类和对象

类和对象

- 引用
  - 形参与实参
  - 值传递和引用传递
- 成员变量和局部变量
  - 成员变量
  - 局部变量
- 匿名对象
- 面向对象三大特征之1: 封装(Encapsulation)
  - 实现封装的方法： private(私有)关键字
    - private关键字：
      - 常用方法：
- 构造函数
  - 构造函数的特点
  - 构造代码块
    - 和构造函数的区别：
- This关键字
  - 用法1：用于区分**局部变量和成员变量同名**的情况
  - 用法2: 构造函数间的相互调用
- 静态(Static)关键字
  - static特点
  - 实例变量和类变量(静态变量)的区别
  - 静态使用注意事项
  - 使用静态的利弊
    - 利处
    - 弊端
  - 什么时候使用静态?
    - 什么时候定义静态变量(类变量)
    - 什么时候定义静态函数?
  - 静态的应用：工具类
    - 使用工具类的问题
  - 静态代码块
- 内部类
  - 成员内部类
  - 局部内部类
  - 静态内部类
  - 匿名内部类
- main函数

- 主函数的定义
- 对象初始化过程
- 单例设计模式
  - 单例设计模式：解决**一个类在内存只存在一个对象**。
    - 保证对象唯一性
  - 饿汉式和懒汉式
    - 饿汉式
    - 懒汉式
    - **开发时一般用饿汉式**
- 面向对象三大特征之2: 继承(Inheritance)
  - 继承的优点
  - 如何使用一个继承体系中的功能
  - 子父类出现后，类成员的特点
    - 变量
    - 方法
      - 函数的另一个特性：override重写
        - 注意事项
      - 构造函数的继承
        - 原因
- final关键字
- 抽象类
  - 为什么要使用抽象类
- 接口
  - 接口的定义：
  - 接口的特征
  - 接口型引用
  - JDK1.8之后的新增特性
    - lambda(λ)表达式
      - 函数型接口
    - default 修饰符
- 面向对象三大特性之3: 多态(Polymorphism)
  - 多态的体现
  - 多态的前提
  - 多态的好处
  - 多态的弊端
  - 多态的应用
  - 转型
    - 向上转型upcasting
    - 向下转型downcasting
    - instance of
- 异常处理机制
  - 异常的由来
    - 对于问题的划分
  - 异常的处理

异常对象的常见方法操作
throws语句
对多异常的处理
自定义异常
自定义异常的特点
throws和throw的区别
RuntimeException子类
异常的分类

引用

如果一个变量的类型是**类**类型，而非**基本**类型，那么该变量又叫做**引用**。
引用-HOW2J

形参与实参

- 形参是指在**定义函数时使用的参数**，目的是用于接收调用该函数时传入的参数。简单理解，就是所有函数（即方法）的参数都是形参。
- 实参是**指调用函数时，传递给函数的参数**。

```
public static void main(String[] args) {
    int num = 3;
    printVal(num); //这里num是实参
}

private static void printVal(int num) {
    num = 5; //这里num就是形参
}
```

值传递和引用传递

- 值传递(值调用,call by value)：是指在调用函数(方法)时，**将实际参数复制一份传递给函数**，方法接收的是调用者提供的值，在函数中修改参数时，不会影响到实际参数。其实，就是在说值传递时,方法接收的是调用者提供的变量地址，只会改变形参，不会改变实参。
- 引用传递(引用调用,call by reference)：是指在调用函数时，**将实际参数的地址传递给函数**，方法接收的是调用者提供的变量地址。这样在函数中对参数的修改将影响到实际参数。

需要特别强调的是，**千万不要以为传递的参数是值就是值传递，传递的是引用就是引用传递。也不要以为传递的参数是基本数据类型就是值传递，传递的是对象就是引用传递。** 判断是值传递还是引用传递的标准，和传递参数的类型是没有一毛钱关系的。

参数类型可以分为以下三种情况：

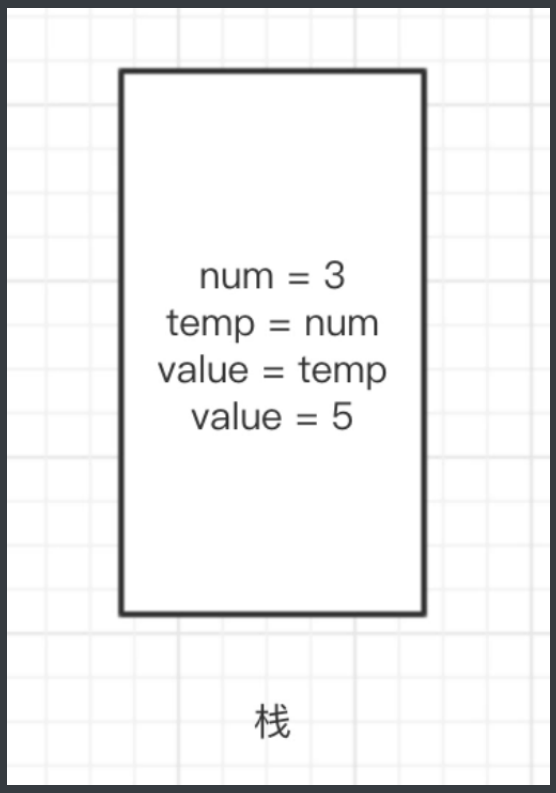
1. **当传递的参数是基本数据类型时：**

```
public class TestNum {
        public static void main(String[] args) {
            int num = 3;
            System.out.println("修改前的num值:"+num);
            changeValue(num);//实参
            System.out.println("修改后的num值:"+num);
        }

        private static void changeValue(int num) { //形参
            num = 5;
            System.out.println("形参num值:"+num);
        }

    }
```

```
修改前的num值:3
形参num值:5
修改后的num值:3
```

可以发现，传递基本数据类型时，在函数中修改的仅仅是形参，对实参的值的没有影响。

需要明白一点，**值传递不是简单的把实参传递给形参，而是，实参建立了一个副本，然后把副本传递给了形参。**下面用图来说明一下参数传递的过程：图中num是实参，然后创建了一个副本temp，把它传递给形参value，**修改value值对实参num没有任何影响。**



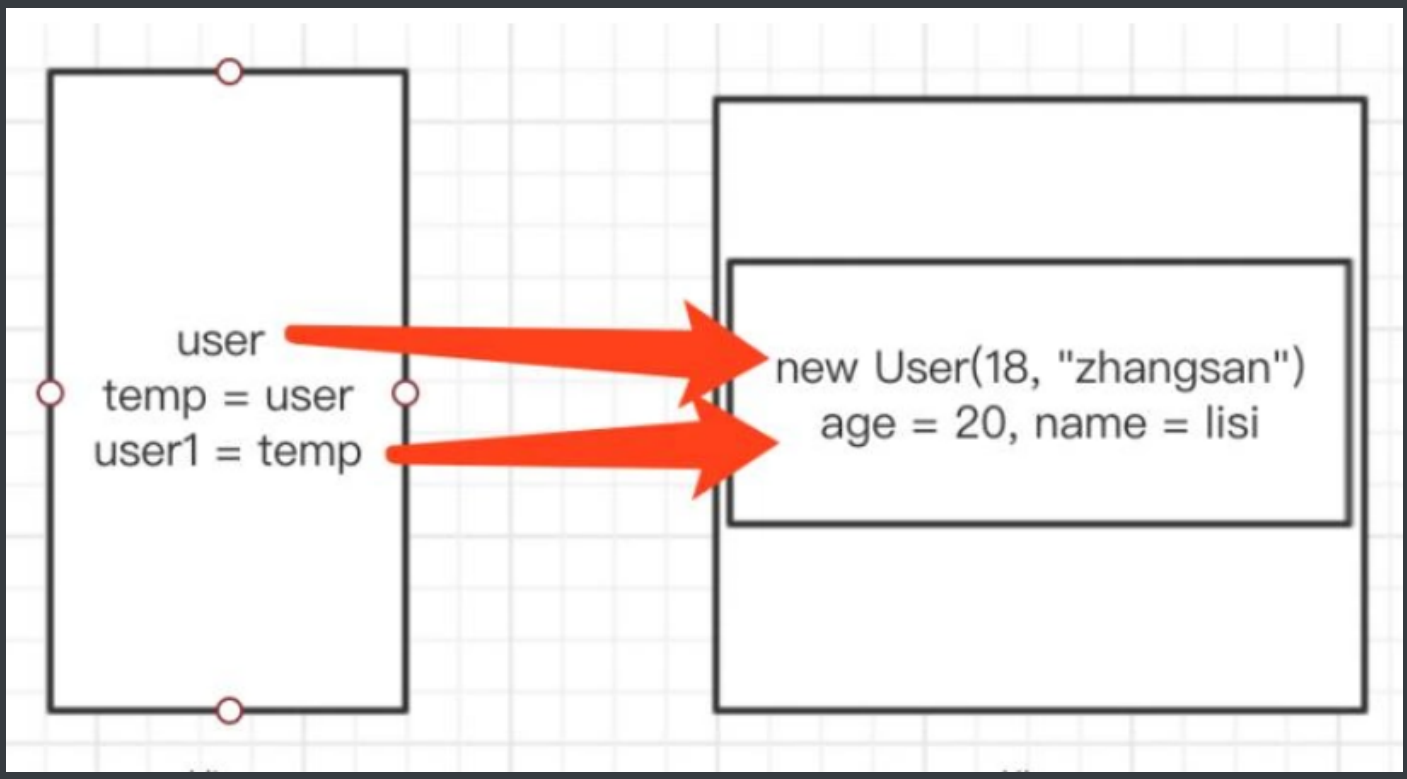
2. **传递类型是引用类型时：**

```
public class User {
    private int age;
    private String name;
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public User(int age, String name) {
        this.age = age;
        this.name = name;
```

```
        }
        @Override
        public String toString() {
            return "User{" +
                    "age=" + age +
                    ", name='" + name + '\'' +
                    '}';
        }

    }
    public class TestUser {
        public static void main(String[] args) {
            User user = new User(18, "zhangsan");
            System.out.println("修改对象前:"+user);
            changeUser(user);
            System.out.println("修改对象后:"+user);
        }

        private static void changeUser(User user) {
            user.setAge(20);
            user.setName("lisi");
        }

    }
```

```
修改对象前:User{age=18, name='zhangsan'}
修改对象后:User{age=20, name='lisi'}
```

可以发现，传过去的user对象，属性值被改变了。由于，user对象存放在堆里边，其引用存放在栈里边，其参数传递图如下**:user是对象的引用，为实参，然后创建一个副本temp，把它传递给形参user1。**但是，他们实际操作的都是堆内存中的同一个User对象。因此，对象内容的修改也会体现到实参user上。



3. **传递类型是String类型（Integer等基本类型的包装类等同）**

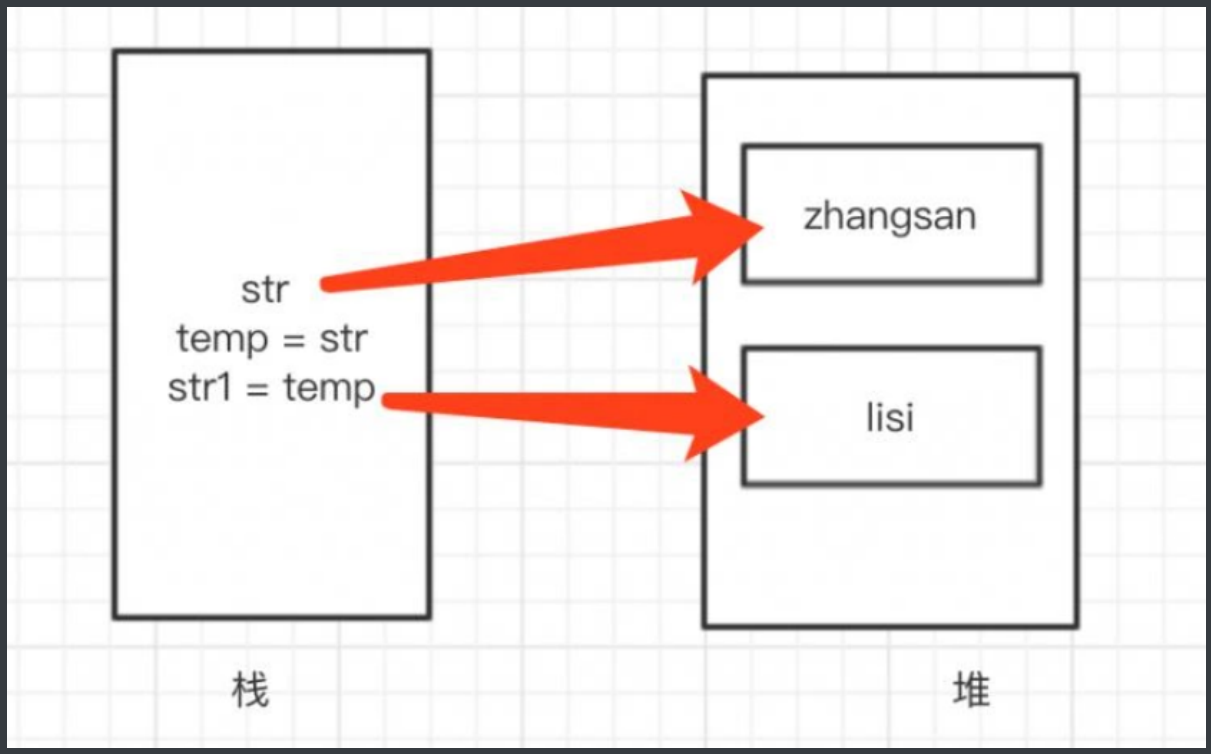
```
public class TestStr {
    public static void main(String[] args) {
        String str = new String("zhangsan");
        System.out.println("字符串修改前:"+str);
        changeStr(str);
        System.out.println("字符串修改后:"+str);
    }

    private static void changeStr(String str) {
        str = "lisi";
    }

}
```

```
字符串修改前:zhangsan
字符串修改后:zhangsan
```

按照第二种情况，传递参数是引用类型时，不是可以修改对象内容吗，String也是引用类型，为什么在这又不变了呢?

再次强调一下，**传递参数是引用类型，并不代表就是引用传递，其实它还是值传递。**此时的 lisi 和上边的 zhangsan 根本不是同一个对象。画图理解下:

图中，str是对象 zhangsan 的引用，为实参，然后创建了一个副本temp，把它传递给了形参str1。**此时，创建了一个新的对象 lisi ，形参str1指向这个对象，但是原来的实参str还是指向zhangsan。**因此，形参内容的修改并不会影响到实参内容。所以，两次打印结果都是zhangsan。



第三种情况和第二种情况虽然传递的都是引用类型变量，但是处理方式却不一样。第三种情况是创建了一个新的对象，然后把形参指向新对象，而第二种情况并没有创建新对象，操作的还是同一个对象。如果把上边changeUser方法稍作改变，你就会理解:

```
private static void changeUser(User user) {
    //添加一行代码，创建新的User对象
    user = new User();
    user.setAge(20);
    user.setName("lisi");
}
```

运行以上代码，你就会惊奇的发现，最终打印修改前和修改后的内容是一模一样的。
这种情况，就等同于第三种情况。因为，这里的形参和实参引用所指向的对象是不同的对象。因此，修改形参对象内容并不会影响实参内容。

```
修改对象前:User{age=18, name='zhangsan'}
修改对象后:User{age=18, name='zhangsan'}
```

从以上三个例子中，我们就能理解了，为什么Java中只有值传递，并没有引用传递。**值传递，不论传递的参数类型是值类型还是引用类型，都会在调用栈上创建一个形参的副本。**不同的是，对于值类型来说，复制的就是整个原始值的复制。而对于引用类型来说，由于在调用栈中只存储对象的引用，因此**复制的只是这个引用，而不是原始对象。**

最后，再次强调一下，传递参数是引用类型，或者说是对象时，并不代表它就是引用传递。引用传递不是用来形容参数的类型的，不要被“引用”这个词本身迷惑了。这就如同我们生活中说的地瓜不是瓜，而是红薯一样。

1. 参数传递时，是**拷贝实参的副本，然后传递给形参。**（值传递）
2. 在函数中，只有修改了**实参**所指向的对象内容，才会影响到实参。以上第三种情况修改的实际上只是形参所指向的对象，因此不会影响实参。

成员变量和局部变量

成员变量

成员变量**定义在类中，在整个类中都可以被访问**。

成员变量**随着对象的建立而建立**，存在于对象所在的**堆内存**中。

成员变量**有默认初始化值**

局部变量

- 局部变量只定义在局部范围内，如：函数(例如for循环)内，语句内等
- 局部变量存在于**栈内存**中。作用的范围结束，变量空间会自动释放。
- 局部变量**没有默认初始化值**(必须手动初始化)。 当没有手动初始化局部变量时，编译器会报错 **(Variable might not have been initialized error)**
  - 原因：初始化时机不同，类的成员变量在开辟堆内存时就会被JVM统一初始化，很方便，但是局部变量可能在方法被调用时还没有定义，每次都让JVM定义很消耗资源。
  - 局部变量运行时被分配在栈中，量大，生命周期短，如果虚拟机给每个局部变量都初始化一下，是一笔很大的开销，但变量不初始化为默认值就使用是不安全的。出于速度和安全性两个方面的综合考虑，解决方案就是虚拟机不初始化，但要求编写者一定要在使用前给变量赋值。

```
    for (int i ; i<= test2.length ; i++){// 编译会报错

            }
```

匿名对象

匿名对象是对象的简化形式

匿名对象两种使用情况

• 当对对象方法**仅进行一次调用**的时

• 匿名对象可以**作为实际参数进行传递**

面向对象三大特征之1: 封装(Encapsulation)

所谓封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。封装是面向对象的特征之一，是对象和类概念的主要特性。简单的说，一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。

指**隐藏对象的属性和实现细节**，仅对外提供公共访问方式, 通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

实现封装的方法： private(私有)关键字

private关键字：

- 是一个权限修饰符。
- 用于修饰成员(成员变量和成员函数)
- 被私有化的成员只在本类中有效。

常用方法：

将**成员变量**私有化，对外提供对应的**set ， get 方法**对其进行访问。提高对数据访问的安全性。

构造函数

- 作用：**对对象进行初始化**，设定基本特性/行为
- 对象一建立，就会调用与之对应的构造函数

构造函数的特点

- **函数名与类名相同**
- **不用定义返回值类型**
- **不可以写return语句**
- 当一个类中没有构造函数时，会**自动建立无参构造函数**
- 但是自定义构造函数够，就不自动建立了。
- 构造函数也可以**重载**(初始化时的**参数列表可能不一样**，所以我们要分别建立构造函数，对参数不同进行不同的初始化)
- 构造函数**只在对象建立时执行一次**
- 构造函数也可以私有化(private)，但是这样就不能通过这个构造方法建立对象(如果所有的构造函数都被私有化，那么这个类的对象不可能被创建)
- **构造函数间互相调用**不能写函数名，要用this关键字(但要注意避免死循环)

构造代码块

- 对象一建立就运行（**优先于构造函数**）

和构造函数的区别：

- 构造代码块是给**所有对象**进行统一初始化，因为构造函数可以重载，运行哪个构造函数就会创建什么样的对象，但是无论构建什么样的对象，都会先执行构造代码块；
- 而构造函数是给对应的对象进行初始化。
- 构造代码块一定会执行，所以写所有对象的**共性初始化内容**

This关键字

- this代表**本类对象的引用**(在哪个对象的方法里，就代表哪个对象)
- 但凡**本类功能内部使用了了本类对象**，都用this表示

用法1：用于区分局部变量和成员变量同名的情况

```
class PersonDemo3
{
  public static void main(String[] args)
  {

    Person p = new Person();

  }
}

class Person
```

```
    {
      private int age;
      Person(int age)
      {
        this.age = age;// 这里是Person类的对象p,所以this.age 就是 p.age
      }
    }
```

用法2: 构造函数间的相互调用

- 在构造函数互相调用时，this语句**必须**写在该构造方法的**第一行**。因为**初始化动作要先执行**，如果初始化里面还有更细节的初始化，应当先执行这个更细节的初始化

静态(Static)关键字

- 是一个修饰符，用于**修饰成员(成员变量，成员函数).**
- 当成员被静态修饰后，**除了可以被对象调用外，还可以直接被类名调用**。用法：类名.静态成员
- 特有内容随着对象存储，共享内容随着类存储(存放在**方法区**)

static特点

- **随着类的加载而加载。也就是说：静态会随着类的消失而消失**。说明它的**生命周期最长。**
- 优先于的对象存在 **静态是先存在。对象是后存在的。**
- 被**所有对象**所共享
- 可以直接被类名所调用。

实例变量和类变量(静态变量)的区别

1. 存放位置。
   类变量随着类的加载而**存在于方法区**中。
   实例变量随着对象的建立而存在于堆内存中。
2. 生命周期：
   类变量生命周期最长，**随着类的消失而消失**。
   实例变量生命周期**随着对象的消失而消失**。

静态使用注意事项

1. 静态方法**只能访问静态成员**(因为非静态成员属于对象)，非静态方法既可以访问静态也可以访问非静态成员
2. 静态方法中**不可以**定义this，super关键字。因为静态优先于对象存在。所以静态方法中不可以出现this
3. **主函数是静态的**

使用静态的利弊

利处

- 对对象的共享数据进行单独空间的存储，**节省内存**(没有必要每一个对象中都存储一份)，还**可以直接被类名调用**。

弊端

1. 生命周期过长
2. 访问出现局限性。(静态**只能访问静态**)

什么时候使用静态?

什么时候定义静态变量(类变量)

1. 当对象中出现**共享数据**时，该数据被静态所修饰。
2. 对象中**的特有数据要定义成非静态**存在于堆内存中。

什么时候定义静态函数?

- 当功能内部**没有访问到非静态数据**(对象的特有数据)，那么该功能可以定义成静态的。

静态的应用：工具类

- 每一个应用程序中都有共性的功能，可以将这些功能进行抽取和独立封装。**以便复用**。
- 可以通过建立ArrayTool的对象使用这些工具方法，对数据进行操作。

使用工具类的问题

- 对象是用于封装数据的，可是ArrayTool对象**并未封装特有数据**。
- 操作数组的**每一个方法都没有用到ArrayTool对象中的特有数据**。

所以其实**是不需要对象的**。可以**将ArrayTool中的方法都定义成static的。直接通过类名调用**即可。

但是该类**还是可以被其他程序建立对象的**。为了更为严谨，强制让该类不能建立对象。
可以通过**将构造函数私有化**完成。

静态代码块

用staitc声明，**jvm加载类时执行，仅执行一次**

```
static {//静态代码块
}
```

1. 它是**随着类的加载而执行，只执行一次，并优先于主函数**。具体说，**静态代码块是由类调用**的。类调用时，先执行静态代码块，然后才执行主函数的。
2. 静态代码块其实就是**给类初始化**的，而构造代码块是**给对象初始化**的。
3. 静态代码块中的变量是局部变量，与普通函数中的局部变量性质没有区别。
4. 一个类中可以有多个静态代码块

执行顺序优先级：静态块 -> 成员变量声明 -> main() -> (*构造块,成员变量赋值*) -> 构造方法。

构造代码块和成员变量的赋值的顺序是根据代码顺序决定的

```
public class Z extends X {
    public static void main(String[] args) {
        new Z();
    }
    Z() {
        System.out.println("Z");
    }
    Y y = new Y();
}
class X {
    X() {
        System.out.println("X");
    }
    Y b = new Y();
}
class Y {
    Y() {
        System.out.println("Y");
    }
}
```

上述代码执行时，输出的结果会是YXYZ, 原因为在执行Z类的构造函数之前，首先要执行其父类的成员变量赋值(输出Y)和无参构造函数(输出X),其后要执行自己的成员变量赋值(输出Y),最后才是Z类的构造函数(输出Z)

内部类

成员内部类

当描述事物时，**事物的内部还有事物**，该事物用**内部类**来描述。**因为内部事物在使用外部事物的内容**(例如心脏在人体内部,所以可以直接访问人体内部的其他器官)。

即作为外部类的一个**成员**存在，与外部类的属性、方法并列。

```
//注意：成员内部类中不能定义静态变量,但可以访问外部类的所有成员。
public class Outer{
    private static int i = 1;
    private int j=10;
    private int k=20;
    public static void outer_f1(){
        //do more something
    }
    public void outer_f2(){
        //do more something
    }

    //成员内部类
    class Inner{
        //static int inner_i =100; //内部类中不允许定义静态变量
        int j=100;//内部类中外部类的实例变量可以共存
        int inner_i=1;
        void inner_f1(){
        System.out.println(i);//外部类的变量如果和内部类的变量没有同名的，则可以直接用变量名访问外部类的变量
        System.out.println(j);//在内部类中访问内部类自己的变量直接用变量名
        System.out.println(this.j);//也可以在内部类中用"this.变量名"来访问内部类变量
    //访问外部类中与内部类同名的实例变量可用"外部类名.this.变量名"。
        System.out.println(k);//外部类的变量如果和内部类的变量没有同名的，则可以直接用变量名访问外部类的变量
        outer_f1();
        outer_f2();
    }
}
    //外部类的非静态方法访问成员内部类
    public void outer_f3(){
        Inner inner = new Inner();
        inner.inner_f1();
    }
```

```
        //外部类的静态方法访问成员内部类，与在外部类外部访问成员内部类一样
        public static void outer_f4(){
            //step1 建立外部类对象
            Outer out = new Outer();
            //***step2 根据外部类对象建立内部类对象***
            Inner inner=out.new Inner();
            //step3 访问内部类的方法
            inner.inner_f1();
        }

        public static void main(String[] args){
            outer_f4();
        }
    }
```

所以类中除了可以定义变量和方法外，还可以定义其他类——内部类

- 一个类里可以**定义多个内部类**
- 内部类可以**直接访问外部类中的成员**(包括private的)(因为**内部类里持有了一个内部类的引用**)
- 外部类要访问内部类，**必须要建立内部类对象**
- 在建立内部类的对象时，要**使用全名**称实例化对象

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

- 成员内部类里的变量**不可以用static修饰**(因为是局部变量，static 只能修饰成员)

成员内部类的优点：
⑴ 内部类作为外部类的成员，可以访问外部类的私有成员或属性。（即使将外部类声明为private，但是对于处于其内部的内部类还是可见的。）
⑵ 用内部类定义在外部类中不可访问的属性。这样就在外部类中**实现了比外部类的private还要小的访问权限**。
注意：内部类是一个编译时的概念，一旦编译成功，就会成为完全不同的两类。对于一个名为outer的外部类和其内部定义的名为inner的内部类。编译完成后出现outer.class和outer$inner.class两类。

局部内部类

与局部变量类似,在方法中定义的内部类，在局部内部类前不加修饰符public或private，**其范围为定义它的代码块。**
注意：局部内部类中不可定义静态变量，可以访问外部类的局部变量(即方法内的变量)，但是变量必须是final的。

```
    public class Outer {
         private int s = 100;
         private int out_i = 1;
         public void f(final int k){
         final int s = 200;
```

```
        final int j = 10;
        class Inner{ //定义在方法内部
            int s = 300;//可以定义与外部类同名的变量
            //static int m = 20;//不可以定义静态变量
            Inner(int k){
                inner_f(k);
            }
            int inner_i = 100;
            void inner_f(int k){
                System.out.println(out_i);//如果内部类没有与外部类同名的变量，在内部类中可以直接访问外部类的实例变量
                System.out.println(k);//*****可以访问外部类的局部变量(即方法内的变量)，但是变量必须是final的*****
                //System.out.println(i);
                System.out.println(s);//如果内部类中有与外部类同名的变量，直接用变量名访问的是内部类的变量
                System.out.println(this.s);//用"this.变量名" 访问的也是内部类变量
                System.out.println(Outer.this.s);//用外部"外部类类名.this.变量名" 访问的是外部类变量
            }
        }
        new Inner(k);
    }

    public static void main(String[] args) {
        //访问局部内部类必须先有外部类对象
        Outer out = new Outer();
        out.f(3);
    }

    }
```

注意：
在类外不可直接生成局部内部类（保证局部内部类对外是不可见的）。要想使用局部内部类时需要生成对象，对象调用方法，在方法中才能调用其局部内部类。通过内部类和接口达到一个强制的弱耦合，用局部内部类来实现接口，并在方法中返回接口类型，使局部内部类不可见，屏蔽实现类的可见性。

静态内部类

静态内部类定义在类中，用static定义。

```
//注意：静态内部类中可以定义静态或者非静态的成员
public class Outer {
    private static int i = 1;
    private int j = 10;
    public static void outer_f1(){

}
public void outer_f2(){

}
// 静态内部类可以用public,protected,private修饰
// 静态内部类中可以定义静态或者非静态的成员
static class Inner{
    static int inner_i = 100;
    int inner_j = 200;
    static void inner_f1(){
        System.out.println("Outer.i"+i);//静态内部类只能访问外部类的静态成员
        outer_f1();//包括静态变量和静态方法
    }
    void inner_f2(){
        //System.out.println("Outer.i"+j);//静态内部类不能访问外部类的非静态成员
        //outer_f2();//包括非静态变量和非静态方法
    }
}
public void outer_f3(){
    //外部类访问内部类的静态成员：内部类.静态成员
    System.out.println(Inner.inner_i);
    Inner.inner_f1();
    //外部类访问内部类的非静态成员:实例化内部类即可
    Inner inner = new Inner();
    inner.inner_f2();
}
public static void main(String[] args) {
    new Outer().outer_f3();
}

}
```

注意：生成（new）一个静态内部类不需要外部类成员：这是静态内部类和成员内部类的区别。静态内部类的对象可以直接生成：

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

而不需要通过生成外部类对象来生成。这样实际上使静态内部类成为了一个顶级类。静态内部类不可用private来进行定义。

匿名内部类

- 匿名内部类其实就是内部类的简写格式。
- **定义匿名内部类的前提**：
- - 内部类必须**继承一个类或者实现接口**。
- 匿名内部类的格式： ==**new** 父类或者接口(){ 定义子类的内容 }==
- 内部类本质上是一个**匿名子类对象**。而且这个对象有点胖(可以理解为**带内容的对象**)
- 匿名对象**只能调用一次方法**
- 在建立引用时建立的是父类的引用

```
AbsDemo d = new AbsDemo()//父类引用指向子类对象
    {
      int num = 9;
      void show()
      {
        System.out.println("num==="+num);
      }
      void abc()
      {
        System.out.println("haha");
      }
    };
```

- 匿名内部类中定义的方法最好不要超过2个(因为使用匿名内部类继承的接口或抽象类中所有的方法都要实现)

main函数

- 主函数是一个特殊的函数。作为**程序的入口**，可以被JVM调用

主函数的定义

- public：代表着该函数访问权限是最大的。
- static：代表主函数随着类的加载就已经存在了。
- void：主函数没有具体的返回值。
- main：**不是关键字**，但是是一个特殊的单词，可以被jvm识别。
- （String[] arr）:函数的参数，参数类型是一个**字符串类型的数组**。
- 主函数是固定格式的：jvm识别。jvm只能识别String[] args这种参数列表的main函数(重载也没用)
- jvm在调用主函数时，传入的是**new String[0]**;

对象初始化过程

```
person p = new Person("zhangsan",20);
```

该句话都做了什么事情?

1. 因为new用到了Person.class.所以会**先找到Person.class文件并加载到内存**中。
2. 执行该类中的**static代码块**，如果有的话，给Person.class类进行初始化。
3. 在**堆内存中开辟空间，分配内存地址**。
4. 在堆内存中**建立对象的特有属性。并进行默认初始化**。
5. 对属性进行**显示初始化**。
6. 对对象进行**构造代码块**初始化。
7. 对对象进行对应的**构造函数**初始化。
8. 将**内存地址付给栈内存中的p变**量。

单例设计模式

- 设计模式：解决某一类问题**最行之有效的**方法。
- java中有23种设计模式：

单例设计模式：解决一个类在内存只存在一个对象。

当**需要将该事物的对象保证在内存中唯一时**，就采用单例设计模式

保证对象唯一性

1. 为了避免其他程序过多建立该类对象。先**禁止其他程序建立该类对象（方法：将构造函数私有化）**
2. 还为了让其他程序可以访问到该类对象，只好在本类中，**自定义一个对象（方法:在类中创建一个本类对象）**
3. 为了方便其他程序对自定义对象的访问，可以*对外提供一些访问方式(方法：提供一个方法来获取到该对象)。

饿汉式和懒汉式

饿汉式

先初始化对象，**类被加载到内存中时就已经建立好了对象**

```
class Single
{
  private static Single s = new Single();
  private Single(){}
  public static Single getInstance()
  {
    return s;
  }
}
```

懒汉式

对象是方法**被调用时，才初始化，也叫做对象的延时加载**。

```
class Single
{
  private static Single s = null;
  private Single(){}
  public static Single getInstance()
  {
    if(s==null)
    {
      synchronized(Single.class)
      {
        if(s==null)
          s = new Single();
      }
    }
    return s;
  }
}
```

开发时一般用饿汉式

面向对象三大特征之2: 继承(Inheritance)

继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用基类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力；

语法： class A extends B

继承的优点

1. **提高了代码的复用性**。
2. 让**类与类之间产生了关系**。有了这个关系，才有了**多态**的特性。

- 注意：千万不要为了获取其他类的功能，简化代码而继承。**必须是类与类之间有所属关系**才可以继承。
- 判断所属关系的方式： 继承一下，然后**看看父类中所有的内容是不是子类都应该具备**
- 在java中，**一个类只能有一个直接父类**（当多个父类中定义了多个同名内容时就会产生歧义）

如何使用一个继承体系中的功能

1. 先查阅体系父类的描述，因为父类中定义的是该体系中**共性功能**。
2. 通过了解共性功能，就可以知道该体系的基本功能。
3. 在具体调用时，要**创建最子类的对象**，为什么呢?

- **有可能父类不能创建对象**（如静态）
- **创建子类对象可以使用更多的功能**，包括基本的也包括特有的。

总结：**查阅父类功能，创建子类对象使用功能。**

子父类出现后，类成员的特点

- 在堆内存中使用new语句创建对象时，**先加载父类.class文件到内存，再加载子类.class文件到内存。**

变量

如果子类中出现非私有的同名成员变量时，

- 子类要访问**本类中的变量**，用this
- 子类要访问**父类中的同名变量**，用super。
- super的使用和this的使用几乎一致。this代表的是本类对象的引用。super代表的是父类对象的引用。

方法

函数的另一个特性：override重写

- 当子类出现和父类一模一样的函数时，当**子类对象调用该函数，会运行子类函数的内容**。如同父类的函数被覆盖一样。
- 当子类继承父类，沿袭了父类的功能到子类中，但是**子类虽具备该功能，但是功能的内容却和父类不一致**，这时，没有必要定义新功能，而是使用重写，**保留父类的功能定义，并重写功能内容**。

注意事项

- 子类重写父类，必须**保证子类权限==大于等于==父类权限**，才可以重写，否则编译失败。

- **静态只能重写静态**。（先有静态才有对象）

构造函数的继承

- 构造函数**不可以重写**（因为**构造函数就不是函数**，没有返回值类型，自然没有函数可以被重写的特点）
- **在对子类对象进行初始化时，父类的构造函数也会运行**
- 因为**子类的所有构造函数默认第一行有一条隐式的语句 super()**(访问父类中**空参数的构造函数**)
- 当父类中没有空参数的构造函数时，就要**手动指定**访问父类中哪一个构造函数
- super和this关键字一样，都**只能放在构造函数的第一行**
- 所以**若要调用本类中的其他构造函数this()，隐式的super()就会被省略**

原因

- 因为父类中的数据子类可以直接获取。所以子类对象在建立时，需要**先查看父类是如何对这些数据进行初始化的**。

```
class Father
{
  int num ; // 这里先进行了隐式初始化
  Father()
  {
    num = 60; //这里后进行显示初始化
    System.out.println("father run");
  }
}

class Son extends Fu
{
  Son()
  {

    //super();
    System.out.println("son run");
  }
```

```
}

class  ExtendsDemo
{
  public static void main(String[] args)
  {
    Son z = new Son(0);
    System.out.println(z.num);
  }
}
```

在这个例子中，因为**父类的构造函数对num进行了赋值**，访问子类对象的同名参数值时就需要先访问父类的构造函数

- 如果要访问父类中**指定的构造函数**，可以通过**手动定义super语句**的方式来指定。
- 所以子类中 **==至少会有一个==构造函数会访问父类中的构造函数**。
- Object类**是所有类的父类**

final关键字

继承的存在一定程度上**打破了封装性**。所以我们会用final修饰符

1. 可以**修饰类，函数，变量**。
2. ==被final修饰的类==**不可以被继承**。为了避免被继承，被子类复写功能。
3. ==被final修饰的方法==**不可以被override**。
4. ==被final修饰的变量==是一个常量，**只能赋值一次**，既可以修饰成员变量，有可以修饰局部变量。当在描述事物时，一些数据的出现值是固定的，那么这时为了增强阅读性，都给这些值起个名字。方便于阅读。而这个值不需要改变，所以加上final修饰。作为常量：**常量的书写规范所有字母都大写，如果由多个单词组成。单词间通过_连接。**
5. 常和public static连用，组成全局常量。
6. 内部类定义在类中的局部位置上时，只能访问该局部被final修饰的局部变量。

抽象类

当多个类中出现相同功能，但是功能主体不同，
这是可以进行向上抽取。这时，**只抽取功能定义，而不抽取功能主体**。
这种方法没有方法体，要用abstract关键字修饰，写法为

```
abstract class Student
{
  abstract void study();// 没有方法体

}
```

1. 抽象**方法一定在抽象类**中
2. 抽象方法和抽象类都必须**被abstract关键字修饰**。
3. 抽象类**不可以用new创建对象**。因为调用抽象方法没意义(没有方法体)。
4. 如果要使用抽象类中的抽象方法，必须由子类**override==所有==的抽象方法**后，建立子类对象调用。**如果子类只覆盖了部分抽象方法，那么该子类还是一个抽象类。**

为什么要使用抽象类

- 强迫子类override抽象方法
- 抽象类中**可以不定义抽象方法，这样做仅仅是不让该类建立对象**。

接口

接口可以认为是一个特殊的抽象类。

- 当**抽象类中的方法都是抽象的，那么该类可以通过接口的形式来表示。**
- - class用于定义类
- - ==interface 用于定义接口==。

接口的定义：

- 接口中常见定义：常量，抽象方法。
- 接口中的成员**都有固定修饰符**。(即使不写也会自动补上)
  - 常量：public static **final**
  - 方法：public **abstract**

接口的特征

- 接口是**不可以创建对象**的(因为有抽象方法)
- 接口的**所有方法都需要被子类实现**(override)，子类才可以实例化，否则子类仍然是抽象的。
- **一个类可以实现多个接口**(实现多继承的形式)(因为接口没有方法体，即使两个接口里的同名抽象类被子类实现了也不会混淆)
- 一个类可以同时**继承一个类和实现多个接口**
- 接口之间也可以有继承关系，甚至**一个接口可以继承多个接口**

接口型引用

接口型引用只能指向自己的子类对象

```
interface User{ //一个接口
    public  abstract void someMethod();
}
public class Main {
    public static void main(String[] args) {
        User thisUser = new UserImpl();// 指向thisUser引用，所有User接口的实现类及其子类都可以放到thisUser这个容器里
    }
}
class UserImpl implements User{
    @Override
    public void someMethod() {
    }
}
```

JDK1.8之后的新增特性

lambda(λ)表达式

函数型接口

default 修饰符

- 1.8之前：接口类只能定义方法名，返回类型和参数列表
- 1.8之后：接口类可以有静态static方法，默认default方法，也就是说接**口中可以有方法的具体实现**。

如果想在接口类中定义了方法并给出具体的方法体，可以通过

1. 将该方法定义为静态方法。
2. 为该方法加上default关键字。

```
interface User{
    public  abstract void someMethod();//定义方法
    public static final  int someVariation  = 0;//定义变量

    public static void someMethod2() {//JDK1.8之后新增：接口中可以定义static方法
        System.out.println("do some thing...");
    }

    public default void someMethod3(){//JDK1.8之后新增：接口中可以定义default方法(相当于类的成员方法)
        System.out.println("do some thing...");
    }
}
```

面向对象三大特性之3: 多态(Polymorphism)

所谓多态就是指**一个类实例的相同方法在不同情形有不同表现形式**。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）**可以通过相同的方式予以调用。**

最常见的多态就是将子类传入父类参数中，运行时调用父类方法时通过传入的子类决定具体的内部结构或行为。

多态可以理解为**事物存在的多种体现形态**

多态的体现

- **父类的引用指向了自己的子类对象**。
- 父类的引用也可以接收自己的子类对象。

多态的前提

- 必须是**类与类之间有关系**。要么**继承**，要么**实现**。
- 通常还有一个前提：存在**override**(方法重写)。

多态的好处

- 多态的出现大大的**提高程序的扩展性**。

多态的弊端

- 提高了扩展性，但是**只能使用父类的引用访问父类中的成员**。

多态的应用

转型

向上转型upcasting

```
    Animal a = new Cat();//类型提升。 向上转型。
```

向下转型downcasting

强制将父类的引用。转成子类类型。向下转型

```
    //如果想要调用猫的特有方法时，如何操作?
        Cat c = (Cat)a;
        c.catchMouse();
```

- 注意，**不要出现将父类对象转成子类类型这样的操作**
- 多态自始至终都是 **==子类==对象在做着变化**

instance of

instanceof : **用于判断对象的类型**。 使用方法：对象 intanceof 类型(类类型 接口类型)

异常处理机制

异常就是**程序在运行时出现不正常情况**。

异常的由来

程序遇到的问题也是现实生活中一个具体的事物，也可以**通过java的类的形式进行描述。并封装成对象。**
其实就是java对不正常情况进行描述后的对象体现。

对于问题的划分

一种是不可解决的问题，一种是可解决的问题。

- 对于不可解决的，java通过**Error类**进行描述。
  对于Error**一般不编写针对性的代码**对其进行处理。
- 对于可解决的，java通过**Exception类**进行描述。

对于Exception**可以使用针对性的处理方式**进行处理。

无论Error或者Exception都具有一些共性内容。
比如：不正常情况的信息，引发原因等。

异常的处理

如果不对异常做任何处理，虚拟机就会调用系统自动的异常处理机制，导致程序停止。

java 提供了特有的语句进行处理。

```
try
{
  需要被检测的代码;
}
catch（异常类 变量）
{
  处理异常的代码;（处理方式）
}
finally
{
  一定会执行的语句,例如socket释放连接;
}
```

当try和catch中有return时，finally仍然会执行；
finally是在return后面的表达式运算后执行的（**此时并没有返回运算后的值，而是先把要返回的值保存起来**，不管finally中的代码怎么样，返回的值都不会改变，依然是之前保存的值），所以**函数返回值是在finally执行前确定的；**

异常对象的常见方法操作

- **e.getMessage()：获取异常信息**
- **e.toString(): 异常名称 ： 异常信息**
- **e.printStackTrace(): 异常名称，异常信息，异常出现的位置。**
- **printStackTrace()异常名称，异常信息，异常出现的位置。其实JVM默认的异常处理机制，就是在调用printStackTrace方法。打印异常的堆栈的跟踪信息。**

throws语句

- 开发者在功能上通过throws的关键字声明了**该功能有可能会出现问题，** 让使用者来处理，否则编译失败
- **子类抛出的异常只能是父类抛出的异常或者它的子异常**

对多异常的处理

1. 声明异常时，建议声明**更为具体的异常**。这样处理的可以更具体。
2. **声明几个异常，就对应有几个catch块**。不要定义多余的catch块
   。
3. 如果多个catch块中的异常出现继承关系，**父类异常catch块放在最下面**。
4. 建立在进行catch处理时，catch中**要定义具体处理方式**。
   不要简单定义一句 e.printStackTrace(),也不要简单的就书写一条输出语句。

自定义异常

因为项目中会出现特有的问题，而这些问题并未被java所描述并封装对象。
所以对于这些特有的问题可以按照java的对问题封装的思想**将特有的问题。进行自定义的异常封装。**

自定义异常的特点

- java所描述并封装的异常**可以自动抛出，也可以手动抛出**
- 自定义异常不能被java所识别，**必须封装成对象并用throw手动抛出**
- 当在函数内部出现了throw所抛出的异常对象，那么就必须要采取相应的处理。

1. 要么在内部try catch处理。
2. 要么在函数上声明让调用者处理(也是通过try catch)。

- 一般情况在，函数内出现异常，函数上需要声明

throws和throw的区别

- **throws使用在函数头部。**
- **throw使用在函数内**。
- throws后面跟的**异常类**。可以跟多个。用逗号隔开。
- throw后跟的是**异常对象**。

RuntimeException子类

- Exceptoin中有一个特殊的子类异常RuntimeException **运行时异常**。
- 如果在函数内容抛出该异常，**函数上可以不用声明**，编译一样通过。
- 如果在函数上声明了该异常。**调用者可以不用进行处理**。编译一样通过

之所以不用在函数声明，是因为**不需要让调用者处理**。
当该异常发生，**希望程序停止**。因为在运行时，出现了无法继续运算的情况，希望停止程序后，**对代码进行修正**。

```
class Person
{
  public void checkName(String name)
  {

    //if(name.equals("lisi"))//会发生NullPointerException
    if("lisi".equals(name))//if(name!=null && name.equals("lisi"))
      System.out.println("YES");
    else
      System.out.println("no");
  }
}

main()
{
  Person p = new Person();
  p.checkName(null/ 字符串类型可以理解为一个类，所以可以接收"null"这个值
}
```

在上面这个例子中，传入空后程序员希望程序停止(因为名字为空是有问题的，**必须让程序停掉，修正代码**)如果**捕获这个异常相当于把这个问题隐藏**

自定义异常时：**如果该异常的发生使得程序无法继续运行**，
就让自定义异常继承RuntimeException。

异常的分类

1. **检查型异常（Checked Exception）**

   所谓检查（**Checked**）是指**编译器**要检查这类异常，检查的目的一方面是因为该类异常的发生难以避免，另一方面就是让开发者去解决掉这类异常，所以称为**必须处理（try ...catch或throws）的异常**。如果不处理这类异常，集成开发环境中的**编译器一般会给出错误提示。**编译时**被检测**的异常。(是可处理的，必须要用throws标识，让调用者处理(try 或继续抛出)，例如计时器超时)
   例如：一个读取文件的方法代码逻辑没有错误，但程序运行时可能会因为文件找不到而抛出FileNotFoundException，如果不处理这些异常，程序将来肯定会出错。所以编译器会提示你要去捕获并处理这种可能发生的异常，不处理就不能通过编译。

2. **非检查型异常（Unchecked Exception）**

   所谓非检查（**Unchecked**）是指**编译器**不会检查这类异常，不检查的则开发者在代码的编辑编译阶段就**不是必须处理**，这类异常**一般可以避免**，因此无需处理（try ...catch）。如果不处理这类异常，集成开发环境中的编译器也不会给出错误提示。

例如： 你的程序逻辑本身有问题，比如数组越界、访问null对象，这种错误你自己是可以避免的。编译器不会强制你检查这种异常。