java基础

Java简介

Java语言是**面向对象的、简单的、分布式的、健壮的、安全的、可移植的、多线程的、高性能的、动态的。**

Java语法规范：

对象：对象是类的一个实例，有状态和行为。例如：一只猫是一个对象，他的状态有：颜色、品种等等，行为有：吃、叫

类：类是一个模板，它描述义类对象的行为和状态

方法：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有东作都是再方法中完成的

实例变量：每个对象都有独特的实例便来你，对象的状态由这些实例变量的值决定。

Java修饰符：

访问控制修饰符：default、public、protected、private

非访问控制修饰符：final、abstract、static、synchronized（线程\同步）

Java变量：

局部变量、类变量（静态变量）、成员变量（非静态变量）

继承：

在Java中，一个类可以由其他类派生。如果要创建一个类，而且已经存在一个类具有你所需要的属性或方法，那么你可以将新创建的类继承该类。

利用继承，可以宠用已经存在的类的方法和属性，而且不用重写这些代码。被继承的类称为超类（super class)派生类称为子类（subclass)

接口：

在Java中，接口可以理解为对象相互通信的协议，接口在继承中扮演着很重要的角色。

接口只是定义派生要用到的方法，但是方法的具体实现完全取决于派生类。

内置数据类型：

Java提供了八种基本类型，六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。

byte：八位、有符号的、以二进制补码表示的整数

最小值-128（-2^7） 最大值127（2^7-1）默认值为0

byte类型用在大型数组中节约空间，主要代替整数，因为byte变量占用的空间只有int类型的四分之一。

short：十六位、有符号的以二进制补码表示的整数

最小值-32768（-2^15）最大值32767（2^15-1） 默认值为0

short数据类型也可以像byte那样节省空间，一个short变量是int型变量所占空间的二分之一。

int：三十二位、有符号的以二进制补码表示的整数

最小值：-2^31 最大值2^31-1 默认值为0

一般整型变量默认为int类型

long：六十四位、有符号的以二进制补码表示的整数

最小值：-2^63 最大值 2^63-1 默认值为0L

在long数据类型的数据后面都需要加L,理论上不区分大小写，但为了区分l和1，一般写作大写

float：单精度、三十二位、

float在存储大型浮点数组的时候可以节省内存空间

默认值0.0f 浮点数不能用来表示精确的值

double：双精度、六十四位

浮点数的默认类型为double类型

double类型同样不能表示精确的值。

默认值是0.0d

boolean:表示一位的信息

只有两个取值：true和false

默认值是false

char：一个单一的16位Unicode字符

最小值是\u0000（即为0） 最大值是\uffff（即为65535）

char数据类型可以存储任何字符

特殊转义字符

符号	字符含义
\n	换行（0x0a)
\r	回车(0x0d)
\f	换页符(0x0c)
\b	退格(0x08)
\0	空字符(0x20)
\s	字符串
\t	制表符
\ddd	八进制字符(ddd)
\uxxxx	十六进制Unicode字符

自动类型转换（转换从低级到高级）

```
byte,short,char->int->long->float->double
```

数据类型转换需要满足的规则：

1、不能对boolean类型进行类型转换

2、不能把对象类型转换成不想管类的对象

3、在把容量大的类型转换为容量小的类型时必须使用强制类型转换

4、转换过程中可能导致数据溢出或损失精度

Java支持的变量类型有：

类变量：独立于方法之外的变量，用static修饰

实例变量：独立于方法之外的变量，不过没有static修饰

局部变量：类的方法中的变量

```
public class variableType {
    static String  mapclick = null;//类变量
    public static void main(String[] args){
        String str = "hello java";//实例变量
    }
    public void method(){
        String  i = "this is java";//局部变量
    }
}
```

实例变量

- 实例变量声明在一个类中，但在方法、构造方法和语句块之外；
- 当一个对象被实例化之后，每个实例变量的值就跟着确定；
- 实例变量在对象创建的时候创建，在对象被销毁的时候销毁；
- 实例变量的值应该至少被一个方法、构造方法或者语句块引用，使得外部能够通过这些方式获取实例变量信息；
- 实例变量可以声明在使用前或者使用后；
- 访问修饰符可以修饰实例变量；
- 实例变量对于类中的方法、构造方法或者语句块是可见的。一般情况下应该把实例变量设为私有。通过使用访问修饰符可以使实例变量对子类可见；
- 实例变量具有默认值。数值型变量的默认值是0，布尔型变量的默认值是false，引用类型变量的默认值是null。变量的值可以在声明时指定，也可以在构造方法中指定；
- 实例变量可以直接通过变量名访问。但在静态方法以及其他类中，就应该使用完全限定名：ObejectReference.VariableName

```
package javaclass;

import java.util.Scanner;

public class variableTypeTest {
//    name和age均为实例变量，对子类可见
    public String name;
    public Integer age;
    public static void main(String[] args){


```

```
        variableTypeTest vtt = new variableTypeTest();//创建新的对象，便于后期对方法的调用
        Scanner sc = new Scanner(System.in);//调用输入
        System.out.println("请输入宠物的名字：");
        String name = sc.next();
        System.out.println("请输入宠物的年龄：");
        Integer age = sc.nextInt();
        vtt.pupyName(name);
        vtt.pupyAge(age);

    }
//    在构造器中对name赋值
    public void pupyName(String Name){
        name = Name;
        System.out.println("pupy's name is:"+name);

    }
//    在构造器中对age进行赋值
    public void pupyAge(Integer Age){
        age = Age;
        System.out.println("pupy's age is :"+age);
    }
}
```

类变量（静态变量）

- 类变量也称为静态变量，在类中以 static 关键字声明，但必须在方法之外。
- 无论一个类创建了多少个对象，类只拥有类变量的一份拷贝。
- 静态变量除了被声明为常量外很少使用。常量是指声明为public/private，final和static类型的变量。常量初始化后不可改变。
- 静态变量储存在静态存储区。经常被声明为常量，很少单独使用static声明变量。
- 静态变量在第一次被访问时创建，在程序结束时销毁。
- 与实例变量具有相似的可见性。但为了对类的使用者可见，大多数静态变量声明为public类型。
- 默认值和实例变量相似。数值型变量默认值是0，布尔型默认值是false，引用类型默认值是null。变量的值可以在声明的时候指定，也可以在构造方法中指定。此外，静态变量还可以在静态语句块中初始化。
- 静态变量可以通过：*ClassName.VariableName*的方式访问。
- 类变量被声明为public static final类型时，类变量名称一般建议使用大写字母。如果静态变量不是public和final类型，其命名方式与实例变量以及局部变量的命名方式一致。

```
package javaclass;
public class variableTypeStatic {
    private static String Error = "you should input a new string";//定义静态变量
    public static void main(String[] args){
        System.out.println(Error);//调用静态变量
    }
}
```

访问类修饰符

修饰符	当前类	同一包内	子孙类(同一包)	子孙类(不同包)	其他包
`public`	Y	Y	Y	Y	Y
`protected`	Y	Y	Y	Y/N	N
`default`	Y	Y	Y	N	N
`private`	Y	N	N	N	N

```
/**
 * @author licaidie
 * @Date 20210601
 */
package javaclass;
public class Modifier {

        boolean processOrder(){  //default默认访问修饰符--不适用任何关键字
            return true;
        }
        public class Logger{
            private String format;  //private私有访问修饰符
            public String getFormat(){   //返回format的值
                return this.format;
            }
            public void setFormat(String format){
                this.format = format;   //设置format的值 （方法中含有局部变量和成员变量同名。程序需要访问这个被覆盖的成员变量，则必须使用this前缀）
            }
        }
        class AudioPlayer{
            protected <Speaker> boolean openSpeaker(Speaker sp){  //受保护的访问修饰符
                System.out.println("dfdg");
                return true;
            }
        }
        class StreamAudioPlayer extends AudioPlayer{
            protected <Speaker> boolean openSpeaker(Speaker sp){
                return false;
            }
        }
        public static void main(String[] args){}  //public公有访问修饰符

}
```

访问控制和继承

父类声明中为public的方法在子类中也必须为public

父类中声明为proected的方法在子类中要么声明为protected要么声明为public,不能声明为private

父类中声明为private的方法，不能被继承（private仅在该类中生效）

非访问类修饰符

static 修饰符，用来修饰类、方法和变量

final 修饰符，用来修饰类、方法和变量，final修饰的类不能够被继承，修饰的方法不能被继承重新定义，修饰的变量为常量，是不可修改的。

abstract 修饰符，用来创建抽象类和抽象方法。

synchronized和volatile修饰符，主要用于线程的编程。

静态修饰符：

- static 关键字用来声明独立于对象的静态方法。静态方法不能使用类的非静态变量。静态方法从参数列表得到数据，然后计算这些数据。

对类变量和方法的访问可以直接使用 **classname.variablename** 和 **classname.methodname** 的方式访问。

```
public class StaticModifier {
    private static int numInstance = 0;    //定义初始instance变量值
    protected static int getCount(){    //返回instance变量
        return numInstance;
    }  //获取getCount()方法的返回值
    private static void addInstance(){  //设置静态的
        numInstance++;
    }
    StaticModifier(){  //调用addInstance方阿飞
        StaticModifier.addInstance();
    }
    public static void main(String[] args){
        System.out.println("starting with
"+StaticModifier.getCount()+"instances");
        for(int i = 0;i < 500;++i){
            new StaticModifier();
        }
        System.out.println("Created " + StaticModifier.getCount() +
"instances");
    }
```

final修饰符：

final 表示"最后的、最终的"含义，变量一旦赋值后，不能被重新赋值。被 final 修饰的实例变量必须显式指定初始值。

final 修饰符通常和 static 修饰符一起使用来创建类常量。

```
public class FinalModifier {
    final int value = 10;//final变量
    //一下为声明常量的实例
    public static final int BOXWIDTH = 7;
    static final String TITLE = "manger";
    public void changeValue(){
        //value = 12; //将不能通过编译
        System.out.println(TITLE);
    }
    //final方法
    public class Test{
        public final void changeName(){
            System.out.println("请输入新的名称");
        }
    }
```

```
    //final类
   /* public fianl calss Test{
     //类体
    }*/
}
```

算数运算符

操作符	描述	例子
+	加法 - 相加运算符两侧的值	A + B 等于 30
-	减法 - 左操作数减去右操作数	A – B 等于 -10
*	乘法 - 相乘操作符两侧的值	A * B等于200
/	除法 - 左操作数除以右操作数	B / A等于2
%	取余 - 左操作数除以右操作数的余数	B%A等于0
++	自增: 操作数的值增加1	B++ 或 ++B 等于 21（区别详见下文）
--	自减: 操作数的值减少1	B-- 或 --B 等于 19（区别详见下文）

关系运算符

运算符	描述	例子
==	检查如果两个操作数的值是否相等，如果相等则条件为真。	（A == B）为假。
!=	检查如果两个操作数的值是否相等，如果值不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是那么条件为真。	（A> B）为假。
<	检查左操作数的值是否小于右操作数的值，如果是那么条件为真。	（A <B）为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是那么条件为真。	（A> = B）为假。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是那么条件为真。	（A <= B）为真。

位运算符

操作符	描述	例子
&	如果相对应位都是1，则结果为1，否则为0	（A&B），得到12，即0000 1100
|	如果相对应位都是0，则结果为0，否则为1	（A | B）得到61，即0011 1101
^	如果相对应位值相同，则结果为0，否则为1	（A ^ B）得到49，即0011 0001
~	按位取反运算符翻转操作数的每一位，即0变成1，1变成0。	（~A）得到-61，即1100 0011
<<	按位左移运算符。左操作数按位左移右操作数指定的位数。	A << 2得到240，即1111 0000
>>	按位右移运算符。左操作数按位右移右操作数指定的位数。	A >> 2得到15即 1111
>>>	按位右移补零操作符。左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充。	A>>>2得到15即

```
public class BitOperator {
    public static void main(String [] args){
        int a = 55;
        int b = 30;
        int c = 0;
        c = a & b ; //按位与
        System.out.println("a&b = " + c);
        c = a|b;     //按位或
        System.out.println("a|b = " + c);
        c = a^b;    //按位相同
        System.out.println("a^b = "+c);
        c = ~a;     //按位取反
        System.out.println("~a = " +c );
        c = a << 4; //按位左移
        System.out.println("a<<4 = "+c);
        c = a >> 5; //按位右移
        System.out.println("a >> 5 = "+c);
        c = a >>> 2; //按位右移补0操作符
        System.out.println("a >>> 2 = " + c);
    }
}
```

逻辑运算符

操作符	描述	例子
&&	称为逻辑与运算符。当且仅当两个操作数都为真，条件才为真。	（A && B）为假。
||	称为逻辑或操作符。如果任何两个操作数任何一个为真，条件为真。	（A || B）为真。
!	称为逻辑非运算符。用来反转操作数的逻辑状态。如果条件为true，则逻辑非运算符将得到false。	！（A && B）为真。

赋值运算符

操作符	描述	例子
=	简单的赋值运算符，将右操作数的值赋给左侧操作数	C = A + B将把A + B得到的值赋给C
+ =	加和赋值操作符，它把左操作数和右操作数相加赋值给左操作数	C + = A等价于C = C + A
- =	减和赋值操作符，它把左操作数和右操作数相减赋值给左操作数	C - = A等价于C = C - A
* =	乘和赋值操作符，它把左操作数和右操作数相乘赋值给左操作数	C * = A等价于C = C * A
/ =	除和赋值操作符，它把左操作数和右操作数相除赋值给左操作数	C / = A等价于C = C / A
（%）=	取模和赋值操作符，它把左操作数和右操作数取模后赋值给左操作数	C%= A等价于C = C%A
<< =	左移位赋值运算符	C << = 2等价于C = C << 2
>> =	右移位赋值运算符	C >> = 2等价于C = C >> 2
&=	按位与赋值运算符	C&= 2等价于C = C&2
^ =	按位异或赋值操作符	C ^ = 2等价于C = C ^ 2
| =	按位或赋值操作符	C | = 2等价于C = C | 2

条件运算符

条件运算符也被称为三元运算符。该运算符有3个操作数，并且需要判断布尔表达式的值。该运算符的主要是决定哪个值应该赋值给变量。

```
variable x = (expression) ? value if true : value if false
```

```
public class ConditionalOperator {
    public static void main(String[] args){
        int a,b;
        System.out.println("please input a number: ");
        Scanner sc = new Scanner(System.in);
        a = sc.nextInt();

        b = (a == 1) ? 20 : 30; //如果a = 1从成立，则将20赋给b,否则将30赋给b;
        System.out.println("value of b is: " + b);

    }
}
```

instanceof运算符

该运算符用于操作对象实例，检查该对象是否是一个特定类型（类类型或接口类型）。

```
String name = "jam"
boolean result = name instanceof  String; //name是字符串类型，因此返回true
```

java运算符优先级

类别	操作符	关联性
后缀	() [] . (点操作符)	左到右
一元	+ + - ! ~	从右到左
乘性	* / %	左到右
加性	+ -	左到右
移位	>> >>> <<	左到右
关系	>> = << =	左到右
相等	== !=	左到右
按位与	&	左到右
按位异或	^	左到右
按位或	|	左到右
逻辑与	&&	左到右
逻辑或	||	左到右
条件	? :	从右到左
赋值	= + = - = * = / =%= >> = << =&= ^ = | =	从右到左
逗号	,	左到右

增强for循环

Java 增强 for 循环语法格式如下:

```
for (声明语句 : 表达式) {    //代码句子 }
```

声明语句：声明新的局部变量，该变量的类型必须和数组元素的类型匹配。其作用域限定在循环语句块，其值与此时数组元素的值相等。

表达式：表达式是要访问的数组名，或者是返回值为数组的方法。

```
public class IntensifierFor {
    public static void main(String[] args){
        int [] numbers = {10,20,30,40,50};
        for (int x : numbers){
            System.out.println(x);
            System.out.println("\n");
        }
        String [] names = {"james","joe","lucy","jenny"};
        for(String na : names){
            System.out.println(na);
        }
    }
}
```

for-reach循环

JDK 1.5 引进了一种新的循环类型，被称为 For-Each 循环或者加强型循环，它能在不使用下标的情况下遍历数组。

语法格式如下：

```
for (type element: array) {    System.out.println(element); }
```

实例

该实例用来显示数组 myList 中的所有元素：

```
public class For_Each {
    //使用for-each遍历数组中的所有数据
    public static void  main(String[] args){
        double [] price = {2.2,36.4,456.0,26.5,15.6};
        for (double x:price
             ) {
            System.out.println(x);
        }
    }
}
```

数组作为函数的参数

```
public class ArrayParameter {
    //在printArray方法中使用数组作为参数
    public static void printArray(int[] array) {
        for (int i = 0; i < array.length; i++) {
            System.out.println(array[i] + " ");
        }
    }
    public static void main(String [] args){
        //调用printArray方法
        printArray(new int[]{1,5,68,1,7,5,4,2,});
    }
}
```

数组作为函数的返回值

```
public class ReturnArray {
    public static void main(String[] args){
        //调用reverse方法
        int [] a = reverse(new int[]{1,56,78,15,78,25});
        int [] b = {1,56,78,15,78,25};
        for (int x:b) {
            System.out.print(x+" "); //将原数组进行输出
        }
        b = reverse(b); //与int [] a = reverse(new int[]{1,56,78,15,78,25});效果
相同
        for(int s:a){//遍历数组
            System.out.print(s + " ");
        }
    }
    //此函数是将数组进行反转
    public static int[] reverse(int[] list){
        int[] result = new int[list.length]; //冲刺你创建一个与目标数组相同类型、长度的
空数组
        for (int i = 0, j  = result.length -1 ;i < list.length; i++,j--) {
            result[j] = list[i];//将倒序的目标数组内容存入新的数组中
        }
        return result;//返回新数组内容值
    }
}
```

可变参数

JDK 1.5 开始，Java支持传递同类型的可变参数给一个方法。

方法的可变参数的声明如下所示：

```
typeName... parameterName
```

在方法声明中，在指定参数类型后加一个省略号(...) 。

一个方法中只能指定一个可变参数，它必须是方法的最后一个参数。任何普通的参数必须在它之前声明。

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

```
        //调用可变参数的方法
        printMax(34,56,5,86,45,89,2);
        printMax(new int[]{1,3,5,8});
        printMin(56,89,57,5,45,9,59,43,4);

    }
    public static void printMax(int... numbers){//在方法声明中，在指定参数类型后加一个省略号
        if(numbers.length == 0){
            System.out.println("no argument passed");
        }
        double reslut = numbers[0];//将数组中的第一个值赋给result
        for (int i = 1; i < numbers.length; i++) {//筛选出最大数
            if(numbers[i] > reslut){
                reslut = numbers[i];
            }

        }
        System.out.println("the max number is "+ reslut);
    }
    public static void  printMin(double... numbers){//一个方法中只能有一个可变参数
        if(numbers.length == 0){
            System.out.println("no argument passed");
        }
        double result = numbers[0];
        for(int i = 1;i < numbers.length;i++){//遍历得出最小数
            if(result > numbers[i]){
                result = numbers[i];
            }
        }
        System.out.println("thi min number is: "+result);
    }
}
```

finalize()方法

Java 允许定义这样的方法，它在对象被垃圾收集器析构(回收)之前调用，这个方法叫做 finalize( )，它用来清除回收对象。

例如，你可以使用 finalize() 来确保一个对象打开的文件被关闭了。

在 finalize() 方法里，你必须指定在对象销毁时候要执行的操作。

finalize() 一般格式是：

```
protected void finalize() {   // 在这里终结代码 }
```

关键字 protected 是一个限定符，它确保 finalize() 方法不会被该类以外的代码调用。

当然，Java 的内存回收可以由 JVM 来自动完成。如果你手动使用，则可以使用上面的方法。

```
public class finalze {
    public static void main(String[] args){
        Cake c1 = new Cake(1);
        Cake c2 = new Cake(2);
        Cake c3 = new Cake(3);
        c2 = c3 = null;
```

```
        System.gc();//调用Java垃圾收集器，由于才c2和c3为null,因此被回收，回收顺序按照栈的规则进行，即最后创建的对象最先被回收
    }
}
class Cake extends Object{
    private int id;
    public Cake(int id){ //创建构造方法，为id赋初值
        this.id = id;  //设置id
        System.out.println("Cake Object " + id + " is created");
    }
    protected void finalize() throws  java.lang.Throwable{  //finalize用域清除回收对象
        super.finalize();//super调用父类构造方法
        System.out.println("Cake Object "+ id +" is disposed");
    }
}
```

super和this的区别

this 指的是当前对象的引用，super 是当前对象的父对象的引用。下面先简单介绍一下 super 和 this 关键字的用法。

super 关键字的用法：

- super.父类属性名：调用父类中的属性
- super.父类方法名：调用父类中的方法
- super()：调用父类的无参构造方法
- super(参数)：调用父类的有参构造方法

如果构造方法的第一行代码不是 this() 和 super()，则系统会默认添加 super()。

this 关键字的用法：

- this.属性名：表示当前对象的属性
- this.方法名(参数)：表示调用当前对象的方法

当局部变量和成员变量发生冲突时，使用`this.`进行区分。

关于 Java super 和 this 关键字的异同，可简单总结为以下几条。

1. 子类和父类中变量或方法名称相同时，用 super 关键字来访问。可以理解为 super 是指向自己父类对象的一个指针。在子类中调用父类的构造方法。
2. this 是自身的一个对象，代表对象本身，可以理解为 this 是指向对象本身的一个指针。在同一个类中调用其它方法。
3. this 和 super 不能同时出现在一个构造方法里面，因为 this 必然会调用其它的构造方法，其它的构造方法中肯定会有 super 语句的存在，所以在同一个构造方法里面有相同的语句，就失去了语句的意义，编译器也不会通过。
4. this( ) 和 super( ) 都指的是对象，所以，均不可以在 static 环境中使用，包括 static 变量、static 方法和 static 语句块。
5. 从本质上讲，this 是一个指向对象本身的指针, 然而 super 是一个 Java 关键字。

Scanner

next()

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

```
        //从键盘接收数据
        Scanner sc = new Scanner(System.in);
        //判断是否还有输入
        if(sc.hasNext()){
            String str1 = sc.next();
            /*
            *next()方法在遇到的空白，next()会自动去掉
            * next()必须在读取到有效字符后才能结束输入：例如仅输入空格是无效的输入，若第一个字符后含有有效内容，则在第一个空格后，下一个空格前的内容为有效内容
            * next()不能得到带有空格的字符串：例如输入hello word，结果只能得到hello
             */
            System.out.println("输入的数据为: " + str1 );
        }
        sc.close();//关闭输入
    }
}
```

nextLine()

```
public class NextLineDemo {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        if(sc.hasNextLine()){//使用nextLine作为节后方式
            String str1 = sc.nextLine();
            /*
            * nextLine()是以enter作为结束方式，有效内容为回车前的所有内容：例如输入为：hello word则有效数据为hello word
            * nextLine()可以接收空白
            * */
            System.out.println("输入的数据为：  "+str1);
        }
        sc.close();
    }
}
```

如果要输入 int 或 float 类型的数据，在 Scanner 类中也有支持，但是在输入之前最好先使用 hasNextXxx() 方法进行验证，再使用 nextXxx() 来读取：

```
Scanner sc = new Scanner(System.in);
if(sc.hasNextFloat()){
    float f1 = sc.nextFloat();
    System.out.println("浮点数为：" +f1);
}else{
    System.out.println("所输入的数据不是浮点数");
}
sc.close();
```

java对象和类

- **对象**：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- **类**：类是一个模板，它描述一类对象的行为和状态。

  例如：动物是一个类，而吃、睡等称为对象

一个类可以包含以下类型变量（一个类可以拥有多个方法）：

- **局部变量**：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。
- **成员变量**：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。
- **类变量**：类变量也声明在类中，方法体之外，但必须声明为static类型。

每个类都有构造方法。如果没有显式地为类定义构造方法，Java编译器将会为该类提供一个默认构造方法。

在创建一个对象的时候，至少要调用一个构造方法。构造方法的名称必须与类同名，一个类可以有多个构造方法。

```
public class ClassDemo {
    public ClassDemo(String name) {
        //这个构造器仅有一个参数
        System.out.println("the dog's name is: " +name);
    }

    public static void main(String[] args){
        //创建一个新的foo对象
        ClassDemo dog = new ClassDemo("car");
    }

}
```

访问实例变量和方法

```
public class Dog {
    int Age;
    public Dog(String Name){
        System.out.println("this dog's name is "+ Name);
    }

    public void setAge(int age){
        Age = age;
    }
    public int getAge() {
        System.out.println("the dos's age is: " + Age);
        return Age;
    }
    public static void main(String[] args){
        //创建对象
        Dog dog = new Dog("Dommy");
        //通过调用setAge方法来设置小狗的年龄
        dog.setAge(2);
        //通过调用getAge方法来获取小狗的年龄
        dog.getAge();
        System.out.println("dog's age is : "+ dog.getAge());
    }
}
```

```
public class ConstructorDemo {
    int age;
    String name;
```

```
    float salary;
    public int getAge() {
        return age;
    }

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

    public String getName() {
        return name;
    }

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

    public float getSalary() {
        return salary;
    }

    public void setSalary(float salary) {
        this.salary = salary;
    }
    public void printTracher(){
        System.out.println("老师的名字为：" +name);
        System.out.println("老师的名字为：" +age);
        System.out.println("老师的名字为：" +salary);
    }
    public static void main(String[] args){
        ConstructorDemo teacher1 = new ConstructorDemo();
        ConstructorDemo teacher2 = new ConstructorDemo();
        teacher1.setAge(33);
        teacher1.setName("王海");
        teacher1.setSalary(1200);
        teacher1.printTracher();
        teacher2.setName("陈菊");
        teacher2.setSalary(3000);
        teacher2.setAge(45);
        teacher2 .printTracher();
    }
}
```

java封装

在面向对象程式设计方法中，封装（英语：Encapsulation）是指一种将抽象性函式接口的实现细节部份包装、隐藏起来的方法。
封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。
要访问该类的代码和数据，必须通过严格的接口控制。
封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。
适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

封装的优点

- 良好的封装能够减少耦合。
- 类内部的结构可以自由修改。

- 可以对成员变量进行更精确的控制。
- 隐藏信息，实现细节。

重写和重载

方法的重写规则

- 参数列表必须完全与被重写方法的相同。
- 返回类型与被重写方法的返回类型可以不相同，但是必须是父类返回值的派生类（java5 及更早版本返回类型要一样，java7 及更高版本可以不同）。
- 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为 public，那么在子类中重写该方法就不能声明为 protected。
- 父类的成员方法只能被它的子类重写。
- 声明为 final 的方法不能被重写。
- 声明为 static 的方法不能被重写，但是能够被再次声明。
- 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为 private 和 final 的方法。
- 子类和父类不在同一个包中，那么子类只能够重写父类的声明为 public 和 protected 的非 final 方法。
- 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
- 构造方法不能被重写。
- 如果不能继承一个方法，则不能重写这个方法。

重写(Override)

```
public class OverrideDemo {
    public void move(){
        System.out.println("动物可以移动");
    }
    static class Dog extends OverrideDemo{//dog类继承OverrideDemo类
        public void move(){//重写move方法
            System.out.println("狗可以走和跑");
        }
    }
    public static void main(String[] args){
        OverrideDemo a = new OverrideDemo();//创建新的对象
        OverrideDemo b = new Dog();
        a.move();//指定OverrideDemo中的move方法
        b.move();//执行dog类中的新的move方法
    }
}
```

super关键字

```
public class OverrideDemo {
    public void move(){
        System.out.println("动物可以移动");
    }
    static class Dog extends OverrideDemo{//dog类继承OverrideDemo类
        public void move(){//重写move方法
            //在子类中调用父类被重写的方法使用super关键字
            super.move();
```

```
            System.out.println("狗可以走和跑");
        }
    }
    public static void main(String[] args){
        OverrideDemo b = new Dog();
        b.move();//执行dog类中的新的move方法以及原本的move方法
    }
}
```

当子类对象调用重写的方法时，调用的是子类的方法，而不是父类中被重写的方法。

要想调用父类中被重写的方法，则必须使用关键字 **super**。

```
public class OverrideTest {
    public static  void main(String[] args){
        Salary s = new Salary("员工A","背景",3,2500);
        OverrideTest e = new Salary("员工B","上海",2,5600);
        System.out.println("使用Salary中的mailCkeck方法");
        s.mailCheck();
        System.out.println("使用OverrideTest中的mailCkeck方法");
        e.mailCheck();
    }
    private String name;
    private String province;
    private int number;
    public OverrideTest(String name,String province,int number){
        //构造函数
        this.name = name;
        this.province = province;
        this.number = number;
    }
    public void mailCheck(){
        System.out.println("邮寄给：  "+this.name+" " +this.province);
    }
    public String toString(){//toString方法为返回一个字符串
        return name+" "+province+" "+number;
    }

    public String getName() {
        return name;
    }

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

    public String getProvince() {
        return province;
    }

    public void setProvince(String province) {
        this.province = province;
    }

    public int getNumber() {
        return number;
    }
```

```
    public void setNumber(int number) {
        this.number = number;
    }
}
class Salary extends OverrideTest{
    private double salary;//全年工资

    public void setSalary(double newSalary) {
        if(newSalary >= 0.0){
            salary = newSalary;
        }
    }
    public double getSalary() {
        return salary;
    }

    public Salary(String name, String province, int number, double salary) {//重
载方法
        super(name, province, number);//调用父类中的属性
        setSalary(salary);//设置salary

    }
    public void mailCheck(){
        System.out.println("Salary类的mailCheck方法");
        System.out.println("邮寄支票给：" + getName() + ",工资为：" + salary);
    }
    public double computePay(){
        System.out.println("计算工资，付给"+getName());
        return salary/52;
    }
}
```

重载（overload)

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

重载规则:

- 被重载的方法必须改变参数列表(参数个数或类型不一样);
- 被重载的方法可以改变返回类型;
- 被重载的方法可以改变访问修饰符;
- 被重载的方法可以声明新的或更广的检查异常;
- 方法能够在同一个类中或者在一个子类中被重载。
- 无法以返回值类型作为重载函数的区分标准。

```
public class OverloadDemo {
    //方法的重载，方法名称相同而参数不同
    public int test(){
        System.out.println("test1");
        return 1;
    }
```

```
    public void test(int a){
        System.out.println(a);
    }
    public void test(String name,int a){
        System.out.println(name + " "+ a);
    }
    public static void main(String[] args){
        OverloadDemo ovload = new OverloadDemo();
        //调用相同的方法，而使用不同的参数
       ovload.test();
       ovload.test(6);
       ovload.test("张三",22);
    }
}
```

区别点	重载方法	重写方法
参数列表	必须修改	一定不能修改
返回类型	可以修改	一定不能修改
异常	可以修改	可以减少或删除，一定不能抛出新的或者更广的异常
访问	可以修改	一定不能做更严格的限制（可以降低限制）

java多态

多态就是同一个接口，使用不同的实例而执行不同操作

多态的优点

- 消除类型之间的耦合关系
- 可替换性
- 可扩充性
- 接口性
- 灵活性
- 简化性

多态存在的三个必要条件

- 继承
- 重写
- 父类引用指向子类对象

多态的好处：可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

```
public class PolyTest {
    public static void main(String[] args){
        show(new Cat());//以cat对象调用show方法
        show(new Dog());//以dog对象调用show方法
        Animals a = new Cat();//向上转型，拆昂见猫的对象
        a.eat();//调用的是cat的eat方法
        Cat c = (Cat)a;//向下转型
        c.work();//调用的是cat的work方法

    }
```

```
    public static void show(Animals a){
        a.eat();//调用eat方法
        //判断类型
        if(a instanceof Cat){//判断a的类型:猫咪
            Cat c = new Cat();//创建新的猫咪对象
            c.work();//调用猫咪的work方法
        }else if(a instanceof Dog){//判断a的类型:猫咪
            Dog d = new Dog();//创建新的狗狗对象
            d.work();//调用狗狗的wirk方法
        }
    }
}
abstract class Animals {//抽象animal类
    abstract void eat();//抽象类中的抽象方法
}
class Cat extends Animals{
    public void eat(){//重写eat方法
        System.out.println("吃鱼");
    }
    public void work(){//在子类中增加work方法
        System.out.println("抓老鼠");
    }
}
class Dog extends Animals{
    public void eat(){
        System.out.println("啃骨头");
    }
    public void work(){
        System.out.println("看家");
    }
}
```

接口

接口与类相似点：

- 一个接口可以有多个方法。
- 接口文件保存在 .java 结尾的文件中，文件名使用接口名。
- 接口的字节码文件保存在 .class 结尾的文件中。
- 接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别：

- 接口不能用于实例化对象。
- 接口没有构造方法。
- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多继承。

接口特性

- 接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 **public abstract**（只能是 public abstract，其他修饰符都会报错）。

- 接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量（并且只能是 public，用 private 修饰会报编译错误）。
- 接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

抽象类和接口的区别

- 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。
- 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 **public static final** 类型的。
- 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。
- 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

> **注**：JDK 1.8 以后，接口里可以有静态方法和方法体了。

接口有以下特性：

- 接口是隐式抽象的，当声明一个接口的时候，不必使用**abstract**关键字。
- 接口中每一个方法也是隐式抽象的，声明时同样不需要**abstract**关键字。
- 接口中的方法都是公有的。

当类实现接口的时候，类要实现接口中所有的方法。否则，类必须声明为抽象的类。

类使用implements关键字实现接口。在类声明中，Implements关键字放在class声明后面。

java目录层级架构

根目录：**src.main.java**
1.工程启动类(Application.java)：置于com.cy.project包下或者com.cy.project.app包下

2.实体类(domain)：置于com.cy.project.domain

3.数据访问层(Dao)：置于com.cy.project.repository（dao）

4.数据服务层(Service)：置于com.cy.project.service
5.数据服务接口的实现(serviceImpl)：同样置于com.cy.project.service或者置于com.cy.project.service.impl

6.前端控制器(Controller)：置于com.cy.project.controller

7.工具类(utils)：置于com.cy.project.utils

8.常量接口类(constant)：置于com.cy.project.constant

9.配置信息类(config)：置于com.cy.project.config

资源文件：**src.main.resources**
1.页面以及js/css/image等置于static文件夹下的各自文件下

2.使用模版相关页面等置于templates文件夹下的各自文件下

java核心API

实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情形。为了解决这个问题，Java 语言为每一个内置数据类型提供了对应的包装类。所有的包装类（**Integer、Long、Byte、Double、Float、Short**）都是抽象类 Number 的子类。

Character 方法

下面是Character类的方法：

序号	方法与描述
1	isLetter() 是否是一个字母
2	isDigit() 是否是一个数字字符
3	isWhitespace() 是否是一个空白字符
4	isUpperCase() 是否是大写字母
5	isLowerCase() 是否是小写字母
6	toUpperCase() 指定字母的大写形式
7	toLowerCase() 指定字母的小写形式
8	toString() 返回字符的字符串形式，字符串的长度仅为1

Number & Math 类方法

下面的表中列出的是 Number & Math 类常用的一些方法：

序号	方法与描述
1	xxxValue() 将 Number 对象转换为xxx数据类型的值并返回。
2	compareTo() 将number对象与参数比较。
3	equals() 判断number对象是否与参数相等。
4	valueOf() 返回一个 Number 对象指定的内置数据类型
5	toString() 以字符串形式返回值。
6	parseInt() 将字符串解析为int类型。
7	abs() 返回参数的绝对值。
8	ceil() 返回大于等于(>=)给定参数的的最小整数，类型为双精度浮点型。
9	floor() 返回小于等于（<=）给定参数的最大整数 。
10	rint() 返回与参数最接近的整数。返回类型为double。
11	round() 它表示**四舍五入**，算法为 **Math.floor(x+0.5)**，即将原来的数字加上 0.5 后再向下取整，所以，Math.round(11.5) 的结果为12，Math.round(-11.5) 的结果为-11。
12	min() 返回两个参数中的最小值。
13	max() 返回两个参数中的最大值。
14	exp() 返回自然数底数e的参数次方。
15	log() 返回参数的自然数底数的对数值。
16	pow() 返回第一个参数的第二个参数次方。
17	sqrt() 求参数的算术平方根。
18	sin() 求指定double类型参数的正弦值。
19	cos() 求指定double类型参数的余弦值。
20	tan() 求指定double类型参数的正切值。
21	asin() 求指定double类型参数的反正弦值。
22	acos() 求指定double类型参数的反余弦值。
23	atan() 求指定double类型参数的反正切值。
24	atan2() 将笛卡尔坐标转换为极坐标，并返回极坐标的角度值。
25	toDegrees() 将参数转化为角度。
26	toRadians() 将角度转换为弧度。
27	random() 返回一个随机数。

String

```
public class StringDemo {
    public static void main(String[] args){
        char [] helloArray = {'h','e','l','l','o'};
        //使用String,将字符数组内容转换为字符串内容
        String helloString = new String(helloArray);
        System.out.println(helloArray);
    }
}
```

注意: String 类是不可改变的，所以你一旦创建了 String 对象，那它的值就无法改变了。

```
public class StringDemo {
    public static void main(String[] args){
        char [] helloArray = {'h','e','l','l','o'};
        //使用String,将字符数组内容转换为字符串内容
        String helloString = new String(helloArray);
        //获取字符串长度
        int len = helloString.length();
        //链接字符串
        String s = helloString.concat(" word");
        System.out.println(helloArray);
        System.out.println(len);
        System.out.println(s);
        //格式化字符串
        String fs ;
        float floatVar = 0;
        int intVar = 0;
        String stringVar = " ";
        fs = String.format("浮点类型: "+"整型: "+"字符串类
型",floatVar,intVar,stringVar);
    }
}
```

StringBuffer(线程安全的，可以被同步访问)

当对字符串进行修改的时候，需要使用 StringBuffer 和 StringBuilder 类。

和 String 类不同的是，StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。

```
public class StringBufferDemo {
    public static  void main(String[] args){
        //StringBuffer是可以追加新的内容
        StringBuffer sBuffer = new StringBuffer("welcome ");
        //对原始内容进行追加
        sBuffer.append("to ");
        sBuffer.append("java ");
        sBuffer.append("word!");
        System.out.println(sBuffer);
        //对原始内容进行删除操作，参数为起始位置
        sBuffer.delete(1,2);
        System.out.println(sBuffer);
        //将原始内容进行翻转
        sBuffer.reverse();
        System.out.println(sBuffer);
```

```
        //在原始内容中插入
        sBuffer.insert(3,"www.");
        System.out.println(sBuffer);
        //使用目标字符串替换原始内容
        sBuffer.replace(5,7,"sss");
        System.out.println(sBuffer);
    }
}
```

StringBuilder(非线程安全的，不能被同步访问)

读取控制台输入

ava.io 包几乎包含了所有操作输入、输出需要的类。所有这些流类代表了输入源和输出目标。

Java.io 包中的流支持很多种格式，比如：基本类型、对象、本地化字符集等等。

一个流可以理解为一个数据的序列。输入流表示从一个源读取数据，输出流表示向一个目标写数据。

Java 为 I/O 提供了强大的而灵活的支持，使其更广泛地应用到文件传输和网络编程中。

为了获得一个绑定到控制台的字符流，你可以把 System.in 包装在一个 BufferedReader 对象中来创建一个字符流。

下面是创建 BufferedReader 的基本语法：

```
BufferedReader br =  new BufferedReader( new InputStreamReader(System.in));
```

BufferedReader 对象创建后，我们便可以使用 read() 方法从控制台读取一个字符，或者用 readLine() 方法读取一个字符串。

从控制台读取多字符输入

从 BufferedReader 对象读取一个字符要使用 read() 方法，它的语法如下：

```
int read( )  throws IOException
```

每次调用 read() 方法，它从输入流读取一个字符并把该字符作为整数值返回。 当流结束的时候返回 -1。该方法抛出 IOException。

```
public class StreamDemo {
    public static void main(String[] args) throws IOException {
        char c ;
        //使用System.in创建BufferReader
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("输入字符，按q退出");
        //读取字符
        do{
            c = (char) br.read();
            System.out.print(c);
        }while(c!= 'q');
    }
}
```

从控制台读取字符串输入

```
public class StreamInputDemo1 {
    public static void main(String[] args) throws IOException {
        //使用System.in从控制台创建bufferReader
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        String st;
        System.out.println("enter lines of text");
        System.out.println("enter 'end' to quit");
        do{
            //使用readLinde读取整行数据
            st = br.readLine();
            System.out.println(st);
        }while(!st.equals("end"));
    }
}
```

控制台输出

在此前已经介绍过，控制台的输出由 print( ) 和 println() 完成。这些方法都由类 PrintStream 定义，System.out 是该类对象的一个引用。

PrintStream 继承了 OutputStream类，并且实现了方法 write()。这样，write() 也可以用来往控制台写操作。

PrintStream 定义 write() 的最简单格式如下所示：

```
void write( int byteval)
```

该方法将 byteval 的低八位字节写到流中

```
public class StreamOutputDemo {
    public static void main (String[] args){
        int a;
        a = 'A';
        //write（）方法是向控制台中写
        System.out.write(a);
        System.out.write('\n');
    }
}
```

读取文件（FileInputStream)

该流用于从文件读取数据，它的对象可以用关键字 new 来创建。

有多种构造方法可用来创建对象。

可以使用字符串类型的文件名来创建一个输入流对象来读取文件：

```
InputStream f =  new FileInputStream( "C:/java/hello" );
```

也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用 File() 方法来创建一个文件对象：

```
File f =  new File( "C:/java/hello" );
InputStream out = new FileInputStream(f);
```

创建了InputStream对象，就可以使用下面的方法来读取流或者进行其他的流操作。

```
public class FileInputStreamDemo {
    public static void main(String[] args) throws IOException {
        //创建一个输入流对象来读取文件
        InputStream f  = new FileInputStream("C:/Users/ASUS/Desktop/密钥.txt");
         /*
       * 上面内容也可写作：
       * File f = new File("C:/Users/ASUS/Desktop/密钥.txt")
       * InputStream = new FileInputStream(f);
       * */
        //创建bufferReader对象
        BufferedReader br = new BufferedReader(new InputStreamReader(f));
        String fs;
        //通过行读取方式读取文件内容
        fs = br.readLine();
        System.out.println(fs);

    }
```

```
public class FileOutputStream {
    public static void main(String[] args) throws IOException {
        //创建输入流
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        //读取输入内容
        String st = br.readLine();
        //创建输出流
        OutputStream ot = new
java.io.FileOutputStream("C:/Users/ASUS/Desktop/hello.txt");
       //创建写入流
        OutputStreamWriter wr = new OutputStreamWriter(ot,"UTF-8");
        wr.append(st);
        wr.append("\n");
        wr.close();//关闭写入流，同时会把缓冲区内容写入文件
        ot.close();//关闭输出流，释放系统资源
        br.close();//关闭输入流，释放资源
    }
}
```

目录

创建目录

```
public class CreatDirDemo {
    public static void main(String[] args){
        String dirname = "E:/Java学习/ja/va";
        File d = new File(dirname);
        //mkdir()方法为创建一个文件夹，创建成功则返回true，失败返回false
        //mkdirs()方法创建一个文件夹和它的所有父文件夹
        d.mkdirs();
    }
}
```

读取目录

```
public class ReadDirDemo {
    public static void main(String[] args){
        String dirname = "E:/Java学习";//目标路径地址
        File f1 = new File(dirname);//创建文件对象
        if(f1.isDirectory()){//判断目标路径是否为一个文件夹
            System.out.println("目录" + dirname);
            String  s[] = f1.list();//创建字符串数组：即将文件夹内的子文件以数组的形式展现
            for (int i = 0; i <s.length ; i++) {//遍历数组
                File f = new File(dirname+"/"+s[i]);//创建新的文件对象
                if(f.isDirectory()){//判断是否为文件夹
                    System.out.println(s[i] + "是一个目录");
                }else{
                    System.out.println(s[i] + "不是一个目录");
                }
            }
        }else{
            System.out.println(dirname+"不是一个目录");
        }
    }
}
```

删除目录

```
public class DelDirDemo {
    public static void main(String[] args){
        //创建新的文件对象
        File folder = new File("E:/Java学习/ja");
        //调用deleteFoleder方法
        deleteFolder(folder);

    }

    public static void deleteFolder(File folder) {//deleteFolder()方法的参数为文件夹
        File[] files = folder.listFiles();//将文件枯井转换为list形式
        if(files != null){//判断文件是否为空
            for (File f:files){
                if(f.isDirectory()){
                    deleteFolder(f);
                }else{
                    f.delete();
                }
```

```
            }
        }
        folder.delete();
    }
}
```

```
//finally 关键字用来创建在 try 代码块后面执行的代码块。
//无论是否发生异常，finally 代码块中的代码总会被执行。
public class ExceptionDemo1 {
    public static void main(String[] args){
        int []a = new int[2];
        try{
            System.out.println("Access element three: " + a[3]);
        }catch (ArrayIndexOutOfBoundsException e){
            System.out.println("Exception thrown: "+ e);
        }finally{
            a[0] = 6;
            System.out.println("First element value: "+ a[0]);
            System.out.println("the finally statement is execute");
        }
    }
}
```

枚举（Enumeration）

接口虽然它本身不属于数据结构,但它在其他数据结构的范畴里应用很广。 枚举（The Enumeration）接口定义了一种从数据结构中取回连续元素的方式。

例如，枚举定义了一个叫nextElement 的方法，该方法用来得到一个包含多元素的数据结构的下一个元素。

```
public class EnumDemo {
    public static void main(String[] args){
        Enumeration<String> days;
        Vector<String> dayNames = new Vector<String>();
        dayNames.add("Sunday");
        dayNames.add("Monday");
        dayNames.add("Tuesday");
        dayNames.add("Wednesday");
        dayNames.add("Thursday");
        dayNames.add("Friday");
        dayNames.add("Saturday");
        days = dayNames.elements();
        while (days.hasMoreElements()){
            System.out.println(days.nextElement());
        }
    }
}
```

栈（stack）

```
public class StackDemo {
    static void showpush(Stack<Integer> st,int a){
        //push()把项压入对战顶部
```

```
        st.push(new Integer(a));
        System.out.println("push("+a+")");
        System.out.println("stack" + st);
    }
    static void showpop(Stack<Integer> st){
        System.out.print("pop->");
        //pop()移除堆栈顶部的对象，并作为此函数的值返回对象
        Integer a = (Integer) st.pop();
        System.out.println(a);
        System.out.println("stack "+ st);
    }
    public static void main(String[] args){
        Stack<Integer> st = new Stack<Integer>();
        System.out.println("Stack: "+ st);
        showpush(st,43);
        showpush(st,44);
        showpush(st,20);
        showpop(st);
        showpop(st);
        showpop(st);
        try{
            showpop(st);
        }catch (EmptyStackException e){
            System.out.println("empty stack");
        }
    }
}
```

Map接口

Map接口中键和值一一映射. 可以通过键来获取值。

- 给定一个键和一个值，你可以将该值存储在一个Map对象. 之后，你可以通过键来访问对应的值。
- 当访问的值不存在的时候，方法就会抛出一个NoSuchElementException异常.
- 当对象的类型和Map里元素类型不兼容的时候，就会抛出一个 ClassCastException异常。
- 当在不允许使用Null对象的Map中使用Null对象，会抛出一个NullPointerException 异常。
- 当尝试修改一个只读的Map时，会抛出一个UnsupportedOperationException异常。

```
public class MapDemo {
    public static void main(String[] args){
        Map m1 = new HashMap();//创建map对象
        m1.put("Zara","8");//向map对象中添加信息
        m1.put("susan","22");
        m1.put("Dssiy","14");
        System.out.println();
        System.out.println("map elements");
        System.out.println("\t" + m1);//输出map对象中的内容,将以集合的形式展现
    }
}
```

Properties 类

Properties 继承于 Hashtable.表示一个持久的属性集.属性列表中每个键及其对应值都是一个字符串。

```
public class PropertiesDemo {
```

```
    public static void main(String[] args){
        Properties captials = new Properties();
        Set states;
        String str;
        captials.put("Illinois","Springfield");
        captials.put("Missouri","Jefferson City");
        captials.put("Washington","Olympia");
        captials.put("California","Sacramento");
        //展示哈希表中所有的state和capital
        states = captials.keySet();
        Iterator itr = states.iterator();
        while(itr.hasNext()){
            str = (String) itr.next();
            System.out.println("the capital of "+ str+" is
"+captials.getProperty(str));
        }
        System.out.println();

        str = captials.getProperty("Florida","Not Found");
        System.out.println("the capital of Florida is " + str +".");
    }
}
```

集合

Java 集合框架主要包括两种类型的容器，一种是集合（Collection），存储一个元素集合，另一种是图（Map），存储键/值对映射。Collection 接口又有 3 种子类型，List、Set 和 Queue，再下面是一些抽象类，最后是具体实现类，常用的有 ArrayList、LinkedList、HashSet、LinkedHashSet、HashMap、LinkedHashMap 等等。

集合框架是一个用来代表和操纵集合的统一架构。所有的集合框架都包含如下内容：

- **接口**：是代表集合的抽象数据类型。例如 Collection、List、Set、Map 等。之所以定义多个接口，是为了以不同的方式操作集合对象
- **实现（类）**：是集合接口的具体实现。从本质上讲，它们是可重复使用的数据结构，例如：ArrayList、LinkedList、HashSet、HashMap。
- **算法**：是实现集合接口的对象里的方法执行的一些有用的计算，例如：搜索和排序。这些算法被称为多态，那是因为相同的方法可以在相似的接口上有着不同的实现。

除了集合，该框架也定义了几个 Map 接口和类。Map 里存储的是键/值对。尽管 Map 不是集合，但是它们完全整合在集合中。

Set和List的区别

- Set 接口实例存储的是无序的，不重复的数据。List 接口实例存储的是有序的，可以重复的元素。
- Set检索效率低下，删除和插入效率高，插入和删除不会引起元素位置改变 **(实现类有HashSet,TreeSet)**。
- List和数组类似，可以动态增长，根据实际存储的数据的长度自动增长List的长度。查找元素效率高，插入删除效率低，因为会引起其他元素位置改变 **(实现类有ArrayList,LinkedList,Vector)** 。

一般遍历数组都是采用for循环或者增强for，这两个方法也可以用在集合框架，但是还有一种方法是采用迭代器遍历集合框架，它是一个对象，实现了Iterator 接口或ListIterator接口。

迭代器，使你能够通过循环来得到或删除集合的元素。ListIterator 继承了Iterator，以允许双向遍历列表和修改元素。

```
public class ArrayListDemo {
    public static void main(String[] args){
        List<String> list = new ArrayList<String>();
        list.add("hello");
        list.add("world");
        list.add("!!!");
        //使用for-each遍历list
        for (String str: list) {
            System.out.println(str);
        }
        //将链表变为数组相关的内容进行遍历
        String[] strArry = new String[list.size()];
        list.toArray(strArry);
        for (int i = 0; i <strArry.length ; i++) {
            System.out.println(strArry[i]);
        }
        //使用迭代器进行相关遍历
        Iterator<String> ite = list.iterator();
        while (ite.hasNext()){
            System.out.println(ite.next());
        }
    }
}
```

Map

```
public class MapDemo1 {
    public static void main(String[] args){
        Map<String,String> map = new HashMap<String,String>();
        map.put("1","value1");
        map.put("2","value2");
        map.put("3","value3");
        //第一种：普遍使用，二次取值
        System.out.println("通过Map.keySet遍历Key和value:");
        for (String key:map.keySet()) {
            System.out.println("key = "+key+"and value = "+map.keySet());
        }
        //第二种
        System.out.println("通过Map.entrySet使用iterator遍历key和value");
        Iterator<Map.Entry<String,String>> it = map.entrySet().iterator();
        while(it.hasNext()){
            Map.Entry<String,String> entry = it.next();
            System.out.println("key = "+entry.getKey()+" and value = "+
entry.getValue());
        }
        //第三种，推荐，尤其是容量大时
        System.out.println("通过Map.entrySet遍历key和value");
        for(Map.Entry<String,String> entry:map.entrySet()){
            System.out.println("key = "+entry.getKey()+" and value =
"+entry.getValue());
        }
        //第四种
        System.out.println("通过Map.values()遍历所有calue,但不能遍历key");
        for(String v: map.values()){
            System.out.println("value = "+ v);
```

```
        }
    }
}
```

List、ArrayList、Map、HashMap 区别

List是接口，List特性就是有序,会确保以一定的顺序保存元素。ArrayList是它的实现类,是一个用数组实现的List.

Map是接口,Map特性就是根据一个对象查找对象。HashMap是它的实现类,HashMap用hash表实现的Map,就是利用对象的hashcode(hashcode()是Object的方法)进行快速散列查找.

ArrayList是有序的，会确保以一定的顺序保存元素，而HashMap是无需存储，并且是通过键值对的形式。
ArrayList：是一个list集合的实现类，动态存储多个对象，集合的长度随着集合中的对象的个数而变化。
HashMap：是map的一个实现类，存储一对对象(key-value)。通过key来查找value。（键值对中：一个键最多对应1个值。map中的key不能重复，不能是重复的对象。）
扩展：list中是以数组的结构实现存储数据，高效查找，低效修改，说道数组，他和list的区别就是：数组初始化的时候必须声明数据的数量，而list是根据你加的数据的个数而变化。

java泛型

泛型提供了编译时类型安全检测机制，该机制允许程序员在编译时检测到非法的类型。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

你可以写一个泛型方法，该方法在调用时可以接收不同类型的参数。根据传递给泛型方法的参数类型，编译器适当地处理每一个方法调用。

下面是定义泛型方法的规则：

- 所有泛型方法声明都有一个类型参数声明部分（由尖括号分隔），该类型参数声明部分在方法返回类型之前（在下面例子中的`<E>`）。
- 每一个类型参数声明部分包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。
- 类型参数能被用来声明返回值类型，并且能作为泛型方法得到的实际参数类型的占位符。
- 泛型方法体的声明和其他方法一样。注意类型参数只能代表引用型类型，不能是原始类型（像int,double,char的等）。

```
//使用泛型打印不同类型的数组
public class GenericMethod {
    //泛型方法printArray
    public static <E> void printArray(E[] inputArray){
        //输出数组元素
        for (E element:inputArray) {
            System.out.printf("%s ",element);
        }
        System.out.println();
    }
    public static void main(String[] args){
        //创建不同类型数组：Integer,Double和Character
        Integer[] intArray = {1,2,3,4,5};
```

```
        Double[] doubleArrsy = {1.1,2.2,3.3,4.4,5.5};
        Character[] charArray = {'H','E','L','L','O'};
        System.out.println("整型数组元素为：");
        printArray(intArray);//传递一个整型数组
        System.out.println("双精度数组元素为：");
        printArray(doubleArrsy);//传递一个双精度数组
        System.out.println("字符型数组元素为：");
        printArray(charArray);//传递一个字符型数组
    }
}
```

有界的类型参数

要声明一个有界的类型参数，首先列出类型参数的名称，后跟extends关键字，最后紧跟它的上界。

```
public class MaximumTest {
    public static <T extends Comparable<T>> T maxmum(T x,T y,T z){
        T max = x;//假设x是最初始最大值
        if(y.compareTo(max) > 0){
            max = y;//y更大，将y的值赋给max
        }
        if(z.compareTo(max) > 0){
            max = z;//z更大，将z的值赋给max
        }
        return max;//返回最大值
    }
    public static void main(String[] args){
        System.out.printf("%d,%d和%d种最大的数为：%d\n",3,4,5,maxmum(3,4,5));
        System.out.printf("%.1f,%.1f和%.1f种最大的数为：
%.1f\n",3.2,4.1,5.5,maxmum(3.2,4.1,5.5));
        System.out.printf("%s,%s和%s种最大的数为：
%s\n","apple","orange","pear",maxmum("apple","orange","pear"));
    }
}
```

泛型类

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。

和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

```
public class Box<T> {
    private T t;
    public void add(T t){
        this.t = t;
    }
    public T getT(){
        return t;
    }
    public static void main(String[] args){
        Box<Integer> integerBOx = new Box<Integer>();//新建一个整型的泛型类对象
        Box<String> stringBox = new Box<String>();//新建一个字符串类型的泛型类对象
        integerBOx.add(new Integer(10));//向整型泛型类对象中添加数据
```

```
        stringBox.add(new String("hello world"));
        System.out.printf("整型值为：%d\n",integerBOx.getT());//打印输出泛型对象值
        System.out.printf("字符串为：%s\n",stringBox.getT());
    }
}
```

类型通配符

1、类型通配符一般是使用?代替具体的类型参数。例如 **List<?>** 在逻辑上是**List,List** 等所有List<具体类型实参>的父类。

解析： 因为getData()方法的参数是List类型的，所以name，age，number都可以作为这个方法的实参，这就是通配符的作用

```
public class GenericTest1 {
    public static void main(String[] args){
        List<String> name = new ArrayList<String>();
        List<Integer> age = new ArrayList<Integer>();
        List<Number> number = new ArrayList<Number>();
        name.add("lucy");
        age.add(22);
        number.add(2333);
        getData(name);
        getData(age);
        getData(number);

    }
    public static void getData(List<?> data){
        System.out.println("data: "+data.get(0));
    }
}
```

```
public class GenericTest2 {
    public static void main(String[] args){
        List<String> name = new ArrayList<String>();
        List<Integer> age = new ArrayList<Integer>();
        List<Number> number = new ArrayList<Number>();
        name.add("zoey");
        age.add(20);
        number.add(254);
        getUpNumber(age);
        getUpNumber(number);
    }
    public static void getData(List<?> data){
        System.out.println("data: "+ data.get(0));
    }
    public static void getUpNumber(List<? extends Number> data){
        System.out.println("data: "+ data.get(0));
    }
}
```