译通过 运行没问题
Object o = new Student();
Person p = (Person)o;

//编译通过 运行没问题
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

```
Object o = new Student();
Student s = (Student)o;

//编译通过,运行报错
Object o = new Teacher();
Student s = (Student)o;

即:
X x = (X)o;
运行是否报错,主要是变量o所指向的对象实现类型,是不是X类型的子类型,如果不是则运行就会报错。
```

【总结】

1、父类引用可以指向子类对象，子类引用不能指向父类对象。

2、把子类对象直接赋给父类引用叫upcasting向上转型，向上转型不用强制转型。

如Father father = new Son();

3、把指向子类对象的父类引用赋给子类引用叫向下转型（downcasting），要强制转型。

如father就是一个指向子类对象的父类引用，把father赋给子类引用son 即Son son =（Son）father;

其中father前面的（Son）必须添加，进行强制转换。

4、upcasting 会丢失子类特有的方法,但是子类overriding 父类的方法，子类方法有效

5、向上转型的作用，减少重复代码，父类为参数，调有时用子类作为参数，就是利用了向上转型。这样使代码变得简洁。体现了JAVA的抽象编程思想。

修饰符

1、static修饰符

1、static变量

在类中,使用static修饰的成员变量,就是静态变量,反之为非静态变量。

静态变量和非静态变量的区别
静态变量属于类的,"可以"使用类名来访问,非静态变量是属于对象的,"必须"使用对象来访问.

```
public class Student{
    private static int age;
    private double score;

    public static void main(String[] args) {
        Student s = new Student();
        //推荐使用类名访问静态成员
        System.out.println(Student.age);
        System.out.println(s.age);

        System.out.println(s.score);
    }
}
```

静态变量对于类而言在内存中只有一个,能被类的所有实例所共享。实例变量对于类的每个实例都有一份,它们之间互不影响.

```java
public class Student{
    private static int count;
    private int num;
    public Student() {

        count++;
        num++;
    }
    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student();
        Student s3 = new Student();
        Student s4 = new Student();
        //因为还是在类中,所以可以直接访问私有属性
        System.out.println(s1.num);
        System.out.println(s2.num);
        System.out.println(s3.num);
        System.out.println(s4.num);

        System.out.println(Student.count);
        System.out.println(s1.count);
        System.out.println(s2.count);
        System.out.println(s3.count);
        System.out.println(s4.count);
    }
}
```

在加载类的过程中为静态变量分配内存,实例变量在创建对象时分配内存，所以静态变量可以使用类名来直接访问,而不需要使用对象来访问.

2、static方法

在类中,使用static修饰的成员方法,就是静态方法,反之为非静态方法。

静态方法和非静态方法的区别

```
静态方法数属于类的,"可以"使用类名来调用,非静态方法是属于对象的,"必须"使用对象来调用.
```

静态方法"不可以"直接访问类中的非静态变量和非静态方法,但是"可以"直接访问类中的静态变量和静态方法
注意:this和super在类中属于非静态的变量.(静态方法中不能使用)

```java
public class Student{
    private static int count;
    private int num;
    public void run(){}
    public static void go(){}

    public static void test(){
        //编译通过
        System.out.println(count);
        go();

        //编译报错
        System.out.println(num);
        run();
    }

```

```
}
```

非静态方法"可以"直接访问类中的非静态变量和非静态方法,也"可以"直接访问类中的静态变量和静态方法

```
public class Student{
    private static int count;
    private int num;
    public void run(){}
    public static void go(){}

    public void test(){
        //编译通过
        System.out.println(count);
        go();

        //编译通过
        System.out.println(num);
        run();
    }

}
```

思考:为什么静态方法和非静态方法不能直接相互访问? 加载顺序的问题!

父类的静态方法可以被子类继承,但是不能被子类重写

```
public class Person {
    public static void method() {}
}

//编译报错
public class Student extends Person {
    public void method(){}
}


例如:
public class Person {
    public static void test() {
        System.out.println("Person");
    }
}

//编译通过,但不是重写
public class Student extends Person {
    public static void test(){
        System.out.println("Student");
    }
}

main:
Perosn p = new Student();
p.test();//输出Person
p = new Person();
p.test();//输出Perosn

和非静态方法重写后的效果不一样
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

父类的非静态方法不能被子类重写为静态方法；

```
public class Person {
    public void test() {
        System.out.println("Person");
    }
}

//编译报错
public class Student extends Person {
    public static void test(){
        System.out.println("Student");
    }
}
```

3、代码块和静态代码块

【类中可以编写代码块和静态代码块】

```
public class Person {
    {
        //代码块(匿名代码块)
    }

    static{
        //静态代码块
    }
}
```

【匿名代码块和静态代码块的执行】

因为没有名字,在程序并不能主动调用这些代码块。

匿名代码块是在创建对象的时候自动执行的,并且在构造器执行之前。同时匿名代码块在每次创建对象的时候都会自动执行.

静态代码块是在类加载完成之后就自动执行,并且只执行一次.

注:每个类在第一次被使用的时候就会被加载,并且一般只会加载一次.

```
public class Person {
    {
        System.out.println("匿名代码块");
    }

    static{
        System.out.println("静态代码块");
    }

    public Person(){
        System.out.println("构造器");
    }
}
main:
Student s1 = new Student();
Student s2 = new Student();
Student s3 = new Student();

//输出
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

```
静态代码块
匿名代码块
构造器

匿名代码块
构造器

匿名代码块
构造器
```

【匿名代码块和静态代码块的作用】

匿名代码块的作用是给对象的成员变量初始化赋值,但是因为构造器也能完成这项工作,所以匿名代码块使用的并不多。

静态代码块的作用是给类中的静态成员变量初始化赋值。
例如:

```java
public class Person {
    public static String name;
    static{
        name = "tom";
    }
    public Person(){
        name = "zs";
    }
}

main:
System.out.println(Person.name);//tom
```

注：在构造器中给静态变量赋值,并不能保证能赋值成功,因为构造器是在创建对象的时候才指向,但是静态变量可以不创建对象而直接使用类名来访问.

4、创建和初始化对象的过程

```java
Student s = new Student();
```

【Student类之前没有进行类加载】

1. 类加载,同时初始化类中静态的属性
2. 执行静态代码块
3. 分配内存空间,同时初始化非静态的属性(赋默认值,0/false/null)
4. 调用Student的父类构造器
5. 对Student中的属性进行显示赋值(如果有的话)
6. 执行匿名代码块
7. 执行构造器
8. 返回内存地址

注:子类中非静态属性的显示赋值是在父类构造器执行完之后和子类中的匿名代码块执行之前的时候

```java
public class Person{
    private String name = "zs";
    public Person() {
        System.out.println("Person构造器");
        print();
    }
    public void print(){
```

```
        System.out.println("Person  print方法: name = "+name);
    }
}

public class Student extends Person{
    private String name = "tom";
    {
        System.out.println("Student匿名代码块");
    }

    static{
        System.out.println("Student静态代码块");
    }
    public Student(){
        System.out.println("Student构造器");
    }
    public void print(){
        System.out.println("student print方法: name = "+name);
    }
    public static void main(String[] args) {
        new Student();
    }
}

//输出:
Student静态代码块
Person构造器
student print方法: name = null
Student匿名代码块
Student构造器

Student s = new Student();
Student类之前已经进行了类加载
1.分配内存空间,同时初始化非静态的属性(赋默认值,0/false/null)
2.调用Student的父类构造器
3.对Student中的属性进行显示赋值(如果有的话)
4.执行匿名代码块
5.执行构造器
6.返回内存地址
```

5、静态导入

静态导包就是java包的静态导入，用import static代替import静态导入包是JDK1.5中的新特性。

意思是导入这个类里的静态方法。

好处：这种方法的好处就是可以简化一些操作，例如打印操作System.out.println(...);就可以将其写入一个静态方

法print(...)，在使用时直接print(...)就可以了。但是这种方法建议在有很多重复调用的时候使用，如果仅有一到两次调用，不如直接写来的方便。

```java
import static java.lang.Math.random;
import static java.lang.Math.PI;

public class Test {
    public static void main(String[] args) {
        //之前是需要Math.random()调用的
        System.out.println(random());
        System.out.println(PI);
    }
}
```

2、final修饰符

1、修饰类

用final修饰的类不能被继承,没有子类。
例如:我们是无法写一个类去继承String类,然后对String类型扩展的,因为API中已经被String类定义为final的了.
我们也可以定义final修饰的类:

```java
public final class Action{

}

//编译报错
public class Go extends Action{

}
```

2、修饰方法

用final修饰的方法可以被继承,但是不能被子类的重写。
例如:每个类都是Object类的子类,继承了Object中的众多方法,在子类中可以重写toString方法、equals方法等,但是不能重写getClass方法 wait方法等,因为这些方法都是使用fianl修饰的。
我们也可以定义final修饰的方法:

```java
public class Person{
    public final void print(){}
}

//编译报错
public class Student extends Person{
    public void print(){

    }
}
```

3、修饰变量

用final修饰的变量表示常量,只能被赋一次值.其实使用final修饰的变量也就成了常量了,因为值不会再变了。

【修饰局部变量】

```java
public class Person{
    public void print(final int a){
        //编译报错,不能再次赋值,传参的时候已经赋过了
        a = 1;
```

```
    }
}

public class Person{
    public void print(){
        final int a;
        a = 1;
        //编译报错,不能再次赋值
        a = 2;
    }
}
```

【修饰成员变量-非静态成员变量】

```
public class Person{
    private final int a;
}
只有一次机会,可以给此变量a赋值的位置:
声明的同时赋值
匿名代码块中赋值
构造器中赋值(类中出现的所有构造器都要写)
```

【修饰成员变量-静态成员变量】

```
public class Person{
    private static final int a;
}
只有一次机会,可以给此变量a赋值的位置:
声明的同时赋值
静态代码块中赋值
```

【修饰引用变量】

```
main:
final Student s = new Student();
//编译通过
s.setName("tom");
s.setName("zs");

//编译报错,不能修改引用s指向的内存地址
s = new Student();
```

3、abstract修饰符

abstract修饰符可以用来修饰方法也可以修饰类,如果修饰方法,那么该方法就是抽象方法;如果修饰类,那么该类就是抽象类。

1、抽象类和抽象方法的关系

抽象类中可以没有抽象方法,但是有抽象方法的类一定要声明为抽象类。

2、语法

```
public abstract class Action{
    public abstract void doSomething();
}

public void doSomething(){...}
```

对于这个普通方法来讲:
"public void doSomething()"这部分是方法的声明
"{...}"这部分是方法的实现,如果大括号中什么都没写,就叫方法的空实现

声明类的同时,加上abstract修饰符就是抽象类
声明方法的时候,加上abstract修饰符,并且去掉方法的大口号,同时结尾加上分号,该方法就是抽象方法。

3、特点及作用

抽象类,不能使用new关键字来创建对象,它是用来让子类继承的。
抽象方法,只有方法的声明,没有方法的实现,它是用来让子类实现的。

注:子类继承抽象类后,需要实现抽象类中没有实现的抽象方法,否则这个子类也要声明为抽象类。

```
public abstract class Action{
    public abstract void doSomething();
}

main:
//编译报错,抽象类不能new对象
Action a = new Action();

//子类继承抽象类
public class Eat extends Action{
    //实现父类中没有实现的抽象方法
    public void doSomething(){
        //code
    }
}

main:
Action a = new Eat();
a.doSomething();
```

注:子类继承抽象类,那么就必须要实现抽象类没有实现的抽象方法,否则该子类也要声明为抽象类。

4、思考

思考1 : 抽象类不能new对象,那么抽象类中有没有构造器?

```
抽象类是不能被实例化,抽象类的目的就是为实现多态中的共同点,抽象类的构造器会在子类实例化时调用,因此它也是用来实现多态中的共同点构造，不建议这样使用!
```

思考2 : 抽象类和抽象方法意义(为什么要编写抽象类、抽象方法)

```
打个比方，要做一个游戏。如果要创建一个角色，如果反复创建类和方法会很繁琐和麻烦。建一个抽象类后。若要创建角色可直接继承抽象类中的字段和方法，而抽象类中又有抽象方法。如果一个角色有很多种职业，每个职业又有很多技能，要是依次实例这些技能方法会显得想当笨拙。定义抽象方法，在需要时继承后重写调用，可以省去很多代码。
总之抽象类和抽象方法起到一个框架作用。很方便后期的调用和重写
抽象方法是为了程序的可扩展性。重写抽象方法时即可实现同名方法但又非同目的的要求。
```

接口

1、接口的本质

普通类：只有具体实现

抽象类：具体实现和规范(抽象方法) 都有！

接口：只有规范！

【为什么需要接口?接口和抽象类的区别?】

- 接口就是比“抽象类”还“抽象”的“抽象类”，可以更加规范的对子类进行约束。全面地专业地实现了：规范和具体实现的分离。
- 抽象类还提供某些具体实现，接口不提供任何实现，接口中所有方法都是抽象方法。接口是完全面向规范的，规定了一批类具有的公共方法规范。
- 从接口的实现者角度看，接口定义了可以向外部提供的服务。
- 从接口的调用者角度看，接口定义了实现者能提供那些服务。
- 接口是两个模块之间通信的标准，通信的规范。如果能把你要设计的系统之间模块之间的接口定义好，就相当于完成了系统的设计大纲，剩下的就是添砖加瓦的具体实现了。大家在工作以后，做系统时往往就是使用“面向接口”的思想来设计系统。

【接口的本质探讨】

- 接口就是规范，定义的是一组规则，体现了现实世界中“如果你是...则必须能...”的思想。如果你是天使，则必须能飞。如果你是汽车，则必须能跑。如果你好人，则必须干掉坏人；如果你是坏人，则必须欺负好人。
- 接口的本质是契约，就像我们人间的法律一样。制定好后大家都遵守。
- OO的精髓，是对对象的抽象，最能体现这一点的就是接口。为什么我们讨论设计 模式都只针对具备了抽象能力的语言（比如c++、java、c#等），就是因为设计模式所研究的，实际上就是如何合理的去抽象。

2、接口与抽象类的区别

抽象类也是类,除了可以写抽象方法以及不能直接new对象之外,其他的和普通类没有什么不一样的。接口已经另一种类型了,和类是有本质的区别的,所以不能用类的标准去衡量接口。

声明类的关键字是class,声明接口的关键字是interface。

抽象类是用来被继承的,java中的类是单继承。

类A继承了抽象类B,那么类A的对象就属于B类型了,可以使用多态
一个父类的引用,可以指向这个父类的任意子类对象
注:继承的关键字是extends

接口是用来被类实现的,java中的接口可以被多实现。
类A实现接口B、C、D、E..,那么类A的对象就属于B、C、D、E等类型了,可以使用多态
一个接口的引用,可以指向这个接口的任意实现类对象
注:实现的关键字是implements

3、接口中的方法都是抽象方法

接口中可以不写任何方法,但如果写方法了,该方法必须是抽象方法

```
public interface Action{
    public abstract void run();

    //默认就是public abstract修饰的
    void test();
    public void go();
}
```

4、接口中的变量都是静态常量(public static final修饰)

接口中可以不写任何属性,但如果写属性了,该属性必须是public static final修饰的静态常量。
注:可以直接使用接口名访问其属性。因为是public static修饰的

注:声明的同时就必须赋值.(因为接口中不能编写静态代码块)

```
public interface Action{
    public static final String NAME = "tom";
    //默认就是public static final修饰的
    int AGE = 20;
}
main:
System.out.println(Action.NAME);
System.out.println(Action.AGE);
```

5、一个类可以实现多个接口

```
public class Student implements A,B,C,D{
    //Student需要实现接口A B C D中所有的抽象方法
    //否则Student类就要声明为抽象类,因为有抽象方法没实现
}
main:
A s1 = new Student();
B s2 = new Student();
C s3 = new Student();
D s4 = new Student();
```

注:
s1只能调用接口A中声明的方法以及Object中的方法
s2只能调用接口B中声明的方法以及Object中的方法
s3只能调用接口C中声明的方法以及Object中的方法
s4只能调用接口D中声明的方法以及Object中的方法

注:必要时可以类型强制转换

例如 : 接口A 中有test() ， 接口B 中有run()

```
A s1 = new Student();
s1.test();

B s2 = new Student();
s2.run();

if(s1 instanceof B){
    ((B)s1).run();
}
```

6、一个接口可以继承多个父接口

```
public interface A{
    public void testA();
}

public interface B{
    public void testB();
}

//接口C把接口A B中的方法都继承过来了
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

```java
public interface C extends A,B{
    public void testC();
}

//Student相当于实现了A B C三个接口,需要实现所有的抽象方法
//Student的对象也就同时属于A类型 B类型 C类型
public class Student implements C{
    public viod testA(){}
    public viod testB(){}
    public viod testC(){}
}

main:
C o = new Student();
System.out.println(o instanceof A);//true
System.out.println(o instanceof B);//true
System.out.println(o instanceof C);//true
System.out.println(o instanceof Student);//true
System.out.println(o instanceof Object);//true
System.out.println(o instanceof Teacher);//false

//编译报错
System.out.println(o instanceof String);
```

注:System.out.println(o instanceof X);

```
如果o是一个接口类型声明的变量,那么只要X不是一个final修饰的类,该代码就能通过编译,至于其结果是不是true,就要看变量o指向的对象的实际类型,是不是X的子类或者实现类了。
```

注:一个引用所指向的对象,是有可能实现任何一个接口的。(java中的多实现)

7、接口的作用

接口的最主要的作用是达到统一访问，就是在创建对象的时候用接口创建

【接口名】【对象名】= new 【实现接口的类】

这样你像用哪个类的对象就可以new哪个对象了，不需要改原来的代码。

假如我们两个类中都有个function()的方法，如果我用接口，那样我new a()；就是用a的方法，new b（）就是用b的方法

这个就叫统一访问，因为你实现这个接口的类的方法名相同，但是实现内容不同

总结:

1、Java接口中的成员变量默认都是public,static,final类型的(都可省略),必须被显示初始化,即接口中的成员变量为常量(大写,单词之间用"_"分隔)

2、Java接口中的方法默认都是public,abstract类型的(都可省略),没有方法体,不能被实例化

3、Java接口中只能包含public,static,final类型的成员变量和public,abstract类型的成员方法

4、接口中没有构造方法,不能被实例化

5、一个接口不能实现(implements)另一个接口,但它可以继承多个其它的接口

6、Java接口必须通过类来实现它的抽象方法

7、当类实现了某个Java接口时,它必须实现接口中的所有抽象方法,否则这个类必须声明为抽象类

8、不允许创建接口的实例(实例化),但允许定义接口类型的引用变量,该引用变量引用实现了这个接口的类的实例

9、 一个类只能继承一个直接的父类,但可以实现多个接口,间接的实现了多继承.

【实例】

```
interface SwimInterface{
    void swim();
}
class Fish{
    int  fins=4;
}
class Duck {
    int  leg=2;
    void egg(){};
}

class Goldfish extends Fish implements  SwimInterface {
    @Override
    public void swim() {
        System.out.println("Goldfish  can swim ");
    }
}

class SmallDuck  extends Duck implements  SwimInterface {
    public void egg(){
        System.out.println("SmallDuck  can lay  eggs ");
    }
    @Override
    public void swim() {
        System.out.println("SmallDuck  can swim ");
    }
}

public class InterfaceDemo {
    public static void main(String[] args) {
        Goldfish goldfish=new Goldfish();
        goldfish.swim();

        SmallDuck smallDuck= new SmallDuck();
        smallDuck.swim();
        smallDuck.egg();
    }
}
```

内部类

上一小节，我们学习了接口，在以后的工作中接口是我们经常要碰到的，所以一定要多去回顾。接下来介绍一下内部类。很多时候我们创建类的对象的时候并不需要使用很多次，每次只使用一次，这个时候我们就可以使用内部类了。

1、内部类概述

内部类就是在一个类的内部在定义一个类，比如，A类中定义一个B类，那么B类相对A类来说就称为内部类，而A类相对B类来说就是外部类了。

狂神社群笔记资料，禁止外传，本人QQ：24736743

内部类不是在一个java源文件中编写俩个平行的俩个类,而是在一个类的内部再定义另外一个类。 我们可以把外边的类称为外部类,在其内部编写的类称为内部类。

内部类分为四种：

1. 成员内部类
2. 静态内部类
3. 局部内部类
4. 匿名内部类

2、成员内部类（实例内部类、非静态内部类）

注：成员内部类中不能写静态属性和方法

【定义一个内部类】

```
//在A类中申明了一个B类，此B类就在A的内部，并且在成员变量的位置上，所以就称为成员内部类
public class Outer {
    private int id;
    public void out(){
        System.out.println("这是外部类方法");
    }

    class Inner{
        public void in(){
            System.out.println("这是内部类方法");
        }
    }
}
```

【实例化内部类】

实例化内部类，首先需要实例化外部类，通过外部类去调用内部类

```
public class Outer {
    private int id;
    public void out(){
        System.out.println("这是外部类方法");
    }

    class Inner{
        public void in(){
            System.out.println("这是内部类方法");
        }
    }
}


public class Test{
    public static void main(String[] args) {
        //实例化成员内部类分两步
        //1、实例化外部类
        Outer outObject = new Outer();
        //2、通过外部类调用内部类
        Outer.Inner inObject = outObject.new Inner();
        //测试，调用内部类中的方法
```

```
        inObject.in();//打印：这是内部类方法
    }
}
```

分析：想想如果你要使用一个类中方法或者属性，你就必须要先有该类的一个对象，同理，一个类在另一个类的内部，那么想要使用这个内部类，就必须先要有外部类的一个实例对象，然后在通过该对象去使用内部类。

【成员内部类能干什么？】

1. 访问外部类的所有属性(这里的属性包括私有的成员变量，方法)

```
public class Outer {
    private int id;
    public void out(){
        System.out.println("这是外部类方法");
    }

    class Inner{
        public void in(){
            System.out.println("这是内部类方法");
        }

        //内部类访问外部类私有的成员变量
        public void useId(){
            System.out.println(id+3);。
        }

        //内部类访问外部类的方法
        public void useOut(){
            out();
        }
    }
}

public class Test{
    public static void main(String[] args) {
        //实例化成员内部类分两步
        //1、实例化外部类
        Outer outObject = new Outer();
        //2、通过外部类调用内部类
        Outer.Inner inObject = outObject.new Inner();
        //测试
        inObject.useId();//打印3，因为id初始化值为0，0+3就为3，其中在内部类就使用了外部类的私有成员变量id。
        inObject.useOut();//打印：这是外部类方法
    }
}
```

1. 如果内部类中的变量名和外部类的成员变量名一样，要通过创建外部类对象 "."属性来访问外部类属性，通过this.属性访问内部类成员属性

```
public class Outer {
    private int id;//默认初始化0
    public void out(){
        System.out.println("这是外部类方法");
    }

```

```
    class Inner{
        private int id=8;     //这个id跟外部类的属性id名称一样。
        public void in(){
            System.out.println("这是内部类方法");
        }

        public void test(){
            System.out.println(id);//输出8，内部类中的变量会暂时将外部类的成员变量给隐藏
            //如何调用外部类的成员变量呢？通过Outer.this，想要知道为什么能通过这个来调用，就得明白下面这个原理
            //想实例化内部类对象，就必须通过外部类对象，当外部类对象来new出内部类对象时，会
            //把自己(外部类对象)的引用传到了内部类中，所以内部类就可以通过Outer.this来访问外部类的属性和方法，到这里，你也就可以知道为什么内部类可以访问外部类的属性和方法，这里由于有两个相同的
            属性名称，所以需要显示的用Outer.this来调用外部类的属性，平常如果属性名不重复，那么我们在内部类中调用外部类的属性和方法时，前面就隐式的调用了Outer.this。

                System.out.println(Outer.this.id);//输出外部类的属性id。也就是输出0
        }
    }
}
```

借助成员内部类，来总结内部类(包括4种内部类)的通用用法：

1、要想访问内部类中的内容，必须通过外部类对象来实例化内部类。

2、能够访问外部类所有的属性和方法，原理就是在通过外部类对象实例化内部类对象时，外部类对象把自己的引用传进了内部类，使内部类可以用通过Outer.this去调用外部类的属性和方法，

一般都是隐式调用了，但是当内部类中有属性或者方法名和外部类中的属性或方法名相同的时候，就需要通过显式调用Outer.this了。

【写的一个小例子】

```
public class MemberInnerClassTest {
    private String name;
    private static int age;

    public void run(){}

    public static void go(){}

    public class MemberInnerClass{
        private String name;

        //内部类访问外部类
        public void test(String name){
            System.out.println(name);
            System.out.println(this.name);
            System.out.println(MemberInnerClassTest.this.name);
            System.out.println(MemberInnerClassTest.age);
            MemberInnerClassTest.this.run();
            MemberInnerClassTest.go();
        }
    }

```

```
    //外部类访问成员内部类
    //成员内部类的对象要 依赖于外部类的对象的存在
    public void test(){
        //MemberInnerClass mic = MemberInnerClassTest.this.new MemberInnerClass();
        //MemberInnerClass mic = this.new MemberInnerClass();
        MemberInnerClass mic = new MemberInnerClass();
        mic.name = "tom";
        mic.test("hua");

    }

    public static void main(String[] args) {
        //MemberInnerClass mic = new MemberInnerClass();这个是不行的，this是动态的。
        //所以要使用要先创建外部类对象，才能使用
        MemberInnerClassTest out = new MemberInnerClassTest();
        MemberInnerClass mic = out.new MemberInnerClass();
        //如果内部类是private,则不能访问，只能铜鼓内部方法来调用内部类
        mic.name="jik";
        mic.test("kkk");

    }

}
```

3、静态内部类

看到名字就知道，使用你static修饰的内部类就叫静态内部类。

既然提到了static，那我们就来复习一下它的用法：一般只修饰变量和方法，平常不可以修饰类，但是内部类却可以被static修饰。

1）static修饰成员变量：整个类的实例共享静态变量

2）static修饰方法：静态方法，只能够访问用static修饰的属性或方法，而非静态方法可以访问static修饰的方法或属性

3）被static修饰了的成员变量和方法能直接被类名调用。

4）static不能修饰局部变量，切记，不要搞混淆了，static平常就用来修饰成员变量和方法。

写了一个例子，可以给大家看一下：

```
public class StaticInnerClassTest {
    private String name;
    private static int age;

    public void run(){}

    public static void go(){}

    //外部类访问静态内部类
    public void test(){
        StaticInnerClass sic = new StaticInnerClass(); //静态的内部类不需要依赖外部类，所以不用this
        sic.name = "tom";
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

```
        sic.test1("jack");

        StaticInnerClass.age=10;
        StaticInnerClass.test2("xixi");

    }

    private static class StaticInnerClass{
        private String name;
        private static int age;
        public void test1(String name){
            System.out.println(name);
            System.out.println(this.name);
            System.out.println(StaticInnerClass.age);

            System.out.println(StaticInnerClassTest.age);
            //System.out.println(StaticInnerClassTest.this.name);静态类不能访问非静态属性

            StaticInnerClassTest.go();
            //StaticInnerClassTest.this.run();静态类不能访问非静态方法
        }
        public static void test2(String name){
            //只能访问自己和外部类的静态属性和方法
            System.out.println(name);
            //System.out.println(this.name);静态方法里面连自己类的非静态属性都不能访问
            System.out.println(StaticInnerClass.age);

            System.out.println(StaticInnerClassTest.age);
            //System.out.println(StaticInnerClassTest.this.name);静态方法不能访问非静态属性
            StaticInnerClassTest.go();
            //StaticInnerClassTest.this.run();静态方法不能访问非静态方法
        }

    }
}
```

注意：

1、我们上面说的内部类能够调用外部类的方法和属性，在静态内部类中就行了，因为静态内部类没有了指向外部类对象的引用。除非外部类中的方法或者属性也是静态的。这就回归到了static关键字的用法。

2、静态内部类能够直接被外部类给实例化，不需要使用外部类对象

```
Outer.Inner inner = new Outer.Inner();
```

3、静态内部类中可以声明静态方法和静态变量，但是非静态内部类中就不可以声明静态方法和静态变量

4、局部内部类

局部内部类是在一个方法内部声明的一个类
局部内部类中可以访问外部类的成员变量及方法
局部内部类中如果要访问该内部类所在方法中的局部变量,那么这个局部变量就必须是final修饰的

```
public class Outer {
    private int id;
    //在method01方法中有一个Inner内部类，这个内部类就称为局部内部类
    public void method01(){class Inner{
            public void in(){
                System.out.println("这是局部内部类");
            }
        }
    }
}
```

局部内部类一般的作用跟在成员内部类中总结的差不多，但是有两个要注意的地方：

1. 在局部内部类中，如果要访问局部变量，那么该局部变量要用final修饰

为什么需要使用final?

final修饰变量：变为常量，会在常量池中放着，逆向思维想这个问题，如果不实用final修饰，当局部内部类被实例化后，方法弹栈，局部变量随着跟着消失，这个时候局部内部类对象在想去调用该局部变量，就会报错，因为该局部变量已经没了，当局部变量用fanal修饰后，就会将其加入常量池中，即使方法弹栈了，该局部变量还在常量池中呆着，局部内部类也就是够调用。所以局部内部类想要调用局部变量时，需要使用final修饰，不使用，编译度通不过。

```
public class Outer {
    private int id;
    public void method01(){
        final int cid = 3;    //这个就是局部变量cid。要让局部内部类使用，就得变为final并且赋值，如果不使用final修饰，就会报错
        class Inner{
            //内部类的第一个方法
            public void in(){
                System.out.println("这是局部内部类");
            }
            //内部类中的使用局部变量cid的方法
            public void useCid(){
                System.out.println(cid);
            }
        }

    }
}
```

1. 局部内部类不能通过外部类对象直接实例化，而是在方法中实例化出自己来，然后通过内部类对象调用自己类中的方法。看下面例子就知道如何用了。

```
public class Outer {
    private int id;

    public void out(){
        System.out.println("外部类方法");
    }
    public void method01(){
        class Inner{
            public void in(){
```

```
                System.out.println("这是局部内部类");
            }
        }
//关键在这里，如需要在method01方法中自己创建内部类实例，然后调用内部类中的方法，等待外部类调用method01方法，就可以执行到内部类中的方法了。
        Inner In = new Inner();
        In.in();
    }
}
```

使用局部内部类需要注意的地方就刚才上面说的：

1、在局部内部类中，如果要访问局部变量，那么该局部变量要用final修饰

2、如何调用局部内部类方法。

```
public class LocalInnerClassTest {
    private String name;
    private static int age;

    public void run(){}

    public static void go(){}

    //局部内部类要定义在方法中
    public void test(){
        final String myname="";
        class LocalInnerClass{
            private String name;
//            private static int age;不能定义静态属性

            public void test(String name){
                System.out.println(name);
                System.out.println(this.name);
                System.out.println(myname);
                System.out.println(LocalInnerClassTest.this.name);

                LocalInnerClassTest.this.run();
                LocalInnerClassTest.go();
            }
        }
        //局部内部类只能在自己的方法中用,因为局部内部类相当于一个局部变量，除了方法就找不到了。
        LocalInnerClass lic = new LocalInnerClass();
        lic.name="tom";
        lic.test("test");
    }

}
```

5、匿名内部类

在这四种内部类中，以后的工作可能遇到最多的是匿名内部类，所以说匿名内部类是最常用的一种内部类。

什么是匿名对象？如果一个对象只要使用一次，那么我们就是需要new Object().method()。 就可以了，而不需要给这个实例保存到该类型变量中去。这就是匿名对象。

```
public class Test {
    public static void main(String[] args) {
        //讲new出来的Apple实例赋给apple变量保存起来，但是我们只需要用一次，就可以这样写
        Apple apple = new Apple();
        apple.eat();
        //这种就叫做匿名对象的使用，不把实例保存到变量中。
        new Apple().eat();
    }
}
class Apple{
    public void eat(){
        System.out.println("我要被吃了");
    }
}
```

匿名内部类跟匿名对象是一个道理：

匿名对象：我只需要用一次，那么我就不用声明一个该类型变量来保存对象了，

匿名内部类：我也只需要用一次，那我就不需要在类中先定义一个内部类，而是等待需要用的时候，我就在临时实现这个内部类，因为用次数少，可能就这一次，那么这样写内部类，更方便。不然先写出一个内部类的全部实现来，然后就调用它一次，岂不是用完之后就一直将其放在那，那就没必要那样。

1. 匿名内部类需要依托于其他类或者接口来创建
    - 如果依托的是类,那么创建出来的匿名内部类就默认是这个类的子类
    - 如果依托的是接口,那么创建出来的匿名内部类就默认是这个接口的实现类。
2. 匿名内部类的声明必须是在使用new关键字的时候
    - 匿名内部类的声明及创建对象必须一气呵成,并且之后能反复使用,因为没有名字。

【示例】

A是一个类(普通类、抽象类都可以)，依托于A类创建一个匿名内部类对象

```
main:

A a = new A(){
    //实现A中的抽象方法
    //或者重写A中的普通方法
};
注:这个大括号里面其实就是这个内部类的代码,只不过是声明该内部类的同时就是要new创建了其对象,并且不能反复使用,因为没有名字。

例如:
B是一个接口，依托于B接口创建一个匿名内部类对象

B b = new B(){
    //实现B中的抽象方法
};
```

1. 匿名内部类除了依托的类或接口之外,不能指定继承或者实现其他类或接口,同时也不能被其他类所继承,因为没有名字。
2. 匿名内部中,我们不能写出其构造器,因为没有名字。
3. 匿名内部中,除了重写上面的方法外,一般不会再写其他独有的方法,因为从外部不能直接调用到。(间接是调用到的)

```
public interface Work{
    void doWork();
}
public class AnonymousOutterClass{
    private String name;
    private static int age;
    public void say(){}
    public static void go(){}

    public void test(){
        final int i = 90;

        Work w = new Work(){
            public void doWork(){
                System.out.println(AnonymousOutterClass.this.name);
                System.out.println(AnonymousOutterClass.age);
                AnonymousOutterClass.this.say();
                AnonymousOutterClass.go();

                System.out.println(i);
            }
        };
        w.doWork();
    }
}
```

我们可以试一下不用匿名内部类和用匿名内部类实现一个接口中的方法的区别

【不用匿名内部类】

```
public class Test {
    public static void main(String[] args) {
       //如果我们需要使用接口中的方法，我们就需要走3步，1、实现接口 2、创建实现接口类的实例对象 3、通过对象调用方法
          //第二步
        Test02 test = new Test02();
          //第三步
        test.method();
    }
}

//接口Test1
interface Test01{
    public void method();
}
//第一步、实现Test01接口
class Test02 implements Test01{

    @Override
    public void method() {
        System.out.println("实现了Test接口的方法");
    }

}
```

【使用匿名内部类】

```
public class Test {
```

```
    public static void main(String[] args) {
          //如果我们需要使用接口中的方法，我们只需要走一步，就是使用匿名内部类，直接将其类的对象创建出来。
         new Test1(){
             public void method(){
                 System.out.println("实现了Test接口的方法");
             }
          }.method();

    }
}

interface Test1{
    public void method();
}
```

解析：其实只要明白一点，new Test1(){实现接口中方法的代码}; Test1(){...}这个的作用就是将接口给实现了，只不过这里实现该接口的是一个匿名类，也就是说这个类没名字，

只能使用这一次，我们知道了这是一个类， 将其new出来，就能获得一个实现了Test1接口的类的实例对象，通过该实例对象，就能调用该类中的方法了，因为其匿名类是在一个类中实现的，

所以叫其匿名内部类，不要纠结为什么Test1(){...}就相当于实现了Test1接口，这其中的原理等足够强大了，在去学习，不要钻牛角尖，这里就仅仅是需要知道他的作用是什么，做了些什么东西就行。

狂神社群笔记资料，禁止外传，本人QQ：24736743