

# 书签：P516

---

## eclipse

---

### 快捷键

---

- ctrl+shift+c 注释
- alt+/ 自动补全
- ctrl+shift+f 格式化（需要配合英文模式，win+space切换）

## Java 学习笔记（重要节选）

---

### 关于JDK、JRE、JVM

---

#### JDK

（Java Development Kit，Java 开发工具包），是针对 Java 开发人员的产品，是整个 Java 的核心，包括了 Java 运行环境 JRE、Java 工具和 Java 基础类库。（JDK=JRE+多种 Java 开发工具）

#### JRE

（Java Runtime Environment，Java 运行环境）是运行 JAVA 程序所必须的环境的集合，包含 JVM 标准实现及 Java 核心类库。（JRE=JVM+各种类库）

#### JVM

（Java Virtual Machine，Java 虚拟机）是整个 Java 实现跨平台的最核心的部分，能够运行以 Java 语言写作的软件程序



JDK、JRE、JVM的关系如图，这三者的关系是一层层的嵌套关系。JDK>JRE>JVM

保留字

- 数据类型：boolean、int、long、short、byte、float、double、char、class、interface。
- 流程控制：if、else、do、while、for、switch、case、default、break、continue、return、try、catch、finally
- 修饰符：public、protected、private、final、void、static、strict、abstract、transient、synchronized、volatile、native。
- 动作：package、import、throw、throws、extends、implements、this、super、instanceof、new。
- 保留字：true、false、null、goto、const。

### javadoc

---

### java的编译

---

Java 编译的结果不是生成机器码，而是生成字节码，字节码不能直接运行，必须通过 JVM 翻译成机器码才能运行。不同平台下编译生成的字节码是一样的，但是由 JVM 翻译成的机器码却不一样。

# java的main

(源自: <https://www.csdn.net/tags/MtzaggwsOTkxNDQtYmxvZw00O0O0O0O0.html>)

1.如果源文件中有多类，那么只能有一个类是public类；如果有一个类是public类，那么源文件的名字必须与这个类的名字完全相同，扩展名是.java。

2.如果源文件中没有public类，那么源文件的名字可以任意取。这样编译是没问题的，运行时选择主类(包含main方法)运行就可以。

3.main()方法不是必须要放在public类中才能运行程序。

但是，一般情况是这样的：

1.一般我们都把main()方法放在了public类中。

- 其实这不是必须的，main方法放在哪个类中都不会影响程序的执行。
- 大家都这么写的原因是因为某些软件(例如eclipse)运行时默认到public类中去找main函数，是可以设置的。
- 这么写只是方便之举。

2.如果源文件中没有public类，那么源文件的名字和含有main()方法的类的名字相同。

- 不是必须的，如果源文件中没有public类，那么源文件的名字可以任意取。这样编译是没问题的，运行时选择主类(包含main方法)运行就可以。
- 大家都这么写的原因是因为某些软件(例如EditPlus)运行时默认到和源文件名相同的类名中去找main函数。
- 这么写只是方便之举。

原因：

1.public类名=文件名：对于一个public类，它是可以被项目中任何一个类所引用的，只需在使用它前import一下它所对应的class文件即可，将类名与文件名一一对应就可以方便虚拟机在相应的路径(包名)中找到相应的类的信息。

2.类名不用public修饰：原文件中可以没有public类，该类可以在同一个包内被访问。加public的目的在于public类在包内包外均可访问，只需在使用它前import一下它所对应的class文件即可。

(C语言中文系列 Java P404)

- **访问控制权限是公有的 (public) 。**
- **main() 方法是静态的。**如果要在 main() 方法中调用本类中的其他方法，则该方法也必须是静态的，否则需要先创建本类的实例对象，然后再通过对象调用成员方法。
- **main() 方法没有返回值，只能使用 void。**
- main() 方法具有一个字符串数组参数，用来接收执行 Java 程序的命令行参数。命令行参数作为字符串，按照顺序依次对应字符串数组 中的元素。
- 字符串中数组的名字（代码中的 args）可以任意设置，但是根据习惯，这个字符串数组的名字一般和 Java 规范范例中 main() 参数名 保持一致，命名为 args，而方法中的其他内容都是固定不变的。
- **在 main() 方法中只能直接调用静态方法**，如果想调用非静态方法，需要将当前类实例化，然后通过类的对象来调用。
- main() 方法可以以字符串的形式接收命令行参数，然后在方法体内进行处理。
- main() 方法定义必须是“public static void main(String[] 字符串数组参数名)”。

## main()方法格式固定原因

(C语言中文系列 Java P407)

## java 包 (package)

(C语言中文系列 Java P417)

- 提供了类的多层命名空间，用于解决类的命名冲突、类文件管理等问题。
- package 语句应该放在源文件的第一行，在每个源文件中只能有一个包定义语句
- Java 默认为所有源文件导入 java.lang 包下的所有类

## java继承

(资料: [https://blog.csdn.net/weixin\\_27207591/article/details/114280188](https://blog.csdn.net/weixin_27207591/article/details/114280188))

- 在Java中，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所有可继承的属性和方法。在程序中，如果想声明一个类继承另一个类，需要使用extends关键字。
- 简单而言，就是函数里面调用函数（对C的概念熟悉点）。
- Java为了保证数据安全，只允许单继承。
- 多重继承指的是一个类可以同时从多于一个的父类那里继承行为和特征。**Java提供了两种方法能够实现多重继承：接口和内部类。**

## 常量

(C语言中文系列 Java P58)

- 语法：final dataType variableName = value
- 为了与变量区别，常量取名一般都用大写字符。
- 常量有三种类型：静态常量、成员常量和局部常量。


## 变量

(C语言中文系列 Java P61-P, 较难，新的东西比较多，java对定义比较严格)

- 语法：DataType identifier;  
或者 DataType identifier=value;
- 根据作用域的不同，一般将变量分为不同的类型：成员变量和局部变量。
- 局部变量分为方法参数变量（形参）、方法局部变量（方法内定义）、代码块局部变量（代码块内定义）
- 类或者结构中的字段的变量，不初始化，默认为0
- 方法中的变量，不初始化会出错

## 数据类型

- 数据类型分为两种：基本数据类型（Primitive Type）和引用数据类型（Reference Type）。
- 基本数据类型又可分为 4 大类，即整数类型（包括 byte、short、int 和 long）、浮点类型（包括 float 和 double）、布尔类型和字符类型（char）

- image-20220827221532692
- **引用数据类型**建立在基本数据类型的基础上，包括数组、类和接口。引用数据类型是由用户自定义，用来限制其他数据的类型。另外，Java 语言中不支持 C++ 中的指针类型、结构类型、联合类型和枚举类型。

## 数据类型转换

- 数据类型的转换可以分为隐式转换（自动类型转换）和显式转换（强制类型转换）两种。
- 显式转换（强制类型转换）

语法：(type)variableName(同C)

## 算术运算符

- 运算符按照操作数的数量可以分为单目运算符、双目运算符和三目运算符。
- 这里逻辑运算符里的短路与、或和逻辑与、或（即位运算部分）要注意一下。
- Java 提供了一个特别的三元运算符（也叫三目运算符）经常用于取代某个类型的 if-then-else 语句。条件运算符的符号表示为“?:”，使用该运算符时需要有三个操作数，因此称其为三目运算符。

一般语法结构为： image-20220829162116938

## 逻辑运算符

- 简洁运算：||、&&
- 非简洁运算：&、|
- 非简洁运算在必须计算完左右两个表达式之后，才取结果值
- 对于||，只要左边表达式为true，就不计算右边表达式，则整个表达式为true。
- 而简洁运算可能只计算左边的表达式而不计算右边的表达式
- 对于&&，只要左边表达式为false，就不计算右边表达式，则整个表达式为false
- 对于异或运算，若两个值相同，则值为假

```
/*
b = x<y || ++x == --y
x=3
y=5
b=true
*/
int x=3;
int y=5;
boolean b= x<y || ++x == --y;
System.out.println("b = x<y || ++x == --y");
System.out.println("x="+x);
System.out.println("y="+y);
System.out.println("b="+b);

/*
b = x<y | ++x===--y
x=4
y=4
b=true
*/

int x1=3;
```

```

int y1=5;
boolean b1= x1<y1 | ++x1==--y1;
System.out.println();
System.out.println("b = x<y | ++x==--y");
System.out.println("x="+x1);
System.out.println("y="+y1);
System.out.println("b="+b1);

```

## 运算优先级

运算符说明	Java运算符
分隔符	. [] () {} , ;
单目运算符	++ -- ~ !
强制类型转换运算符	(type)
乘法、除法、求余	* / %
加法、减法	+ -
移位运算符	<< >> >>>
关系运算符	< <= > >= instanceof
等价运算符	== !=
按位与	&
按位异或	^
按位或	
条件与	&&
条件或	
三目运算符	?:
赋值	= += -= *= /= &=  = ^= %= <<= >>= >>>=

<https://blog.csdn.net/yuandifang>

## break

格式：break;/break 标号;

可用于switch和循环中，终止某个循环，程序跳转到循环块外的下一条语句

## continue

格式：continue;/continue 标号;

可以用于循环中，跳出本次循环，进入下一次循环

# 常见输出函数

printf 主要继承了 C 语言中 printf 的一些特性，可以进行格式化输出。

print 就是一般的标准输出，但是不换行。

println 和 print 基本没什么差别，就是最后会换行。

## 字符串

### String类

#### 初始和赋值

Java 没有内置的字符串类型，而是在**标准 Java 类库**中提供了一个 String 类来创建和操作字符串。

```
//以下为给String类型赋值的几种方式：
String str1 = new String("Hello Java");

String str2 = new String(str1);

char a[] = {'H','e','l','l','o'};
String schar = new String(a);
a[1] = 's';

char a[]={ 'H','e','l','l','o'};
String schar=new String(a,1,4);
a[1]='s';
```

#### String和int相互转换

除了下面这些，还有三个：valueOf()、parse()和 toString()，是三个“方法”，具体看（C语言中文系列，Java，P162）。

##### String-->int

```
//方法一：Integer.parseInt(str)的方法
int n = Integer.parseInt(str);

//方法二：Integer.valueOf(str).intValue()
int n = Integer.valueOf(str).intValue();
```

##### int-->String

```
//使用第三种方法相对第一第二种耗时比较大。
//方法一：String.valueOf(i);
String str = String.valueOf(num);

//方法二：Integer.toString(i);
String str2 = Integer.toString(num);

//方法三："" + i;
String str3 = num + "";
```

## String的一些“方法”

接下来的方法都是“打包”好了的，一般都是用“.”就可以引用的那种，不用记，直接回来查吧。。。

### 字符串拼接

连接运算符“+”

concat() 方法：字符串1.concat(字符串2);

### 字符串长度

字符串.length()

### 字符串大小写转换

```
// 将字符串中的字母全部转换为小写，非字母不受影响
字符串名.toLowerCase();
// 将字符串中的字母全部转换为大写，非字母不受影响
字符串名.toUpperCase();
```

### 字符串去除空格

```
字符串名.trim();
```

trim() 只能去掉字符串中前后的半角空格（英文空格），而无法去掉全角空格（中文空格），可用以下代码将全角空格替换为半角空格再进行操作，其中替换是 String 类的 replace() 方法：

```
str = str.replace((char) 12288, ' '); // 将中文空格替换为英文空格
str = str.trim();
```

### 字符串截取

#### substring(start,end)

- 字符串.substring(start,end)
- 从第star个开始（包含这个），到第end个结束（不包含这个），个数从0开始数。
- star>=end，star和end不可出现负数，不然都会报错

```
eg1:
String str = "helloworld";
String sstr = str.substring(1,5);

System.out.println("str = " + str);
System.out.println("sstr = " + sstr);

eg2:
String day = "Today is Monday";

//substring(0)结果: Today is Monday
//substring(2)结果: day is Monday
//substring(10)结果: onday
//substring(2,10)结果: day is M
//substring(0,5)结果: Today
```

## 字符串分割

String 类的 split() 方法可以按指定的分割符对目标字符串进行分割，分割后的内容存放在字符串数组中。

str.split(String sign)

str.split(String sign,int limit)

- 字符串.str.split(String sign)
- sign为指定的分割符，可用是任意字符串，但注意，"."、"|" 是转移字符，要加"\"。
- 有多种分隔符也可以用 "|"。

```
String colors = "Red,Black,white.Yellow,Blue";
//用字符串数组来存储。
// 不限制元素个数
String[] arr1 = colors.split(",");
// 限制元素个数为 3
String[] arr2 = colors.split(",", 3);
// 分隔符为 " , " 和 " . " 。
String[] arr3 = colors.split(",|\\.");
String words = "account=? and uu =? or n=?";
// 分隔符为"and"和"or"
String[] arr4 = words.split("and|or");
```

## 字符串替换

### replace

- 字符串.replace(String oldChar, String newChar)
- 将目标字符串中的指定字符（串）替换成新的字符（串）

### replaceFirst()

- 字符串.replaceFirst(String regex, String replacement)
- 用于将目标字符串中匹配某正则表达式的第一个子字符串替换成新的字符串，这是学术翻译，正则表达式解释放在碎本里了。个人可以理解成将特定的第一次出现的字符串换成另外的特定的字符串。

### replaceAll()

- 字符串.replaceAll(String regex, String replacement)
- 同上区别在于不是第一次出现才替换，而是所有出现的都替换。

## 字符串比较

### equals()

- str1.equals(str2)
- 比较两字符串是否相同，返回值true、false，区分大小写

### equalsIgnoreCase()

- str1.equalsIgnoreCase(str2)
- 比较两字符串是否相同，语法完全同equals，返回值true、false，不区分大小写



## compareTo()

- str.compareTo(String otherstr)
- 按字典顺序比较两字符串大小，比较是基于字符串各个字符的 Unicode 值，返回值负数、正数和 0。
- str 位于 otherster 之前：负整数；  
str 位于 otherstr 之后：正整数；  
两个字符串相等：0。

## equals()和==的比较

(C语言系列, java, P184-185)

equals比较的是内容，==比较的是对象。

## 字符串查找

字符串查找分为两种形式：一种是在字符串中获取匹配字符（串）的索引值，另一种是在字符串中获取指定索引位置的字符

## indexOf()

- 字符串.indexOf(value)  
或者 字符串.indexOf(value,fromIndex)
- 可以指定从字符串中的某个位置（fromIndex）开始查找，默认从字符串初始开始
- 用于在字符串中获取匹配字符（串）的索引值，返回值为字符（串）在指定字符串中首次出现的索引位置，找不到返回-1

## lastIndexOf()

- 字符串.lastIndexOf(value)  
或者 字符串.lastIndexOf(value,fromIndex)
- 查找策略是从左往右找，默认从字符串结尾查找，可以指定fromIndex
- 用于返回字符（串）在指定字符串中最后一次出现的索引位置，找不到返回-1

## charAt()

- 可以在字符串内根据指定的索引查找字符（字符串=字符数组成立），索引从0开始

## StringBuffer类

可变的字符型，创建后可以随意修改字符串的内容

## 初始和创建

```
//StringBuffer()
//构造一个空的字符串缓冲区，初始化为 16 个字符的容量
StringBuffer str1 = new StringBuffer();

//StringBuffer(int length)
//创建一个空的字符串缓冲区，初始化为指定长度 length 的容量
StringBuffer str2 = new StringBuffer(10);

//StringBuffer(String str)
//创建一个字符串缓冲区，将其内容初始化为指定的字符串内容 str，字符串缓冲区的初始容量为 16 加上字符串 str 的长度
//str3含有（16+4）的字符串缓冲区，一个中文一个字符
StringBuffer str3 = new StringBuffer("编程语言");
```

## StringBuffer的一些“方法”

### 查看容量

```
//str1.capacity() 用于查看 str1 的容量
System.out.println(str1.capacity());
```

### 追加字符串

- StringBuffer 对象.append(String str)
- StringBuffer 类的 append() 方法用于向原有 StringBuffer 对象中追加字符串。

```
StringBuffer str1 = new StringBuffer("hello");
String str2 = new String("world");
str1.append(str2);
```

### 替换字符

- StringBuffer 对象.setCharAt(int index, char ch)
- 用于在字符串的指定索引（index）位置替换一个字符（ch）

### 反转字符串

- StringBuffer 对象.reverse()
- 将字符串反转，然后替换原字符串

### 删除字符串

#### deleteCharAt()

- StringBuffer 对象.deleteCharAt(int index)
- 用于移除序列中指定位置（index）的字符

#### delete()

- StringBuffer 对象.delete(int start,int end)
- 用于移除子字符串的字符，删除从第star个开始，到第end个结束，开始包含，结束不包含。

### 空字符串和null

null 是空引用，表示一个对象的值，没有分配内存，调用 null 的字符串的方法会抛出空指针异常。

空字符串 "" 是长度为 0 的字符串，是一个java对象。

String 变量还可以存放一个特殊的值，名为 null，这表示目前没有任何对象与该变量关联。

### String、StringBuffer和StringBuilder的区别

(详见C语言中文系列，Java，P200)

## Java正则表达式

(参考资料：

<https://juejin.cn/post/6844903677119954958>

<https://juejin.cn/post/6844903680349585422>

[https://www.r2coding.com/#/?](https://www.r2coding.com/#/?id=%e6%ad%a3%e5%88%99%e8%a1%a8%e8%be%be%e5%bc%8f)

[id=%e6%ad%a3%e5%88%99%e8%a1%a8%e8%be%be%e5%bc%8f\)](https://www.r2coding.com/#/?id=%e6%ad%a3%e5%88%99%e8%a1%a8%e8%be%be%e5%bc%8f)

java与正则相关的工具主要在java.util.regex包中；此包中主要有两个类：**Pattern**、**Matcher**。

### 元字符

元字符	说明
.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线或汉字
\s	匹配任意的空白符
\d	匹配数字
\b	匹配单词的开始或结束
^	匹配字符串的开始
\$	匹配字符串的结束

```
//匹配8位数字
^\d{8}$
//匹配是14~18位的数字
^\d{14,18}$
```

### 重复限定符

语法	说明
*	重复零次或更多次
+	重复一次或更多次

语法	说明
?	重复零次或一次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

## Pattern

- 模式类，正则表达式的编译表示形式。

## Matcher


## 概念与理解

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。（来自百度百科）

个人理解：正则表达式就是对字符串进行过滤的“方法”。

## 用法

image-20220902152323201

image-20220902152714518

## 参考链接

<https://juejin.cn/post/6844903677119954958>

<https://juejin.cn/post/6844903680349585422>

<https://www.cnblogs.com/gengaixue/p/8066294.html>

## 数字处理类

## Math类

(C中文系列, Java, P214)

- 用法：Math.方法
- 包含在java.util包里

## 最大最小值和绝对值

## 求整运算

## 三角函数

## 指数运算

## 随机数生成

- Math.random()
- 只能产生 double 类型的 0~1 的随机数（暂时看上去没啥用）

## Random类

(C中文系列, Java, P219)

- **专门用来生成随机数的类**，可以产生 boolean、int、long、float、byte 数组以及 double 类型的随机数
- 所有方法生成的随机数字都是均匀分布，区间内部生成的概率是均等的
- 包含在java.util包里

```
Random r = new Random();  
//生成一个布尔型的值 (true、false)  
System.out.println(r.nextBoolean());  
// 生成[0,10]区间的整数  
System.out.println(r.nextInt(10));
```

## DecimalFormat 类

(C中文系列, Java, P222)

- **专门用于对数字进行格式化处理的类**（**自己编辑**输出的格式，比如保留几位啥的），例如将小数位统一成 2 位，不足 2 位的以 0 补齐。DecimalFormat 是 NumberFormat 的一个子类，用于格式化十进制数字。
- 包含在java.text包里

```
double number = 3.145;  
DecimalFormat def1 = new DecimalFormat("#.00");  
System.out.println(def1.format(3.145));  
System.out.println(def1.format(number));
```

## BigInteger 类

(C中文系列, Java, P224)

- 构造语法: BigInteger(**String** val)
- **用于大数字运算的类**，用于高精度计算。BigInteger 类是针对**整型**大数字的处理类。
- 数字范围比 Integer 类型的大得多，支持任意精度的整数。
- BigInteger 类还封装了很多操作，像求绝对值、相反数、最大公约数以及判断是否为质数等。
- 加(add)减 (subtract)乘(multiply)除(divide)  
divideAndRemainder也是除法，但是返回数组，第一个值是商，第二个是余数
- 包含在java.math包里

```
String bignumber = "958125125654";
String smallnumber = "100202220000";
BigInteger bi = new BigInteger(bignumber);
BigInteger si = new BigInteger(smallnumber);

System.out.println(bi.add(si));
```

## BigDecimal类

(C中文系列, Java, P224)

- 构造: BigDecimal(double val)、BigDecimal(String val), 实例化时将对应类型转换为 BigDecimal 类型。
- **用于大数字运算的类**, 用于高精度计算。BigDecimal类加入了**小数的概念**, **支持任意精度的浮点数**, 可以用于精确计算货币值。
- 包含在java.math包里

```
BigDecimal bignumber = new BigDecimal("51232566.365426");
BigDecimal smallnumber = new BigDecimal("2");
```

```
//除法也可以divide(BigDecimal divisor,int scale,int roundingMode ), 分别为除法操作,
除数、商小数点后的位数和近似值处理模式(处理模式有参考表格)。
System.out.println(bignumber.divide(smallnumber));
```

## 日期类

### Data类

(C中文系列, Java, P229)

- 主要封装系统日期和时间的信息。**对时间日期获取的类**, 表示系统特定的时间戳, 可以精确到毫秒。

### 构造

#### 无参数构造

- 获取的是系统当前的时间, 显示顺序为星期、月、日、小时、分、秒、年。

#### 带 long 参数构造

- 获取的是距离 GMT 指定毫秒数的时间, 60000 毫秒是一分钟。
- 注意GMT (格林尼治标准时间) 是东一区的时间, CST (中央标准时间) 是东八区, 中间相差8小时。

```
//无参数构造
Date del = new Date();
System.out.println(del.toString());
//带 long 参数构造
Date del2 = new Date(60000);
System.out.println(del2.toString());
```

# Calendar类

(C中文系列, Java, P229)

- 是一个**抽象类**, 有自己的静态常量, **是对日期时间进行手动处理的类**。
- 创建 Calendar 对象不能使用 new 关键字, 因为 Calendar 类是一个抽象类, 但是可用 getInstance() 方法。

```
//getInstance() 方法返回一个 Calendar 对象, 其日历字段已由当前日期和时间初始化,getInstance
Calendar c = Calendar.getInstance();

//setTime设置时间,需要Date类型, new Date获取系统时间。
c.setTime(new Date());

//set设置时间
//至少需要年月日, 时分秒不设置默认采用当前值。
c.set(2012,8,8);
c.set(2002,8,8,15,8,36);
//可以单独设置对应静态常量的值
c.set(Calendar.YEAR, 1999);

//getTime返回Date对象表示时间。
System.out.println(c.getTime());

//get返回对应静态常量的值
System.out.println(c.get(Calendar.YEAR));
```

# DateFormat类

(C中文系列, Java, P236)

- **对日期进行格式化** (就是改变表示格式) **的类**。有自己的静态常量。允许进行格式化 (也就是日期→文本)、解析 (文本→日期) 和标准化日期。
- 是一个**抽象类**, 创建时不能用new, 要用getDateInstance()。

```
//设置时间日期的格式, 日期默认只有年月日, 地区默认中国, 不可单独设置地区。
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getDateInstance(DateFormat.FULL);
DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL, Locale.CHINA);

//format将Date转化为字符串String
String date13 = df3.format(new Date());

//输出字符串
System.out.println(date13);
```

# SimpleDateFormat类

(C中文系列, Java, P239)

- DateFormat 类的子类, **对日期进行格式化自行编辑处理** (就是改变表示格式) **的类**。
- 可以用new创建。

```
//编辑格式，默认格式：2022/9/6 下午5:44。  
//HH: 03/13; H: 3/13;  
SimpleDateFormat fd = new SimpleDateFormat("今天是" + "yyyy年MM月dd日，E，HH点mm分ss秒");  
  
Date times = new Date();  
System.out.println(fd.format(times));
```

## 包装类

- Java将每种**基本数据类型**分别设计了对应的类，称为包装类。也有地方称为外覆类或数据类型类。
- 基本数据类型转换为包装类的过程称为**装箱**
- 包装类变为基本数据类型的过程称为**拆箱**
- 应用：ing和Integer转换、字符串-->数值、整数-->字符串

```
int m = 5;  
//自动装箱（自动转换）  
Integer obj1 = m;  
//自动拆箱（自动转换）  
System.out.println(obj1);  
  
Integer obj2 = 500;  
System.out.println(obj2.equals(obj1));  
  
String str1 = "30";  
String str2 = "30.3";  
  
// 将字符串变为 int 型  
int x = Integer.parseInt(str1);  
// 将字符串变为 float 型  
float y = Float.parseFloat(str2);  
// 将整数转换为字符串  
String s = Integer.toString(m);
```

## Integer类

(C中文系列, Java, P259)

### 构造

```
// 以 int 型变量作为参数创建 Integer 对象  
Integer integer1 = new Integer(100);  
// 以 String 型变量作为参数创建 Integer 对象  
Integer integer2 = new Integer("100");
```

### 常量



```
// 获取 int 类型可取的最大值
int max_value = Integer.MAX_VALUE;
// 获取 int 类型可取的最小值
int min_value = Integer.MIN_VALUE;
// 获取 int 类型的二进制位
int size = Integer.SIZE;
// 获取基本类型 int 的 Class 实例
Class c = Integer.TYPE;
```

## Float类

(C中文系列, Java, P262)

### 构造

```
// 以 double 类型的变量作为参数创建 Float 对象
Float float1 = new Float(3.14145);
// 以 float 类型的变量作为参数创建 Float 对象
Float float2 = new Float(6.5);
// 以 String 类型的变量作为参数创建 Float 对象
Float float3 = new Float("3.1415");
```

### 常量

(常用的)

```
// 获取 float 类型可取的最大值
float max_value = Float.MAX_VALUE;
// 获取 float 类型可取的最小值
float min_value = Float.MIN_VALUE;
// 获取 float 类型可取的最小标准值
float min_normal = Float.MIN_NORMAL;
// 获取 float 类型的二进制位
float size = Float.SIZE;
```

## Double类

(C中文系列, Java, P242)

### 构造

```
// 以 double 类型的变量作为参数创建 Double 对象
Double double1 = new Double(5.456);
// 以 String 类型的变量作为参数创建 Double 对象
Double double2 = new Double("5.456");
```

### 常量

(自己找)

## Character类

(C中文系列, Java, P267)

- char的包装类。

### 构造

```
Character character = new Character('S');
```

## Boolean类

(C中文系列, Java, P272)

### 构造

```
//boolean值为 true
Boolean b1 = new Boolean(true);
Boolean b3 = new Boolean("true");
//boolean值为 false
Boolean b4 = new Boolean(false);
Boolean b2 = new Boolean("ok");
```

### 常量

- TRUE: 对应基值 true 的 Boolean 对象。
- FALSE: 对应基值 false 的 Boolean 对象。
- TYPE: 表示基本类型 boolean 的 Class 对象。

## Byte类

### 构造

```
//通过byte创建Byte
byte tmp = 5;
Byte a = new Byte(tmp);
//通过String创建Byte
Byte b = new Byte("1000.2");
```

## Number类

(C中文系列, Java, P245)

- Number 是一个**抽象类**, 也是一个超类 (即父类)。抽象类不能直接实例化, 而是必须实例化其具体的子类。
- 所有的包装类 (如 Double、Float、Byte、Short、Integer 以及 Long) 都是抽象类 Number 的子类
- 属于 java.lang 包

```
Number num = new Double(12.55265);
System.out.println(num.doubleValue());
System.out.println(num.intValue());
System.out.println(num.floatValue());
```

# Object类

(C中文系列, Java, P252)

- Object 是 Java 类库中的一个特殊类, 也是所有类的父类, 当一个类被定义后, 如果没有指定继承的父类, 那么默认父类就是 Object 类。
- 任何 Java 对象都可以调用 Object 类的方法
- toString()、equals() 方法和 getClass() 方法在 Java 程序中比较常用

## getClass()

- getClass() 方法**返回对象所属的类**, 是一个 Class 对象。通过 Class 对象可以获取该类的各种信息, 包括类名、父类以及它所实现接口 的名字等

## 接收任意引用类型的对象 (没看懂)

(C中文系列, Java, P256)

- 所有的对象都可以向 Object 进行转换,即一切的引用数据 类型都可以使用 Object 进行接收。

# System类

(C中文系列, Java, P276)

- System 类代表当前 Java 程序的运行平台, 系统级的很多属性和控制方法都放置在该类的内部。由于该类的构造方 法是 private 的, 所以无法创建该类的对象, 也就是无法实例化该类。
- 位于 java.lang 包

## 成员变量

(C中文系列, Java, P276)

PrintStream out、InputStream in 和 PrintStream err为system类的三个静态成员变量。

## PrintStream out

## InputStream in

## PrintStream err

## 成员方法

(C中文系列, Java, P277)

System 类中提供一些系统级的操作方法。常用的方法有 arraycopy()、currentTimeMillis()、exit()、gc() 和 getProperty()。

## arraycopy()

- ```
public static void arraycopy(Object src,int srcPos,Object dest,int destPos,int length
```
- src 源数组, srcPos 源数组中复制的**起始位置**, dest 目标数组, destPos 复制到目标数组的**起始位置**, length 复制数组的元素个数。
- 复制数组

## currentTimeMillis()

- `long m = System.currentTimeMillis();`
- 将获得一个长整型的数字，该数字就是以差值表达的当前时间。
- 返回当前计算机时间

```
//计算程序运行时间
long start = System.currentTimeMillis();
for (int i = 0; i < 100000000; i++)
{
    int temp = 0;
}
long end = System.currentTimeMillis();
long time = end - start;
System.out.println("程序执行时间" + time + "秒");
```

## exit()

- 终止程序

## gc()

- 请求系统垃圾回收，清除内存垃圾。

## getProperty()

- 获取系统中属性名为key的属性对应的值。（key让我想起了“句柄”）

```
String jversion = System.getProperty("java.version");
String oName = System.getProperty("os.name");
String userName = System.getProperty("user.name");

System.out.println(jversion);
System.out.println(oName);
System.out.println(userName);
```

# AveNumber类

平均值类

Average(a,b): 求两个数的平均值

Average(a,b,c): 求三个数的平均值

Average(a[],n): 求包含n个成员的数组a中所有成员的均值

## 数组

- java数组工作原理和C的不同。
- java数组下标从零开始，下标访问运算符是中括号（arr[10]），长度属性是length。
- 数组类型是从抽象基类Array派生的引用类型，是**引用数据类型**。数组元素可以是任何类型，包括数组类型。

- int 类型是基本类型，但 int[] 是引用类型。引用数据类型在使用之前一定要做声明和初始化这两件事情。

## 一维数组


### 创建

(C中文系列, Java, P282)

```
int[] arr2;  
int arr1[];
```

- 两个都可以，但是推荐用第一类。
- 声明数组时不需要规定数组长度，但不可漏写[]。

以下为原文 (C中文系列, Java, P282)

 image-20220910170137012

### 分配空间

```
arr2 = new int[10];
```

### 初始化

```
//三种方式  
int[] arr2 = new int[]{1,2,3,4,5};  
int[] arr2 = new int[5]{1,2,3,4,5};  
int[] arr2 = {1,2,3,4,5};
```

## 二维数组

同C。

## 多维数组

(C中文系列, Java, P295)

(C中文系列, Java, P300)

- 多维数组被解释为是数组的数组。
- 三维数组、四维数组和五维数组等，它们都属于多维数组。
- 想要提高数组的维数，只要在声明数组时将索引与中括号再加一组即可。

## 不规则数组

- 多维数组被解释为是数组的数组，所以因此会衍生出一种不规则 数组。
- 动态初始化不规则数组比较麻烦，不能使用 new 语句，而是先初始化高维数组，然后再分别逐个初始化低维数组。

```
//静态初始化
int arr[][] = {{1,2}, {11}, {21,22,23}, {31,32,33}};

//动态初始化
//先初始化高维数组为 4
int arr[][] = new int[4][];
// 逐一初始化低维数组
arr[0] = new int[2];
arr[1] = new int[1];
arr[2] = new int[3];
arr[3] = new int[3];
```

## 数组与字符串

### 字符串-->数组

#### toCharArray()

```
String str = "123abc";
char[] arr = str.toCharArray(); // char 数组
for (int i = 0; i < arr.length; i++)
{
    System.out.println(arr[i]); // 输出 1 2 3 a b c
}
```

### 数组-->字符串

#### char字符数组-->字符串

```
char[] arr = { 'a', 'b', 'c' };
String string = String.valueOf(arr);
System.out.println(string); // 输出 abc
```

#### String字符串数组-->字符串

```
String[] arr = { "123", "abc" };
//StringBuffer: 可变字符串类型。
StringBuffer sb = new StringBuffer();

for (int i=0;i<arr.length;i++)
{
    //StringBuffer的append, 追加字符。
    sb.append(arr[i]);
}

//将StringBuffer转换成String
String str = sb.toString();
```

# Arrays类

(C中文系列, Java, P300)

- 是一个工具类，其中包含了数组操作的很多方法。
- 类里均为 static 修饰的方法（static 修饰的方法可以直接通过 类名调用），可以直接通过 **Arrays.xxx(xxx)** 的形式调用方法。
- 处于 java.util 包下
- 该类的方法都需要导入 java.util.Arrays 包。

## binarySearch

- 二分法查询元素值在数组中出现的索引，如果 a 数组不包含元素值，则返回负数。
- 要求数组中元素已经按升序排列。
- **int binarySearch(type[] a, type key)**  
**int binarySearch(type[] a, int fromIndex, int toIndex, type key)**
- fromIndex 到 toIndex 是索引范围。

## copyOf

- 把 original 数组复制成一个新数组，其中 length 是新数组的长度。
- length<original 数组长度，新数组为原数组前面 length 个元素。
- length>original 数组长度，新数组前面元素就是原数组所有元素，后面补充 0（数值类型）、false（布尔类型）或者 null（引用类型）。
- 属于浅拷贝（浅复制）。浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化。
- **type[] copyOf(type[] original, int length)**  
**type[] copyOfRange(type[] original, int from, int to)**
- from 到 to 是索引范围。

## CopyOfRange

- 复制数组
- startIndex 必须在 0 到 srcArray.length 之间
- endIndex >= startIndex，可以 > srcArray.length，如果大于 srcArray.length，则目标数组中使用默认值填充
- 属于浅拷贝（浅复制）。浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化。
- **Arrays.copyOfRange(dataType[] srcArray,int startIndex,int endIndex)**

## equals

- 数组完全相同，返回true。
- **boolean equals(type[] a, type[] a2)**

## fill

- 将会把 a 数组的所有元素都赋值为 val，可用于批量初始化数值。
- **void fill(type[] a, type val)**  
**void fill(type[] a, int fromIndex, int toIndex, type val)**
- fromIndex 到 toIndex 是索引范围。

## sort

- 对 a 数组的数组元素进行排序。
- **void sort(type[] a)**  
**void sort(type[] a, int fromIndex, int toIndex)**
- fromIndex 到 toIndex 是索引元素范围。

## 降序方法

(C中文系列, Java, P322)

两个办法，一个是用Collections类的方法，另一个是实现Comparator 接口的复写 compare() 方法（但看不懂，暂时没记录上去）

```
//要想改变默认的排列顺序，不能使用基本类型（int,double,char），要使用它们对应的类（包装类）
Integer[] a = { 9, 8, 7, 2, 3, 4, 1, 0, 6, 5 };

//利用Collections类的reverseOrder，Collections 是一个包装类。
Arrays.sort(a, Collections.reverseOrder());
```

## toString

- 将一个数组转换成一个字符串。
- 把多个数组元素连缀在一起，多个数组元素使用英文逗号,和空格隔开。
- **String toString(type[] a)**

—————（以下为Java 8 为 Arrays 类增加的工具方法）—————

(C中文系列, Java, P305)

## 其他方法

### clone()

- 复制数组，是类 Object 中的方法，可以创建一个有单独内存空间的对象。
- 因为数组也是一个 Object 类，因此也可以使用数组对象的 clone() 方法来复制数组
- 属于浅拷贝（浅复制）。浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的 值，另一个值都会随之变化。

```
int scores[] = new int[] { 100, 81, 68};
//(int[])强制类型转换。
int newScores[] = (int[]) scores.clone();
```



# ArrayList

## 排序

来源: <https://www.mianshigee.com/note/detail/82361gbd/>

## 冒泡排序法

---

(C中文系列, Java, P324)

基本思想是: 对比相邻的元素值, 如果满足条件就交换元素值, 把较小的元素值移动到数组前面, 把大的元素值移动到数组后面 (也就是交换两个元素的位置), 这样数组元素就像气泡一样从底部上升到顶部。

## 快速排序法

---

(C中文系列, Java, P326)

基本思想是: 通过一趟排序, 将要排序的数据分隔成独立的两部分, 其中一部分的所有数据比另外一部分的所有数据都要小, 然后再按此方法对这两部分数据分别进行快速排序, 整个排序过程可以递归进行, 以此使整个数据变成有序序列。

## 选择排序法

---

(C中文系列, Java, P328)

选择排序是指每一趟从待排序的数据元素中选出最大 (或最小) 的一个元素, 顺序放在已排好序的数列的最后, 直到全部待排序的数据元素排完。

## 直接插入排序法

---

(C中文系列, Java, P330)

基本思想是: 将  $n$  个有序数存放在数组  $a$  中, 要插入的数为  $x$ , 首先确定  $x$  插在数组中的位置  $p$ , 然后将  $p$  之后的元素都向后移一个位置, 空出  $a(p)$ , 将  $x$  放入  $a(p)$ , 这样可实现插入  $x$  后仍然有序。

## 类和对象

---

类是对象的模板, 对象是类的实例。

基本概念

- 类是对象的抽象, 对象是类的具体
- 类是概念模型, 定义对象的所有特性和所需的操作, 对象是真实的模型, 是一个具体的实体。
- 类是描述了一组有相同特性 (属性) 和相同行为 (方法) 的一组对象的集合
- 类是实体对象的概念模型, 因此通常是笼统的、不具体的。
- 类是构造面向对象程序的基本单位, 是抽取了同类对象的共同属性和方法所形成的对象或实体的“模板”。
- 对象或实体所拥有的特征在类中表示时称为类的属性。
- 对象执行的操作称为类的方法。

- 对象是现实世界中实体的 描述，对象要创建才存在，有了对象才能对对象进行操作。

## 面向对象设计

(C中文系列, Java, P352)

### 继承

- Java 语言是单继承的，即只能有一个父类，可以防止多继承所引起的冲突问题。
- Java 可以实现多个接口（接口类似于类，但接口的成员没有执行体）。

### 封装

- 封装将代码及其处理的数据绑定在一起的一种编程机制，该机制保证了程序和数据都不受外部干扰且不被误用。
- 封装的目的在于保护信息。
- Java 语言的基本封装单位是类。
- Java 提供了私有和公有的访问模式，类的公有接口代表外部的用户应该知道或可以知道的每件东西，私有的方法数据只能通过该类的成员代码来访问，这就可以确保不会发生不希望的事情。

### 多态

- 多态性体现在父类中定义的属性和方法被子类继承后，可以具有不同的属性或表现方式。
- **一个接口，多个方法。**
- 多态性允许一个接口被多个同类使用，弥补了单继承的不足。

## 类

### 类的定义

Java 中定义一个类，需要使用 **class 关键字、一个自定义的类名和一对表示程序体的大括号**。完整语法如下：

```
[public][abstract|final]class<class_name>[extends<class_name>]
[implements<interface_name>] {
    // 定义属性部分
    <property_type><property1>;
    <property_type><property2>;
    <property_type><property3>;
    ...
    // 定义方法部分
    function1();
    function2();
    function3();
    ...
}
```

**public**：表示“共有”的意思。如果使用 public 修饰，则可以被其他类和程序访问。每个 Java 程序的主类都必须是 public 类，作为 公共工具供其他类和程序使用的类应定义为 public 类。

**abstract**：如果类被 abstract 修饰，则该类为抽象类，抽象类不能被实例化，但抽象类中可以有抽象方法（使用 abstract 修饰的方法）和具体方法（没有使用 abstract 修饰的方法）。继承该抽象类的所有子类都必须实现该抽象类中的所有抽象方法（除非子类也是抽象 类）。

**final**：如果类被 final 修饰，则不允许被继承

class：声明类的关键字。

extends：表示继承其他类。

implements：表示实现某些接口。

property\_type：表示成员变量的类型。

property：表示成员变量名称。

function()：表示成员方法名称。

## 属性

### 成员变量

- 类的成员变量可以分为
  - 1.静态变量（或称为类变量）：被 static 修饰的成员变量。
  - 2.实例变量：没有被 static 修饰的成员变量。

### 实例变量

- 每创建一个实例，Java 虚拟机就会为实例变量分配一次内存。
- 在类的内部，可以在非静态方法中直接访问实例变量。
- 在本类的静态方法或其他类中则需要通过类的实例对象进行访问。

### 语法

```
[public|protected|private][static][final]<type><variable_name>
```

### 初始化默认值

整数型（byte、short、int 和 long）的基本类型变量的默认值为 0。

单精度浮点型（float）的基本类型变量的默认值为 0.0f。

双精度浮点型（double）的基本类型变量的默认值为 0.0d。

字符型（char）的基本类型变量的默认值为 “\u0000”。


布尔型的基本类型变量的默认值为 false。

数组引用类型的变量的默认值为 null。如果创建了数组变量的实例，但没有显式地为每个元素赋值，则数组中的元素初始化值采用数组 数据类型对应的默认值

### 行为

行为表示一个对象能够做的事情或者能够从一个对象取得的信息。

### 成员方法

- 声明成员方法可以定义类的行为，
- 类的各种功能操作都是用方法来实现的，属性只不过提供了相应的数据。
- 一个完整的方法通常包括方法名称、方法主体、方法参数和方法返回值类型。
- image-20221125100733808
- 与成员变量类似，成员方法也可以分为以下两种

1.静态方法（或称为类方法）：被 static 修饰的成员方法。

2.实例方法：没有被 static 修饰的成员方法。

## 构造方法

(C中文系列, Java, P410)

- **方法名必须与类名相同**
- **可以有 0 个、1 个或多个参数。**在一个类中，与类名相同的方法就是构造方法。每个类可以具有多个构造方法，但要求它们各自包含不同的方法参数。
- **没有任何返回值，包括 void。**类的构造方法是有返回值的，当使用 new 关键字来调用构造方法时，构造方法返回该类的实例，可以把这个类的实例当成构造器的返回值，因此构造器的返回值类型总是当前类，无须定义返回值类型。但必须注意不要在构造方法里使用 return 来返回当前类的对象，因为构造方法的返回值是隐式的
- **默认返回类型就是对象类型本身。**构造方法不能被 static、final、synchronized、abstract 和 native（类似于 abstract）修饰。构造方法用于初始化一个新对象，所以用 static 修饰没有意义。构造方法不能被子类继承，所以用 final 和 abstract 修饰没有意义。多个线程不会同时创建内存地址相同的同一个对象，所以用 synchronized 修饰没有必要。
- **只能与 new 运算符结合使用**

## 析构方法

(C中文系列, Java, P415)

析构方法与构造方法相反，当对象脱离其作用域时（例如对象所在的方法已调用完毕），系统自动执行析构方法。

析构方法往往用来做 清理垃圾碎片的工作，例如在建立对象时用 new 开辟了一片内存空间，应退出前在析构方法中将其释放。

## 实例方法

- 在实例方法中可以直接访问所属类的静态变量、静态方法、实例变量和实例方法。
- 在访问非静态方法时，需要通过实例对象来访问

## 语法

```
public class Test {  
    [public|private|protected][static]<void|return_type><method_name>([paramList]) {  
        // 方法体  
    }  
}
```

## 返回值

用return

## 形参、实参

- 形参是定义方法时参数列表中出现的参数
- 形参只有在方法内部有效，方法调用结束返回主调方法后则不能再使用该形参变量。
- 实参是调用方法时为方法传递的参数
- 实参和形参在数量、类型和顺序上应严格一致，否则会发生“类型不匹配”的错误。
- 在方法调用过程中，形参的值发生改变，而实参中的值不会变化。

## 调用成员方法

程序中执行到调用成员方法时，Java 把实参值复制到一个临时的存储区（栈）中，形参的任何修改都在栈中进行，当退出该成员方法时，Java 自动清除栈中的内容。

## 方法的可变参数

格式：methodName({paramList},paramType...paramName)

methodName 表示方法名称；paramList 表示方法的固定参数列表；paramType 表示可变参数的类型；... 是声明可变参数的标识；paramName 表示可变参数名称。

```
public void print(String...names) {
}
public static void main(String[] args) {
    StudentTestMethod student = new StudentTestMethod();
    // 传入 3 个值
    student.print("张强","李成","王勇");
    student.print("马丽","陈玲");
}
```

## 内部类

(C中文系列, Java, P466)

- **在类内部可定义成员变量和方法，且在类内部也可以定义另一个类。**
- 在类 Outer 的内部再定义一个类 Inner，此时类 Inner 就称为 内部类（或称为嵌套类），而类 Outer 则称为外部类（或称为宿主类）。内部类与外部类不能重名。
- **内部类拥有外部类的所有元素的访问权限。**内部类可以很好地实现隐藏，一般的非内部类是不允许有 private 与 protected 权限的，但内部类可以。
- 外部类访问级别：public、默认；内部类访问级别：public、protected、private、默认。
- 内部类分为实例内部类、静态内部类和成员内部类。（具体图示看pdf）
- **java实现多重继承的方法：接口和内部类。**

(C中文系列, Java, P477)

•

```
public class Test
{
    public class InnerClass
    {
        public int getSum(int x,int y)
        {
            return x+y;
        }
    }
    public static void main(String[] arge)
    {
        // InnerClass 为内部类的类名
        InnerClass ic = new InnerClass();

        // Test.InnerClass 是内部类的完整类名
        Test.InnerClass ti = new Test().new InnerClass();
    }
}
```

```
}  
}
```

## 实例内部类（为非静态内部类）

实例内部类是指没有用 `static` 修饰的内部类。

### 特点

- 在外部类的静态方法和外部类以外的其他类中，必须通过外部类的实例创建内部类的实例。
- 在实例内部类中，可以访问外部类的所有成员。（如果有多层嵌套，则内部类可以访问所有外部类的成员）
- 在外部类中不能直接访问内部类的成员，而必须通过内部类的实例去访问。（如果类 A 包含内部类 B，类 B 中包含内部类 C，则在类 A 中不能直接访问类 C，而应该通过类 B 的实例去访问类 C。）
- 外部类实例与内部类实例是一对多的关系，也就是说一个内部类实例只对应一个外部类实例，而一个外部类实例则可以对应多个内部类实例。
- 在实例内部类中不能定义 `static` 成员，除非同时使用 `final` 和 `static` 修饰

```
//外部类  
public class Outer  
{  
    public int a = 10;  
    static int b = 10;  
    final int c = 10;  
    private int d = 10;  
  
    //实例内部类（非静态内部类）  
    class Inner  
    {  
        //访问外部类的变量（什么类型都可）  
        int a2 = a;  
        int b2 = b;  
        int c2 = c;  
        int d2 = d;  
  
        int a = 20;  
        int f1 = a; //f1=20  
        int f2 = this.a; //f2=20  
        int f3 = Outer.this.a; //f3=10  
  
        //访问外部类的实例方法  
        method1();  
        //访问外部类的静态方法  
        method2();  
    }  
  
    //创建内部类实例（不需要创建外部类实例）  
    Inner i = new Inner();  
  
    //外部类的方法  
    public void method1()  
    {  
        //创建内部类实例（不需要创建外部类实例）  
        Inner i = new Inner();  
    }  
}
```

```

//外部类的静态方法
public static void method2()
{
    //创建内部类实例(需要创建外部类实例)
    Inner i = new Outer().new Inner();
}

//外部类的其他内部类
class OtherInner
{
    //创建内部类实例(不需要创建外部类实例)
    Inner i = new Inner();
}

public static void main(String[] args)
{
    //创建内部类实例
    Inner i = new Outer().new Inner();
}
}
class OtherClass
{
    //创建内部类实例(需要创建外部类实例)
    Outer.Inner i = new Outer().new Inner();
}
}

```

## 静态内部类

静态内部类是指使用 static 修饰的内部类。

### 特点

- **创建静态内部类的实例时，不需要创建外部类的实例。**
- **静态内部类中可以定义静态成员和实例成员。**（外部类以外的类，访问静态对象需要通过完整的类名；访问实例成员需要通过静态内部类的实例）
- **静态内部类可以直接访问外部类的静态成员。**访问外部类的实例成员，则需要通过外部类的实例。

```

//外部类
public class Outer
{
    //外部类实例变量
    int a = 0;
    //外部类静态变量
    static int b = 0;

    static class Inner
    {
        //静态内部类实例变量
        int a = 0;
        //静态内部类静态变量
        static int b = 0;
    }
}

```

```

        Outer o = new Outer();
        //访问外部类实例成员
        int a3 = o.a;
        //访问外部类静态成员
        int b3 = b;
    }
}
//其他类
class OtherClass
{
    //创建静态内部类
    Outer.Inner oi = new Outer.Inner();
    //访问静态内部类实例成员
    int a2 = oi.a;
    //访问静态内部类静态成员
    int b2 = Outer.Inner.b;
}

```

## 局部内部类

局部内部类是指在一个方法中定义的内部类。

### 特点

- **局部内部类、局部内部类中的内部类**不能使用**访问控制修饰符**（public、private 和 protected）和 **static 修饰符**修饰。
- **局部内部类**只在当前方法中有效。
- **局部内部类**中不能定义 **static 成员**。
- **局部内部类**中可以访问外部类的所有成员。
- 局部内部类中只可以访问**当前方法中 final 类型的参数与变量**。
- 局部内部类方法中的成员与外部类中的成员同名，可以使用<OuterClassName>.this.<MemberName>的形式访问外部类中的成员。

```

public class Test
{
    //以下都会编译错误
    /*
    Inner i = new Inner();
    Test.Inner ti = new Test.Inner();
    Test.Inner ti2 = new Test().new Inner();
    */

    int a = 0;
    int d = 0;

    public void method()
    {
        int b = 0;
        final int c = 0;
        final int d = 10;
        //局部内部类
        class Inner
        {
            //编译错误

```



```

        //b2 = b;

        //访问方法中的成员
        int a2 = a;
        int d2 = d; //d2=10
        //访问外部类的成员
        int d3 = Test.this.d; //d3=0
    }

    //创建局部内部类
    Inner i = new Inner();
}
}

```

## 匿名类（匿名内部类）

匿名类是指没有类名的内部类，必须在创建时使用 new 语句来声明类。

- 这种形式的 new 语句声明一个新的匿名类，它对一个给定的类进行扩展，或者实现一个给定的接口。使用匿名类可使代码更加简洁、紧凑，模块化程度更高。
- 实现方式有两种，一种是继承一个类，重写其方法；另一种是实现一个接口（可以是多个），实现其方法。

```

public class Out
{
    void show()
    {
        System.out.println("Out-show");
    }
}

public class Test
{
    private void show()
    {
        //在方法中构造匿名内部类
        Out anonyInter = Out()
        {
            //获取匿名内部类中的实例
            void show()
            {
                System.out.println("Test-show-anonyInter-show");
            }
        };

        anonyInter.show();
    }

    public static void main(String[] args)
    {
        Test test = new test();
        test.show(); //输出Test-show-anonyInter-show
    }
}

```

```
}  
}
```

## 特点

- **匿名类和局部内部类一样，可以访问外部类的所有成员。**如果匿名类位于一个方法中，则匿名类只能访问方法中 final 类型的局部变量和参数。

```
public static void main(String[] args)  
{  
    int a = 10  
    final int b = 100;  
    Out anonyInter = new Out()  
    {  
        void show()  
        {  
            //编译错误  
            //System.out.println("main-anonyinter-show" + a);  
            System.out.pritln("main-anonyInter-show" + b);  
        }  
    };  
  
    anonyInter.show();  
}
```

- **匿名类中允许使用非静态代码块进行成员初始化操作。**

```
Out anonyInter = new Out()  
{  
    int i;  
    { //非静态代码块  
        i=10; //成员初始化  
    }  
  
    public void show()  
    {  
        System.out.println("anonyInter-show" + i);  
    }  
};
```

- **匿名类的非静态代码块会在父类的构造方法之后被执行。**

## 实现多重继承

(C中文系列, Java, P477)

- **内部类可以继承一个与外部类无关的类，从而保证内部类的独立性，正是基于这一点，多重继承才会成为可能。**

## Effectively final 功能

(C中文系列, Java, P479)

从 Java 8 开始，局部内部类和匿名内部类访问的局部变量可以不加 final 修饰符，由系统默认添加Java将这个功能称为 Effectively final 功能。

它不要求程序员必须将访问的局部变量显式的声明为 final 的。只要该变量不被重新赋值就可以。

一个非 final 的局部变量或方法参数，其值在初始化后就从未更改，那么该变量就是 effectively final。

## 覆盖

### 方法

子类从父类继承的方法进行重写。

## 重载

### 方法

类对自身已有的方法的重写，且要求重载的方法必须有不同的参数列表

## 对象

### 创建

(C中文系列, Java, P371)

- 每个对象都是相互独立的，在内存中占有独立的内存地址，并且每个对象都具有自己的生命周期，当一个对象的生命周期结束时，对象就变成了垃圾，由 Java 虚拟机自带的垃圾回收机制处理。
- 无论采用哪种方式创建对象，Java 虚拟机在**创建一个对象时都包含以下步骤**：
  - 给对象分配内存。
  - 将对象的实例变量自动初始化为其变量类型的默认值。
  - 初始化对象，给实例变量赋予正确的初始值。

### 显式创建对象

(C中文系列, Java, P371)

#### new 关键字

- 类名 对象名 = **new** 类名();
- 会调用类的构造方法。

#### newInstance() 方法

- 调用 java.lang.Class 的 **newInstance()** 方法：
- 会调用类的默认构造方法，即无参构造方法。

#### clone() 方法

- 调用对象的 **clone()** 方法
- 使用该方法创建对象时，要实例化的类必须继承 Cloneable 接口，否则会抛出 java.lang.CloneNotSupportedException 异常。
- 不会调用类的构造方法，它会创建一个复制的对象，这个对象和原来的对象具有不同的内存地址，但它们的属性值相同。

## readObject() 方法

- 调用 java.io.ObjectInputStream 对象的 **readObject()** 方法

```
// 使用 new 关键字创建对象
Student student1 = new Student("小刘", 22);

// 调用 java.lang.Class 的 newInstance() 方法创建对象
Class c1 = Class.forName("Student");
Student student2 = (Student)c1.newInstance();

// 调用对象的 clone() 方法创建对象
Student student3 = (Student)student2.clone();
```

## 隐含创建对象

(C中文系列, Java, P373)

- String strName = "strValue", 其中的"strValue"就是一个 String 对象, 由 Java 虚拟机隐含地创建。
- 字符串的"+"运算符运算的结果为一个新的 String 对象
- 当 Java 虚拟机加载一个类时, 会隐含地创建描述这个类的 Class 实例

## 匿名对象

- 匿名对象就是没有明确的给出名字的对象, 是对象的一种简写形式, 可以作为实际参数传递。
- 匿名对象在实际开发中基本都是作为其他类实例化对象的参数传递的
- 匿名对象实际上就是个堆内存空间, 对象不管是匿名的还是非匿名的, 都必须在开辟堆空间之后才可以使用。

```
//类名称 对象名 = new 类名称();
new Person("张三", 30).tell(); // 匿名对象
```

## 访问对象属性和行为

成员变量对应对象的属性, 成员方法对应对象的行为。

## 对象的销毁

系统会自动进行内存回收, 不需要用户额外处理。

Java 语言的内存自动回收称为垃圾回收 (Garbage Collection) 机制, 简称 GC。

垃圾回收机制是指 JVM 用于释放那些不再使用的对象所占用的内存。

Java 的 Object 类中还提供了一个 protected 类型的 finalize() 方法, 任何 Java 类都可以覆盖这个方法, 在这个方法中进行释放对象所占有的相关资源的操作。

## 方法

### 静态方法

静态方法中不能访问实例成员变量

## 实例方法

实例方法中能够访问静态成员变量和实例成员变量

## Lambda表达式

(C中文系列, Java, P381)

- Lambda 表达式 (Lambda expression) 是一个匿名函数, 基于数学中的 $\lambda$ 演算得名, 也可称为闭包 (Closure)。
- Lambda 表达式是推动 Java 8 发布的重要新特性, 它**允许把函数作为一个方法的参数 (函数作为参数传递进方法中)**
- **Lambda 表达式实现的接口不是普通的接口, 而是函数式接口。**

函数式接口: 有且只有一个抽象的方法 (Object 类中的方法不包括在内) 的接口。

函数式接口只能有一个方法。如果接口中声明多个抽象方法, 那么 Lambda 表达式会发生编译错误 (提示信息为: The target type of this expression must be a functional interface)

- Java8提供的声明函数式接口注释: @FunctionalInterface (具体在C中文系列, Java, P484)
- 优点:

代码简洁, 开发迅速;

方便函数式编程;

非常容易进行并行计算; Java 引入 Lambda, 改善了集合操作 (引入 Stream API)

- 缺点:

代码可读性变差;

在非并行计算中, 很多计算未必有传统的 for 性能要高;

不容易进行调试

## 语法

->被称为箭头操作符或 Lambda 操作符, 箭头操作符将 Lambda 表达式拆分成两部分:

左侧——Lambda 表达式的参数列表。

右侧——Lambda 表达式中所需执行的功能, 用{}包起来, 即 Lambda 体。

```
(参数列表) -> {  
    // Lambda 表达式体  
}
```

```
public interface Calculable  
{  
    int calculateInt(int a,int b);  
} //!!!!注意, 此笔记后所有的Lambda表达式, 若非特殊说明, 都是使用此接口!!!!  
  
public class Test  
{  
    public static Calculable calculate(char opr)  
    {  
        Calculable reslt;  
    }  
}
```

```

        if (opr == '+')
        {
            //匿名内部类实现Calculable接口
            result = new Calculable()
            {
                public int calculateInt(int a,int b)
                {
                    return a+b;
                }
            };

            //Lambda表达式实现Calculable接口
            result = (int a,int b) ->
            {
                return a+b;
            };
        }
        else
        {
            //匿名内部类实现Calculable接口
            result = new Calculable()
            {
                public int calculateInt(int a, int b)
                {
                    return a-b;
                }
            };

            //Lambda表达式实现Calculable接口
            result = (int a,int b) ->
            {
                return a-b
            };
        }
        return result;
    }
}

```

## 简写方式

### 省略参数类型

Lambda 表达式可以根据上下文环境推断出参数类型。

### 省略参数小括号

如果 Lambda 表达式中的参数只有一个，可以省略参数小括号。

### 省略return和大括号

如果 Lambda 表达式体中只有一条语句，那么可以省略 return 和大括号

```

//幂次方功能
public interface Calculable

```

```

{
    int calculateInt(int a);
}

public static Calculable calculate(int power)
{
    Calculable result;
    if (power == 2)
    {
        //result = (int a) 省略参数类型
        result = (a) ->
        {
            return a*a;
        };
    }
    else
    {
        //result = (int a) 省略参数小括号
        result = a ->
        {
            return a*a*a;
        };

        //省略大括号和return
        result = a -> a*a*a;
    }

    return result;
}

```

## 使用

### 作为参数

Lambda 表达式一种常见的用途就是作为参数传递给方法，这需要声明参数的类型声明为函数式接口类型。

```

public static void display(Calculable calc,int n1,int n2)
{
    System.out.println(calc.calculateInt(n1,n2));
}

public static void main(String[] args)
{
    int n1 = 10;
    int n2 = 5;

    display
    ((a,b) ->
    {
        return a+b;
    }, n1, n2);

    display((a,b) -> a-b,n1,n2);
}

```

## 访问变量

### 成员变量

成员变量包括**实例成员变量**和**静态成员变量**。在 Lambda 表达式中可以访问这些成员变量，此时的 Lambda 表达式与普通方法一样，可以读取成员变量，也可以修改成员变量。

Lambda 表达式对静态变量和成员变量可读可写。

```
public class LambdaDemo
{
    private int value = 10;
    private static int staticValue = 5;

    //add 方法是静态方法，静态方法中不能访问实例成员变量，所以代码中的 Lambda 表达式中也不能
    //访问实例成员变量，也不能访问实例成员方法。
    public static Calculable add()
    {
        Calculable result = (int a,int b) ->
        {
            static value++;
            int c = a + b + staticValue;
            //this.value;
            return c;
        };

        return result;
    }

    //sub 方法是实例方法，实例方法中能够访问静态成员变量和实例成员变量，所以代码中的 Lambda
    //表达式中可以访问这些变量，当然实例方法和静态方法也可以访问。
    //当访问实例成员变量或实例方法时可以使用this，如果不与局部变量发生冲突情况下可以省略this。
    public Calculable sub()
    {
        Calculable result = (int a,int b)->
        {
            staticValue++;
            this.value++;
            int c = a - b - staticValue - this.value;
            return c;
        };

        return result;
    }
}
```

### 局部变量

对于成员变量的访问 Lambda 表达式与普通方法没有区别，但是**访问局部变量时，变量必须是 final 类型的（不可改变）**。

Lambda 表达式只能访问局部变量而不能修改，否则会发生编译错误。

```
public class LambdaDemo
```



```

{
    final int localValue = 20;
    public static Calculable add()
    {
        calculable result = (int a,int b) ->
        {
            //编译错误
            //localValue++;
            //localValue = c;

            int c = a + b + localValue;
            return c;
        };

        return result;
    }
}

```

## 方法引用

- 方法引用类似于Lambda表达式的快捷写法，比表达式更加的简洁，可读性更高，有很好的重用性。
- 如果实现比较简单，复用的地方又不多，推荐使用 Lambda 表达式，否则应该使用方法引用。
- Java 8 之后增加了双冒号::运算符，该运算符用于“方法引用”
- “方法引用”虽然没有直接使用 Lambda 表达式，但也与 Lambda 表达式有关，与函数式接口有关。
- 被引用方法的参数列表和返回值类型，必须与函数式接口方法参数列表和方法返回值类型一致
- 语法：ObjectRef::methodName

```

public class LambdaDemo
{
    //静态方法
    public static int add(int a, int b)
    {
        return a+b;
    }

    //实例方法
    public int sub(int a,int b)
    {
        return a-b;
    }
}

public class Test
{
    public static void main(String[] args)
    {
        int n1 = 10;
        int n2 = 5;

        //静态引用，打印实现
        display(LambdaDemo::add,n1,n2);
    }
}

```

```
//实例引用，需先实例化变量
LambdaDemo d = new LambdaDemo();
display(d::sub,n1,n2);
}

public static void display(Calculable calc, int n1, int n2)
{
    System.out.println(calc.calculateInt(n1, n2));
}

}
```

## Lambda和匿名内部类

(C中文系列, Java, P493)

Lambda 表达式与匿名内部类一样，**都可以直接访问 effectively final 的局部变量，以及外部类的成员变量（包括实例变量和类变量）。**

Lambda 表达式创建的对象与匿名内部类生成的对象一样，**都可以直接调用从接口中继承的默认方法。**

(具体的详见pdf)

## this 关键字

(C中文系列, Java, P366)

- 一个方法访问该类中定义的其他方法、成员变量时加不加 this 前缀的效果是完全一样的。
- 对于 static 修饰的方法而言，可以使用类来直接调用该方法，如果在 static 修饰的方法中使用 this 关键字，则这个关键字就无法指向合适的对象。所以，**static 修饰的方法中不能使用 this 引用。**并且Java 语法规定，**静态成员不能直接访问非静态成员。**

## this.属性名

如果方法里有个局部变量和成员变量同名，但程序又需要在该方法里访问这个被覆盖的成员变量，则必须使用 this 前缀

## this.方法名

this 关键字可以让类中一个方法，访问该类里的另一个方法或实例变量。

## this()访问构造方法

this() 用来访问本类的构造方法，括号中可以有参数，如果有参数就是调用指定的有参构造方法。

## new运算符

- 在 Java 中 new 的操作往往意味着在内存中开辟新的空间，这个内存空间分配在内存的堆区。
- 对整数或字符这样的简单变量不使用 new 运算符，是因为 Java 的简单类型不是作为对象实现的。出于效率的考虑，它们作为“常规”变量实现的。
- 内存是有限的，因此 new 有可能由于内存不足而无法给一个对象分配内存。如果出现这种情况，就会发生运行时异常。

## 堆和栈

- **堆**是用来存放由 new 创建的对象和数组，即动态申请的内存都存放在堆区。
- **栈**是用来存放在方法中定义的一些基本类型的变量和对象的引用变量。

不同方式定义字符串时堆和栈的变化：

1. String a;

只是在栈中创建了一个 String 类的对象引用变量 a。

2. String a = "C 语言中文网";

在栈中创建一个 String 类的对象引用变量 a，然后查找栈中有没有存放“C 语言中文网”，如果有则直接指向“C 语言中文网”，如果没有，则将“C 语言中文网”存放在堆中，再指向。

3. String a = new String("C 语言中文网");

不仅在栈中创建一个 String 类的对象引用变量 a，同时也在堆中开辟一块空间存放新建的 String 对象“C 语言中文网”，变量 a 指向堆中的新建的 String 对象“C 语言中文网”。

## ==符号

- ==用来比较两个对象在堆区存放的地址是否相同。
- 使用 new 运算符创建的 String 对象进行 == 操作时，两个地址是不同的。这就说明，每次对象进行 new 操作后，系统都为我们开辟堆区空间，虽然值是一样，但是地址却是不一样的。
- \== 当我们没有使用 new 运算符的时候，系统会默认将这个变量保存在内存的栈区。如果变量的值存放在栈中，使用 == 比较时，比较的是具体的值。如果变量的值存放在堆中，使用 == 比较时，比较的是值所在的地址。因此在变量 a 与变量 c 进行 == 操作的时候，返回 true，因为变量 a 和变量 c 比较的是具体的值，即“C 语言中文网”。
- 在改变变量 a 的值后（如 a = "Java"），再次输出时，我们发现输出的结果是“Java”。事实上原来的那个“C 语言中文网”在内存中并没有清除掉，而是在栈区的地址发生了改变，这次指向的是“Java”所在的地址。
- 如果需要比较两个使用 new 创建的对象具体的值，则需要通过“equal()”方法去实现，这样才是比较引用类型变量具体值的正确方式。

## 作用域修饰符

- 在 Java 语言中提供了多个作用域修饰符，其中常用的有 public、private、protected、final、abstract、static、transient 和 volatile，这些修饰符有类修饰符、变量修饰符和方法修饰符。

## 访问控制修饰符

- 访问控制符是一组限定类、属性或方法是否可以被程序里的其他部分访问和调用的修饰符
- **类的访问控制符**：空或者 public
- **方法和属性的访问控制符**：public、private、protected 和 friendly
- friendly 是一种没有定义专门的访问控制符的默认情况。

| 范围       | private | friendly | protected | public |
|----------|---------|----------|-----------|--------|
| 同一类内     | 可访问     | 可访问      | 可访问       | 可访问    |
| 同一包的其他类内 | 不可访问    | 可访问      | 可访问       | 可访问    |
| 不同包的子类   | 不可访问    | 不可访问     | 可访问       | 可访问    |
| 不同包的非子类  | 不可访问    | 不可访问     | 不可访问      | 可访问    |

## private

- 用 private 修饰的**类成员**，只能被该类自身的方法访问和修改，而不能被任何其他类（包括该类的子类）访问和引用。
- private 具有最高的保护级别。

## friendly

- 默认访问控制权规定，该类只能被同一个包中的类访问和引用，而不能被其他包中的类使用，即使其他包中有该类的子类。这种访问特性又称为包访问性（package private）。

## protected

- 用保护访问控制符 protected 修饰的类成员可以被三种类所访问：该类自身、与它在同一个包中的其他类以及在其它包中的该类的子类。
- protected 的主要作用，是允许其他包中它的子类来访问父类的特定属性和方法，否则可以使用默认访问控制符。

## public

- 被声明为 public 的类可以被其他包中的类访问
- 包中的其他类在程序中使用 import 语句引入 public 的这个类，就被可以访问和引用
- 类中被设定为 public 的方法是这个类对外的接口部分，避免了程序的其他部分直接去操作类内的数据，实际就是数据封装思想的体现。**每个 Java 程序的主类都必须是 public 类**，也是基于相同的原因。

## static 关键字

(C中文系列, Java, P391)

- static 的属性（成员变量）称为**静态变量**或类变量
- static 的常量称为**静态常量**
- static 的方法称为**静态方法**或类方法
- 静态变量、静态常量、静态方法统称为**静态成员**，归整个类所有
- 静态成员不依赖于类的特定实例，被类的所有实例共享，就是说 **static 修饰的方法或者变量不需要依赖于对象来进行访问，只要这个类被加载，Java 虚拟机就可以根据类名找到它们。**
- static 修饰的成员变量和方法，从属于类。
- 普通变量和方法从属于对象。
- 静态方法不能调用非静态成员，编译会报错。
- java中静态属性和静态方法**可以被继承，但是没有被重写(overwrite)而是被隐藏。**

## 静态变量

- 运行时，Java 虚拟机**只为静态变量分配一次内存**，在加载类的过程中完成静态变量的内存分配。
- 类的内部可以在任何方法内直接访问静态变量。
- 其他类中可以通过类名访问该类中的静态变量。
- **静态变量可以被类的所有实例共享**，因此静态变量可以作为实例之间的共享数据，增加实例之间的交互性。
- 如果类的所有实例都包含一个相同的常量属性，则可以把这个属性定义为静态常量类型，从而节省内存空间。例如，在类中定义一个静态常量 PI。
- 在类中定义静态的属性（成员变量），在 main() 方法中可以直接访问，也可以通过类名访问，还可以通过类的实例对象来访问。
- 静态变量是被多个实例所共享的。

## 静态方法

- 静态方法不需要通过它所属的类的任何实例就可以被调用，因此**静态方法中不能使用 this 关键字，也不能直接访问所属类的实例变量和实例方法，但是可以直接访问所属类的静态变量和静态方法**
- 和 this 关键字一样，super 关键字也与类的特定实例相关，所以在静态方法中也不能使用 super 关键字。
- 在访问静态方法时，可以直接访问，也可以通过类名来访问，还可以通过实例化对象来访问。

## 静态代码块

- **{ } 代码块为非静态代码块**，非静态代码块是在创建对象时自动执行的代码，不创建对象不执行该类的非静态代码块。
- **静态代码块指 Java 类中的 static{ } 代码块**，主要用于初始化类，为类的静态变量赋初始值，提升程序性能。
- 静态代码块类似于一个方法，但它不可以存在于任何方法体中。
- 静态代码块可以置于类中的任何地方，类中可以有多个静态初始化块。
- Java 虚拟机在加载类时执行静态代码块，所以很多时候会将一些只需要进行一次的初始化操作都放在 static 代码块中进行。
- 如果类中包含多个静态代码块，则 Java 虚拟机将按它们在类中出现的顺序依次执行它们，**每个静态代码块只会被执行一次**。
- 静态代码块与静态方法一样，不能直接访问类的实例变量和实例方法，而需要通过类的实例对象来访问

## 静态导入 (import static)

(C中文系列, Java, P396)

- 在 **JDK 1.5** 之后增加了一种静态导入的语法，用于**导入指定类的某个静态成员变量、方法或全部的静态成员变量、方法**。
- 如果一个类中的方法**全部是使用 static 声明的静态方法**，则在导入时就可以直接使用 **import static** 的方式导入。

## 常见问题

(C中文系列, Java, P397)

- static 关键字一般运用在以下两个场景：
  - 1.只想为特定域分配单一存储空间，不考虑要创建多少对象或者说根本就不创建任何对象
  - 2.想在没有创建对象的情况下也想调用方法

- “static”关键字表明一个成员变量或者是成员方法可以在没有所属的类的实例变量的情况下被访问。
- Java 中 **static 方法不能被覆盖**，因为方法覆盖是基于运行时动态绑定的，而 static 方法是编译时静态绑定的，static 方法跟类的任何实例都不相关
- **static 静态方法不能引用非静态资源**，因为new 的时候才会产生的东西，对于初始化后就存在的静态资源来说，不能引用它。
- **static 静态方法里面能引用静态资源**，因为静态资源都是类初始化的时候加载的
- 非静态方法里面能引用静态资源，因为非静态方法就是实例方法，那是 new 之后才产生的，那么属于类的内容它都认识

## 使用误区

- **static 关键字不会改变类中成员的访问权限**。与 C/C++ 中的 static 不同，Java 中的 static 关键字不会影响到变量或者方法的作用域，能够影响访问权限的只有 private、public、protected、friendly。
- **static 关键字并不会改变变量和方法的访问权限**。静态成员变量虽然 独立于对象，但是不代表不可以通过对对象去访问，所有的静态方法和静态变量都可以通过对对象访问（只要访问权限足够）。
- **能通过 this 访问静态成员变量**。静态成员变量虽然独立于对象，但是不代表不可以通过对对象去访问，所有的静态方法和静态变量都可以通过对对象访问（只要访问权限足够）。
- **Java 语法规则 static 是不允许用来修饰局部变量**。

## final 修饰符

(C中文系列, Java, P399)

final 在 Java 中的意思是最终，也可以称为完结器，表示对象是最终形态的，不可改变的意思。

## 修饰变量

- final 用在变量的前面表示变量的值不可以改变，此时该变量可以被称为常量。
- final 修饰的变量不能被赋值这种说法是错误的，严格的说法是，**final 修饰的变量不可被改变**，获得了初始值后，**final 变量的值就不能被重新赋值**。
- **final修饰的类不可以有子类**。

### final 局部变量和成员变量区别

- final 修饰的**局部变量**必须使用之前被赋值一次才能使用。
- final 修饰的**成员变量**在声明时没有赋值的叫“空白 final 变量”。空白 final 变量必须在构造方法或静态代码块中初始化。

### final 基本类型变量和引用类型变量区别

- final 修饰**基本类型变量**时，不能对基本类型变量重新赋值，因此**基本类型变量不能被改变**。
- final 修饰**引用类型变量**时，它保存的仅仅是一个引用，final 只保证这个引用类型变量所引用的地址不会改变，即一直引用同一个对象，但这个对象完全可以发生改变。

```
// final 修饰数组变量，iArr 是一个引用变量
final int[] iArr = { 5, 6, 12, 9 };
System.out.println(Arrays.toString(iArr));
// 对数组元素进行排序，合法
Arrays.sort(iArr);
System.out.println(Arrays.toString(iArr));
// 对数组元素赋值，合法
iArr[2] = -8;
```

```

System.out.println(Arrays.toString(iArr));
// 下面语句对 iArr 重新赋值,非法
// iArr = null;

// final 修饰 Person 变量, p 是一个引用变量
final Person p = new Person(45);
// 改变 Person 对象的 age 实例变量, 合法
p.setAge(23);
System.out.println(p.getAge());
// 下面语句对 P 重新赋值, 非法
// p = null;

```

## 修饰方法

- **final 修饰的方法不可被重写。**如果出于某些原因,不希望子类重写父类的某个方法,则可以使用 final 修饰该方法。
- **final 修饰的方法能被重载。**对于一个 private 方法,因为它仅在当前类中可见,其子类无法访问该方法,所以子类无法重写该方法——如果子类中定义一个与父类 private 方法有相同方法名、相同形参列表、相同返回值类型的方法,也不是方法重写,只是重新定义了一个新方法。因此,即使使用 final 修饰一个 private 访问权限的方法,依然可以在其子类中定义与该方法具有相同方法名、相同形参列表、相同返回值类型的方法。

```

public class PrivateFinalMethodTest {
    private final void test() {
    }
    public final void test() {
    }
}
class Sub extends PrivateFinalMethodTest {
    // 下面方法定义将出现编译错误,不能重写 final 方法
    public void test() {
    }
    // 下面的方法定义不会出现问题
    public void test() {
    }
}

```

## 修饰类

**final 修饰的类不能被继承。**当子类继承父类时,将可以访问到父类内部数据,并可通过重写父类方法来改变父类方法的实现细节,这可能导致一些不安全的因素。为了保证某个类不可被继承,则可以使用 final 修饰这个类。

## abstract 修饰符

### 抽象类

- **abstract 修饰的类称为是抽象类。**所谓抽象类就是没有具体实例对象的类。
- 如果一个类中含有抽象方法,那么这个类必须被声明为抽象类。
- 抽象类是没有具体实例对象的类,不能用 new 方法进行实例化。

## 抽象方法

- **abstract修饰的方法**称为是抽象方法。所谓抽象方法就是没有方法体的方法。
- **包含抽象方法的类一定是抽象类**
- **抽象类不一定包含抽象方法**（抽象类中可以包含非抽象方法）
- **abstract 关键字只能用于普通方法，不能用于 static 方法或者构造方法中。**
- **在使用 abstract 关键字修饰抽象方法时不能使用 private 修饰**，因为抽象方法必须被子类重写，而如果使用了 private 声明，则 子类是无法重写的。

## 继承和多态

---

### 封装

- 封装将类的某些信息隐藏在类内部，不允许外部程序直接访问，只能通过该类提供的方法来实现对隐藏信息的操作和访问。
- 通过封装，实现了对属性的数据访问限制，满足了年龄的条件。在属性的赋值方法中可以对属性进行限制操作，从而给类中的属性赋予 合理的值， 并通过取值方法获取类中属性的值（也可以直接调用类中的属性名称来获取属性值）。

### 继承

关键字：extends

继承是面向对象的三大特征之一。

Java不支持多继承性，一个类只能由一个父类。

已经存在的类称为父类、基类或超类，而新产生的类称为子类或 派生类。

语法：修饰符 class class\_name **extends** extend\_class {//类的主体}。extends 关键字直接跟在子类名之后，其后 面是该类要继承的父类名称。

## 接口

---

（C中文系列，Java，P460）

- 关键字：interface（定义接口）、implements（使用接口）
- 接口是Java为了克服单继承的缺点而产生的。
- 接口是抽象方法和常量值的定义的集合。
- 接口是一种特殊的抽象类，只包含常量和方法的声明，而没有变量和方法的实现。
- **一个接口可以有多个直接父接口，但接口只能继承接口，不能继承类。**
- 接口的思想在于它可以增加很多类都需要实现的功能，使用相同的接口类不一定有继承关系。
- 若接口被继承，则实现子接口时必须实现子、父接口中的全部方法。

### 定义接口

- 关键字：interface（定义接口）
- 接口若无public修饰符，则其访问将局限于所属的包，若有public，则允许任何类使用。
- **一个接口不能够实现另一个接口**，但它可以继承多个其他接口。子接口可以对父接口的方法和常量进行重写。



## 方法定义

- 接口没有构造方法
- 接口内方法的定义无需其他修饰符（在接口中声明的方法将被隐式声明成public、abstract，共有抽象的方法）

## 变量（常量）定义

- 接口中的变量都是常量（在接口中声明的变量将被隐式声明为public、static和final，即常量）

```
public interface MyInterface //接口声明
{
    //String name; //不合法，变量name必须初始化
    int age = 20; //public static final int age = 20;
    void getInfo(); //public abstract void getInfo();
}
```

## 实现接口

- 关键字：implements（使用接口）、extends（继承）
- 接口的主要用途就是被实现类实现
- 一个类可以实现一个或多个接口，多个接口之间以逗号分隔。
- **一个类可以继承一个父类**，并同时实现多个接口，implements 部分必须放在 extends 部分之后。
- 一个类实现一个或多个接口后，需要完成（重写）这些接口中定义的所有抽象方法，否则该类将保留从父接口那里继承到的抽象方法，需要继续定义成抽象类。（接口中的方法只能是抽象方法）

```
public interface IMath
{
    public int sum();
    public int maxNum(int a,int b);
}

public class MathClass implements IMath
{
    private int num1;
    private int num2;
    public MathClass(int a,int b)
    {
        this.num1=a;
        this.num2=b;
    }
    public int sum()
    {
        return num1+num2;
    }
    public int maxNum(int a,int b)
    {
        if (a>=b) return a;
        else return b;
    }
}
```

# 接口与抽象类

(C中文系列, Java, P464)

## 相同点

- 都不能直接创建他们的对象（即都不能被实例化）
- 都能定义抽象方法，这些抽象方法用于描述系统能提供哪些操作，而不提供具体的实现。
- 实现子类都必须实现这些抽象方法

## 不同点

- 接口作为系统与外界交互的窗口。
- 在一个程序中使用接口时，接口是多个模块间的耦合标准；在多个应用程序之间使用接口时，接口是多个程序之间的通信标准。
- 接口类似于整个系统的“总纲”，它制定了系统各模块应该遵循的标准，因此一个系统中的接口不应该经常改变。
- 接口体现的是一种规范、模板式设计。
- 接口是一种特殊的抽象类。
- 在抽象类中可以为部分方法提供默认的实现，从而避免在子类中重复实现它们，提高代码的可重用性，这是抽象类的优势；而接口中只能包含抽象方法。
- 一个类只能继承一个直接的父类，这个父类有可能是抽象类  
但一个类可以实现多个接口，这是接口的优势。

(其余具体请参考pdf)

# 异常处理

## 异常介绍

- Java 中的异常又称为例外，是一个在程序执行期间发生的事件，它中断正在执行程序的正常指令流。
- 在一个方法的运行过程中，如果发生了异常，则这个方法会产生代表该异常的一个对象，并把它交给运行时的系统，运行时系统寻找相应的代码来处理这一异常。
- 把生成异常对象，并把它提交给运行时系统的过程称为抛出 (throw) 异常。
- 运行时系统在方法的调用栈中查找，直到找到能够处理该类型异常的对象，这一个过程称为捕获 (catch) 异常。

## 异常类型

- 为了能够及时有效地处理程序中的运行错误，Java 专门引入了异常类。
- Java 中所有异常类型都是内置类 java.lang.Throwable 类的子类，即 Throwable 位于异常类层次结构的顶层。
- 在 Java 代码中只有继承了 Throwable 类的实例才能被 throw 或者 catch。
- **Throwable 类是所有异常和错误的超类，下面有 Error 和 Exception 两个子类分别表示错误和异常。**
- **Exception 类用于用户程序可能出现的异常情况，它也是用来创建自定义异常类型类的类。**
- **Error 定义了通常在通常环境下不希望被程序捕获的异常。一般指的是 JVM 错误，如堆栈溢出。**

## Exception

(C中文系列, Java, P498、P499)

- Exception 是程序正常运行过程中可以预料到的意外情况, 并且 应该被开发者捕获, 进行相应的处理。
- Exception 可分为运行时异常 (不检查异常) 和非运行时异常 (检查异常)。
- **运行时异常**都是 RuntimeException 类及其子类异常, 如 NullPointerException、IndexOutOfBoundsException 等, 这些异常是不 检查异常, 程序中可以选择捕获处理, 也可以不处理。这些异常**一般由程序逻辑错误引起, 程序应该从逻辑角度尽可能避免这类异常的发生。**
- **非运行时异常**是指 RuntimeException 以外的异常, 类型上都属于 Exception 类及其子类。**从程序语法角度讲是必须进行处理的异常, 如果不处理, 程序就不能编译通过。**如 IOException、ClassNotFoundException 等以及用户自定义的 Exception 异常 (一般情况下 不自定义检查异常)。

## Error

(C中文系列, Java, P499)

- Error 是指正常情况下不大可能出现的情况, 绝大部分的 Error 都会导致程序处于非正常、不可恢复状态。所以不需要被开发者捕获。
- Error 错误是任何处理技术都无法恢复的情况, 肯定会导致程序非正常终止。并且 Error 错误属于未检查类型, 大多数发生在运行时。

(常见具体的Exception和Error, 见pdf)

## 异常处理机制

- Java 的异常处理通过 5 个关键字来实现: **try**、**catch**、**throw**、**throws** 和 **finally**。
- try catch 语句用于捕获并处理异常
- finally 语句 用于在任何情况下 (除特殊情况外) 都必须执行的代码
- throw 语句用于抛出异常
- throws 语句用于声明可能会出现的异常。

## 异常处理程序

异常处理程序的基本结构如下:

```
try
{
    逻辑程序块
}
catch (ExceptionType1 e)
{
    处理代码块 1
}
catch (ExceptionType2 e)
{
    处理代码块 2
    throw(e); // 再抛出这个"异常"
}
finally
{
    释放资源代码块
}
```

## try catch

(C中文系列, Java, P502)

- 用 try catch 语句捕获并处理异常，catch 语句可以有多个，用来匹配多个异常。
- 对于处理不了的异常或者要转型的异常，在方法的声明处通过 throws 语句抛出异常，即由上层的调用方法来处理。
- 语法格式：

```
try
{
    // 可能发生异常的语句
}
catch(ExceptionType e)
{
    // 处理异常语句
}
```

执行情况：如果 try 语句块中发生异常，那么一个相应的异常对象就会被抛出，然后 catch 语句就会依据所抛出异常对象的类型进行捕获，并处理。处理之后，程序会跳过 try 语句块中剩余的语句，转到 catch 语句块后面的第一条语句开始执行。

- try 和 catch 后面的花括号{ }不可以省略
- try 块里声明的变量只是代码块内的局部变量，它只在 try 块内有效，其它地方不能访问该变量。
- 处理异常块中，可以使用以下方法输出相应的异常信息：

**printStackTrace()** 方法：指出异常的类型、性质、栈层次及出现在程序中的位置

**getMessage()** 方法：输出错误的性质。

**toString()** 方法：给出异常的类型与性质

```
import java.util.Scanner;
public class Test
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);

        try
        {
            String name = scan.next();
            int age = scan.nextInt();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            System.out.println("worry!");
        }
    }
}
```

## 多重catch

(C中文系列, Java, P504)

- 如果 try 代码块中有很多语句会发生异常，而且发生的异常种类又很多。那么可以在 try 后面跟有多个 catch 代码块。
- 在多个 catch 代码块的情况下，当一个 catch 代码块捕获到一个异常时，其它的 catch 代码块就不再匹配。
- 当捕获的多个异常类之间存在父子关系时，捕获异常时一般先捕获子类，再捕获父类。所以子类异常必须在父类异常的前面，否则子类捕获不到。

## try catch finally

(C中文系列, Java, P508)

- 根据 try catch 语句的执行过程，try 语句块和 catch 语句块有可能不被完全执行。
- 而无论是否发生异常（除特殊情况外），finally 语句块中的代码都会被执行。
- 使用 try-catch-finally 语句时需注意以下几点：
  - 1、异常处理语法结构中只有 **try 块是必需的**，如果没有 try 块，则不能有 catch 块和 finally 块；
  - 2、**catch 块和 finally 块至少出现其中之一**；
  - 3、可以有多个 catch 块，**捕获父类异常的 catch 块必须位于捕获子类异常的后面**；
  - 4、**不能只有 try 块**，既没有 catch 块，也没有 finally 块；
  - 5、多个 catch 块必须位于 try 块之后
  - 6、finally 块必须位于所有的 catch 块之后。
  - 7、finally 与 try 语句块匹配的语法格式，此种情况会导致异常丢失，所以不常见。
- 运行情况：
  - 1、如果 try 代码块中没有抛出异常，则执行完 try 代码块之后直接执行 finally 代码块，然后执行 try catch finally 语句块之后的语句。
  - 2、如果 try 代码块中抛出异常，并被 catch 子句捕捉，那么在抛出异常的地方终止 try 代码块的执行，转而执行相匹配的 catch 代码块，之后执行 finally 代码块。如果 finally 代码块中没有抛出异常，则继续执行 try catch finally 语句块之后的语句；如果 finally 代码块中抛出异常，则把该异常传递给该方法的调用者。
  - 3、如果 try 代码块中抛出的异常没有被任何 catch 子句捕捉到，那么将直接执行 finally 代码块中的语句，并把该异常传递给该方法的调用者。

## finally和return执行顺序

(C中文系列, Java, P512)

### try + return

```
public class tryDemo
{
    public static int show()
    {
        try
        {
            return 1;
        }
        finally
        {
        }
```

```

        System.out.println("finally");
    }
}

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

/*
finally
1
*/

```

### try、catch + return

try 代码块或者 catch 代码块中有 return 时，finally 中的代码总会被执行，且 finally 语句在 return 返回之前执行

```

public class tryDemo
{
    public static int show()
    {
        try
        {
            int a = 8/0;
            return 1;
        }
        catch (Exception e)
        {
            return 2;
        }
        finally
        {
            System.out.println("finally");
        }
    }

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

/*
finally
2
*/

```

## finally + return

当 finally 有返回值时，会直接返回该值，不会去返回 try 代码块或者 catch 代码块中的返回值。

注意：finally 代码块中最好不要包含 return 语句，否则程序会提前退出。

```
public class tryDemo
{
    public static int show()
    {
        try
        {
            int a = 8/0;
            return 1;
        }
        catch (Exception e)
        {
            return 2;
        }
        finally
        {
            System.out.println("finally");
            return 0;
        }
    }

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

/*
finally
0
*/
```

## finally中改变返回值

在 finally 代码块中改变返回值并不会改变最后返回的内容。当返回值类型是引用类型时，结果也是一样。

```
public class tryDemo
{
    public static int show()
    {
        int result = 0;
        try
        {
            return result;
        }
        finally
        {
            System.out.println("finally");
        }
    }
}
```

```
        result = 1;
    }
}

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

}

/*
finally
0
*/
```

## 总结

try 代码块和 catch 代码块中有 return 语句时，finally 仍然会被执行。

执行 try 代码块或 catch 代码块中的 return 语句之前，都会先执行 finally 语句。

无论在 finally 代码块中是否修改返回值，返回值都不会改变，仍然是执行 finally 代码块之前的值。

finally 代码块中的 return 语句一定会执行。

## 自动资源管理

## 行为

---

## 注释

---



