Java代码要写在类中，不能写到类外

语句要写到方法(函数)中

语句结束要加分号";"

Java是严格区分大小写的

注意缩进，空4个空格，也可以用tab键

一行一个语句

加号(+)，如果是字符串中，表示拼接

常见拼接： 字符串+变量

注释：是给程序员看的，编译的时候会忽略，有三种注释

- 单行注释 // 双斜线后面的一行都被注释掉
- 多行注释 /* */ 在这个范围内都会被注释掉
- 文档注释 /** */

Java标识符 任意顺序的大小写字母、数字、下划线（_）和美元符号（$）组成，但不能以数字开头，不能是Java中的关键字,可以是汉字，但不建议

类名首字母大写，如果有多个单词组成，每个单词首字母都大写 Dog GoToSchool

变量名和方法名的第一个单词首字母小写，从第二个单词开始每个单词首字母大写 bookName ,price , bookPress

标识符要见名知意

整数：分为 byte(字节类型) 、short(短整型)、int(整型)、long(长整型)

以ob开头的是二进制，0开头的数字是八进制，以0x开头的是16进制

小数： 单精度的float，双精度的double，小数默认是double类型的

- 单精度浮点数后面以F或f结尾，而双精度浮点数则以D或d结尾(一般不写) float e = 3.14f;

字符 char 单引号之内，只能是一个 ，可以是转义字符，如‘\n’,也可以是unicode字符，以"\u"开头，后面个4个数字

Java采用的是Unicode字符集，一个字符占2个字节 ‘abc123’

字符串 String S是大写的 String是一个类 双引号之内

print方法中，双引号之内的东西，原样输出，非双引号内的当做变量

布尔类型 boolean 真(true)和假(false)

- 错误写法： ture flase

Java的数据类型

- 基本类型(8个) byte(字节类型) 、short(短整型)、int(整型)、long(长整型)、float(单精度浮点数)、double (多精度浮点数) 、char(字符类型)、boolean(布尔类型)
- 引用类型(5个) 类、接口、数组、枚举、注解

Java是强类型的，java类型长度是固定的，整数默认都是int类型，小数默认都是double类型

byte类型变量可以赋值为整型常量(byte a=10)，不能赋值为整型变量(int b = 10; byte a = b)

java采用unicode编码，不管是英语、数字、中文，只要是char类型，都占2个字节，‘a’，’1‘，’中‘，char类型可以是整数互相转化

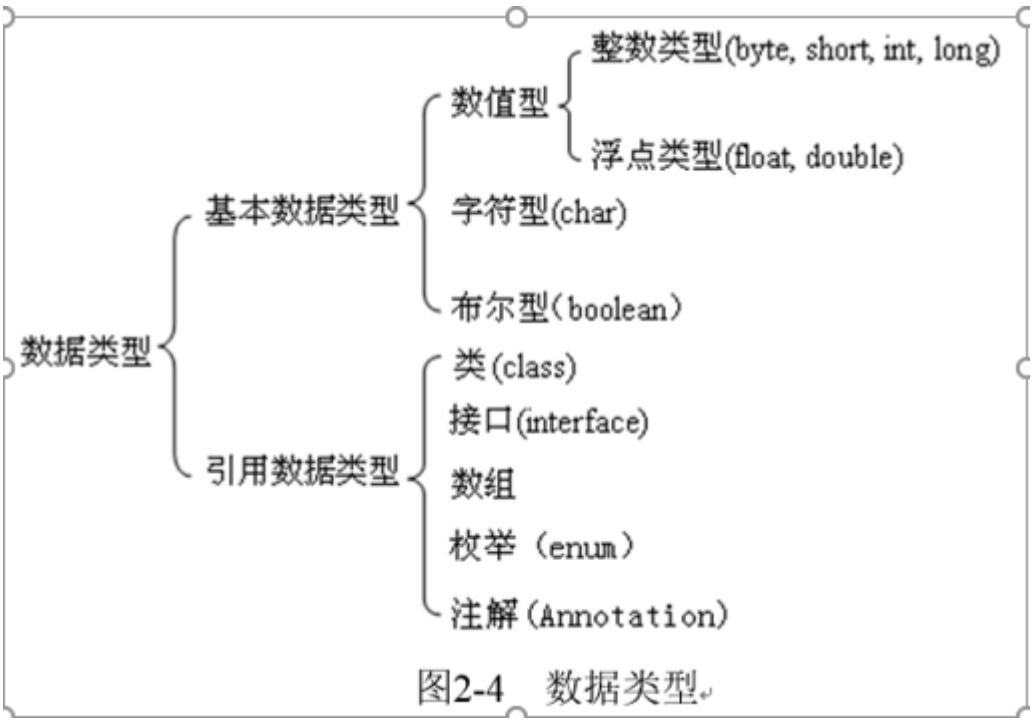


图2-4 数据类型

类型名	占用空间	取值范围
byte	8 位（1 个字节）	-2^7 ～ 2^7-1
short	16 位（2 个字节）	-2^{15} ～ $2^{15}-1$
int	32 位（4 个字节）	-2^{31} ～ $2^{31}-1$
long	64 位（8 个字节）	-2^{63} ～ $2^{63}-1$

类型名	占用空间	取值范围
float	32 位（4 个字节）	1.4E-45 ～ 3.4E+38,-3.4E+38 ～ -1.4E-45
double	64 位（8 个字节）	4.9E-324 ～ 1.7E+308, -1.7E+308 ～ -4.9E-324

数据类型转换：自动类型转换(默认转换)、强制类型转换

自动类型转换：低精度自动向高精度转换

强制类型转换：高精度向低精度转化的时候，需要强制转化 强制转换，只对括号后的类型起作用

(转化后的类型)目标值

在程序中，变量一定会被定义在某一对大括号中，该大括号所包含的代码区域便是这个变量的作用域

Java运算符：算术运算符、赋值运算符、比较运算符、逻辑运算符

运算符	运算	范例	结果
+	正号	+3	3
-	负号	b=4;-b;	-4
+	加	5+5	10
-	减	6-4	2
*	乘	3*4	12
/	除	5/5	1
%	取模（即算术中的求余数）	7%5	2
++	自增（前）	a=2;b=++a;	a=3;b=3;
++	自增（后）	a=2;b=a++;	a=3;b=2;
--	自减（前）	a=2;b=--a	a=1;b=1;
--	自减（后）	a=2;b=a--	a=1;b=2;

a++ 先使用，后自增 ++a 先自增，后使用

整数相除，结果一定是一个整数。如果有小数参与，结果会是一个小数

在进行取模（%）运算时，运算结果的正负取决于%左边的数的符号

运算符	运算	范例	结果
=	赋值	a=3;b=2;	a=3;b=2;
+=	加等于	a=3;b=2;a+=b;	a=5;b=2;
-=	减等于	a=3;b=2;a-=b;	a=1;b=2;
*=	乘等于	a=3;b=2;a*=b;	a=6;b=2;
/=	除等于	a=3;b=2;a/=b;	a=1;b=2;
%=	模等于	a=3;b=2;a%=b;	a=1;b=2;

运算符	运算	范例	结果
==	相等于	4 == 3	false
!=	不等于	4 != 3	true
<	小于	4 < 3	false
>	大于	4 > 3	true
<=	小于等于	4 <= 3	false
>=	大于等于	4 >= 3	true

逻辑运算符：逻辑与&&、逻辑或||、逻辑非！（常用）位与&、位或|、异或^（二进制）

短路特性：在&&中，如果左边结果是false，右边不再计算 (在&中，不管左边结果是否为false，右边永远参与运算)

在||中，如果左边结果是true，右边不再计算 (在|中，不管左边结果是否为true，右边永远参与运算)

```
int a = 5;
int b = 10;
if( a>10 && b++>5) {
    System.out.println(b);
}else {
    System.out.println("else:"+b);
```

```
        }


        if( a>10 & b++>5) {
            System.out.println(b);
        }else {
            System.out.println("else:"+b);
        }

        if( a>0 || b++>5) {
            System.out.println(b);
        }else {
            System.out.println("else:"+b);
        }

        if( a>0 | b++>5) {
            System.out.println(b);
        }else {
            System.out.println("else:"+b);
        }
```

运算符	运算	范例	结果
&	与	true & true	true
		true & false	false
		false & false	false
		false &true	false
|	或	true | true	true
		true | false	true
		false| false	false
		false| true	true
^	异或	true ^ true	false
		true ^ false	true
		false ^ false	false
		false ^ true	true
!	非	!true	false
		!false	true
&&	短路与	true && true	true
		true && false	false
		false && false	false
		false && true	false
||	短路或	true || true	true
		true || false	true
		false|| false	false
		false|| true	true

运算符优先级：一元运算符>二元运算符>三元运算符，赋值运算符优先级最低

优先级	运算符
1	. [] ()
2	++ -- ~ ! (数据类型)
3	* / %
4	+ -
5	<< >> >>>
6	< > <= >=
7	== !=
8	&
9	^
10	|
11	&&
12	||
13	?:
14	= *= /= %= += -= <<= >>= >>>= &= ^= |=

三种结构：顺序、选择、循环

选择结构 :非此即彼

> if语句
>
> if..else
>
> if..else if ..else
>
> switch

if语句 ，后面跟括号，括号里是条件语句（值为布尔结果的，关系运算符、逻辑运算符、boolean变量），后面不管是几句，都要用大括号括起来（如果不加大括号，只影响紧跟其后的第一条语句）

else语句不能单独出现，与紧跟其上的if匹配

三元运算符： 条件？表达式1 ：表达式2

```
if(条件){
    表达式1；
}else{
    表达式2;
}

int a=5;
int b=4;
a>b?a:b
```

```
        int score = 65;
        if(score<60) {
            System.out.println("不及格");
        }else {
            if(score<70) {
                System.out.println("及格");
            }else {
                if(score<80) {
                    System.out.println("中等");
                }else {
                    if(score<90) {
```

```
                    System.out.println("良好");
                }else {
                    System.out.println("优秀");
                }
            }
        }
    }
```

switch语句之后跟着小括号，小括号里是一个表达式语句（结果只能是byte，short，int，char，枚举、String类型（JDK1.7引入的）），紧跟着是大括号，大括号里是case语句，case之后是一个常量值（不能是变量、表达式语句），值的类型要与switch括号里的类型一致，case之后最好要跟break，如果没有加break，会在匹配的case之后继续往下执行，直到遇到break或者switch结束，可以加default语句，表示所有的case都不匹配情况下要执行的语句

```
        int score = 65;
        switch(score/10) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
            System.out.println("不及格");
            break;
        case 6:
            System.out.println("及格");
            break;
        case 7:
            System.out.println("中等");
            break;
        case 8:
            System.out.println("良好");
            break;
        case 9:
            System.out.println("不优秀");
            break;
```

```
        default:
            System.out.println("成绩错误");
            break;
        }
```

循环结构：需要反复多次执行

> while循环
>
> do..while循环
>
> for循环

while循环：主要针对循环次数不确定的情况，while之后紧跟小括号，小括号里是条件语句(结果是布尔值就行)，只要条件成立，{}内的执行语句就会执行，直到条件不成立，while循环结束

do..while：do之后紧跟一个大括号，大括号里是需要执行的语句，大括号后跟着while条件，最后要加分号

区别：while先判断后执行，可能一次都不执行，do..while是先执行，后判断，至少会执行一次

```
        int i=1;
        int sum=0;
        /*while(i<=10) {
            sum+=i++;
            i++;
        }*/
        do {
            sum+=i;
            i++;
        }while(i<=10);
        System.out.println(sum);
```

for循环：一般是针对循环次数确定的时候

```
for (① ; ② ; ③) {
    ④
}
```

第一步，执行①

第二步，执行②，如果判断结果为 true，执行第三步，如果判断结果为 false，执行第五步

第三步，执行④

第四步，执行③，然后重复执行第二步

第五步，退出循环

①表示初始化表达式、②表示循环条件、③表示操作表达式、④表示循环体

for循环括号里的三个语句都可以省略，for语句可以嵌套

```
        int sum=0;
        for(int i=1;i<=10;i++) {
            sum+=i;
        }
        System.out.println(sum);

        #省略条件1,需要把条件1放到for语句之前
        int sum=0;
        int i=1;
        for(;i<=10;i++) {
            sum+=i;
        }
        System.out.println(sum);

        //省略条件2，意味着判断语句永远为真(死循环),需要在执行语句中判断并能够跳出循环
        int sum=0;
        for(int i=1;;i++) {
            if(i<=10) {
                sum+=i;
            }else {
                break;
            }
        }
        System.out.println(sum);

//省略条件3,需要把条件3放到执行语句的最后
        int sum=0;
        for(int i=1;i<=10;) {
            sum+=i;
            i++;
```

```
        }
        System.out.println(sum);

        for(int a=1;a<=5;a++) {
            for(int b=1;b<=10;b++) {
                System.out.print(a*b+"  ");
            }
            System.out.println();
        }
```

跳转语句：break语句和continue语句

break语句：用在switch条件语句和循环语句中，它的作用是终止某个case并跳出switch结构或者跳出整个循环。

continue语句：用在循环语句中，它的作用是终止本次循环，执行下一次循环

```
        int sum=0;
        for(int i=1;i<10;i++) {
            if(i%3!=0) {
                sum+=i;
            }else {
                break;
                 //continue;
            }
        }
        System.out.println(sum);
```

•1、break语句：用在switch条件语句和循环语句中，它的作用是终止某个case并跳出switch结构。

•2、continue语句：用在循环语句中，它的作用是终止本次循环，执行下一次循环

方法：也就是C语言中的函数，只不过是放到了类中，充当类的一个功能

为了特定目的而组合起来的一组代码语句，目的是为了解决代码重复编写的问题

三个要素： 返回类型 方法名(形式参数 1,形参2....)

```
修饰符 返回值类型 方法名([参数类型 参数名1,参数类型 参数名2,..... ]){
    执行语句
    ..... 
    return 返回值;
}
```

方法不能在其它方法中定义，方法通过return返回需要的返回值，返回值的类型要和方法定义前面的类型要一致或者兼容，return语句之后不能再加代码(执行到return，代码就结束)

```
public class TestMethod {
    //方法
    int getMax(int a,int b) {
        return a>b?a:b;
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```

方法重载(overload)：多个功能类似的方法，可以命名一样

方法重载特征：多个方法的方法名一样，方法的参数个数或者类型不一样(与参数名称无关)，与方法的返回值类型无关

```
int getMax(int a,int b) {
    return a>b?a:b;
}

int getMax(int a,int b,int c) {
    return a>b?a:b;
}
int getMax(int a,short b) {
    return a>b?a:b;
}
/*double getMax(int a,int b) {
    return a>b?a:b;
}*/
/*int getMax(int b,int a) {
    return a>b?a:b;
}*/
```

数组：一组相同类型元素的集合

一维数组、多维数组

数组声明 ： 元素类型[] 数组名

不能再声明数组的时候，加上长度

int[] a;

数组定义: 分配空间，确定长度

数组名 = new 元素类型[长度];

数组是引用类型，需要通过new分配空间

a = new int[10];

声明的同时进行定义：元素类型[] 数组名 = new 元素类型[长度];

int[] b = new int[10];

数组创建完后，会根据元素类型自动赋默认值

数据类型	默认初始化值
byte、short、int、long	0
float、double	0.0
char	一个空字符，即'\u0000'
boolean	false
引用数据类型	null，表示变量不引用任何对象

数组如何初始化

静态初始化：定义的时候，就已经赋值了,多个值之间用逗号分开,值之间不能空(连续)

元素类型[] 数组名 = {值1,值2,值3...};

int[] b = {10,20,30,40,60};

System.out.println(b[0]);

动态初始化：定义完成之后，对元素单独赋值（可以跳跃赋值）

int[] b = new int[10];

b[0] = 10;

b[2] = 30;

如何对数组元素进行读取与更改值

通过数组元素的下标 数组名[下标] 下标从0开始，最大到“数组长度-1”

System.out.println(b[0]);

b[0] = 80;

ArrayIndexOutOfBoundsException：超出索引界限，出现此问题时，去计算数组元素的下标是不是超出了“数组长度-1”或者是负数

如何遍历数组：for循环、foreach循环

数组的长度： 数组名.length

```
    for(int i=0;i<10;i++) {
            System.out.print(b[i]+"  ");
        }

    for(int i=0;i<b.length;i++) {
            System.out.print(b[i]+"  ");
        }
    for(int m : b) {
            System.out.print(m+"  ");
        }
```

foreach循环：对数组中的每一个元素进行遍历

for(元素类型 名称: 数组名)

foreach只能从头开始，顺序遍历，不能修改数组中预算的值，只适合遍历

一维数组的静态初始化

```
1. 类型[] 数组名 = new 类型[]{元素,元素,……};
2. 类型[] 数组名 = {元素,元素,元素,……};
```

一个数组可以指向另一个数组，这样两个数组都是指向同一个东西，任意一个数组的改变，都影响另一个

```
int[] b = new int[] {10,20,30,40};
int[] a = {10,20,30,40};  //推荐，数组初始化的时候，数组的长度和元素的值就已经确定下来了
int[] a ;
a = {10,20,30,40};  //错误
a = new int[]{10,20,30,40};   //先声明，后静态初始化
    //数组赋值为另一个数组
        int[] a = {10,20,30,40};
        System.out.println(a[0]);
        int[] b = {50,60,70};
        a = b;
        System.out.println(a[0]);
        b[0] = 100;
        System.out.println(a[0]);
```

多维数组(只讲二维数组)

二维数组：一维数组的一维数组，就是说本质上是一个一维数组，但这个一维数组中存放的每一个元素都是一个数组

二维数组的声明

```
//类型[][] 数组名；
int[][] a;   //推荐
int b[][];
int[] c[];
```

二维数组的定义（new）

规则二维数组：数组中每一行的长度是固定的(相同的)

不规则二维数组：数组中每一行的长度是不固定的

定义的时候，如果指定列的长度，就是规则的，不指定列的长度，就是不规则的

不管规则还是不规则二维数组，new的时候，第一个[]里一定要有值(行)，第二个(列)可以没有

二维数组定义后(new),会根据元素类型，赋默认值

```
//1、先声明，后定义
// 数组名 = new 类型[行的长度][列的长度];
int[][] a;
a = new int[3][4];    //3行4列的二维数组，规则的，
a = new int[3][];     //不规则的3行二维数组，每行的长度不定
//2、声明的同时，进行定义
//类型[][] 数组名 = new 类型[行的长度][列的长度];
int[][] a = new int[3][4];
int[][] a = new int[3][];
//对不规则的二维数组，可以单独定义每一行长度
//  数组名[下标] = new 类型[长度];
int[][] a = new int[3][];
a[0] = new int[5];
a[1] = new int[2];
a[2] = new int[4];
```

二维数组的初始化

```
//1、静态初始化：  声明的同时，就确定各元素的初始值
//  类型[][] 数组名 = { {值1，值2，值3...},{值a,值b...}，{...}  };
int[][] a = {{10,20,30,40},{50,60,70,80},{90,100,110,120}};//3行4列的二维数组
int[][] b = {{10,20,30,40},{50,60},{80,90,100}};    //3行的不规则数组，第一行有4个元素，第二行有2个元素，第三行有3个元素
//另一种形式，与上面形式等价，推荐上面形式
int[][] a = new int[3][4];
a[0] = new int[]{10,20,30,40};
a[1] = new int[]{50,60,70,80};
int[][] b = new int[3][];
b[0] = new int[]{10,20,30,40};
```

```
b[1] = new int[] {50,60};
//2、动态初始化：单独对每一个元素赋值
int[][] a = new int[3][4];
a[0][0] = 10;
a[0][1] = 20;
a[0][2] = 30;
a[0][3] = 40;
a[1][0] = 50;
a[1][1] = 60;
a[1][2] = 70;
a[1][3] = 80;
int[][] b = new int[3][];
b[0][0] = 10;
b[0][1] = 20;
b[0][2] = 30;
b[0][3] = 40;
b[1][0] = 50;
b[1][1] = 60;
```

二维数组元素的读写

```
//通过行和列的下标来读写数组中的元素，行和列的下标仍然是从0开始
//数组名[行下标][列下标]
int[][] a = new int[3][4];
System.out.println(a[0][2]);//   读取第1行第3个元素
a[1][3] = 100;     //对第2行的第4个元素赋值为100；
```

二维数组的长度

```
//通过length属性获得
//行的长度：    数组名.length
//列的长度：    数组名[行下标].length
int[][] a = {{10,20,30,40},{50,60,70,80},{90,100,110,120}};//3行4列的二维数组
int[][] b = {{10,20,30,40},{50,60}};
System.out.println(a.length);     //a数组的长度，是行数，3
System.out.println(b.length);     //b数组的长度，是行数，2
System.out.println(a[0].length);  //a数组的第1行的长度，是列数，4
System.out.println(a[1].length);  //a数组的第2行的长度，是列数，4
System.out.println(b[0].length);  //b数组的第1行的长度，是列数，4
System.out.println(b[1].length);  //a数组的第2行的长度，是列数，2
```

二维数组的遍历

```
//1、for循环：
```

```
int[][] a = {{10,20,30,40},{50,60,70,80},{90,100,110,120}};//3行4列的二维数组
        //只针对于规则数组，不推荐
        for(int i=0;i<3;i++) {
            for(int j=0;j<4;j++) {
                System.out.print(a[i][j]+" ");
            }
            System.out.println();
        }
        //推荐使用下面方式，不管规则还是不规则的，都适用
        for(int i=0;i<a.length;i++) {
            for(int j=0;j<a[i].length;j++) {
                System.out.print(a[i][j]+" ");
            }
            System.out.println();
        }
//2、foreach循环：
//  for( 数组元素类型[]  变量名1 ： 数组名){
//        for(数组元素类型 变量名2 ：  变量名1 ){
              对变量名2 的操作 （变量名2就是具体的元素）
//        }
//  }
int[][] a = {{10,20,30,40},{50,60,70,80},{90,100,110,120}};//3行4列的二维数组
        for(int[] b : a) {     //行
            for(int c : b) {   //列
                System.out.print(c+" ");  //c 相当于 a[i][j]
            }
            System.out.println();
        }
```

面向对象：一切以类和对象为核心，就是将静态特征(变量)和动态特征(函数)整合在一起，而不是分割

面向对象的特征：封装、继承、多态

封装：把细节隐藏，目的是为了安全

继承：后代拥有前代的特征，目的是为了代码复用

多态：一个名称，不同动作

类：具有相同特征的东西的抽象(共性)。 学生、动物

对象：类中具体的某一个东西(个性)。 学生张三、狗

类相当于模具，对象相当于是具体的产品，一个类可以实例化多个对象，要现有类，后创建对象，对象拥有类的所有特征

类是对象的抽象，它用于描述一组对象的共同特征和行为。类中可以定义成员变量和成员方法，其中成员变量用于描述对象的特征，也被称作属性，成员方法用于描述对象的行为，可简称为方法

属性是以变量的形式体现的，方法是以函数的形式体现

```
class Student{    //类
    String name;  //属性
    int age;
    void study() {    //方法
        System.out.println("好好学习");
    }
}
```

万事万物皆对象

类中可以包含：

1、属性：静态特征，名词或者形容词，以变量形式体现

2、方法：动态特征，动词，以函数的形式体现

```
[修饰符] class 类名{
    属性；
    方法；
}
//修饰符是可选的，属性和方法不一定同时存在


class Student{
    int id;  //学号，属性
    String name;  //姓名，属性
    void printScore() {    //方法
        int score;   //局部变量
        System.out.println("打印成绩...");
    }
}


class Student{
    int id;  //学号，属性
    String name;  //姓名，属性
    void printScore() {    //方法
        int score;   //局部变量
        int id=0;    //方法中定义的变量叫局部变量，可以与类的属性同名，局部变量的作用范围只在本方法中起作用，属性可以在整个类中使用
        System.out.println("打印成绩..."+id);  //使用的局部变量id，就近原则，先使用自己范围内有的变量，如果没有，则去属性中查找
    }
    void add() {
        System.out.println(id);
    }
}
```

对象的创建：

```
类名 对象名称 = new 类名();
```

```
//声明一个对象：      类名 对象名；
//定义一个对象：   需要使用new（new的时候才会在内存中分配空间）          对象名 = new 类名();
//声明的同时，进行定义   :    类名 对姓名 = new 类名();
class Student{
    int id;  //学号，属性
    String name;  //姓名，属性
    void printScore() {    //方法
        System.out.println("打印成绩...");
    }
}

//声明：
Student tom;   //声明了一个Student类型的对象，叫tom
int a;
int[] b;
//定义：
tom = new Student();   //定义
b = new int[5];
//声明并定义：
Student jack = new Student();
Scanner sc = new Scanner(System.in);
```

对象实例化的时候(new)，里面的属性会被自动赋默认值（与数组类似）

成员变量类型	初始值
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	空字符，'\u0000'
boolean	false
引用数据类型	null

属性的访问： 对象名.属性名

注意： 后续很少直接访问属性

方法的访问： 对象名.方法名(实参值);

```
class Student{
    int id;
    double a;
    char b;
```

```
    boolean c;
    String name;  //Sting是一个类，这是引用类型的属性
    void printScore() {    //方法
        System.out.println("打印成绩..."+id);
    }
    void add(int a,int b) {

    }
}

public class Test {
    public static void main(String[] args) {
        Student tom = new Student();  //实例化对象、创建对象
        //属性的访问：对象名.属性名；
        System.out.println(tom.id);
        System.out.println(tom.a);
        System.out.println(tom.b);
        System.out.println(tom.c);
        System.out.println(tom.name);
        tom.id = 1001;
        System.out.println(tom.id);
        //方法的方法： 对象名.方法名(实参);
        tom.printScore();
        tom.add(3, 4);
        Student jack = new Student();
        System.out.println(jack.id);
    }

}
```

new关键字作用：分配内存空间，返回首地址

构造方法：创建对象的同时，对属性进行初始化(赋值)

构造方法是特殊的方法：1、构造方法名要和类名完全一样。2、在方法名的前面没有返回值类型的声明(即使void也不行)

构造方法的调用：构造方法不能通过对象名调用，只能是系统自己调用，在创建对象new的时候，系统自动调用。

区分属性和变量：通过this关键字，this.属性名；

构造方法语法：

类名(属性类型 属性名...){

this.属性名 = 属性名;

}

构造方法中，并不一定要对所有属性赋值

```
class Student{
    int id;
    String name;
    /*Student(){    //构造方法
        System.out.println("hello");
    }*/
   /* Student(int a,String b){    //构造方法
        id = a;
        name = b;
    }*/
    /*      类名(属性类型 属性名...){
               this.属性名 = 属性名;
            }
    */
    Student(int id,String name){    //构造方法，
        this.id = id;
        this.name = name;
    }
    void print() {
        System.out.println(id+"的姓名是："+name);
    }
}

public class Test {
    public static void main(String[] args) {
        //Student a = new Student();  //实例化对象、创建对象  ,new的时候会自动调用构造方法
        //a.Student();  //错误，不能自己调用
        /*Student a = new Student();
        a.id = 1001;
        a.name = "张三";*/
        Student a = new Student(1001,"张三");  //实例化的同时，调用构造方法，分别对属性初始化,与上面的3句话等价
        System.out.println(a.id);
        System.out.println(a.name);
    }
```

```
}
```

构造方法重载：一个类之中可以有多个构造方法

方法重载：1、方法名相同。2、参数的类型或者个数不同

```
class Student{
    int id;
    String name;
    Student(int id){    //构造方法，
        this.id = id;
    }
    Student(int id,String name){    //构造方法，
        this.id = id;
        this.name = name;
    }
    void print() {
        System.out.println(id+"的姓名是："+name);
    }
}

public class Test {
    public static void main(String[] args) {
        Student a = new Student(1001);  //此时会自动调用第一个构造方法
        System.out.println(a.id);
        System.out.println(a.name);
        Student b = new Student(1002,"张三");  //此时会自动调用第二个构造方法
        System.out.println(b.id);
        System.out.println(b.name);

    }

}
```

在Java中的每个类都至少有一个构造方法，如果在一个类中没有定义构造方法，系统会自动为这个类创建一个默认的构造方法，这个默认的构造方法没有参数，在其方法体中没有任何代码，即什么也不做

默认构造方法：

类名(){

}

```
class Student{
    int id;
    String name;
    /*Student(){     //默认构造方法，小括号里没有参数，大括号里没有内容，如果去掉注释，就不叫默认构造方法了

    }*/
    Student(){     //无参构造方法,人为提供了构造方法，系统就不在提供默认构造方法
        System.out.println("hello");
    }
}


public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student stu = new Student();//new的时候，一定会根据括号中参数的类型和个数调用最匹配的构造方法，如果类中没有构造方法，会调用系统提供的默认构造方法
        System.out.println(stu.id);
    }

}
```

题目：如果有一个类Hello，写出它的默认构造方法：

Hello(){

}

this关键字：只能在本类中使用

1、用来访问属性：this.属性名，在属性和局部变量同名的情况下，可以通过this关键字区分属性

```
class Student{
    int id;
    String name;
    //this关键字区分属性最常用场合
    Student(int id,String name){
        //this.属性名 = 形参;
        this.id = id;
        this.name = name;
    }
    //this关键字区分属性另一种场合
    void show() {
        int id=10;
        System.out.println(id);   //就近原则，采用的自己的id
        System.out.println(this.id);   //this关键字，采用的是类中的属性id
    }
}


public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student stu = new Student(1001,"张三");
        stu.show();
    }

}
```

2、可用用来调用方法：this.方法名(实参)

```
class Student{
    int id;
    String name;
    //this关键字区分属性另一种场合
    void show() {
        System.out.println(id);   //输出属性b，如果自己没有变量b，则去类中去查找有没有属性b
        //如果想在一个方法中，调用另一方法，只需要通过方法名加实参就可以
        print();
        //还可以通过this调用方法（很少使用）
        this.print();
    }
    void print() {
        System.out.println("hello");
    }
```

```
}


public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student stu = new Student();
        stu.show();
    }

}
```

3、指代当前对象

```
class Student{
    int id;
    String name;
    Student(int id,String name){
        this.id = id;
        this.name = name;
    }
    void show() {
        System.out.println(this.id);   //  this当前对象
    }
    Student returnStudent() {
        return this;    //表示返回当前对象
    }
}


public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student stu = new Student(1001,"张三");  //当前对象stu
        stu.show();     //当前对象是stu
        System.out.println(stu.returnStudent());    //返回stu对象,Student@7852e922,对象的类型@
对象的地址
        System.out.println(stu);    //返回stu对象，与上面代码等价
        Student student = new Student(1002,"李四");  //当前对象student
        student.show();     //当前对象是student
    }

}
```

4、调用本类的其它构造方法

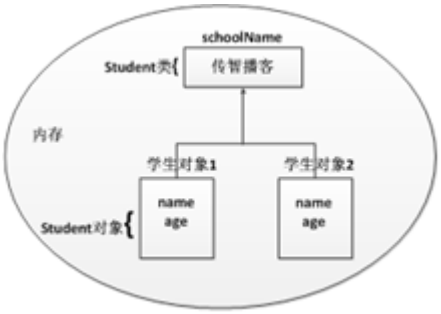
```
class Student{
    int id;
```

```
    String name;
    int age;
    char gender;
    Student(int id,String name){
        this.id = id;
        this.name = name;
        System.out.println("1");
    }
    Student(int id,String name,int age){
        /*this.id = id;
        this.name = name;*/
        this(id,name);  //通过this关键字来调用其它构造法方法，相当于调用了上面的构造方法
Student(id,name)
        this.age = age;
        System.out.println("2");
    }
    Student(int id,String name,int age,char gender){
        this(id, name,age);
        this.age = age;
        this.gender = gender;
        System.out.println("3");
    }

}


public class Test {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //Student stu = new Student(1001,"张三");
        Student stu1 = new Student(1002,"李四",20,'男');
    }

}
```

垃圾回收：Java虚拟机会自动回收垃圾对象所占用的内存空间(c++中析构函数)

除了等待Java虚拟机进行自动垃圾回收外，还可以通过调用System.gc()方法来通知Java虚拟机立即进行垃圾回收。当一个对象在内存中被释放时，它的finalize()方法会被自动调用

static关键字：静态，归整个类所有，所有对象共享这个静态的属性或静态方法



静态属性：用static修饰的属性，又称为类属性，通过"类名.静态属性名"调用

非静态属性：没有用static修饰的属性，又称为对象属性，通过"对象名.属性名"调用

```
class Student{
    static String schoolName="青岛科技大学";  //static修饰，静态属性，类属性，归整个类所有，不管创建了多少个对象，静态属性只会被创建一次，所有对象共享它
    int id;         //不用static，非静态属性，又称为对象属性，归具体的对象所有，每创建一个对象，非静态属性会被复制到当前对象里
    String name;
    Student(int id,String name){
        this.id = id;
        this.name = name;
    }
}


public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student stu = new Student(1001,"张三");
        //类名.静态属性名
        System.out.println(Student.schoolName);
        //对象名.静态属性名       //不建议
        System.out.println(stu.schoolName);
        System.out.println(stu.id);
        Student stu1 = new Student(1002,"李四");
        System.out.println(Student.schoolName);
        System.out.println(stu1.id);
        stu1.schoolName = "高密校区";
        System.out.println(Student.schoolName);

    }
```

```
}
```

静态方法：用static修饰的方法，又称为类方法，通过"类名.静态方法名"调用

非静态方法：没有用static修饰的方法，又称为对象方法，通过"对象名.方法名"调用

```
class Student{
    static String schoolName="青岛科技大学";  //static修饰，静态属性，类属性，归整个类所有，不管创建了多少个对象，静态属性只会被创建一次，所有对象共享它
    int id;         //不用static，非静态属性，又称为对象属性，归具体的对象所有，每创建一个对象，非静态属性会被复制到当前对象里
    String name;
    Student(int id,String name){
        this.id = id;
        this.name = name;
    }
    void show() {    //没有用static修饰，非静态方法
        System.out.println(schoolName+id+name);
    }
    static void print() {    //用static修饰的，静态方法
        System.out.println(schoolName);
    }
}


public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student stu = new Student(1001,"张三");
        stu.show();
        //类名.静态方法名(实参)
        Student.print();
        //对象名.静态方法名(实参)    //不建议
        stu.print();

    }

}
```

静态方法只能使用静态属性和其它静态方法，非静态方法可以使用静态属性、静态方法、非静态属性、非静态方法。

静态方法只能使用静态的，非静态方法没限制

```
class Student{
    static String schoolName="青岛科技大学";  //static修饰，静态属性，类属性，归整个类所有，不管创建了多少个对象，静态属性只会被创建一次，所有对象共享它
    int id;         //不用static，非静态属性，又称为对象属性，归具体的对象所有，每创建一个对象，非静态属性会被复制到当前对象里
    String name;
    Student(int id,String name){
        this.id = id;
        this.name = name;
    }
    void show() {    //没有用static修饰，非静态方法，可以使用任意属性和方法
        System.out.println(schoolName+id+name);  //使用静态属性和非静态属性
        print();     //调用静态方法
        add();       //调用非静态方法
    }
    static void print() {    //用static修饰的，静态方法，只能使用静态属性和静态方法
        System.out.println(schoolName);   //使用静态属性
        //System.out.println(id);   //使用非静态属性会报错，非静态的在创建类的时候就会被创建出来，此时对象还不存在
        sub();   //调用静态方法
        //add();  //调用非静态方法会报错

    }
    void add() {
        System.out.println("hello");
    }
    static void sub() {
        System.out.println("world");
    }
}


public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student stu = new Student(1001,"张三");
        stu.show();
        Student.print();

    }

}
```

getter方法与setter方法

```
//Java bean,对属性设置成私有的，再设置属性的getter和setter方法（public）
class Person{
    String name;          //不写访问权限，表示默认
    public char gender;  //public 公共权限，可以被任意访问
    private int age;      //private  私有权限，只能在本类范围内使用，其它地方不能直接访问,目的：对外不可见，达到封装的目的
    /*对私有权限的属性，对外不能直接访问，需要设置公开的方法来间接使用，一般是采用更改器(setter方法)和访问器(getter方法)
        更改器(setter方法)：用来对属性赋值
        方法名：以set开头，后面跟要赋值属性的名字，要求属性名首字母大写
        形参：参数类型就是要赋值的属性的类型，参数的名字一般写成要赋值的属性的名字
        返回值类型：一定是void
                void setXxx(属性的类型  属性名){
                     this.属性名 = 属性名;
                }
     */
    void setName(String name) {
        this.name = name;
    }
    void setGender(char gender) {
        this.gender = gender;
    }
    void setAge(int age) {
        this.age = age;
    }
    /*
    访问器(getter方法)：用来对获取属性的值
    方法名：以get开头，后面跟要读取的属性的名字，要求属性名首字母大写
    形参：不需要参数
    返回值类型：就是要读取的属性的类型
            属性的类型   getXxx(){
                 return 属性名;
            }
 */
    String getName() {
        return name;
    }
    char getGender() {
        return gender;
    }
    int getAge() {
        return age;
    }
    public void show() {
        System.out.println(name+gender+age);
    }
}
```

```java
public class TestPerson {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Person p = new Person();
        p.name = "张三";
        p.gender = '男';
        //p.age = 20;     //age是私有权限，不能再本类之外访问
        p.show();
        p.setName("李四");
        p.setGender('女');
        p.setAge(18);     //通过setter方法进行赋值
        p.show();
        //System.out.println(p.age);   //age是私有权限，不能再本类之外访问
        System.out.println(p.getAge());   //通过getter方法获得私有属性age的值
        System.out.println(p.getName());
        System.out.println(p.getGender());

    }

}
```

面向对象：封装、继承、多态

类：共性特征的抽象，包含属性和方法

对象：类的实例化

创建对象： 类名 对象名 = new 类名(参数);

属性和方法的访问： 对象名.属性名、 对象名.方法名(参数)

类中习惯通过访问权限(private)将属性名隐藏(封装)，通过属性的公开的getter和setter方法进行读写

getter方法：以get开头，后面跟着属性名，属性名首字母大写

setter方法：以set开头，后面跟着属性名，属性名首字母大写

构造方法：对属性的初始化，特征：与类名完全一样，没有返回值类型，构造方法可以重载(根据需要，一个类中可以包含多个构造方法)

每个类都要有构造方法，如果没有，系统会提供一个默认的构造方法，构造方法只能由系统在创建对象(new)的时候自动调用

new关键字：分配内存空间、返回对象地址

this关键字：1、访问属性和方法 this.属性名，2、表示当前对象，3、用来在构造方法中调用其它的构造方法 this(实参)

```
class Student{
    int id;
    String name;
    int age;
    char gender;
    Student(int id,String name){
        this.id = id;
        this.name = name;
    }
    Student(int id,String name,int age,char gender){
       this(id,name);  //通过this关键字，来调用对应的构造方法,限定：只能是在首行，并且只能是在构造方法中使用
        this.age = age;
        this.gender = gender;
        //this(id,name);    //出错，通过this来调用构造方法，只能是在首行
    }
    void print(int id,String name) {
        //this(id,name);   //出错，通过this来调用构造方法，只能是在构造方法中使用
    }

}
```

static关键字：静态 归整个类所有，可以修饰属性和方法，通过"类名.静态属性","类名.静态方法"访问。

静态方法只能访问静态属性和静态方法

内部类：在一个类只能，可以定义另一个类，此时定义的类，被称为内部类

内部类：根据内部类定义的位置，可以分为：成员内部类、静态内部类、方法内部类

成员内部类：在类中定义的类，当做类的成员看待，与属性和方法同级别。

成员内部类使用场合：如果一个类，只在另一个类中使用，其它类都不使用它，此时没必要将这个类单独做成一个外部类

成员内部类的好处：可以使用外部类中所有的成员，反过来不成立(外部类不能直接访问内部类的成员)

其它类中使用内部类对象：通过外部类.内部类访问

```
外部类名.内部类名 变量名 = new 外部类名().new 内部类名();
```

```
class Car{     //外部类
    String name;
    String color;
    void print() {   //外部类的成员方法
        System.out.println(name+color);
        //System.out.println(brand);    //外部内不能直接使用内部类的成员
        //间接访问的办法：创建一个内部类对象，通过内部类对象去访问
        Engine engine = new Engine();
        engine.brand = "bba";

    }
    class Engine{   //成员内部类
        String brand;
        double power;
        void show() {  //内部类的方法
            System.out.println(name+color+brand+power); //内部类可以访问外部类中所有的属性和方法
        }
    }

}
public class Test {

    public static void main(String[] args) {
        // 创建内部类对象，用的比较少
        //Engine en = new Engine();    //出错，不能值创建内部类对象
        //办法1、先创建外部类对象，然后在创建内部对象
```

```
        Car car = new Car();
        Car.Engine en = car.new Engine();
        en.brand = "宝马";
        //方法2：直接创建
        Car.Engine en1 = new Car().new Engine();
        en1.brand = "奔驰";
    }

}
```

继承：子类继承父类的东西，目的是为了代码复用

类的继承是指在一个现有类的基础上去构建一个新的类，构建出来的新类被称作子类，现有类被称作父类，子类会自动拥有父类除了构造方法和private修饰的成员之外的所有属性和方法(构造方法和私有成员不能被继承)

继承关键字：extends

class 子类名 extends 父类名{

}

```
class Animal{
    String name;
    int age;
    String color;
}
//  class 子类名  extends 父类名
class Dog extends Animal{  //继承
    String speak;
}

class Cat extends Animal{
    String speak;
    void eat() {

    }
}
```

继承特征：1、java中只支持单继承(一个子类只能有一个父类)，不支持多继承。2、多个类可以继承同一个父类。3、可以多层次继承

```
class Animal{
    String name;
    int age;
    String color;
}
//  class 子类名  extends 父类名
class Dog extends Animal{  //继承
    //会把父类中的可继承的成员继承下来，所以Dog也有name、age、color属性
    String speak;
}
//单继承，一个父类可以有多个子类
class Cat extends Animal{
    String speak;
    void eat() {

    }
}
//多层次继承，哈士奇拥有Dog的特征，是Dog的直接子类，同时又拥有Animal的特征(Dog继承自Animal)，是Animal的间接子类
class HaShiQi extends Dog{
    void show() {
        System.out.println(speak+name+age+color);     //使用属性的时候，先找自己类中是否有，没有的话，去父类中找，父类没有，按照继承层次，去更高一级父类中找
    }
}
```

继承：可以继承父类的成员、修改父类的成员、增加自己的成员。(继承、修改、增加)

```
class Animal{
    String name="动物";
    int age;
    String color;
}
//  class 子类名  extends 父类名
class Dog extends Animal{  //继承
    //继承属性：会把父类中的可继承的成员继承下来，所以Dog也有name、age、color属性
    //修改属性：
    String name="狗";    //如果子类中有与父类同名的属性，会产生同名隐藏问题，使用的时候，优先使用自己的属性
    //新增属性：
```

```
    String speak;
    void show() {
        System.out.println(name); //同名隐藏，使用的自己的
    }
}



public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Dog dog = new Dog();
        dog.show();
    }

}
```

方法重写：overWriter，子类中有与父类同名的方法，此时就是对父类方法的重写

使用场合：子类要有父类的方法名，但方法的实现不同的时候

方法重写的限定：1、子类中重写的方法需要和父类被重写的方法具有相同的方法名、参数列表以及返回值类型。2、重写的子类方法不能比父类的访问权限更严格(最严格的是private，后面依次是protected、默认、public，public是最宽松的)

```
class Animal{
    String name="动物";
    int age;
    void eat() {
        System.out.println("动物要吃东西！");
    }
}
//  class 子类名  extends 父类名
class Dog extends Animal{  //继承
    //继承属性：会把父类中的可继承的成员继承下来，所以Dog也有name、age、color属性
    //修改属性：
    String name="狗";    //如果子类中有与父类同名的属性，会产生同名隐藏问题，使用的时候，优先使用自己的属性
```

```
    //新增属性：
    String speak;
    void show() {
        System.out.println(name); //同名隐藏，使用的自己的
    }
    void eat() {    //方法重写，与fulei方法名称、参数列表、返回值类型一直
        System.out.println("狗吃骨头！");
    }
}



public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Dog dog = new Dog();
        dog.eat();  //方法重写的是时候，会调用自己的方法
    }

}
```

代码块

```
class Box{
    private int length;
    private int width;
    private int height;
    private String color;
    /*对属性进行初始化的办法：构造方法、初始化代码块、静态初始化代码块
     * {}之内的叫代码块块，初始化代码块，就是直接在类中写一对大括号
     * 执行：在创建对象(new)的时候，会被自动调用
     * 如果同时存在构造方法和初始化代码块，会先执行初始化代码块，在执行对应的构造方法，每创建一个对象，初始化代码块和对应的构造方法会被又执行一次
     * 静态代码块：就是在代码块{}前面加上static
     * 静态代码块中，只能使用静态成员
     * 如果同时存在构造方法、初始化代码块、静态初始化代码块，先执行静态代码块，再执行初始化代码块，最后执行对应的构造方法
     * 如果多次创建对象，静态代码块只有在第一次创建对象的时候被调用，并且值调用一次，后面不管创建多少个对象，都不再调用
     */
    //静态代码块，使用比较多
    static{
        System.out.println("执行了静态代码块....");
    }
    //初始化代码块
    {
        this.length = 10;
        this.width = 20;
        this.height = 30;
        System.out.println("长宽高分别是："+length+" "+width+" "+height);
    }
    //带参数构造方法
    Box(String color){
        this.color = color;
        System.out.println("颜色是："+this.color);
    }
}

public class TestBox {
    public static void main(String[] args) {
        Box box = new Box("红色");//会自动调用初始化代码块,会先执行静态代码块，再执行初始化代码块，最后执行对应的构造方法
        Box box1 = new Box("绿色");//先执行初始化代码块，再执行对应的构造方法
        Box box3 = new Box("黑色");//先执行初始化代码块，再执行对应的构造
    }
}
```

super关键字：指代的是父类

super关键字的用法：1、用来调用父类的属性和方法“super.父类属性名”、“super.父类方法名(实参)”，(如果子类有和父类同名的属性和方法，想调用父类的属性和方法)，注意:不能出现类似"super.super.成员"

```
class Animal{
    String speak="动物";
    void show() {
        System.out.println("叫声是：");
    }
}
class Dog extends Animal{
    String speak="狗";
    void show() {
        System.out.println(speak);  //如果有与父类同名的属性或方法(同名隐藏)，先使用自己的，如果没有，再去查找父类的
        System.out.println(super.speak);  //通过super关键字，调用父类的属性
        super.show();    //通过super关键字调用父类的方法
    }
}
class Hashiqi extends Dog{
    String speak = "哈士奇";
    void print() {
        System.out.println(speak);  //自己属性speak
        System.out.println(super.speak);  //父类Dog的属性speak
        //System.out.println(super.super.speak);  //出错，没有super.super
    }
}

public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.show();
    }

}
```

2、通过super关键字来调用父类的构造方法 "super(实参...)"

实例化子类的执行顺序：创建子类的时候，按照继承的层次，从高到低，依次执行父类的构造方法，最后执行子类的构造方法

```
class Animal{
    String speak;
    Animal(){
        System.out.println("Animal的无参构造方法....");
    }
    Animal(String speak){
        this.speak = speak;
        System.out.println("Animal的有参构造方法....");
    }

}
class Dog extends Animal{
    String name;
    Dog(){
        //如果没有通过super关键字来调用父类构造方法，默认会在第一句调用父类无参的构造方法，相当于第一句隐式加了"super();",如果父类没有无参构造方法，此时会报错
        System.out.println("Dog的无参构造方法...");
    }
    Dog(String name,String speak){
        //隐含着第一句，系统会调用父类的构造方法super();如果子类构造方法的第一句就是通过super来调用父类构造方法，则此时系统不再自动添加super(),会按照你提供的匹配的父类构造方法执行
        //super();   //如果第一句自己添加了super(),此时系统不会再自动调用父类的构造方法了
        //this.speak = speak;
        super(speak);   //调用父类的带参数的构造方法
        this.name = name;
        System.out.println("Dog的有参构造方法...");
    }

}

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Dog dog = new Dog("哈士奇","wowo");   //执行子类的时候

    }

}
```

限定：1、super调用父类构造方法的时候，一定要在子类构造方法的第一句(除非隐式调用父类无参构造方法，可以不写，由系统自动增加)；

```
class Animal{
    String speak;
    Animal(){
        System.out.println("Animal的无参构造方法....");
    }
    Animal(String speak){
        this.speak = speak;
        System.out.println("Animal的有参构造方法....");
    }

}
class Dog extends Animal{
    String name;
    Dog(){
        //如果想调用父类无参的无参构造方法，可以不用自己写super();
        System.out.println("Dog的无参构造方法...");
    }
    Dog(String name,String speak){
        super(speak);    //调用父类的带参数的构造方法,一定要放到子类构造方法的第一句，否则会出错
        this.name = name;
        System.out.println("Dog的有参构造方法...");
    }

}
```

2、通过this关键字来调用本类构造方法与通过super关键字来调用父类构造方法，不能共存

```
lass Animal{
    String speak;
    Animal(){
        System.out.println("Animal的无参构造方法....");
    }
    Animal(String speak){
        this.speak = speak;
        System.out.println("Animal的有参构造方法....");
    }

}
class Dog extends Animal{
    String name;
    Dog(){
        System.out.println("Dog的无参构造方法...");
```

```
    }
    Dog(String name,String speak){
        this();    //  this(参数...)与super(参数..)不能共存,即使第一句是通过this调用子类的其它构造方法，系统仍然会先自动调用父类无参构造方法(相当于自动添加super())
        //super(speak);
        this.name = name;
        System.out.println("Dog的有参构造方法...");
    }

}
```

3、通过super关键字调用父类构造方法、只能是在子类的构造方法中，普通方法不行

```
class Animal{
    String speak;
    Animal(){
        System.out.println("Animal的无参构造方法....");
    }
    Animal(String speak){
        this.speak = speak;
        System.out.println("Animal的有参构造方法....");
    }

}
class Dog extends Animal{
    String name;
    Dog(){
        System.out.println("Dog的无参构造方法...");
    }
    Dog(String name,String speak){
        super(speak);
        this.name = name;
        System.out.println("Dog的有参构造方法...");
    }
    void show() {
        //super();    //出错，super调用构造方法，只能是在构造方法中使用
    }

}
```

final关键字：不能修改，可以修饰属性、方法、类

final修饰的变量（成员变量和局部变量）是常量，只能赋值一次

final修饰的方法不能被子类重写

final修饰的类不能被继承

```
final class Animal{    //final修饰的类，不能被继承
    String speak;
    final void show() {   //final修饰的方法，不能被子类重写
        System.out.println("动物会叫...");
    }

}
//出错，父类是final的，不能别继承
/*class Dog extends Animal{
    final String name = "dog";  //final修饰属性，成为常量，常量定义的时候要赋值，一经定义，值不能再被修改
    //出错 ,父类方法是final的，不能被重写
    // void show() {
        //name = "cat";  //出错，常量值不能修改
    //}


}*/
```

抽象类：将一组东西的共性抽象出来，但又不能就是实例化的类，比如动物、水果

抽象方法：没有方法体的方法。原因：需要描述行为特征，但是有没法实现，比如所有的形状都有其面积的方法，但不知道怎么实现

抽象方法语法：在方法前面加上abstract关键字，方法没有方法体 "abstract 方法返回值类型 方法名(形参);"

包含抽放方法的类，一定是抽象类，抽象类中，不一定包含抽象方法

抽象类：使用abstract关键字修饰的类为抽象类，就是在类定义前面，加上abstract关键字

```
abstract class Animal{     //抽象类，类定义前面加上absctract，包含抽象方法的类一定是抽象类
    String name;
    abstract void speak();    //抽象方法，用abstract修饰，没有方法体
}
abstract class Dog{    //抽象类中，可以包含抽象方法和普通方法，也可以只有普通方法
    int age;
    void print() {
        System.out.println("因为有方法体{}，这是普通方法，不是抽象方法");
    }
}
```

如果有些方法需要在所有子类中都存在，但父类又不知道如何实现方法，此时可以做成抽象方法

子类可以继承抽象类，继承的时候，需要对抽象类中的所有抽象方法都重写，如果有任何一个抽象方法没有被重写，则子类应被定义成抽象类

```
abstract class Shape{
    abstract double getArea();    //想让所有的形状都根据自己的情况，计算面积
}

abstract class Rectangle extends Shape{    //子类没有重写抽象类的抽象方法，则只能定义成抽象类

}

class Circle extends Shape{
    double r;
    double getArea() {    //重写了抽象类的抽象方法，所以子类就可以是普通类
        return 3.14*r*r;
    }
}
```

抽象类不能直接实例化(new)，但可以用实现抽象方法的具体子类去实例化

抽象类 对象名 = new 子类(参数);

接口：interface ，特殊的抽象类，是所有方法都是抽象方法的抽象类

接口特征：接口中只能有常量，不能有变量；接口中只能有抽象方法，不能有普通方法；

接口语法：用关键字interface声明

interface 接口名{

常量

抽象方法

}

接口中的常量，默认是公共静态常量(public static final),在写的时候，一般都省略，系统自己添加

j接口中的方法，默认是公共抽象的(public abstract),在写的时候，一般都省略，系统自己添加

```
interface IShape{
    //final int a = 10;      //可以这样写
    int a = 10;   //接口中的常量，通常写的时候，省略final，默认会自动添加public static final
    //abstract double getArea();  //可以这样写
    double getArea();  //接口中的方法，通常写的时候，省略abstrac，默认会自动添加public abstract
}
```

接口的实现：关键词implements,子类可以实现接口，可以同时实现多个接口，多个接口间用逗号分开。

子类实现接口的时候，要把接口中所有的方法都实现，实现的方法前面要加public(原因：接口中的方法默认是public，子类重写父类方法的时候，要求访问权限不能比父类更严格)

语法： class 类名 implements 接口1,接口2..{

重写接口中的方法；

}

```
interface IShape{
    //final int a = 10;
    int a = 10;   //接口中的常量，通常写的时候，省略final，默认会自动添加public static final
    //abstract double getArea();
    abstract double getArea();  //接口中的方法，通常写的时候，省略abstrac，默认会自动添加public
abstract

}
interface IColor{
    void getColor() ;
}

class Circle implements IShape{
    public double getArea() {  //子类实现父接口所有的方法，并且要加上public
        return 0;
    }
}
class Rectangle implements IShape,IColor{
```

```
    public double getArea() {  //子类实现父接口所有的方法，并且要加上public
        return 0;
    }
    public void getColor() {

    }
}
```

接口不能直接实例化(new)，但可以用实现了接口的子类来实例化

接口 对象名 = new 子类(参数);

```
interface IShape{
    //final int a = 10;
    int a = 10;   //接口中的常量，通常写的时候，省略final，默认会自动添加public static final
    //abstract double getArea();
    abstract double getArea();  //接口中的方法，通常写的时候，省略abstrac，默认会自动添加public
abstract

}
interface IColor{
    void getColor() ;
}

class Circle implements IShape{
    public double getArea() {  //子类实现父接口所有的方法，并且要加上public
        return 0;
    }
}
class Rectangle implements IShape,IColor{
    public double getArea() {  //子类实现父接口所有的方法，并且要加上public
        return 0;
    }
    public void getColor() {

    }
}

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //IShape s = new IShape();  //出错，不能直接实例化
        IShape s = new Circle();
        IShape s1 = new Rectangle();
    }

}
```

接口可以继承其它的接口(extends)，并且可以多继承

interface 接口名 extends 接口1，接口2

```
interface IShape{
    double getArea();

}
interface IColor{
    void getColor() ;
}

interface ICircle extends IShape,IColor{   //接口ICircle同时继承了IShape和IColor接口，所以它会继承这两个接口中所有的属性和方法
    double getRadius() ;
}

class Circle implements ICircle{   //子类实现ICircle接口的时候，除了实现ICircle接口中的方法外，还需要实现ICircle继承的父接口中的方法
    double radius;
    public double getRadius() {
        return radius;
    }
    public double getArea() {
        return 3.14*radius*radius;
    }
    public void getColor() {
    }
}
```

接口可以多层次继承(子接口继承自父接口，父接口也可以继承自祖先接口)

```
interface IA{
    void a();
}
interface IB extends IA{
    void b();
}
interface IC extends IB{
    void c();
}
class Character implements IC{
```

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

如果一个类既要继承某个父类，又要实现多个接口，要求要先写继承，后写实现接口

class 子类 extends 父类 implements 接口1,接口2...

```
class Shape{
    String name;
}
interface IShape{
    double getArea();

}
interface IColor{
    void getColor() ;
}


//  class 子类  extends 父类 implements 接口1,接口2...
class Circle extends Shape implements IShape,IColor{
    double radius;
    public double getRadius() {
        return radius;
    }
    public double getArea() {
        return 3.14*radius*radius;
    }
    public void getColor() {
    }
}
```

接口与抽象类的区别：

1、接口中的属性只能是常量，抽象类中可以有常量、变量；2、接口中能够的方法只能是抽象方法，抽象类中可以有抽象方法，普通方法；3、接口可以多继承其它接口，抽象类只能单继承父类

抽象类可以实现接口(可以实现接口中部分方法，也可以全部实现)，接口不能继承抽象类

接口和抽象类的使用场合：如果子类与上级有逻辑上的关联关系，建议用抽象类，否自用接口。(实际使用中，接口使用比较多，面向接口编程)

多态：同一方法，不同动作。（原理：子类可以重写父类的方法）

在同一个方法中，这种由于参数类型不同而导致执行效果各异的现象就是多态。继承是多态得以实现的基础

向上转型：把子类当做父类来看待(原因：子类继承父类，会拥有父类的特征)。(狗是动物)

父类 对象名 = new 子类(参数);

```
class Animal{
    String name;
    void speak() {
        System.out.println("动物会叫....");
    }
}

class Dog extends Animal{
    void speak() {
        System.out.println("小狗汪汪叫....");
    }
}

public class Test {

    public static void main(String[] args) {
        Animal a = new Animal();  //直接创建父类对象
        a.speak();    //调用的是Animal里的方法
        Dog b = new Dog();   //直接创建子类对象
        b.speak();   //调用的是Dog里的方法
        Animal c = new Dog();   //向上转型,用子类来实例化父类对象
```

```
    }

}
```

向下转型：把父类当子类来看待，需要强制转化，可能会出错(编译的时候不提示错误，运行的时候可能会报错)。(动物是狗)

子类 对象名 = (子类)new 父类(参数);

```
class Animal{
    String name;
    void speak() {
        System.out.println("动物会叫....");
    }
}

class Dog extends Animal{
    void speak() {
        System.out.println("小狗汪汪叫....");
    }
}
class Cat extends Animal{
    void speak() {
        System.out.println("小猫喵喵叫....");
    }
}

public class Test {

    public static void main(String[] args) {
        Animal a = new Animal();  //直接创建父类对象
        a.speak();    //调用的是Animal里的方法
        Dog b = new Dog();   //直接创建子类对象
        b.speak();   //调用的是Dog里的方法
        Animal c = new Dog();   //向上转型,用子类来实例化父类对象,
        Dog d = (Dog)new Animal();  //向下转型,将实例化的父类强转成子类,,此处会出错，编译不出错，运行会出错：ClassCastException类型转换异常
        Dog e = (Dog)a;  //向下转型,将实例化的父类强转成子类,与上面语句等价，也会出错
        Dog f = (Dog)c;  //向下转型,将实例化的父类强转成子类，不会出错（ 当父类已经用指定的子类向上转型，再将父类向下转型成指定的子类）
         Animal g = new Cat();   //向上转型
        Dog h = (Dog)g;    //语法不出错，运行会出错。
    }

}
```

多态：同一方法，不同结果

向上转型：将子类当做父类来看待

多态的条件：1、要有继承关系，2、子类要重写父类的同名方法，3、向上转型，4、利用向上转型里的父类对象名来调用子类重写的方法

多态执行结果：1、如果是调用的重写的方法，则调用的是向上转型时，具体创建的对象里的方法(也就是向上转型的时候，“=”后面的对象，new)，2、如果调用的是属性，则使用的是声明的对象里的属性(也就是向上转型的时候，“=”前面的对象)(很少用)

```
class Animal{
    String name;
    void speak() {
        System.out.println("动物都会叫....");
    }
}

class Dog extends Animal{
    void speak() {
        System.out.println("小狗汪汪叫....");
    }
}

class Cat extends Animal{
    void speak() {
        System.out.println("小猫喵喵叫....");
    }
}

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Animal a = new Animal();
        a.speak();   //调用Animal自己的speak方法
        Dog b = new Dog();
```

```
        b.speak();    //调用Dog自己的speak方法
        Animal c = new Dog();  //向上转型
        c.speak();    //发生多态，会根据向上转型时，调用具体指向的对象(new)里的方法,此例是调用Dog里的speak方法
        c = new Cat();   //向上转型
        c.speak();   //多态，调用具体指向的对象(new)里的方法,此例是调用Cat里的speak方法
    }

}
```

```
class Animal{
    String name = "动物";
    void speak() {
        System.out.println("动物都会叫....");
    }
}

class Dog extends Animal{
    String name = "小狗";
    void speak() {
        System.out.println("小狗汪汪叫....");
    }
}

class Cat extends Animal{
    String name = "小猫";
    void speak() {
        System.out.println("小猫喵喵叫....");
    }
}

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        /*Animal a = new Animal();
        a.speak();   //调用Animal自己的speak方法
        Dog b = new Dog();
        b.speak();    //调用Dog自己的speak方法
*/      Animal c = new Dog();  //向上转型
        c.speak();    //发生多态，会根据向上转型时，调用具体指向的对象(new)里的方法,此例是调用Dog里的speak方法
        System.out.println(c.name);   //调用属性，会调用向上转型的时候，声明的对象里的属性，此例是声明的Animal类型的c，所以调用的是Animal里的name属性
        c = new Cat();   //向上转型
        c.speak();   //多态，调用具体指向的对象(new)里的方法,此例是调用Cat里的speak方法
        System.out.println(c.name);   //调用属性，会调用向上转型的时候，声明的对象里的属性，此例是声明的Animal类型的c，所以调用的是Animal里的name属性
    }

}
```

注意事项：向上转型的时候，父类对象名不能调用子类独有的成员

```
class Animal{
    String name = "动物";
    void speak() {
        System.out.println("动物都会叫....");
    }
}

class Dog extends Animal{
    String color = "red";
    void speak() {
        System.out.println("小狗汪汪叫....");
    }
    void eat() {
        System.out.println("小狗喜欢吃肉....");
    }
}
public class Test {

    public static void main(String[] args) {
        Animal c = new Dog();  //向上转型
        //c.eat();   //出错，向上转型的时候调用了父类中没有，子类中独有的方法
        //System.out.println(c.color);  //出错，向上转型的时候调用了父类中没有，子类中独有的属性
    }

}
```

向上转型常用的地方：出现在方法参数中，方法中调用的是父类对象，实际传递的是子类对象，这个时候会发生多态，然后按照具体子类来调用

```
class Animal{
    void speak() {
        System.out.println("动物都会叫....");
    }
}

class Dog extends Animal{
    void speak() {
        System.out.println("小狗汪汪叫....");
    }
}

class Cat extends Animal{
```

```
    void speak() {
        System.out.println("小猫喵喵叫....");
    }
}

public class Test {
    static void guard(Animal a) {   //方法中，使用的是父类对象，具体调用的时候，会将实参的值传递给形参，也就是Annimal a = 具体的是实参值;
        a.speak();
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        guard(dog);           //调用的时候，会将具体的子类对象参数传递给形参，隐含着发生了向上转型,产生了多态
        Cat cat = new Cat();
        guard(cat);
    }

}
```

Object类：是所有对象的祖先类，如果定义一个类的时候，没有写继承的父类，则默认是继承自Object

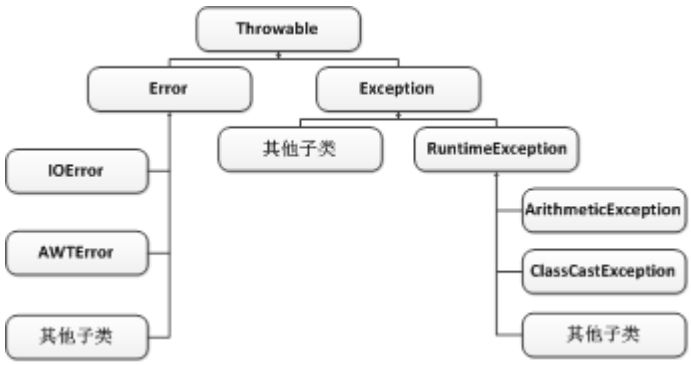
Object常用方法：任何类中都隐含继承了Obejcet的四个方法，常用的重写toString方法

方法名称	方法说明
equals()	指示其他某个对象是否与此对象“相等”
getClass()	返回此 Object 的运行时类
hashCode()	返回该对象的哈希码值
toString()	返回该对象的字符串表示

```
class Animal{     //没有写继承父类，默认继承自Object
    public String toString() {   //重写了Object类中的toString方法
        return "我是一个动物";
    }

}

public class Test {

    public static void main(String[] args) {
        Animal a = new Animal();
        System.out.println(a.toString());
```

```
        System.out.println(a);     //输出对象的时候，会调用对象里的toString方法
    }

}
```

异常：Exception，所有的异常都继承自Throwable类，Throwable有两个直接子类Error和Exception，其中Error代表程序中产生的错误，Exception代表程序中产生的异常



常见异常：

ArithmeticException，算数运算异常，出现这种错误，检查算数运算，比如是偶除以0，或者非数字。

ArrayIndexOutOfBoundsException，数组下标超出界限，出现这种错误，检查数组的下标。

NullPointerException，空指针异常，出现这种错误，检查对象是否为null

ClassCastException，类型转换异常，出现这种错误，检查要转换的对象类型是否兼容

```
class Animal {
    void speak() {
        System.out.println("动物会叫....");
    }
}
class Dog extends Animal{

}
```

```
class Cat extends Animal{

}
public class Test {

    public static void main(String[] args) {
        int a = 1/0;    //ArithmeticException
        int[] b = new int[5];
        b[5] = 10;       //ArrayIndexOutOfBoundsException
        Animal c = null;
        c.speak();     //NullPointerException
        Animal d = new Dog();
        Cat e = (Cat)d;   //ClassCastException
    }

}
```

Throwable类中常用方法：

方法声明	功能描述
String getMessage()	返回此 throwable 的详细消息字符串
void printStackTrace()	将此 throwable 及其追踪输出至标准错误流
void printStackTrace(PrintStream s)	将此 throwable 及其追踪输出到指定的输出流

异常处理：异常捕获通常使用try...catch语句，在try代码块中编写可能发生异常的Java语句，catch代码块中编写针对异常进行处理的代码。

当try代码块中的程序发生了异常，系统会将这个异常的信息封装成一个异常对象(xxxException,比如上线的常见的4中异常)，并将这个对象传递给catch代码块

```
try{
    //程序代码块
}catch(ExceptionType(Exception类及其子类)  e){
    //对 ExceptionType 的处理
}
```

try语句不能单独存在，后面一定要跟着catch，一个try后面可以跟着多个catch，出现异常的时候，会按照catch的先后顺序匹配，当匹配到符合的异常的时候，就执行catch里的语句，之后的catch就不再执行了(类似switch...case)

注意事项：catch是按照从上到下匹配的，所以需要按照异常对象的继承关系，从小到大罗列(先捕获子类异常，后捕获父类异常)，如果没有特别需要，习惯上catch中只捕获Exception异常对象

```
class Animal {
    void speak() {
        System.out.println("动物会叫....");
    }
}
class Dog extends Animal{

}
class Cat extends Animal{

}
public class Test {

    public static void main(String[] args) {
        try {
            //int a = 1/0;    //会抛出ArithmeticException异常对象
            int a = 5/2;
            int[] b = new int[5];
            //b[5] = 10;       //ArrayIndexOutOfBoundsException
            b[4] = 10;
            Animal c = null;
            c.speak();     //NullPointerException
            Animal d = new Dog();
            Cat e = (Cat)d;   //ClassCastException
        }catch(ArithmeticException e) {
            System.out.println("出现了算数运算异常，请检查运算前后是否是数字，或者除以0情况");
        }catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("出现了数组索引超界限异常");
        }catch(Exception e) {
            System.out.println("出现了Exception异常，Exception是所有异常的父类，可以捕获所有的异常，所以只能放在最后,习惯上只用这一个catch就够了");
        }
    }
```

```
}
```

finally关键字：在try..catch语句之后可以跟着finally语句，表示无论是否出错，都要执行的语句。

```
public class Test {

    public static void main(String[] args) {
        try {
            int a = 5/2;
            System.out.println(a);
            int b = 1/0;    //在出异常的地方，不继续往下执行try代码块
            System.out.println(b);
            System.out.println("hello");
        }catch(Exception e) {
            System.out.println("出现了Exception异常，Exception是所有异常的父类，可以捕获所有的异常，所以只能放在最后");
        }finally {
            System.out.println("finally语句块不是必须的，不管try中是否出现异常，finally代码块中的东西都要执行");
        }
    }

}
```