Java 补充知识

1.Vector 的子类

1.1.Stack--栈

1.1.1.Stack

```
import java.util.Stack;
public class Games {
    public static void main(String args[]) {
        Stack <String>stk= new Stack<String>();
        System.out.println("创建的栈--"+stk);

        stk.push("一号请求入栈");//压入栈中

        stk.add("二号请求入列");

        stk.push("三号请求入列");

        System.out.println("原来"+stk);
        String str=stk.peek();
        System.out.println("返回第一个但不删除--"+str);
```

2. Map 的子类

2.1. Properties--属性

```java
import java.util.Iterator;
import java.util.Map.Entry;
import java.util.Properties;
import java.util.Set;
public class Games {
    public static void main(String args[]) {

            Properties  capital =new Properties();//新建属性势力类型为 Map 类型

        capital.put("中国", "北京");

        capital.put("韩国", "首尔");

        capital.put("加拿大", "温哥华");

        capital.put("芬兰", "斯德哥尔摩");

        capital.put("俄罗斯", "莫斯科");

        Set<Entry<Object,Object>>set = capital.entrySet();//取出对值

        Iterator<Entry<Object,Object>> it=set.iterator();
        while(it.hasNext()) {
            Entry<Object,Object>en=(Entry<Object,Object>)it.next();
```

3.字符流和字节流的转换（插入内容）

```java
import java.io.*;
public class Writ {
    public static void main(String[] args) throws Exception {
        BufferedReader bReader = new BufferedReader(new InputStreamReader(System.in));
        /**

         * 参数为 reader 类，但是用于输入的 system.in 为 字节输入流， 由于要将要输入的内容转换为

字符，所以要包装成字节转字符的实例

         * */
        String str = null;

        System.out.println("请输入一个数--");

        str = bReader.readLine();

        System.out.println("你输入的数是---" + str);
```

4，进制的表示

同一个自然数，用不同的进制表示的话，结果可能是不一样的。例如，数字10，如果是二进制，表示数字2；如果是八进制，表示数字8；如果是十进制，表示数字10；如果是十六进制，表示数字16。因此，不同的进制，需要有不同的标识，来区分不同的进制。

二进制：以 `0b` 作为开头，表示一个二进制的数字，例如：0b10、0b1001...

八进制：以 `0` 作为开头，表示一个八进制的数字，例如：010、027...

十进制：没有以任何其他的内容作为开头，表示一个十进制的数字，例如：123、29...

十六进制：以 `0x` 作为开头，表示一个十六进制的数字，例如：0x1001、0x8FC3...

4.1 二进制　以 0b 开头　表示　如 0b1001

4.2 八进制　0 开头　表示　如 010　024

4.3 十进制　没有内容为开头　就像我们平常用一样

4.4 十六进制 以 0x 作为开头，　如 0x1001

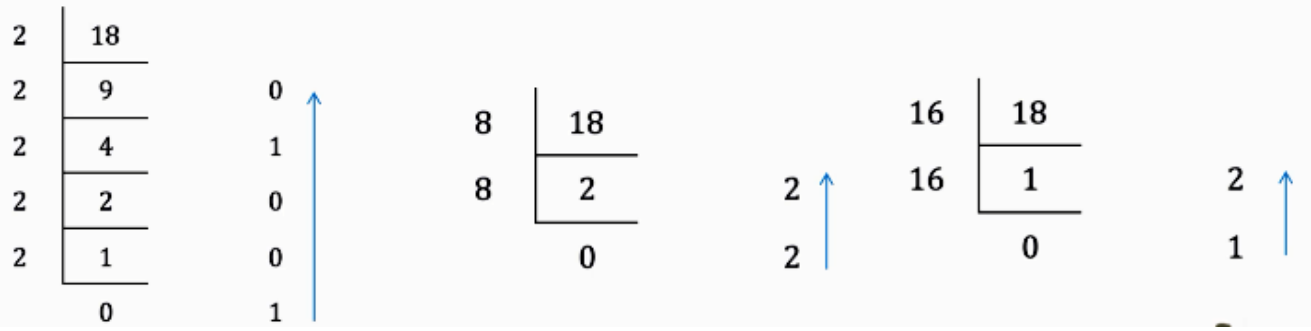
4.5 十六进制 超过 10，则用 a,b,c,d....来表示，不区分大小写

以上不区分大小写

3.4. 进制的转换

3.4.1. 十进制转其他进制

短转相除法：用数字除进制，再用商除进制，一直到商为零结束，最后将每一步得到的余数倒着连接以来，就是这个数字的指定的进制表示形式。



在计算机中，所有的数据存储都是以二进制的形式存储的。文字、图片、视频...，在计算机中都是二进制。那么，在计算机的存储系统中，每一个文件都有大小。那么文件的大小是如何计算的？

每一个二进制位称为一个 **比特位 (bit)**

8个比特位称为一个 **字节 (byte)**

从字节开始，每1024个单位向上增1。

> 8bit = 1byte
>
> 1024byte = 1kb
>
> 1024kb = 1mb
>
> 1024mb = 1GB
>
> 1024Gb = 1Tb
>
> 1024Tb = 1Pb
>
> 1024Pb = 1Eb
>
> 1024Eb = 1Zb
>
> ...
>
> ZB, BB, NB, DB .

每一个二进制位成为一个比特位，每八个比特位占一个字节；

十进制的 8 表示为二进制为 0000 1000

-8 怎么表示呢？二进制最高位不表示大小，只是表示着正负，所以-8 二进制就表示为 1000 1000

计算机中是只进行加法的，不进行直接的减法的，

例如

十进制的 10 + 8

0000 1010 + 0000 1000 =0001 0010 ————————》18

10 + （-8）

为了规避在计算过程中，符号位的参与运算，导致计算结果出错。人们引入了补码，规避了这个问题。在计算机中，所有的数据存储和运算，都是以 **补码** 的形式进行的。

- **原码**：一个数字的二进制表示形式，前面的计算二进制表示形式，得到的就是原码。
- **反码**：正数的反码与原码相同；负数的反码是原码符号位不变，其他位按位取反。
- **补码**：正数的补码与原码相同；负数的补码是反码 + 1 。

> 8，因为是正数，原反补都是 0000 1000
>
> -8[原] = 1000 1000
>
> -8[反] = 1111 0111
>
> -8[补] = 1111 1000

计算机中所有的运算都是以补码的形式进行的

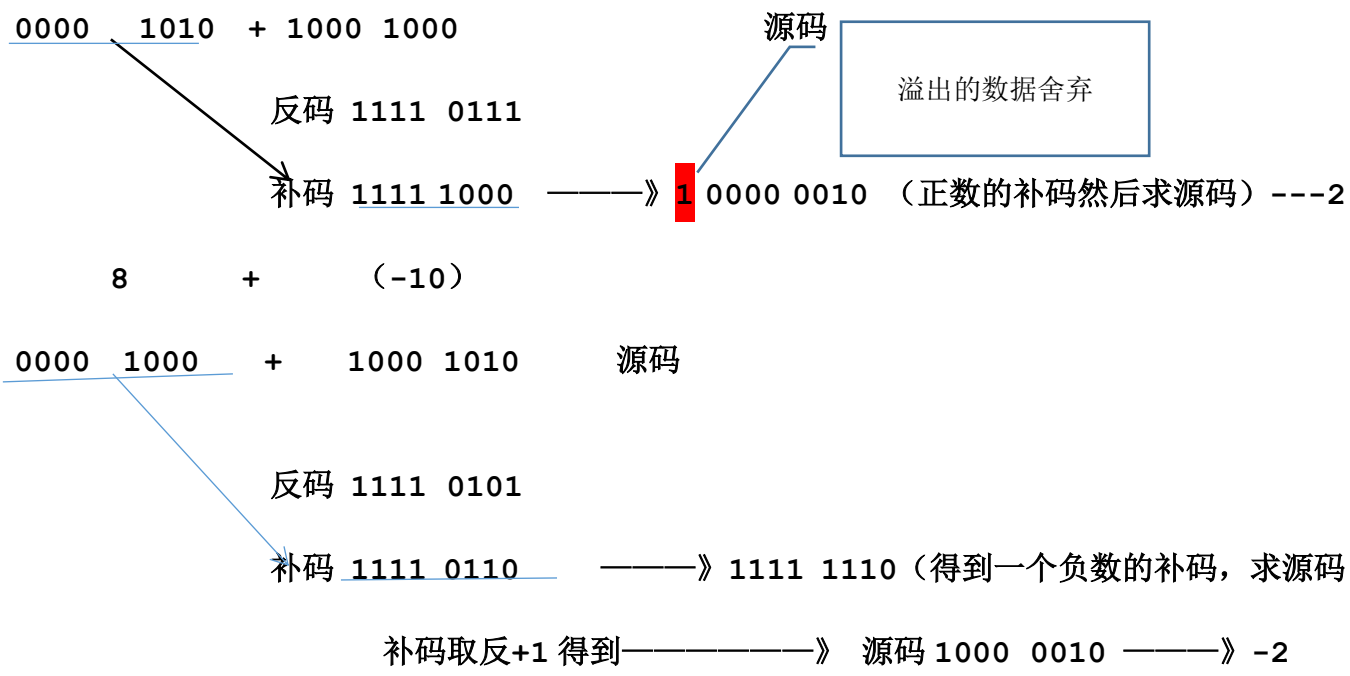
正数的反码与补码相同

负数的反码是源码符号不变，其他位置按为取反

补码：正数的补码与原码相同，负数的补码是反码+1

负数的源码是其符号位不变，补码按为取反+1

所以 10+（-8）



5，可变参数

在进行方法参数定义的时候有了一些变化（参数类型...变量），而这个时候的参数可以说就是数组形式，即：在可变参数之中，虽然定义的形式不是数组，但却是按照数组方式进行操作的。

需要注意的是，如果此时方法中需要接收普通参数和可变参数，则可变参数一定要定义在最后，并且一个方法

只允许定义一个可变参数。

范例：混合参数

在本方法调用时，前面的两个参数必须传递，而可变参数就可以根据需求传递。

```
class ArrayUtil {
    public static void sum(String name, String url, int... data) {
        ...
    }
}
```

```
class ArrayUtil {
    public static void sum(String name, String url, int... data) {
        ...
    }
}
```

```
package homewokk;
class array{
   public static int sum(int ...data) {
      int sum = 0;     for(int temp:data) {
         sum+=temp;     }
      return sum;   }}
public class ArrayCopys {
   public static void main(String[] args) {
      System.out.println(array.sum(1,2,3));
      System.out.println(array.sum(new int [] {1,2,3}));
      }     }
```

```
class Car {                                                    // 描述汽车
    private String name;                                       // 汽车名称
    private double price;                                      // 汽车价值
    private Person person;                                     // 车应该属于一个人
    public Car(String name, double price) {                    // 构造传入汽车信息
        this.name = name;
        this.price = price;
    }
    public void setPerson(Person person) {                     // 配置汽车与人的关系
        this.person = person;
    }
    public Person getPerson() {                                // 获取汽车拥有人信息
        return this.person;
    }
    public String getInfo() {                                  // 获取汽车信息
        return "汽车品牌型号：" + this.name + "、汽车价值：" + this.price;
    }
    // setter、getter略
}
class Person {                                                 // 描述人
    private String name;                                       // 人的姓名
    private int age;                                           // 人的年龄
    private Car car;                                           // 一个人有一辆车，没车为null
```

```
    private Person children[];                                          // 一个人有多个孩子
    public Person(String name, int age) {                               // 构造传入人的信息
        this.name = name;
        this.age = age;
    }
    public void setCar(Car car) {                                       // 设置人与汽车的关系
        this.car = car;
    }
    public void setChildren(Person children[]) {                        // 设置人与孩子关联
        this.children = children;
    }
    public Person[] getChildren() {                                     // 获取人的孩子信息
        return this.children;
    }
    public Car getCar() {                                               // 获取人对应的汽车信息
        return this.car;
    }
    public String getInfo() {                                           // 获取人员信息
        return "姓名：" + this.name + "、年龄：" + this.age;
    }
    // setter、getter略
}
public class JavaDemo {
    public static void main(String args[]) {
        // 第一步：声明对象并且设置彼此的关系
        Person person = new Person("林希勒", 29);                       // 实例化Person类对象
        Person childA = new Person("吴小伟", 18);                       // 孩子（人）对象
        Person childB = new Person("郭小任", 19);                       // 孩子（人）对象
        childA.setCar(new Car("BMW X1", 300000.00));                    // 匿名对象
        childB.setCar(new Car("法拉利", 126789.00));                    // 匿名对象
        person.setChildren(new Person[] { childA, childB });           // 一个人有多个孩子
        Car car = new Car("奔驰G50", 1588800.00);                       // 实例化Car类对象
        person.setCar(car);                                             // 一个人有一辆车
        car.setPerson(person);                                          // 一辆车属于一个人
        // 第二步：根据关系获取数据
        System.out.println(person.getCar().getInfo());                  // 通过人获取汽车的信息
        System.out.println(car.getPerson().getInfo());                  // 通过汽车获取拥有人的信息
        for (int x = 0; x < person.getChildren().length; x++) {         // 获取孩子信息
            System.out.println("\t|- " + person.getChildren()[x].getInfo());
            System.out.println("\t\t|- " + person.getChildren()[x].getCar().getInfo());
        }
    }
}
```

程序执行结果：

```
汽车品牌型号：奔驰G50、汽车价值：1588800.00
姓名：林希勒、年龄：29
    |- 姓名：吴小伟、年龄：18
        |- 汽车品牌型号：BMW X1、汽车价值：300000.00
```

6，合成设计模式

```
class 计算机 {                                  // 【父结构】描述计算机组成
    private 显示器 对象数组 [] ;                 // 一台计算机可以连接多台显示器
    private 主机 对象 ;                          // 一台计算机只允许有一台主机
}
class 显示器 {}                                 // 【子结构】显示器是一个独立类
class 主机 {                                    // 【子结构】定义主机类
    private 主板 对象 ;                          // 主机有一块主板
    private 鼠标 对象 ;                          // 主机上插一个鼠标
    private 键盘 对象 ;                          // 主机上插一个键盘
}
class 主板 {                                    // 【子结构】定义主板类，实际上也属于一个父结构
    private 内存 对象数组 [] ;                   // 主板上可以追加多条内存
    private CPU 对象数组 [] ;                    // 主板上可以有多块CPU
    private 显卡 对象 ;                          // 主板上插有一块显卡
    private 硬盘 对象数组 [] ;                   // 主板上插有多块硬盘
}
class 键盘 {}                                   // 【子结构】键盘类
class 鼠标 {}                                   // 【子结构】鼠标类
class 内存 {}                                   // 【子结构】内存类
class CPU {}                                    // 【子结构】CPU类
class 显卡 {}                                   // 【子结构】显卡类
class 硬盘 {}                                   // 【子结构】硬盘类
```

7，字符串

7.1 静态常量池

是指程序（*.class）在加载的时候会自动将此程序中保存的字符串、普通的常量、类和方法等信息，全部进行分配。

```
public class StringDemo {
    public static void main(String args[]) {
        String strA = "www.yootk.com";                  // 开辟新对象并入池
        // 使用“+”进行字符串连接，由于所有的内容都是常量，本质上表示一个字符串
        String strB = "www." + "yootk" + ".com";        // 直接赋值
        System.out.println(strA == strB);               // 判断结果：true
    }
}
程序执行结果：
true
```

7.2 运行时常量池：

当一个程序（*.class）加载之后，有一些字符串内容是通过 String 对象的形式保存后再实现字符串连接处理，由于 String 对象的内容可以改变，所以此时称为运行时常量池。

```
public class StringDemo {
    public static void main(String args[]) {
        String logo = "yootk" ;                          // 定义一个变量
        String strA = "www.yootk.com";                   // 开辟新对象并入池
        // 使用“+”进行字符串连接，由于所有的内容都是常量，本质上表示一个字符串
        String strB = "www." + logo + ".com";            // 动态拼凑，logo为变量
        System.out.println(strA == strB);                // 判断结果：false
    }
}
程序执行结果：
false
```

8，泛型

通配符“？”除了可以匹配任意的泛型类型外，也可以通过泛型上限和下限的配置实现更加严格的类范围定义。

【类和方法】设置泛型的上限（？extends 类）：只能够使用当前类或当前类的子类设置泛型类型。

？extends Number：可以设置 Number 或 Number 子类（例如，Integer、Double）。

【方法】设置泛型的下限（？super 类）：只能够设置指定的类或指定类的父类。

？super String：只能够设置 String 或 String 的父类 Object。

```
class Point<T>{//泛型
    private T x;
    public void  setX(T x){
        this.x=x;    }
    public  T getX(){
        return this.x;    }}
public class Fanxing {
    public static void main(String[] args) {
        Point <Integer> point = new Point<Integer>();
      point.setX(100000);
      getInfo(point);          }
```

不在 getInfo（）方法上设置泛型类型，实际上可以解决当前不同泛型类型的对象传递问题，但同时也会有新的问题产生：允许随意修改数据。

```
class Point<T>{//泛型
    private T x;
    public void  setX(T x){
        this.x=x;
    }
    public  T getX(){
        return this.x;    }
}
public class Fanxing {
    public static void main(String[] args) {
        Point <Integer> point = new Point<Integer>();
      point.setX(100000);
      getInfo(point);          }
    public static void getInfo(Point <Integer>temp){
        temp.setX(2000);
        System.out.println(temp.getX());    }
```

8.1 泛型接口

两种实现方式：在子类中继续声明泛型和子类中为父类设置泛型类型。

```
 interface Jiekou <T> {//泛型接口
      public String echo(T msg);
}
class message <T> implements Jiekou<T>{

      @Override
      public String echo(T t) {//在方法覆写的时候定义泛型类型
           return "t"+t;
      }
}
public class Ceshi {
```

子类时继续声明了一个泛型标记 T，并且实例化 message 子类对象时设置的泛型类型也会传递到接口中。

```
interface    Usb <S>{
        public String Shibie(S s);

}
class Com<S> implements Usb<String>{//实现的时候指定泛型类型


        @Override
        public String Shibie(String s) {
                return "---helloworld";
        }
}

public class Haha {
        public static void main(String[] args) {
                Com <String>cc= new Com();
                cc.Shibie("ss");
                System.out.println( cc.Shibie("ss"));
        }
}
```

```
interface    Usb <S>{
        public String Shibie(S s);

}
class Com implements Usb<String>{//实现的时候指定泛型类型


        @Override
        public String Shibie(String s) {
                return "---helloworld";
        }
}

public class Haha {
```

```
interface    Usb <S>{
        public String Shibie(S s);

}
class Com<S> implements Usb<String>{//实现的时候指定泛型类型


        @Override
        public String Shibie(String s) {
                return "---helloworld";
        }
}

public class Haha {
        public static void main(String[] args) {
                Com cc= new Com();
                cc.Shibie("ss");
                System.out.println( cc.Shibie("ss"));
        }
}
```

子类时没有定义泛型标记，而是为父接口设置泛型类型为 **`String`**，所以在覆写 **`shibie`**（）方法时参数的类型为 **`String`**

8.2 泛型方法

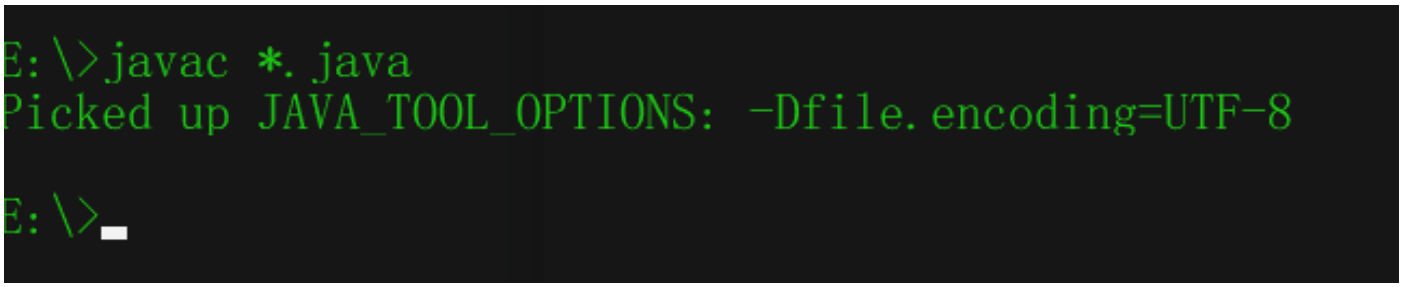
```
public class Haha {
        public static void main(String[] args) {
                String num = print("1");
                        System.out.println(num);
        }
        public static <T> T print(T args) {
```

```
public class Haha {
	public static void main(String[] args) {
		String num[] = print("1", "2", "3");
		for (String str : num) {
			System.out.println(str);
		}
	}

	public static <T> T[] print(T... args) {
		return args;
	}

}
```

可以借助 **JVM** 进行快速编译所有 **java** 文件，不过会在有错误的时候不会继续向下编译



9，枚举

在枚举类中除了可以定义若干个实例化对象之外，也可以像普通类那样定义成员属性、构造方法、普通方法，但是需要记住的是，枚举的本质上是属于多例设计模式，所以构造方法不允许使用 **public** 进行定义。如果类中没有提供无参构造方法，则必须在定义每一个枚举对象时明确传入参数内容。

```
enum Color {
    红色("RED"), 黄色("YELLOW"),绿色("GREEN");
    private String title;
    private Color(String title){
        this.title=title;
    }
    @Override
    public String toString(){
        return this.title;
    }
}

public class Test1 {
    public static void main(String[] args) {
        for (Color color : Color.values()) {
            System.out.println(color.ordinal()+"--"+color.name()+"--"+color);        }    }}
```

```
package com.company;
interface Colorful{
    public String getColor();}

enum Colors implements Colorful{
    RED("红色"),YELLOW("黄色"),GREEN("绿色");
private String title;
    private Colors(String title){
        this.title=title;      }
  public String toString(){
        return this.title;   }
    @Override
```

泛型可以指定多个

```
package com.company;
class Mai<T ,S extends Number>{//定义一个泛型类,为 Number 及其子类
    private T x;
    private S y;
    public Mai(T x,S y){
        this.x=x;
        this.y =y;      }
    public T getX() {
        return x;      }
    public void setX(T x) {
        this.x = x;      }
    public S getY() {
        return y;      }
    public void setY(S y) {
        this.y = y;      }
    public String toString(){
        return "坐标是-["+this.x+" "+this.y+"]";
    }}
public class fanxingdiandian   {
    public static void main(String[] args) {
        Mai   <Integer,Double>   mai = new Mai<Integer,Double>(1,2.5);
        System.out.println(mai);
    }}
```

10，异常处理

为什么一定要将异常交给调用处处理呢？

在整体设计中 **div**（）方法自己来进行异常的处理不是更方便吗？为什么此时必须强调将产生的异常继续抛给调用处来处理？

回答：将此程序中的开始提示信息和结束提示信息想象为资源的打开与关闭。

为了解释这个问题，首先来研究两个现实中的场景。

场景 1（异常抛给调用处执行的意义）：你所在的公司要求你进行一些高难度的工作，在完成这些工作的过程中你突然不幸摔伤，那么请问，这种异常的状态是由公司来负责还是你个人来负责？如果要是由公司来负责，那么就必须将你的问题抛给公司来解决。

场景 2（资源的打开与关闭）：现在假设你需要打开自来水管进行洗手的操作，在洗手的过程中可能会发生一些临时的小问题导致洗手暂时中断问题，然而不管最终是否成功处理了这些问题，总要有人来关闭自来水管。

理解了以上两个生活场景之后，再回到程序设计的角度来讲，例如，在现实的项目开发中都是基于数据库实现的，于是这种情况下往往需要有以下 3 个核心步骤。

步骤 1：打开数据库的连接（等价于"【START】..."代码）。

步骤 2：进行数据库操作，如果操作出现问题则应该交由调用处进行异常的处理，所以需要将异常进行抛出。

步骤 3：关闭数据库的连接（等价于"【END】..."代码）。

以上所采用的是标准的处理结构，实质上这种结构里面，在有需要的前提下，每一个 **catch** 语句里除了简单地抛出异常对象之外，也可以进行一些简单的异常处理。但是如果说此时的代码确定不再需要本程序做任何的异常处理，也可以直接使用 **try** … **finally** 结构捕获执行 **finally** 代码后直接抛出。

```
class Shuxue {
    int x = 10;
    int result;

    public int Divdid(int temp) throws Exception {//抛出异常

        try{
```

```
class Shuxue01 {
    int x = 10;
    //int result;

    public int DivdidED(int temp) throws Exception {//抛
```

以上代码中如果在方法中出现 try，那么在 main 方法中（如果 main 方法不在把此异常抛出-如果抛出则就交给 main 得上一级--jvm 去处理，则程序就会中断运行）就需要就要进行处理（因为最终是在 main 方法中调用--方法调用处）；

11，Assert 关键字

其主要的功能是断言，指程序运行到某处后，其结果是预期结果；

默认是不开启 assert，其主要有两种用法

Assert boolean 表达式；

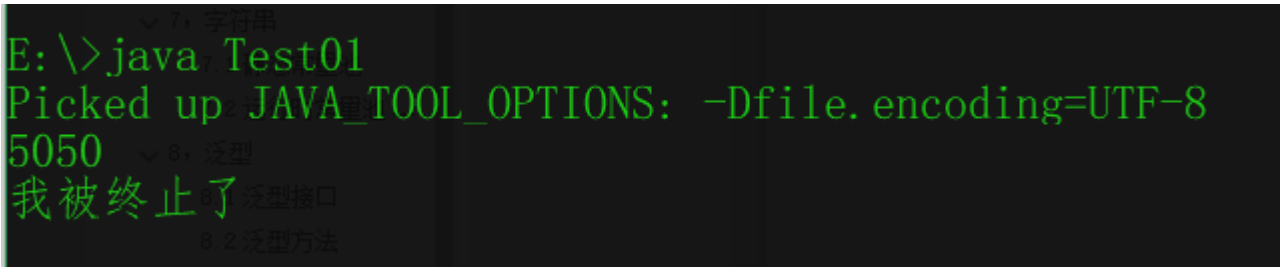
如果 boolean 为 true 则程序继续执行，否则抛出 AssertionException，并程序终止运行；

Assert boolean 表达式:错误信息表达式；

如果如果 boolean 为 true 则程序继续执行，否则抛出 AssertionException，输出错误表达式信息；

```
class Sum{
    int result =0;
    public void ms(int x){
        for(int i =0;i<101;i++){
            result +=i;
        }
        assert result ==5049;
        System.out.println(result);
```

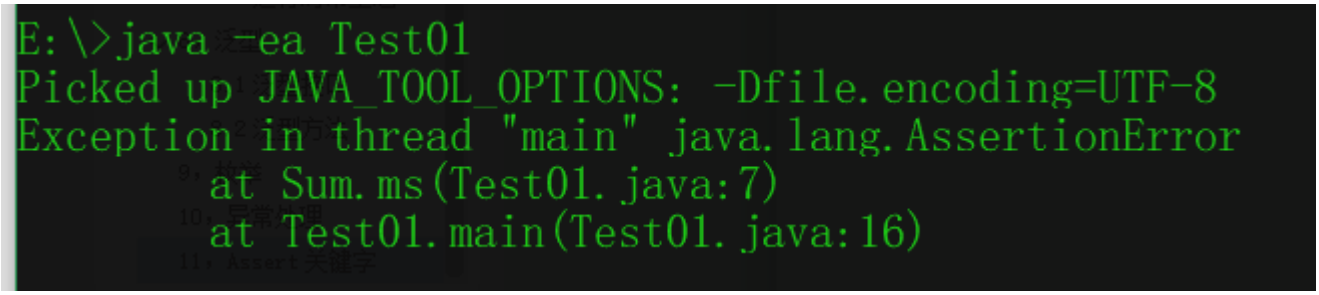
默认是不会执行 **assert** 的




```
class Sum{
    int result =0;
    public void ms(int x){
        for(int i =0;i<101;i++){
            result +=i;        }
        assert result ==5049;
        System.out.println(result);    }}
public class Test01 {
    public static void main(String[] args) {
Sum sum= new Sum();
sum.ms(100);

System.out.println("我被终止了");    }}
```

此时因为执行结果与预测结果不符，抛出 **AssertException**，程序终止运行；



```
class Sum{
    int result =0;
    public void ms(int x){
        for(int i =0;i<101;i++){
            result +=i;         }

        assert result ==5049:"程序异常";

        System.out.println(result);     }}
public class Test01 {
    public static void main(String[] args) {
Sum sum= new Sum();
sum.ms(100);
```

```
E:\>java -ea Test01
Picked up JAVA_TOOL_OPTIONS: -Dfile.encoding=UTF-8
Exception in thread "main" java.lang.AssertionError: 程序异常
        at Sum.ms(Test01.java:7)
        at Test01.main(Test01.java:16)
```

12. 内部类

```
class Outer{

    private String str ="加油";

    public void print(){
        Inner in = new Inner();
        in.printed();       }
    class Inner{
        public void printed(){
            System.out.println(str); }    }}
public class Niming {
    public static void main(String[] args) {
        Outer out = new Outer();
```

```
class Outer {

    private String str = "加油";

    public void print() {
        Inner in = new Inner();
        in.printed();       }
    public String getStr() {
```

```
class Outer {
    private String str = "加油";
    public void print() {
        Inner in = new Inner(this);
        in.printed();       }
    public String getStr() {
        return str;       }}
class Inner {
    private Outer out;
    public Inner(Outer out){
        this.out=out;       }
    public void printed() {
        System.out.println(this.out.getStr());       }}
    public class Niming {
        public static void main(String[] args) {
            Outer out = new Outer();
```

在内部类的结构中，不仅内部类可以方便地访问外部类的私有成员，外部类也同样可以访问内部类的私有成员。内部类本身是一个独立的结构，这样在进行普通成员属性访问时，为了明确地标记出

```
class Outer {
    private String str = "加油";
    public void print() {
        Inner in = new Inner();
        in.printed();
        System.out.println(in.str1);
      //   System.out.println(Inner.this.str1);   //因为 Outer 不是一个内部类，所以在此不也能用此种方式
        }
    class Inner {
        private String str1= "奥利给";
        public void printed() {
            System.out.println("形式 1-"+str);
            System.out.println("形式 2-"+Outer.this.str);//
```

12.1 内部类的实例化

```
class Outer {
    private String str = "加油";
    public void print() {
        Inner in = new Inner();              in.printed();
        System.out.println(in.str1);                    }
    class Inner {
        private String str1= "奥利给";
        public void fun(){
            System.out.println("我被实例化了");          }
        public void printed() {
            System.out.println("形式 1-"+str);

            System.out.println("形式 2-"+Outer.this.str);}    }}
public class Niming {
    public static void main(String[] args) {
        //内部类实例化方式一
        Outer out = new Outer();
```

内部类虽然被外部类所包裹，但是其本身也属于一个完整类，所以也可以直接进行内部类对象的实例化，此时可以采用以下的语法格式。

外部类.内部类 内部类对象=new 外部类（）.new 内部类（）；

在此语法格式中，要求必须先获取相应的外部类实例化对象后，才可以利用外部类的实例化对象进行内部类对象实例化操作。

进行内部类源代码编译后，读者会发现有一个

内部类私有化的意义在于这时候内部类只能被该外部类使用，

```
class Outer {
    private String str = "加油";
public void print(){
    new Outer().getInstance().fun();}
    private class Inner {
        private String str1 = "奥利给";

        public void fun() {
            System.out.println("我被实例化了");          }
        public void printed() {
            System.out.println("形式 1-" + str);

            System.out.println("形式 2-" + Outer.this.str);      }    }
    public Inner getInstance() {
```

此时 **Inner** 类使用了 **private** 定义，表示此类只允许被 **Outer** 一个类中使用。另外，需要提醒读者的是，**private**、**protected** 定义类的结构只允许出现在内部类声明处。

内部类不仅可以在类中定义，也可以应用在接口和抽象类之中，即可以定义内部的接口或内部抽象类

```
interface Outered<T> {
    public String sendDemo(T t);
    interface Innered {
        public String Received();       }}

class OuteredDemo implements Outered {

    private String str = "阳光明媚";

    @Override
    public String sendDemo(Object o) {
        return o.toString();       }
    class Innered implements Outered.Innered {
        @Override
        public String Received() {
            return new OuteredDemo().sendDemo(str); }    }}
public class jiekounei {
    public static void main(String[] args) {
        Outered.Innered innered = new OuteredDemo().new Innered();
        String str1 = innered.Received();
        System.out.println(str1);       }}
```

可以在接口中定义抽象类

```
interface Hello<T>{
    public String sendMessage(T t);

    abstract class World{//接口中的抽象类

        public abstract String receiveMessage();       }}
class Ni implements Hello{

private String str = "好想爱这个世界呀！！！";

    @Override
    public String sendMessage(Object obj) {
        return obj.toString();       }
    class Hao extends World{
        @Override

        public String receiveMessage() {//内部类实现抽象类--也可以不去实现
```

当然也可以在接口中定义静态方法去返回接口的子类实例，那么此时代码就是

```
interface Hello {
    public String sendMessage(String t);
    abstract class World {//接口中的抽象类
        public abstract String receiveMessage();    }
    public  static  Hello getInstance(){//返回接口的子类实例
        return new Ni();    }}

class Ni implements Hello {
    @Override
    public String sendMessage(String obj) {
        return obj;    }
    class Hao extends World {
        @Override
        public String receiveMessage() {//内部类实现抽象类--也可以不去实现

            String str0=Hello.getInstance().sendMessage("好想爱这个世界呀");
            return str0 ;            }    }}
public class Haha {
    public static void main(String[] args) {
      Hello hello= Hello.getInstance() ;//得到子类实例
      String str =(String) new Ni().new Hao().receiveMessage();
       System.out.println(new String (str));
```

在方法中定义内部类

```
class JavaDemo{
    private  String name ="张三的歌";
    public void sendName(String name1){
        class JavaD{
            public void getName(){
            System.out.println(JavaDemo.this.name);
                System.out.println(name1); }  }
        new JavaD().getName(); }}
public class NiHao {
    public static void main(String[] args) {
JavaDemo Java = new JavaDemo();
```

```
interface    Book{
      public void send(String str);
}

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

            //接口无法直接进行实例化，这个时候使用匿名内部类后就可以利用对象的实例化格式来获取接口

实力，

            Book book = new Book() {//要想实例化接口有两种方式，一种是通过子类进行是，实例化，另一种在

实例化的时候将接口进行实现

                  @Override
                  public void send(String str) {
                        System.out.println(str);
```

本程序利用匿名内部类的概念实现了 **IMessage** 接口的实例化处理操作，但是匿名内部类由于没有具体的名称，所以只能够使用一次，而使用它的优势在于：减少类的定义数量。

提示：关于匿名内部类的说明。

之所以强调匿名内部类可以减少一个类的定义，主要的原因在于：实际项目开发中一个***.java** 文件往往只会使用 **public class** 定义一个类，如果现在不使用匿名内部类，并且在 **IMessage** 接口子类只使用一次的情况下，那么就需要定义一个 **MessageImpl.java** 源代码文件，这样就会显得比较浪费了。

另外，匿名内部类大部分情况下都是结合接口、抽象类来使用的，因为这两种结构中都包含有抽象方法，普通类也可以使用匿名内部类进行方法覆写，但是这样做的意义不大。

如果现在结合接口中的 **static** 方法，也可以直接将匿名内部类定义在接口中，这样也可以方便地进行引用。

```
interface    Book{
      public void send(String str);
      public static Book getInstance(){
            return new Book() {
                  @Override
                  public void send(String str) {
```

本程序利用匿名内部类直接在接口中实现了自身，这样的操作形式适合于接口只有一个子类的时候，并且也可以对外部调用处隐藏子类。

13,Lambda 表达式

所谓的 **Lambda** 表达式，是指应用在 **SAM**（**Single Abstract Method**，含有一个抽象方法的接口）环境下的一种简化定义形式，用于解决匿名内部类的定义复杂问题，在 **Java** 中 **Lambda** 表达式的基本语法形式如下。

Lambda 表达式的基本语句形式：

定义方法体--（参数，参数，参数.....）->{方法体}

直接返回结果-- （参数，参数，参数.....）->语句

在给定的格式中，参数与要覆写的抽象方法的参数对应，抽象方法的具体操作就通过方法体来进行定义。

```
interface Imassage{
    public void send (String str,String str1);
}
public class Lamb {
    public static void main(String[] args) {
        Imassage msg =(str,str1)->{
            System.out.println("发送信息："+str+str1);
        };
```

进行 Lambda 表达式定义的过程中，如果要实现的方法体只有一行，则可以省略“{}”

```
interface Imassage{
    public void send (String str,String str1);
}
public class Lamb {
    public static void main(String[] args) {
        Imassage msg =(str,str1)->
            System.out.println("发送信息："+str+str1);

        msg.send("Hello","World");
    }
```

利用 Lambda 表达式可以进一步简化了匿名内部类的定义结构；

在 Lambda 表达式中已经明确要求 Lambda 是应用在接口上的一种操作，并且接口中只允许定义有一个抽象方法。但是在一个项目开发中往往会定义大量的接口，而为了分辨出 Lambda 表达式的使用接口，可以在接口上使用 @FunctionalInterface 注解声明，这样表示此为函数式接口，里面只允许定义一个抽象方法。

```
interface    Add{
    public int sum(int x,int y);}
public class SUN {
    public static void main(String[] args) {

        Add ad =(t1,t2)->{return t1+t2;};//原始格式

        Add ad1 = (t1,t2) ->t1+t2;//由于只是单行计算，所以可以直接编写语句
```

13，方法引用

在 Java 中利用对象的引用传递可以实现不同的对象名称操作同一块堆内存空间的操作，而从 JDK 1.8 开始，方法也支持引用操作，这样就相当于为方法定义了别名。方法引用的形式一共有以下 4 种。

引用静态方法：类名称：：static 方法名称。

引用某个对象的方法：实例化对象：：普通方法。

引用特定类型的方法：特定类：：普通方法。

引用构造方法：类名称：：new。

```
@FunctionalInterface
interface    Add<P , Q>{//泛型，参数为 P。返回值为 Q
    public Q Sun(P p);
}
public class SUN {
     public static void main(String[] args) {
          Add<Integer,String> as = String::valueOf;//引用静态方法              类名称::静态方法名称
          String str =as.Sun(100);
```

```
interface Booked <S>{
     public S Uppercase();
}
public class Upread {
     public static void main(String[] args) {
          Booked<String> booked ="Hello"::toUpperCase;       //对象调用普通方法
          System.out.println(booked.Uppercase());
     }
}
```

```
class Car{
     private String brand;
     private String color;
     public Car (String brand,String color){
          this.brand=brand;
          this.color=color;
     }
     public String toString (){
```

构造方法的引用在实际开发中可以实现类中构造方法的对外隐藏，更加彰显了面向对象的封装性。

13.方法的简介

6.1. 方法的简介

6.1.1. 方法的概念

在程序中，有些情况下，有些代码是需要被多次执行的。如果需要在每次使用到这些逻辑的时候，都去写代码实现，此时代码将会非常的冗余，不利于维护。

此时，我们可以将这些需要被多次执行的逻辑单独包装起来，在需要的时候直接调用即可。

方法，就是为了实现这个需求而存在的。

方法，其实就是一个实现特定功能的代码段，方法中的逻辑可以被重复使用。

6.1.2. 为什么要使用方法

可以使用方法，将需要多次执行的逻辑封装起来，哪里需要，哪里直接调用即可。

降低了代码的冗余，提高了代码的复用性与维护性。

6.2. 方法的定义与调用

6.2.1. 方法的定义

```
[访问权限修饰符] [其他的修饰符] 返回值类型 方法名字([参数列表]) {
    方法体
}
```

6.4.3. return关键字

- 表示方法的执行结果。

  return关键字后面跟上方法的执行结果，这个值的类型，必须和方法定义中的返回值类型一致。

```
static int add(int a, int b) {
    return a + b;        // 将 a + b 的运算结果，作为整个方法的执行结果。
}
```

- 表示结束方法的执行

  在一个返回值类型为void的方法中，也可以使用return关键字。此时，return后面什么都不要写。

  return写在方法中，表示结束方法的结束。

```
static int add(int a, int b) {
    return a + b;
    // return执行后，后面的所有的代码都不执行。
    // 方法结束。
}
```

6.6. 方法的递归

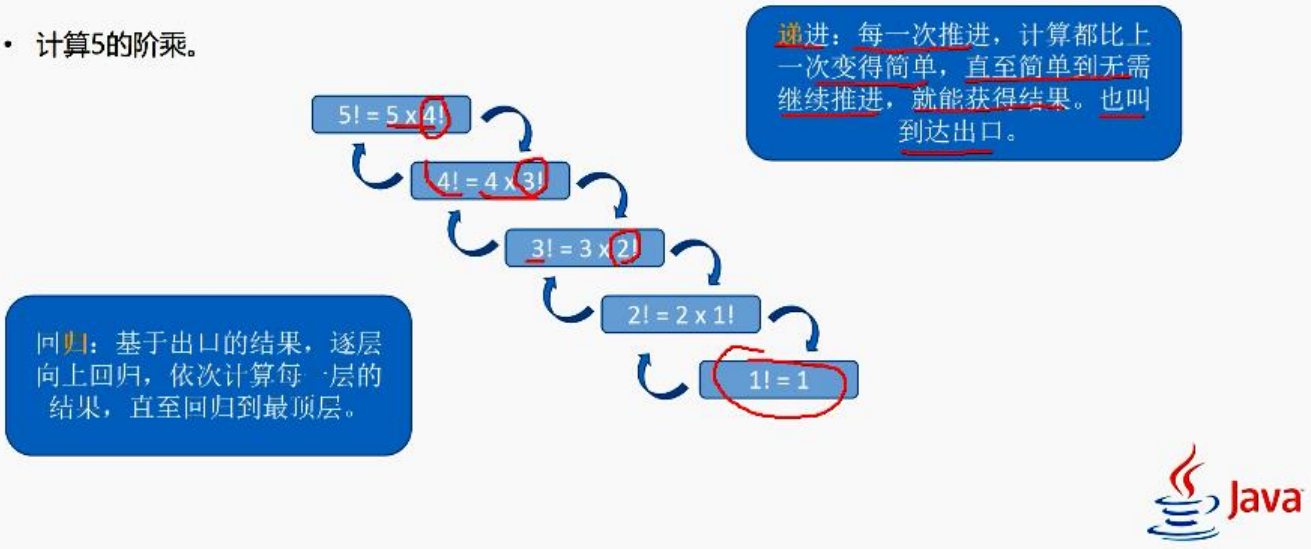
6.6.1. 递归是什么

递归，是一种程序设计的思想。在解决问题的时候，可以将问题拆分成若干个小问题。通过解决这些小的问题，逐渐解决这个大的问题。

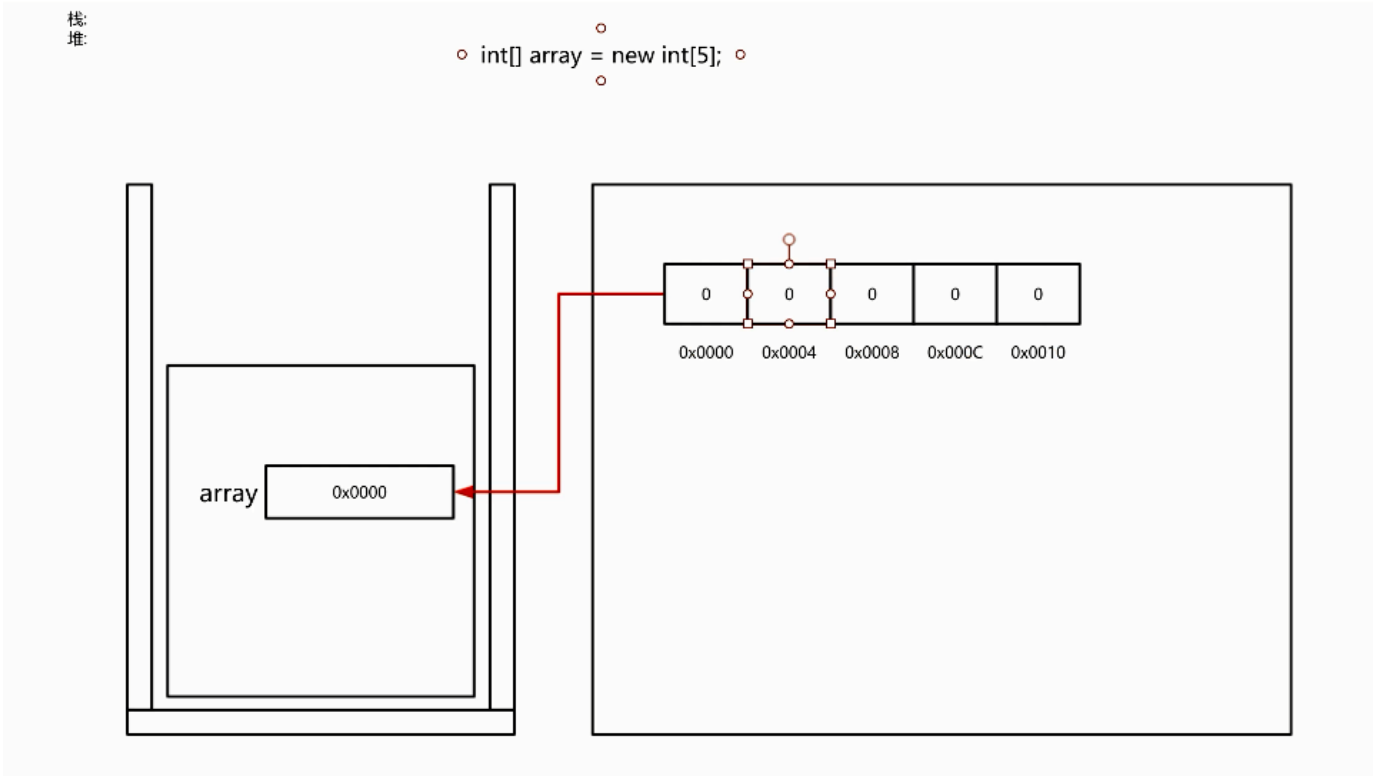
6.6.2. 何时使用递归

- 当需要解决的问题可以拆分成若干个小问题，大小问题的解决方法相同。

- 计算5的阶乘。




```
    数组中的数据默认值:
    整型: 0
    浮点型: 0.0
    字符型: '\u0000'
    布尔型: false
    引用数据类型
/
blic class Array1 {
  public static void main(String[] args) {
```


栈:
堆:
int[] array = new int[5];
0
0
0
0
0
0x0000
0x0004
0x0008
0x000C
0x0010
array
0x0000