

Dos 命令行运行

Cd 命令行指令 进入文件夹

JRE 与 JDK JRE 包含虚拟机

JRE(Java Runtime Environment Java运行环境) 包括Java虚拟机(JVM Java Virtual Machine)和Java程序所需的核心类库等, 如果想要运行一个开发好的Java程序, 计算机中只需要安装 JRE 即可。	JDK(Java Development Kit Java开发工具包) JDK是提供给Java开发人员使用的, 其中包含了java的开发工具, 也包括了 JRE 。所以安装了 JDK , 就不用在单独安装 JRE 了。 其中的开发工具: 编译工具(javac.exe) 打包工具(jar.exe)等
简单而言: 使用 JDK 开发完成的 java 程序, 交给 JRE 去运行。	

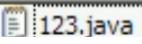
常用Dos命令

- d: 回车 盘符切换
- dir(directory):列出当前目录下的文件以及文件夹
- md (make directory) : 创建目录
- rd (remove directory): 删除目录
- cd (change directory)改变指定目录(进入指定目录)
- cd.. : 退回到上一级目录
- cd\ : 退回到根目录
- del (delete): 删除文件,删除一堆后缀名一样的文件*.txt
- exit : 退出dos命令行
- cls : (clear screen)清屏

安装目录中的 bin (binary 二进制) 文件夹里都是开发工具

Dos 命令行中环境变量临时配置方式 设置 java 编译文件默认路径

C:\>set path=D:\jdk1.6.0_24\bin C:\>javac 此时就会运行 javac

一个 java 文件是源文件  1 KB JAVA 文件

需要被 javac(编译工具)翻译成虚拟机能识别的文件

D:\java0217\day01> 输入 javac 123.java 就可以对 123.java 进行 javac 的编译
此时会在路径下生成 Demo.class 文件 这个是 java 运行文件

此时再用 java.exe 即可运行这个 demo 文件 D:\java0217\day01>java Demo

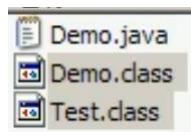
临时配置路径 在任意路径下直接运行 java 运行文件 设置虚拟机运行文件默认路径

C:\>set classpath=D:\java0217\day01

C:\>java Demo

hello .java set classpath=.;c:\;d:\;“.” 代表当前路径

Path 可执行文件 (.exe) java 文件 classpath (.java)



java 文件中每有一个类就会被 javac 编译生成一个.class 文件

进制

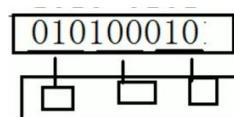
十进制: 0-9 , 满10进1.

八进制: 0-7 , 满8进1. 用0开头表示。

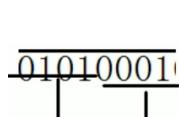
十六进制: 0-9, A-F, 满16进1. 用0x开头表示。 04 八进制 0x3b4a 十六进制

每个字节由八个二进制数组成 0010-0110 1111-1111 是 255

010..001— A 编码表
011..001— B ASCII



三位代表一位--八进制



四位代表一位--十六进制

都是为了更简洁的表示数据

负数就是其正数的二进制取反+1 结论:负数的二进制最高位都是1

```
System.out.println('a'+1); 结果 98 a 为 2 个八位 1 为 4 个八位 a 自动提升为四个八位进行运算  
结果为四个八位结果 98 (97+1) System.out.println((char)('a'+1)); 强转输出 b
```

Int a=3 int b b=a++ b3 a4 b=++a b4 a4

```
//面试题:看下面的程序是否有问题,如果有问题,请指出并说明理由。  
byte b1 = 3;  
byte b2 = 4;  
//byte b3 = b1 + b2;  
/*  
从两方面  
1,byte与byte(或short,char)进行运算的时候会提升为int,两个int类型相加的结果也是int类型  
2,b1和b2是两个变量,变量存储的值是变化,在编译的时候无法判断里面具体的值,相加有可能会超出byte的取值  
*/  
//System.out.println(b3);  
//byte b4= 3 + 4; //java编译器有常量优化机制  
byte b4 = 7;  
System.out.println(b4);
```

重载和返回值类型没关系

```
void show(int a,char b,double c){}  
  
a.  
void show(int x,char y,double z){} //没有,因为和原函数一样。  
  
b.  
int show(int a,double c,char b){} //重载,因为参数类型不同。注意:重载和返回值类型没关系。  
c.  
  
void show(int a,double c,char b){} //重载,因为参数类型不同。注意:重载和返回值类型没关系。  
  
d.  
boolean show(int c,char b){} //重载了,因为参数个数不同。  
  
e.  
void show(double c){} //重载了,因为参数个数不同。  
  
f.  
double show(int x,char y,double z){} //没有,|
```

内存结构

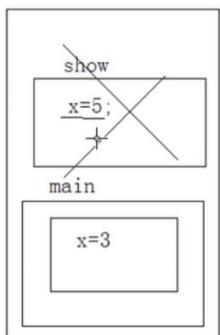
Java 在数据存储中划分了 5 部分空间: 栈内存 (stack) 堆内存 (heap) 方法区 本地方法区 寄存器

当我 int x=3 时 会在栈中开辟一个空间 main 函数方法中的 x=3 当在 show 函数中有 x=5 时 会开辟一个名字为 show 的空间 x=5 两个 x 不冲突 当 show 函数运行结束时 自动被销毁

栈内存的特点就是数据使用完毕,会自动释放

所有的**局部变量**(定义在方法中的变量, 定义在方法中参数的变量, for 循环中的变量)都在栈中

栈



凡是 new 出来的实体 (数组和对象) 都在堆里

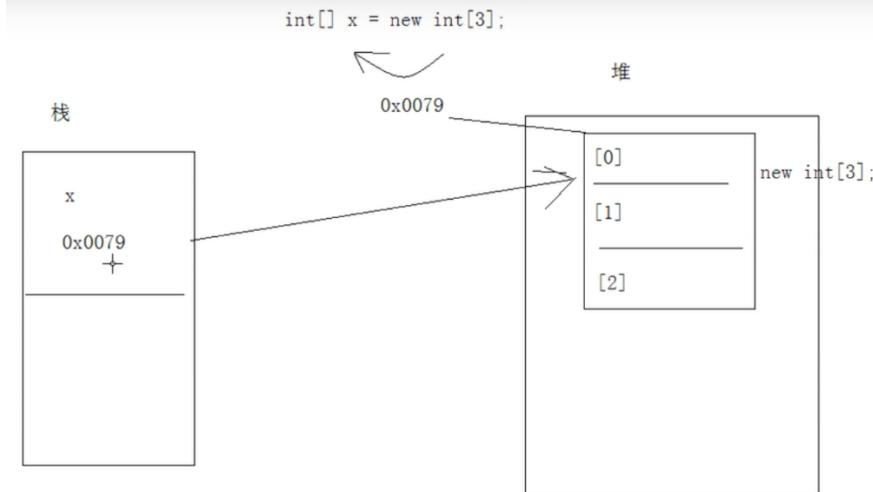
New int [3] 会在堆中开辟一个空间 储存数组 此数组包含 3 个格子，每个格子都有自己的编号

堆中的每一个实体都有一个存放位置 用地址表示数据存放位置 每个实体都有一个起始位置 称为首地址值

将此地址赋予栈中的 x x 就有值了 此时称为 x 引用了这个数组 x 只储存了数组的地址

若直接 System.out.println(x) 输出的是地址编号 需要通过 for 循环来输出数组里的值

地址: [I@de6ced [此为数组 | 此为 int de6ced 十六进制的哈希值



Int[] x=new int [3] 数组自动有值 0, 0, 0

string[]x=new string[2] x[0]是 null

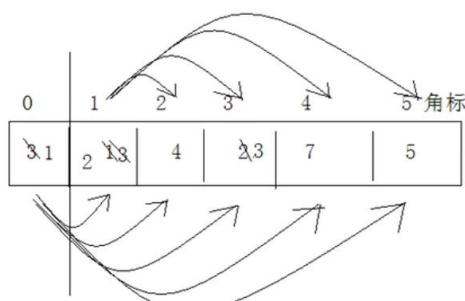
堆内存的实体是用于封装数据的 这些实体都有默认初始化值 默认值由元素类型而定

X=null ;

在堆内存中若一个实体无任何引用指向它时，会在不定时时间内，启动垃圾回收机制在堆内存中清除
栈是自动释放 而堆需要垃圾回收机制

排序

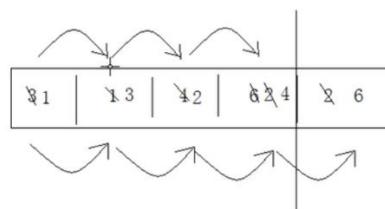
选择排序



对给定数组进行排序

```
public static void sort(int[]a) {  
    int c;  
    for(int b=0;b<a.length-1;b++) {  
        for(int d=b+1;d<a.length;d++) {  
            if(a[b]>a[d]) {  
                c=a[b];  
                a[b]=a[d];  
                a[d]=c;  
            }  
        }  
    }  
}
```

冒泡排序



相邻两个元素进行比较 符合条件换位

```
public static void sort(int[]a) {  
    for(int n=1;n<a.length;n++) {  
        for(int b=0;b<a.length-n;b++) {  
            if(a[b]>a[b+1]) {  
                int c;  
                c=a[b];  
                a[b]=a[b+1];  
                a[b+1]=c;  
            }  
        }  
    }  
}
```

实际中直接使用 array.sort() 进行排序

排序快速方法 尽量降低对堆内存中的实际元素改变位置的操作

第一次遍历记住最大值序号(栈中的)然后将最大值与最后一个位置的值换位置 依次类推

数组折半查找

```
public static int halfSearch(int[] arr, int key)
{
    int min, max, mid;
    min = 0;
    max = arr.length - 1;
    mid = (max + min) / 2;

    while (arr[mid] != key) {
        if (key > arr[mid])
            min = mid + 1;
        else if (key < arr[mid])
            max = mid - 1;

        if (min > max)
            return -1;
        mid = (max + min) / 2;
    }
    return mid;
}
```

注意返回-1的条件 $min > max$

想办法夹逼成一个数 上下界相等 接着如果还不是 条件使得上下界必颠倒

```
int c=0;
int d=a.length-1;
int mid=(d+c)/2;
```

数组中间位置的求法 $(\text{首位置} + \text{尾位置})/2 = \text{中间位置}$

数组复制的几种方法

1. System.arraycopy(原数组, 原数组的起始位置, 目标数组, 目标数组起始位置, 要 copy 的长度)

```
int[] a1 = {1, 2, 3, 4, 5};
int[] a2 = new int[10];

System.arraycopy(a1, 1, a2, 3, 3);
System.out.println(Arrays.toString(a1)); // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(a2)); // [0, 0, 0, 2, 3, 4, 0, 0, 0, 0]
```

2. .clone()

clone 方法是从 Object 类继承过来的, 基本数据类型 (int , boolean, char, byte, short, float , double, long) 都可以直接使用 clone 方法进行克隆, 注意 String 类型是因为其值不可变所以才可以使用。

int 类型示例

```
1 int[] a1 = {1, 3};
2 int[] a2 = a1.clone();
3
4 a1[0] = 666;
5 System.out.println(Arrays.toString(a1)); // [666, 3]
6 System.out.println(Arrays.toString(a2)); // [1, 3]
```

String类型示例

```
1 String[] a1 = {"a1", "a2"};
2 String[] a2 = a1.clone();
3
4 a1[0] = "b1"; //更改a1数组中元素的值
5 System.out.println(Arrays.toString(a1)); // [b1, a2]
6 System.out.println(Arrays.toString(a2)); // [a1, a2]
```

3. Arrays.copyOfRange (原数组, 开始位置, 拷贝的个数)

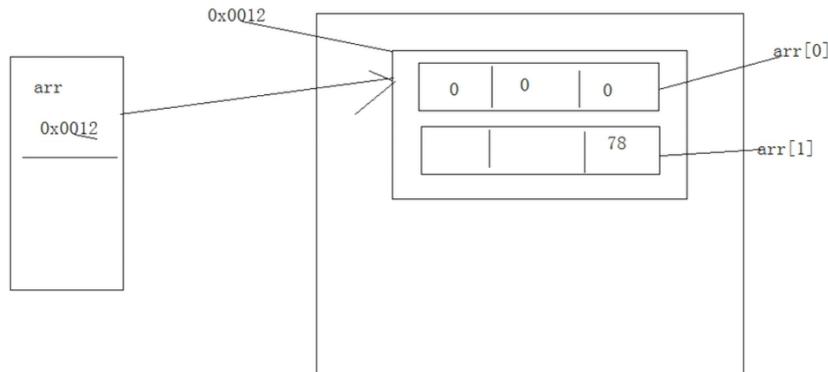
```
int[] a1 = {1, 2, 3, 4, 5};
int[] a2 = Arrays.copyOfRange(a1, 0, 1);

System.out.println(Arrays.toString(a1)) // [1, 2, 3, 4, 5]
System.out.println(Arrays.toString(a2)) // [1]
```

注意的是基本类型的拷贝是不影响原数组的值的, 如果是引用类型, 就不能在这用了, 因为数组的拷贝是浅拷贝, 对于基本类型可以, 对于引用类型是不适合的。除了 string string 的值是不可更改的

二维数组

```
int[] arr = new int[3];//一维数组。  
int[][] arr = new int[3][4];//定义了名称为arr的二维数组。二维数组中有3个一维数组。  
//每一个一维数组中有四个元素。  
int[][] arr = new int[2][3];  
arr[1][2] = 78;  
arr[0][3] = 90;no
```

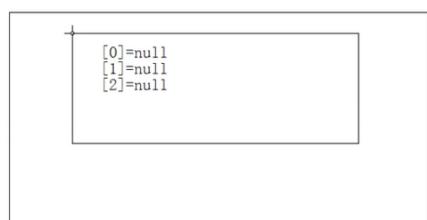


打印 arr 为地址值 arr[0]为另一个地址值

任何引用的初始化值都为 null

`int[][] arr = new int[3][];` 可以先只定义二维数组的长度，随后再定义数组里数组的长度

此时打印 arr[0]输出为 null

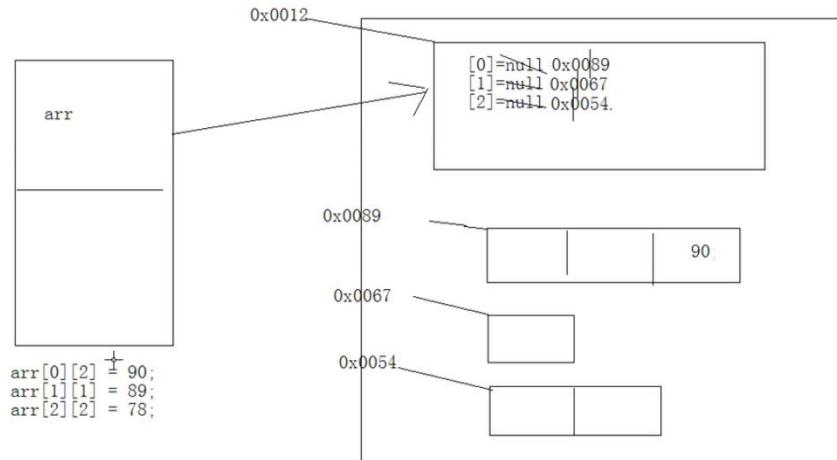


在堆中只命名了 0, 1, 2 并未分配

在手动分配后

```
arr[0] = new int[3];  
arr[1] = new int[1];  
arr[2] = new int[2];
```

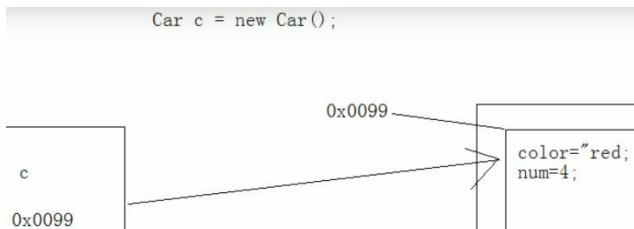
```
int[][] arr = new int[3][]; arr[0] = new int[3];  
arr[1] = new int[1]; arr[2] = new int[2];
```



面向对象

面向对象三个特征：封装 继承 多态

类和对象关系：类是对现实生活中对事物的描述 对象是这类事物实实在在存在的个体(在堆内存中)



单纯 new car 时 color=null(string) num=0(int) 堆内存所有值都有初始值

静态变量

成员变量 整个类中有效 存于堆内存中

局部变量 存于栈内存中

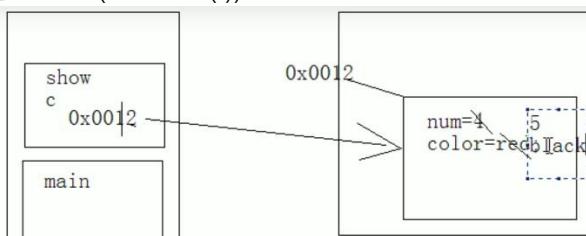
匿名对象使用

1. 直接使用对象方法 new car().run() 只对对象方法调用一次时 可以用匿名对象完成 这样写简化
2. 将匿名对象作为实际参数传递 show(new Car())

```

main()
{
    // Car c = new Car();
    // show(c);
    show(new Car());
}
public static void show(Car c)
{
    c.num = 3;
    c.color = "black";
    c.run();
}

```



函数的生命周期只在运行时 运行结束后变成垃圾被清理

show()结束后被清理---对象的指向被清理 对象自身也随后被清理

封装

Private: 只在本类中有效 可以通过 get/set 来访问 通过控制函数来控制成员变量范围

```

private int age;

public void setAge(int a)
{
    age = a;
}

public int getAge()
{
    return age;
}

public void setAge(int a)
{
    if(a>0 && a<130)
        age = a;
    else
        System.out.println("feifa age");
}

```

之所以对外提供访问方式，就因为可以在访问方式中加入逻辑判断等语句。
对访问的数据进行操作。提高代码健壮性。

函数都在栈中 包括 main 函数

成员变量---在堆内存中---都有默认初始化值

局部变量---栈内存中---不初始化无法参与运算

Main 在栈中 set 方法在栈中 里面的 int a 是局部变量 也在栈中 a 被赋予值
再把 a 的值赋予 age 而不是指向堆中的 age a 是 int 简单类型-按值传递

```

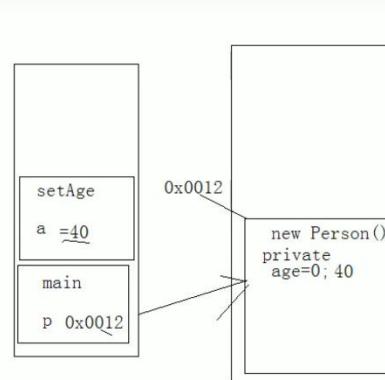
private int age;
public void setAge(int a)
{
    if(a>0 && a<130)
    {
        age = a;
        speak();
    }
    else
        System.out.println("feifa age");
}

public int getAge()
{
    return age;
}

void speak()
{
    System.out.println("age="+age);
}

class PersonDemo
{
    main()
    {
        Person p = new Person();
        //p.age = -20;
        p.setAge(40);
        p.speak(); 40
    }
}

```



构造函数

作用：用于给对象进行初始化 对象一建立就会调用与之对应的构造函数

当类中无定义的构造函数时，系统自动创建一个空参数的构造函数，类中有自定义的构造函数默认的就没有

```

private String name;
private int age;
/*
Person()
{
    System.out.println("A: name="+name+", age="+age);
}
*/
Person(String n)
{
    name = n;
    System.out.println("B: name="+name+", age="+age);
}

```

这种情况系统不会给你默认的空参构造函数 Person p1=new person()会报错

构造代码块：给所有对象进行统一初始化 先于构造函数执行

```

class Person
{
    private String name;
    private int age;

    /*
    构造代码块。
    作用：给对象进行初始化。
    对象一建立就运行，而且优先于构造函数执行。
    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，
    而构造函数是给对应的对象初始化。|
    */
    {
        System.out.println("person code run");
    }

    Person()
    {
        System.out.println("A: name="+name+", age="+age);
        cry();
    }
}

```

This

This 代表本类的对象 是所属对象的引用 那个对象调用 this 所在函数，this 就代表哪个对象

成员变量 name 为 null 当这种情况时 有局部变量优先认为是局部变量 所以局部=局部 成员=null this.name 来指明成员变量

```

private String name;
private int age;

Person(String name)
{
    name = name;
}

```

This 在构造函数间的使用

```

private String name;
private int age;

Person(String name)
{
    this.name = name;
}
Person(String name,int age)
{
    //this.name = name;
    this(name); //p(name);
    this.age = age;
}

```

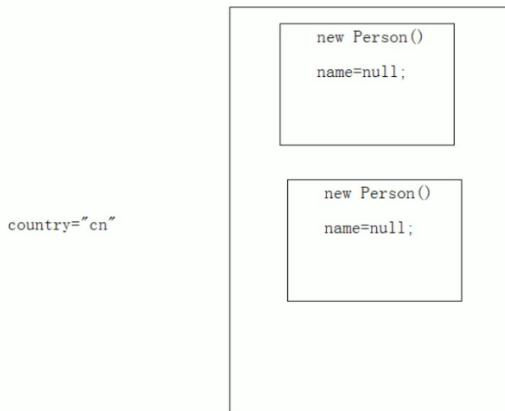
但只能放在构造函数中的第一行

防止想自定义的属性被初始化的属性覆盖掉

Static 静态

Static 优先于对象 随着类的加载而加载 在 static 中是没有 this 关键字的(静态时对象可能还未创建)—this 代表当前对象的引用

当成员变量或成员函数 static 后就不在堆内存中了 被单独提取出来放在外面 (方法区/共享区) 每个对象都可以使用 而不是在每个对象中都存在一份



方法名放在栈中 静态变量, 全局变量, class 放在方法区 成员变量随着对象存在堆中

Static 后此静态变量或方法就属于类的 随着类的加载而加载 随着类的消失而消失 优先于对象存在 被所有对象共享 非静态变量 对象消失后自动垃圾回收 静态只有类消失后才消失 对内存消耗较大

当类加载到内存中时 country 还在 name 不存在 因为只有创建对象后 对象里面才有 name(成员变量)

```

class Person
{
    String name;
    static String country = "CN";
    public void show()
    {
        System.out.println(name+";"+country);
    }
}

```

静态方法中不能直接引用非静态变量 有对象后才可以

错误 没有 new 个 person 时 对象不存在 因此 name 成员变量还不存在 非静态方法同理-无对象依靠

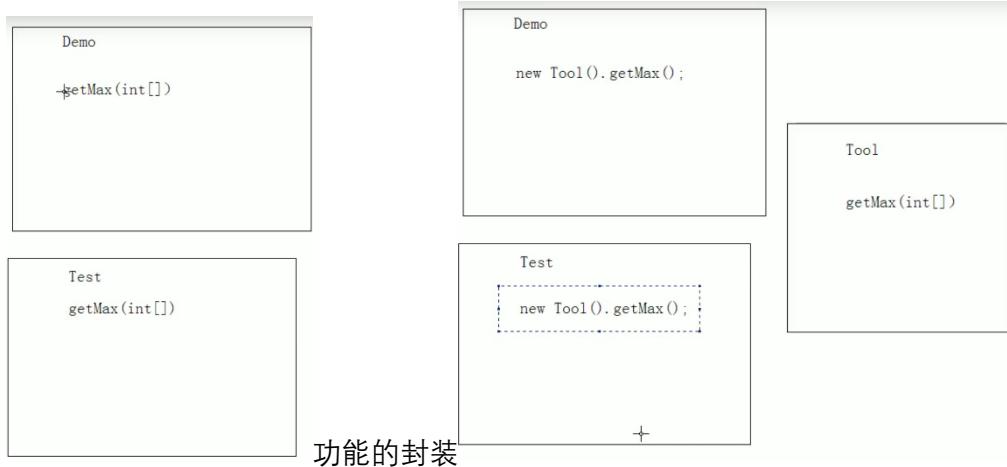
```

class Person
{
    String name;//成员变量, 实例变量。
    static String country = "CN";//静态的成员变量, 类变量。
    public static void show()
    {
        System.out.println(":::"+name);
    }
}

```

静态方法中无 this.---有对象才行

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



代码块

代码块概述

- * 在Java中，使用{}括起来的代码被称为代码块。

代码块分类

- * 根据其位置和声明的不同，可以分为局部代码块，构造代码块，静态代码块，同步代码块(多线程讲解)。

常见代码块的应用

a: 局部代码块

- * 在方法中出现；限定变量生命周期，及早释放，提高内存利用率

b: 构造代码块（初始化块）

- * 在类中方法外出现；多个构造方法方法中相同的代码存放到一起，每次调用构造都执行，并且在构造方法前执行

c: 静态代码块

- * 在类中方法外出现，并加上static修饰；用于给类进行初始化，在加载的时候就执行，并且只执行一次。

- * 一般用于加载驱动

构造代码块：给所有对象进行统一初始化 先于构造函数执行

```
class Person
{
    private String name;
    private int age;

    /*
     * 构造代码块。
     * 作用：给对象进行初始化。
     * 对象一建立就运行，而且优先于构造函数执行。
     * 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，
     * 而构造函数是给对应的对象初始化。|
    */
    {
        System.out.println("person code run");
    }

    Person()
    {
        System.out.println("A: name="+name+", age="+age);
        cry();
    }
}
```

静态代码块 随着类的加载而执行 只执行一次 用于给类初始化 (顺序：静态代码块→main 函数)

```
class StaticCode
{
    static
    {
        System.out.println("a");
    }
}
```

此时不会打印“a” s 只在栈中 类实体在堆中并没有被创建 =null 空有引用而已 只有实际内容时才会加载类

```

class StaticCode
{
    static
    {
        System.out.println("a");
    }
    public static void show()
    {
        System.out.println("show run");
    }
}

```

```

class StaticCodeDemo
{
    static
    {
        //System.out.println("b");
    }
    public static void main(String[] args)
    {
        //new StaticCode();
        //new StaticCode();
        //System.out.println("over");
        //StaticCode.show();
        StaticCode s = null;
    }
}

```

s=new staticCode()会加载

结果 a c9 d

```

class StaticCode
{
    int num = 9;
    StaticCode()
    {
        System.out.println("b");
    }

    static
    {
        System.out.println("a");
    }
    {
        System.out.println("c"+this.num);
    }
}

StaticCode(int x)
{
    System.out.println("d");
}
public static void show()
{
    System.out.println("show run");
}
}

```

```

class StaticCodeDemo
{
    static
    {
        //System.out.println("b");
    }
    public static void main(String[] args)
    {
        new StaticCode(4); //a c d
    }
}

```

无 this.num 时是 acd

静态代码块 static{} 构造代码块 {}

静态代码块 类创建时执行 构造代码块 创建对象时执行 先于构造函数运行

Main 函数

主函数作为程序的入口 被虚拟机调用 固定格式 被 jvm 识别

Public static void main(String [] args) 可以有重名的重载函数 但虚拟机只认这个函数

```

public static void main(String[] args)
{
    System.out.println(args);
}

```

虚拟机传入的真是一个字符串数组 jvm 在调用主函数时，传入的是 new String[0];

Javac 命令启动编译器 编译 java 文件 生成.class 文件

Java 命令启动底层虚拟机 虚拟机执行后面的类 然后自动调用类中的 main 方法

```

D:\java0217\day06>javac MainDemo.java
D:\java0217\day06>java MainDemo

```

```

public static void main(String[] args)//new String[]
{
    System.out.println(args[0]);
}

```

下标越界

```

D:\java0217\day06>java MainDemo haha hehe heihei
haha

```

此时输出 haha args 数组包含三个值 ha he hei
ha he hei 当作 main 函数中传入的参数----String[]args

若一个 java 文件中包含另一个类的引用 会先将引用的类翻译成.class 再把自己编译成.class

```

ArrayTool.class
ArrayToolDemo.class

```

arraytoolDemo 类中引用了 arraytool 类 javac arraytoolDemo.java 时会出现两个 class 文件

```

class ArrayTool
{
    public int getMax(int[] arr)
    {
        int max = 0;
        for(int x=1; x<arr.length; x++)
        {
            if(arr[x]>arr[max])
                max = x;
        }
        return arr[max];
    }

    public int getMin(int[] arr)
    {
        int min = 0;
    }
}

```

虽然可以通过建立ArrayTool的对象使用这些工具方法，对数组进行操作。

发现了问题：

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

这时就考虑，让程序更严谨，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。无需创造 arraytool 对象

```

class ArrayTool
{
    //ArrayTool(){}
    public static int getMax(int[] arr)
    {
        int max = 0;
        for(int x=1; x<arr.length; x++)
        {
            if(arr[x]>arr[max])
        }
    }
}

```

工具类中方法都改成 static

```

class ArrayToolDemo
{
    public static void main(String[] args)
    {
        int[] arr = {3,1,87,32,8};
        ArrayTool tool = new ArrayTool();

        int max = tool.getMax(arr);
        System.out.println("max="+max);
    }
}

```

将方法都静态后，可以方便于使用，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。

```

class ArrayTool
{
    private ArrayTool(){}
    public static int getMax(int[] arr)
    {
        int max = 0;
        for(int x=1; x<arr.length; x++)
        {
            if(arr[x]>arr[max])
        }
    }
}

```

此时别的类无法再创建 arraytool 对象

Private arraytool(){ } 通过私有化创建空参数构造函数的方式来防止其他类创建其对象

Javadoc 说明文档制作

<https://www.bilibili.com/video/av47654363?p=78>

对象初始化过程

Person p=new person("me", 3)

先栈中开辟 main 方法空间 内部出现 p 引用 → 静态代码块执行 → 堆中开始初始化 person 类(在堆中开辟空间) → 构造函数，成员变量(默认初始化或有初始值)

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



先 main 函数在栈中创建空间 空间内含有 p 变量

- 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。

显示初始化: public int a=1;

画图说明一个对象的创建过程做了哪些事情?

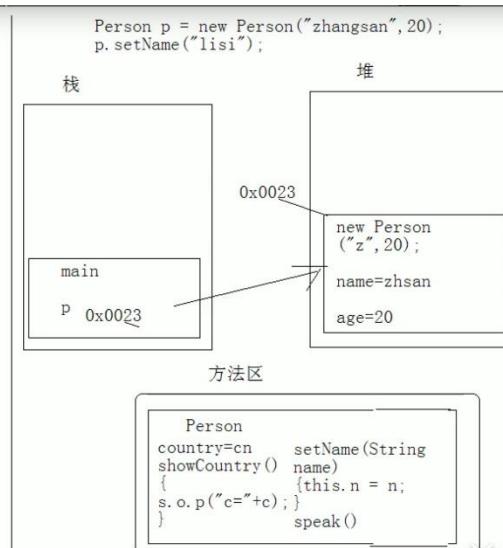
```
Student s = new Student();  
1, Student.class加载进内存  
2, 声明一个Student类型引用s  
3, 在堆内存创建对象,  
4, 给对象中属性默认初始化值  
5, 属性进行显示初始化  
6, 构造方法进栈, 对象中的属性赋值, 构造方法弹栈  
7, 将对象的地址值赋值给s
```

对象调用过程

```
Person p=new person("me", 3);    p.setName("you");
```

栈 方法区 堆

```
class Person  
{  
    private String name;  
    private int age;  
    private static String country = "cn";  
    Person(String name, int age)  
    {  
        this.name = name;  
        this.age = age;  
    }  
  
    public void setName(String name)  
    {  
        this.name = name;  
    }  
  
    public void speak()  
    {  
        System.out.println(this.name+"..."+this.age)  
    }  
  
    public static void showCountry()  
    {  
        System.out.println("country="+country);  
    }  
}
```



静态变量 静态方法 普通方法 方法代码 都存在方法区中

```

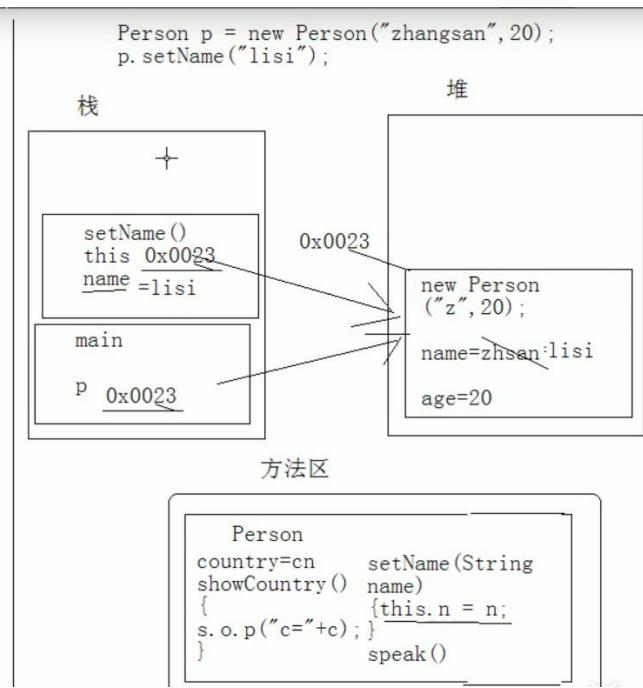
class Person
{
    private String name;
    private int age;
    private static String country = "cn";
    Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

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

    public void speak()
    {
        System.out.println(this.name+"..."+this.age);
    }

    public static void showCountry()
    {
        System.out.println("country="+country);
    }
}

```



This 包含对象的引用(p 的值(引用地址)赋予 this) 谁调用它 就给谁赋值

方法区内左边是静态区-静态变量，静态方法 右边是非静态区-成员变量，成员方法 this 在非静态区中
Name 是局部变量

静态方法 showCountry 直接走的 class 从方法区 不从堆中走 因为不需要调用对象

SetName 函数运行完之后自动在栈中消失 垃圾处理

栈中 方法内的局部变量/方法内的引用

单例设计模式

目的：解决一个类在内存中只存在一个对象 保持对象的唯一性

1.禁止其它程序建立该类对象 2.为了让其他程序可访问该类对象，在本类中自定义对象 3.对外提供访问接口
构造函数私有化 创建一个本类对象 提供方法获取对象

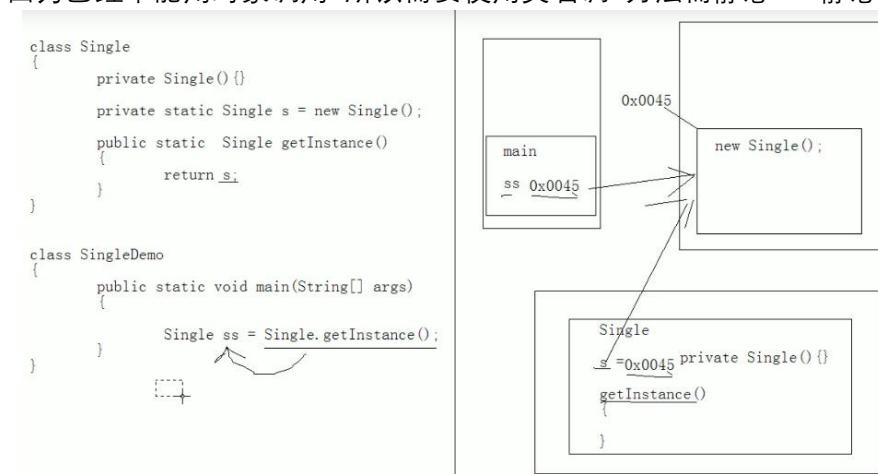
方法被调用两种方式：1.通过实例化对象然后调用 2.通过类名直接调用

```

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

```

因为已经不能用对象调用 所以需要使用类名调 方法需静态 静态方法只能访问静态成员 所以成员静态



首先 main 方法栈中开辟空间 里面包含 ss 局部变量

其次 Single.getInstance 时 single 时 在方法区中开辟 single 类空间：

首先 single 构造函数 其次有静态 s 变量-->在堆中创建一个 single 对象，把地址赋予 s

再有静态 getInstance 方法 .getInstance 时 返回一个地址值 把值又赋予了 ss

单例设计模式三步：1.私有化构造函数 2.创建私有并静态对象 3.公开并静态 get return 对象方法

先初始化对象 --- 饿汉式

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

方法被调用时才初始化对象(对象的延迟加载) --- 懒汉式

```
class Singleton {
    //1, 私有构造方法, 其他类不能访问该构造方法
    private Singleton(){}
    //2, 声明一个引用
    private static Singleton s;
    //3, 对外提供公共的访问方法
    public static Singleton getInstance() {
        if(s == null) {
            s = new Singleton();
        }
        return s;
    }
}
```

null 时堆中无对象，调用方法时才创建

但这个是线程不安全的

可能 A 进来后被暂停 B 又进来了 此时会创建 A&B 两个对象
解决方法：加锁(同步)---A 进来后不允许其它再进来

```
public static synchronized Single getInstance()
{
    if(s==null)
    {
        -->A
        -->B
        s = new Single();
    }
}
```

但是 synchronized 加上这个效率会变低 完美方案

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

继承

继承时不会继承构造函数

提高代码复用性 让类与类之间产生关系，有了这个关系才有多态的特性

关键字 extends java 只支持单继承 (用可实现多个接口弥补)

```
class Fu
{
    int num = 4;
}

class Zi extends Fu
{
    int num = 5;
    void show()
    {
        System.out.println(num);
    }
}
```

Zi z=new Zi();

z.num=5

这里的 println 指的是 this.num(5) 若想指父类 使用 super.sum

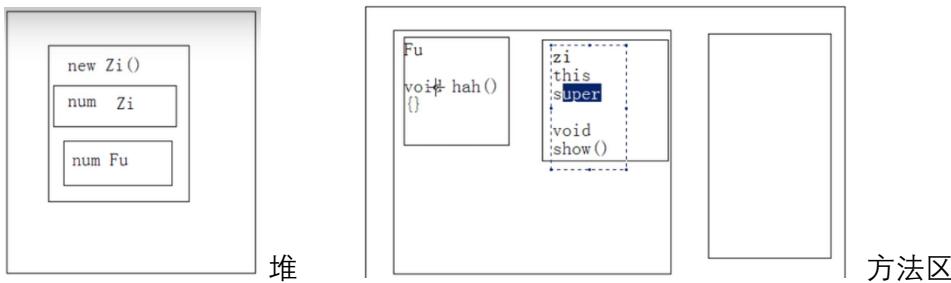
this 指本类对象的引用 super 指父类对象的引用 都在方法区内(非静态区)

堆中

New Zi () 时

会包括子类父类两个类的成员变量(即使父类成员变量私有)

加载 Zi.class 时会先加载 Fu.class 无论是不是 private 都会加载 只不过是能不能访问的问题



当子类和父类有同名方法 zi.method 会指向子类方法 此叫 override

只要函数名相同 参数列表以及各个参数类型相同就可以 返回值相同或是其返回类型的子类型

子类的方法权限必须大于父类要被重写的权限(父类 public 子类 protect 不行)

static 方法不能被重写 基本类型不是类(int double) 不能被继承

若同名后还想用父类的方法 可在同名方法内用 super super.show()

```
class Tel
{
    void show()
    {
        System.out.println("number");
    }
}

class NewTel extends Tel
{
    void show()
    {
        //System.out.println("number");
        super.show();
        System.out.println("name");
        System.out.println("pic");
    }
}
```

```
class Fu
{
    Fu()
    {
        System.out.println("fu run");
    }
    Fu(int x)
    {
        System.out.println("fu..."+x);
    }
}

class Zi extends Fu
{
    Zi()
    {
        //super();
        System.out.println("zi run");
    }
    Zi(int x)
    {
        //super();
        System.out.println("zi..."+x);
    }
}

class ExtendsDemo4
{
    public static void main(String[] args)
    {
        Zi z = new Zi();
        Zi z1 = new Zi(4);
    }
}
```

在对子类对象进行初始化时，父类的构造函数也会运行，
那是因为子类的构造函数默认第一行有一条隐式的语句 super();
super():会访问父类中空参数的构造函数。而且子类中所有的构造函数默认第一行都是super();

子类构函都会隐式调用父类空参数构函，若父类只有有参数构函(无 Fu()), 会报错，父类无参必须显式声明

若只想子类调用父类有参构函，子类中显示声明 super(4)，此时不会再隐式调用

总之子类一定会显式或隐式的调用父类构造函数 (子类对象建立时看看父类是如何初始化的)

子类所有的构造函数，默认都会访问父类中空参数的构造函数。
因为子类每一个构造函数内的第一行都有一句隐式super();

当父类中没有空参数的构造函数时，子类必须手动通过super语句形式来指定要访问父类中的构造函数。

当然，子类的构造函数第一行也可以手动指定this语句来访问本类中的构造函数。
子类中至少会有一个构造函数会访问父类中的构造函数。

父类其实也有 super()---object

父类中定义完的子类不需要重新定义

```

class Person
{
    private String name;
    Person(String name)
    {
        this.name = name;
    }

    void show(){}
}

class Student extends Person
{
    Student(String name)
    {
        super(name);
    }
    void method()
    {
        super.show();
    }
}

```

```

zi()
{
    //super();
    //super(4);
    System.out.println("zi run");
}
zi(int x)
{
    this();
    //super();
    //super(3);
    System.out.println("zi..."+x);
}

```

this()/super()只能放第一行 只能存在一个

方法重写的面试题

- * Override和Overload的区别?Overload能改变返回值类型吗?
- * overload可以改变返回值类型,只看参数列表
- * 方法重写 : 子类中出现了和父类中方法声明一模一样的方法。与返回值类型有关,返回值是一致(或者是子父类)的
- * 方法重载 : 本类中出现的方法名一样,参数列表不同的方法。与返回值类型无关。
- * 子类对象调用方法的时候 :
 - * 先找子类本身,再找父类。

`Final` 修饰后:

类不能继承

方法不能覆写

变量只能赋值一次,常量

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

当出现这种情况时 `study` 方法中不需要有内容 只是想借用方法名 此时可以把它变成抽象

```

class Student
{
    void study(){}
}

class BaseStudent extends Student
{
    void study()
    {
        System.out.println("base study");
    }
}

class AdvStudent extends Student
{
    void study()
    {
        System.out.println("adv study");
    }
}

```

当多个类中出现相同功能 但是功能主体不同 这时可以进行向上抽取

此时 只抽取功能定义 而不抽取功能主体

强迫子类去实现其方法

抽象类的特点：

- 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。
- 4, 抽象类中的方法要被使用，必须由子类复写起所有的抽象方法后，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。

```
/*
abstract class Student
{
    abstract void study();
}
```

子类中实现相同的直接定义 子类中为不同实现的 抽取出为抽象方法

```
abstract class Student
{
    abstract void study();
    //abstract void study1();
    void sleep()
    {
        System.out.println("躺着");
    }
}
```

子类学习内容各自不同 但都有 sleep 躺着 只需抽象 study 即可

A:面试题1

- * 一个抽象类如果没有抽象方法，可不可以定义为抽象类？如果可以，有什么意义？
- * 可以
- * 这么做目的只有一个，就是不让其他类创建本类对象，交给子类完成

B:面试题2

- * abstract不能和哪些关键字共存
abstract和static
被abstract修饰的方法没有方法体
被static修饰的可以用类名.调用，但是类名.调用抽象方法是没有意义的
abstract和final
被abstract修饰的方法强制子类重写
被final修饰的不让子类重写，所以他俩是矛盾
abstract和private
被abstract修饰的是为了让子类看到并强制重写
被private修饰不让子类访问，所以他俩是矛盾的

模板方法设计模式

在定义功能时，功能的一部分是确定的，但是有一部分不确定，而确定的部分在使用不确定的部分，那么这时把不确定的方法暴露出去，抽出来变成一个单独的抽象方法，让子类去实现---runCode

需求：获取一段程序运行的时间。

原理：获取程序开始和结束的时间并相减即可。

```
获取时间：System.currentTimeMillis();
```

Final 是因为不想让子类覆写方法

```
abstract class GetTime
{
    public final void getTime()
    {
        long start = System.currentTimeMillis();
        runCode();
        long end = System.currentTimeMillis();
        System.out.println("毫秒：" + (end - start));
    }
    public abstract void runCode();
}

class SubTime extends GetTime
{
    public void runCode()
    {
        for(int x=0; x<4000; x++)
        {
            System.out.print(x);
        }
    }
}

class TemplateDemo
{
    public static void main(String[] args)
    {
        GetTime gt = new GetTime();
        gt.getTime();
    }
}
```

多态

多态前提：有继承关系 有方法重写 要有父类引用指向子类对象

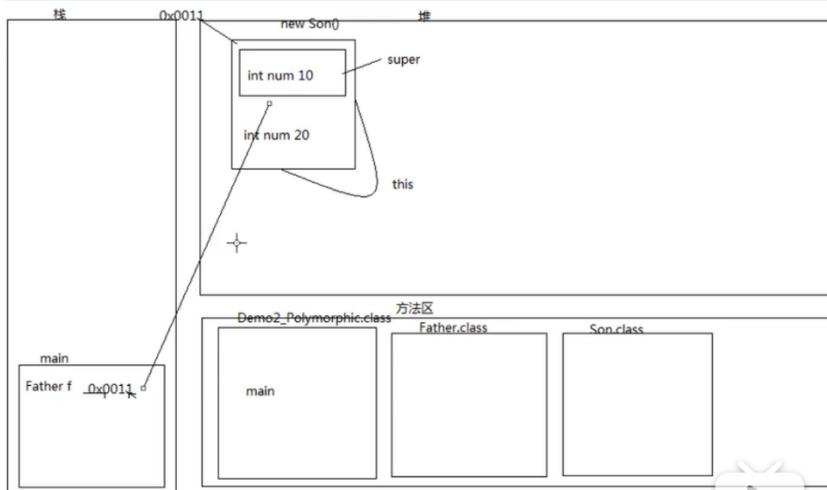
Father f = new son(); f.num 结果为 10

```

class Demo2_Polymerphic {
    public static void main(String[] args) {
        Father f = new Son();
        System.out.println(f.num);
    }
}
/*
成员变量
*/
class Father {
    int num = 10;
}

class Son extends Father {
    int num = 20;
}

```



首先解析 class Demo 进入方法区

再解析其中的 main 方法在方法区中

主方法运行后就会进栈，此时再在栈中开辟 main 空间

Main 中有 father 所以方法区中加载 father 的 class

New son 所以方法区中加载 son 的 class

Father f 此时出现在栈中 在堆中 new 出 son 对象

son 中有一个空间叫 super super()是子类用来访问父类东西的

this 代表本类对象的引用

子类对象的引用将赋给 father f 此时 father 只能看到 super 内容—father 自己的 因为他只是 father 看不到 num=20 子类它自己的成员变量

成员变量 编译看左边(父类) 运行看左边(父类)

Father f = new son();

```

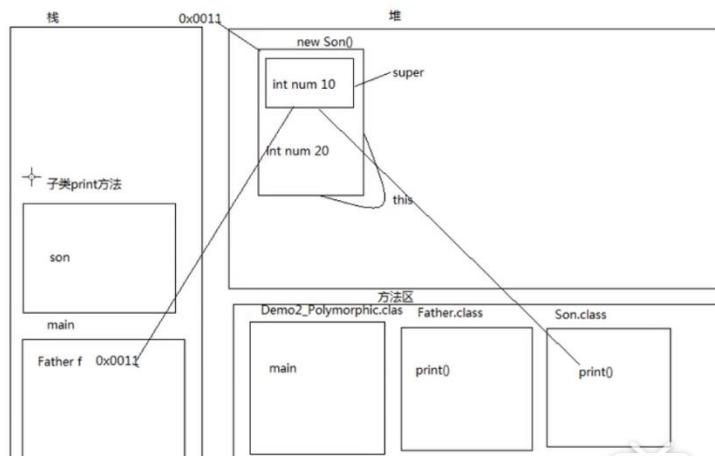
class Demo2_Polymerphic {
    public static void main(String[] args) {
        Father f = new Son();
        f.print();
    }
}

class Father {
    int num = 10;
    public void print() {
        System.out.println("father");
    }
}

class Son extends Father {
    int num = 20;

    public void print() {
        System.out.println("son");
    }
}

```



成员方法 编译看左边(父类) 运行看右边(子类)

Father f = new son();

编译的时候会先看 father.class 中有没有 print 方法(若无编译不会通过, f 中没有 print 方法) 运行的时候会调用子类的 print 方法---动态绑定(静态方法不会被绑定, 还是 father 的方法)

真正进栈的是子类的 print 方法 若子类无 print 则继承使用父类的 print

静态方法 编译看左边(父类) 运行看左边(父类)

Father 类中

```

public static void method() {
    System.out.println("father static method");
}

```

Father f = new son(); f.method

静态和类相关 相当于调用类的方法 f.method(father.method) 算不上重写(每个类都有各自的方法) 所以访问还是左边的 只有非静态的成员方法 运行看右边

父类中有方法 a 子类中有方法 a 和 b f=new z f.a 是子重写的方法 f.b 会报错 f 中无 b 方法 此时想使用 b 方法，通过向下转型(强制类型转换)f->s s.b 来实现 注意必须 f 你原来就是个 s 才能实现 基本数据类型中

```
int i = 10;
byte b = 20;
//i = b; //自动类型提升
//b = (byte)i; //强制类型转换
```

对象中

Father f = new son(); 向上转型(自动类型提升)

Son s= (Son) f ; 向下转型(强制类型转换) 此时 f 必须本来就是个 son

多态的好处：提高代码的维护性(继承保证) 提高代码的扩展性(由多态保证)

弊端：不能使用子类特有的属性和行为

多态真正使用的情景

```
public static void main(String[] args) {
    //Cat c1 = new Cat();
    //c1.eat();
    method(new Cat());
    method(new Dog());

    //Animal a = new Cat(); //开发的是很少在创建对象的时候用父类引用指向子类对象
}

//Cat c = new Dog(); 狗是一只猫,这是错误的
/*public static void method(Cat c) {
    c.eat();
}

public static void method(Dog d) {
    d.eat();
}*/

public static void method(Animal a) { //当作参数的时候用多态最好,因为扩展性强
    a.eat();
}
```

instanceof 关键字

```
//关键字 instanceof 判断前边的引用是否是后边的数据类型
if (a instanceof Cat) {
    Cat c = (Cat)a;
    c.eat();
    c.catchMouse();
```

例题

A a=new b(); 结果 爱 b 中由一个继承的 show()

```
class A {
    public void show() {
        show2();
    }
    public void show2() {
        System.out.println("我");
    }
}
class B extends A {
    public void show2() {
        System.out.println("爱");
    }
}
class C extends B {
    public void show() {
        super.show();
    }
    public void show2() {
        System.out.println("你");
    }
}
```

B b=new c();
b.show(); 结果 你

接口

对外提供规则的是接口

Class 类名 implements 接口名 {}

接口内都是抽象方法 用多态的方式实例化

```

interface Inter {
    public abstract void print();
}

class Demo implements Inter {
    public void print() {
        System.out.println("print");
    }
}
Inter i = new demo()    接口相当于父类引用 指向子类对象
i.print()   结果 print

```

在接口中定义的变量自动是常量 隐式 final 都是 static 可以直接 print inter.num 结果为 10

```

interface Inter {
    public static final int num = 10;
}
interface Inter {
    int num = 10;
    ==    }

```

接口无构造方法

接口内中的方法默认 public abstract

A:类与类,类与接口,接口与接口的关系

- * a:类与类:
 - * 继承关系,只能单继承,可以多层继承。
- * b:类与接口:
 - * 实现关系,可以单实现,也可以多实现。
 - * 并且还可以在继承一个类的同时实现多个接口。
- * c:接口与接口:
 - * 继承关系,可以单继承,也可以多继承。

```

class Demo implements InterA,InterB {
interface InterC extends InterB,InterA {

```

9.19 面向对象(抽象类和接口的区别)

成员区别

- * 抽象类:
 - * 成员变量:可以变量,也可以常量
 - * 构造方法:有
 - * 成员方法:可以抽象,也可以非抽象
- * 接口:
 - * 成员变量:只可以常量
 - * 成员方法:只可以抽象

设计理念区别

- * 抽象类 被继承体现的是:"is a"的关系。抽象类中定义的是该继承体系的共性功能。
- * 接口 被实现体现的是:"like a"的关系。接口中定义的是该继承体系的扩展功能。

动物类: 姓名, 年龄, 吃饭, 睡觉。

猫和狗

动物培训接口: 跳高 I 动物类抽象出来 猫和狗类 跳高接口

```

abstract class Animal {
    private String name;           //姓名
    private int age;               //年龄

    public Animal() {}             //空参构造
    public Animal(String name,int age) {} //有参构造
        this.name = name;
        this.age = age;
    }

    public void setName(String name) { //设置姓名
        this.name = name;
    }

    public String getName() {       //获取姓名
        return name;
    }
}

class Cat extends Animal {
    public Cat() {}               //空参构造
    public Cat(String name,int age) {} //有参构造
        super(name,age);
    }

    public void eat() {
        System.out.println("猫吃鱼");
    }

    public void sleep() {
        System.out.println("侧着睡");
    }
}

interface Jumping {
    public void jump();
}

```

```

class JumpCat extends Cat implements Jumping {
    public JumpCat() {} //空参构造
    public JumpCat(String name, int age) { //有参构造
        super(name, age);
    }
    public void jump() {
        System.out.println("猫跳高");
    }
}

public static void main(String[] args) {
    Cat c = new Cat("加菲", 8);
    c.eat();
    c.sleep();

    JumpCat jc = new JumpCat("跳高猫", 3);
    jc.eat();
    jc.sleep();
    jc.jump();
}

```

内部类

a: 内部类可以直接访问外部类的成员，包括私有。

b: 外部类要访问内部类的成员，必须创建对象。

外部类名.内部类名 对象名 = 外部类对象.内部类对象；

创建内部类对象

```

class Outer {
    private int num = 10;
    class Inner {
        public void method() {
            System.out.println(num);
        }
    }
}

Outer.Inner oi = new Outer().new Inner();
oi.method();

```

若成员内部类被私有 通过在外部类中创建 public 方法访问内部类来实现

```

class Outer {
    private int num = 10;
    private class Inner {
        public void method() {
            System.out.println(num);
        }
    }
}

public void print() {
    Inner i = new Inner();
    i.method();
}

Outer o = new Outer();
main: o.print();

```

静态内部类中的普通方法

```

class Outer {
    static class Inner {
        public void method() {
            System.out.println("method");
        }
    }
}

//外部类名.内部类名 对象名 = 外部类名.内部类对象;
Outer.Inner oi = new Outer.Inner();
oi.method();

```

静态内部类的静态方法

Outer.Inner.print();

面试题

内部类之所以能获取到外部类的成员，因为他能获取到外部类的引用外部类名.this

```

class Test1_InnerClass {
    public static void main(String[] args) {
        Outer.Inner oi = new Outer().new Inner();
        oi.show();
    }
}

//要求: 使用已知的变量，在控制台输出30, 20, 10。

class Outer {
    public int num = 10;
    class Inner {
        public int num = 20;
        public void show() {
            int num = 30;
            System.out.println();
            System.out.println(?);
            System.out.println(?);
        }
    }
}

```

```

class Test1_InnerClass {
    public static void main(String[] args) {
        Outer.Inner oi = new Outer().new Inner();
        oi.show();
    }
}

//要求: 使用已知的变量，在控制台输出30, 20, 10。

class Outer {
    public int num = 10;
    class Inner {
        public int num = 20;
        public void show() {
            int num = 30;
            System.out.println(num);
            System.out.println(this.num);
            System.out.println(Outer.this.num);
        }
    }
}

```

局部内部类

```

//局部内部类
class Outer {
    public void method() {
        class Inner {
            public void print() {
                System.out.println("Hello World!");
            }
        }
        Inner i = new Inner();      I
        i.print();
    }
}

/*public void run() {
    Inner i = new Inner();
    i.print();
} */

```

//局部内部类,只能在其所在的方法中访问

```

class Demo1_InnerClass {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.method();
    }
}

```

局部内部类访问局部变量

```

class Outer {
    public void method() {
        final int num = 10;
        class Inner {
            public void print() {
                System.out.println(num);
            }
        }
        Inner i = new Inner();
        i.print();
    }
}

```

局部内部类访问局部变量必须用final修饰

局部内部类在访问他所在方法中的局部变量必须用final修饰,为什么?

因为当调用这个方法时,局部变量如果没有用final修饰,他的生命周期和方法的生命周期是一样的,当方法弹栈,这个局部变量也会消失,那么如果局部内部类对象还没有马上消失想用这个局部变量,就没有了,如果用final修饰会在类加载的时候进入常量池,即使方法弹栈,常量池的常量还在,也可以继续使用

但是jdk1.8取消了这个事情,所以我认为这是个bug

T

匿名内部类

匿名内部类

* 就是内部类的简化写法。

前提:存在一个类或者接口

* 这里的类可以是具体类也可以是抽象类。

I

格式:

```

new 类名或者接口名 () {
    重写方法;
}

```

本质是什么呢?

* 是一个继承了该类或者实现了该接口的子类匿名对象。 前提存在一个类或接口

与有名内部类作对比

```

interface Inter {
    public void print();
}

class Outer {
    class Inner implements Inter {
        public void print() {
            System.out.println("print");
        }
    }

    public void method(){
        Inner i = new Inner();
        i.print(); I
    }
}

public static void main(String[] args) {
    Outer o = new Outer();
    o.method(); I
}

```

```

interface Inter {
    public void print();
}

class Outer {
    class Inner implements Inter {
        public void print() {
            System.out.println("print");
        }
    }

    public void method(){
        //Inner i = new Inner();
        //i.print();

        new Inter() {           //实现Inter接口
            public void print() { //重写抽象方法
                System.out.println("print");
            }
        }.print();
    }
}

```

直接 new inter()是不行的 这代表创建接口对象 不允许

new inter(){} 可以 代表实现 inter 接口 整体算 inter 的子类对象 所以可以直接在{}外加.print 调用 inter 子类对象的方法

new 一个类名(){} 代表继承这个类 new 一个接口(){} 代表实现这个接口(需要重写其内的抽象方法)
匿名内部类需要写到方法里?

```
public void method() {
    //Inner i = new Inner();
    //i.print();
    new Inner().print();
```



```
    new Inter() {
        public void print() {
            System.out.println("print")
        }
    }.print();
}
```

new inner 代表 inter 的子类对象 inter i =new inner 代表父类引用指向子类对象
inter i =new inter(){} 也代表父类引用指向子类对象

```
interface Inter {
    public void show1();
    public void show2();
}

class Outer {
    public void method() {
        new Inter(){
            public void show1() {
                System.out.println("show1");
            }

            public void show2() {
                System.out.println("show2");
            }
        }.show1();
    }
}

public static void main(String[] args) {
    Outer o = new Outer();
    o.method();
}
```

结果 show1 此时若想调用 show2 需要再次创建一个 inter 对象 麻烦

所以匿名内部类只针对重写一个方法时候

实际应用： 匿名内部类直接当参数传递

```
class Test1_NoNameInnerClass {
    public static void main(String[] args) {
        //如何调用PersonDemo中的method方法呢?
        PersonDemo pd = new PersonDemo ();
        pd.method();
    }
}

//这里写抽象类，接口都行
abstract class Person {
    public abstract void show();
}

class PersonDemo {
    public void method(Person p) {
        p.show();
    }
}
```

方法一：

```

class Test1_NoNameInnerClass {
    public static void main(String[] args) {
        //如何调用PersonDemo中的method方法呢?
        PersonDemo pd = new PersonDemo ();
        pd.method(new Student());
    }
}
//这里写抽象类, 接口都行
abstract class Person {
    public abstract void show();
}

class PersonDemo {
    public void method(Person p) {      //Person p = new Student();
        p.show();
    }
}

class Student extends Person {
    public void show() {
        System.out.println("show");
    }
}

```

方法二:

```

class Test1_NoNameInnerClass {
    public static void main(String[] args) {
        //如何调用PersonDemo中的method方法呢?
        PersonDemo pd = new PersonDemo ();
        //pd.method(new Student());
        pd.method(new Person(){           //匿名内部类当作参数传递(本质把匿名内部类看作一个对象)
            public void show() {
                System.out.println("show");
            }
        });
    }
}
//这里写抽象类, 接口都行
abstract class Person {
    public abstract void show();
}

class PersonDemo {
    //public void method(Person p) {      //Person p = new Student();
    public void method(Person p) {      //Person p = new Student();
        p.show();
    }
}

```

面试题

```

class Test2_NoNameInnerClass {
    public static void main(String[] args) {
        outer.method().show();
    }
}
//按照要求, 补齐代码
interface Inter {
    void show();
}

class Outer {
    //补齐代码
}

//要求在控制台输出"HelloWorld"

```

题中时 outer.method---用类名调用方法 所以 method 应是静态 static

Show 在 inter 中 代表它非静态---interface 中方法是抽象的 抽象不可与静态共存 而非静态一定要与对象才能调用(不能通过类名调) .show 代表 outer.method 返回的是一个对象

```

class Test2_NoNameInnerClass {
    public static void main(String[] args) {
        Outer.method().show();
    }
}
//按照要求，补齐代码
interface Inter {
    void show();
}

class Outer {
    //补齐代码
    public static Inter method() {
        return new Inter() {
            public void show() {
                System.out.println("HelloWorld");
            }
        };
    }
}

```

常见对象

API

- * 应用程序编程接口
- Java API
- * 就是Java提供给我们使用的类，这些类将底层的实现封装了起来，
- * 我们不需要关心这些类是如何实现的，只需要学习这些类如何使用。

```

public static void main(String[] args) {
    Student s = new Student("张三", 23);

    //Class clazz = new Class()
    Class clazz = s.getClass();           //获取该对象的字节码文件
    String name = clazz.getName();        //获取名称
    System.out.println(name);
```

引用别忘了未重写 `toString` 方法的话是输出地址值

```

System.out.println(s.toString());
System.out.println(s);           //如果直接打印对象的引用，会默认调用toString方法
```

`Object` 中的 `equals` 方法是比较对象的地址值的，没有什么意义，我们需要重写他

因为在开发中我们通常比较的是对象中的属性值，我们认为相同属性是同一个对象，这样我们就需要重写 `equals` 方法

父类引用指向子类对象 不可以访问子类特有的属性

```

public boolean equals(Object obj) {           //s1.equals(s2);
    return this.name.equals(obj.name);
```

报错

需要先转型

```

public boolean equals(Object obj) {           //s1.equals(s2);
    Student s = (Student)obj;
    return this.name.equals(s.name) && this.age == s.age;
```

区别：1、`==`号是比较运算符，既可以比较基本数据类型，也可以比较引用数据类型，基本数据类型比较的是值，引用数据类型比较的是地址值

2、`equals` 方法只能比较的是引用数据类型，`equals` 方法在没重写之前，比较的是地址值，底层依赖的是`==`号，但是比较地址值是没有意义的，我们需要重写 `equals` 方法 比较对象中的属性值

HashCode 比较地址 `equals` 比较值

Scanner

Scanner 其中一个构造是可以接受 `System.in` 流 `nextInt` 接受下一个数

```

Scanner sc = new Scanner(System.in);           //键盘录入
int i = sc.nextInt();                         //键盘录入整数存储在i中
System.out.println(i);
```

`hasNextXxx()` 判断是否还有下一个输入项，其中 `Xxx` 可以是 `Int`, `Double` 等。如果需要判断是否包含下一个字符串，则可以省略 `Xxx`
`nextXxx()` 获取下一个输入项。`Xxx` 的含义和上个方法中的 `Xxx` 相同，默认情况下，`Scanner` 使用空格，回车等作为分隔符

```

System.out.println("请输入第一个字符串:");
String line1 = sc.nextLine();
```

Nextline 和 nextint 在一起用的小问题

```
* nextInt() 是键盘录入整数的方法,当我们录入10的时候
* 其实在键盘上录入的是10和\r\n,nextInt()方法只获取10就结束了
* nextLine() 是键盘录入字符串的方法,可以接收任意类型,但是他凭什么能获取一行呢?
* 通过\r\n,只要遇到\r\n就证明一行结束
*/
System.out.println("请输入第一个整数:");
int i = sc.nextInt();
System.out.println("请输入第二个字符串:");
String line2 = sc.nextLine();
//System.out.println("i = " + i + ", line2 = " + line2);
System.out.println(i);
System.out.print("1111111111");
System.out.print(line2);
System.out.println("22222222222222");
```

解决方案

```
* 1, 创建两次对象,但是浪费空间
* 2, 键盘录入的都是字符串,都用nextLine方法,后面我们会学习讲整数字符串转换成整数的方法
/*
int i = sc.nextInt();
Scanner sc2 = new Scanner(System.in);
String line = sc2.nextLine();
System.out.println(i);
System.out.println(line);
```

String

Str 是一个对象,若一个对象直接打印不是引用地址 代表其已经重写了 toString 方法 string 默认重写 toString

```
String str = "abc"; |I
System.out.println(str);
```

字符串不可变 变的是对象 只不过引用 str 没变 实际指向的对象不同了

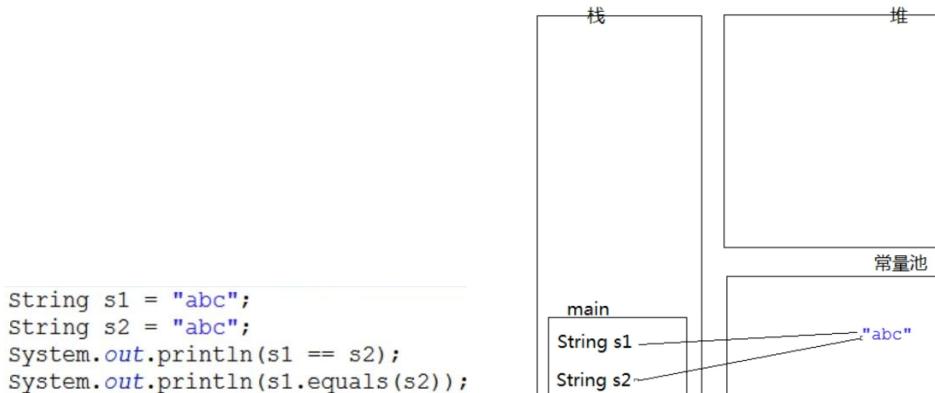
```
String str = "abc";
str = "def";
System.out.println(str); 结果 def
```

String a=New String(); a 为 “”

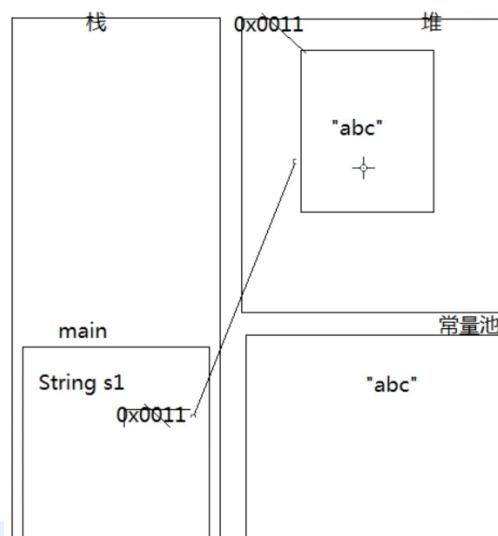
```
byte[] arr1 = {97,98,99};
String s2 = new String(arr1);
System.out.println(s2); 结果 abc 会把字节数组自动转成字符串 成为解码
```

```
char[] arr3 = {'a','b','c','d','e'};
String s4 = new String(arr3);
System.out.println(s4); 结果 abcde 字符数组转成字符串
```

面试题



常量池在方法区中 若一个常量在常量池中有，则不会创建，若没有，则创建 常量池中对象也有各自地址值



首先 main 方法进栈，创建 s1，接着 abc 存入常量池中，之后 new string 创建对象在堆中拥有其地址值，接着复制常量池中的 abc 到堆中的对象里 此对象再将其地址值复制给 s1 此过程共创建 2 个对象(1 堆 1 池)

```

String s1 = new String("abc");
String s2 = "abc";
System.out.println(s1 == s2);           //false
System.out.println(s1.equals(s2));      //true

```

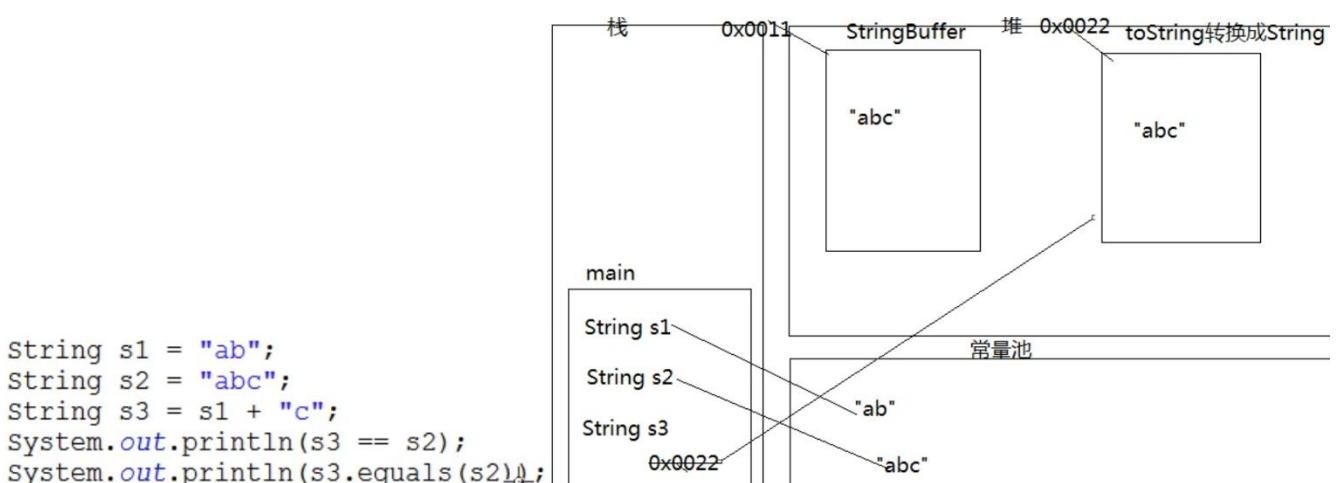
s1 是堆中地址值 s2 是常量池中地址值

```

//byte b = 3 + 4;                      //在编译时就变成7, 把7赋值给b, 常量优化机制
String s1 = "a" + "b" + "c";          //true, java中有常量优化机制
String s2 = "abc";
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));

```

先在池中把 a,b,c 合起来



False true

字符串与其它类型进行+号操作 底层是通过先创建一个 StringBuffer/StringBuilder 对象，调用 append 方法添加进来(相当于转成了对应的字符串，此时还是一个 StringBuffer 对象)，再调用 toString 方法

```
boolean equals(Object obj) : 比较字符串的内容是否相同, 区分大小写  
boolean equalsIgnoreCase(String str) : 比较字符串的内容是否相同, 忽略大小写  
boolean contains(String str) : 判断大字符串中是否包含小字符串  
boolean startsWith(String str) : 判断字符串是否以某个指定的字符串开头  
boolean endsWith(String str) : 判断字符串是否以某个指定的字符串结尾  
boolean isEmpty() : 判断字符串是否为空。
```

"" 和 null 的区别

"" 是字符串常量，同时也是一个 String 类的对象，既然是对象当然可以调用 String 类中的方法

null 是空常量，不能调用任何的方法，否则会出现空指针异常，null 常量可以给任意的引用数据类型赋值 null 的 string 调用 isEmpty() 会报错

String 方法

数组 length 是属性

```
* int length() : 获取字符串的长度。  
char charAt(int index) : 获取指定索引位置的字符  
int indexOf(int ch) : 返回指定字符在此字符串中第一次出现处的索引。  
int indexOf(String str) : 返回指定字符串在此字符串中第一次出现处的索引。  
int indexOf(int ch, int fromIndex) : 返回指定字符在此字符串中从指定位置后第一次出现处的索引。  
int indexOf(String str, int fromIndex) : 返回指定字符串在此字符串中从指定位置后第一次出现处的索引。  
lastIndexOf  
String substring(int start) : 从指定位置开始截取字符串, 默认到末尾。  
String substring(int start, int end) : 从指定位置开始到指定位置结束截取字符串。
```

左闭右开

```
String s = "woaiheima";  
s.substring(4);  
System.out.println(s);
```

结果还是 woaiheima 返回值才变了

遍历字符串

```
String s = "heima";  
  
for(int i = 0; i < s.length(); i++) {  
    char c = s.charAt(i);  
    System.out.println(c);  
}
```

需求：统计一个字符串中大写字母字符，小写字母字符，数字字符出现的次数，其他字符出现的次数。

ABCDEabcd123456!@#\$%^

分析：字符串是有字符组成的，而字符的值都是有范围的，通过范围来判断是否包含该字符

如果包含就让计数器变量自增

```
String s = "ABCDEabcd123456!@#$%^";  
int big = 0;  
int small = 0;  
int num = 0;  
int other = 0;  
// 1. 获取每一个字符，通过 for 循环遍历  
for(int i = 0; i < s.length(); i++) {  
    char c = s.charAt(i); // 通过索引获取每一个字符  
    // 2. 判断字符是否在这个范围内  
    if(c >= 'A' && c <= 'Z') {  
        big++; // 如果满足是大写字母，就让其对应的变量自增  
    } else if(c >= 'a' && c <= 'z') {  
        small++;  
    } else if(c >= '0' && c <= '9') {  
        num++;  
    } else {  
        other++;  
    }  
}
```

```
byte[] getBytes() : 把字符串转换为字节数组。  
char[] toCharArray() : 把字符串转换为字符数组。  
static String valueOf(char[] chs) : 把字符数组转成字符串。  
static String valueOf(int i) : 把 int 类型的数据转成字符串。  
* 注意：String 类的 valueOf 方法可以把任意类型的数据转成字符串。
```

```

String s1 = "abc";
byte[] arr = s1.getBytes();
for (int i = 0; i < arr.length; i++) {
    System.out.print(arr[i] + " ");
}
结果 97 98 99 gbk 码表 一个中文两字节(第一个字节是负数)

char[] arr = {'a','b','c'};
String s = String.valueOf(arr);
System.out.println(s); 结果 abc

```

```

Person p1 = new Person("张三", 23);
System.out.println(p1); 结果都是地址值
String s3 = String.valueOf(p1);
System.out.println(s3); valueof 相当于调用 tostring 方法

```

A: String的替换功能及案例演示

- * String replace(char old, char new)
- * String replace(String old, String new)

B: String的去除字符串两空格及案例演示

- * String trim()

C: String的按字典顺序比较两个字符串及案例演示

- * int compareTo(String str) (暂时不用掌握)
- * int compareToIgnoreCase(String str) (了解)

字符串反转---倒着遍历

```

Scanner sc = new Scanner(System.in);
System.out.println("请输入一个字符串:");
String line = sc.nextLine();

char[] arr = line.toCharArray();

String s = "";
for(int i = arr.length-1; i >= 0; i--) {
    s = s + arr[i];
}

```

统计大串中小串出现次数

```

String max = "woaiheima,heimabutongyubaima,wulunheimahaishibaima,zhaodaogongzuojiushihao";
//定义小串
String min = "heima";

//定义计数器变量
int count = 0;
//定义索引
int index = 0;
//定义循环,判断小串是否在大串中出现
while((index = max.indexOf(min)) != -1) {
    count++; //计数器自增
    max = max.substring(index + min.length());
}

System.out.println(count);

```

Stringbuffer

Stringbuffer 线程安全 可变的字符序列 效率低 Stringbuilder 单线程 可变的字符序列 效率高

A: StringBuffer的构造方法:

- * public StringBuffer():无参构造方法
- * public StringBuffer(int capacity):指定容量的字符串缓冲区对象
- * public StringBuffer(String str):指定字符串内容的字符串缓冲区对象

B: StringBuffer的方法:

- * public int capacity():返回当前容量。 理论值(不掌握)
- * public int length():返回长度(字符数)。实际值 初始容量 16

```

A:StringBuffer的添加功能
* public StringBuffer append(String str):
    * 可以把任意类型数据添加到字符串缓冲区里面，并返回字符串缓冲区本身
* public StringBuffer insert(int offset,String str):
    * 在指定位置把任意类型的数据插入到字符串缓冲区里面，并返回字符串缓冲区本身
StringBuffer是字符串缓冲区，当new的时候是在堆内存创建了一个对象，底层是一个长度为16的字符数组
当调用添加的方法时，不会再重新创建对象，在不断向原缓冲区添加字符】 Stringbuffer 已经重写了 tostring
system.out.println(sb.toString());】当你打印一个东西 无论是什么 都相当于调用它的 tostring 方法
StringBuffer sb = new StringBuffer();
StringBuffer sb2 = sb.append(true);
StringBuffer sb3 = sb.append("heima");
StringBuffer sb4 = sb.append(100);

system.out.println(sb.toString());
system.out.println(sb2.toString());
system.out.println(sb3.toString()); 结果为四个 trueheima100 都为同一对象

```

```

A:StringBuffer的删除功能
* public StringBuffer deleteCharAt(int index):
    * 删除指定位置的字符，并返回本身
* public StringBuffer delete(int start,int end):
    * 删除从指定位置开始指定位置结束的内容，并返回本身
A:StringBuffer的替换功能
* public StringBuffer replace(int start,int end,String str):
    * 从start开始到end用str替换
B:StringBuffer的反转功能
* public StringBuffer reverse():
    * 字符串反转
A:StringBuffer的截取功能
* public String substring(int start):
    * 从指定位置截取到末尾
* public String substring(int start,int end):
    * 截取从指定位置开始到结束位置，包括开始位置，不包括结束位置
注意事项
* 注意：返回值类型不再是StringBuffer本身

```

返回值是 string(新的) 原 stringBuffer 不变

```

String 与 stringBuffer
* A:String -- StringBuffer
    * a:通过构造方法
    * b:通过append()方法
B:StringBuffer -- String
    * a:通过构造方法
    * b:通过toString()方法
    * c:通过subString(0,length);
StringBuffer sb1 = new StringBuffer("heima"); //通过构造方法将字符串转换为StringBuffer对象
String s1 = new String(sb);

```

小陷阱

```

public static void main(String[] args) {
    String s = "heima";
    System.out.println(s);
    change(s);
    System.out.println(s);】
}

public static void change(String s) {
    s += "itcast";】结果还是 heima
}

```

冒泡排序

相邻元素比较

```

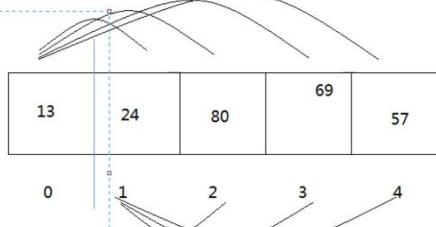
public static void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

```

选择排序

选择排序:用一个索引位置上的元素,依次与其他索引位置上的元素比较

小在前面大的在后面



```

public static void selectSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) { //只需要比较arr.length-1次
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[i] > arr[j]) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

```

二分查找

前提数组有序 返回的是元素在数组中对应的索引

```

public static int getIndex(int[] arr, int value) {
    int min = 0;
    int max = arr.length - 1;
    int mid = (min + max) / 2;

    while (arr[mid] != value) { //当中间值不等于要找的值,就开始循环查找
        if (arr[mid] < value) { //当中间值小于了要找的值
            min = mid + 1; //最小的索引改变
        } else if (arr[mid] > value) { //当中间值大于了要找的值
            max = mid - 1; //最大的索引改变
        }
        mid = (min + max) / 2; //无论最大还是最小改变,中间索引都会随之改变
    }

    if (min > max) { //如果最小索引大于了最大索引,就没有查找的可能性了
        return -1; //返回-1
    }
}
return mid;

```

Array

Array 是一个类 可以直接用这个类的方法

```

public static String toString(int[] a)
public static void sort(int[] a)
public static int binarySearch(int[] a, int key)
int[] arr = {33, 22, 11, 44, 66, 55};
System.out.println(Arrays.toString(arr)); //数组转字符串

Arrays.sort(arr); //排序
System.out.println(Arrays.toString(arr));

int[] arr2 = {11, 22, 33, 44, 55, 66};
System.out.println(Arrays.binarySearch(arr2, 22)); //二分查找

```

包装类

将基本数据类型封装成对象的好处在于可以在对象中定义更多的功能方法操作该数据。
常用操作是用于基本数据类型和字符串间的转换

```

byte          Byte
short         Short
int           Integer
long          Long
float         Float
double        Double
char          Character
boolean       Boolean

```

构造方法

```

* public Integer(int value)
* public Integer(String s)
Integer i1 = new Integer(100);
System.out.println(i1);
System.out.println(Integer.MAX_VALUE);
System.out.println(Integer.MIN_VALUE);

```

```

int -- String
* a:和""进行拼接
* b:public static String valueOf(int i)
* c:int -- String(Integer类的toString方法())
* d:public static String toString(int i) (Integer类的静态方法)
:String -- int
* a:String -- Integer -- int
    * public static int parseInt(String s)           int i5 = Integer.parseInt(s);

```

基本数据类型包装类有八种，其中七种都有parseXxx的方法，可以将这七种的字符串表现形式转换成基本数据类型

char的包装类Character中没有parseXxx的方法

字符串到字符的转换通过toCharArray()就可以把字符串转换为字符数组

自动装箱自动拆箱

A:JDK5的新特性

- * 自动装箱：把基本类型转换为包装类类型
- * 自动拆箱：把包装类类型转换为基本类型

之前：

```

int x = 100;
Integer i1 = new Integer(x);           //将基本数据类型包装成对象，装箱

```

```

int y = i1.intValue();                //将对象转换为基本数据类型，拆箱

```

之后：

```

Integer i2 = 100;                     //自动装箱，把基本数据类型转换成对象
int z = i2 + 200;                    //自动拆箱，把对象转换为基本数据类型
System.out.println(z);

```

在使用时，`Integer x = null;`代码就会出现`NullPointerException`

建议先判断是否为null，然后再使用。

不能用 null 去调用方法 空指针异常 无对象

面试：

```

Integer i1 = new Integer(97);
Integer i2 = new Integer(97);
System.out.println(i1 == i2);           //false
System.out.println(i1.equals(i2));      //true
System.out.println("-----");          integer 已重写 equals 方法

```

```

Integer i5 = 127;
Integer i6 = 127;
System.out.println(i5 == i6);           //true
System.out.println(i5.equals(i6));      //true
System.out.println("-----");

```

```

Integer i7 = 128;
Integer i8 = 128;
System.out.println(i7 == i8);           //true
System.out.println(i7.equals(i8));      //true i7==i8 为 false

```

-128到127是byte的取值范围，如果在这个取值范围内，自动装箱就不会新创建对象，而是从常量池中获取

如果超过了byte取值范围就会再新创建对象

根据 valueof 的源码

正则表达式

* 是指一个用来描述或者匹配一系列符合某个语法规则的字符串的单个字符串。其实也是一种规则。有自己特殊的应用。

* 作用：比如注册邮箱，邮箱有用户名和密码，一般会对其限制长度，这个限制长度的事情就是正则表达式做的

[] 其内包含单个字符

[abc] a、b 或 c (简单类)

[^abc] 任何字符，除了 a、b 或 c (否定)

[a-zA-Z] a 到 z 或 A 到 Z，两头的字母包括在内 (范围)

[a-d[m-p]] a 到 d 或 m 到 p: [a-dm-p] (并集)

[a-z&&[def]] d、e 或 f (交集)

[a-z&&[^bc]] a 到 z，除了 b 和 c: [ad-z] (减去)

[a-z&&[^m-p]] a 到 z，而非 m 到 p: [a-lq-z] (减去)

<https://www.bilibili.com/video/av73935695?p=5>

math 类

```
public static int abs(int a)
public static double ceil(double a)
public static double floor(double a)
public static int max(int a,int b) min自学
public static double pow(double a,double b)
public static double random()
public static int round(float a) 参数为double的自学
public static double sqrt(double a)
```

System 类

```
* public static void gc()
* public static void exit(int status)
* public static long currentTimeMillis()
* public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

Object 中有 finalize 方法 就是如果你变成垃圾了，垃圾回收器调用此方法，回收此对象或者你可以用 gc 手动调用垃圾回收器回收此对象

protected void finalize() 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

调用垃圾回收器方法

System.gc();

退出 java 虚拟机 system.exit() 非 0 状态是异常终止

Arraycopy 用 b 复制数组 a

Biginteger 类

A:BigInteger的概述

* 可以让超过Integer范围内的数据进行运算

* B:构造方法

* public BigInteger(String val)

* C:成员方法

* public BigInteger add(BigInteger val)

* public BigInteger subtract(BigInteger val)

* public BigInteger multiply(BigInteger val)

* public BigInteger divide(BigInteger val)

* public BigInteger[] divideAndRemainder(BigInteger val)取商取余

BigDecimal 类

A:BigDecimal的概述

- * 由于在运算的时候，float类型和double很容易丢失精度，演示案例。
- * 所以，为了能精确的表示、计算浮点数，Java提供了BigDecimal

* 不可变的、任意精度的有符号十进制数。

B:构造方法

* public BigDecimal(String val)

C:成员方法

```
* public BigDecimal add(BigDecimal augend)
* public BigDecimal subtract(BigDecimal subtrahend)
* public BigDecimal multiply(BigDecimal multiplicand)
* public BigDecimal divide(BigDecimal divisor)
```

D:案例演示

* BigDecimal类的构造方法和成员方法使用

例如你直接运算 2.0-1.1 结果为 0.89999

```
/*BigDecimal bd1 = new BigDecimal("2.0");           //通过构造中传入字符串的方式，开发时推荐
BigDecimal bd2 = new BigDecimal("1.1");
System.out.println(bd1.subtract(bd2));*/
```

0.9

Date 类 现已由 calendar 类代替

* 类 Date 表示特定的瞬间，精确到毫秒。

B:构造方法

```
* public Date()
* public Date(long date)
```

C:成员方法

```
* public long getTime()
* public void setTime(long time)    gettime 得到当前时间毫秒值(与 1970 年对比)
```

A:DateFormat类的概述

* DateFormat 是日期/时间格式化子类的抽象类，它以与语言无关的方式格式化并解析日期或时间。是抽象类，所以使用其子类实现

B:SimpleDateFormat构造方法

```
* public SimpleDateFormat()
* public SimpleDateFormat(String pattern)
```

C:成员方法

```
* public final String format(Date date)
* public Date parse(String source)
```

```
Date d = new Date();                                //获取当前时间对象
SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss"); //创建日期格式化类对象
System.out.println(sdf.format(d));
```

//将时间字符串转换成日期对象

```
String str = "2000年08月08日 08:08:08";
SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");
Date d = sdf.parse(str);                            //将时间字符串转换成日期对象
```

* * A:案例演示

* 需求：算一下你来到这个世界多少天？

* 分析：

* 1, 将生日字符串和今天字符串存在String类型的变量中

* 2, 定义日期格式化对象

* 3, 将日期字符串转换成日期对象

* 4, 通过日期对象后期时间毫秒值

* 5, 将两个时间毫秒值相减除以1000, 再除以60, 再除以60, 再除以24得到天

```

//1,将生日字符串和今天字符串存在String类型的变量中
String birthday = "1983年07月08日";
String today = "2088年6月6日";
//2,定义日期格式化对象
SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日");
//3,将日期字符串转换成日期对象
Date d1 = sdf.parse(birthday);
Date d2 = sdf.parse(today);
//4,通过日期对象后期时间毫秒值
long time = d2.getTime() - d1.getTime();

```

需求：键盘录入任意一个年份，判断该年是闰年还是平年
`Calendar c = Calendar.getInstance();`

分析：

- 1, 键盘录入年Scanner
- 2, 创建Calendar c = Calendar.getInstance()
- 3, 通过set方法设置为那一年的3月1日
- 4, 将日向前减去1
- 5, 判断日是多少天,如果是29天返回true否则返回false

```

String line = sc.nextLine();
int year = Integer.parseInt(line);           //录入数字字符串
boolean b = getYear(year);                  //将数字字符串转换成数字

private static boolean getYear(int year) {
    //2,创建Calendar c = Calendar.getInstance();
    Calendar c = Calendar.getInstance();
    //设置为那一年的3月1日
    c.set(year, 2, 1);
    //将日向前减去1
    c.add(Calendar.DAY_OF_MONTH, -1);
    //判断是否是29天
    return c.get(Calendar.DAY_OF_MONTH) == 29;
}

```

数组

数组可以储存 引用数据类型 和 基本数据类型

```

package com.heima.collection;

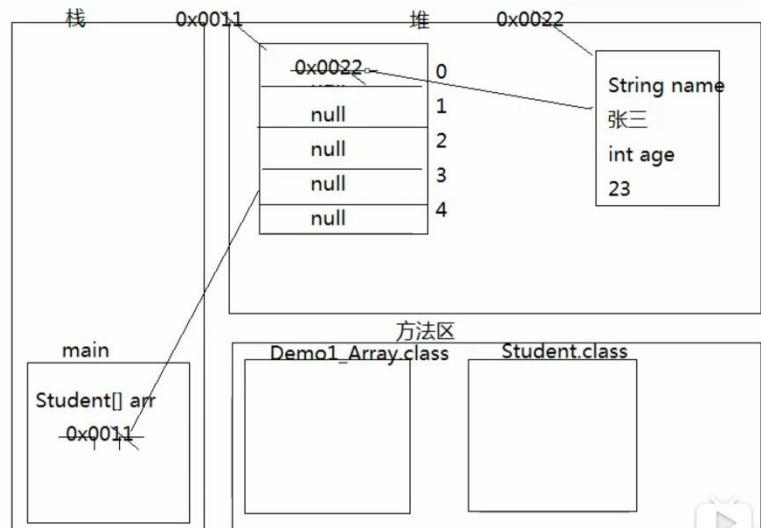
import com.heima.bean.Student;

public class Demo1_Array {

    public static void main(String[] args) {
        Student[] arr = new Student[5];
        arr[0] = new Student("张三", 23);
        arr[1] = new Student("李四", 24);
        arr[2] = new Student("王五", 25);

        for(int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
}

```



Demo.class 加载进内存—方法区

main 主方法进栈 发现 student 类---student 类加载进内存--方法区

student[]arr 引用在栈中出现 堆中出现相应的 arr 对象(数组需要连续的空间), 并具有 5 个数组索引, 5 个元素赋予初始值 null, 把栈中的 arr 赋值真正的数组对象索引

arr[0]=new student("Tom",23) 数组元素位置存储的不是真正的对象, 而是对象的地址值
堆中会出现一个 student 对象, 具有地址值, 并把地址值赋予数组对应的元素

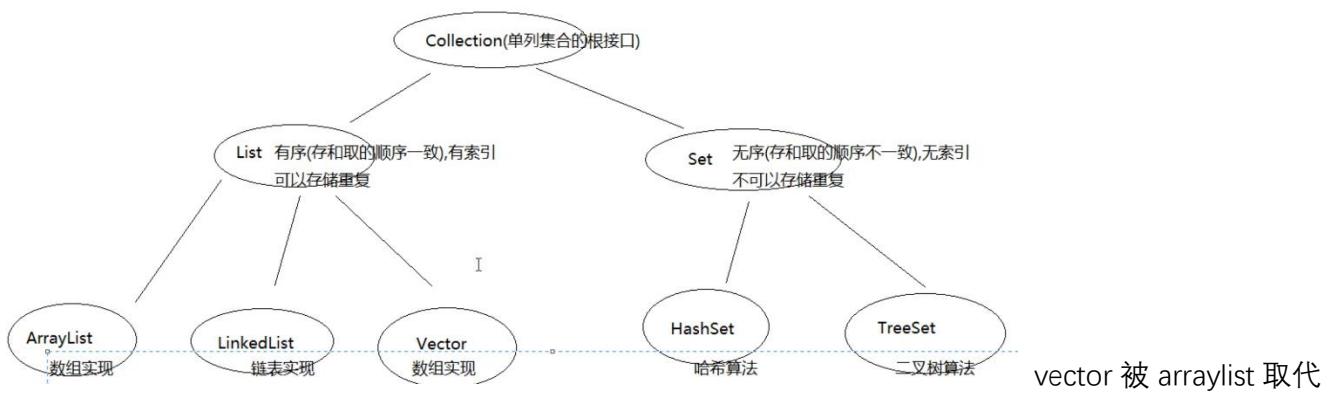
数组中若是基本数据类型, 默认初始值都是 0, 被修改后都是具体值, 若是引用数据类型, 默认都是 null, 被修改后都是各自对象的地址值

集合类

与数组不同, 数组是固定大小, 而集合类是可以扩展的, 长度可变

数组可储存基本数据类型(值)和引用数据类型(地址值), 而集合只能储存引用数据类型---对象

其实你存 100 进去也不会报错, 集合会把基本数据类型自动装箱变成对象



Collection

list 有序 有索引 可重复

arraylist 数组实现 linkedlist 链表实现

collection a=new arrayList(); 父类引用指向子类对象

boolean b=a.add("c") 编译检查的时候 collection 里的 add 方法实际运行的是 arrayList 里的 add 方法

Collection 方法(list 和 set 共有方法)

```

boolean add(E e)
boolean remove(Object o)
void clear()
boolean contains(Object o)
boolean isEmpty()
int size()

```

注意 add 方法添加的都是 object 类型，被自动提升/装箱，后续使用时需要向下转型或拆箱

集合可以通过转数组来进行遍历

```

collection a=new arrayList();
object [] arr=c.toArray();
然后 for 循环 arr [ i ]

```

若数组中传入的是一个 student 类 此时不能输出 System.out.println(arr[i].getName());
 因为 c.add(new Student("张三", 23)); 添加的都是 object 类对象 Object[] arr = c.toArray();
 此时为父类引用指向子类对象 调用子类方法时会编译检查父类中有无同名方法 结果当然没有 报错
 若想使用子类特有方法--向下转型 Student s = (Student)arr[i];

Collection 里的 all 方法 添加/删除整个集合

```

boolean addAll(Collection c)
boolean removeAll(Collection c)
boolean containsAll(Collection c)
boolean retainAll(Collection c)    retainall 取交集 如果调用的集合改变则 true, 未改变返回 false

```

迭代器(遍历) collection 所有的子类(list/set)都可以用迭代器遍历

```

Collection c = new ArrayList();
Iterator it = c.iterator();         获取迭代器
while(it.hasNext()) {
    System.out.println(it.next());
}

```

hasnext 判断集合中是否有元素 有则返回 true it.next 返回此元素

若集合中传入的是自定义类型，想调用对象方法的时候同样需要向下转型，因为你传入的时候自动变成 object 类了，不想调用它方法也可以，重写 tostring 方法直接输出

```

Iterator it = c.iterator();
while(it.hasNext()) {
    //System.out.println(it.next());
    Student s = (Student)it.next();           //向下转型
    System.out.println(s.getName() + "..." + s.getAge());
}

```

A:迭代器原理
 * 迭代器原理：迭代器是对集合进行遍历，而每一个集合内部的存储结构都是不同的，所以每一个集合存和取都是不一样，那么就需要在每一个类中定义hasNext()和next()方法，这样做是可以的，但是会让整个集合体系过于臃肿，迭代器是将这样的方法向上抽取出接口，然后在每个类的内部，定义自己迭代方式，这样做的好处有二，第一规定了整个集合体系的遍历方式都是hasNext()和next()方法，第二，代码有底层内部实现，使用者不用管怎么实现的，会用即可

B:迭代器源码解析
 * 1,在eclipse中ctrl + shift + t找到ArrayList类
 * 2,ctrl+t查找iterator()方法
 * 3,查看返回值类型是new Itr(),说明Itr这个类实现Iterator接口
 * 4,查找Itr这个内部类，发现重写了Iterator中的所有抽象方法

List

list 有序 有索引 可重复

List list=new ArrayList();

父类引用指向子类对象 调用 list 的特有方法 编译时检查 list 接口方法，运行 ArrayList 的方法，缺点是不能访问 ArrayList 的特有属性和方法，所以开发时一般直接 ArrayList a=new ArrayList();
 多态最大的好处是可以当参数传递，可以接收任意子类对象

```
public static void method(Animal a) { //当作参数的时候用多态最好，因为扩展性强}
```

List 特有方法

```

void add(int index, E element)
E remove(int index)
E get(int index)
E set(int index, E element)      index 为 list 中索引  注 remove 方法传入 11 不会自动装箱，当作索引

```

并发修改异常 迭代器循环时集合被修改，报错 string 别忘向下转型

```

* 需求：我有一个集合，请问，我想判断里面有没有"world"这个元素，如果有，我就添加一个"javaee"元素，请写代码实现。
*/
public static void main(String[] args) {
    List list = new ArrayList();
    list.add("a");
    list.add("b");
    list.add("world");
    list.add("c");
    list.add("d");
    list.add("e");

    /*Iterator it = list.iterator();
    while(it.hasNext()) {
        String str = (String)it.next();
        if("world".equals(str)) {
            list.add("javaee");
        }
    }
    */

```

解决方法 使用 listIterator 迭代器 并使用它的 add 方法

```

ListIterator lit = list.listIterator();
while(lit.hasNext()) {
    String str = (String)lit.next();
    if("world".equals(str)) {
        //list.add("javaee");
        lit.add("javaee");
    }
}

```

Vector

Vector 底层数组实现，被 ArrayList 取代 线程安全 效率低 遍历通过 enumeration Vector vector=new Vector();

数组查修快，增删慢 链表增删快，查修慢

Vector和ArrayList的区别

Vector是线程安全的，效率低

ArrayList是线程不安全的，效率高

共同点：都是数组实现的

ArrayList和LinkedList的区别

ArrayList底层是数组结果，查询和修改快

LinkedList底层是链表结构的，增和删比较快，查询和修改比较慢

共同点：都是线程不安全的

ArrayList

需求：ArrayList去除集合中字符串的重复值

```
* 1, 创建新集合  
* 2, 根据传入的集合(老集合)获取迭代器  
* 3, 遍历老集合  
* 4, 通过新集合判断是否包含老集合中的元素, 如果包含就不添加, 如果不包含就添加  
*/  
public static ArrayList getSingle(ArrayList list) {  
    ArrayList newList = new ArrayList<>();  
    Iterator it = list.iterator();  
  
    while(it.hasNext()) {  
        Object obj = it.next();  
        if(!newList.contains(obj)) {  
            newList.add(obj);  
        }  
    }  
    return newList;  
}
```

Remove 和 Contain 方法底层靠 equals 实现，源码是 ==，比较的是类的地址值，所以要比较实际内容是否相等的话，需要自定义类中重写 equals 方法

```
public boolean equals(Object obj) {  
    Person p = (Person) obj;  
    return this.name.equals(p.name) && this.age == p.age;
```

对象用 equals 基本类型用 ==

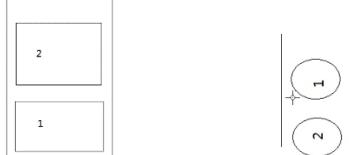
LinkedList

LinkedList 特有方法

```
public void addFirst(E e) 及 addLast(E e)  
* public E getFirst() 及 getLast()  
* public E removeFirst() 及 public E removeLast()  
* public E get(int index);
```

get 方法源码：传入一个 index，先判断 index 与链表长度/2 的大小，大于则从后往前一个一个找，小于则从前向后一个一个找 表面看起来快速简单 但底层实现很慢

栈：先进后出 队列：先进先出 队列恶心



用 linkedlist 模拟栈

```
public class Stack {  
    private LinkedList list = new LinkedList(); 表面 new 个 stack 对象 实际创建的是 linkedlist 对象  
    public void in(Object obj) {  
        list.addLast(obj); 模拟进栈  
    }  
    public Object out() {  
        return list.removeLast(); 模拟出栈  
    }  
    public boolean isEmpty() {  
        return list.isEmpty(); 模拟栈结构是否为空  
    }  
}
```

泛型

提高安全性 无需向下转型了 <> 里放的必须是引用数据类型 --- 必须是对象

无泛型时 add 的是 object 类型，使用需要强转 Person p = (Person) it.next();

有泛型后 ArrayList<Person> list = **new** ArrayList<Person>();

Add 方法自动把接受 object 类型变为接受相应泛型，Add 和迭代器都只是对应类型，不需要强转了

```

public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
    list.add("a");
    list.add("b");
    list.add("c");
    list.add("d");
}
Iterator<String> it = list.iterator();
while(it.hasNext()) {
    System.out.println(it.next());
}

```

```

Iterator<Person> it = list.iterator();
while(it.hasNext()) {
    //System.out.println(it.next());
    Person p = it.next();           //向下转型
    System.out.println(p.getName() + "..." + p.getAge());
}

```

不能用两个 next，那就不是同一个对象了，需要先用 p 记录

泛型声明

```

public class Tool<Q> {
    private Q q;

    public Q getObj() {
        return q;
    }

    public void setObj(Q q) {
        this.q = q;
    }

    public<T> void show(T t) {           //方法泛型最好与类的泛型一致
        System.out.println(t);           //如果不一致，需要在方法上声明该泛型
    }
}

```

静态方法随着类的加载而加载 而类上的泛型指类实例化对象后接受的泛型 两者不同

```
public static<Q> void print(Q q) {           //静态方法必须声明自己的泛型
```

泛型接口

把泛型定义到接口上 `public interface 接口名<泛型类型>`

```

interface Inter<T> {
    public void show(T t);
}

class Demo implements Inter<String> {

    @Override
    public void show(String t) {
        System.out.println(t);
    }
}

```

泛型通配符

A:泛型通配符<?>

* 任意类型，如果没有明确，那么就是Object以及任意的Java类了

B:?: extends E

* 向下限定，E及其子类

C:?: super E

* 向上限定，E及其父类

`List<?> list = new ArrayList<Integer>();`

当右边泛型不确定时，你可以指定左边为？ 例如你还可以 `new arraylist<string>();`

```

ArrayList<Person> list1 = new ArrayList<>();
list1.add(new Person("张三", 23));
list1.add(new Person("李四", 24));
list1.add(new Person("王五", 25));

```

```

ArrayList<Student> list2 = new ArrayList<>();
list2.add(new Student("赵六", 26));
list2.add(new Student("周七", 27));

```

```

list1.addAll(list2);
System.out.println(list1);

```

list2 中 student 会自动向上转型变换成 person，相当于父类引用指向子类对象 `list2.addAll(list1)` 则会报错
`addAll` 中的是`<? extends E>`泛型

foreach

`for(int i : arr) {` 用于数组和 collection 集合的遍历

底层依赖的是迭代器 iterator

集合删除

```
//1,普通for循环删除,索引要--  
for(int i = 0; i < list.size(); i++) {  
    if("b".equals(list.get(i))) {  
        list.remove(i--);  
    }  
}  
若不i--集合中连续的两个b元素会只删除一个  
//2,迭代器删除  
Iterator<String> it = list.iterator();  
while(it.hasNext()) {  
    if("b".equals(it.next())) {  
        //list.remove("b");  
        it.remove();  
    }  
}  
System.out.println(list);  
不能用集合的remove，并发修改异常，需要迭代器自身的方法  
//3,增强for循环,增强for循环不能删除,只能遍历  
for (String string : list) {  
    if("b".equals(string)) {  
        list.remove("b");  
    }  
}  
并发修改异常
```

可变参数

可变参数概述

* 定义方法的时候不知道该定义多少个参数

格式

* 修饰符 返回值类型 方法名(数据类型... 变量名) {}

注意事项：

* 这里的变量其实是一个数组

* 如果一个方法有可变参数，并且有多个参数，那么，可变参数肯定是最后一个

```
/*public static void print(int ... arr) {  
    for (int i = 0; i < arr.length; i++) {  
        System.out.println(arr[i]);  
        print(11,22,33,44,55);  
        System.out.println("-----");  
        print();  
    }  
}
```

接收0-无穷参数

public static void print(int x, int ... arr) { 此时你至少接收一个int参数}

数组集合互转

数组转换成集合虽然不能增加或减少元素，但是可以用集合的思想操作数组，也就是说可以使用其他集合中的方法

```
String[] arr = {"a", "b", "c"};  
List<String> list = Arrays.asList(arr);  
System.out.println(list); //将数组转换成集合
```

list.add("s")不可以，报错

```
int[] arr = {11, 22, 33, 44, 55};  
List list = Arrays.asList(arr);  
System.out.println(list);
```

相当于List<int> list = Arrays.asList(arr);

输出的是地址值，把arr当成一个对象存入list，因为list只能接受引用数据类型，当你传入的是基本数据类型的数组，所以把数组当成一个整体存入list的第一个元素，输出的是地址值，string可以是因为string内的所有元素都是引用数据类型，可以依次输出

```
Integer[] arr = {11, 22, 33, 44, 55};  
List<Integer> list = Arrays.asList(arr);  
System.out.println(list); //将数组转换成集合，数组必须是引用数据类型
```

这样才行

集合转数组

```
String[] arr = list.toArray(new String[0]);
```

0可变 若多余list长度多出来的会打印出null，所以设0就好，反正长度自动增加

ArrayList嵌套

学科(班级(人))

```
ArrayList<ArrayList<Person>> list = new ArrayList<>();

ArrayList<Person> first = new ArrayList<>();
first.add(new Person("杨幂", 30));
first.add(new Person("李冰冰", 33));
first.add(new Person("范冰冰", 20));

ArrayList<Person> second = new ArrayList<>();
second.add(new Person("黄晓明", 31));
second.add(new Person("赵薇", 33));
second.add(new Person("陈坤", 32));

//将班级添加到学科集合中
list.add(first);
list.add(second);

//遍历学科集合
for(ArrayList<Person> a : list) {
    for(Person p : a) {
        System.out.println(p);
    }
}
```

Collections 工具类

Collections中的常见方法

```
public static <T> void sort(List<T> list)
public static <T> int binarySearch(List<?> list, T key)
public static <T> T max(Collection<?> coll)
public static void reverse(List<?> list)
public static void shuffle(List<?> list) shuffle 随机置换
```

当一个类中所有方法都是静态的，它就会私有它的构造方法，目的是不让其它类创建本类对象，直接通过类名，来调用

```
ArrayList<String> list = new ArrayList<>();  
Collections.sort(list);          集合排序(前提使集合中元素具备可排序性)
```

Set

无序，无索引，不可重复

只能用 foreach 或迭代器遍历(无索引不能用 for 遍历, 只要迭代器可用, foreach 就可用)

Hashset

若想保证存入的自定义类唯一，需要将自定义类重写 equals() 和 hashCode() 方法

当插入一个对象时，需要使用 hashCode 算出一个值分配给它，不同的对象拥有不同的 hash 值，就无需调用 equals 方法作比较，无法保证唯一，所以需要重写 hashCode 方法返回同一值进而调用 equals 方法进行比较

```
HashSet<Person> hs = new HashSet<Person>();  
hs.add(new Person("张三", 23));  
hs.add(new Person("张三", 23));  
hs.add(new Person("李四", 24));  
hs.add(new Person("李四", 24));  
hs.add(new Person("李四", 24));
```

若 hash 都相同，效率低，退化成链表(同 hash 值而 equals 不同则会调用 add 方法添加)
hash 值都不同，无法保证数值唯一性

通常重写 hashCode 的方法

```
    所以重写 hashCode 方法时重校对类的属性值相同的对象用 equals 方法进行判  
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + age;  
    result = prime * result + ((name == null) ? 0 : name.hashCode());  
    return result;  
}
```

String 已经重写了 hashCode，返回字段的哈希值(同字段相同)

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    Person other = (Person) obj;  
    if (age != other.age)  
        return false;  
    if (name == null) {  
        if (other.name != null)  
            return false;  
    } else if (!name.equals(other.name))  
        return false;  
    return true;  
}
```

重写 equals

HashSet 原理

- * 我们使用 Set 集合都是需要去掉重复元素的，如果在存储的时候逐个 equals() 比较，效率较低，哈希算法提高了去重复的效率，降低了使用 equals() 方法的次数
- * 当 HashSet 调用 add() 方法存储对象的时候，先调用对象的 hashCode() 方法得到一个哈希值，然后在集合中查找是否有哈希值相同的对象
 - * 如果没有哈希值相同的对象就直接存入集合
 - * 如果有哈希值相同的对象，就和哈希值相同的对象逐个进行 equals() 比较，比较结果为 false 就存入，true 则不存
- 将自定义类的对象存入 HashSet 去重复
- * 类中必须重写 hashCode() 和 equals() 方法
- * hashCode(): 属性相同的对象返回值必须相同，属性不同的返回值尽量不同（提高效率）
- * equals(): 属性相同返回 true，属性不同返回 false，返回 false 的时候存储

所以不重写 hashCode，存入值无唯一性 不重写写 equals，存入值也无唯一性

LinkedHashSet

底层是链表实现的，是 set 集合中唯一一个能保证怎么存就怎么取的集合对象

因为是 HashSet 的子类，所以也是保证元素唯一的，与 HashSet 的原理一样

比普通 HashSet 保证了怎么存怎么取---顺序一致

例 1：

需求：编写一个程序，获取 10 个 1 至 20 的随机数，要求随机数不能重复。并把最终的随机数输出到控制台。

分析：

```
1, 有 Random 类创建随机数对象  
2, 需要存储 10 个随机数，而且不能重复，所以我们用 HashSet 集合  
3, 如果 HashSet 的 size 是小于 10 就可以不断的存储，如果大于等于 10 就停止存储  
4, 通过 Random 类中的 nextInt(n) 方法获取 1 到 20 之间的随机数，并将这些随机数存储在 HashSet 集合中  
5, 遍历 HashSet  
//1, 有 Random 类创建随机数对象  
Random r = new Random();  
//2, 需要存储 10 个随机数，而且不能重复，所以我们用 HashSet 集合  
HashSet<Integer> hs = new HashSet<>();  
//3, 如果 HashSet 的 size 是小于 10 就可以不断的存储，如果大于等于 10 就停止存储  
while (hs.size() < 10) {  
    //4, 通过 Random 类中的 nextInt(n) 方法获取 1 到 20 之间的随机数，并将这些随机数存储在 HashSet 集合中  
    hs.add(r.nextInt(20) + 1);  
}  
// 5, 遍历 HashSet  
for (Integer integer : hs) {  
    System.out.println(integer);  
}
```

例 2：

```
* 分析  
* 去除 List 集合中的重复元素  
* 1, 创建一个 LinkedHashSet 集合  
* 2, 将 List 集合中所有的元素添加到 LinkedHashSet 集合  
* 3, 将 list 集合中的元素清除  
* 4, 将 LinkedHashSet 集合中的元素添加回 List 集合中  
*/  
public static void getSingle(List<String> list) {  
    //1, 创建一个 LinkedHashSet 集合  
    LinkedHashSet<String> lhs = new LinkedHashSet<>();  
    //2, 将 List 集合中所有的元素添加到 LinkedHashSet 集合  
    lhs.addAll(list);  
    //3, 将 list 集合中的元素清除  
    list.clear();  
    //4, 将 LinkedHashSet 集合中的元素添加回 List 集合中  
    list.addAll(lhs);  
}
```

Treeset

用来对元素进行排序，同样保证元素唯一

底层是二叉树(两个叉，小的存在左边(负)，大的存在右边(正)，相等不存(0))---compareto 方法

若想对自定义类用 treeset 存储并排序，需要继承 comparable 接口并重写其内的 compareto 方法 (string 类已重写---按首字符的 ascii 码比较)

当 compareTo 方法返回 0 的时候集合中只有一个元素

当 compareTo 方法返回正数的时候集合会怎么存就怎么取

当 compareTo 方法返回负数的时候集合会倒序存储

若想对 person 类中的年龄排序 重写 person 类的 compareto 方法

```
TreeSet<Person> ts = new TreeSet<>();  
ts.add(new Person("张三", 23));  
ts.add(new Person("李四", 13));  
ts.add(new Person("王五", 43));  
public int compareTo(Person o) {  
    int num = this.age - o.age; //年龄是比较的主要条件  
    return num == 0 ? this.name.compareTo(o.name) : num; //姓名是比较的次要条件  
    A ? B : C //如果 A 为真执行 B 否则执行 C
```

按姓名排序

```
public int compareTo(Person o) {  
    int num = this.name.compareTo(o.name); //姓名是主要条件  
    return num == 0 ? this.age - o.age : num; //年龄是次要条件
```

System.out.println('赵' + 0); 字符与数字相加会输出字符所对应的 ascii 码表值

Treeset 的比较器

当元素自身不具备比较性，或者具备的比较性不是所需要的，这时需要让容器自身具备比较性。

定义比较器，将比较器对象作为参数传递给 TreeSet 集合的构造函数，当两种排序都存在时，以比较器为主

定义一个类，实现 Comparator 接口，覆盖 compare 方法。

Comparator 接口内有 compare 和 equals 两个方法，无需重写 equals-CBL 类继承 object 类已经重写了

```
TreeSet<String> ts = new TreeSet<>(new CompareByLen());  
ts.add("aaaaaaaa");  
ts.add("z");  
ts.add("wc");  
T 传入一个子类对象即可 new CBL
```

```
class CompareByLen /*extends Object*/ implements Comparator<String> {  
  
    @Override  
    public int compare(String s1, String s2) { //按照字符串的长度比较  
        int num = s1.length() - s2.length(); //长度为主要条件  
        return num == 0 ? s1.compareTo(s2) : num; //内容为次要条件
```

a. 需要类实现 comparable 接口并重写 compareto，add 时调用此对象的 compareto 与集合中元素作比较

a. 自然顺序(Comparable)

- * TreeSet 类的 add() 方法中会把存入的对象提升为 Comparable 类型
- * 调用对象的 compareTo() 方法和集合中的对象比较
- * 根据 compareTo() 方法返回的结果进行存储

b. 比较器顺序(Comparator)

- * 创建 TreeSet 的时候可以制定一个 Comparator
- * 如果传入了 Comparator 的子类对象，那么 TreeSet 就会按照比较器中的顺序排序
- * add() 方法内部会自动调用 Comparator 接口中 compare() 方法排序
- * 调用的对象是 compare 方法的第一个参数，集合中的对象是 compare 方法的第二个参数

c. 两种方式的区别

- * TreeSet 构造函数什么都不传，默认按照类中 Comparable 的顺序(没有就报错 ClassCastException)
- * TreeSet 如果传入 Comparator，就优先按照 Comparator

如 string 和比较器都有各自规则，按比较器的进行比较

练习

看清需求---无重复(set)---有序(treeset)---存取一致(linkedhashset)

例 1：

在一个集合中存储了无序并且重复的字符串，定义一个方法，让其有序(字典顺序)，而且不能去除重复

分析：

- 1, 定义一个 List 集合，并存储重复的无序的字符串
- 2, 定义方法对其排序保留重复
- 3, 打印 List 集合

```

ArrayList<String> list = new ArrayList<>(); * 定义方法,排序并保留重复
list.add("aaa");
list.add("aaa");
list.add("ccc");
list.add("ddd");
list.add("ffffffffff");
list.add("heima");
list.add("itcast");
list.add("bbb");
list.add("aaa");
list.add("aaa");
//2,定义方法对其排序保留重复
sort(list);
//3,打印list
System.out.println(list);

```

```

public static void sort(List<String> list) {
    //1,创建TreeSet集合对象,因为String本身就具备比较功能,但是重复不会保留,所以我们用比较器
    TreeSet<String> ts = new TreeSet<>(new Comparator<String>() {
        @Override
        public int compare(String s1, String s2) {
            int num = s1.compareTo(s2);
            return num == 0 ? 1 : num;
        }
    });
    //2,将list集合中所有的元素添加到TreeSet集合中,对其进行排序,保留重复
    ts.addAll(list);
    //3,清空list集合
    list.clear();
    //4,将TreeSet集合中排序好的元素添加到list中
    list.addAll(ts);
}

```

匿名内部类

实现一个接口 整个代表此接口的子类对象

例 2:

从键盘接收一个字符串， 程序对其中所有字符进行排序，例如键盘输入： helloitcast 程序打印： acehillostt

分析：

- 1, 键盘录入字符串, Scanner
- 2, 将字符串转换为字符数组
- 3, 定义TreeSet集合, 传入比较器对字符排序并保留重复
- 4, 遍历字符数组, 将每一个字符存储在TreeSet集合中
- 5, 遍历TreeSet集合, 打印每一个字符

```

//2,将字符串转换为字符数组
char[] arr = line.toCharArray();
//3,定义TreeSet集合,传入比较器对字符排序并保留重复
TreeSet<Character> ts = new TreeSet<>(new Comparator<Character>() {

```

```

    @Override
    public int compare(Character c1, Character c2) {
        int num = c1 - c2;           //自动拆箱
        return num == 0 ? 1 : num;
    }
});

//4,遍历字符数组,将每一个字符存储在TreeSet集合中
for(char c : arr) {
    ts.add(c);                  //自动装箱
}

//5,遍历TreeSet集合,打印每一个字符
for(Character c : ts) {
    System.out.print(c);
}

```

例 3:

程序启动后， 可以从键盘输入接收多个整数， 直到输入quit时结束输入。 把所有输入的整数倒序排列打印。

- 1, 创建Scanner对象, 键盘录入
- 2, 创建TreeSet集合对象, TreeSet集合中传入比较器

- 3, 无限循环不断接收整数, 遇到quit退出, 因为退出是quit, 所以键盘录入的时候应该都以字符串的形式录入
- 4, 判断是quit就退出, 不是将其转换为Integer, 并添加到集合中

- 5, 遍历TreeSet集合并打印每一个元素

```

public static void main(String[] args) {
    //1,创建Scanner对象, 键盘录入
    Scanner sc = new Scanner(System.in);
    //2,创建TreeSet集合对象, TreeSet集合中传入比较器
    TreeSet<Integer> ts = new TreeSet<>(new Comparator<Integer>() {

```

```

        @Override
        public int compare(Integer i1, Integer i2) {
            //int num = i2 - i1;           //自动拆箱
            int num = i2.compareTo(i1);
            return num == 0 ? 1 : num;
        }
    });
}

```

```

// 3,无限循环不断接收整数,遇到quit退出,因为退出是quit,所以键盘录入的时候应该都以字符串的形式录入
while(true) {
    String line = sc.nextLine(); // 将键盘录入的字符串存储在line中
    if("quit".equals(line)) {
        break;
    }
    // 4,判断是quit就退出,不是将其转换为Integer,并添加到集合中
    Integer i = Integer.parseInt(line);
    ts.add(i);
}

// 5,遍历TreeSet集合并打印每一个元素
for (Integer integer : ts) {
    System.out.println(integer);
}

```

字符串和常量比较,字符串放前面,不会出现空指针异常,变量可能是 null

例 4:

需求: 键盘录入5个学生信息(姓名,语文成绩,数学成绩,英语成绩),按照总分从高到低输出到控制台。

分析:

1, 定义一个学生类

成员变量:姓名,语文成绩,数学成绩,英语成绩,总成绩
 成员方法:空参,有参构造,有参构造的参数分别是姓名,语文成绩,数学成绩,英语成绩
`toString`方法,在遍历集合中的Student对象打印对象引用的时候会显示属性值

2, 键盘录入需要Scanner,创建键盘录入对象

3, 创建TreeSet集合对象,在TreeSet的构造函数中传入比较器,按照总分比较

4, 录入五个学生,所以以集合中的学生个数为判断条件,如果size是小于5就进行存储

5, 将录入的字符串切割,用逗号切割,会返回一个字符串数组,将字符串数组中从二个元素转换成int数,
 6, 将转换后的结果封装成Student对象,将Student添加到TreeSet集合中

7, 遍历TreeSet集合打印每一个Student对象

Student 类

```

private String name;
private int chinese;
private int math;
private int english;
private int sum;

public Student() {
    super();
}

public Student(String name, int chinese, int math, int english) {
    super();
    this.name = name;
    this.chinese = chinese;
    this.math = math;
    this.english = english;
    this.sum = this.chinese + this.math + this.english;
}

```

Main

```

// 2, 键盘录入需要Scanner, 创建键盘录入对象
Scanner sc = new Scanner(System.in);
System.out.println("请输入学生成绩格式是:姓名,语文成绩,数学成绩,英语成绩");
// 3, 创建TreeSet集合对象,在TreeSet的构造函数中传入比较器,按照总分比较
TreeSet<Student> ts = new TreeSet<>(new Comparator<Student>() {

    @Override
    public int compare(Student s1, Student s2) {
        int num = s2.getSum() - s1.getSum();
        return num == 0 ? 1 : num;
    }
});
```

```

//4, 录入五个学生, 所以以集合中的学生个数为判断条件, 如果size是小于5就进行存储
while(ts.size() < 5) {
    //5, 将录入的字符串切割, 用逗号切割, 会返回一个字符串数组, 将字符串数组中从二个元素转换成int数,
    String line = sc.nextLine();
    String[] arr = line.split(",");
    int chinese = Integer.parseInt(arr[1]);
    int math = Integer.parseInt(arr[2]);
    int english = Integer.parseInt(arr[3]);
    //6, 将转换后的结果封装成Student对象, 将Student添加到TreeSet集合中
    ts.add(new Student(arr[0], chinese, math, english));
}

//7, 遍历TreeSet集合打印每一个Student对象
System.out.println("排序后的学生信息:");
for (Student s : ts) {
    System.out.println(s);
}

```

Map

Map 接口特点: 无序, 键不可重复

- * 将键映射到值的对象
- * 一个映射不能包含重复的键
- * 每个键最多只能映射到一个值

Map 接口与 Collection 接口的不同:

Map 双列, Collection 单列

Map 的键唯一, Collection 的子体系 set 是唯一

Map 分为 hashmap 和 treemap---数据结构指键

Set 分为 hashset 和 treeset---数据结构指元素

Set 底层依赖 Map, Set 所有方法根据 Map 的方法实现

Set 的 add 方法其实 add 的是双列, 只不过值的部分不显示, 底层都是都一个方法

Map 集合的功能

```
Map<String, Integer> map = new HashMap<>();
```

a: 添加功能

- * V put(K key, V value): 添加元素。
 - * 如果键是第一次存储, 就直接存储元素, 返回null
 - * 如果键不是第一次存在, 就用值把以前的值替换掉, 返回以前的值

b: 删除功能

- * void clear(): 移除所有的键值对元素
- * V remove(Object key): 根据键删除键值对元素, 并把值返回

c: 判断功能

- * boolean containsKey(Object key): 判断集合是否包含指定的键
- * boolean containsValue(Object value): 判断集合是否包含指定的值
- * boolean isEmpty(): 判断集合是否为空

d: 获取功能

- * Set<Map.Entry<K,V>> entrySet():
- * V get(Object key): 根据键获取值
- * Set<K> keySet(): 获取集合中所有键的集合
- * Collection<V> values(): 获取集合中所有值的集合

e: 长度功能

- * int size(): 返回集合中的键值对的个数

```
Collection<Integer> c = map.values();
```

Map 集合没实现迭代器接口(iterator), 不能直接遍历和 foreach

Set 实现了迭代器 通过遍历 set 键的集合来遍历 map

```

Set<String> keySet = map.keySet();           //获取所有键的集合
Iterator<String> it = keySet.iterator();       //获取迭代器
while(it.hasNext()) {                         //判断集合中是否有元素
    String key = it.next();                   //获取每一个键
    Integer value = map.get(key);             //根据键获取值
    System.out.println(key + " = " + value);
}

```

Foreach 更简单

```
//使用增强for循环遍历
for(String key : map.keySet()) {           //map.keySet()是所有键的集合
    System.out.println(key + "=" + map.get(key));
}
```

Map.entrySet 遍历

Map集合的第二种迭代，根据键值对对象，获取键和值

A: 键值对对象找键和值思路：

- * 获取所有键值对对象的集合
- * 遍历键值对对象的集合，获取到每一个键值对对象
- * 根据键值对对象找键和值

```
//Map.Entry说明Entry是Map的内部接口，将键和值封装成了Entry对象，并存储在Set集合中
Set<Map.Entry<String, Integer>> entrySet = map.entrySet();
//获取每一个对象
Iterator<Map.Entry<String, Integer>> it = entrySet.iterator();
while(it.hasNext()) {
    //获取每一个Entry对象
    Map.Entry<String, Integer> en = it.next();
    String key = en.getKey();           //根据键值对对象获取键
    Integer value = en.getValue();      //根据键值对对象获取值
    System.out.println(key + "=" + value);
}
```

Foreach

```
for(Map.Entry<String, Integer> en : map.entrySet()) {
    System.out.println(en.getKey() + "=" + en.getValue());
}
```

```
interface Inter {
    interface Inter2 {
        public void show();
    }
}
```

```
class Demo implements Inter.Inter2 {
    @Override
    public void show() {
    }
}
```

当接口内部还有接口时 需要有“.”

Map 接口里有 entry 接口 子类对象 Hashmap 实现了 里有 entry 类，为 map.entrySet 的子类 两种写法都可以

```
Map.Entry<String, Integer> en = it.next(); //父类引用指向子类对象
//Entry<String, Integer> en = it.next(); //直接获取的是子类对象
```

Hashmap

如何保证自定义类的键不重复

HashSet 里的元素要求自定义类重写 hashCode 和 equals 方法，因为两对象算出 hashCode 不同，就不会调用 equals 方法进行比较

```
HashMap<Student, String> hm = new HashMap<>();
hm.put(new Student("张三", 23), "北京");
hm.put(new Student("张三", 23), "上海");
hm.put(new Student("李四", 24), "广州");
```

hashmap 同理，自定义类若不重写两方法，无法使键唯一

Linkedhashmap

```
LinkedHashMap<String, Integer> lhm = new LinkedHashMap<>();
存取有序
```

Treemap

```
TreeMap<Student, String> tm = new TreeMap<>();
```

想要用 treemap 需要把自定义类实现 Comparable 接口并重写 compareTo 方法

```
public int compareTo(Student o) {
    int num = this.age - o.age;           //以年龄为主要条件
    return num == 0 ? this.name.compareTo(o.name) : num;
}
```

当然也可以用 treemap 的比较器 comparator 重写其中的 compare 方法

```
TreeMap<Student, String> tm = new TreeMap<>(new Comparator<Student>() {  
    @Override  
    public int compare(Student s1, Student s2) {  
        int num = s1.getName().compareTo(s2.getName()); //按照姓名比较  
        return num == 0 ? s1.getAge() - s2.getAge() : num;  
    }  
}
```

例 1:

需求：统计字符串中每个字符出现的次数

分析：

- 1, 定义一个需要被统计字符的字符串
- 2, 将字符串转换为字符数组
- 3, 定义双列集合, 存储字符串中字符以及字符出现的次数
- 4, 遍历字符数组获取每一个字符, 并将字符存储在双列集合中
- 5, 存储过程中要做判断, 如果集合中不包含这个键, 就将该字符当作键, 值为1存储, 如果集合中包含这个键, 就将值加1存储
- 6, 打印双列集合获取字符出现的次数

```
String s = "aaaabbbbbccccccccc";  
//2, 将字符串转换为字符数组  
char[] arr = s.toCharArray();  
//3, 定义双列集合, 存储字符串中字符以及字符出现的次数  
HashMap<Character, Integer> hm = new HashMap<>();  
//4, 遍历字符数组获取每一个字符, 并将字符存储在双列集合中  
for(char c: arr) {  
    //5, 存储过程中要做判断, 如果集合中不包含这个键, 就将该字符当作键, 值为1存储, 如果集合中包含这个键, 就将值加1存储  
    /*if(!hm.containsKey(c)) { //如果不包含这个键  
        hm.put(c, 1);  
    } else {  
        hm.put(c, hm.get(c) + 1);  
    }*/  
    hm.put(c, !hm.containsKey(c) ? 1 : hm.get(c) + 1);  
}  
for (Character key : hm.keySet()) { //hm.keySet() 代表所有键的集合  
    System.out.println(key + "=" + hm.get(key)); //hm.get(key) 根据键获取值  
}
```

例 2:

集合嵌套之HashMap嵌套HashMap

需求：

双元课堂有很多基础班

第88期基础班定义成一个双列集合, 键是学生对象, 值是学生的归属地

第99期基础班定义成一个双列集合, 键是学生对象, 值是学生的归属地

无论88期还是99期都是班级对象, 所以为了便于统一管理, 把这些班级对象添加到双元课堂集合中

需要 student 重写 hashCode 和 equals 方法(所有 map 和 set 都需要这样做)

```
//定义88期基础班  
HashMap<Student, String> hm88 = new HashMap<>();  
hm88.put(new Student("张三", 23), "北京");  
hm88.put(new Student("李四", 24), "北京");  
hm88.put(new Student("王五", 25), "上海");  
hm88.put(new Student("赵六", 26), "广州");  
  
//定义99期基础班  
HashMap<Student, String> hm99 = new HashMap<>();  
hm99.put(new Student("唐僧", 1023), "北京");  
hm99.put(new Student("孙悟空", 1024), "北京");  
hm99.put(new Student("猪八戒", 1025), "上海");  
hm99.put(new Student("沙和尚", 1026), "广州");  
  
//定义双元课堂  
HashMap<HashMap<Student, String>, String> hm = new HashMap<>();  
hm.put(hm88, "第88期基础班");  
hm.put(hm99, "第99期基础班");
```

```

//遍历双列集合
for(HashMap<Student, String> h : hm.keySet()) {           //hm.keySet() 代表的是双列集合中键的集合
    String value = hm.get(h);                                //get(h) 根据键对象获取值对象
    //遍历键的双列集合对象
    for(Student key : h.keySet()) {                         //h.keySet() 获取集合总所有的学生键对象
        String value2 = h.get(key);

        System.out.println(key + "==" + value2 + "==" + value);
    }
}

```

Hashtable

Hashtable 基本被 Hashmap 取代了

面试题

HashMap和Hashtable的区别

共同点：

底层都是哈希算法，都是双列集合

区别：

1, HashMap是线程不安全的, 效率高, JDK1.2版本

Hashtable是线程安全的, 效率低, JDK1.0版本的

2, HashMap可以存储null键和null值

Hashtable不可以存储null键和null值

基本上像 vector 和 hashtable 这种被取代的都是线程安全的

例：

模拟斗地主洗牌和发牌，牌没有排序

分析：

1, 买一副扑克，其实就是自己创建一个集合对象，将扑克牌存储进去

2, 洗牌

3, 发牌

4, 看牌

//1, 买一副扑克，其实就是自己创建一个集合对象，将扑克牌存储进去

String[] num = {"A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"};

String[] color = {"红桃", "黑桃", "方片", "梅花"};

ArrayList<String> poker = new ArrayList<>();

//拼接花色和数字

for(String s1 : color) {
 for(String s2 : num) {
 poker.add(s1.concat(s2)); //concat连接两个字符串
 }
}

poker.add("小王");
poker.add("大王");
//2, 洗牌

Collections.shuffle(poker);
//3, 发牌
ArrayList<String> gaojin = new ArrayList<>();
ArrayList<String> longwu = new ArrayList<>();
ArrayList<String> me = new ArrayList<>();
ArrayList<String> dipai = new ArrayList<>();

```

for(int i = 0; i < poker.size(); i++) {
    if(i >= poker.size() - 3) {           //将三张底牌存储在底牌集合中
        dipai.add(poker.get(i));
    } else if(i % 3 == 0) {
        gaojin.add(poker.get(i));
    } else if(i % 3 == 1) {
        longwu.add(poker.get(i));
    } else {
        me.add(poker.get(i));
    }
}

```

若想每个人手里的牌再排序，需要重新组织下整体架构

HashMap<Integer, String>

0	梅花3
1	红桃3
2	方片3
3	黑桃3
4	梅花4
5	红桃4
...	...
53	大王

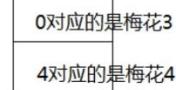
ArrayList,洗牌洗的索引,然后根据索引获取值

0
1
2
3
4
5
...
53

值就相当于被洗乱了



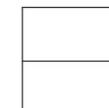
TreeSet



TreeSet



TreeSet



```
//1,买一副扑克,其实就是自己创建一个集合对象,将扑克牌存储进去
String[] num = {"3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A", "2"};
String[] color = {"红桃", "黑桃", "方片", "梅花"};
HashMap<Integer, String> hm = new HashMap<>(); //存储索引和扑克牌
ArrayList<Integer> list = new ArrayList<>(); //存储索引
int index = 0;

//拼接扑克牌并索引和扑克牌存储在hm中
for(String s1 : num) { //获取数字
    for(String s2 : color) { //获取颜色
        hm.put(index, s2.concat(s1));
        list.add(index); //将索引0到51添加到list集中
        index++;
    }
}

//将小王添加到双列集合中
hm.put(index, "小王");
list.add(index); //将52索引添加到集合中
index++;
hm.put(index, "大王");
list.add(index); //将52索引添加到集合中

//2,洗牌
Collections.shuffle(list);

//3,发牌
TreeSet<Integer> gaojin = new TreeSet<>();
TreeSet<Integer> longwu = new TreeSet<>();
TreeSet<Integer> me = new TreeSet<>();
TreeSet<Integer> dipai = new TreeSet<>();

for(int i = 0; i < list.size(); i++) {
    if(i >= list.size() - 3) { //将三张底牌存储在底牌集合中
        dipai.add(list.get(i));
    } else if(i % 3 == 0) {
        gaojin.add(list.get(i));
    } else if(i % 3 == 1) {
        longwu.add(list.get(i));
    } else {
        me.add(list.get(i));
    }
}

看牌
public static void lookPoker(HashMap<Integer, String> hm, TreeSet<Integer> ts ,String name) {
    System.out.print(name + "的牌是:");
    for(Integer i : ts) { //i代表双列集合中的每一个键
        System.out.print(hm.get(i) + " ");
    }
}
```



泛型固定下边界
[? super E]

泛型固定上边界
? extends E

总结

Collection

List(存取有序,有索引,可以重复)

ArrayList

底层是数组实现的,线程不安全,查找和修改快,增和删比较慢

LinkedList

底层是链表实现的,线程不安全,增和删比较快,查找和修改比较慢

Vector

底层是数组实现的,线程安全的,无论增删改查都慢

Set(存取无序,无索引,不可以重复)

HashSet

底层是哈希算法实现

LinkedHashSet

底层是链表实现,但是也是可以保证元素唯一,和HashSet原理一样

TreeSet

底层是二叉树算法实现

一般在开发的时候不需要对存储的元素排序,所以在开发的时候大多用HashSet, HashSet的效率比较高

TreeSet在面试的时候比较多,问你有几种排序方式,和几种排序方式的区别

Map

HashMap

底层是哈希算法,针对键

LinkedHashMap

底层是链表,针对键

TreeMap

底层是二叉树算法,针对键

开发中用HashMap比较多

异常

异常的继承体系

- * Throwable

- * Error

- * Exception

- * RuntimeException

error 异常都是服务器宕机, 数据库崩溃等大错误

安卓，客户端开发，如何处理异常？`try{} catch(Exception e){}`
ee，服务端开发，一般都是底层开发，从底层向上抛

A: JVM默认是如何处理异常的

- * main函数收到这个问题时，有两种处理方式：
- * a:自己将该问题处理，然后继续运行
- * b:自己没有针对的处理方式，只有交给调用main的jvm来处理
- * jvm有一个默认的异常处理机制，就将该异常进行处理。
- * 并将该异常的名称、异常的信息、异常出现的位置打印在了控制台上，同时将程序停止运行

当认为你有问题时，抛出异常对象，int x 无法接受这个对象，由 jvm 进行处理

```
public int div(int a, int b) {           //a = 10, b = 0
    return a / b;                      // 10 / 0 被除数是10，除数是0当除数是0的时候违背了算数运算法则，抛出异常
}
public static void main(String[] args) {
    //demo1();
    Demo d = new Demo();
    int x = d.div(10, 0);
    System.out.println(x);
}
```

* A: 异常处理的两种方式

- * a: try...catch...finally
 - * try catch
 - * try catch finally
 - * try finally
- * b: throws

* B: try...catch 处理异常的基本格式

- * try...catch...finally

* C: 案例演示

- * try...catch 的方式处理1个异常

try: 用来检测异常

I

catch: 用来捕获异常的

finally: 释放资源

当认为你有问题时，抛出异常对象，catch 来接收这个对象

多个 exception exception e 接受所有异常

```
try {
    //System.out.println(a / b);
    //System.out.println(arr[10]);
    arr = null;
    System.out.println(arr[0]);
} catch (ArithmaticException e) {
    System.out.println("除数不能为零");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("索引越界了");
} catch (Exception e) {
    System.out.println("出错了");
}
```

父类引用指向子类对象

```
//Exception e = new NullPointerException();
```

A: 编译期异常和运行期异常的区别

- * Java中的异常被分为两大类：编译时异常和运行时异常。
- * 所有的RuntimeException类及其子类的实例被称为运行时异常，其他的异常就是编译时异常

* 编译时异常

- * Java程序必须显示处理，否则程序就会发生错误，无法通过编译

* 运行时异常

- * 无需显示处理，也可以和编译时异常一样处理

A: Throwable的几个常见方法

- * a:getMessage()
 - * 获取异常信息，返回字符串。
- * b:toString()
 - * 获取异常类名和异常信息，返回字符串。
- * c:printStackTrace()
 - * 获取异常类名和异常信息，以及异常出现在程序中的位置。返回值void。

B: 案例演示

- * Throwable的几个常见方法的基本使用

```
try {
    System.out.println(1/0);
} catch (Exception e) {           //Exception e = new ArithmeticException("/ by zero")
    //System.out.println(e.getMessage());      //获取异常信息
    System.out.println(e);      //调用toString方法，打印异常类名和异常信息
```

当出现异常时就相当于创建出一个除零异常对象

输出自动调用 tostring

抛出异常

```
public void setAge(int age) throws Exception {
    if(age >0 && age <= 150) {
        this.age = age;
    }else {
        throw new Exception("年龄非法");
    }
}
```

若想再 else 里 throw new exception, 需要在这个方法上表明 throws Exception

同样的，调用此方法的方法需要声明它会抛出异常

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

效果：

```
Exception in thread "main" java.lang.Exception: 年龄非法
    at com.heima.exception.Person.setAge(Demo6_Exception.java:46)
    at com.heima.exception.Demo6_Exception.main(Demo6_Exception.java:15)
```

同样的你也可以变为运行时异常(上面的叫编译时异常)，此时方法不用声明 exception

```
public void setAge(int age) {
    if(age >0 && age <= 150) {
        this.age = age;
    }else {
        throw new RuntimeException("年龄非法");
    }
}
```

* 编译时异常的抛出必须对其进行处理

* 运行时异常的抛出可以处理也可以不处理

A: throw的概述

- * 在功能方法内部出现某种情况，程序不能继续运行，需要进行跳转时，就用throw把异常对象抛出。

B: 案例演示

- * 分别演示编译时异常对象和运行时异常对象的抛出

C: throws和throw的区别

- * a:throws
 - * 用在方法声明后面，跟的是异常类名
 - * 可以跟多个异常类名，用逗号隔开
 - * 表示抛出异常，由该方法的调用者来处理
- * b:throw
 - * 用在方法体内，跟的是异常对象名
 - * 只能抛出一个异常对象名
 - * 表示抛出异常，由方法体内的语句处理

```
try {
    return 2;
} catch (error){
    return 1;
} finally {
    return 0;
}
```

结果 0 try return 执行到一半时，会转到 finally，执行完后会回到之前的 return，但已经在 finally 回去了，所以就是 0

Finally 拥有释放资源，在 IO 流和数据库操作会看到
`System.exit(0);`退出 java 虚拟机 此时不会执行 return

`final, finally 和 finalize 的区别`
final 可以修饰类，不能被继承
修饰方法，不能被重写
修饰变量，只能赋值一次

`finally` 是 try 语句中的一个语句体，不能单独使用，用来释放资源

`finalize` 是一个方法，当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

```
public int method() {
    int x = 10;
    try {
        x = 20;
        System.out.println(1/0);
        return x;
    } catch (Exception e) {
        x = 30;
        return x;
    } finally {
        x = 40;
    }
}
```

catch 中的 return 不是未执行，而是已经建立了一个返回路径，x=30 被装箱，这时执行 finally，x=40，执行完后回到 catch 语句中，catch 中的 x 还是 30 语句块中的值只在语句块内有效

自定义异常

```
class AgeOutOfBoundsException extends Exception {

    public AgeOutOfBoundsException() {
        super();
    }

    public AgeOutOfBoundsException(String message) {
        super(message);
    }
}
```

自定义出两种异常，一种无参，一种传参

```
public void setAge(int age) throws AgeOutOfBoundsException {
    if(age > 0 && age <= 150) {
        this.age = age;
    } else {
        //Exception e = new Exception("年龄非法");
        //throw e;
        throw new AgeOutOfBoundsException("年龄非法");
    }
}
```

然后抛出自定义异常

也可以继承于 RuntimeException

异常注意事项

- * a: 子类重写父类方法时，子类的方法必须抛出相同的异常或父类异常的子类。（父亲坏了，儿子不能比父亲更坏）
- * b: 如果父类抛出了多个异常，子类重写父类时，只能抛出相同的异常或者是他的子集，子类不能抛出父类没有的异常
- * c: 如果被重写的方法没有异常抛出，那么子类的方法绝对不可以抛出异常，如果子类方法内有异常发生，那么子类只能 try，不能 throws
- * 如何使用异常处理
- * 原则：如果该功能内部可以将问题处理，用 try，如果处理不了，交由调用者处理，这是用 throws
- * 区别：
 - * 后续程序需要继续运行就 try
 - * 后续程序不需要继续运行就 throws
- * 如果 JDK 没有提供对应的异常，需要自定义异常。

例：

键盘录入一个int类型的整数，对其求二进制表现形式
如果录入的整数过大，给予提示，录入的整数过大请重新录入一个整数BigInteger
如果录入的是小数，给予提示，录入的是小数，请重新录入一个整数
如果录入的是其他字符，给予提示，录入的是非法字符，请重新录入一个整数

若能用 BigInteger 储存代表是因为太大了

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    System.out.println("请输入一个整数：");  
  
    while(true) {  
        String line = sc.nextLine();  
        //将键盘录入的结果存储在line中  
        try {  
            int num = Integer.parseInt(line);  
            //将字符串转换为整数  
            System.out.println(Integer.toBinaryString(num)); //将整数转换为二进制  
            break;  
        } catch (Exception e) {  
            try {  
                new BigInteger(line);  
                System.out.println("录入错误，您录入的是一个过大整数，请重新输入一个整数：");  
            } catch (Exception e2) {  
                //alt + shift + z (try catch快捷键)  
                try {  
                    new BigDecimal(line);  
                    System.out.println("录入错误，您录入的是一个小数，请重新输入一个整数：");  
                } catch (Exception e1) {  
                    System.out.println("录入错误，您录入的是非法字符，请重新输入一个整数：");  
                }  
            }  
        }  
    }  
}
```

File 类

File(String pathname)：根据一个路径得到File对象

File(String parent, String child)：根据一个目录和一个子文件/目录得到File对象

File(File parent, String child)：根据一个父File对象和一个子文件/目录得到File对象

注意你写一个\ 代表转义字符 需要写 \\ 代表 \

```
File file = new File("F:\\双元课堂\\day19\\video\\001_今日内容.avi");  
System.out.println(file.exists());
```

```
String parent = "F:\\双元课堂\\day19\\video";  
String child = "001_今日内容.avi";  
File file = new File(parent,child);
```

A: 创建功能

- * public boolean createNewFile()：创建文件 如果存在这样的文件，就不创建了
- * public boolean mkdir()：创建文件夹 如果存在这样的文件夹，就不创建了
- * public boolean mkdirs()：创建文件夹，如果父文件夹不存在，会帮你创建出来

```
public static void main(String[] args) throws IOException {  
    File file = new File("yyy.txt");  
    System.out.println(file.createNewFile());  
}  
File dir1 = new File("aaa");  
System.out.println(dir1.mkdir());  
File dir3 = new File("ccc\\ddd");  
System.out.println(dir3.mkdirs()); // ccc 文件夹下有 ddd 文件夹
```

如果你创建文件或文件夹忘写盘符路径，那么默认在当前项目路径下

A: 重命名和删除功能

- * public boolean renameTo(File dest)：把文件重命名为指定的文件路径
- * public boolean delete()：删除文件或者文件夹

B: 重命名注意事项

- * 如果路径名相同，就是改名。
- * 如果路径名不同，就是改名并剪切。

C: 删除注意事项：

- * Java中的删除不走回收站。
- * 要删除一个文件夹，请注意该文件夹内不能包含文件或者文件夹

删除文件夹文件夹需要是空的

```
File file1 = new File("xxx.txt");
File file2 = new File("ooo.txt");
System.out.println(file1.renameTo(file2)); 路径相同
File file1 = new File("ooo.txt");
File file2 = new File("D:\\xxx.txt");
System.out.println(file1.renameTo(file2)); 路径不同
```

A:判断功能

```
public boolean isDirectory(): 判断是否是目录
public boolean isFile(): 判断是否是文件
public boolean exists(): 判断是否存在
public boolean canRead(): 判断是否可读
public boolean canWrite(): 判断是否可写
public boolean isHidden(): 判断是否隐藏
```

```
File file = new File("zzz");
file.setReadable(false);
System.out.println(file.canRead()); //windows系统认为所有的文件都是可读的
file.setWritable(true);
System.out.println(file.canWrite()); //windows系统可以设置为不可写
```

A:获取功能

```
public String getAbsolutePath(): 获取绝对路径
public String getPath(): 获取路径
public String getName(): 获取名称
public long length(): 获取长度。字节数
public long lastModified(): 获取最后一次的修改时间，毫秒值
public String[] list(): 获取指定目录下的所有文件或者文件夹的名称数组
public File[] listFiles(): 获取指定目录下的所有文件或者文件夹的File数组
```

```
Date d = new Date(file1.lastModified()); //文件的最后修改时间
SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");
System.out.println(sdf.format(d));

File[] subFiles = dir.listFiles();
for (File file : subFiles) {
    System.out.println(file);
}
file 已经重写了 toString 输出的是路径
```

需求：判断E盘目录下是否有后缀名为.jpg的文件，如果有，就输出该文件名称

- 1.

```
File dir = new File("E:\\\\");
String[] arr = dir.list(); //获取e盘下所有的文件或文件夹
for (String string : arr) {
    if(string.endsWith(".jpg")) {
        System.out.println(string);
    }
}
```

- 2.

```
File[] subFiles = dir.listFiles(); //获取e盘下所有的文件或文件夹对象
for (File subFile : subFiles) {
    if(subFile.isFile() && subFile.getName().endsWith(".jpg")) {
        System.out.println(subFile);
    }
}
```

Filenamefilter 是过滤器一个接口 想要使用需要建立子类对象并重写其内部方法(匿名内部类)

```

文件名称过滤器的概述
} public String[] list(FilenameFilter filter)
} public File[] listFiles(FileFilter filter)

String[] arr = dir.list(new FilenameFilter() {
    @Override
    public boolean accept(File dir, String name) {
        //System.out.println(dir);
        //System.out.println(name);
        File file = new File(dir, name);
        return file.isFile() && file.getName().endsWith(".jpg");
    }
});

for (String string : arr) {
    System.out.println(string);
}

```

IO流(字节和字符，其它用的很少)

用于处理设备间数据传输

流按流向分为两种：输入流，输出流。

流按操作类型分为两种：

- * 字节流：字节流可以操作任何数据，因为在计算机中任何数据都是以字节的形式存储的
- * 字符流：字符流只能操作纯字符数据，比较方便。

IO流常用父类

* 字节流的抽象父类：

- * `InputStream`
- * `OutputStream`

* 字符流的抽象父类：

- * `Reader`
- * `Writer`

字节流

需要设置文件不存在时，出现的异常情况

IO程序书写

- * 使用前，导入IO包中的类
- * 使用时，进行IO异常处理
- * 使用后，释放资源

FileInputStream

```

public static void main(String[] args) throws IOException {
    FileInputStream fis = new FileInputStream("xxx.txt");      //创建流对象
    int x = fis.read();                                         //从硬盘上读取一个字节
    System.out.println(x);
    int y = fis.read();
    System.out.println(y);                                     //关流释放资源
    fis.close();
}

```

`Read()`一次指针向后移一位(每次只能读取一个字符)，越界后会返回-1，

文档中是“abc” 输出结果 97 98 “a”真正在硬盘上(硬盘上只能存储二进制)是以 97 存储 asc 码表翻译成了 97

读取文档 `Read()`方法读取的是一个字节

```

public static void main(String[] args) throws IOException {
    //demo1();
    FileInputStream fis = new FileInputStream("xxx.txt");      //创建流对象
    int b;
    while((b = fis.read()) != -1) {
        System.out.println(b);
    }

    fis.close();
}

```

Read()方法, 为什么返回的是 int, 而不是 byte:

10000001 byte类型-1的原码
11111110 -1的反码
11111111 -1的补码 一个字节是八位 00010010

因为字节输入流可以操作任意类型的文件, 比如图片音频等, 这些文件底层都是以二进制形式的存储的, 如果每次读取都返回byte, 有可能在读到中间的时候遇到11111111那么这11111111是byte类型的-1, 我们的程序是遇到-1就会停止不读了, 后面的数据就读不到了, 所以在读取的时候用int类型接收, 如果11111111会在其前面补上24个0凑足4个字节, 那么byte类型的-1就变成int类型的255了这样可以保证整个数据读完, 而结束标记的-1就是int类型

Fileoutputstream

输出文档 write 每次写入一个字符

```
public static void main(String[] args) throws IOException {
    FileOutputStream fos = new FileOutputStream("yyy.txt");           //创建字节输出流对象, 如果没有就自动创建一个
    fos.write(97);
    fos.write(98);
    fos.write(99);

    fos.close();
}
```

结果 txt 里显示的是 “abc”

虽然写出的是 int 数, 但是到文件上是一个字节, 会自动去除前三个八位

```
FileOutputStream fos = new FileOutputStream("yyy.txt");
若不存在yyy, 则创建, 若存在, 则清空yyy的内容, 等待写入
FileOutputStream fos = new FileOutputStream("yyy.txt", true);      //如果想续写就在第二个参数传true
此时可以不清空的情况下写入
```

复制文件

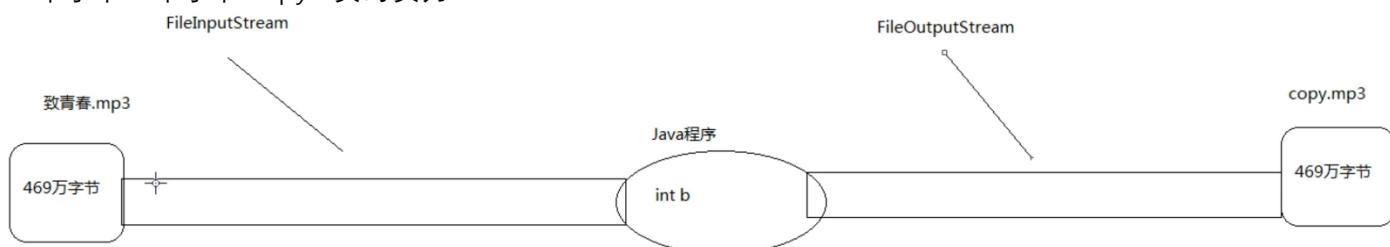
读取一个图片, 输出这个图片(copy)

```
public static void main(String[] args) throws IOException {
    FileInputStream fis = new FileInputStream("双元.jpg");           //创建输入流对象, 关联双元.jpg
    FileOutputStream fos = new FileOutputStream("copy.jpg");            //创建输出流对象, 关联copy.jpg

    int b;
    while((b = fis.read()) != -1) {                                     //在不断的读取每一个字节
        fos.write(b);                                                    //将每一个字节写出
    }

    fis.close();                                                       //关流释放资源
    fos.close();
}
```

一个字节一个字节 copy 费时费力



不推荐, 此方式虽速度快, 但若 copy 特别大的(好像是大于 170MB), 会造成 java 虚拟机内存溢出

```
//第二种拷贝
FileInputStream fis = new FileInputStream("致青春.mp3");           //创建输入流对象, 关联致青春.mp3
FileOutputStream fos = new FileOutputStream("copy.mp3");             //创建输出流对象, 关联copy.mp3
//int len = fis.available();
//System.out.println(len);

byte[] arr = new byte[fis.available()];                                //创建与文件一样大小的字节数组
fis.read(arr);                                                        //将文件上的字节读取到内存中
fos.write(arr);                                                       //将字节数组中的字节数据写到文件上

fis.close();
fos.close();
```

fis.write(arr)返回值代表读到的有效字节个数，未读到有效字节时返回-1

第三种拷贝 小数组拷贝

```
FileInputStream fis = new FileInputStream("xxx.txt");
FileOutputStream fos = new FileOutputStream("yyy.txt");

byte[] arr = new byte[1024 * 8];
int len;
while((len = fis.read(arr)) != -1) {
    fos.write(arr, 0, len);
}

fis.close();
fos.close();
```

Write(字符数组，数组起始位置，读取个数) 若直接读，会出现第一次 9798，第二次 9998，第二次只读一个，但 arr[2]未覆盖会被保留
一般是 1024 的整数倍，计算机的进制就是 1024，1024 字节是 1kb

BufferedInputStream/OutputStream

BufferedInputStream/OutputStream 使正常输入输出更强大

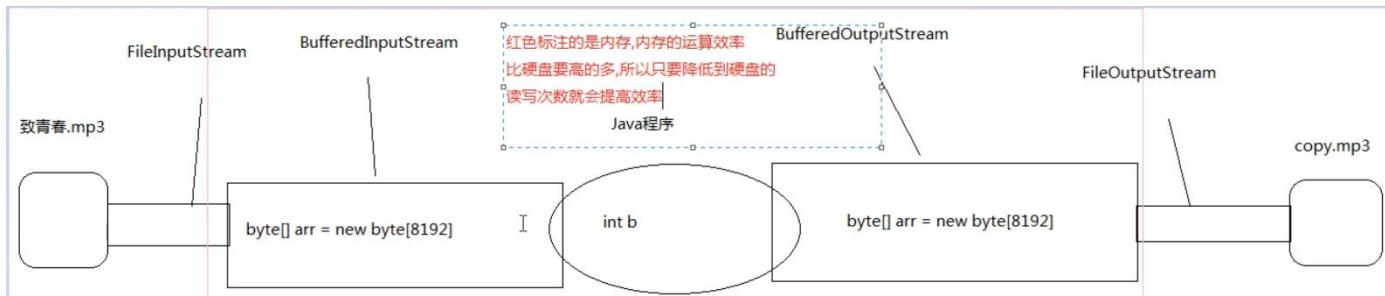
```
public static void main(String[] args) throws IOException {
    FileInputStream fis = new FileInputStream("致青春.mp3");
    FileOutputStream fos = new FileOutputStream("copy.mp3");
    BufferedInputStream bis = new BufferedInputStream(fis);
    BufferedOutputStream bos = new BufferedOutputStream(fos);

    int b;
    while((b = bis.read()) != -1) {
        bos.write(b);
    }

    bis.close();
    bos.close();
}
```

//创建输入流对象，关联致青春.mp3
//创建输出流对象，关联copy.mp3
//创建缓冲区对象，对输入流进行包装让其变得更强大

底层是一个字符数组



A: 缓冲思想

- * 字节流一次读写一个数组的速度明显比一次读写一个字节的速度快很多，
- * 这是加入了数组这样的缓冲区效果，java本身在设计的时候，
- * 也考虑到了这样的设计思想(装饰设计模式后面讲解)，所以提供了字节缓冲区流

B. BufferedInputStream

- * BufferedInputStream内置了一个缓冲区(数组)
- * 从BufferedInputStream中读取一个字节时
- * BufferedInputStream会一次性从文件中读取8192个，存在缓冲区中，返回给程序一个
- * 程序再次读取时，就不用找文件了，直接从缓冲区中获取
- * 直到缓冲区中所有的都被使用过，才重新从文件中读取8192个

C. BufferedOutputStream

- * BufferedOutputStream也内置了一个缓冲区(数组)
- * 程序向流中写出字节时，不会直接写到文件，先写到缓冲区中
- * 直到缓冲区写满了，BufferedOutputStream才会把缓冲区中的数据一次性写到文件里

小数组的读写和带Buffered的读取哪个更快？

- * 定义小数组如果是8192个字节大小和Buffered比较的话
- * 定义小数组会略胜一筹，因为读和写操作的是同一个数组
- * 而Buffered操作的是两个数组

Flush 和 close

flush()方法

- * 用来刷新缓冲区的，刷新后可以再次写出

close()方法

- * 用来关闭流释放资源的，如果是带缓冲区的流对象的close()方法，不但会关闭流，还会再关闭流之前刷新缓冲区，关闭后不能再写出

若你不使用 close, copy 的文件最后会少一点, 最后的一次的数组没有被刷新到输出流中

Flush 实时可看到, 不用满足 8196 满数组后才到输出流使之能看到

字节流异常

一个中文是两个字节

字节流读取中文的问题

* 字节流在读中文的时候有可能会读到半个中文, 造成乱码

字节流写出中文的问题

* 字节流直接操作的字节, 所以写出中文必须将字符串转换成字节数组

* 写出回车换行 `write("\r\n".getBytes());`

解决方法: 1.字符流读取 2. ByteArrayOutputStream

```
public static void main(String[] args) throws IOException {
    FileInputStream fis = new FileInputStream("yyy.txt");
    byte[] arr = new byte[4];
    int len;
    while((len = fis.read(arr)) != -1) {
        System.out.println(new String(arr, 0, len));
    }

    fis.close();
```

若有, 则会造成乱码, 两个字节拼不起来

```
FileOutputStream fos = new FileOutputStream("zzz.txt");
fos.write("我读书少, 你不要骗我".getBytes());
fos.write("\r\n".getBytes());
fos.close();
```

Getbytes 将字符串转化成字符数组

异常处理

局部变量在使用前必须进行赋值, 静态变量会被赋予初始值, 所以下面出错

```
FileInputStream fis;
FileOutputStream fos;
try {
    fis = new FileInputStream("xxx.txt");
    fos = new FileOutputStream("yyy.txt");

    int b;
    while((b = fis.read()) != -1) {
        fos.write(b);
    }
} finally {
    fis.close();
    fos.close();
}
```

若等于 null, 你 close, 会发生空指针异常

标准的异常处理 1.6 以前

```
public static void main(String[] args) throws IOException {
    FileInputStream fis = null;
    FileOutputStream fos = null;
    try {
        fis = new FileInputStream("xxx.txt");
        fos = new FileOutputStream("yyy.txt");

        int b;
        while((b = fis.read()) != -1) {
            fos.write(b);
        }
    } finally {
        try {
            if(fis != null)
                fis.close();
        } finally {
            if(fos != null)
                fos.close();
        }
    }
}
```

1.7 版本之后 当把流对象写到 try 的小括号中时, 当在大括号执行完流代码之后, 会自动执行流关闭代码

```

public static void main(String[] args) throws IOException {
    //demol();
    try {
        FileInputStream fis = new FileInputStream("xxx.txt");
        FileOutputStream fos = new FileOutputStream("yyy.txt");
    } {
        int b;
        while((b = fis.read()) != -1) {
            fos.write(b);
        }
    }
}

```

实现了 autocloseable 接口中 close 方法

图片加密

将写出的字节异或上一个数，这个数就是密钥，解密的时候再次异或就可以了

异或运算，相同为 1，不同为 0，对一个东西异或一次，结果改变，异或两次，结果相同
加密：

```

public static void main(String[] args) throws IOException {
    BufferedInputStream bis = new BufferedInputStream(new FileInputStream("双元.jpg"));
    BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("copy.jpg"));

    int b;
    while((b = bis.read()) != -1) {
        bos.write(b ^ 123);
    }

    bis.close();
    bos.close();
}

```

解密：

```

public static void main(String[] args) throws IOException {
    BufferedInputStream bis = new BufferedInputStream(new FileInputStream("copy.jpg"));
    BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("copy2.jpg"));

    int b;
    while((b = bis.read()) != -1) {
        bos.write(b ^ 123);
    }

    bis.close();
    bos.close();
}

```

例题

在控制台录入文件的路径，将文件拷贝到当前项目下

分析：

1, 定义方法对键盘录入的路径进行判断，如果是文件就返回

2, 在主方法中接收该文件

3, 读和写该文件

```

public static File getFile() {
    Scanner sc = new Scanner(System.in);           // 创建键盘录入对象

    while(true) {
        String line = sc.nextLine();               // 接收键盘录入的路径
        File file = new File(line);                // 封装成File对象，并对其进行判断
        if(!file.exists()) {
            System.out.println("您录入的文件路径不存在，请重新录入：");
        }else if(file.isDirectory()) {
            System.out.println("您录入的是文件夹路径，请重新录入：");
        }else {
            return file;
        }
    }
}

```

```
public static void main(String[] args) throws IOException {
    File file = getFile(); //获取文件
    BufferedInputStream bis = new BufferedInputStream(new FileInputStream(file));
    BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(file.getName()));
    int b;
    while((b = bis.read()) != -1) {
        bos.write(b);
    }
    bis.close();
    bos.close();
}
```

将键盘录入的数据拷贝到当前项目下的text.txt文件中，键盘录入数据当遇到quit时就退出

分析：

- 1, 创建键盘录入对象
- 2, 创建输出流对象, 关联text.txt文件
- 3, 定义无限循环
- 4, 遇到quit退出循环
- 5, 如果不quit, 就将内容写出
- 6, 关闭流

```
//1, 创建键盘录入对象
Scanner sc = new Scanner(System.in);
//2, 创建输出流对象, 关联text.txt文件
FileOutputStream fos = new FileOutputStream("text.txt");
//3, 定义无限循环
while(true) {
    String line = sc.nextLine(); //将键盘录入的数据存储在line中
    //4, 遇到quit退出循环
    if("quit".equals(line)) {
        break;
    }
    //5, 如果不quit, 就将内容写出
    fos.write(line.getBytes()); //字符串写出必须转换成字节数组
    fos.write("\r\n".getBytes());
}
//6, 关闭流
fos.close();
```

字符流

字符流 FileReader/Writer (字符：字母、数字、字和符号)

字符流是可以直接读写字符的IO流

字符流读取字符，就要先读取到字节数据，然后转为字符。如果要写出字符，需要把字符转为字节再写出。

FileReader

读到的是 int 字节数，强转到 char 进行输出

```
public static void main(String[] args) throws IOException {
    //demo1();
    FileReader fr = new FileReader("xxx.txt");
    int c;

    while((c = fr.read()) != -1) { //通过项目默认的码表一次读取一个字符
        System.out.print((char)c);
    }
    fr.close();
```

FileWriter

yyy.txt 后加 true 代表不清空原文件

```
public static void main(String[] args) throws IOException {
    FileWriter fw = new FileWriter("yyy.txt");
    fw.write("大家好，基础班快接近尾声了，大家要努力，要坚持!!!!");

    fw.close();
}
```

write 接收的是字符串，直接 write(97) 在文件中显示 a

复制文件

```
public static void main(String[] args) throws IOException {
    FileReader fr = new FileReader("xxx.txt");
    FileWriter fw = new FileWriter("zzz.txt");

    int c;
    while((c = fr.read()) != -1) {
        fw.write(c);
    }
    fr.close();
    fw.close();
}
```

此时不需要强转了

若 fw.close() 没有写，结果可能为空或少数据，filewriter 自带缓冲区(2kb)，结果在缓冲区内没有放出来

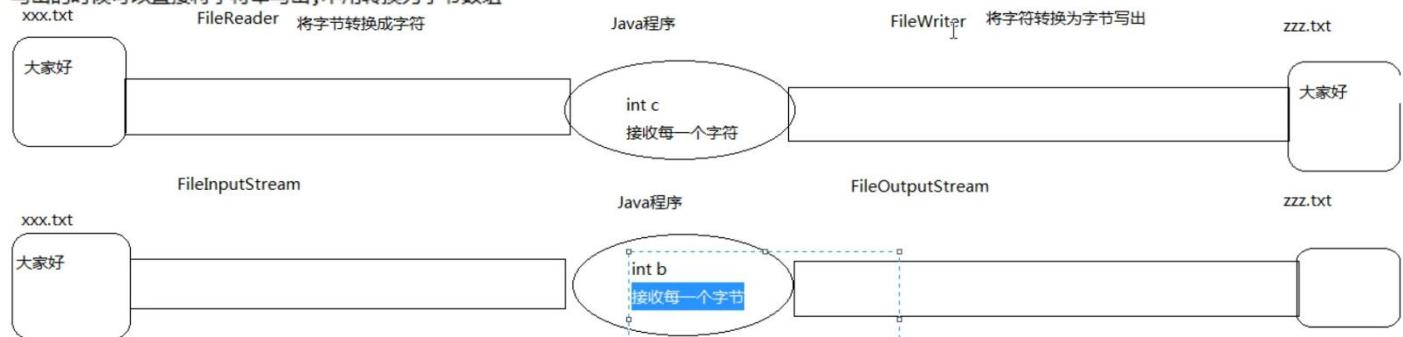
在只读或只写的情况下推荐字符流，拷贝文本推荐字节流

字符流也可以拷贝文本文件，但不推荐使用。因为读取时会把字节转为字符，写出时还要把字符转回字节

程序需要读取一段文本，或者需要写出一段文本的时候可以使用字符流

读取的时候是按照字符的大小读取的，不会出现半个中文

写出的时候可以直接将字符串写出，不用转换为字节数组



字符流不可以拷贝非纯文本文件，使用字节流，因为不是每一种字节都有对应的字符转化

通过自定义字符数组增加效率

```
public static void main(String[] args) throws IOException {
    //demo1();
    //demo2();
    FileReader fr = new FileReader("xxx.txt");
    FileWriter fw = new FileWriter("yyy.txt");

    char[] arr = new char[1024];
    int len;
    while((len = fr.read(arr)) != -1) {           //将文件上的数据读取到字符数组中
        fw.write(arr, 0, len);                     //将字符数组中的数据写到文件上
    }

    fr.close();
    fw.close();
```

通过 bufferedReader/Writer 增加效率 缓冲区 16k

```
BufferedReader br = new BufferedReader(new FileReader("xxx.txt"));
BufferedWriter bw = new BufferedWriter(new FileWriter("yyy.txt"));

int c;
while((c = br.read()) != -1) {
    bw.write(c);
}

br.close();
bw.close();
```

BufferedReader&Writer

BufferedReader 的 readLine 方法 读取一行，读到/r/n 停止(不读/r/n)

```

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader("zzz.txt"));
    String line;

    while((line = br.readLine()) != null) {
        System.out.println(line);
    }

    br.close();
}

```

BufferedReader 的 newLine 方法写入一个换行符

newLine()与\r\n的区别

newLine()是跨平台的方法

\r\n只支持的是windows系统

```

public static void main(String[] args) throws IOException {
    //demo1();
    BufferedReader br = new BufferedReader(new FileReader("zzz.txt"));
    BufferedWriter bw = new BufferedWriter(new FileWriter("aaa.txt"));

    String line;
    while((line = br.readLine()) != null) {
        bw.write(line);
        //bw.newLine(); //写出回车换行符
        bw.write("\r\n");
    }

    br.close();
}

```

将一个文本文档上的文本反转，第一行和倒数第一行交换，第二行和倒数第二行交换

分析：

1, 创建输入输出流对象

2, 创建集合对象

3, 将读到的数据存储在集合中

4, 倒着遍历集合将数据写到文件上

5, 关流

流对象尽量晚开早关

```

// 1, 创建输入输出流对象
BufferedReader br = new BufferedReader(new FileReader("zzz.txt"));
BufferedWriter bw = new BufferedWriter(new FileWriter("revzzz.txt"));
//2, 创建集合对象
ArrayList<String> list = new ArrayList<>();
//3, 将读到的数据存储在集合中
String line;
while((line = br.readLine()) != null) {
    list.add(line);
}
//4, 倒着遍历集合将数据写到文件上
for(int i = list.size() - 1; i >= 0; i--) {
    bw.write(list.get(i));
    bw.newLine();
}
//5, 关流
br.close();
bw.close();

```

LineNumberReader

是 BufferedReader 子类 多了一个行号 get/setLineNumber

```

public static void main(String[] args) throws IOException {
    LineNumberReader lnr = new LineNumberReader(new FileReader("zzz.txt"));

    String line;
    lnr.setLineNumber(100);           I
    while((line = lnr.readLine()) != null) {
        System.out.println(lnr.getLineNumber() + ":" + line);
    }

    lnr.close();
}

```

若你 setlinenumber 100 则从 101 开始 若无 set 从 1 开始记行号

装饰设计模式

1. 获取被装饰类的引用
2. 在构造方法中传入被装饰类的对象
3. 对原有的功能进行升级

(类似于 bufferedReader 是对 fileReader 的一个装饰)

好处：耦合性不强，被装饰的类的变化与装饰类的变化无关

```

interface Coder {
    public void code();
}

class Student implements Coder {

    @Override
    public void code() {
        System.out.println("javase");
        System.out.println("javaweb");
    }
}

class HeiMaStudent implements Coder {
    //1, 获取被装饰类的引用
    private Student s;                      //获取学生引用

    //2, 在构造方法中传入被装饰类的对象
    public HeiMaStudent(Student s) {
        this.s = s;
    }

    //3, 对原有的功能进行升级
    @Override
    public void code() {
        s.code();
        System.out.println("ssh");
        System.out.println("数据库");
        System.out.println("大数据");
        System.out.println("...");
    }
}

public static void main(String[] args) {
    HeiMaStudent hms = new HeiMaStudent(new Student());
    hms.code();
}

```

使用指定的编码表读写

InPutStreamReader/OutPutStreamWriter 注意其接收的是（字节流，编码表）

```

InputStreamReader isr = new InputStreamReader(new FileInputStream("utf-8.txt"), "uTf-8");
OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("gbk.txt"), "gbk");

```

```

int c;
while((c = isr.read()) != -1) {
    osw.write(c);
}

isr.close();
osw.close();

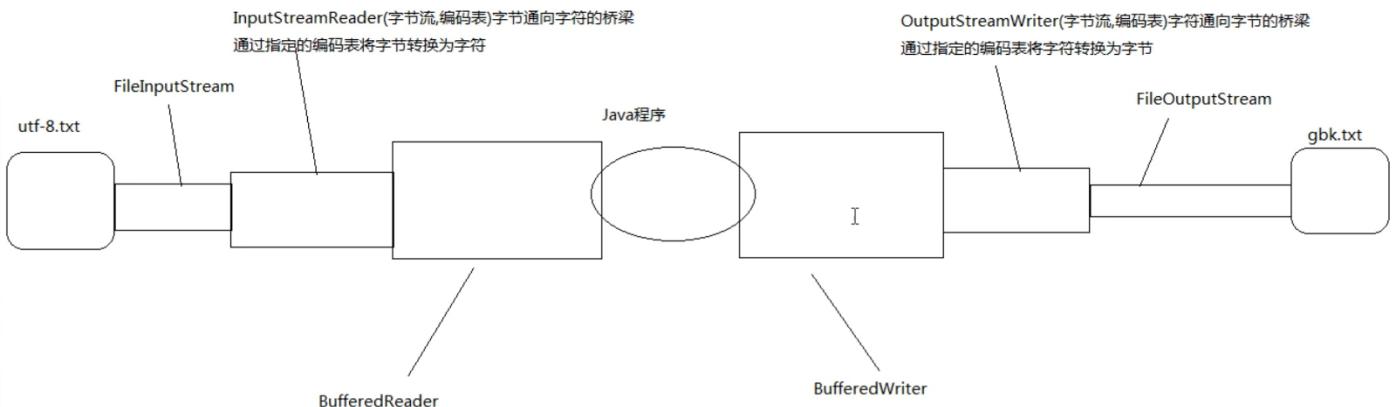
```

提高效率

```
BufferedReader br =           I                                //更高效的读
    new BufferedReader(new InputStreamReader(new FileInputStream("utf-8.txt"), "utf-8")));
BufferedWriter bw =           I                                //更高效的写
    new BufferedWriter(new OutputStreamWriter(new FileOutputStream("gbk.txt"), "gbk")));
int c;
while((c = br.read()) != -1) {
    bw.write(c);
}

br.close();
bw.close();
```

转换流图解



例题

获取一个文本上每个字符出现的次数，将结果写在times.txt上

分析：

- 1, 创建带缓冲的输入流对象
- 2, 创建双列集合对象TreeMap
- 3, 将读到的字符存储在双列集合中, 存储的时候要做判断, 如果不包含这个键, 就将键和1存储, 如果包含这个键, 就将该键和值加1存储
- 4, 关闭输入流
- 5, 创建输出流对象
- 6, 遍历集合将集合中的内容写到times.txt中

7, 关闭输出流

```
//1, 创建带缓冲的输入流对象
BufferedReader br = new BufferedReader(new FileReader("zzz.txt"));
//2, 创建双列集合对象TreeMap
TreeMap<Character, Integer> tm = new TreeMap<>();
//3, 将读到的字符存储在双列集合中, 存储的时候要做判断, 如果不包含这个键, 就将键和1存储, 如果包含这个键, 就将该键和值加1存储
int ch;
while((ch = br.read()) != -1) {
    char c = (char)ch;                                //强制类型转换
    /*if(!tm.containsKey(c)) {
        tm.put(c, 1);
    }else {
        tm.put(c, tm.get(c) + 1);
    }*/
    tm.put(c, !tm.containsKey(c) ? 1 : tm.get(c) + 1);
}
//4, 关闭输入流
br.close()
//5, 创建输出流对象
BufferedWriter bw = new BufferedWriter(new FileWriter("times.txt"));
//6, 遍历集合将集合中的内容写到times.txt中
for(Character key : tm.keySet()) {
    bw.write(key + "=" + tm.get(key));                //写出键和值
    bw.newLine();
}
//7, 关闭输出流
bw.close();
```

改善

```
for(Character key : tm.keySet()) {  
    switch (key) {  
        case '\t':  
            bw.write("\t" + "=" + tm.get(key));  
            break;  
        case '\n':  
            bw.write("\n" + "=" + tm.get(key));  
            break; I  
        case '\r':  
            bw.write("\r" + "=" + tm.get(key));  
            break;  
        default:  
            bw.write(key + "=" + tm.get(key)); //写出键和值  
            break;  
    }  
    bw.newLine();  
}
```

当我们下载一个试用版软件，没有购买正版的时候，每执行一次就会提醒我们还有多少次使用机会用学过的IO流知识，模拟试用版软件，试用10次机会，执行一次就提示一次您还有几次机会，如果次数到了提示请购买正版

@throws IOException

分析：

- 1, 创建带缓冲的输入流对象，因为要使用readLine方法，可以保证数据的原样性
- 2, 将读到的字符串转换为int数
- 3, 对int数进行判断，如果大于0，就将其--写回去，如果不大于0，就提示请购买正版
- 4, 在if判断中要将--的结果打印，并将结果通过输出流写到文件上

此处使用字符流或字节流都不合适，因为读取10每次都先把1当一个字符或字节，不能整体10读取
应使用bufferedReader的ReadLine整行读取

注意不能这样写，config中有数据，但当你new filewriter时就清空了config，只能读到null

```
BufferedReader br = new BufferedReader(new FileReader("config.txt"));  
FileWriter fw = new FileWriter("config.txt");
```

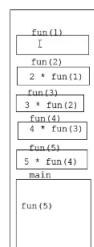
```
public static void main(String[] args) throws IOException {  
    //1, 创建带缓冲的输入流对象，因为要使用readLine方法，可以保证数据的原样性  
    BufferedReader br = new BufferedReader(new FileReader("config.txt"));  
    //2, 将读到的字符串转换为int数  
    String line = br.readLine();  
    int times = Integer.parseInt(line); //将数字字符串转换为数字  
    //3, 对int数进行判断，如果大于0，就将其--写回去，如果不大于0，就提示请购买正版  
    if(times > 0) {  
        //4, 在if判断中要将--的结果打印，并将结果通过输出流写到文件上  
        System.out.println("您还有" + times-- + "次机会");  
        FileWriter fw = new FileWriter("config.txt");  
        fw.write(times + "");  
        fw.close();  
    } else {  
        System.out.println("您的试用次数已到，请购买正版");  
    }  
}
```

Filewriter内有小缓冲区 若fw忘记关流，会出现数字格式异常，null无法转换成int，没关流时数字会被卡在fw缓冲区中，所以为fw内为null

递归

阶乘

```
public static int fun(int num) {  
    if(num == 1) {  
        return 1;  
    } else {  
        return num * fun(num - 1);  
    }  
}
```



栈： 递归弊端：不能调用次数过多---栈内存溢出

构造方法不能使用递归调用

递归调用不一定必须有返回值

输出路径下所有文件名

需求：从键盘输入接收一个文件夹路径，打印出该文件夹下所有的.java文件名

分析：

从键盘接收一个文件夹路径

1, 如果录入的是不存在, 给与提示

2, 如果录入的是文件路径, 给与提示

3, 如果是文件夹路径, 直接返回

打印出该文件夹下所有的.java文件名

1, 获取到该文件夹路径下的所有的文件和文件夹, 存储在File数组中

2, 遍历数组, 对每一个文件或文件夹做判断

3, 如果是文件, 并且后缀是.java的, 就打印

4, 如果是文件夹, 就递归调用

获取录入的文件夹路径

```
public static File getDir() {
    Scanner sc = new Scanner(System.in); // 创建键盘录入对象
    System.out.println("请输入一个文件夹路径");
    while(true) {
        String line = sc.nextLine(); // 将键盘录入的文件夹路径存储
        File dir = new File(line); // 封装成File对象
        if(!dir.exists()) {
            System.out.println("您录入的文件夹路径不存在, 请重新录入");
        } else if(dir.isFile()) {
            System.out.println("您录入的是文件路径, 请重新录入文件夹路径");
        } else { // 如果是文件夹
            return dir;
        }
    }
}
```

获取文件夹路径下的所有 java 文件

```
public static void printJavaFile(File dir) {
    //1, 获取到该文件夹路径下的所有的文件和文件夹, 存储在File数组中
    File[] subFiles = dir.listFiles();
    //2, 遍历数组, 对每一个文件或文件夹做判断
    for (File subFile : subFiles) {
        //3, 如果是文件, 并且后缀是.java的, 就打印
        if(subFile.isFile() && subFile.getName().endsWith(".java")) {
            System.out.println(subFile);
        } //4, 如果是文件夹, 就递归调用
        else if (subFile.isDirectory()) {
            printJavaFile(subFile);
        }
    }
}
```

Main

```
public static void main(String[] args) {
    File dir = getDir();
    printJavaFile(dir);
}
```

序列流

序列流可以把多个字节输入流整合成一个，从序列流中读取数据时，将从被整合的第一个流开始读，读完一个之后继续读第二个，以此类推。
整合两个输入流 SequenceInputStream (InputStream, InputStream)

```
public static void main(String[] args) throws IOException {
    //demo1();
    FileInputStream fis1 = new FileInputStream("a.txt");
    FileInputStream fis2 = new FileInputStream("b.txt");
    SequenceInputStream sis = new SequenceInputStream(fis1, fis2);
    FileOutputStream fos = new FileOutputStream("c.txt");

    int b;
    while((b = sis.read()) != -1) {
        fos.write(b);
    }

    sis.close();
    fos.close();
}
```

整合多个输入流

```
FileInputStream fis1 = new FileInputStream("a.txt"); //创建输入流对象,关联a.txt
FileInputStream fis2 = new FileInputStream("b.txt"); //创建输入流对象,关联b.txt
FileInputStream fis3 = new FileInputStream("c.txt"); //创建输入流对象,关联c.txt
Vector<InputStream> v = new Vector<>(); //创建vector集合对象
v.add(fis1); //将流对象添加
v.add(fis2);
v.add(fis3);
Enumeration<InputStream> en = v.elements(); //获取枚举引用
SequenceInputStream sis = new SequenceInputStream(en); //传递给SequenceInputStream构造
FileOutputStream fos = new FileOutputStream("d.txt");
int b;
while((b = sis.read()) != -1) {
    fos.write(b);
}

sis.close();
fos.close();
```

内存输出流

ByteArrayOutputStream 把读到的数据存到内存中去 它无需 close

```
public static void main(String[] args) throws IOException {
    //demo1();
    FileInputStream fis = new FileInputStream("e.txt");
    ByteArrayOutputStream baos = new ByteArrayOutputStream(); //在内存中创建了可以增长的内存数组

    int b;
    while((b = fis.read()) != -1) {
        baos.write(b); //将读取到的数据逐个写到内存中
    }

    //byte[] arr = baos.toByteArray();
    //System.out.println(new String(arr)); //将缓冲区的数据全部获取出来,并赋值给arr数组

    System.out.println(baos.toString()); //将缓冲区的内容转换为了字符串,在输出语句中可以省略调
    fis.close(); //将缓冲区的内容转换为了字符串,在输出语句中可以省略调
}
```

定义一个文件输入流,调用read(byte[] b)方法,将a.txt文件中的内容打印出来 (byte数组大小限制为5)

分析:

1, reda(byte[] b)是字节输入流的方法,创建FileInputStream,关联a.txt

2, 创建内存输出流,将读到的数据写到内存输出流中

3, 创建字节数组,长度为5

4, 将内存输出流的数据全部转换为字符串打印

5, 关闭输入流

```
public static void main(String[] args) throws IOException {
    //1,reda(byte[] b)是字节输入流的方法,创建FileInputStream,关联a.txt
    FileInputStream fis = new FileInputStream("a.txt");
    //2,创建内存输出流,将读到的数据写到内存输出流中
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    //3,创建字节数组,长度为5
    byte[] arr = new byte[5];
    int len;

    while((len = fis.read(arr)) != -1) {
        baos.write(arr, 0, len);
    }
    //4,将内存输出流的数据全部转换为字符串打印
    System.out.println(baos); //即使没有调用,底层也会默认帮我们调用toString()方法
    //5,关闭输入流
    fis.close();
}
```

随机访问流

RandomAccessFile类不属于流,是Object类的子类。但它融合了InputStream和OutputStream的功能。
支持对随机访问文件的读取和写入。

```

public static void main(String[] args) throws IOException {
    RandomAccessFile raf = new RandomAccessFile("g.txt", "rw");
    //raf.write(97);
    //int x = raf.read();
    //System.out.println(x);
    raf.seek(0); //在指定位置设置指针
    raf.write(98);
    raf.close();
}

```

从指针位置开始读写 rw 代表读写模式

对象操作流

该流可以将一个对象写出，或者读取一个对象到程序中。也就是执行了序列化和反序列化的操作

将对象写到文件上---序列化(游戏角色存档) 从文件中读取对象---反序列化(游戏角色读档)

若对象想可以被序列化，此类需实现 serializable 接口

```
public class Person implements Serializable {
```

也是一种装饰设计模式 ObjectOutputStream (fileoutputstream)

ObjectOutputStream 对象输入流

```

public static void main(String[] args) throws IOException {
    Person p1 = new Person("张三", 23);
    Person p2 = new Person("李四", 24);

    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("e.txt"));
    oos.writeObject(p1);
    oos.writeObject(p2);

    oos.close();
}

```

把对象转化成字节数组，用计算机码表进行翻译，输出文档---乱码

```
oos.writeo
```

 writeObject(Object obj) : void - ObjectOutputStream p1 会自动类型提升为 object，注意读取时需要强制类型转换

ObjectInputStream 对象输出流

```

public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream("e.txt"));

    Person p1 = (Person) ois.readObject();
    Person p2 = (Person) ois.readObject();

    System.out.println(p1);
    System.out.println(p2);

    ois.close();
}

```

当你有一个 person p3 读取第三次时会出现文件已经读取到末尾异常，解决方法：将对象储存到集合中

```

Person p1 = new Person("张三", 23);
Person p2 = new Person("李四", 24);
Person p3 = new Person("马哥", 18);
Person p4 = new Person("辉哥", 20);

ArrayList<Person> list = new ArrayList<>();
list.add(p1);
list.add(p2);
list.add(p3);
list.add(p4);

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("f.txt"));
oos.writeObject(list); //写出集合对象

oos.close();

```

读取到的是一个集合对象

```

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("f.txt"));
ArrayList<Person> list = (ArrayList<Person>)ois.readObject(); //泛型在 //想去掉
for (Person person : list) {
    System.out.println(person);
}

```

数据输入输出流

DataInputStream/DataOutputStream

字节流和字符流按照字节(8个二进制位)和字符大小读取数据

但基本数据类型 int(4个字节---32个二进制位), 无法用正常方法读取

```
00000000 00000000 00000011 11100101 int类型997
```

```
11100101
```

```
00000000 00000000 00000000 11100101
```

会自动去除前三个八位, 待读取后自动补上三个八位的 0, 会混乱

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream("h.txt"));
dos.writeInt(997);
dos.writeInt(998);
dos.writeInt(999);
dos.close();
```

此时文档中写入的内容仍然是乱码, 但是可以读取出正确值

```
DataInputStream dis = new DataInputStream(new FileInputStream("h.txt"));
int x = dis.readInt();
int y = dis.readInt();
int z = dis.readInt();

System.out.println(x);
System.out.println(y);
System.out.println(z);
输出 997 998 999
```

Printwriter

```
ps.println(97); //底层通过Integer.toString()将97转换成字符串并打印
ps.write(97); //查找码表, 换到对应的a并打印
```

println 默认调用 tostring 方法 write 接收字符, 按照码表输出

```
Person p = null;
ps.println(p); //如果是null, 就返回null, 如果不是null, 就调用对象的toString() 输出 null
```

```
PrintWriter pw = new PrintWriter("f.txt");
pw.println(97);
pw.write(97);
```

```
pw.close(); 输出 97 a
```

```
InputStream is = System.in;
int x = is.read();
System.out.println(x);

is.close(); read 只读第一个字节 输入 48 读取 4 输出 52 (码表中 4 对应 52)
```

改变默认的输入输出流目的地

```
System.setIn(new FileInputStream("a.txt")); //改变标准输入流
System.setOut(new PrintStream("b.txt")); //改变标注输出流
```

```
InputStream is = System.in; //获取标准的键盘输入流, 默认指向键盘, 改变后指向文件
PrintStream ps = System.out; //获取标准输出流, 默认指向的是控制台, 改变后就指向文件
int b;
while((b = in.read()) != -1) { //从a.txt上读取数据
    ps.write(b); //将数据写到b.txt上
}

in.close();
ps.close();
```

BufferedReader 也可以对 System.in 进行包装 读取键盘输入 但更加强大 readLine 读取一行

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); //InputStreamReader转换流
String line = br.readLine();
System.out.println(line);
br.close();
输入 123 输出 123
```

```
Scanner sc = new Scanner(System.in);
String line = sc.nextLine();
System.out.println(line);
sc.close();
```

第二种方法 若无与硬盘间交互，无需关闭

Properties

是一种双列集合，表示持久的属性集， hashtable 的子类(无序)，可以保存在流中或从流中加载
key 和 value 都是字符串，已重写 tostring 方法

```
Properties prop = new Properties();
prop.put("abc", 123);
System.out.println(prop);
```

输出 {abc=123}

Properties的特殊功能

```
* public Object setProperty(String key, String value)
* public String getProperty(String key)
* public Enumeration<String> stringPropertyNames()
```

拿到所有的键，把键存到枚举里，遍历它，通过键获取值

Properties 里默认接收 object，所以用别的东西接收时需要强转

```
Properties prop = new Properties();
prop.setProperty("name", "张三");
prop.setProperty("tel", "18912345678");
```

```
//System.out.println(prop);
Enumeration<String> en = (Enumeration<String>) prop.propertyNames();
while(en.hasMoreElements()) {
    String key = en.nextElement(); //获取Properties中的每一个键
    String value = prop.getProperty(key); //根据键获取值
    System.out.println(key + "=" + value);
}
```

这些操作都是在内存中，但是配置文件都在硬盘上，所以需要硬盘和内存通过 IO 流进行交互

Demo10_Properties.java config.properties

```
1 username=zhangsan
2 tel=18987654321
3 qq=12345
```

在项目下建立一个 txt 或 pro 的文件 输入配置

把键值对读入双列集合中

```
Properties prop = new Properties();
System.out.println("读取前: " + prop);
prop.load(new FileInputStream("config.properties"));

prop.setProperty("tel", "18988888888");
System.out.println("读取后: " + prop);
```

注意你 set 更改的东西改的是内存中的，硬盘中(txt 文件)并没有被修改，使用 store 更改硬盘中的
prop.store(new FileOutputStream("config.properties"), null);
System.out.println("读取后: " + prop);

例题

需求:1, 从键盘接收一个文件夹路径, 统计该文件夹大小

从键盘接收一个文件夹路径

- 1, 创建键盘录入对象
- 2, 定义一个无限循环
- 3, 将键盘录入的结果存储并封装成File对象
- 4, 对File对象判断
- 5, 将文件夹路径对象返回

统计该文件夹大小

- 1, 定义一个求和变量
- 2, 获取该文件夹下所有的文件和文件夹listFiles();
- 3, 遍历数组
- 4, 判断是文件就计算大小并累加
- 5, 判断是文件夹, 递归调用

```
public static File getDir() {  
    //1, 创建键盘录入对象  
    Scanner sc = new Scanner(System.in);  
    System.out.println("请输入一个文件夹路径:");  
    //2, 定义一个无限循环  
    while(true) {  
        //3, 将键盘录入的结果存储并封装成File对象  
        String line = sc.nextLine();  
        File dir = new File(line);  
        //4, 对File对象判断  
        if(!dir.exists()) {  
            System.out.println("您录入的文件夹路径不存在, 请输入一个文件夹路径:");  
        } else if(dir.isFile()) {  
            System.out.println("您录入的是文件路径, 请输入一个文件夹路径:");  
        } else {  
            //5, 将文件夹路径对象返回  
            return dir;  
        }  
    }  
}  
  
public static long getFileLength(File dir) {  
    //1, 定义一个求和变量  
    long len = 0;  
    //2, 获取该文件夹下所有的文件和文件夹listFiles();  
    File[] subFiles = dir.listFiles();  
    //3, 遍历数组  
    for (File subFile : subFiles) {  
        //4, 判断是文件就计算大小并累加  
        if(subFile.isFile()) {  
            len = len + subFile.length();  
        } //5, 判断是文件夹, 递归调用  
        else {  
            len = len + getFileLength(subFile);  
        }  
    }  
    return len;  
}  
  
public static void main(String[] args) {  
    File dir = getDir();  
    System.out.println(getFileLength(dir));  
}
```

需求:2, 从键盘接收一个文件夹路径, 删除该文件夹

删除该文件夹

分析:

- 1, 获取该文件夹下的所有的文件和文件夹
- 2, 遍历数组
- 3, 判断是文件直接删除
- 4, 如果是文件夹, 递归调用
- 5, 循环结束后, 把空文件夹删掉

此删除不走回收站

```

public static void deleteFile(File dir) {
    //1, 获取该文件夹下的所有的文件和文件夹
    File[] subFiles = dir.listFiles();
    //2, 遍历数组
    for (File subFile : subFiles) {
        //3, 判断是文件直接删除
        if(subFile.isFile()) {
            subFile.delete();
        //4, 如果是文件夹, 递归调用
        }else {
            deleteFile(subFile);
        }
    }
    //5, 循环结束后, 把空文件夹删掉
    dir.delete();
}
public static void main(String[] args) {
    File dir = Test1.getDir();           //获取文件夹路径
    deleteFile(dir);
}

```

需求:3, 从键盘接收两个文件夹路径, 把其中一个文件夹中(包含内容)拷贝到另一个文件夹中

I

把其中一个文件夹中(包含内容)拷贝到另一个文件夹中

分析:

- 1, 在目标文件夹中创建原文件夹
- 2, 获取原文件夹中所有的文件和文件夹, 存储在File数组中
- 3, 遍历数组
- 4, 如果是文件就用io流读写
- 5, 如果是文件夹就递归调用

字符流只能拷贝文本, 所以此处使用字节流

```

public static void copy(File src, File dest) throws IOException {
    //1, 在目标文件夹中创建原文件夹
    File newDir = new File(dest, src.getName());
    newDir.mkdir();
    //2, 获取原文件夹中所有的文件和文件夹, 存储在File数组中
    File[] subFiles = src.listFiles();
    //3, 遍历数组
    for (File subFile : subFiles) {
        //4, 如果是文件就用io流读写
        if(subFile.isFile()) {
            BufferedInputStream bis = new BufferedInputStream(new FileInputStream(subFile));
            BufferedOutputStream bos =
                new BufferedOutputStream(new FileOutputStream(new File(newDir, subFile.getName())));
            I
            int b;
            while((b = bis.read()) != -1) {
                bos.write(b);
            }
            bis.close();
            bos.close();
        //5, 如果是文件夹就递归调用
        }else {
            copy(subFile,newDir);
        }
    }
public static void main(String[] args) throws IOException {
    File src = Test1.getDir();
    File dest = Test1.getDir();
    if(src.equals(dest)) {
        System.out.println("目标文件夹是源文件夹的子文件夹");
    }else {
        copy(src,dest);
    }
}

```

需求：4，从键盘接收一个文件夹路径，把文件夹中的所有文件以及文件夹的名字按层级打印，例如：

把文件夹中的所有文件以及文件夹的名字按层级打印

分析：

1, 获取所有文件和文件夹，返回的File数组

2, 遍历数组

3, 无论是文件还是文件夹，都需要直接打印

4, 如果是文件夹，递归调用

```
public static void printLev(File dir, int lev) {  
    //1, 把文件夹中的所有文件以及文件夹的名字按层级打印  
    File[] subFiles = dir.listFiles();  
    //2, 遍历数组  
    for (File subFile : subFiles) {  
        for (int i = 0; i <= lev; i++) {  
            System.out.print("\t");  
        }  
        //3, 无论是文件还是文件夹，都需要直接打印  
        System.out.println(subFile);  
        //4, 如果是文件夹，递归调用  
        if (subFile.isDirectory()) {  
            printLev(subFile, lev + 1);  
        }  
    }  
}  
public static void main(String[] args) {  
    File dir = Test1.getDir();  
    printLev(dir, 0);  
}
```

递归练习

斐波那契数列

```
1 1 2 3 5 8 13 21  
1 = fun(1)  
1 = fun(2)  
2 = fun(1) + fun(2)  
3 = fun(2) + fun(3)  
public static int fun(int num) {  
    if (num == 1 || num == 2) {  
        return 1;  
    } else {  
        return fun(num - 2) + fun(num - 1);  
    }  
}
```

求 1000 的阶乘所有零和尾部零的个数

```
/*int result = 1;  
for (int i = 1; i <= 1000; i++) {  
    result = result * i;  
}  
  
System.out.println(result); //因为1000的阶乘远远超出了int的取值范围  
*/
```

不能直接做，超出 int 取值范围

获取所有 0

```
BigInteger bi1 = new BigInteger("1");  
for (int i = 1; i <= 1000; i++) {  
    BigInteger bi2 = new BigInteger(i + "");  
    bi1 = bi1.multiply(bi2); //将bi1与bi2相乘的结果赋值给bi1  
}
```

BigInteger 只能接收字符串(通过 int 加空字符串解决)

```
String str = bi1.toString(); //获取字符串表现形式  
int count = 0;  
for (int i = 0; i < str.length(); i++) {  
    if ('0' == str.charAt(i)) { //如果字符串中出现了0字符  
        count++; //计数器加1  
    }  
}
```

获得尾部 0 个数

```
String str = bil.toString();      //获取字符串表现形式
StringBuilder sb = new StringBuilder(str);
str = sb.reverse().toString();    //链式编程

int count = 0;                  //定义计数器
for(int i = 0; i < str.length(); i++) {
    if('0' != str.charAt(i)) {
        break;
    }else {
        count++;
    }
}
```

约瑟夫环

```
public static int getLuckyNum(int num) {
    ArrayList<Integer> list = new ArrayList<>();           //创建集合存储1到num的对象
    for(int i = 1; i <= num; i++) {                           //将1到num存储在集合中
        list.add(i);
    }

    int count = 1;                                         //用来数数的，只要是3的倍数就杀人
    for(int i = 0; list.size() != 1; i++) {                //只要集合中人数超过1，就要不断的杀
        if(i == list.size()) {                            //如果i增长到集合最大的索引+1时
            i = 0;                                       //重新归零
        }

        if(count % 3 == 0) {                            //如果是3的倍数
            list.remove(i--);                          //就杀人
        }
        count++;
    }

    return list.get(0);
}
```

多线程

线程是程序执行的一条路径，一个进程可以包含多条线程
多线程并发执行可以提高程序效率，同时完成多项工作



cpu 其实每次只执行一个程序，只不过切换快，使看起来像同时执行

并行和并发

并行就是两个任务同时运行，就是甲任务进行的同时，乙任务也在进行。（需要多核CPU）
并发是指两个任务都请求运行，而处理器只能接受一个任务，就把这两个任务安排轮流进行，由于时间间隔较短，使人感觉两个任务都在运行。比如我跟两个网友聊天，左手操作一个电脑跟甲聊，同时右手用另一台电脑跟乙聊天，这就叫并行。
如果用一台电脑我先给甲发个消息，然后立刻再给乙发消息，然后再跟甲聊，再跟乙聊。这就叫并发。

Java 启动的运行原理

Java 命令启动虚拟机---启动 JVM 相当于启动一个进程---该进程会自动启动一个“主线程”---此主线程调用某个类的 main 方法

A: Java 程序运行原理

* Java 命令会启动 java 虚拟机，启动 JVM，等于启动了一个应用程序，也就是启动了一个进程。该进程会自动启动一个“主线程”，然后主线程去调用某个类的 main 方法。

B: JVM 的启动是多线程的吗

* JVM 启动至少启动了垃圾回收线程和主线程，所以是多线程的。

创建多线程的两种方法

第一种：继承 thread 重写其中的 run()

```
class MyThread extends Thread {           //1, 继承Thread
    public void run() {                  //2, 重写run方法
        for(int i = 0; i < 1000; i++) {   //3, 将要执行的代码写在run方法中
            System.out.println("aaaaaaaaaaaa");
        }
    }
    public static void main(String[] args) {
        MyThread mt = new MyThread();      //4, 创建Thread类的子类对象
        mt.start();                      //5, 开启线程
    }
}
for(int i = 0; i < 1000; i++) {
    System.out.println("bb");
}
```

结果 aaa 和 bb 间隔输出

若没有用 start 而是用 run，则只会主线程运行，因为并没有启动线程，启动线程要使用 start()，若不启动直接用 run，将不会多线程运行，只是主线程运行，结果会为 aaaaaaaaa 后 bbbbbbbb

第二种：实现 runnable 接口

```
class MyRunnable implements Runnable {      //1, 定义一个类实现Runnable
    @Override
    public void run() {                      //2, 重写run方法
        for(int i = 0; i < 1000; i++) {     //3, 将要执行的代码写在run方法中
            System.out.println("aaaaaaaaaaaa");
        }
    }
    public static void main(String[] args) {
        MyRunnable mr = new MyRunnable();    //4, 创建Runnable的子类对象
        //Runnable target = mr;
        Thread t = new Thread(mr);          //5, 将其当作参数传递给Thread的构造函数
        t.start();                         //6, 开启线程
    }
}
for(int i = 0; i < 1000; i++) {
    System.out.println("bb");
}
```

实现 runnable 原理

[查看源码](#)

- * 1, 看 Thread 类的构造函数，传递了 Runnable 接口的引用
- * 2, 通过 init() 方法找到传递的 target 给成员变量的 target 赋值
- * 3, 查看 run 方法，发现 run 方法中有判断，如果 target 不为 null 就会调用 Runnable 接口子类对象的 run 方法

两种方式区别

[查看源码的区别](#)

- * a. 继承 Thread：由于子类重写了 Thread 类的 run()，当调用 start() 时，直接找子类的 run() 方法
- * b. 实现 Runnable：构造函数中传入了 Runnable 的引用，成员变量记住了它，start() 调用 run() 方法时内部判断成员变量 Runnable 的引用是否为空，不为空编译时看的是 Runnable 的 run()，运行时执行的是子类的 run() 方法

先判断这个类有无父类，有父类用 runnable，无父类用 thread

继承Thread

- * 好处是：可以直接使用Thread类中的方法，代码简单
- * 弊端是：如果已经有了父类，就不能用这种方法

实现Runnable接口

- * 好处是：即使自己定义的线程类有了父类也没关系，因为有了父类也可以实现接口，而且接口是可以多实现的
- * 弊端是：不能直接使用Thread中的方法需要先获取到线程对象后，才能得到Thread的方法，代码复杂

匿名内部类实现线程的两种方式

```
一:  
public static void main(String[] args) {  
    new Thread() {  
        public void run() {  
            for(int i = 0; i < 1000; i++) {  
                System.out.println("aaaaaaaaaaaaaa");  
            }  
        }  
    }.start();  
}  
二:  
new Thread(new Runnable() {  
    public void run() {  
        for(int i = 0; i < 1000; i++) {  
            System.out.println("bb");  
        }  
    }  
}).start();
```

//1, 继承Thread类
//2, 重写run方法
//3, 将要执行的代码写在run方法中
//4, 开启线程

线程方法

获取线程名字 getname

```
new Thread() {  
    public void run() {  
        System.out.println(this.getName() + "....aaaaaaaaa");  
    }  
}.start();
```

this 表示哪个对象调用它，它代表谁，名字

默认为 Thread-0，再开启一个为 Thread-1

设置名字两种方法

1.构造方法

```
new Thread("芙蓉姐姐") {  
    public void run() {  
        System.out.println(this.getName() + "....aaaaaaaaa");  
    }  
}.start();
```

2.setName

```
new Thread() {  
    public void run() {  
        this.setName("张三");  
        System.out.println(this.getName() + "....aaaaaaaaaaa");  
    }  
}.start();
```

获取当前线程对象

Thread.currentThread()

```

public static void main(String[] args) {
    new Thread() {
        public void run() {
            System.out.println(getName() + "....aaaaaa");
        }
    }.start();

    new Thread(new Runnable() {
        public void run() {
            //Thread.currentThread() 获取当前正在执行的线程
            System.out.println(Thread.currentThread().getName() + "...bb");
        }
    }).start();
}

System.out.println(Thread.currentThread().getName());
}

```

Thread-0....aaaaaa
main
Thread-1...bb

此方法中一共有三条线程 主线程 线程 0 线程 1

休眠线程

Thread.sleep(毫秒) 控制当前线程休眠若干时间 $1s = 1000ms = 1 \times 10^9$ s

```

public static void main(String[] args) throws InterruptedException {
    for(int i = 20; i >= 0; i--) {
        Thread.sleep(1000);
        System.out.println("倒计时第" + i + "秒");
    }
}

```

此方法需要抛出休眠异常

此例中休眠的是主线程

此例中休眠自定义线程

```

new Thread() {
    public void run() {
        for(int i = 0; i < 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {

                e.printStackTrace();
            }
            System.out.println(getName() + "...aaaaaaaaaa");
        }
    }
}.start();

```

父类 thread 中的 run 方法是没有抛异常的，子类在重写 run 方法的时候就不能抛异常

父亲坏了，儿子不能比父亲更坏，父亲如果没坏，儿子坏了必须自己处理，所以此处需要 try-catch 而不能 throw 中断异常

守护线程

setDaemon(true)

---当被守护线程挂掉守护线程也不做了

```

public static void main(String[] args) {
    Thread t1 = new Thread() {
        public void run() {
            for(int i = 0; i < 2; i++) {
                System.out.println(getName() + "...aaaaaaaaaaaaaaaaaa");
            }
        }
    };

    Thread t2 = new Thread() {
        public void run() {
            for(int i = 0; i < 50; i++) {
                System.out.println(getName() + "...bb");
            }
        }
    };

    t2.setDaemon(true); //当传入true就是意味着设置为守护线程
    t1.start();
    t2.start();
}

```

加入线程 join() 当前线程暂停，等待指定的线程执行结束后，当前线程再继续(同样需要 try-catch 中断异常)

t1.join(int) 在 t1 插队指定的毫秒后 t1 t2 继续交替运行

注：匿名内部类在使用它所在方法中的局部变量时，局部变量(t1)必须用 final 修饰

```

final Thread t1 = new Thread() {
    public void run() {
        for(int i = 0; i < 10; i++) {
            System.out.println(getName() + "...aaaaaaaaaaaaa");
        }
    }
};

Thread t2 = new Thread() {
    public void run() {
        for(int i = 0; i < 10; i++) {
            if(i == 2) {
                try {
                    t1.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println(getName() + "...bb");
        }
    }
}

```

刚开始时 t1, t2 同时进行，当执行到 join 时，t2 暂停，让 t1 插队，t1 结束后 t2 继续

设置线程优先级 SetPriority(1~10) 最小 1 最大 10 默认 5

同步代码块

```

public static void main(String[] args) {
    final Printer p = new Printer();

    new Thread() {
        public void run() {
            while(true) {
                p.print1();
            }
        }
    }.start();

    new Thread() {
        public void run() {
            while(true) {
                p.print2();
            }
        }
    }.start();
}

不要忘记匿名内部类需要使用 final 修饰的局部对象

```

```

class Printer {
    public void print1() {
        System.out.print("黑");
        System.out.print("马");
        System.out.print("程");
        System.out.print("序");
        System.out.print("员");
        System.out.print("\r\n");
    }

    public void print2() {
        System.out.print("传");
        System.out.print("智");
        System.out.print("播");
        System.out.print("客");
        System.out.print("\r\n");
    }
}

```

可能黑马到一半就开始传智了，这个时候需要加一个锁，锁对象是任意的，只要传入的是对象就行

```

public void print1() {
    synchronized(new Demo()) { //同步代码块，锁机制，锁对象可以是任意的
        System.out.print("黑");
        System.out.print("马");
        System.out.print("程");
        System.out.print("序");
        System.out.print("员");
        System.out.print("\r\n");
    }
}

public void print2() {
    synchronized(new Demo()) { //锁对象不能用匿名对象，因为匿名对象不是同一个对象
        System.out.print("传");
        System.out.print("智");
        System.out.print("播");
        System.out.print("客");
        System.out.print("\r\n");
    }
}

```

synchronized(){ … } 同步代码块 不能传入匿名对象，需要是同一个对象

同步方法

只需要在方法上加 synchronized 关键字即可

```

//非静态的同步方法的锁对象是黑马？
//答：非静态的同步方法的锁对象是this
public synchronized void print1() {
    System.out.print("黑");
    System.out.print("马");
    System.out.print("程");
    System.out.print("序");
    System.out.print("员");
    System.out.print("\r\n");
}

```

```

public void print2() {
    //synchronized(new Demo()) {
    synchronized(this) {
        System.out.print("传");
        System.out.print("智");
        System.out.print("播");
        System.out.print("客");
        System.out.print("\r\n");
    }
}

```

非静态方法 谁调用 this, this 就是谁对象, synchronized 调用, 下面的直接 this 就可表示传入的是同一个对象

```

class Printer2 {
    Demo d = new Demo();
    //非静态的同步方法的锁对象是神马?
    //答:非静态的同步方法的锁对象是this
    //静态的同步方法的锁对象是什么?
    //是该类的字节码对象
    public static synchronized void print1() {
        System.out.print("黑");
        System.out.print("马");
        System.out.print("程");
        System.out.print("序");
        System.out.print("员");
        System.out.print("\r\n");
    }
}

public static void print2() {
    //synchronized(new Demo()) {
    synchronized(Printer2.class) {
        System.out.print("传");
        System.out.print("智");
        System.out.print("播");
        System.out.print("客");
        System.out.print("\r\n");
    }
}

```

静态方法的锁对象

例题

售票 100 张，分四个窗口卖 (线程安全问题)

```

class Ticket extends Thread {
    private static int ticket = 100;

    public void run() {
        while(true) {
            if(ticket == 0) {
                break;
            }
            System.out.println(getName() + "...这是第" + ticket-- + "号票");
        }
    }

    public static void main(String[] args) {
        new Ticket().start();
        new Ticket().start();
        new Ticket().start();
        new Ticket().start();
    }
}

```

若不加 static 则每个线程都有各自的 100 ticket，需要 static 四个线程共享 ticket

此时还有问题，若 1 线程执行，此时剩 1 张，在过了 if 判断后，此时被线程 2 抢去，此时还是 1 张，又被线程 3 抢去，此时还是 1 张，在都过了判断后，然后线程 1 继续，0 张，线程 2 继续，为 -1 张，线程 3 为 -2 张

```

class Ticket extends Thread {
    private static int ticket = 100;

    public void run() {
        while(true) {
            synchronized(Ticket.class) {
                if(ticket <= 0) {
                    break;
                }
                try {
                    Thread.sleep(10);           //线程1睡,线程2睡,线程3睡,线程4睡
                } catch (InterruptedException e) {

```

需要同步代码块锁住

```

class Ticket extends Thread {
    private static int ticket = 100;
    private Object obj = new Object();
    public void run() {
        while(true) {
            synchronized(obj) {
                if(ticket <= 0) { 不可以这样，因为你 new 了 4 个 ticket，相当于 4 个不同的对象
但是类是唯一的，若非要使用，可以加 static //private static Object obj = new Object();}

```

用 runnable 接口实现 此时无需 static 了，因为 myticket 是 thread 子类，只需要一个 thread 就可以了

```

public static void main(String[] args) {
    MyTicket mt = new MyTicket();
    new Thread(mt).start();
    new Thread(mt).start();
    new Thread(mt).start();
    new Thread(mt).start();
}

class MyTicket implements Runnable {
    private int tickets = 100;
    @Override
    public void run() {
        while(true) {
            synchronized(Ticket.class) {
                if(tickets <= 0) {
                    break;
                }
                try {
                    Thread.sleep(10); //线程1睡，线程2睡，线程3睡，线程4睡
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + "...这是第" + tickets-- + "号票");
            }
        }
    }
}

```

因为你创建对象只有 1 次 只有一个 mt

死锁

```

private static String s1 = "筷子左";
private static String s2 = "筷子右";
public static void main(String[] args) {
    new Thread() {
        public void run() {
            while(true) {
                synchronized(s1) {
                    System.out.println(getName() + "...获取" + s1 + "等待" + s2);
                    synchronized(s2) {
                        System.out.println(getName() + "...拿到" + s2 + "开吃");
                    }
                }
            }
        }
    }.start();

    new Thread() {
        public void run() {
            while(true) {
                synchronized(s2) {
                    System.out.println(getName() + "...获取" + s2 + "等待" + s1);
                    synchronized(s1) {
                        System.out.println(getName() + "...拿到" + s1 + "开吃");
                    }
                }
            }
        }
    }.start();
}

```

回顾 collection

Vector 是线程安全的, ArrayList 线程不安全 add()
StringBuffer 线程安全, StringBuilder 线程不安全 append()
HashTable 线程安全, HashMap 线程不安全 put()
Collections.synchronized(xxx(list/set/map)) 可以将线程不安全的变成安全的---淘汰了 vector

Runtime 类(单例类)

```
public static void main(String[] args) throws IOException {
    Runtime r = Runtime.getRuntime(); //获取运行时对象
    //r.exec("shutdown -s -t 300");
    r.exec("shutdown -a"); I
}
```

Timer 计时器

```
class MyTimerTask extends TimerTask {
    @Override
    public void run() {
        System.out.println("起床背英语单词");
    }
}
指定时间开始执行一次
public static void main(String[] args) throws InterruptedException {
    Timer t = new Timer();
    t.schedule(new MyTimerTask(), new Date(188, 6, 1, 14, 21, 30));
}
指定时间开始执行一次, 间隔 3s 后反复执行
t.schedule(new MyTimerTask(), new Date(188, 6, 1, 14, 22, 50), 3000);
```

两线程间通信

等待唤醒机制 wait-notify

```
//等待唤醒机制
class Printer {
    private int flag = 1;
    public void print1() throws InterruptedException {
        synchronized(this) {
            if(flag != 1) {
                this.wait();
            }
            System.out.print("黑");
            System.out.print("马");
            System.out.print("程");
            System.out.print("序");
            System.out.print("员");
            System.out.print("\r\n");
            flag = 2; I
            this.notify(); //随机唤醒单个等待的线程
        }
    }
    public void print2() throws InterruptedException {
        synchronized(this) {
            if(flag != 2) {
                this.wait();
            }
            System.out.print("传");
            System.out.print("智");
            System.out.print("播");
            System.out.print("客");
            System.out.print("\r\n");
            flag = 1;
            this.notify();
        }
    }
}
```

多个线程间通信

将 this.notify() 改为 this.notifyAll() 将 if 改为 while

注意：在同步代码块中，用什么对象调用锁，就用什么对象调用 wait 和 notify 方法

2, 为什么 wait 方法和 notify 方法定义在 Object 这类中？

因为锁对象可以是任意对象，Object 是所有的类的基类，所以 wait 方法和 notify 方法需要定义在 Object 这个类中

3, sleep 方法和 wait 方法的区别？

a, sleep 方法必须传入参数，参数就是时间，时间到了自动醒来

wait 方法可以传入参数也可以不传入参数，传入参数就是在参数的时间结束后等待，不传入参数就是直接等待

b, sleep 方法在同步函数或同步代码块中，不释放锁，睡着了也抱着锁睡

wait 方法在同步函数或者同步代码块中，释放锁

Sleep 代表卡在这，别人也不准动，wait 代表别的线程可以开始执行

互斥锁 ReentrantLock

```
public static void main(String[] args) {  
    final Printer3 p = new Printer3();  
  
    new Thread() {  
        public void run() {  
            while(true) {  
                try {  
                    p.print1();  
                } catch (InterruptedException e) {  
  
                    e.printStackTrace();  
                }  
            }  
        }  
    }.start();
```

创建 3 个 Thread 分别运行 print123 方法

```
class Printer3 {  
    private ReentrantLock r = new ReentrantLock();  
    private Condition c1 = r.newCondition();  
    private Condition c2 = r.newCondition();  
    private Condition c3 = r.newCondition();  
  
    private int flag = 1;  
    public void print1() throws InterruptedException {  
        r.lock();  
        if(flag != 1) {  
            c1.await();  
        }  
        System.out.print("黑");  
        System.out.print("马");  
        System.out.print("程");  
        System.out.print("序");  
        System.out.print("员");  
        System.out.print("\r\n");  
        flag = 2;  
        //this.notify();  
        c2.signal();  
    }  
    public void print2() throws InterruptedException {  
        r.lock();  
        if(flag != 2) {  
            c2.await();  
        }  
        System.out.print("传");  
        System.out.print("智");  
        System.out.print("播");  
        System.out.print("客");  
        System.out.print("\r\n");  
        flag = 3;  
        //this.notify();  
        c3.signal();  
    }  
}
```

在指定类中先创建 ReentrantLock，几个线程创建几个条件 接着第一个线程用 r.lock 和 r.unlock 来代替同步代码块，在锁中 c1.await 后 c2.signal 唤醒 c2 然后 c2await c3.signal 唤醒 c3，接着 c3 唤醒 c1

线程组 ThreadGroup

```
public static void main(String[] args) {  
    MyRunnable mr = new MyRunnable();  
    Thread t1 = new Thread(mr, "张三");  
    Thread t2 = new Thread(mr, "李四");  
  
    ThreadGroup tg1 = t1.getThreadGroup();  
    ThreadGroup tg2 = t2.getThreadGroup();  
  
    System.out.println(tg1.getName()); //默认的是主线程  
    System.out.println(tg2.getName());  
}
```

所有线程默认是主线程组 main

```
public static void main(String[] args) {  
    //demo1();  
    ThreadGroup tg = new ThreadGroup("我是一个新的线程组"); //创建新的线程组  
    MyRunnable mr = new MyRunnable(); //创建Runnable的子类对象  
  
    Thread t1 = new Thread(tg, mr, "张三"); //将线程t1放在组中  
    Thread t2 = new Thread(tg, mr, "李四"); //将线程t2放在组中  
  
    System.out.println(t1.getThreadGroup().getName()); //获取组名  
    System.out.println(t2.getThreadGroup().getName());  
}
```

Thread 其中一个构造方法是接收线程组, runnable, 线程名 就是将此线程加到线程组中 结果“wsxdxcz”

线程组的好处是可以整组设置 如 tg.setDaemon(true) 守护线程 或 设置优先级等

线程的五种状态

新键 就绪 运行 阻塞 死亡

线程的生命周期：

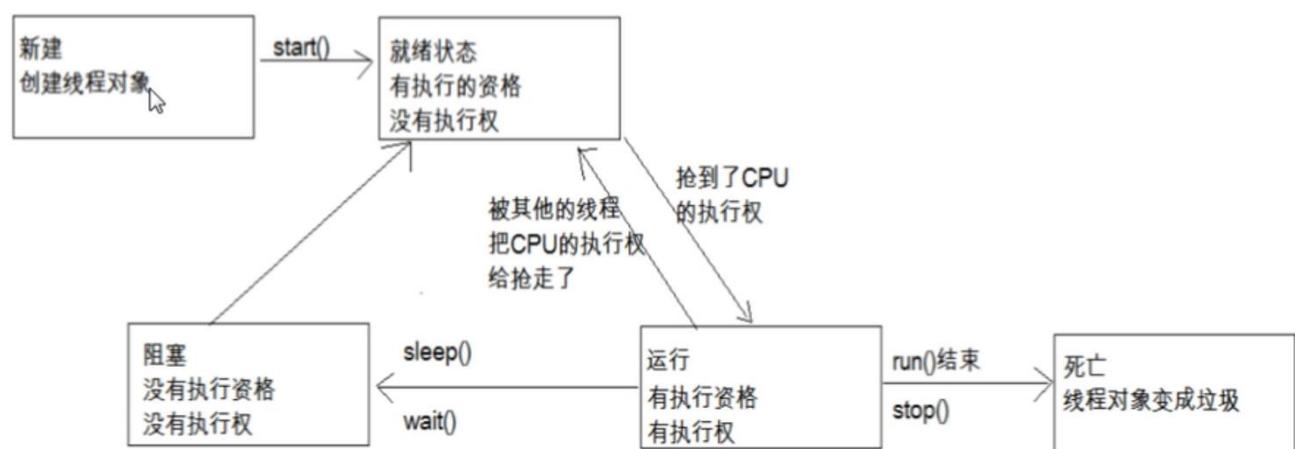
新建 创建线程对象

就绪 线程对象已经启动了，但是还没有获取到CPU的执行权

运行 获取到了CPU的执行权

阻塞：没有CPU的执行权 回到就绪

死亡 代码运行完毕，线程消亡



线程池

线程池概述

* 程序启动一个新线程成本是比较高的，因为它涉及到要与操作系统进行交互。而使用线程池可以很好的提高性能，尤其是当程序中要创建大量生存期很短的线程时，更应该考虑使用线程池。线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。在JDK5之前，我们必须手动实现自己的线程池，从JDK5开始，Java内置支持线程池 T

内置线程池的使用概述

- * JDK5新增了一个Executors工厂类来产生线程池，有如下几个方法
 - * public static ExecutorService newFixedThreadPool(int nThreads)
 - * public static ExecutorService newSingleThreadExecutor()
 - * 这些方法的返回值是ExecutorService对象，该对象表示一个线程池，可以执行Runnable对象或者Callable对象代表的线程。它提供了如下方法
 - * Future<?> submit(Runnable task)
 - * <T> Future<T> submit(Callable<T> task)
- * 使用步骤：
 - * 创建线程池对象
 - * 创建Runnable实例
 - * 提交Runnable实例
 - * 关闭线程池

```

public static void main(String[] args) {
    ExecutorService pool = Executors.newFixedThreadPool(2); //创建线程池
    pool.submit(new MyRunnable()); //将线程放进池子里并执行
    pool.submit(new MyRunnable());

    pool.shutdown(); //关闭线程池
}

```

线程的第三种实现方式 Callable

简单工厂模式

创建一个工厂(父类)专门用来对各个子类对象的创建

简单工厂模式概述

* 又叫静态工厂方法模式，它定义一个具体的工厂类负责创建一些类的实例

优点

* 客户端不需要在负责对象的创建，从而明确了各个类的职责

缺点

* 这个静态工厂类负责所有对象的创建，如果有新的对象增加，或者某些对象的创建方式不同，就需要不断的修改工厂类，不利于后期的维护

案例演示

* 动物抽象类：public abstract Animal { public abstract void eat(); }

* 具体狗类：public class Dog extends Animal {}

* 具体猫类：public class Cat extends Animal {}

* 开始，在测试类中每个具体的内容自己创建对象，但是，创建对象的工作如果比较麻烦，就需要有人专门做这个事情，所以就知道了一个专门的类来创建对象。

Test:

```

public static void main(String[] args) {
    //Dog d = AnimalFactory.createDog();

    Dog d = (Dog) AnimalFactory.createAnimal("dog");
    d.eat();

    Cat c = (Cat) AnimalFactory.createAnimal("cat");
    c.eat();
}

```

工厂类：

```

public static Animal createAnimal(String name) {
    if("dog".equals(name)) {
        return new Dog();
    }else if("cat".equals(name)) {
        return new Cat();
    }else {
        return null;
    }
}

```

动物类

猫类

```

public class Cat extends Animal {

    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
}

public abstract class Animal {
    public abstract void eat();
}

```

工厂方法模式

有一个总工厂，若要猫就创建一个猫工厂，要狗就创个狗工厂，再由专属的工厂创建实例
好处是不用修改猫狗本类的代码

工厂方法模式概述

- * 工厂方法模式中抽象工厂类负责**定义创建对象的接口**，具体对象的创建工作由继承抽象工厂的具体类实现。

优点

- * 客户端不需要在负责对象的创建，从而明确了各个类的职责，如果有新的对象增加，只需要增加一个具体的类和具体的工厂类即可，不影响已有的代码，后期维护容易，增强了系统的扩展性

缺点

- * 需要额外的编写代码，增加了工作量

```

动物抽象类: public abstract Animal { public abstract void eat(); }
工厂接口: public interface Factory { public abstract Animal createAnimal(); }
具体狗类: public class Dog extends Animal {}
具体猫类: public class Cat extends Animal {}
开始，在测试类中每个具体的内容自己创建对象，但是，创建对象的工作如果比较麻烦，就需要有人专门做这个事情，所以就知道了一个专门的类来创建对象。发现每次修改代码太麻烦，用工厂方法改进，针对每一个具体的实现提供一个具体工厂。
狗工厂: public class DogFactory implements Factory {
    public Animal createAnimal() { ... }
}
猫工厂: public class CatFactory implements Factory {
    public Animal createAnimal() { ... }
}
```

工厂接口

猫工厂

```

public class CatFactory implements Factory {

    @Override
    public Animal createAnimal() {
        return new Cat();
    }

    public interface Factory {
        public Animal createAnimal();
    }
}
```

Test

```

public static void main(String[] args) {
    DogFactory df = new DogFactory();
    Dog d = (Dog) df.createAnimal();
    d.eat();
```

好处就是为了模块化

适配器设计模式

如监听器是个接口，有许多方法需要重写，我们可以用适配器重写所有方法为空，然后只重写适配器中需要的方法

什么是适配器

- * 在使用监听器的时候，需要定义一个类事件监听器接口。
 - * 通常接口中有多个方法，而程序中不一定所有的都用到，但又必须重写，这很繁琐。
 - * 适配器简化了这些操作，我们定义监听器时只要继承适配器，然后重写需要的方法即可。
- 适配器原理**
- * 适配器就是一个类，实现了监听器接口，所有抽象方法都重写了，但是方法全是空的。
 - * 适配器类需要定义成抽象的，因为创建该类对象，调用空方法是没有意义的。
 - * 目的就是为了简化程序员的操作，定义监听器时继承适配器，只重写需要的方法就可以了。

Object 类方法

<code>protected Object</code>	<code>clone()</code>	创建并返回此对象的一个副本。
<code>boolean</code>	<code>equals(Object obj)</code>	指示其他某个对象是否与此对象“相等”。
<code>protected void</code>	<code>finalize()</code>	当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
<code>Class<?></code>	<code>getClass()</code>	返回此 Object 的运行时类。
<code>int</code>	<code>hashCode()</code>	返回该对象的哈希码值。
<code>void</code>	<code>notify()</code>	唤醒在此对象监视器上等待的单个线程。
<code>void</code>	<code>notifyAll()</code>	唤醒在此对象监视器上等待的所有线程。
<code>String</code>	<code>toString()</code>	返回该对象的字符串表示。
<code>void</code>	<code>wait()</code>	在其他线程调用此对象的 <code>notify()</code> 方法或 <code>notifyAll()</code> 方法前，导致当前线程等待。
<code>void</code>	<code>wait(long timeout)</code>	在其他线程调用此对象的 <code>notify()</code> 方法或 <code>notifyAll()</code> 方法，或者超过指定的时间量前，导致当前线程等待。
<code>void</code>	<code>wait(long timeout, int nanos)</code>	在其他线程调用此对象的 <code>notify()</code> 方法或 <code>notifyAll()</code> 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

网络编程

IP

每个设备在网络中的唯一标识

每台网络终端在网络中都有一个独立的地址，我们在网络中传输数据就是使用这个地址。

ipconfig : 查看本机IP192.168.12.42

ping : 测试连接192.168.40.62

IPv4 : 4个字节组成，4个0-255。大概42亿，30亿都在北美，亚洲4亿。2011年初已经用尽。

IPv6 : 8组，每组4个16进制数。

1a2b:0000:aaaa:0000:0000:aabb:1f2f

端口号

每个程序在设备上的唯一标识

每个网络程序都需要绑定一个端口号，传输数据的时候除了确定发到哪台机器上，还要明确发到哪个程序。

端口号范围从0-65535

编写网络应用就需要绑定一个端口号，尽量使用1024以上的，1024以下的基本上都被系统程序占用了。

常用端口

- * mysql: 3306
- * oracle: 1521
- * web: 80
- * tomcat: 8080
- * QQ: 4000
- * feiQ: 2425



Ip 相当于办公楼地址，端口号相当于房间号

TCP/UDP

为计算机网络中进行数据交换而建立的规则、标准或约定的集合。

UDP

面向无连接，数据不安全，速度快。不区分客户端与服务端。

TCP

- * 面向连接（三次握手），数据安全，速度略低。分为客户端和服务端。
- * 三次握手：客户端先向服务端发起请求，服务端响应请求，传输数据

UDP 相当于发短信，你可以不开机，号码也可以不存在，我只管发

TCP 相当于打电话，接听方必须在 三次握手：客户端请求---服务端响应---数据传输

Socket 通信

Socket套接字概述：

- * 网络上具有唯一标识的IP地址和端口号组合在一起才能构成唯一能识别的标识符套接字。
- * 通信的两端都有Socket。
- * 网络通信其实就是Socket间的通信。
- * 数据在两个Socket间通过IO流传输。
- * Socket在应用程序中创建，通过一种绑定机制与驱动程序建立关系，告诉自己所对应的IP和port。 socket 插座 port 端口号

UDP 传输

底层是 IO 流

1.发送Send

- * 创建DatagramSocket，随机端口号
- * 创建DatagramPacket，指定数据，长度，地址，端口
- * 使用DatagramSocket发送DatagramPacket
- * 关闭DatagramSocket

2.接收Receive

- * 创建DatagramSocket，指定端口号
- * 创建DatagramPacket，指定数组，长度
- * 使用DatagramSocket接收DatagramPacket
- * 关闭DatagramSocket
- * 从DatagramPacket中获取数据

3.接收方获取ip和端口号

- * String ip = packet.getAddress().getHostAddress();
- * int port = packet.getPort();

发送端：（发送给自己-主机名是自己）

```
public static void main(String[] args) throws Exception {
    String str = "what are you 弄啥呢?";
    DatagramSocket socket = new DatagramSocket(); //创建Socket相当于创建码头
    DatagramPacket packet = //创建Packet相当于集装箱
        new DatagramPacket(str.getBytes(), str.getBytes().length, InetAddress.getByName("127.0.0.1"), 666
    socket.send(packet); //发货,将数据发出去
    socket.close(); //关闭码头
}
```

接收端

```
public static void main(String[] args) throws Exception {
    DatagramSocket socket = new DatagramSocket(6666); //创建Socket相当于创建码头
    DatagramPacket packet = new DatagramPacket(new byte[1024], 1024); //创建Packet相当于创建集装箱
    socket.receive(packet); //接货,接收数据

    byte[] arr = packet.getData(); //获取数据
    int len = packet.getLength(); //获取有效的字节个数
    System.out.println(new String(arr, 0, len));
    socket.close();
}
```

优化(持续接收输入)

发送端不断发，遇到 quit 停止

```
public static void main(String[] args) throws Exception {
    Scanner sc = new Scanner(System.in); //创建键盘录入对象
    DatagramSocket socket = new DatagramSocket(); //创建Socket相当于创建码头

    while(true) {
        String line = sc.nextLine(); //获取键盘录入的字符串
        if("quit".equals(line)) {
            break;
        }
        DatagramPacket packet = //创建Packet相当于集装箱
            new DatagramPacket((line.getBytes(), line.getBytes().length, InetAddress.getByName("127.0.0.1"
        socket.send(packet); //发货,将数据发出去
    }
    socket.close(); //关闭码头
}
```

接收端

```
public static void main(String[] args) throws Exception {
    DatagramSocket socket = new DatagramSocket(6666); //创建Socket相当于创建码头
    DatagramPacket packet = new DatagramPacket(new byte[1024], 1024); //创建Packet相当于创建集装箱

    while(true) {
        socket.receive(packet); //接货,接收数据
        byte[] arr = packet.getData(); //获取数据
        int len = packet.getLength(); //获取有效的字节个数
        String ip = packet.getAddress().getHostAddress(); //获取ip地址
        int port = packet.getPort(); //获取端口号
        System.out.println(ip + ":" + port + ":" + new String(arr, 0, len));
    }
}
```

继续优化(UDP 传输多线程---同窗口发送接收)

接收端

```
class Receive extends Thread {
    public void run() {
        try {
            DatagramSocket socket = new DatagramSocket(6666); //创建Socket相当于创建码头
            DatagramPacket packet = new DatagramPacket(new byte[1024], 1024); //创建Packet相当于集装箱

            while(true) {
                socket.receive(packet); //接货,接收数据
                byte[] arr = packet.getData(); //获取数据
                int len = packet.getLength(); //获取有效的字节个数
                String ip = packet.getAddress().getHostAddress(); //获取ip地址
                int port = packet.getPort(); //获取端口号
                System.out.println(ip + ":" + port + ":" + new String(arr, 0, len));
            }
        }
    }
}
```

发送端

```

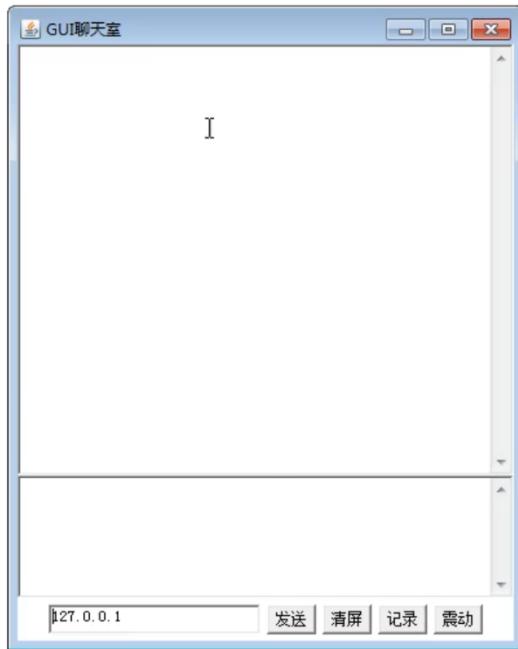
class Send extends Thread {
    public void run() {
        try {
            Scanner sc = new Scanner(System.in);
            DatagramSocket socket = new DatagramSocket();
            //创建键盘录入对象
            //创建Socket相当于创建端口

            while(true) {
                String line = sc.nextLine();
                //获取键盘录入的字符串
                if("quit".equals(line)) {
                    break;
                }
                DatagramPacket packet =
                    new DatagramPacket(line.getBytes(), line.getBytes().length, InetAddress.getByName("127.0.0.1"), 9999);
                //创建Packet相当于集装箱
                socket.send(packet);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

Main
public static void main(String[] args) {
    new Receive().start();
    new Send().start();
}

```

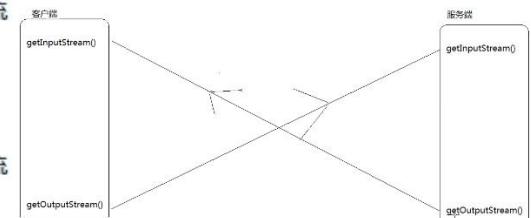
UDP 聊天程序



TCP 协议

客户端

- * 创建Socket连接服务端(指定ip地址,端口号)通过ip地址找对应的服务器
 - * 调用Socket的getInputStream()和getOutputStream()方法获取和服务端相连的IO流
 - * 输入流可以读取服务端输出流写出的数据
 - * 输出流可以写出数据到服务端的输入流
- 服务端**
- * 创建ServerSocket(需要指定端口号)
 - * 调用ServerSocket的accept()方法接收一个客户端请求，得到一个Socket
 - * 调用Socket的getInputStream()和getOutputStream()方法获取和客户端相连的IO流
 - * 输入流可以读取客户端输出流写出的数据
 - * 输出流可以写出数据到客户端的输入流



服务器端：

```

public static void main(String[] args) throws IOException {
    ServerSocket server = new ServerSocket(12345);

    Socket socket = server.accept();           I           //接受客户端的请求
    InputStream is = socket.getInputStream();      //获取客户端输入流
    OutputStream os = socket.getOutputStream();    //获取客户端的输出流

    os.write("百度一下你就知道".getBytes());        //服务器向客户端写出数据

    byte[] arr = new byte[1024];
    int len = is.read(arr);                     //读取客户端发过来的数据
    System.out.println(new String(arr, 0, len));   //将数据转换成字符串并打印

    socket.close();
}

```

客户端：登录网站表面登录域名，其实底层是登陆的 ip 地址

```

public static void main(String[] args) throws UnknownHostException, IOException {
    Socket socket = new Socket("127.0.0.1", 12345);

    InputStream is = socket.getInputStream();           //获取客户端输入流
    OutputStream os = socket.getOutputStream();         //获取客户端的输出流

    byte[] arr = new byte[1024];
    int len = is.read(arr);                          //读取服务器发过来的数据
    System.out.println(new String(arr, 0, len));     //将数据转换成字符串并打印

    os.write("学习挖掘机哪家好强?".getBytes());       I           //客户端向服务器写数据

    socket.close();
}

```

服务器 server 不需要关闭

代码优化

```

public static void main(String[] args) throws IOException {
    ServerSocket server = new ServerSocket(12345);

    Socket socket = server.accept();           //接受客户端的请求

    BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream())); //将字节流包装成了
    PrintStream ps = new PrintStream(socket.getOutputStream());                      //PrintStream中有写出换行的方法

    ps.println("欢迎咨询黑马程序员");
    System.out.println(br.readLine());
    ps.println("不好意思,爆满了");
    System.out.println(br.readLine());
    socket.close();
}

```

注意 readLine 是以/r/n 为结束标记 所以必须使用 println 而不是 print

```

public static void main(String[] args) throws UnknownHostException, IOException {
    Socket socket = new Socket("127.0.0.1", 12345);
    BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream())); //将字节流包装成了
    PrintStream ps = new PrintStream(socket.getOutputStream());                      //PrintStream中有写出换行的方法

    System.out.println(br.readLine());
    ps.println("我想报名黑马程序员");
    System.out.println(br.readLine());
    ps.println("大哭!!!能不能给次机会");

    socket.close();
}

```

关闭 socket 会自动关闭 inputstream 和 outputstream
多线程

字节流包装成字符流

```

ServerSocket server = new ServerSocket(9999); //创建服务器
while(true) {
    final Socket socket = server.accept(); //接受客户端的请求
    new Thread() {
        public void run() {
            try {
                BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                PrintStream ps = new PrintStream(socket.getOutputStream());
                ps.println("欢迎咨询传智播客");
                System.out.println(br.readLine());
                ps.println("报满了,请报下一期吧");
                System.out.println(br.readLine());
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }.start();
}

```

例

题

客户端向服务器写字符串(键盘录入), 服务器(多线程)将字符串反转后写回, 客户端再次读取到是反转后的字符串

客户端:

```

public static void main(String[] args) throws UnknownHostException, IOException {
    Scanner sc = new Scanner(System.in); //创建键盘录入对象
    Socket socket = new Socket("127.0.0.1", 54321); //创建客户端, 指定ip地址和端口号

    BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream())); //获取输入流
    PrintStream ps = new PrintStream(socket.getOutputStream()); //获取输出流
    ps.println(sc.nextLine()); //将字符串写到服务器去
    System.out.println(br.readLine()); //将反转后的结果读出来

    socket.close();
}

```

服务端:

```

public static void main(String[] args) throws IOException {
    ServerSocket server = new ServerSocket(54321);
    System.out.println("服务器启动, 绑定54321端口");
    while(true) {
        final Socket socket = server.accept(); //接受客户端的请求

        new Thread() { //开启一条线程
            public void run() {
                try {
                    BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                    PrintStream ps = new PrintStream(socket.getOutputStream()); //获取输出流

                    String line = br.readLine(); //将客户端写过来的数据读取出来
                    line = new StringBuilder(line).reverse().toString(); //链式编程
                    ps.println(line); //反转后写回去

                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }.start();
    }
}

```

例二:

客户端向服务端上传文件

客户端:

1. 提示输入要上传的文件路径, 验证路径是否存在以及是否是文件夹
2. 发送文件名到服务端
6. 接收结果, 如果存在给予提示, 程序直接退出
7. 如果不存在, 定义FileInputStream读取文件, 写出到网络

服务端:

3, 建立多线程的服务器

4. 读取文件名

5. 判断文件是否存在， 将结果发回客户端

8. 定义`FileOutputStream`, 从网络读取数据, 存储到本地

<https://www.bilibili.com/video/av74379966?p=20>

反射

概述

A: 类的加载概述

- * 当程序要使用某个类时, 如果该类还未被加载到内存中, 则系统会通过加载, 连接, 初始化三步来实现对这个类进行初始化。
- * 加载
 - * 就是指将class文件读入内存, 并为之创建一个Class对象。任何类被使用时系统都会建立一个Class对象。
- * 连接
 - * 验证 是否有正确的内部结构, 并和其他类协调一致
 - * 准备 负责为类的静态成员分配内存, 并设置默认初始化值
 - * 解析 将类的二进制数据中的符号引用替换为直接引用
- * 初始化 就是我们以前讲过的初始化步骤 []

B: 加载时机

- * 创建类的实例
- * 访问类的静态变量, 或者为静态变量赋值
- * 调用类的静态方法
- * 使用反射的方式来强制创建某个类或接口对应的`java.lang.Class`对象
- * 初始化某个类的子类
- * 直接使用`java.exe`命令来运行某个主类

.jar 是字节码文件

A: 类加载器的概述

- * 负责将.class文件加载到内存中, 并为之生成对应的Class对象。虽然我们不需要关心类加载机制, 但是了解这个机制我们就能更好的理解程序的运行。

B: 类加载器的分类

- * Bootstrap ClassLoader 根类加载器
- * Extension ClassLoader 扩展类加载器
- * System ClassLoader 系统类加载器

C: 类加载器的作用

- * Bootstrap ClassLoader 根类加载器
 - * 也被称为引导类加载器, 负责Java核心类的加载
 - * 比如`System`, `String`等。在JDK中JRE的lib目录下rt.jar文件中
- * Extension ClassLoader 扩展类加载器
 - * 负责JRE的扩展目录中jar包的加载。
 - * 在JDK中JRE的lib目录下ext目录
- * System ClassLoader 系统类加载器
 - * 负责在JVM启动时加载来自`java`命令的class文件, 以及`classpath`环境变量所指定的jar包和类路径

反射

A: 反射概述

- * JAVA反射机制是在运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法;
- * 对于任意一个对象, 都能够调用它的任意一个方法和属性;
- * 这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。
- * 要想解剖一个类, 必须先要获取到该类的字节码文件对象。
- * 而解剖使用的就是`Class`类中的方法, 所以先要获取到每一个字节码文件对应的`Class`类型的对象。

B: 三种方式

- * a:`Object`类的`getClass()`方法, 判断两个对象是否是同一个字节码文件
- * b: 静态属性`class`, 锁对象
- * c:`Class`类中静态方法`forName()`, 读取配置文件

C: 案例演示

- * 获取class文件对象的三种方式

只要获得一个类的字节码文件, 就能知道这个类的所有方法属性

不同阶段的不同获取方式



读取配置文件 当作静态方法的锁对象 判断是否是同一个字节码对象

```
Class clazz1 = Class.forName("com.heima.bean.Person");
Class clazz2 = Person.class;

Person p = new Person();
Class clazz3 = p.getClass();

System.out.println(clazz1 == clazz2);
System.out.println(clazz2 == clazz3);
```

用反射实现多态

多态

```
interface Fruit {
    public void squeeze();
}

class Apple implements Fruit {
    public void squeeze() {
        System.out.println("榨出一杯苹果汁儿");
    }
}

class Orange implements Fruit {
    public void squeeze() {
        System.out.println("榨出一杯橘子汁儿");
    }
}

class Juicer {
    public void run(Fruit f) {
        f.squeeze();
    }
}
```

Package 包下的类名 com.heima.reflect.person

```
Demo2_Reflect.java config.properties Demo1_Reflect.java
1 package com.heima.reflect;
2
```

在配置文件(目录下新建 file, config)中写

```
Demo2_Reflect.java config.properties Demo1_Reflect.java
1 com.heima.reflect.Apple
```

用 bufferedReader 是因为它可以读整行 .newInstance() 创建此 class 对象表示类的一个实例(object 需强转)

```
//用反射和配置
BufferedReader br = new BufferedReader(new FileReader("config.properties"));
Class clazz = Class.forName(br.readLine());
Fruit f = (Fruit) clazz.newInstance();           //父类引用指向子类对象，水果的引用指向了苹果对象
Juicer j = new Juicer();
j.run(f);
```

想输出什么直接改配置文件中的类就可以了 而不需要这样一个个创建实例

```
//没有反射，只在说多态
Juicer j = new Juicer();                         //购买榨汁机
//j.run(new Apple());                            //向榨汁机中放入苹果
//j.run(new Orange());                           //Apple a = new Orange();

//在榨汁机类中添加run方法可以接收橘子
j.run(new Orange());                            //Fruit f = new Orange();
```

用反射获取构造方法

```
Class类的newInstance()方法是使用该类无参的构造函数创建对象，如果一个类没有无参的构造函数，  
就不能这样创建了，可以调用Class类的getConstructor()  
(String.class,int.class)方法获取一个指定的构造函数然后再调用Constructor类的newInstance  
("张三",20)方法创建对象  
通过有参构造获取对象  
public static void main(String[] args) throws Exception {  
    Class clazz = Class.forName("com.heima.bean.Person");  
    //Person p = (Person) clazz.newInstance();  
    //System.out.println(p);  
    Constructor c = clazz.getConstructor(String.class,int.class); //获取有参构造  
    Person p = (Person) c.newInstance("张三",23); //通过有参构造创建对象  
    System.out.println(p);  
}
```

用反射获取成员变量

```
Class.getField(String)方法可以获取类中的指定字段(可见的)，  
如果是私有的可以用getDeclaredField("name")方法获取，通过set(obj, "李四")方法可以设置指定对象上该字段的值，  
如果是私有的需要先调用setAccessible(true)设置访问权限，用获取的指定的字段调用get(obj)可以获取指定对象中该字段的值  
public static void main(String[] args) throws Exception {  
    Class clazz = Class.forName("com.heima.bean.Person");  
    Constructor c = clazz.getConstructor(String.class,int.class); //获取有参构造  
    Person p = (Person) c.newInstance("张三",23); //通过有参构造创建对象  
  
    Field f = clazz.getField("name"); //获取姓名字段  
    f.set(p, "李四"); //修改姓名的值  
    System.out.println(p);  
}  
如是私有的成员变量  
Field f = clazz.getDeclaredField("name"); //暴力反射获取字段  
f.setAccessible(true); //去除私有权限  
f.set(p, "李四");  
  
System.out.println(p);
```

用反射获取方法

```
* Class.getMethod(String, Class...) 和 Class.getDeclaredMethod(String,  
Class...)方法可以获取类中的指定方法，调用invoke(Object,  
Object...)可以调用该方法，Class.getMethod("eat") invoke(obj)  
Class.getMethod("eat",int.class) invoke(obj,10)  
  
public static void main(String[] args) throws Exception {  
    Class clazz = Class.forName("com.heima.bean.Person");  
    Constructor c = clazz.getConstructor(String.class,int.class); //获取有参构造  
    Person p = (Person) c.newInstance("张三",23); //通过有参构造创建对象  
  
    Method m = clazz.getMethod("eat"); //获取eat方法  
    m.invoke(p);  
  
    Method m2 = clazz.getMethod("eat", int.class); //获取有参的eat方法  
    m2.invoke(p, 10);  
}
```

通过反射越过泛型检查

ArrayList<Integer>的一个对象，在这个集合中添加一个字符串数据 泛型只存在于编译期，运行期会被擦除

```

public static void main(String[] args) throws Exception {
    ArrayList<Integer> list = new ArrayList<>();
    list.add(111);
    list.add(222);

    Class clazz = Class.forName("java.util.ArrayList"); //获取字节码对象
    Method m = clazz.getMethod("add", Object.class); //获取add方法
    m.invoke(list, "abc");

    System.out.println(list); // [111, 222, abc]
}

```

创建方法使可更改对象属性值

```

public void setProperty(Object obj, String propertyName, Object value){},
此方法可将obj对象中名为propertyName的属性的值设置为value。
public void setProperty(Object obj, String propertyName, Object value) throws Exception {
    Class clazz = obj.getClass(); //获取字节码对象
    Field f = clazz.getDeclaredField(propertyName); //暴力反射获取字段
    f.setAccessible(true); //去除权限
    f.set(obj, value);
}

```

练习

已知一个类，定义如下：

```

* package cn.itcast.heima;
* public class DemoClass {
    public void run() {
        System.out.println("welcome to heima!");
    }
}

```

- * (1) 写一个Properties格式的配置文件，配置类的完整名称。
- * (2) 写一个程序，读取这个Properties配置文件，获得类的完整名称并加载这个类，用反射的方式运行run方法。

右键创建 file 命名 xxx.properties

```
1 com.heima.test.DemoClass
```

```

public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new FileReader("xxx.properties")); //创建输入流关联xxx.properties
    Class clazz = Class.forName(br.readLine()); //读取配置文件中类名，获取字节码对象

    DemoClass dc = (DemoClass) clazz.newInstance(); //通过字节码对象创建对象
    dc.run();
}

```

动态代理

动态代理概述

- * 代理：本来应该自己做的事情，请了别人来做，被请的人就是代理对象。
- * 举例：春节期间买票让人代买
- * 动态代理：在程序运行过程中产生的这个对象，而程序运行过程中产生对象其实是我们刚才反射讲解的内容，所以，动态代理其实就是通过反射来生成一个代理
- * 在Java中java.lang.reflect包下提供了一个Proxy类和一个InvocationHandler接口，通过使用这个类和接口就可以生成动态代理对象。JDK提供的代理只能针对接口做代理。我们有更强大的代理cglib，Proxy类中的方法创建动态代理类对象
- * public static Object newProxyInstance(ClassLoader loader,Class<?>[] interfaces,InvocationHandler h)
- * 最终会调用InvocationHandler的方法
- * InvocationHandler Object invoke(Object proxy,Method method,Object[] args)

接口

```

public interface User {
    public void add();

    public void delete();
}

```

实现类

```

public class UserImp implements User {
    @Override
    public void add() {
        //System.out.println("权限校验");
        System.out.println("添加功能");
        //System.out.println("日志记录");
    }

    @Override
    public void delete() {
        //System.out.println("权限校验");
        System.out.println("删除功能");
        //System.out.println("日志记录");
    }
}

```

应该有一个单独的类来做检验和记录功能

动态代理类

```

public class MyInvocationHandler implements InvocationHandler {
    private Object target;

    public MyInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        System.out.println("权限校验");
        method.invoke(target, args); //执行被代理target对象的方法
        System.out.println("日志记录");
        return null;
    }
}

Main
/*
 * public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,
 * InvocationHandler h)
 */
MyInvocationHandler m = new MyInvocationHandler(ui);
User u = (User) Proxy.newProxyInstance(ui.getClass().getClassLoader(), ui.getClass().getInterfaces(), m);
u.add();
u.delete();

```

可以代理执行某类的方法，使其功能更加强大

模板设计模式

A: 模版设计模式概述

- * 模版方法模式就是定义一个算法的骨架，而将具体的算法延迟到子类中来实现

B: 优点和缺点

- * a: 优点
 - * 使用模版方法模式，在定义算法骨架的同时，可以很灵活的实现具体的算法，满足用户灵活多变的需求
- * b: 缺点
 - * 如果算法骨架有修改的话，则需要修改抽象类

原始写法：

```

public static void main(String[] args) {
    /*long start = System.currentTimeMillis();
    for(int i = 0; i < 1000000; i++) {
        System.out.println("x");
    }
    long end = System.currentTimeMillis();
    System.out.println(end - start)*/
}

```

将查看运行时间的代码抽出来变成一个模板

```

abstract class GetTime {
    public final long getTime() {
        long start = System.currentTimeMillis();
        code();
        long end = System.currentTimeMillis();
        return end - start;
    }

    public abstract void code();
}

```

Demo 类继承 重写想要看运行时间的代码

```

class Demo extends GetTime {

    @Override
    public void code() {
        int i = 0;
        while(i < 100000) {
            System.out.println("x");
            i++;
        }
    }
}

```

Main

```

Demo d = new Demo();
System.out.println(d.getTime());

```

Java 新特性

枚举类

A: 枚举概述

- * 是指将变量的值一一列出来，变量的值只限于列举出来的值的范围内。举例：一周只有7天，一年只有12个月等。

B: 回想单例设计模式：单例类是一个类只有一个实例

- * 那么多例类就是一个类有多个实例，但不是无限个数的实例，而是有限个数的实例。这才能是枚举类。

是单例类的一种扩展 有限个例

自定义实现

无参

```

public class Week {

    public static final Week MON = new Week();
    public static final Week TUE = new Week();
    public static final Week WED = new Week();

    private Week(){} //私有构造，不让其他类创建本类对象
}

public static void main(String[] args) {
    Week mon = Week.MON;
    Week tue = Week.TUE;
    Week wed = Week.WED;

    System.out.println(mon);
}

```

有参

```

public class Week2 {

    public static final Week2 MON = new Week2("星期一");
    public static final Week2 TUE = new Week2("星期二");
    public static final Week2 WED = new Week2("星期三");

    private String name;
    private Week2(String name){
        this.name = name;
    } //私有构造，不让其他类创建本类对象
    public String getName() {
        return name;
    }
}

```

```

public static void main(String[] args) {
    //demo1();
    Week2 mon = Week2.MON;
    System.out.println(mon.getName());
}

抽象
public abstract class Week3 {

    public static final Week3 MON = new Week3("星期一") {
        public void show() {
            System.out.println("星期一");
        }
    };
    public static final Week3 TUE = new Week3("星期二") {
        public void show() {
            System.out.println("星期二");
        }
    };
    public static final Week3 WED = new Week3("星期三") {
        public void show() {
            System.out.println("星期三");
        }
    };
}

private String name;
private Week3(String name) {
    this.name = name;
} //私有构造,不让其他类创建本类对象
public String getName() {
    return name;
}

public abstract void show();

```

Main

```

public static void main(String[] args) {
    //demo1();
    //demo2();
    Week3 mon = Week3.MON;
    mon.show();
}

```

匿名内部类就相当于子类

编译看 week3 的 show 运行看重写的 show

Enum 枚举

无参

```

public enum Week {
    MON, TUE, WED;
}

标记为枚举即可 默认空参构造
public static void main(String[] args) {
    Week mon = Week.MON;
    System.out.println(mon);
}

```

enum 已经重写 tostring

有参

```

public enum Week2 {
    MON("星期一"), TUE("星期二"), WED("星期三");

    private String name;
    private Week2(String name) {
        this.name = name;
    }
}

public static void main(String[] args) {
    //demo1();
    Week2 mon = Week2.MON;
    System.out.println(mon);
}

```

抽象

```

public enum Week3 {
    MON("星期一") {
        public void show() {
            System.out.println("星期一");
        }
    }, TUE("星期二") {
        public void show() {
            System.out.println("星期二");
        }
    }, WED("星期三") {
        public void show() {
            System.out.println("星期三");
        }
    };
    private String name;
    private Week3(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    public abstract void show();
}

Main
public static void main(String[] args) {
    //demo1();
    //demo2();
    Week3 mon = Week3.MON;
    mon.show();
}

```

定义枚举类要用关键字enum

所有枚举类都是Enum的子类

枚举类的第一行上必须是枚举项，最后一个枚举项后的分号是可以省略的，但是如果枚举类有其他的东西，这个分号就不能省略。

建议不要省略

枚举类可以有构造器，但必须是private的，它默认的也是private的。

枚举类也可以有抽象方法，但是枚举项必须重写该方法

枚举在switch语句中的使用

枚举方法

```

* int ordinal()
int compareTo(E o)
String name()
String toString()
<T> T valueOf(Class<T> type, String name)
values()

```

此方法虽然在JDK文档中查找不到，但每个枚举类都具有该方法，它遍历枚举类的所有枚举值非常方便

original 枚举编号 从 0 开始

compareto 比较的就是编号 返回差值

name 获取实例名称

valueof 通过字节码对象获取枚举项

```

Week2 mon = Week2.valueOf(Week2.class, "MON");
System.out.println(mon);

```

Values 得到值数组

```

Week2[] arr = Week2.values();
for (Week2 week2 : arr) {
    System.out.println(week2);
}

```

新特性

```
* A:二进制字面量  
* B:数字字面量可以出现下划线  
* C:switch 语句可以用字符串  
* D:泛型简化,菱形泛型  
* E:异常的多个catch合并,每个异常用或 |  
* F:try-with-resources 语句,1.7版标准的异常处理代码  
*  
* 100_000  
*/
```

```
public static void main(String[] args) {  
    System.out.println(0b110);  
    结果 6
```

接口内可以用 default 修饰一般方法体(有内容的) 可以有 static 方法

```
class Demo1 {  
    public static void main(String[] args) {  
        Demo d = new Demo();  
        d.print();  
  
        Inter.method();  
    }  
}  
  
interface Inter {  
    public default void print() {  
        System.out.println("Hello World");  
    }  
  
    public static void method() {  
        System.out.println("static method");  
    }  
}  
  
class Demo implements Inter {  
}
```

匿名内部类中的方法

```
class Demo implements Inter {  
    public void run() {  
        int num = 10;  
        class Inner {  
            public void fun() {  
                System.out.println(num);  
            }  
        }  
  
        Inner i = new Inner();  
        i.fun();  
    }  
}
```

num 被默认加上 final

Main

```
Demo d = new Demo();  
d.run();
```

局部内部类在访问他所在方法中的局部变量必须用final修饰,为什么?
因为当调用这个方法时,局部变量如果没有用final修饰,他的生命周期和方法的生命周期是一样的,当方法弹栈,这个局部变量也会消失,那么如果局部内部类对象还没有马上消失想用这个局部变量,就没有了,如果用final修饰会在类加载的时候进入常量池,即使方法弹栈,常量池的常量还在,也可以继续使用

https://blog.csdn.net/qq_41701956/article/details/84322748

<https://blog.csdn.net/chenshiyang0806/article/details/79879269>

<https://github.com/ZhongFuCheng3y/3y/blob/master/src/resource.md>

<https://github.com/ZhongFuCheng3y/3y/blob/master/src/resource.md>

web backend 计算机网络

server 计算机系统

熟悉一门语言

最重要 leetcode 100-200 medium 题 链表 数组 区别 crud 对链表数组的时间复杂度 为什么各种链表数组题

ArrayList 底层工作 数组扩容 原理

Hash 表怎么实现 扩容如何实现 hash 冲突

BST 二叉搜索树

数据库 transaction

会设计表

前端-后端-数据库 过程

如何分表 normalization 什么标准

如何设计数据库

索引的实现 b+树 与二叉树区别 好处

基本索引 复合索引

LRU 简单实现 通过 hash 和双向链表?

计算机网络

TCP, UDP

HTTP, HTTPS 区别

用户前端 login 整个过程 前-后-库

Hash sort session id 存到 cookie 返回用户 jsonwebtoken

操作系统

线程 进程 协程

多线程 多进程

多线程出现的问题

死锁

什么是 risk condition 多线程

同步异步

了解 redis

一小部分网络安全