# 目　　录

# 一 Java 语言概述

## 1.1 软件开发介绍

软件开发软件，即一系列按照特定顺序组织的计算机数据和指令的集合。
软件有系统软件和应用软件之分。

人机交互方式
图形化界面 (Graphical User Interface GUI) 这种方式简单直观，使用者易于接受，容易上手操作。

命令行方式 (Command Line InterfaceCLI)：需要有一个控制台，输入特定的指令，让计算机完成一些操作。较为麻烦，需要记录住一些命令。

### 常用的 DOS 命令

Win+R，一起按下，输入 cmd，可以打开 dos 界面。
dir : 列出当前目录下的文件以及文件夹
md: 创建目录
rd : 删除目录
cd: 进入指定目录
cd… : 退回到上一级目录
cd: 退回到根目录
del : 删除文件
exit : 退出 dos 命令行
补充：echo javase>1.doc
常用快捷键
← →：移动光标
↑ ↓：调阅历史操作命令
Delete 和 Backspace：删除字符
注意：在输入 dos 命令时，要是用英文输入，所有标点符号都是英文。

## 1.2 计算机编程语言介绍

什么是计算机语言
语言：是人与人之间用于沟通的一种方式。
例如：中国人与中国人用普通话沟通。而中国人要和英国人交流，就要学习英语。
计算机语言：人与计算机交流的方式。
如果人要与计算机交流，那么就要学习计算机语言。计算机语言有很多种。
如：C ,C++,Java,PHP,Kotlin，Python，Scala 等。
第一代语言
机器语言：指令以二进制代码形式存在。
第二代语言
汇编语言：使用助记符表示一条机器指令。
第三代语言：高级语言
C、Pascal、Fortran 面向过程的语言
C++ 面向过程 / 面向对象
Java 跨平台的纯面向对象的语言
.NET 跨语言的平台
Python、Scala…

面向过程：例如张三打篮球，他打篮球的全部过程 ( 拿球、传球、投篮……)。
面向对象：人的对象，人的运动的动作，运动的器械这三个对象，实例化一个张三的对象，对象有一个打篮球的动作，器械是篮球。
面向对象能更好的在抽象的层面分析问题，在程序实现跨越极大的赋予之前的代码。这些是面向过程编程极难实现的。

## 1.3 Java 语言概述

是 SUN(Stanford University Network，斯坦福大学网络公司 ) 1995 年推出的一门高级编程语言。

是一种面向 Internet 的编程语言。Java 一开始富有吸引力是因为 Java 程序可以在 Web 浏览器中运行。这些 Java 程序被称为 Java 小程序（applet）。applet 使用现代的图形用户界面与 Web 用户进行交互。applet 内嵌在 HTML 代码中。

随着 Java 技术在 web 方面的不断成熟，已经成为 Web 应用程序的首选开发语言。后台开发：Java、PHP、Python、Go、Node.js



### 1.3.1 Java 简史

1991 年 Green 项目，开发语言最初命名为 Oak ( 橡树 )
1994 年开发组意识到 Oak 非常适合于互联网
1996 年发布 JDK 1.0，约 8.3 万个网页应用 Java 技术来制作
1997 年发布 JDK 1.1，JavaOne 会议召开，创当时全球同类会议规模之最
1998 年发布 JDK 1.2，同年发布企业平台 J2EE
1999 年 Java 分成 J2SE、J2EE 和 J2ME，JSP/Servlet 技术诞生
2004 年发布里程碑式版本：JDK 1.5，为突出此版本的重要性，更名为 JDK 5.0
2005 年 J2SE -> JavaSE，J2EE -> JavaEE，J2ME -> JavaME
2009 年 Oracle 公司收购 SUN，交易价格 74 亿美元
2011 年发布 JDK 7.0
2014 年发布 JDK 8.0，是继 JDK 5.0 以来变化最大的版本
2017 年发布 JDK 9.0，最大限度实现模块化
2018 年 3 月发布 JDK 10.0，版本号也称为 18.3
2018 年 9 月发布 JDK 11.0，版本号也称为 18.9
2019 年 3 月 20 日 Java SE 12 发布。Java 12 是短期支持版本。
2019 年 9 月 23 日 Java SE 13 发布，此版本中添加了“ 文本块 ”，文本块是一个多行字符串文字，避免对大多数转义序列的需要，以可预测的方式自动格式化字符串，并在需要时让开发人员控制格式。

### 1.3.2 Java 技术体系平台

#### 1、JavaSE(Java Standard Edition) 标准版

支持面向桌面级应用（如 Windows 下的应用程序）的 Java 平台，提供了完整的 Java 核心 API，此版本以前称为 J2SE

#### 2、JavaEE(Java Enterprise Edition) 企业版

是为开发企业环境下的应用程序提供的一套解决方案。该技术体系中包含的技术如 :Servlet 、Jsp 等，主要针对于 Web 应用程序开发。版本以前称为 J2EE

#### 3、Java ME(Java Micro Edition) 小型版

支持 Java 程序运行在移动终端（手机、PDA）上的平台，对 Java API 有所精简，并加入了针对移动终端的支持，此版本以前称为 J2ME。

#### 4、Java Card

支持一些 Java 小程序（Applets）运行在小内存设备（如智能卡）上的平台。

## 5、从 Java 的应用领域来分，Java 语言的应用方向主要表现在以下几个方面：

企业级应用：主要指复杂的大企业的软件系统、各种类型的网站。Java 的安全机制以及它的跨平台的优势，使它在分布式系统领域开发中有广泛应用。应用领域包括金融、电信、交通、电子商务等。

Android 平台应用：Android 应用程序使用 Java 语言编写。Android 开发水平的高低很大程度上取决于 Java 语言核心能力是否扎实。
大数据平台开发：各类框架有 Hadoop，spark，storm，flink 等，就这类技术生态圈来讲，还有各种中间件如 flume，kafka，sqoop 等等，这些框架以及工具大多数是用 Java 编写而成，但提供诸如 Java，scala，Python，R 等各种语言 API 供编程。

移动领域应用：主要表现在消费和嵌入式领域，是指在各种小型设备上的应用，包括手机、PDA、机顶盒、汽车通信设备等。

## Java 主要特性

Java 语言是易学的。Java 语言的语法与 C 语言和 C++ 语言很接近，使得大多数程序员很容易学习和使用 Java。

Java 语言是强制面向对象的。Java 语言提供类、接口和继承等原语，为了简单起见，只支持类之间的单继承，但支持接口之间的多继承，并支持类与接口之间的实现机制（关键字为 implements）。

Java 语言是分布式的。Java 语言支持 Internet 应用的开发，在基本的 Java 应用编程接口中有一个网络应用编程接口（java net），它提供了用于网络应用编程的类库，包括 URL、URLConnection、Socket、ServerSocket 等。Java 的 RMI（远程方法激活）机制也是开发分布式应用的重要手段。

Java 语言是健壮的。Java 的强类型机制、异常处理、垃圾的自动收集等是 Java 程序健壮性的重要保证。对指针的丢弃是 Java 的明智选择。

Java 语言是安全的。Java 通常被用在网络环境中，为此，Java 提供了一个安全机制以防恶意代码的攻击。如：安全防范机制（类 ClassLoader），如分配不同的名字空间以防替代本地的同名类、字节代码检查。



Java 语言是体系结构中立的。Java 程序（后缀为 java 的文件）在 Java 平台上被编译为体系结构中立的字节码格式（后缀为 class 的文件），然后可以在实现这个 Java 平台的任何系统中运行。

Java 语言是解释型的。如前所述，Java 程序在 Java 平台上被编译为字节码格式，然后可以在实现这个 Java 平台的任何系统的解释器中运行。先编译后解释。

Java 是性能略高的。与那些解释型的高级脚本语言相比，Java 的性能还是较优的。

Java 语言是原生支持多线程的。在 Java 语言中，线程是一种特殊的对象，它必须由 Thread 类或其子（孙）类来创建。

# 1.4 Java 程序运行机制及运行过程

特点一：面向对象
两个基本概念：类、对象
三大特性：封装、继承、多态

特点二：健壮性
吸收了 C/C++ 语言的优点，但去掉了其影响程序健壮性的部分（如指针、内存的申请与释放等），提供了一个相对安全的内存管理和访问机制。

特点三：跨平台性
跨平台性：通过 Java 语言编写的应用程序在不同的系统平台上都可以运行。
“ Write once , Run Anywhere ”
原理：只要在需要运行 java 应用程序的操作系统上，先安装一个 Java 虚拟机 (JVM Java Virtual Machine) 即可。由 JVM 来负责 Java 程序在该系统中的运行。

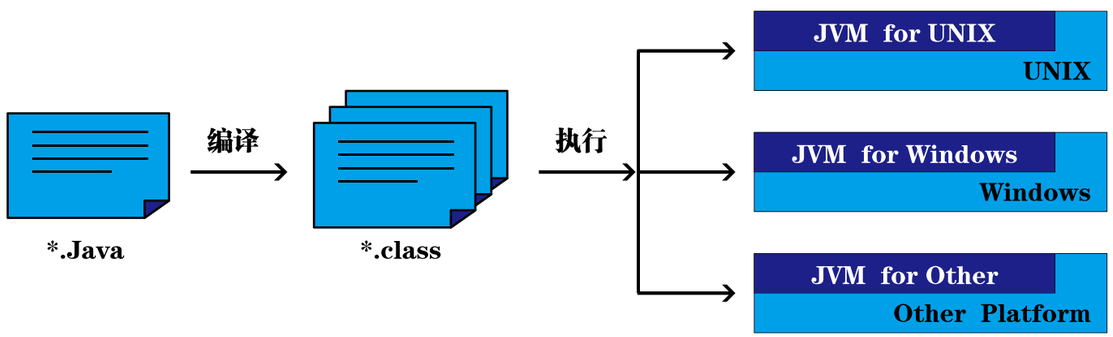
# Java 两种核心机制

## 1、Java 虚拟机 (Java VirtalMachine)

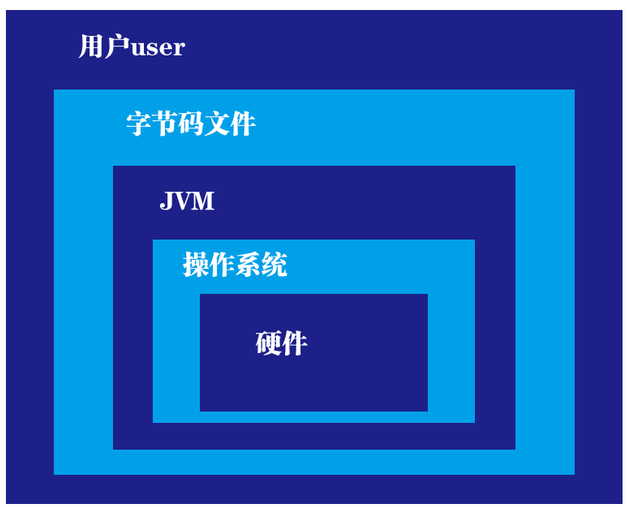
JVM 是一个虚拟的计算机，具有指令集并使用不同的存储区域。负责执行指令，管理数据、内存、寄存器。

对于不同的平台，有不同的虚拟机。

只有某平台提供了对应的 java 虚拟机，java 程序才可在此平台运行。




**Java 虚拟机机制屏蔽了底层运行平台的差别，实现了“ 一次编译，到处运行 ”。**






## 2、垃圾收集机制 (Garbage Collection)

不再使用的内存空间应回收——垃圾回收。

在 C/C++ 等语言中，由程序员负责回收无用内存。

Java 语言消除了程序员回收无用内存空间的责任：它提供一种系统级线程跟踪存储空间的分配情况。并在 JVM 空闲时，检查并释放那些可被释放的存储空间。

垃圾回收在 Java 程序运行过程中自动进行，程序员无法精确控制和干预。

Java 程序还会出现内存泄漏和内存溢出问题吗？ Yes!

# 1.5 Java 语言的环境搭建

## 1、明确什么是 JDK, JRE

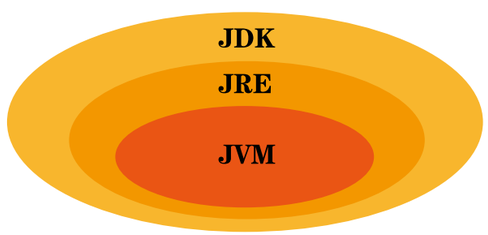
JDK(Java Development Kit Java 开发工具包 )

JDK 是提供给 Java 开发人员使用的，其中包含了 java 的开发工具，也 ** 包括了 JRE。** 所以安装了 JDK，就不用在单独安装 JRE 了。其中的开发工具：编译工具 (javac.exe) 打包工具 (jar.exe) 等。

JRE(Java Runtime Environment Java 运行环境 )
包括 Java 虚拟机 (JVM Java Virtual Machine) 和 Java 程序所需的核心类库等，如果想要运行一个开发好的 Java 程序，计算机中只需要安装 JRE 即可。

## 2、简单而言，使用 JDK 的开发工具完成的 java 程序，交给 JRE 去运行

JDK = JRE + 开发工具集（例如 Javac 编译工具等）
JRE = JVM + Java SE 标准类库



# 1.6 开发体验—HelloWorld

## 1、步骤：

将 Java 代码编写到扩展名为 .java 的文件中。

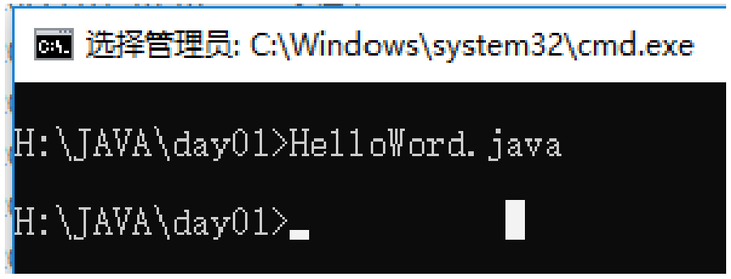
选择最简单的编辑器：记事本。

敲入代码 class Test{} 将文件保存成 Test.java，这个文件是存放 java 代码的文件，称为源文件。

## 2、第一个 Java 程序

```
public class Test {
        public static void main(String[] args) {
                System.out.println("hello world");
        }
}
```

## 3、通过 javac 命令对该 java 文件进行编译。




有了 java 源文件，通过编译器将其编译成 JVM 可以识别的字节码文件。

在该源文件目录下，通过 javac 编译工具对 Test.java 文件进行编译。

如果程序没有错误，没有任何提示，但在当前目录下会出现一个 Test.class 文件，该文件称为字节码文件，也是可以执行的 java 的程序。

## 4、通过 java 命令对生成的 class 文件进行运行。

有了可执行的 java 程序 (Test.class 字节码文件 )
通过运行工具 java.exe 对字节码文件进行执行。
出现提示：缺少一个名称为 main 的方法。

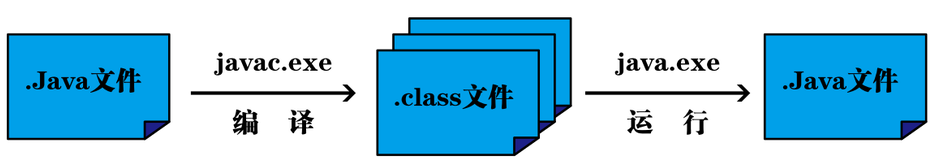


因为一个程序的执行需要一个起始点或者入口，所以在 Test 类中的加入 public static void main(String[] args){}

对修改后的 Test.java 源文件需要重新编译，生成新的 class 文件后，再进行执行。

发现没有编译失败，但也没有任何效果，因为并没有告诉 JVM 要帮我们做什么事情，也就是没有可以具体执行的语句。

想要和 JVM 来个互动，只要在 main 方法中加入一句 System.out.println( “ Hello World"); 因为程序进行改动，所以再重新编译，运行即可。




# 1.7 常见问题及解决方法

## 1、问题 1

源文件名不存在或者写错

当前路径错误

后缀名隐藏问题

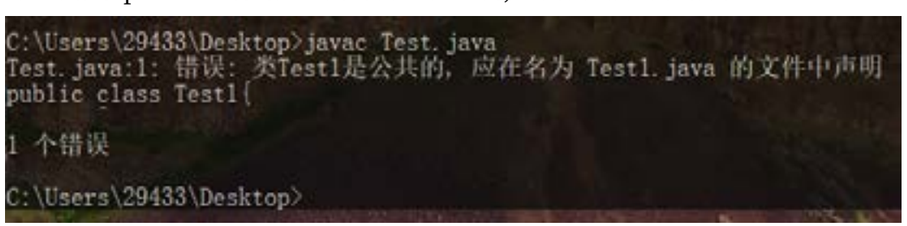
## 2、问题 2

类文件名写错，尤其文件名与类名不一致时，要小心

类文件不在当前路径下，或者不在 classpath 指定路径下


## 3、问题 3

声明为 public 的类应与文件名一致，否则编译失败




## 4、问题 4

编译失败，注意错误出现的行数，再到源代码中指定位置改错


## 总结：

学习编程最容易犯的错是语法错误。Java 要求你必须按照语法规则编写代码。如果你的程序违反了语法规则，例如：忘记了分号、大括号、引号，或者拼错了单词，java 编译器都会报语法错误。尝试着去看懂编译器会报告的错误信息。



# 1.8 注释 (comment)

用于注解说明解释程序的文字就是注释。

Java 中的注释类型：

单行注释

格式：// 注释文字
多行注释

格式：/* 注释文字 /*
注：对于单行和多行注释，被注释的文字，不会被 JVM（java 虚拟机）解释执行。
多行注释里面不允许有多行注释嵌
文档注释 (java 特有 )

格式：

```
/**
 * @author  指定 java 程序的作者 **
 * @version  指定源文件的版本 **
 */
```

提高了代码的阅读性；调试程序的重要方法。

注释是一个程序员必须要具有的良好编程习惯。

将自己的思想通过注释先整理出来，再用代码去体现。

## 1.9 小结第一个程序

Java 源文件以“ java ”为扩展名。源文件的基本组成部分是类（class），如本例中的 HelloWorld 类。

Java 应用程序的执行入口是 main() 方法。它有固定的书写格式：
public static void main(String[] args) {…}

Java 语言严格区分大小写。

Java 方法由一条条语句构成，每个语句以“ ; ”结束。

大括号都是成对出现的，缺一不可。

一个源文件中最多只能有一个 public 类。

其它类的个数不限，如果源文件包含一个 public 类，则文件名必须按该类名命名。



# 二　基本语法

## 2.1 关键字与保留字

### 1、关键字 (keyword) 的定义和特点

定义：被 Java 语言赋予了特殊含义，用做专门用途的字符串（单词）
特点：关键字中所有字母都为小写。

| 用于定义数据类型的关键字 | | | | |
|---|---|---|---|---|
| class | interface | enum | tybe | short |
| int | long | float | double | char |
| boolean | void | | | |
| 用于定义流程控制的关键字 | | | | |
| if | else | switch | case | default |
| while | do | for | break | continue |
| return | | | | |
| 用于定义访问权限修饰符的关键字 | | | | |
| private | protectde | public | | |
| 用于定义类，函数，变量修饰符的关键字 | | | | |
| abstract | final | static | synchronized | |
| 用于定义类于类之间关系的关键字 | | | | |
| new | this | super | instanceof | |
| 用于异常处理的关键字 | | | | |
| try | catch | finally | throw | throws |
| 用于包的关键字 | | | | |
| package | import | | | |
| 其他修饰符关键字 | | | | |
| native | strictfp | transient | volatile | assert |
| 用于定义数据类型值的字面值 | | | | |
| TRUE | FALSE | null | | |

## 2、保留字 (reserved word)

Java 保留字：现有 Java 版本尚未使用，但以后版本可能会作为关键字使用。自己命名标识符时要避免使用这些保留字 goto、const。

# 2.2 标识符

## 1、什么是标识符（Identifier）

Java 对各种变量、方法和类等要素命名时使用的字符序列称为标识符
技巧：凡是自己可以起名字的地方都叫标识符。

## 2、定义合法标识符规则【重要】

1. 由 26 个英文字母大小写，0-9，_ 或 $ 组成
2. 数字不可以开头。
3. 标识符不能包含空格。
4. 不可以使用关键字和保留字，但能包含关键字和保留字。
5. Java 中严格区分大小写，长度无限制。

## 3、Java 中的名称命名规范

1、Java 中的名称命名规范：
包名：多单词组成时所有字母都小写：xxxyyyzzz
类名、接口名：多单词组成时，** 所有单词的首字母大写：**XxxYyyZzz
变量名、方法名：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写：xxxYyyZzz
常量名：所有字母都大写。多单词时每个单词用下划线连接：XXX_YYY_ZZZ

2、注意点
注意 1：在起名字时，为了提高阅读性，要尽量有意义，“ 见名知意 ”。
注意 2：java 采用 unicode 字符集，因此标识符也可以使用汉字声明，但是不建议使用。



# 2.3 变　量

## 2.3.1 变量的声明与使用

## 1、变量的概念：

内存中的一个存储区域；
该区域的数据可以在同一类型范围内不断变化；
变量是程序中最基本的存储单元。包含变量类型、变量名和存储的值。

## 2、变量的作用：

用于在内存中保存数据。

## 3、使用变量注意：

Java 中每个变量必须先声明，后使用；
使用变量名来访问这块区域的数据；
变量的作用域：其定义所在的一对 { } 内；
变量只有在其作用域内才有效；
同一个作用域内，不能定义重名的变量；

## 4、声明变量

语法：< 数据类型 > < 变量名称 >
例如：int var;

## 5、变量的赋值

语法：< 变量名称 > = < 值 >
例如：var = 10;

## 6、声明和赋值变量

语法：< 数据类型 >< 变量名 >= < 初始化值 >
例如：int var = 10

## 7、补充：变量的分类 - 按声明的位置的不同

在方法体外，类体内声明的变量称为成员变量。
在方法体内部声明的变量称为局部变量。

## 8、注意：二者在初始化值方面的异同：

同：都有生命周期
异：局部变量除形参外，需显式初始化。

## 2.3.2 基本数据类型

### 变量的分类 - 按数据类型

对于每一种数据都定义了明确的具体数据类型（强类型语言），在内存中分配了不同大小的内存空间。


### 1、整数类型：byte、short、int、long

Java 各整数类型有固定的表数范围和字段长度，不受具体 OS 的影响，以保证 java 程序的可移植性。

java 的整型常量默认为 int 型，声明 long 型常量须后加‘ l ’或‘ L ’java 程序中变量通常声明为 int 型，除非不足以表示较大的数，才使用 long。

| 类型 | 占用存储空间 | 表数范围 |
|---|---|---|
| byte | 1字节=8bit位 | –128 ~ 127 |
| short | 2字节 | –2^15~ 2^15–1 |
| int | 4字节 | –2^31~ 2^31–1 (约21亿) |
| long | 8字节 | –2^63~ 2^63–1 |

1MB = 1024KB 1KB= 1024B B= byte ? bit?
bit: 计算机中的最小存储单位。byte: 计算机中基本存储单元。



### Java 定义的数据类型

一、变量按照数据类型来分：
基本数据类型：
整型：byte \ short \ int \ long
浮点型：float \ double
字符型：char
布尔型：boolean

引用数据类型：
类：class
接口：interface
数组：array

二、变量在类中声明的位置：
成员变量 vs 局部变量

```
class VariableTest1{
    public static void main(String[] args) {
        //1. 整型：byte(1 字节 =8bit) short(2 字节) \ int (4 字节) \ long(8 字节 )
        // ① byte 范围：-128 ~ 127

        byte b1 = 12;
        byte b2 = -128;
    //  b2 = 128; // 编译不通过
        System.out.println(b1);
        System.out.println(b2);

        // ② 声明 long 型变量，必须以“1”或“L”结尾
        short s1 = 128;
        int i1 = 12345;
        long l1 = 345678586;
        System.out.println(l1);
    }
}
```

## 2、浮点类型：float、double

与整数类型类似，Java 浮点类型也有固定的表数范围和字段长度，不受具体操作系统的影响。

浮点型常量有两种表示形式：

十进制数形式：如：5.12 512.0f .512 ( 必须有小数点)

科学计数法形式 : 如：5.12e2 512E2 100E - 2。

float: 单精度，尾数可以精确到 7 位有效数字。很多情况下，精度很难满足需求。

double: 双精度，精度是 float 的两倍。通常采用此类型。

Java 的浮点型常量默认为 double 型，声明 float 型常量，须后加‘ f ’或‘ F ’。

| 类型 | 占用存储空间 | 表数范围 |
| --- | --- | --- |
| 单精度float | 4字节 | -3.403E38 ~ 3.403E38 |
| 双精度double | 8字节 | -1.798E308 ~ 1.798E308 |

## 3、字符类型：char

char 型数据用来表示通常意义上“ 字符 ”(2 字节 )。

Java 中的所有字符都使用 Unicode 编码, 故一个字符可以存储一个字母, 一个汉字，或其他书面语的一个字符。

字符型变量的三种表现形式：

字符常量是用单引号 ( ‘ ’ ) 括起来的单个字符。

例如：char c1 = ‘ a ’ ; char c2 = ‘ 中 ’ ; char c3 = ‘ 9 ’ ;

Java 中还允许使用转义字符‘ \ ’来将其后的字符转变为特殊字符型常量。

例如：char c3 = ‘ \n ’ ; // ’ \n ’ 表示换行符。

直接使用 Unicode 值来表示字符型常量：‘ \uXXXX ’。其中, XXXX 代表一个十六进制整数。如：\u000a 表示 \n。

char 类型是可以进行运算的。因为它都对应有 Unicode 码。



## 4、布尔类型：boolean

boolean 类型用来判断逻辑条件，一般用于程序流程控制：

if 条件控制语句；

while 循环控制语句；

do - while 循环控制语句；

for 循环控制语句；

boolean 类型数据只允许取值 true 和 false，无 null。

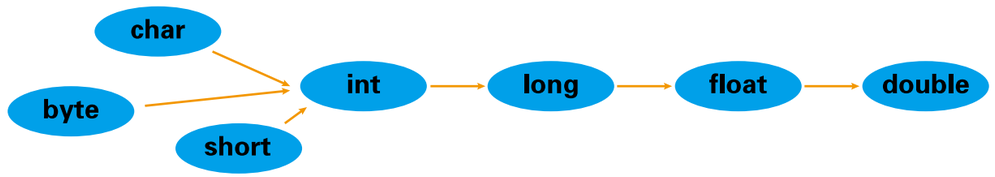
不可以使用 0 或非 0 的整数替代 false 和 true，这点和 C 语言不同。

Java 虚拟机中没有任何供 boolean 值专用的字节码指令，Java 语言表达所操作的 boolean 值，在编译之后都使用 java 虚拟机中的 int 数据类型来代替：true 用 1 表示，false 用 0 表示。———《java 虚拟机规范 8 版》

```
class VariableTest1{
	public static void main(String[] args) {
		//4. 布尔型：boolean
		// ① 只能取两个值之一：true 、false
		// ② 常常在条件判断、循环结构中使用
		boolean bb1 = true;
		System.out.println(bb1);

		boolean isMarried = true;
		if(isMarried){
			System.out.println(" 禁止入内！ ");
		}else{
			System.out.println(" 可以参观！ ");
		}
	}
}
```

## 2.3.3 基本数据类型转换

自动类型转换：容量小的类型自动转换为容量大的数据类型。数据类型按容量大小排序为：




有多种类型的数据混合运算时，系统首先自动将所有数据转换成容量最大的那种数据类型，然后再进行计算。

byte,short,char 之间不会相互转换，他们三者在计算时首先转换为 int 类型。

boolean 类型不能与其它数据类型运算。

当把任何基本数据类型的值和字符串 (String) 进行连接运算时 (+)，基本数据类型的值将自动转化为字符串 (String) 类型。

### 基本数据类型之间的运算规则：

前提：这里讨论只是 7 中基本数据类型变量的运算。不包含 boolean 类型的。

1. 自动类型提升：

当容量小的数据类型的变量和容量大的数据类型的变量做运算时，结果自动提升为容量大的数据类型。

char、byte、short - - >int - - >long - - >float - - >double

特别的：当 byte、char、short 三种类型的变量做运算时，结果为 int 类型。

2. 强制类型转换：

说明：此时容量大小指的是，表示数的范围的大和小。比如：float 容量要大于 long 的容量。



```
class VariableTest2{
	public static void main(String[] args) {
		byte b1 = 2;
		int i1 = 129;
//		byte b2 = b1 + i1;		// 编译不通过
		int i2 = b1 + i1;
		long l1 = b1 + i1;
		System.out.println(i2);
		System.out.println(l1);
		float f = b1 + i1;
		System.out.println(f);
		//*************** 特别的 *************************
		char c1 = 'a';	//97
		int i3 = 10;
		int i4 = c1 + i3;
		System.out.println(i4);
		short s2 = 10;
//		char c3 = c1 + s2;		// 编译错误
		byte b2 = 10;
//		char c3 = c1 + b2;	// 编译不通过
//		short s3 = b2 + s2;  // 编译不通过
//		short s4 = b1 + b2; // 编译不通过
	}	}
class VariableTest4{
	public static void main(String[] args){
		//1. 编码情况
		long l = 123456;
		System.out.println(l);
		// 编译失败：过大的整数
		//long l1 = 452367894586235;
		long l1 = 452367894586235L;
		//*************************
//		float f1 = 12.3;		// 编译失败
		//2. 编码情况 2:
		// 整型变量，默认类型为 int 型
		// 浮点型变量，默认类型为 double 型
		byte b = 12;
	//	byte b1 = b + 1;	// 编译失败
	//	float f1 = b + 12.3;  // 编译失败
	}	}
```

## 2.3.3 字符串类型：String

String 不是基本数据类型，属于引用数据类型。
使用方式与基本数据类型一致。例如：String str=“ abcd ”。
一个字符串可以串接另一个字符串，也可以直接串接其他类型的数据。

String 类型变量的使用：

1. String 属于引用数据类型 。
2. 声明 String 类型变量时，使用一对 "" 。
3. String 可以和 8 种基本数据类型变量做运算，
   且运算只能是连接运算：+
4. 运算的结果任然是 String 类型。

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

		String s1 = "Good Moon!";
		System.out.println(s1);

		String s2 = "a";
		String s3 = "";

//		char c = '';	// 编译不通过

		//*******************************
		int number = 1001;
		String numberStr = " 学号 :";
		String info = numberStr + number;// 连接运算
		boolean b1 = true;
		String info1 = info + true;
		System.out.println(info1);
	}
}
```



## 2.3.4 强制类型转换

自动类型转换的逆过程，将容量大的数据类型转换为容量小的数据类型。使用时要加上强制转换符：()，但可能造成精度降低或溢出，格外要注意。

通常，字符串不能直接转换为基本类型，但通过基本类型对应的包装类则可以实现把字符串转换成基本类型。

如：String a = “ 43 ”; inti= Integer.parseInt(a);
boolean 类型不可以转换为其它的数据类型。

# 2.4 进　制

## 2.4.1 进制与进制间的转换

### 关于进制

所有数字在计算机底层都以二进制形式存在。

对于整数，有四种表示方式：

二进制 (binary)：0,1 ， 满 2 进 1. 以 0b 或 0B 开头。
十进制 (decimal)：0 - 9 ， 满 10 进 1。
八进制 (octal)：0 - 7 ， 满 8 进 1. 以数字 0 开头表示。
十六进制 (hex)：0 - 9 及 A - F, 满 16 进 1. 以 0x 或 0X 开头表示。此处的 A - F 不区分大小写。如：0x21AF +1= 0X21B0。

## 2.4.2 二进制

Java 整数常量默认是 int 类型，当用二进制定义整数时，其第 32 位是符号位；当是 long 类型时，二进制默认占 64 位，第 64 位是符号位。

二进制的整数有如下三种形式：

原码：直接将一个数值换成二进制数。最高位是符号位；

负数的反码：是对原码按位取反，只是最高位（符号位）确定为 1。

负数的补码：其反码加 1。计算机以二进制补码的形式保存所有的整数。正数的原码、反码、补码都相同，负数的补码是其反码 +1。

### 为什么要使用原码、反码、补码表示形式呢？

计算机辨别“ 符号位 ”显然会让计算机的基础电路设计变得十分复杂！于是人们想出了将符号位也参与运算的方法 . 我们知道 , 根据运算法则减去一个正数等于加上一个负数 , 即 : 1-1 = 1 + (-1) = 0 , 所以机器可以只有加法而没有减法 , 这样计算机运算的设计就更简单了。

对于正数来讲：原码、反码、补码是相同的：三码合一。

计算机底层都是使用二进制表示的数值。

计算机底层都是使用的数值的补码保存数据的。

## 2.4.3、进制间转化

二进制转成十进制乘以 2 的幂数。

十进制转成二进制除以 2 取余数。

## 2.5 运算符

**运算符是一种特殊的符号，用以表示数据的运算、赋值和比较等。**

算术运算符
赋值运算符
比较运算符（关系运算符）
逻辑运算符
位运算符
三元运算符



## 2.5.1 算术运算符

| 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（取余） | 7%5 | 2 |
| ++<br>++ | 自增（前）：先运算后取值<br>自增（后）：先取值后运算 | a=2;b=++a;<br>a=2;b=a++; | a=3;b=3<br>a=3;b=2 |
| ––<br>–– | 自减（前）：先运算后取值<br>自减（后）：先取值后运算 | a=2;b=––a;<br>a=3;b=a––; | a=1;b=1<br>a=1;b=2 |
| + | 字符串连接 | "He"+"llo" | "Hello" |

```
class Day3Test{
	public static void main(String[] args) {
		// 除号：/
		int num1 = 12;
		int num2 = 5;
		int resule1 = num1 / num2;
		System.out.println(resule1);	//2

		int result2 = num1 / num2 * num2;
		System.out.println(result2);
		double result3 = num1 / num2;
		System.out.println(result3);	//2.0

		double result4 = num1 / num2 + 0.0;	//2.0
		double result5 = num1 / (num2 + 0.0);	//2.4
		double result6 = (double)num1 / num2;//2.4
		double result7 = (double)(num1 / num2);	//2.0
		System.out.println(result5);
		System.out.println(result6);
		// %：取余运算
		// 结果的符号与被模数的符号相同
```

```
        int m1 = 12;
        int n1 = 5;
        System.out.println("m1 % n1 = " + m1 % n1);
        int m2 = -12;
        int n2 = 5;
        System.out.println("m2 % n2 = " + m2 % n2);
        int m3 = 12;
        int n3 = -5;
        System.out.println("m3 % n3 = " + m3 % n3);
        int m4 = -12;
        int n4 = -5;
        System.out.println("m4 % n4 = " + m4 % n4);
        //(前)++:先自增1，后运算
        //(后)++：先运算，后自增1
        int a1 = 10;
        int b1 = ++a1;
        System.out.println("a1 = " + a1 + ",b1 = " + b1);
        int a2 = 10;
        int b2 = a2++;
        System.out.println("a2 = " + a2 + ",b2 = " + b2);
        int a3 = 10;
        a3++;      //a3++;
        int b3 = a3;
        // 注意点:
        short s1 = 10;
        //s1 = s1 + 1;  // 编译失败
//      s1 = (short)(s1 + 1);// 正确的
        s1++;      // 自增1不会改变本身变量的数据类型
        System.out.println(s1);
        // 问题:
        byte bb1 = 127;
        bb1++;
        System.out.println("bb1 = " + bb1);
        //(前)--:先自减1，后运算
        //(后)--：先运算，后自减1
        int a4 = 10;
        int b4 = a4--;  //int b4 = --a4;
        System.out.println("a4 = " + a4 + ",b4 = " + b4);
    }
}
```



## 算术运算符的注意问题：

如果对负数取模，可以把模数负号忽略不记，如：5%-2=1。但被模数是负数则不可忽略。此外，取模运算的结果不一定总是整数。

对于除号“/”，它的整数除和小数除是有区别的：整数之间做除法时，只保留整数部分而舍弃小数部分。例如：intx=3510;x=x/1000*1000; x 的结果是？

“+”除字符串相加功能外，还能把非字符串转换成字符串 . 例如：System.out.println(“5+5=”+5+5); // 打印结果是？ 5+5=55？

## 2.5.2 赋值运算符

符号：=

当“=”两侧数据类型不一致时，可以使用自动类型转换或使用强制类型转换原则进行处理。

支持连续赋值。

扩展赋值运算符：+=, -=, *=, /=, %=

## 2.5.3 比较运算符

| 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 |
| instanceof | 检查是否是类的对象 | "Hello"instanceof String | TRUE |

比较运算符的结果都是 boolean 型，也就是要么是 true，要么是 false。

比较运算符“==”不能误写成“=”。

## 2.5.4 逻辑运算符

&—逻辑与
|—逻辑或
！—逻辑非
&& —短路与
||—短路或
^ —逻辑异或

| a | b | a&b | a&&b | a\|b | a\|\|b | !a | a^b |
|---|---|---|---|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE | TRUE | TRUE | FALSE | FALSE |
| TRUE | FALSE | FALSE | FALSE | TRUE | TRUE | FALSE | TRUE |
| FALSE | TRUE | FALSE | FALSE | TRUE | TRUE | TRUE | TRUE |
| FALSE | FALSE | FALSE | FALSE | FALSE | FALSE | TRUE | FALSE |

逻辑运算符用于连接布尔型表达式，在 Java 中不可以写成 3<x<6，应该写成 x>3 & x<6 。

“ & ”和“ && ”的区别：

单 & 时，左边无论真假，右边都进行运算；

双 & 时，如果左边为真，右边参与运算，如果左边为假，那么右边不参与运算。

“ | ”和“ || ”的区别同理，|| 表示：当左边为真，右边不参与运算。

异或 (^) 与或 ( | ) 的不同之处是：当左右都为 true 时，结果为 false。理解：异或，追求的是“ 异 ”！



## 2.5.5 位运算符

**位运算是直接对整数的二进制进行的运算**

| 位运算符 | | |
|---|---|---|
| 运算符 | 运算 | 范例 |
| << | 左移 | 3 << 2 = 12 –-> 3*2*2 = 12 |
| >> | 右移 | 3 >> 1 = 1 –-> 3/2 = 1 |
| >>> | 无符号右移 | 3 >>> 1 = 1 –-> 3/2 = 1 |
| & | 与运算 | 6 & 2 = 2 |
| \| | 或运算 | 6 \| 3 = 7 |
| ^ | 异或运算 | 6 ^ 3 = 5 |
| ~ | 取反运算 | ~ 6 = –7 |

**注意：无 <<<**

结论：

1. 位运算符操作的都是整型的数据变量 。

2.<< : 在一定范围内，每向左移一位，相当于 * 2 >> : 在一定范围内，每向右移一位，相当于 / 2。

## 2.5.6 三元运算符

1. 结构：( 条件表达式 ) ？表达式 1 : 表达式 2
2. 说明
   ① 条件表达式的结果为 boolean 类型
   ② 根据条件表达式真或假，决定执行表达式 1，还是表达式 2.
      如果表达式为 true, 则执行表达式 1
      如果表达式为 false, 则执行表达式 2
   ③ 表达式 1 和表达式 2 要求是一致的。
   ④ 三元运算符是可以嵌套的
3. 凡是可以使用三元运算的地方，都是可以改写 if - else。
   反之，则不一定成立！！！

# 2.6 运算符的优先级

运算符有不同的优先级，所谓优先级就是表达式运算中的运算顺序。如右表，上一行运算符总优先于下一行。

只有单目运算符、三元运算符、赋值运算符是从右向左运算的。

| | |
|---|---|
| | . () {} ; , |
| R —> L | ++ -- ~ !(data type) |
| L —> R | * / % |
| L —> R | + - |
| L —> R | << >> >>> |
| L —> R | < > <= >= instanceof |
| L —> R | == != |
| L —> R | & |
| L —> R | ^ |
| L —> R | \| |
| L —> R | && |
| L —> R | \|\| |
| R —> L | ? : |
| R —> L | = *= /= %= |
| | += -= <<= >>= |
| | >>>= &= ^= \|= |




# 2.7 程序流程控制

## 2.7.1 程序流程控制概述

**流程控制语句是用来控制程序中各语句执行顺序的语句，可以把语句组合成能完成一定功能的小逻辑模块。**

**其流程控制方式采用结构化程序设计中规定的三种基本流程结构，即：**

### 顺序结构

程序从上到下逐行地执行，中间没有任何判断和跳转。

### 分支结构

根据条件，选择性地执行某段代码。
有 if…else 和 switch - case 两种分支语句。

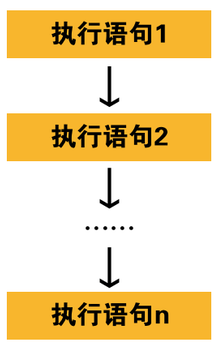
### 循环结构

根据循环条件，重复性的执行某段代码。
有 while、do…while、for 三种循环语句。
注：JDK1.5 提供了 foreach 循环，方便的遍历集合、数组元素。

## 2.7.2 顺序结构

Java 中定义成员变量时采用合法的前向引用。如：

```
public class Test{
        int num1 = 12;
        int num2 = num1 + 2;
}

// 错误形式：
public class Test{
        int num2 = num1 + 2;
        int num1 = 12;
}
```

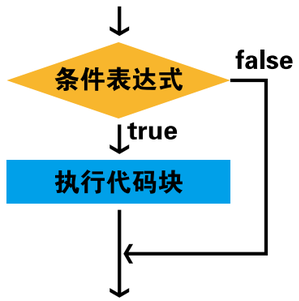


# 2.8 分支语句

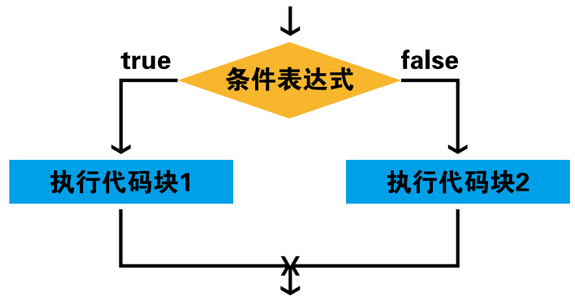
## 1、分支语句：if-else 结构

if语句三种格式：

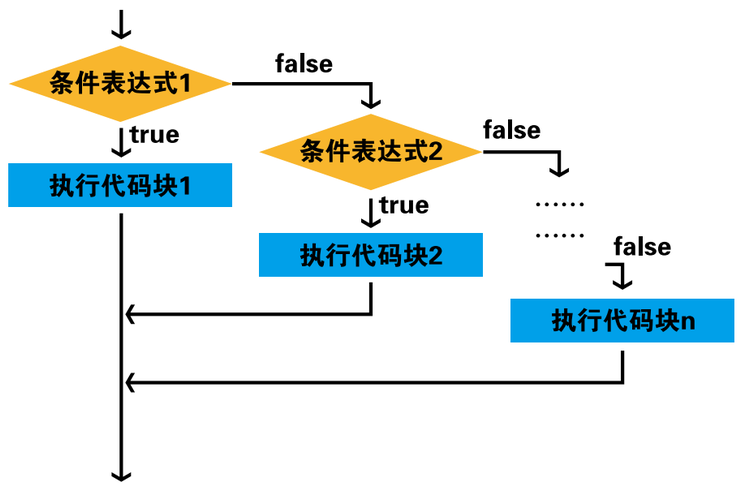
1. if(条件表达式){
   执行代码块;
   }




2. if(条件表达式){
   执行代码块1;
   }else{
   执行代码块2;
   }




3. if(条件表达式){
   执行代码块1;
   }else{
   执行代码块2;
   }
   ……
   else{
   执行代码块n;
   }




## if-else 使用说明：

条件表达式必须是布尔表达式（关系表达式或逻辑表达式）、布尔变量；

语句块只有一条执行语句时，一对 {} 可以省略，但建议保留；

if-else 语句结构，根据需要可以嵌套使用；

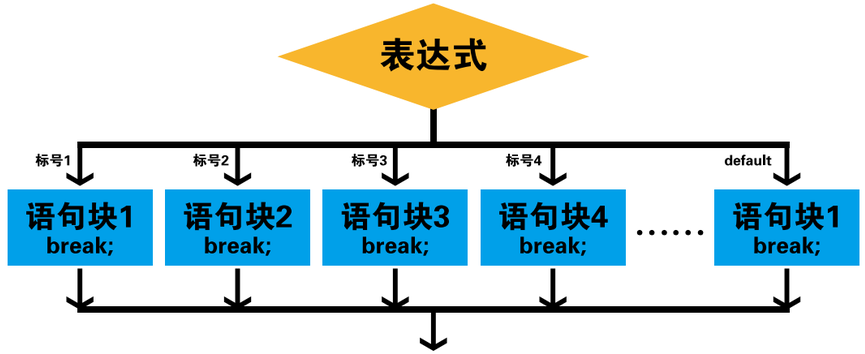
当 if-else 结构是“ 多选一 ”时，最后的 else 是可选的，根据需要可以省略；

当多个条件是“ 互斥 ”关系时，条件判断语句及执行语句间顺序无所谓当多个条件是“ 包含 ”关系时，“ 小上大下 / 子上父下 ”。



## 2、 分支语句：switch-case 结构

```
switch(表达式){
case 常量1:
        语句1;
        //break;
case 常量2:
        语句2;
        //break;
……
case 常量N:
        语句N;
        //break;
default:
        语句;
        //break;
```




# 2.9 循环结构

## 1、循环结构

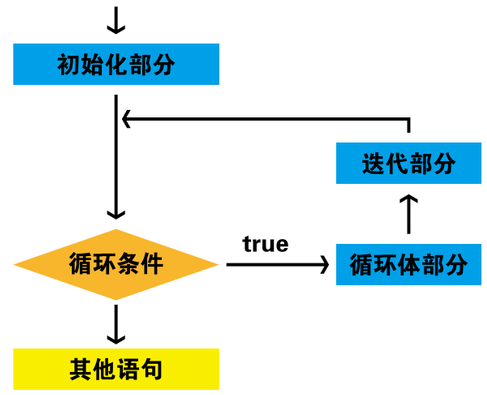
在某些条件满足的情况下，反复执行特定代码的功能。

## 2、循环语句分类

for 循环
while 循环
do-while 循环

循环语句的四个组成部分：

> 初始化部分（init_statement）
> 循环条件部分（test_exp）
> 循环体部分（boby_statement）
> 迭代部分（alter_statement）




## 3、for 循环

语法格式

for( ①初始化部分 ; ②循环条件部分 ; ④迭代部分 ) {
③循环体部分 ;
}

执行过程：① - ② - ③ - ④ - ② - ③ - ④ - ② - ③ - ④ - ..... - ②

说明：

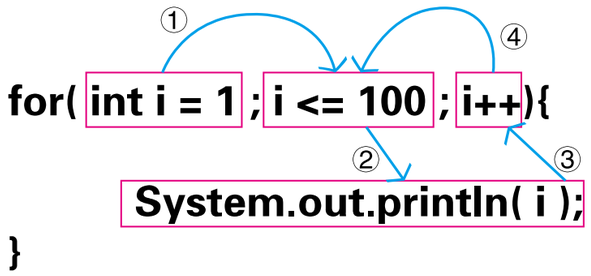
②循环条件部分为 boolean 类型表达式，当值为 false 时，退出循环

①初始化部分可以声明多个变量，但必须是同一个类型，用逗号分隔

④可以有多个变量更新，用逗号分隔

For 循环结构的使用
一、循环结构的四个要素
① 初始化条件
② 循环条件
③ 循环体
④ 迭代条件

二、for 循环的结构
for( ① ; ② ; ④ ){
　　③ ;
}




```
class ForTest{
	public static void main(String[] args){
		for(int i=1;i <= 5 ;i++){
			System.out.println("Hello World!");
		}

		// 练习:
		int num = 1;
		for(System.out.print('a');num <= 3;System.out.
print('c'),num++){
			System.out.print('b');
		}

		// 遍历 100 以内的偶数，获取所有偶数的和，输出偶数的个数
		int sum = 0;	// 记录所有偶数的和
		int count = 0;
		for(int i = 1;i <= 100;i++){
			if(i %2 == 0){
				System.out.println(i);
				sum += i;
				count++;
			}
		}
		System.out.println("100 以内的偶数的和：" + sum);
		System.out.println(" 个数为：" + count);
	}
}
```



## 4、while 循环

①初始化部分
while( ②循环条件部分 ){
　③循环体部分 ;
　④迭代部分 ;
}

执行过程：① - ② - ③ - ④ - ② - ③ - ④ - ② - ③ - ④ - … - ②
说明：
1. 写 while 循环千万要小心不要丢了迭代条件。一旦丢了，就可能导致死循环！
2. 写程序要避免死循环。
3. 能用 while 循环的，可以用 for 循环，反之亦然。二者可以相互转换。
区别：for 循环和 while 循环的初始化条件部分的作用范围不同。

While 循环结构的使用
一、循环结构的四个要素
① 初始化条件
② 循环条件
③ 循环体
④ 迭代条件

二、while 循环的结构
① ;
while( ② ){
③ ;
④ ;
}

```
class WhileTest{
	public static void main(String[] args){
		// 遍历 100 以内的所有偶数
		int i = 1;
		while(i <= 100){
			if(i % 2 == 0){
				System.out.println(i);
			} i++;
		}
	}
}
```

## 5、do-while 循环

do-while 循环结构的使用
一、循环结构的四个要素
① 初始化条件
② 循环条件 - - - > 是 boolean 类型
③ 循环体
④ 迭代条件

二、do-while 循环的结构
①
do{
　　③;
　　④;
}while( ② );

执行过程：① - ③ - ④ - ② - ① - ③ - ④ - ... - ②

说明：do-while 循环至少执行一次循环体。

**1. 不在循环条件部分限制次数的结构：**
**while(true) , for(true)**

**2. 结束循环的几种方式：**
**方式一：循环条件部分返回 false;**
**方式二：在循环体中，执行 break;**

## 6、嵌套循环结构

## 嵌套循环（多重循环）

将一个循环放在另一个循环体内，就形成了嵌套循环。
其中，for ,while ,do…while 均可以作为外层循环或内层循环。
实质上，嵌套循环就是把内层循环当成外层循环的循环体。
当只有内层循环的循环条件为 false 时，才会完全跳出内层循环，才可结束外层的当次循环，开始下一次的循环。
设外层循环次数为 m 次，内层为 n 次，则内层循环体实际上需要执行 m*n 次。

嵌套循环的使用
1. 嵌套循环：将一个循环结构 A 声明在另一个循环结构 B 的循环体中，就构成了嵌套循环。

2.
外层循环：循环结构 B。
内层循环：循环结构 A。

3. 说明
① 内层循环遍历一遍，只相当于外层循环循环体执行了一次
② 假设外层循环需要执行 m 次，内层循环需要执行 n 次。此时内层循环的循环体一共执行了 m * n 次。

4. 技巧
外层循环控制行数，内层循环控制列数。

# 2.10 break、continue 的使用

## 1、break 的使用

break 语句用于终止某个语句块的执行

```
{
        ......
        break;
        ......
}
```

break 语句出现在多层嵌套的语句块中时，可以通过标签指明要终止的是哪一层语句块

```
label1: {
        ......
        label2: {
                ......
                label3: {
                 ......
                break label2;
                ......
                }
        }
}
```

## 2、continue 的使用

continue 语句：
> continue 只能使用在循环结构中。
> continue 语句用于跳过其所在循环语句块的一次执行，继续下一次循环。
> continue 语句出现在多层嵌套的循环语句体中时，可以通过标签指明要跳过的是哪一层循环。



## 3、return 的使用

return：并非专门用于结束循环的，它的功能是结束一个方法。当一个方法执行到一个 return 语句时，这个方法将被结束。

与 break 和 continue 不同的是，return 直接结束整个方法，不管这个 return 处于多少层循环之内。

## 4、特殊流程控制语句说明

break 只能用于 switch 语句和循环语句中。

continue 只能用于循环语句中。

二者功能类似，但 continue 是终止本次循环，break 是终止本层循环。

break、continue 之后不能有其他的语句，因为程序永远不会执行其后的语句。

标号语句必须紧接在循环的头部。标号语句不能用在非循环语句的前面。

很多语言都有 goto 语句，goto 语句可以随意将控制转移到程序中的任意一条语句上，然后执行它。但使程序容易出错。Java 中的 break 和 continue 是不同于 goto 的。

# 项目一

## 目　　标

模拟实现一个基于文本界面的《家庭记账软件》。
掌握初步的变成技巧。
主要掌握以下知识点：

变量的定义，
基本数据类型的使用，
循环语句，
分支语句，
方法声明、调用和返回值的接收，
简单的屏幕输出格式控制

## 需求说明

模拟实现基于文本界面的《家庭记账软件》。
该软件能够记录家庭收入、支出，并能够打印收支明细表。
项目采用分级菜单方式。主菜单如下：

```
- - - - - - - - - - - - - 家庭收支记账软件 - - - - - - - - - - - - -

                    1. 收支明细
                    2. 登记收入
                    3. 登记支出
                    4. 退　　出

                   请选择 (1 - 4):
```

假设家庭其实的生活基本金为：10000 元。
每次登记收入（菜单 2）后，收入的金额应累加到基本金上，并记录本次收入明细，以便后续的查询。
每次登记支出（菜单 3）后，支出的金额应从基本金中扣除，并记录本次支出明细，以便后续的查询。
查询收支明细（菜单 1）时，将显示所有的收入、支出明细表。



## 基本金和收支明细的记录

基本金的记录可以用 int 类型的变量来实现：

```
int balance = 10000 ;
```

收支明细记录可以使用 String 类型的变量来实现，其初始值为明细表的表头。例如：

```
String details = " 收支 \t 账户金额 \t\t 收支金额 \t\t 说　明 \n"
```

在登记收支时，将金额与 balance 相加或相减，收支记录直接串接到 details 后面即可。

## Utility 模块

```
package Utility;
import java.util.Scanner;

// 将不同的功能封装为方法，就是可以直接通过调用方法使用它的功能，而无需考虑具体的功能实现细节
public class Utility {
	private static Scanner scanner = new Scanner(System.in);

	// 用于界面菜单的选择。该方法读取键盘，如果用户输入 ‘1’ - ‘4’ 中的任意字符，
	// 则方法返回，返回值为用户键入字符。
	public static char readMenuSelection() {
		char c;
		for (;;) {
			String str = readKeyBoard(1);
			c = str.charAt(0);
			if (c != '1' && c != '2' && c != '3' && c != '4') {
				System.out.println(" 选择错误，请重新输入： ");
			} else break;
		}
		return c;
	}

	// 用于收入和支出金额的输入。该方法从键盘读取一个不超过 4 位长度的整数，并将其作为方法的返回值。
	public static int readNumber() {
		int n;
		for (;;) {
			String str = readKeyBoard(4);
			try {
				n = Integer.parseInt(str);
				break;
			} catch (NumberFormatException e) {
				System.out.println(" 数字输入错误，请重新输入： ");
			}
		}
		return n;
	}
```



```
	// 用于收入和支出说明的输入。该方法从键盘读取一个不超过 8 位长度的字符串，并将其作为方法的返回值。
	public static String readString() {
		String str = readKeyBoard(8);
		return str;
	}

	// 用于确认选择的输入。该方法从键盘读取 ‘Y’ 或 ‘N’ ，并将其作为方法的返回值。
	public static char readConfirmSelection() {
		char c;
		for (;;) {
			String str = readKeyBoard(1).toUpperCase();
			c = str.charAt(0);
			if (c == 'Y' || c == 'N') {
				break;
			} else {
				System.out.println(" 选择错误，请重新输入： ");
			}
		}
		return c;
	}

	private static String readKeyBoard(int limit) {
		String line = "";
		while (scanner.hasNextLine()) {
			line = scanner.nextLine();
			if (line.length() == 0) {
				continue;
			}
			if (line.length() < 1 || line.length() > limit) {
				System.out.println(" 输入长度 (不大于 " + limit + ") 错误，请重新输入 ");
				continue;
			}
			break;
		}
		return line;
	}
}
```

## FamilyAccount 模块

```
package Utility;

public class FamilyAccount {
	public static void main(String[] args) {
		boolean isFlag = true;

		// 用于记录用户的收入和支出详情
		String details = " 收支 \t 账户金额 \t\t 收支金额 \t\t 说　明 \n";

		// 初始金额
		int balance = 10000;

		while (isFlag) {
			System.out.println("------------------- 家庭收支记账软
件 -------------------\n");
			System.out.println("                1. 收支明细 ");
			System.out.println("                2. 登记收入 ");
			System.out.println("                3. 登记支出 ");
			System.out.println("                4. 退　出 \n");
			System.out.print("                请选择 (1-4)： ");

			// 获取用户的选择：1-4
			char selection = Utility.readMenuSelection();

			switch (selection) {
			case '1':
				System.out.println("------------------- 当前收支明
细记录 -------------------\n");
				System.out.println(" 收支 \t 账户金额 \t\t 收支金额
\t\t 说　明 \n");
				System.out.println(details);
				System.out.println
("-----------------------------------------------------------");
				break;

			case '2':
				System.out.print(" 本次收入金额： ");
				int addMoney = Utility.readNumber();
```



```
				System.out.print(" 本次收入说明： ");
				String addInfo = Utility.readString();
				// 处理 balance
				balance += addMoney;

				// 处理 details
				details += (" 收　入 \t" + balance + "\t\t" +
addMoney + "\t\t" + addInfo + "\n");
				System.out.println("------------------------ 登记完
成 ------------------------\n");
				break;

			case '3':
				System.out.print(" 本次支出金额： ");
				int minusMoney = Utility.readNumber();
				System.out.print(" 本次支出金额： ");
				String minusInfo = Utility.readString();

				// 处理 balance
				if(balance >= minusMoney) {
					balance -= minusMoney;
					details += (" 收　入 \t" + balance + "\t\t" +
minusMoney + "\t\t" + minusInfo + "\n");
				}else {
					System.out.println(" 支出超过账户金额，");
				}

				System.out.println("------------------------ 登记完
成 ------------------------\n");
				break;
			case '4':
				System.out.print(" 确认是否退出 (Y/N):");
				char isExit = Utility.readConfirmSelection();
				if (isExit == 'Y') {
					isFlag = false;
				}
			}
		}
	}
}
```

# 三　数　　组

## 3.1　数组的概述

1. 数组的理解：数组 (Array)，是多个相同类型数据按一定顺序排列的集合，并使用一个名字命名，并通过编号的方式对这些数据进行统一管理。

2. 数组的相关概念：
　　> 数组名
　　> 元素
　　> 角标、下标、索引
　　> 数组的长度：元素的个数

3. 数组的特点：
　　1. 数组属于引用类型的变量。数组的元素，既可以是基本数据类型，也可以是引用数据类型。
　　2. 创建数组对象会在内存中开辟一整块连续的空间；
　　3. 数组的长度一旦确定，就不能修改；
　　4. 数组是有序排列的。

4. 数组的分类：
　　① 按照维数：一维数组、二维数组、三维数组……
　　② 按照数组元素类型：基本数据类型元素的数组、引用类型元素的数组。

## 3.2　一维数组的使用

① 一维数组的声明和初始化
② 如何调用数组的指定位置的元素
③ 如何获取数组的长度
④ 如何遍历数组
⑤ 数组元素的默认初始化值：见 ArrayTest1.java
⑥ 数组的内存解析：见 ArrayTest1.java



### 1、代码案例 1——ArrayTest.java

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

		// 1. 一维数组的声明和初始化
		int num;  // 声明
		num = 10;  // 初始化
		int id = 1001;  // 声明 + 初始化

		int[] ids;  // 声明
		//1.1 静态初始化：数组的初始化和数组元素的赋值操作同时进行
		ids = new int[]{1001,1002,1003,1004};
		//1.2 动态初始化：数组的初始化和数组元素的赋值操作分开进行
		String[] names = new String[5];

		// 错误的写法：
//		int[] arr1 = new int[];	// 未赋值、未指明长度
//		int[5] arr2 = new int[5];
//		int[] arr3 = new int[3]{1,2,3};

		// 也是正确的写法：
		int[] arr7 = {1,2,3,5,4};  // 类型推断

		/* 总结：数组一旦初始化完成，其长度就确定了。
		*/

		// 2. 如何调用数组的指定位置的元素：通过角标的方式调用。
		// 数组的角标 ( 或索引 ) 从 0 开始的，到数组的长度 -1 结束
		names[0] = " 张郃 ";
		names[1] = " 王陵 ";
		names[2] = " 张学良 ";
		names[3] = " 王传志 ";  //charAt(0)
		names[4] = " 李峰 ";
//		names[5] = " 周礼 ";
		// 如果数组超过角标会通过编译，运行失败。

		// 3. 如何获取数组的长度
		// 属性：length
		System.out.println(names.length);  //5
```

```
        System.out.println(ids.length);  //4

        //4. 如何遍历数组
//      System.out.println(names[0]);
//      System.out.println(names[1]);
//      System.out.println(names[2]);
//      System.out.println(names[3]);
//      System.out.println(names[4]);

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

## 2、代码案例 2——ArrayTest1.java

⑤ 数组元素的默认初始化值

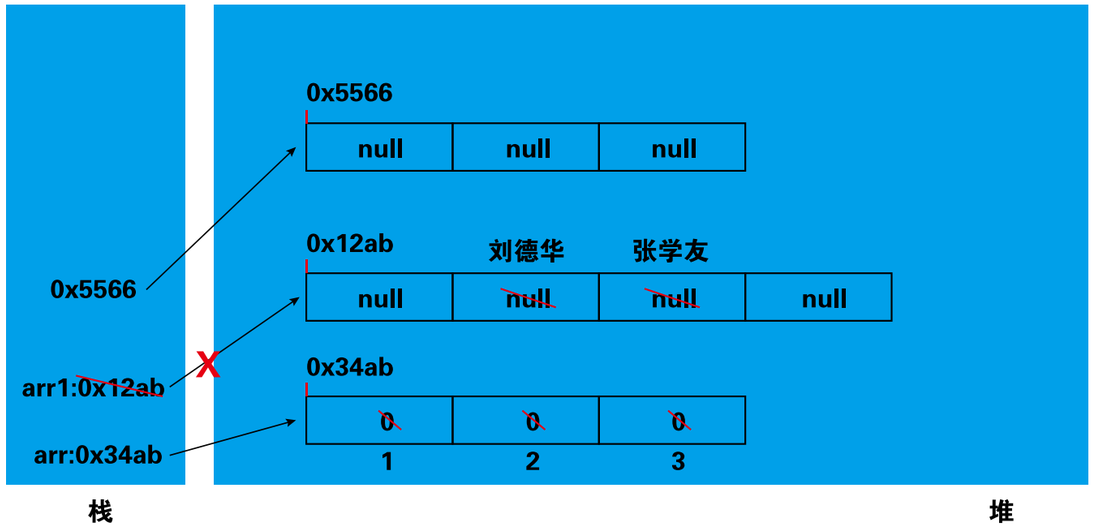
> 数组元素是整形：0
> 数组元素是浮点型：0.0
> 数组元素是 char 型：0 或 '\u0000'，而非 '0'
> 数组元素是 boolean 型 :false
> 数组元素是引用数据类型：null

```
public class ArrayTest1 {
    public static void main(String[] args) {
        //5. 数组元素的默认初始化值
        int[] arr = new int[4];
        for(int i = 0;i < arr.length;i++){
            System.out.println(arr[i]);
        }
        System.out.println("******************");

        short[] arr1 = new short[4];
        for(int i = 0;i < arr1.length;i++){
            System.out.println(arr1[i]);
        }
        System.out.println("******************");

        float[] arr2 = new float[5];
        for(int i = 0;i < arr2.length;i++){
            System.out.println(arr2[i]);
        }
        System.out.println("******************");
        char[] arr3 = new char[5];
        for(int i = 0;i < arr3.length;i++){
            System.out.println("----" + arr3[i] + "****");
        }

        if(arr3[0] == 0){
            System.out.println(" 你好！  ");
        }
        System.out.println("******************");

        boolean[] arr4 = new boolean[5];
        System.out.println(arr4[0]);

        System.out.println("******************");
        String[] arr5 = new String[5];
        System.out.println(arr5[0]);
        // 验证
        if(arr5[0] == null){
            System.out.println(" 北京天气好差！  ");
        }
    }
}
```

### 3.2.1 内存的简化结构

## 3.2.2 一维数组的内存解析

```
int[] arr = new int[]{1,2,3};
String[] arr1 = new String[4];
arr1[1] = “刘德华”;
arr1[2] = “张学友”;
arr1 = new String[3];
System.out.println(arr1[1]);  //null
```




按照图中步骤，最后数组内存解析完成，数组内部值为 null。



# 3.3 多维数组的使用

**Java 语言里提供了支持多维数组的语法。**

## 3.3.1 二维数组

1. 理解

对于二维数组的理解，我们可以看成是一维数组 array1 又作为另一个一维数组 array2 的元素而存在。

其实，从数组底层的运行机制来看，其实没有多维数组。

2. 二维数组的使用：

① 二维数组的初始化。
② 如何调用数组的指定位置的元素。
③ 如何获取数组的长度。
④ 如何遍历数组。
⑤ 数组元素的默认初始化值：见 ArrayTest3.java。
⑥ 数组的内存解析：见 ArrayTest3.java。

### 1、代码案例——ArrayTest2.java

```
public class ArrayTest2 {
	public static void main(String[] args) {
		//1. 二维数组的声明和初始化
		int[] arr = new int[]{1,2,3};
		// 静态初始化
		int[][] arr1 = new int[][]{{1,2,3},{4,5,6},{7,8,9}};
		// 动态初始化 1
		String[][] arr2 = new String[3][2];
		// 动态初始化 2
		String[][] arr3 = new String[3][];

		// 错误的情况
//		String[][] arr4 = new String[][];
//		String[][] arr5 = new String[][4];
//		String[][] arr6 = new String[4][3]{{1,2,3},{4,5,6},{7,8,9}};

		// 正确的情况：
		int arr4[][] = new int[][]{{1,2,3},{4,5,12,6},{7,8,9}};
		int[] arr5[] = new int[][]{{1,2,3},{4,5,6},{7,8,9}};
```

```
int[][] arr6 = {{1,2,3},{4,5,6},{7,8,9}};

//2. 如何调用数组的指定位置的元素
System.out.println(arr1[0][1]); //2
System.out.println(arr2[1][1]); //null

// 定义 arr3 的 [1] 为长度为 4 的字符数组
arr3[1] = new String[4];

System.out.println(arr3[1][0]); // 没有上句，会报错

//3. 获取数组的长度
System.out.println(arr4.length);      //3
System.out.println(arr4[0].length);   //3
System.out.println(arr4[1].length);   //4

//4. 如何遍历二维数组
for(int i = 0;i < arr4.length;i++){
        for(int j = 0;j < arr4[i].length;j++){
                System.out.print(arr4[i][j] + " ");
        }
        System.out.println();
    }
}
```



## 2、代码案例——ArrayTest3.java

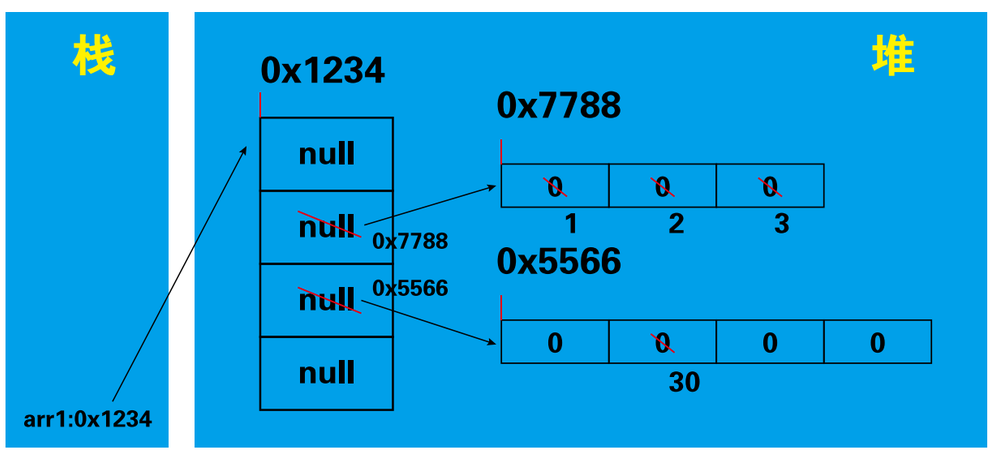
二维数组的使用：
规定：二维数组分为外层数组的元素，内层数组的元素
int[][] arr = new int[4][3];
外层元素 :arr[0],arr[1] 等
内层元素 :arr[0][0],arr[1][2] 等
⑤ 数组元素的默认初始化值
针对于初始化方式一：比如：int[][] arr = new int[4][3];
外层元素的初始化值为：地址值
　　内层元素的初始化值为：与一维数组初始化情况相同
针对于初始化方式而：比如：int[][] arr = new int[4][];
　　外层元素的初始化值为：null
　　内层元素的初始化值为：不能调用，否则报错。
⑥ 数组的内存解析

```
public class ArrayTest3 {
    public static void main(String[] args) {
        int[][] arr = new int[4][3];
        System.out.println(arr[0]);       //[I@15db9742
        System.out.println(arr[0][0]);   //0

//      System.out.println(arr);    //ArrayTest3.java

        System.out.println("***********************");
        float[][] arr1 = new float[4][3];
        System.out.println(arr1[0]);      // 地址值
        System.out.println(arr1[0][0]); //0.0
        System.out.println("***********************");
        String[][] arr2 = new String[4][2];
        System.out.println(arr2[1]);      // 地址值
        System.out.println(arr2[1][1]); //null

        System.out.println("**********************");
        double[][] arr3 = new double[4][];
        System.out.println(arr3[1]);      //null
//      System.out.println(arr3[1][0]); // 报错
    }
}
```

## 3.3.2 二维数组的内存解析

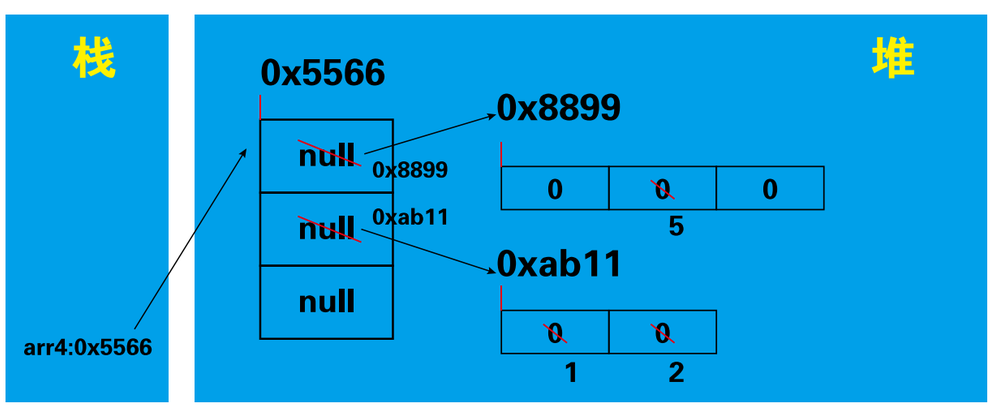
### 1、案例 1

```
int[][] arr1 = new int[4][];
arr1[1] = new int[]{1,2,3};
arr1[2] = new int[4];
arr1[2][1] = 30;
```




### 2、案例 2

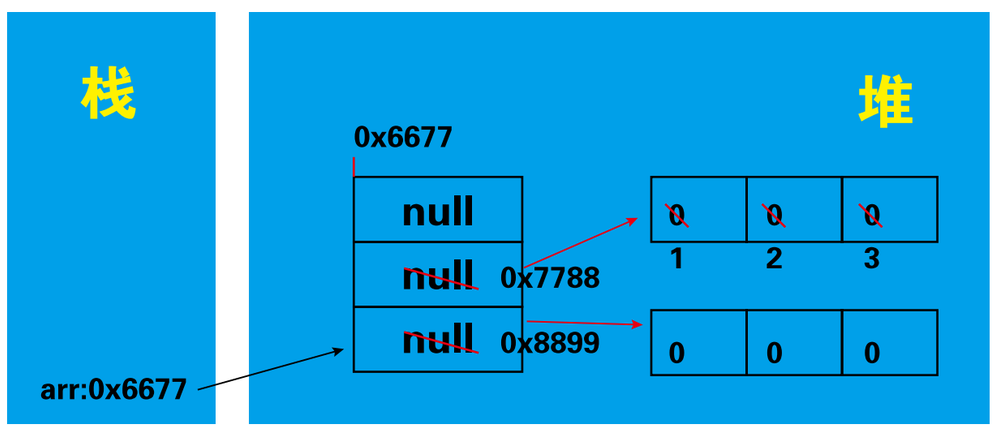
```
int[][] arr4= newint[3][];
System.out.println(arr4[0]);//null
System.out.println(arr4[0][0]);// 报错
arr4[0] = new int[3];
arr4[0][1] = 5;
arr4[1] = new int[]{1,2};
```






### 3、案例 3

```
int[][] arr = new int[3][];
arr[1] = new int[]{1,2,3};
arr[2] = new int[3];
System.out.println(arr[0]);//null
System.out.println(arr[0][0]);// 报异常
```



## 3.4 数组中涉及到的常见算法

1. 数组元素的赋值（杨辉三角、回形数等）。
2. 求数值型数组中元素的最大值、最小值、平均数、总和等。
3. 数组的复制、反转、查找（线性查找、二分法查找）。
4. 数组元素的排序算法。

### 3.4.1 数组元素的赋值

回形数格式方阵的实现
从键盘输入一个整数（1~20）
则以该数字为矩阵的大小，把 1,2,3...n*n 的数字按照顺时针螺旋的形式填入其中。
例如： 输入数字 2，则程序输出：
1 2
4 3
输入数字 3，则程序输出：
1 2 3
8 9 4
7 6 5
输入数字 4， 则程序输出：
1 2 3 4
12 13 14 5
11 16 15 6
10 9 8 7



```
public class ArrayTest {
	public static void main(String[] args) {
		Scanner scanner = new Scanner(System.in);
		System.out.println(" 输入一个数字 :");
		int len = scanner.nextInt();
		int[][] arr = new int[len][len];
		int s = len * len;
		/*
		 * k = 1: 向右 k = 2: 向下 k = 3: 向左 k = 4: 向上
		 */
		int k = 1;
		int i = 0, j = 0;
		for (int m = 1; m <= s; m++) {
			if (k == 1) {
				if (j < len && arr[i][j] == 0) {
					arr[i][j++] = m;
				} else {
					k = 2;
					i++;
					j--;
					m--;
				}
			} else if (k == 2) {
				if (i < len && arr[i][j] == 0) {
					arr[i++][j] = m;
				} else {
					k = 3;
					i--;
					j--;
					m--;
				}
			} else if (k == 3) {
				if (j >= 0 && arr[i][j] == 0) {
					arr[i][j--] = m;
				} else {
					k = 4;
					i--;
					j++;
					m--;
				}
```

```
            } else if (k == 4) {
                if (i >= 0 && arr[i][j] == 0) {
                    arr[i--][j] = m;
                } else {
                    k = 1;
                    i++;
                    j++;
                    m--;
                }
            }
        }
        // 遍历
        for (int m = 0; m < arr.length; m++) {
            for (int n = 0; n < arr[m].length; n++) {
                System.out.print(arr[m][n] + "\t");
            }
            System.out.println();
        }
}
```

## 3.4.2 数组元素的基本操作

**算法的考察：求数值型数组中元素的最大值、最小值、平均数、总和等**

**定义一个 int 型的一维数组，包含 10 个元素，分别赋一些随机整数，然后求出所有元素的最大值，最小值，和值，平均值，并输出出来。**

**要求：所有随机数都是两位数。**

[10,99]
公式：(int)(Math.random() * (99 - 10 + 1) + 10)



```
public class ArrayTest1 {
    public static void main(String[] args) {
        int[] arr = new int[10];
        // 数组赋值
        for(int i = 0;i <arr.length;i++){
            arr[i] = (int)(Math.random() * (99 - 10 + 1) + 10);
        }
        // 遍历
        for(int i =0;i < arr.length;i++){
            System.out.print(arr[i] + " ");
        }
        System.out.println();
        // 求数组元素的最大值
        int maxValue = arr[0];
        for(int i = 1;i <arr.length;i++){
            if(maxValue < arr[i]){
                maxValue = arr[i];
            }
        }
        System.out.println(" 最大值： " + maxValue);

        // 求数组元素的最小值
        int minValue = arr[0];
        for(int i = 1;i <arr.length;i++){
            if(minValue > arr[i]){
                minValue = arr[i];
            }
        }
        System.out.println(" 最小值： " + minValue);
        // 求数组元素的总和
        int sum = 0;
        for(int i = 1;i <arr.length;i++){
            sum += arr[i];
        }
        System.out.println(" 总和： " + sum);
        // 求数组元素的平均数
        double avgVales = sum / arr.length;
        System.out.println(" 平均数： " + avgVales);
    }
}
```

## 3.4.3 数组元素的基本操作 2

使用简单数组
(1) 创建一个名为 ArrayTest 的类，在 main() 方法中声明 array1 和 array2 两个变量，他们是 int[] 类型的数组。
(2) 使用大括号 {}，把 array1 初始化为 8 个素数：2,3,5,7,11,13,17,19。
(3) 显示 array1 的内容。
(4) 赋值 array2 变量等于 array1，修改 array2 中的偶索引元素，使其等于索引值 ( 如 array[0]=0,array[2]=2)。打印出 array1。

```
public class ArrayTest2 {
	public static void main(String[] args) {
		// 声明 array1 和 array2 两个 int[] 变量
		int[] array1,array2;
		//array1 初始化
		array1 = new int[]{2,3,5,7,11,13,17,19};
		// 显示 array1 的内容 == 遍历。
		for(int i = 0;i < array1.length;i++){
			System.out.print(array1[i] + "\t");
		}
		// 赋值 array2 变量等于 array1
	// 不能称作数组的复制。
		array2 = array1;
		// 修改 array2 中的偶索引元素，使其等于索引值 ( 如 array[0]=0,array[2]=2)。
		for(int i = 0;i < array2.length;i++){
			if(i % 2 == 0){
				array2[i] = i;
			}	}
		System.out.println();
		// 打印出 array1。
		for(int i = 0;i < array1.length;i++){
			System.out.print(array1[i] + "\t");
		}
	}	}
```



## 3.4.4 数组的复制、反转、查找

### 1、复制、反转

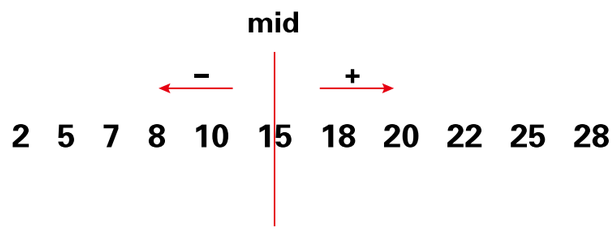
算法的考察：数组的复制、反转、查找 ( 线性查找、二分法查找 )

```
public class ArrayTest3 {
	public static void main(String[] args) {

		String[] arr = new String[]{"SS","QQ","YY","XX","TT","KK","EE","GG"};

		// 数组的复制
		String[] arr1 = new String[arr.length];
		for(int i = 0;i < arr1.length;i++){
			arr1[i] = arr[i];
		}

		// 数组的反转
		// 方法一：
//		for(int i = 0;i < arr.length / 2;i++){
//			String temp = arr[i];
//			arr[i] = arr[arr.length - i - 1];
//			arr[arr.length - i - 1] = temp;
//		}

		// 方法二：
		for(int i = 0,j = arr.length - 1;i < j;i++,j--){
			String temp = arr[i];
			arr[i] = arr[j];
			arr[j] = temp;
		}

		// 遍历
		for(int i = 0;i < arr.length;i++){
			System.out.print(arr[i] + "\t");
		}
		System.out.println();

		// 查找（或搜索）
		// 线性查找
```

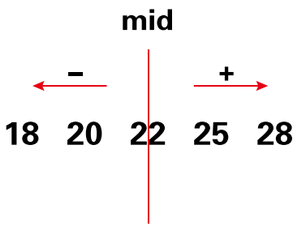
```
        String dest = "BB";  // 要查找的元素
        dest = "CC";
        boolean isFlag = true;

        for(int i = 0;i < arr.length;i++){
            if(dest.equals(arr[i])){
                System.out.println(" 找到了指定元素，位置为: " + i);
                isFlag = false;
                break;
            }
        }
        if(isFlag){
            System.out.println(" 很遗憾，没找到！ ");
        }
    }
}
```

## 2、二分法查找算法

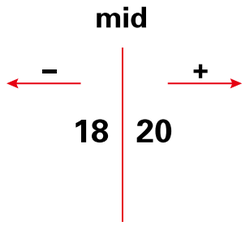
step 1:




step 2:




step 3:






```
public class ArrayTest3 {
    public static void main(String[] args) {
        // 二分法查找：
        // 前提：所要查找的数组必须有序
        int[] arr2 = new int[]{-98,-34,2,34,54,66,79,105,210,333};

        int dest1 = -34;
        int head = 0;    // 初始的首索引
        int end = arr2.length - 1; // 初始的末索引
        boolean isFlag1 = true;
        while(head <= end){
            int middle = (head + end)/2;

            if(dest1 == arr2[middle]){
                System.out.println(" 找到了指定元素，位置为：" +
middle);
                isFlag1 = false;
                break;
            }else if(arr2[middle] > dest1){
                end = middle - 1;
            }else{       //arr2[middle] < dest1
                head = middle + 1;
            }
        }

        if(isFlag1){
            System.out.println(" 很遗憾，没找到！ ");
        }
    }
}
```

### 3.4.5 数组元素的排序算法

**排序：假设含有 n 个记录的序列为 {R1，R2，…,Rn}, 其相应的关键字序列为 {K1，K2，…,Kn}。将这些记录重新排序为 {Ri1,Ri2,…,Rin}, 使得相应的关键字值满足条 Ki1<=Ki2<=…<=Kin, 这样的一种操作称为排序。**

通常来说，排序的目的是快速查找。

**衡量排序算法的优劣：**

时间复杂度：分析关键字的比较次数和记录的移动次数
空间复杂度：分析排序算法中需要多少辅助内存
稳定性：若两个记录 A 和 B 的关键字值相等，但排序后 A、B 的先后次序保持不变，则称这种排序算法是稳定的。

**排序算法分类：内部排序和外部排序。**

内部排序：整个排序过程不需要借助于外部存储器（如磁盘等），所有排序操作都在内存中完成。

外部排序：参与排序的数据非常多，数据量非常大，计算机无法把整个排序过程放在内存中完成，必须借助于外部存储器（如磁盘）。外部排序最常见的是多路归并排序。可以认为外部排序是由多次内部排序组成。



### 3.4.6 十大内部排序算法

**选择排序**
**直接选择排序、堆排序**
**交换排序**
**冒泡排序、快速排序**
**插入排序**
**直接插入排序、折半插入排序、Shell 排序**
**归并排序**
**桶式排序**
**基数排序**

### 3.4.7 算法的 5 大特征

| | |
|---|---|
| 输入（Input） | 有 0 个或多个输入数据，这些输入必须有清楚的描述和定义 |
| 输出（Output） | 至少有 1 个或多个输出结果，不可以没有输出结果 |
| 有穷性（有限性，Finiteness） | 算法在有限的步骤之后会自动结束而不会无限循环，<br>并且每一个步骤可以在可接受的时间内完成 |
| 确定性（明确性，Definiteness） | 算法中的每一步都有确定的含义，不会出现二义性 |
| 可行性（有效性，Effectiveness） | 算法的每一步都是清楚且可行的，能让用户用纸笔计算而求出答案 |

说明：满足确定性的算法也称为：确定性算法。现在人们也关注更广泛的概念，例如考虑各种非确定性的算法，如并行算法、概率算法等。另外，人们也关注并不要求终止的计算描述，这种描述有时被称为过程（procedure）。

### 3.4.8 冒泡排序

**冒泡排序的基本思想：通过对待排序序列从前向后，依次比较相邻元素的排序码，若发现逆序则交换，使排序码较大的元素逐渐从前部移向后部。**

**因为排序的过程中，各元素不断接近自己的位置，如果一趟比较下来没有进行过交换，就说明序列有序，因此要在排序过程中设置一个标志 swap 判断元素是否进行过交换。从而减少不必要的比较。**

**数组的冒泡排序的实现**

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

        int[] arr = new int[]{43,32,76,92,-65,85,71,-42};

        // 冒泡排序
        for(int i = 0;i < arr.length - 1;i++){

            for(int j = 0;j < arr.length - 1 - i;j++){

                if(arr[j] > arr[j+1]){
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }

        for(int i = 0;i < arr.length;i++){
            System.out.print(arr[i] + "\t");
        }
    }
}
```



### 3.4.9 快速排序

**快速排序（Quick Sort）由图灵奖获得者 Tony Hoare 发明，被列为 20 世纪十大算法之一，是迄今为止所有内排序算法中速度最快的一种。冒泡排序的升级版，交换排序的一种。快速排序的时间复杂度为 O(nlog(n))。**

**排序思想：**

1. 从数列中挑出一个元素，称为 " 基准 "（pivot）。
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。
4. 递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

**快速排序：**

通过一趟排序将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分关键字小，
则分别对这两部分继续进行排序，直到整个序列有序。

```
public class QuickSort {
    private static void swap(int[] data, int i, int j) {
        int temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }
    private static void subSort(int[] data, int start, int end) {
        if (start < end) {
            int base = data[start];
            int low = start;
            int high = end + 1;
            while (true) {
                while (low < end && data[++low] - base <= 0)
                    ;
                while (high > start && data[--high] - base >= 0)
                    ;
                if (low < high) {
                    swap(data, low, high);
                } else {
                    break;
                }
            }
            swap(data, start, high);
            subSort(data, start, high - 1); // 递归调用
            subSort(data, high + 1, end);
        }
    }
    public static void quickSort(int[] data){
        subSort(data,0,data.length-1);
    }

    public static void main(String[] args) {
        int[] data = { 9, -16, 30, 23, -30, -49, 25, 21, 30 };
        System.out.println(" 排  序  之  前: \n" + java.util.Arrays.
toString(data));
        quickSort(data);
        System.out.println(" 排  序  之  后: \n" + java.util.Arrays.
toString(data));
    }
}
```



## 3.4.10 排序算法性能对比

| 排序方法 | 时间复杂度（平均） | 时间复杂度（最坏） | 时间复杂度（最好） | 空间复杂度 | 稳定性 |
|---|---|---|---|---|---|
| 插入排序 | $O(n^2)$ | $O(n^2)$ | O(n) | O(1) | 稳定 |
| 希尔排序 | $O(n^{1.3})$ | $O(n^2)$ | O(n) | O(1) | 不稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | O(1) | 不稳定 |
| 堆排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | O(1) | 不稳定 |
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | O(n) | O(1) | 稳定 |
| 快速排序 | $O(n\log_2 n)$ | $O(n^2)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | 不稳定 |
| 归并排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | O(n) | 稳定 |
| 计数排序 | O(n+k) | O(n+k) | O(n+k) | O(n+k) | 稳定 |
| 桶排序 | O(n+k) | $O(n^2)$ | O(n) | O(n+k) | 稳定 |
| 基数排序 | O(n*k) | O(n*k) | O(n*k) | O(n+k) | 稳定 |

### 各种内部排序方法性能比较

1. 从平均时间而言：快速排序最佳。但在最坏情况下时间性能不如堆排序和归并排序。

2. 从算法简单性看：由于直接选择排序、直接插入排序和冒泡排序的算法比较简单，将其认为是简单算法。对于 Shell 排序、堆排序、快速排序和归并排序算法，其算法比较复杂，认为是复杂排序。

3. 从稳定性看：直接插入排序、冒泡排序和归并排序是稳定的；而直接选择排序、快速排序、Shell 排序和堆排序是不稳定排序。

4. 从待排序的记录数 n 的大小看，n 较小时，宜采用简单排序；而 n 较大时宜采用改进排序。

### 排序算法的选择

1. 若 n 较小 ( 如 $n \le 50$)，可采用直接插入或直接选择排序。当记录规模较小时，直接插入排序较好；否则因为直接选择移动的记录数少于直接插入，应选直接选择排序为宜。

2. 若文件初始状态基本有序 ( 指正序 )，则应选用直接插入、冒泡或随机的快速排序为宜；

3. 若 n 较大，则应采用时间复杂度为 O(nlgn) 的排序方法：快速排序、堆排序或归并排序。

# 3.5 Arrays 工具类的使用

**java.util.Arrays 类即为操作数组的工具类，包含了用来操作数组（比如排序和搜索）的各种方法。**

| | | |
|---|---|---|
| 1 | boolean equals(int[] a,int[] b) | 判断两个数组是否相等。 |
| 2 | String toString(int[] a) | 输出数组信息。 |
| 3 | void fill(int[] a,int val) | 将指定值填充到数组之中。 |
| 4 | void sort(int[] a) | 对数组进行排序。 |
| 5 | int binarySearch(int[] a,int key) | 对排序后的数组进行二分法检索指定的值。 |

```
public class ArrayTest4 {
	public static void main(String[] args) {

		//1.boolean equals(int[] a,int[] b) 判断两个数组是否相等。
		int[] arr1 = new int[]{1,2,3,4};
		int[] arr2 = new int[]{1,2,9,3};
		boolean isEquals = Arrays.equals(arr1, arr2);
		System.out.println(isEquals);

		//2.String toString(int[] a) 输出数组信息。
		System.out.println(Arrays.toString(arr1));
		//3.void fill(int[] a,int val) 将指定值填充到数组之中。
		Arrays.fill(arr1, 10);
		System.out.println(Arrays.toString(arr1));
		//4.void sort(int[] a) 对数组进行排序。
		Arrays.sort(arr2);
		System.out.println(Arrays.toString(arr2));

		//5.int binarySearch(int[] a,int key) 对排序后的数组进行二分法检索指定的值。
		int[] arr3 = new int[]{43,32,76,92,-65,85,71,-42};
		int index = Arrays.binarySearch(arr3, 210);
		if(index >= 0){
			System.out.println(index);
		}else{
			System.err.println(" 未找到。");
		}
	}
}
```



# 3.6 数组使用中的常见异常

数组中的常见异常：

1. 数组角标越界的异常：ArrayIndexOutOfBoundsException。
2. 空指针异常 :NullPointerException。

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

		//1. 数组角标越界的异常 :ArrayIndexOutOfBoundsException
		int[] arr = new int[]{1,2,3,4,5,6};

		// 错误 1:
//		for(int i = 0;i <= arr.length;i++){
//			System.out.println(arr[i]);
//		}

		// 错误 2:
//		System.out.println(arr[-2]);

		// 错误 3
//		System.out.println("hello");

		//2. 空指针异常 :NullPointerException
		// 情况一 :
//		int[] arr2= new int[]{1,2,3};
//		arr2 = null;
//		System.out.println(arr2[0]);
		// 情况二 :
//		int[][] arr2 = new int[4][];
//		System.out.println(arr2[0][0]);

		// 情况三 :
//		String[] arr3 = new String[]{"AA","QQ","YY","XX","TT","KK"};
//		arr3[0] = null;
//		System.out.println(arr3[0].toString());
	}
}
```

# 四　面向对象（上）

**Java 面向对象学习的三条主线：**

1.Java 类及类的成员：
　　属性，方法，构造器，代码块，内部类。

2. 面向对象的三大特征：
　　封装性，继承性，多态性。

3. 其他关键字：this、super、static、final、
　　abstract、interface、package、import 等。

## 4.1　面向过程 (POP) 与面向对象 (OOP)

**面向过程：Procedure Oriented Programming**

强调的是功能行为，以函数为最小单位，考虑怎么做。

**面向对象：Object Oriented Programming**

强调具备了功能的对象，以类 / 对象为最小单位，考虑谁来做。

```
人 {
    打开（冰箱）{
        冰箱，开门（）；
    } 操作（大象）{
        大象进入（冰箱）；
    } 关闭（冰箱）{
        冰箱，关门（）；
    }
}
```

```
冰箱 {
    开门（）{
    }
    关门（）{
    }
}

大象 {
    进入（冰箱）{
    }
}
```



## 4.2　类和对象

类：对一类事物的描述，是抽象的，概念上的定义。
对象：是实际存在的该类事物的每个个体，
　　因而也称为实例（instance）。
> 面向对象程序设计的重点是类的设计。
> 设计类，就是设计类的成员。

### 4.2.1　Java 类及类的成员

属性：对应类中的成员变量
行为：对应类中的成员方法

### 4.2.2　类与对象的创建及使用

**一、设计类，就是设计类的成员。**

属性 == 成员变量 == field == 域、字段
方法 == 成员方法 == 函数 == method
创建类的对象 == 类的实例化 == 实例化类

**二、类和对象的使用（面向对象思想落地的实现）。**

1. 创建类，设计类成员。
2. 创建类的对象。
3. 通过“对象．属性”或“对象．方法”调用对象的结构。

**三、如果创建了一个类的多个对象，则每个对象都独立拥有一套类的属性（非 static）。**

> 意味着：如果我们修改一个对象的属性 a，
　　则不影响另外一个对象属性 a 的值。

```java
public class PersonTest{
	public static void main(String[] args){
		// 创建 Person 类的对象
		// 创建对象语法：类名对象名 = new 类名 ();
		Person P1 = new Person;
		// 调用对象的结构：属性、方法
		// 调用属性：对象 . 属性
		P1.name = "Tom";
		P1.age = true;
		P1.isMale = true;
		System,out.println(P1.name);
		// 调用方法：对象 . 方法
		P1.eat();
		P1.isMale = true;
		P1.talk("Chinese");
		Person p2 = new Person();
		System.out.println(p2.name); //null
		System.out.println(p2.isMale);
		Person p3 = p1;
		System.out.println(p3.name);
		p3.age = 10;
		System.out.println(p1.age); //10
	}
}
class Person{
	// 属性
	String name;
	int age;
	boolean isMate;
	// 方法
	public void eat(){
		System,out.println(" 人可以吃饭 ");
	}
	public void sleep(){
		System,out.println(" 人可以睡觉 ");
	}
	public void talk(String language){
		System,out.println(" 人可以说话，使用的是：" + language);
	}
}
```

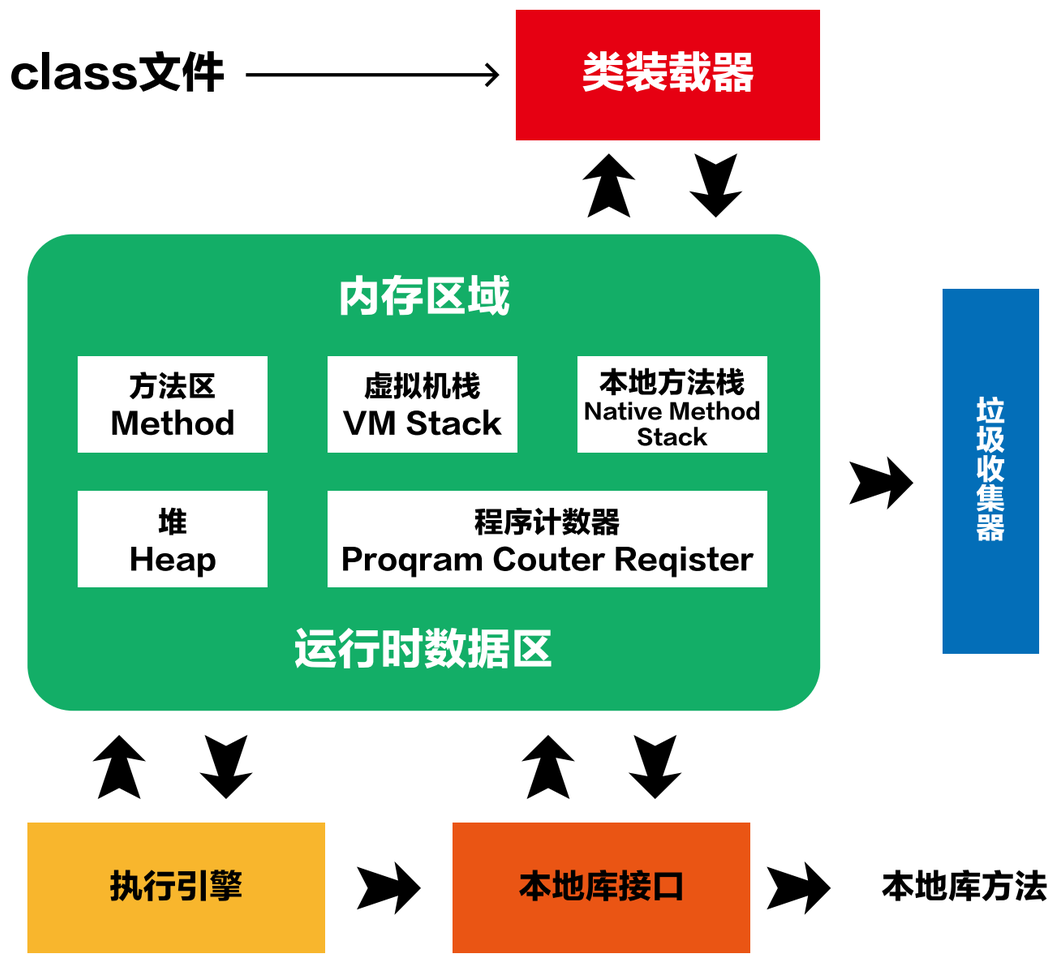


## 4.2.3 对象的创建和使用：内存解析

**堆（Heap）：**此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。这一点在 Java 虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配。

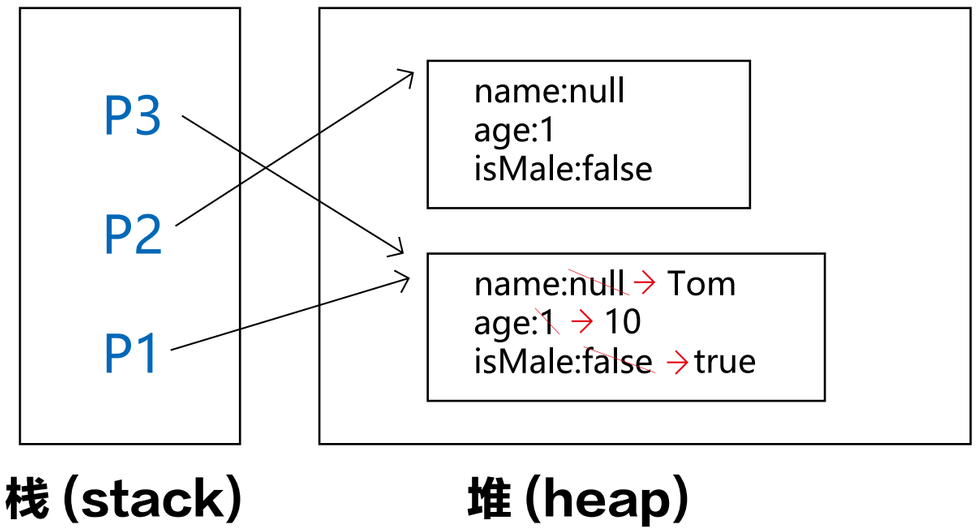
通常所说的**栈（Stack）**：是指虚拟机栈。虚拟机栈用于存储局部变量等。局部变量表存放了编译期可知长度的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不等同于对象本身，是对象在堆内存的首地址）。方法执行完，自动释放。

**方法区（MethodArea）**：用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。



## 对象的内存解析图

```
Person P1 = new Person();
P1.name = "Tom";
P1.isMale = true;
Person P2 = new Person();
System,out.println(P2.name);//null
Person P3 = P1;//Tom
P3.age = 10;
```






# 4.3 类的成员之一：属性

## 类中属性的使用

## 属性（成员变量） VS 局部变量

### 1. 相同点

① 定义变量的格式：数据类型 变量名 = 变量值；
② 先声明，后使用；
③ 变量都有其对应的作用域。

### 2. 不同点

① 在类中声明的位置不同：
属性：直接定义在类的一对 {} 内；
局部变量：声明在方法内、方法形参、代码块内、构造器内部的变量。

② 关于权限修饰符的不同：
属性：可以在声明属性是，指明其权限，使用权限修饰符。
常用的权限修饰符：private、public、缺省、protected -> 封装性
目前声明属性是都可以使用缺省
局部变量：不可以使用权限修饰符。

③ 默认初始化值的情况：
属性：类的属性，根据其类型都有默认初始化值：
整形（byte / short / int / long）：0
浮点型（float / double ）：0.0
字符型（char）：0 或（' /u0000 '）
布尔型（boolean）：false
引用数据类型（类、数组、接口）：null

局部变量：没有默认初始化值。
意味着：我们在调用局部变量之前，一定要显示赋值；
引用地：形参在调用时，赋值即可。

④ 在内存加载中的位置：
属性：加载到堆空间中（非 static）；
局部变量：加载到栈空间中。

```
public class UserTest{
	public ststic main(String[] args){
		User u1 = new User;
		System.out.println(u1.name);
		System.out.println(u1.age);
		System.out.println(u1.isMale);
		u1.talk(" 日语 ");
		u1.eat();
	}
}

// 客户类
class user{
	// 属性（或成员变量）
	String name;		// 不加 private 即为缺省
	public int age;		// 不加 public 即为缺省
	boolean isMale;

	//language：形参，也是局部变量
	public void talk(String language){
		System.out.println(" 我们使用 " + language + " 进行交流 ");
	}

	public void eat(){
		String food = " 烙饼 "; // 局部变量
		System,out.println(" 北方人喜欢吃 " + food) ;
	}
}
```



# 4.4 类的成员之二：方法

## 4.4.1 类中方法的声明和使用

**方法：描述类应该具有的功能**

比如：Math 类：sqrt() / random() / ...
Scanner 类：nextXxx() ...
Arrays 类：sort() / binarysearch() ...
tostring() / equals() / ...

**1. 举例：**

```
public void eat(){}
public void sleep(int hour){}
public String getName(){}
public String getNation(String netion){}
```

**2. 方法的声明：权限修饰符、返回值类型、方法名（形参列表）{ 方法体 }**

注意：static、final、abstract 来修饰的方法，后面再讲

**3. 说明：**

3.1 关于权限修饰符：默认方法的权限修饰符先都使用 public
Java 规定的 4 钟权限修饰符：private、public、缺省、protected

3.2 返回值类型：有返回值 VS 没有返回值
① 如果方法有返回值，则必须在方法声明时，指定返回值的类型。同时方法中需要使用 return 关键字来返回指定类型的变量或常量。
② 如果方法没有返回值，则方法声明时，使用 void 来表示。通常没有返回值的方法中，就不需要使用 return。但是如果使用的话，只能 "return; " 表示结束此方法的意思。
③ 定义方法该不该有返回值？
看题目要求；
凭经验，具体问题具体分析。

3.3 方法名：属于标识符，遵循标识符的规则和规范“ 见名知意 ”。

3.4 形参列表：方法可以声明 0 个，1 个或多个；
格式：数据类型 1，形参 1
数据类型 2，形参 2 ……

3.5 方法体：方法功能的体现。

## 4.return 关键字的使用：

4.1. 使用范围：使用在方法体中；
4.2. 作用：结束方法
针对有返回值类型的方法，使用 "return 数据 " 方法返回索要的数据。
4.3. 注意点：return 关键字后面不可以声明执行语句。

## 5. 方法的使用中，可以调用当前类的属性和方法。

特殊的：方法 A 中调用了方法 A：递归方法；
方法中不可以定义方法。

```
public class CustomerTest {
	public static void main(String[] args) {
		Customer cust1 = new Customer();
		cust1.eat();
		// 测试形参是否需要设置的问题
//		int[] arr = new int[]{3,4,5,2,5};
//		cust1.sort();
		cust1.sleep(8);
	}
}
// 客户类
class Customer{
	// 属性
	String name;
	int age;
	boolean isMale;
	// 方法
	public void eat(){
		System.out.println(" 客户吃饭 ");
		return;
		//return 后不可以声明表达式
//		System.out.println("hello");
	}
	public void sleep(int hour){
		System.out.println(" 休息了 " + hour + " 个小时 ");
		eat();
//		sleep(10);
	}
	public String getName(){
		if(age > 18){
			return name;
		}else{
			return "Tom";
		}
	}
	public String getNation(String nation){
		String info = " 我的国籍是： " + nation;
		return info;
	}
	public void info(){
		// 错误的
//		public void swim(){
//		}
	}
}
```

## 4.4.2 理解“万事万物皆对象”

1. 在 Java 语言范畴中，我们都将功能、结构等封装到类中，通过类的实例化，来调用具体的功能结构：
Scanner，String 等；
文件：File；
网络资源 URL。

2. 涉及到 Java 语言前段 Html，后端的数据库交互时，前后端的结构在 Java 层面交互时，都体现为类，对象。

### 4.4.3 内存解析的说明

引用类型的变量，只能储存两类值：
null 或地址值（含变量的类型）。

### 4.4.4 匿名对象的使用

1. 理解：我们创建对象，没有显示的赋值给一个变量名，即为匿名对象。
2. 匿名对象只能使用一次。

### 4.4.5 自定义数组的工具类

```
public class ArrayUtil{
    // 求数组的最大值
    public int getMax(int[] arr){
        int maxValue = arr[0];
        for(int i = 1; i < arr.length; i++){
            if(maxValue < arr[i]) {
                maxValue = arr[i];
            }
        }
        return maxValue;
}

// 求数组的最小值
public int getMin(int[] arr){
    int minValue = arr[0];
    for(int i = 1; i < arr.length; i++){
        if(maxValue > arr[i]) {
            minValue = arr[i];
        }
    }
    return minValue;
}
```



```
// 求数组的总和
public int getSum(int[] arr){
    int Sum = 0;
    for(int i = 0; i < arr.length; i++){
        Sum += arr[i];
        }
    }
    return Sum;
}
// 求数组的平均值
public int getAvg(int[] arr){
    int AvgValue = getSum(arr)/arr.length;
    return avgValue;
}

// 反转数组
public void reverse(int[] arr){
    for(int i = 0; i < arr.length / 2; i++){
        int temp = arr[i];
        arr[i] = arr[arr.length - i - 1];
        arr[arr.length - i - 1] = temp;
    }
}

// 复制数组
public int[] copy(int[] arr){
    int[] arr1 = new int[arr.length];
    for(int i = 0 ; i < arr.length ; i++){
        arr1[i] = arr[i];
    }
    return null;
}

// 数组排序
public void sort(int[] arr){
    for(int i = 0 ; i < arr.length - 1 ; i++){
        for(int j = 0 ; j < arr.length - 1 - i ; j++){
            if(arr[j] > arr[j + 1]){
                int temp = arr[j];
                arr[j] = arr[j + 1];
```

```
                        arr[j + 1] = temp;
                    }
                }
            }
        }
        // 遍历数组
        public void print(int[] arr){
            system.out.print("["
            for(int i = 0 ; i < arr.length ; i++){
                system.out.print(arr[i] + ",");
            }
            system,out.println("]");
        }
        // 查找指定元素
        public int getIndex(int[] arr , int dest){
            // 线性查找
            for(int i = 0 ; i < arr.length; i++){
                if(dest == arr[i]){
                    return i;
                }
            }
            return i - 1;
        }
    }
        // 测试类
    public class ArrayUtilTest{
        public static void main(String[] args){
            ArrayUtil util = new ArrayUtil( );
            int[] arr =new int[]{.....};
            int max = util.getMax(arr);
            system.out.println(" 最大值为：" + max);
            system.out.println(" 查找：") ;
            int index = util.getIndex(arr, 5);
            if(index > 0){
                system.out.println(" 找到了，索引地址：" + index) ;
            }else{
                system.out.println(" 没找到 ") ;
            }
        }
    }
```



## 4.4.6 方法的重载（over load）

1. 定义：在同一类中，允许存在一个以上的同名方法，只要他们的参数个数或者参数类型不同即可。

2. 举例：

Arrays 类中重载的 sort () / binarySearch()

3. 判断是否是重载：

跟方法的权限修饰符、返回值类型、形参变量名、方法体都没有关系！

两同一不同：

同一个类，同一个方法。

参数列表不同：参数个数不同、参数类型不同。

4. 通过对象调用方法时，如何确定某一个指定的方法：方法名 - - - -> 形参列表

```
public class OverTest{
    public static void main(String[] args){
        OverloadTest test = new OverloadTest();
        test.getSum(1,2);
    }
}

// 如下 4 个方法构成了重载
public void getSum(int i , int j ){
public void getSum(double d1 , double d2 ){
public void getSum(String s , int i ){
public void getSum(int i , String s ){
```

## 4.4.7 可变个数的形参

**JavaSE 5.0 中提供了 Varargs(variable number of arguments) 机制，允许直接定义能和多个实参相匹配的形参。从而，可以用一种更简单的方式，来传递个数可变的实参**

### 可变个数形参的方法

1.jdk 5.0 新增的内容。
2. 具体内容：
　　2.1 可变个数形参的格式：数据类型 … 变量名
　　2.2 当调用可变个数形参的方法时，传入的参数的个数可以是：0 个，1 个，2 个 ……
　　2.3 可变个数形参的方法与本类中方法名相同，形参不同的方法之间构成重载。
　　2.4 可变个数形参的方法与本类中方法名相同，形参类型也相同的数组之间不构成重载。即二者不能共存。
　　2.5 可变个数形参在方法钟的形参中，必须声明在末尾。
　　2.6 可变个数形参在方法中的形参中，最多只能声明一个可变形参。



```
public class MethodArgs {
	public static void main(String[] args) {
		MethodArgs test = new MethodArgs();
		test.show(12);
		// test.show("hell0");
		// test.show("hello","world");
		// test.show();
		test.show(new String[] { "AA", "BB", "CC" });
	}
	public void show(int i) {
	}

	// public void show(String s){
	// System.out.println("show(String)");
	// }
	public void show(String... strs) {
		System.out.println("show(String ...strs)");
		for (int i = 0; i < strs.length; i++) {
			System.out.println(strs[i]);
		}
	}

	// 此方法与上一方法不可共存
	// public void show(String[] strs){
	// }
	public void show(int i, String... strs) {
	}
}
```

## 4.4.8 方法参数的值传递机制

### 关于变量的赋值

**如果变量是基本数据类型，此时赋值的是变量所保存的数据值。**

**如果变量是引用数据类型，此时赋值的是变量所保存的数据的地址值。**

```
public class ValueTransferTest {
	public static void main(String[] args) {
		System.out.println("********** 基本数据类型：***********");
		int m = 10;
		int n = m;
		System.out.println("m = " + m + ", n = " + n);
		n = 20;
		System.out.println("m = " + m + ", n = " + n);

		System.out.println("*********** 引用数据类型 :********");
		Order o1 = new Order();
		o1.orderId = 1001;
		Order o2 = o1;// 赋值后，o1 和 o2 的地址值相同，都指向了
堆空间中同一个对象实体
		System.out.println("o1.orderId = " + o1.orderId + ",o2.
orderId = " + o2.orderId);
		o2.orderId = 1002;
		System.out.println("o1.orderId = " + o1.orderId + ",o2.
orderId = " + o2.orderId);
	}
}
class Order{
	int orderId;
}
```



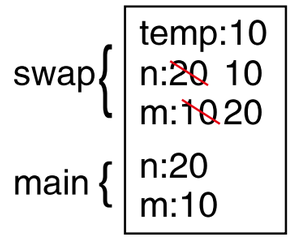
### 一、针对基本数据类型

**方法的形参的传递机制：值传递**

1. 形参：方法定义时，声明的小括号内的参数。
   实参：方法调用时，实际传递给形参的数据。

2. 值传递机制：
如果参数是基本数据类型，此时实参赋值给形参的是实参真是存储的数据值。

```
public class ValueTransferTest1 {
	public static void main(String[] args) {
		int m = 10;
		int n = 20;
		System.out.println("m = " + m + ", n = " + n);
		// 交换两个变量的值的操作
//		int temp = m;
//		m = n;
//		n = temp;
		ValueTransferTest1 test = new ValueTransferTest1();
		test.swap(m, n);
		System.out.println("m = " + m + ", n = " + n);
	}

	public void swap(int m,int n){
		int temp = m;
		m = n;
		n = temp;
	}
}
```



## 二、针对引用数据类型

**如果参数是引用数据类型，此时实参赋值给形参的是实参存储数据的地址值。**

```
public class ValueTransferTest2 {
	public static void main(String[] args) {
		Data data = new Data();
		data.m = 10;
		data.n = 20;
		System.out.println("m = " + data.m + ", n = " + data.n);

		// 交换 m 和 n 的值
//		int temp = data.m;
//		data.m = data.n;
//		data.n = temp;

		ValueTransferTest2 test = new ValueTransferTest2();
		test.swap(data);
		System.out.println("m = " + data.m + ", n = " + data.n);
	}

	public void swap(Data data){
		int temp = data.m;
		data.m = data.n;
		data.n = temp;
	}
}
class Data{
	int m;
	int n;
}
```



# 4.4.9 递归 (recursion) 方法

## 递归方法的使用 ( 了解 )

1. 递归方法：一个方法体内调用它自身。
2. 方法递归包含了一种隐式的循环，它会重复执行某段代码，但这种重复执行无须循环控制。
3. 递归一定要向已知方向递归，否则这种递归就变成了无穷递归，类似于死循环。

```
public class RecursionTest {
	public static void main(String[] args) {
		// 例 1: 计算 1-100 之间所有自然数的和
		// 方法 1:
		int sum = 0;
		for (int i = 1; i <= 100; i++) {
			sum += i;
		}
		System.out.println("sum = " + sum);
		// 方法 2:
		RecursionTest test = new RecursionTest();
		int sum1 = test.getSum(100);
		System.out.println("sum1 = " + sum1);
	}
	// 例 1: 计算 1-n 之间所有自然数的和
	public int getSum(int n) {
		if (n == 1) {
			return 1;
		} else {
			return n + getSum(n - 1);
		}	}
	// 例 2: 计算 1-n 之间所有自然数的乘积
	// 归求阶乘 (n!) 的算法
	public int getSum1(int n) {
		if (n == 1) {
			return 1;
		} else {
			return n * getSum1(n - 1);
		}
	}
}
```

# 4.5 面向对象特征之一：封装与隐藏

## 一、封装性的引入与体现

为什么需要封装？封装的作用和含义？
我要用洗衣机，只需要按一下开关和洗涤模式就可以了。有必要了解洗衣机内部的结构吗？有必要碰电动机吗？
我要开车，…

## 二、我们程序设计追求“高内聚，低耦合”。

高内聚：类的内部数据操作细节自己完成，不允许外部干涉；
低耦合：仅对外暴露少量的方法用于使用。

## 三、隐藏对象内部的复杂性，只对外公开简单的接口。

便于外界调用，从而提高系统的可扩展性、可维护性。通俗的说，把该隐藏的隐藏起来，该暴露的暴露出来。这就是封装性的设计思想。

## 问题的引入：

当我们创建一个类的对象以后，我们可以通过 " 对象 . 属性 " 的方式，对对象的属性进行赋值。这里，赋值操作要受到属性的数据类型和存储范围的制约。但除此之外，没有其他制约条件。但是，实际问题中，我们往往需要给属性赋值加入额外限制条件。这个条件就不能在属性声明时体现，我们只能通过方法进行条件的添加。比如说，setLegs 同时，我们需要避免用户再使用“对象 . 属性”的方式对属性进行赋值。则需要将属性声明为私有的 (private)
--》此时，针对于属性就体现了封装性。

## 封装性的体现：

我们将类的属性私有化 (private), 同时 , 提供公共的 (public) 方法来获取 (getXxx) 和设置 (setXxx)。

拓展：封装性的体现：① 如上
② 单例模式
③ 不对外暴露的私有方法



```
public class AnimalTest {
	public static void main(String[] args) {
		Animal a = new Animal();
		a.name = " 大黄 ";
//		a.age = 1;
//		a.legs = 4;//The field Animal.legs is not visible
		a.show();
//		a.legs = -4;
//		a.setLegs(6);
		a.setLegs(-6);
//		a.legs = -4;//The field Animal.legs is not visible
		a.show();
		System.out.println(a.name);
		System.out.println(a.getLegs());	}	}

class Animal{
	String name;
	private int age;
	private int legs; // 腿的个数
	// 对于属性的设置
	public void setLegs(int l){
		if(l >= 0 && l % 2 == 0){
			legs = l;
		}else{
			legs = 0;	}	}
	// 对于属性的获取
	public int getLegs(){
		return legs;
	}
	public void eat(){
		System.out.println(" 动物进食 ");	}
	public void show(){
		System.out.println("name = " + name + ",age = " + age +
",legs = " + legs);
	}
	// 提供关于属性 age 的 get 和 set 方法
	public int getAge(){
		return age;	}
	public void setAge(int a){
		age = a;	}	}
```

## 4.5.1 四种权限修饰符的理解与测试

**Java 权限修饰符 public、protected、default( 缺省 )、private 置于类的成员定义前，用来限定对象对该类成员的访问权限。**

| 修饰符 | 类内部 | 同一个包 | 不同包的子类 | 同一个工程 |
|---|---|---|---|---|
| private | yes | | | |
| (缺省) | yes | yes | | |
| protected | yes | yes | yes | |
| public | yes | yes | yes | yes |

对于 class 的权限修饰只可以用 public 和 default ( 缺省 )。

public 类可以在任意地方被访问。
default 类只可以被同一个包内部的类访问。

### 1、Order 类

**封装性的体现，需要权限修饰符来配合。**

1.Java 规定的 4 种权限：( 从小到大排序 )private、缺省、protected、public。

2.4 种权限用来修饰类及类的内部结构：属性、方法、构造器、内部类

3. 具体的, 4 种权限都可以用来修饰类的内部结构：属性、方法、构造器、内部类。

修饰类的话，只能使用：缺省、public。

总结封装性：Java 提供了 4 中权限修饰符来修饰类积累的内部结构，体现类及类的内部结构的可见性的方法。



```
public class Order {

	private int orderPrivate;
	int orderDefault;
	public int orderPublic;

	private void methodPrivate(){
		orderPrivate = 1;
		orderDefault = 2;
		orderPublic = 3;
	}

	void methodDefault(){
		orderPrivate = 1;
		orderDefault = 2;
		orderPublic = 3;
	}

	public void methodPublic(){
		orderPrivate = 1;
		orderDefault = 2;
		orderPublic = 3;
	}
}
```

### 2、OrderTest 类

```
public class OrderTest {
	public static void main(String[] args) {
		Order order = new Order();
		order.orderDefault = 1;
		order.orderPublic = 2;
		// 出了 Order 类之后，私有的结构就不可调用了
//order.orderPrivate = 3;//The field Order.orderPrivate is not visible
		order.methodDefault();
		order.methodPublic();
		// 出了 Order 类之后，私有的结构就不可调用了
//		order.methodPrivate();//The method methodPrivate() from the type Order is not visible
	}
}
```

**相同项目不同包的 OrderTest 类**

```
import github.Order;
public class OrderTest {
	public static void main(String[] args) {
		Order order = new Order();
		order.orderPublic = 2;

// 出了 Order 类之后，私有的结构、缺省的声明结构就不可调用了
//order.orderDefault = 1;
//order.orderPrivate = 3;
//The field Order.orderPrivate is not visible

		order.methodPublic();
// 出了 Order 类之后，私有的结构、缺省的声明结构就不可调用了
//order.methodDefault();
//order.methodPrivate();
//The method methodPrivate() from the type Order is not visible
	}
}
```



## 4.5.2 封装性的练习

**1. 创建程序 , 在其中定义两个类：Person 和 PersonTest 类。**
**定义如下：用 setAge() 设置人的合法年龄 (0~130)，用 getAge() 返回人的年龄。**

```
public class Person {
	private int age;
	public void setAge(int a){
		if(a < 0 || a > 130){
//			throw new RuntimeException(" 传入的数据据非法 ");
			System.out.println(" 传入的数据据非法 ");
			return;
		}
		age = a;
	}
	public int getAge(){
		return age;
	}
	// 绝对不能这样写！！！
	public int doAge(int a){
		age = a;
		return age;
	}
}
```

**3、测试类**
**在 PersonTest 类中实例化 Person 类的对象 b,**
**调用 setAge() 和 getAge() 方法，体会 Java 的封装性。**

```
public class PersonTest {
	public static void main(String[] args) {
		Person p1 = new Person();
//		p1.age = 1;	// 编译不通过
		p1.setAge(12);
		System.out.println(" 年龄为 :" + p1.getAge());
	}
}
```

# 4.6 类的成员之三：构造器（构造方法、constructor）

## 4.6.1 构造器的理解

类的成员之三：构造器（构造方法、constructor）的使用

constructor;

一、构造器的作用：

1. 创建对象
2. 初始化对象的属性

二、说明

1. 如果没有显示的定义类的构造器的话，则系统默认提供一个空参的构造器。 声明了构造器，就不会提供空参构造器了
2. 定义构造器的格式：

   权限修饰符 类名（形参列表）{}
3. 一个类中定义的多个构造器，彼此构成重载。
4. 一旦显示的定义了类的构造器之后，系统不再提供默认的空参构造器。
5. 一个类中，至少会有一个构造器。



```java
public class PersonTest {
	public static void main(String[] args) {
		// 创建类的对象 :new + 构造器
		Person p = new Person();//Person() 这就是构造器
		p.eat();
		Person p1 = new Person("Tom");
		System.out.println(p1.name);
	}
}

class Person{
	// 属性
	String name;
	int age;

	// 构造器
	public Person(){
		System.out.println("Person().....");
	}
	public Person(String n){
		name = n;
	}
	public Person(String n,int a){
		name = n;
		age = a;
	}

	// 方法
	public void eat(){
		System.out.println(" 人吃饭 ");
	}
	public void study(){
		System.out.println(" 人学习 ");
	}
}
```

## 4.6.2 总结属性赋值的过程

总结：属性赋值的先后顺序
① 默认初始化值；
② 显式初始化；
③ 构造器中赋值；
④ 通过 " 对象 . 方法 " 或 “ 对象 . 属性 ” 的方式，赋值

以上操作的先后顺序：① - ② - ③ - ④

```
public class UserTest {
	public static void main(String[] args) {
		User u = new User();
		System.out.println(u.age);
		User u1 = new User(2);
		u1.setAge(3);
		System.out.println(u1.age);
	}
}

class User{
	String name;
	int age = 1;
	public User(){
	}
	public User(int a){
		age = a;
	}
	public void setAge(int a){
		age = a;
	}
}
```



## 4.6.3 JavaBean 的使用

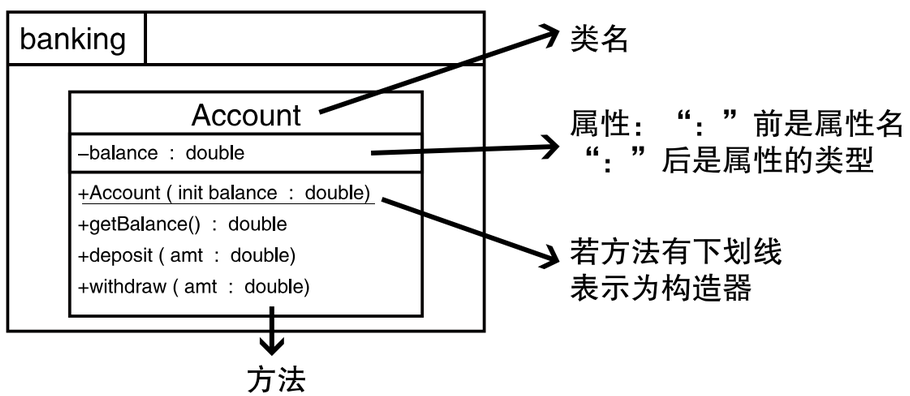
JavaBean 是一种 Java 语言写成的可重用组件。
所谓 javaBean，是指符合如下标准的 Java 类：
> 类是公共的。
> 有一个无参的公共的构造器。
> 有属性，且有对应的 get、set 方法。

```
public class Customer {
	private int id;
	private String name;
	public Customer(){
	}
	public void setId(int i){
		id = i;
	}
	public int getId(){
		return id;
	}
	public void setName(String n){
		name = n;
	}
	public String getName(){
		return name;
	}
}
```

## 4.6.4 UML 类图

+ 表示 public 类型， - 表示 private 类型，# 表示 protected 类型

方法的写法 : 方法的类型 (+、 -) 方法名 ( 参数名：参数类型 )：返回值类型




# 4.7 关键字：this

## 4.7.1 this 调用属性、方法、构造器

### this 关键字的使用

1.this 用来修饰、调用：属性、方法、构造器
2.this 修饰属性和方法 :
this 理解为：当前对象 , 或当前正在创建的对象。
2.1 在类的方法中，我们可以使用 "this. 属性 " 或 "this. 方法 " 的方式，调用当前对象属性或方法。
通常情况下，我们都选择省略 “ this. ” 。特殊情况下，如果方法的形参和类的属性同名，我们必须显式的使用 "this. 变量 " 的方式，表明此变量是属性，而非形参。
2.2 在类的构造器中，我们可以使用 "this. 属性 " 或 "this. 方法 " 的方式，调用正在创建的对象属性和方法。
但是，通常情况下，我们都选择省略 “ this. ” 。特殊情况下，如果构造器的形参和类的属性同名，我们必须显式的使用 "this. 变量 " 的方式，表明此变量是属性，而非形参。



3.this 调用构造器
① 我们可以在类的构造器中，显式的使用 " this( 形参列表 ) " 的方式，调用本类中重载的其他的构造器！
② 构造器中不能通过 "this ( 形参列表 ) " 的方式调用自己。
③ 如果一个类中有 n 个构造器，则最多有 n -1 个构造器中使用了 "this ( 形参列表 ) "。
④ 规定： " this( 形参列表 ) " 必须声明在类的构造器的首行！
⑤ 在类的一个构造器中，最多只能声明一个 "this( 形参列表 )"，用来调用其他构造器。

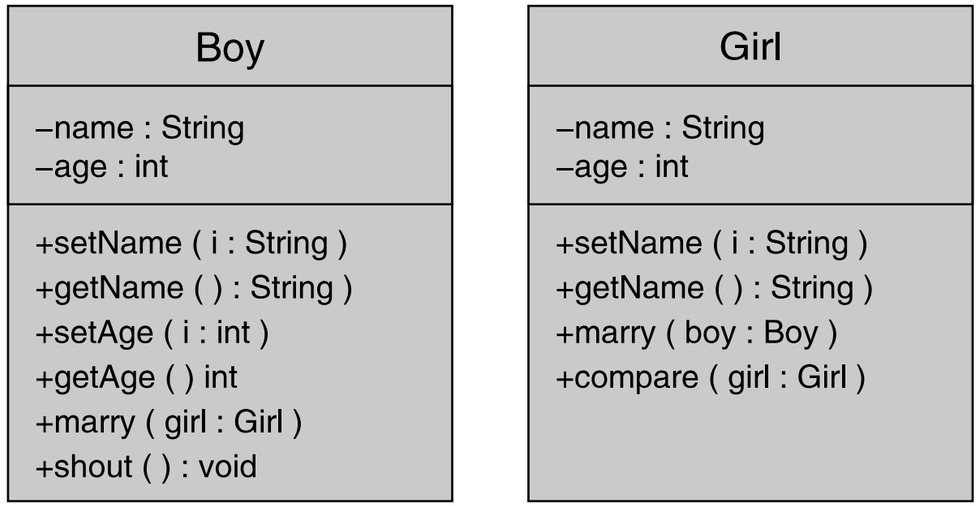
```
public class PersonTest {
	public static void main(String[] args) {
		Person p1 = new Person();
		p1.setAge(1);
		System.out.println(p1.getAge());
		p1.eat();
		System.out.println();
		Person p2 = new Person("jerry" ,20);
		System.out.println(p2.getAge());
	}
}

class Person{
	private String name;
	private int age;
	public Person(){
		this.eat();
		String info = "Person 初始化时，需要考虑如下的 1,2,3,4...( 共 40 行代码 )";
		System.out.println(info);
	}
	public Person(String name){
		this();
		this.name = name;
	}
	public Person(int age){
		this();
		this.age = age;
	}
```

```
    public Person(String name,int age){
        this(age); // 调用构造器的一种方式
        this.name = name;
//      this.age = age;
    }
    public void setNmea(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return this.age;
    }
    public void eat(){
        System.out.println(" 人吃饭 ");
        this.study();
    }
    public void study(){
        System.out.println(" 学习 ");
    }
}
```



## 4.7.2 this 的练习

添加必要的构造器，综合应用构造器的重载，this 关键字




### 1、Boy 类

```
public class Boy {
    private String name;
    private int age;
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
    public Boy(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public void marry(Girl girl){
        System.out.println(" 我想娶 " + girl.getName());
    }
```

```
    public void shout(){
        if(this.age >= 22){
            System.out.println(" 可以考虑结婚 ");
        }else{
            System.out.println(" 好好学习 ");
        }
    }
}
```

## 2、Girl 类

```
public class Girl {
    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Girl(){
    }
    public Girl(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public void marry(Boy boy){
        System.out.println(" 我想嫁给 " + boy.getName());
    }
// 比较两个对象的大小
    public int compare(Girl girl){
//        if(this.age >girl.age){
//            return 1;
//        }else if(this.age < girl.age){
//            return -1;
//        }else{
//            return 0;
//        }
        return this.age - girl.age;
    }
}
```



## 3、测试类

```
public class BoyGirlTest {
    public static void main(String[] args) {
        Boy boy = new Boy(" 罗密欧 ",21);
        boy.shout();

        Girl girl = new Girl(" 朱丽叶 ", 18);
        girl.marry(boy);

        Girl girl1 = new Girl(" 祝英台 ", 19);
        int compare = girl.compare(girl1);
        if(compare > 0){
            System.out.println(girl.getName() + " 大 ");
        }else if(compare < 0){
            System.out.println(girl1.getName() + " 大 ");
        }else{
            System.out.println(" 一样的 ");
        }
    }
}
```

## 4、Account 类

```
public class Account {
    private int id; // 账号
    private double balance; // 余额
    private double annualInterestRate; // 年利率
    public void setId(int id) {
    }
    public double getBalance() {
        return balance;
    }
    public void setBalance(double balance) {
        this.balance = balance;
    }
    public double getAnnualInterestRate() {
        return annualInterestRate;
    }
    public void setAnnualInterestRate(double annualInterestRate) {
        this.annualInterestRate = annualInterestRate;
    }
```

```
    public int getId() {
        return id;
    }
    public void withdraw(double amount) { // 取钱
        if(balance < amount){
            System.out.println(" 余额不足，取款失败 ");
            return;
        }
        balance -= amount;
        System.out.println(" 成功取出 " + amount);
    }
    public void deposit(double amount) { // 存钱
        if(amount > 0){
            balance += amount;
            System.out.println(" 成功存入 " + amount);
        }
    }
    public Account(int id, double balance, double
annualInterestRate) {
        this.id = id;
        this.balance = balance;
        this.annualInterestRate = annualInterestRate;
    }
}
```

## 5、Customer 类

```
public class Customer {
    private String firstName;
    private String lastName;
    private Account account;
    public Customer(String f, String l) {
        this.firstName = f;
        this.lastName = l;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
```



```
    public Account getAccount() {
        return account;
    }
    public void setAccount(Account account) {
        this.account = account;
    }
}
```

## 6、CustomerTest 类

写一个测试程序。
（1）创建一个 Customer，名字叫 Jane Smith, 他有一个账号为 1000，余额为 2000 元，年利率为 1.23%的账户。
（2）对 Jane Smith 操作。存入 100 元，再取出 960 元。再取出 2000 元。
打印出 Jane Smith 的基本信息
成功存入：100.0
成功取出：960.0
余额不足，取款失败
Customer [Smith, Jane] has a account: id is 1000, annualInterestRate is 1.23% , balance is 1140.0

```
public class CustomerTest {
    public static void main(String[] args) {
        Customer cust = new Customer("Jane" , "Smith");
        Account acct = new Account(1000,2000,0.0123);
        cust.setAccount(acct);
        cust.getAccount().deposit(100); // 存入 100
        cust.getAccount().withdraw(960); // 取钱 960
        cust.getAccount().withdraw(2000); // 取钱 2000
        System.out.println("Customer[" + cust.getLastName()
+ cust.getFirstName() + "] has a account: id is "+ cust.
getAccount().getId() + ",annualInterestRate is " + cust.getAccount().
getAnnualInterestRate() * 100 + "%, balance is "+ cust.
getAccount().getBalance());
    }
}
```

## 7、Account 类

```
public class Account {
	private double balance;
	public double getBalance() {
		return balance;
	}
	public Account(double init_balance){
		this.balance = init_balance;
	}
	// 存钱操作
	public void deposit(double amt){
		if(amt > 0){
			balance += amt;
			System.out.println(" 存钱成功 ");
		}
	}
	// 取钱操作
	public void withdraw(double amt){
		if(balance >= amt){
			balance -= amt;
			System.out.println(" 取钱成功 ");
		}else{
			System.out.println(" 余额不足 ");
		}
	}
}
```

## 8、Customer 类

```
public class Customer {
	private String firstName;
	private String lastName;
	private Account account;
	public String getFirstName() {
		return firstName;
	}
	public String getLastName() {
		return lastName;
	}
	public Account getAccount() {
		return account;
	}
	public void setAccount(Account account) {
		this.account = account;
	}
	public Customer(String f, String l) {
		this.firstName = f;
		this.lastName = l;
	}
}
```

## 9、Bank 类

```
public class Bank {
	private int numberOfCustomers;// 记录客户的个数
	private Customer[] customers;// 存放多个客户的数组
	public Bank(){
		customers = new Customer[10];
	}
	// 添加客户
	public void addCustomer(String f,String l){
		Customer cust = new Customer(f,l);
//		customers[numberOfCustomers] = cust;
//		numberOfCustomers++;
		customers[numberOfCustomers++] = cust;
	}
	// 获取客户的个数
	public int getNumberOfCustomers() {
		return numberOfCustomers;
	}
	// 获取指定位置上的客户
	public Customer getCustomers(int index) {
//		return customers;	// 可能报异常
		if(index >= 0 && index < numberOfCustomers){
			return customers[index];
		}
		return null;
	}
}
```



## 10、BankTest 类

```
public class BankTest {
	public static void main(String[] args) {
		Bank bank = new Bank();
		bank.addCustomer("Jane", "Smith");
		bank.getCustomers(0).setAccount(new Account(2000));
		bank.getCustomers(0).getAccount().withdraw(500);
		double balance = bank.getCustomers(0).getAccount().
getBalance();
		System.out.println(" 客户 : " + bank.getCustomers(0).
getFirstName() + " 的账户余额为 : " + balance);
		System.out.println("****************************");
		bank.addCustomer(" 万里 ", " 杨 ");
		System.out.println(" 银行客户的个数为 : " + bank.
getNumberOfCustomers());
	}
}
```

# 4.8 关键字：package、import

## 4.8.1 关键字—package

### 一、package 关键字的使用

1. 为了更好的实现项目中类的管理，提供包的概念。
2. 使用 package 声明类或接口所属的包，声明在源文件的首行。
3. 包：属于标识符，遵循标识符的命名规则、规范 " 见名知意 "。
4. 每 “ . ” 一次 , 就代表一层文件目录。

补充 : 同一个包下，不能命名同名接口或同名类。
不同包下，可以命名同名的接口、类。

### JDK 中主要的包介绍

1.java.lang - - - - 包含一些 Java 语言的核心类，如 String、Math、Integer、System 和 Thread，提供常用功能
2.java.net - - - - 包含执行与网络相关的操作的类和接口。
3.java.io - - - - 包含能提供多种输入 / 输出功能的类。
4.java.util - - - - 包含一些实用工具类，如定义系统特性、接口的集合框架类、使用与日期日历相关的函数。
5.java.text - - - - 包含了一些 java 格式化相关的类
6.java.sql - - - - 包含了 java 进行 JDBC 数据库编程的相关类 / 接口
7.java.awt - - - - 包含了构成抽象窗口工具集（abstractwindowtoolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面 (GUI)。
B/S C/S

## 4.8.2 MVC 设计模式

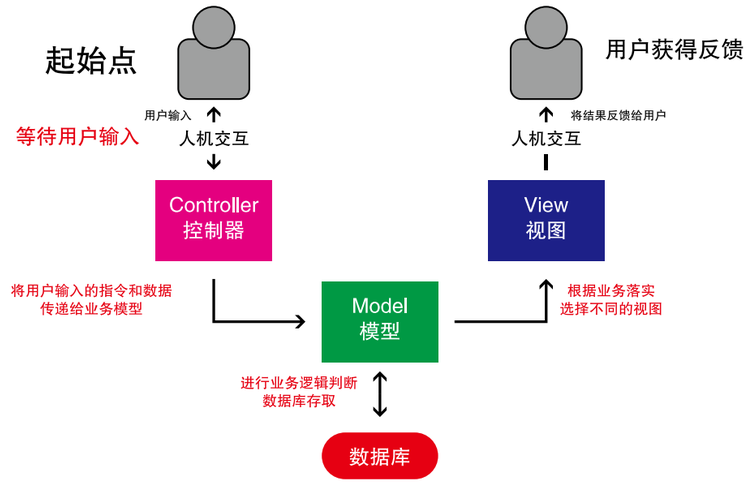
MVC 是常用的设计模式之一，将整个程序分为三个层次：视图模型层，控制器层，数据模型层。这种将程序输入输出、数据处理，以及数据的展示分离开来的设计模式使程序结构变的灵活而且清晰，同时也描述了程序各个对象间的通信方式，降低了程序的耦合性。

模型层 model 主要处理数据
> 数据对象封装 model.bean/domain
> 数据库操作类 model.dao
> 数据库 model.db

视图层 view 显示数据
> 相关工具类 view.utils
> 自定义 view view,ui

控制层 controller 处理业务逻辑
> 应用界面相关 controller.activity
> 存放 fragment controller.fragment
> 显示列表的适配器 controller.adapter
> 服务相关的 controller.service
> 抽取的基类 controller.base






## 4.8.3 关键字—import

### 二、import 关键字的使用

import: 导入
1. 在源文件中显式的使用 import 结构导入指定包下的类、接口。
2. 声明在包的声明和类的声明之间。
3. 如果需要导入多个结构，则并列写出即可。
4. 可以使用 "xxx.*" 的方式 , 表示可以导入 xxx 包下的所有结构。
5. 如果导入的类或接口是 java.lang 包下的，或者是当前包下的，则可以省略此 import 语句。
6. 如果在代码中使用不同包下的同名的类。那么就需要使用类的全类名的方式指明调用的是哪个类。
7. 如果已经导入 java.a 包下的类。那么如果需要使用 a 包的子包下的类的话，仍然需要导入。
8.import static 组合的使用：调用指定类或接口下的静态的属性或方法。

```
public class PackageImportTest {
	public static void main(String[] args) {
		String info = Arrays.toString(new int[]{1,2,3});
		Bank bank = new Bank();
		ArrayList list = new ArrayList();
		HashMap map = new HashMap();
		Scanner s = null;
		System.out.println("hello");
		UserTest us = new UserTest();
	}
}
```

# 项目二

模拟实现基于文本界面的《客户信息管理软件》。
该软件能够实现对客户对象的插入、修改和删除（用数组实现），并能够打印客户明细表。
项目采用分级菜单方式。主菜单如下：

---------- 客户信息管理软件 ----------

1 添加客户
2 修改客户
3 删除客户
4 客户列表
5 退　　出

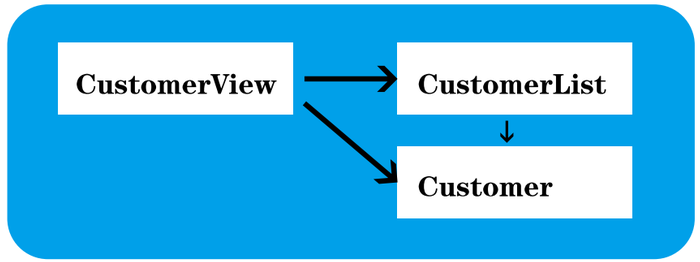
请选择（1-5）：__

需求说明

每个客户的信息被保存在 Customer 对象中。
以一个 Customer 类型的数组来记录当前所有的客户。
每次“ 添加客户 ”（菜单 1）后，客户（Customer）对象呗添加到数组中。
每次“ 修改客户 ”（菜单 2）后，修改的客户（Customer）对象替换数组中原对象。
每次“ 删除客户 ”（菜单 3）后，客户（Customer）对象被从数组中删除。
执行“ 客户列表 ”（菜单 4）时，将列出数组中所有客户的信息。



## 软件架构




| | |
|---|---|
| **CustomerView** | **为主模块，负责菜单的显示和处理用户操作** |
| **CustomerList** | **为Cusotmer对象的管理模块，内部用数组管理一组Customer 对象，并提供相应的添加、修改、删除和遍历方法，供CustomerView调用** |
| **Customer** | **为实体对象，用来封装客户信息** |

## 第一步：Customer 类的设计

Customer 为实体类，用来封装客户信息。
该类封装客户的以下信息：

> String name: 客户姓名
> char gender: 性别
> int age: 年龄
> String phone: 电话号码
> String email: 电子邮箱

提供各属性的 get / set 方法
提供所需的构造器（可自行确定）

按照设计要求编写 Customer 类，并编译。
在 Customer 类中临时添加一个 main 方法中，作为单元测试方法。
在方法中创建 Customer 对象，并调用对象的各个方法，以测试该类是否编写正确。

## 第二步：CustomerList 类的设计

CustomerList 为 Customer 对象的管理模块，内部使用数组管理一组 Customer 对象

本类封装以下信息：

> Customer[] customers: 用来保存客户对象的数组
> int total = 0 : 记录已保存客户对象的数量

该类至少提供以下构造器和方法：

> public CustomerList(int totalCustomer)
> public boolean addCustomer(Customer customer)
> public boolean replaceCustomer(int index, Customer cust)
> public boolean deleteCustomer(int index)
> public Customer[] getAllCustomer()
> public Customer getCustomer(int index)
> public int getTotal()

> public CustomerList(int totalCustomer)
    > 用途：构造器；用来初始化 customer 数组
    > 参数：totalCustomer；指定 customer 数组的最大空间
> public boolean addCustomer(Customer customer)
    > 用途：将参数 customer 添加组中最后一个客户对象记录之后
    > 参数：customer 指定要添加的客户对象
    > 返回：添加成功返回 true; false 表示数组已满，无法添加
> public boolean replaceCustomer(int index, Customer cust)
    > 用途：用参数 customer 替换数组中由 index 指定的对象
    > 参数：customer 指定的新客户对象
        index 指定所替换对象在数组中的位置，从 0 开始
    > 返回：替换成功返回 true; false 表示索引无效，无法替换
> public boolean deleteCustomer(int index)
    > 用途：用参数 customer 删除数组中由 index 指定的对象
    > 参数：customer 指定的删除客户的对象
    > 返回：删除成功返回 true; false 表示删除失败
> public Customer[] getAllCustomer()
    > 用途：获取所有的客户信息
> public Customer getCustomer(int index)
    > 用途：获取指定索引位置上的客户
> public int getTotal()
    > 用途：获取储存客户的数量



1. 按照设计要求编写 CustomerList 类，并编译。
2. 在 CustomerList 类中临时添加一个 main 方法中，作为单元测试方法。
3. 在方法中创建 CustomerList 对象（最多存放 5 个客户对象），然后分别用模拟数据调用以下各方法，以测试方法是否编写正确：

> addCustomer()
> replaceCustomer()
> deleteCustomer()
> getAllCustomer()
> getCustomer()
> getTotal()

## 第三步：CustomerView 类的设计

CustomerView 为主模块，负责菜单的显示和处理用户操作

本类封装一下信息：

> CustomerList customerlist = new CustomerList(10)

创建最大包含 10 个客户对象的 CustomerList 对象，供以下各成员方法使用。

该类至少提供以下方法：

> public void enterMainMenu()
> private void addNewCustomer()
> private void modifyCustomer()
> private void deleteCustomer()
> private void listAllCustomer()
> public static void main(String[] args)

# CMUtility 工具类

```
package guanli.Util;
import java.util.Scanner;

/**
 * CMUtility 工具类， 将不同的功能封装为方法，就是可以直接通过调用方法使用它的
 * 功能，而无需考虑具体的功能细节。
 *
 * @author Administrator
 *
 */
public class CMUtility {
	private static Scanner scanner = new Scanner(System.in);

	public static char readMenuSelection() {
		char c;
		for (;;) {
			String str = readKeyBoard(1, false);
			c = str.charAt(0);
			if (c != '1' && c != '2' && c != '3' && c != '4' && c !=
'5') {
				System.out.println(" 选择错误，请重新输入： ");
			} else
				break;
		}
		return c;
	}

	// 从键盘读取一个字符，并将其作为方法的返回值。
	public static char readChar() {
		String str = readKeyBoard(1, false);
		return str.charAt(0);
	}

	// 从键盘读取一个字符，并将其作为方法的返回值。
	// 如果用户不输入字符而直接回车，方法将以 defaultValue 作为返回值。
	public static char readChar(char defaultValue) {
		String str = readKeyBoard(1, true);
		return (str.length() == 0) ? defaultValue : str.charAt(0);
	}
```



```
	// 从键盘读取一个长度不超过 2 位的整数，并将其作为方法的返回值。
	public static int readInt() {
		int n;
		for (;;) {
			String str = readKeyBoard(2, false);
			try {
				n = Integer.parseInt(str);
				break;

			} catch (NumberFormatException e) {
				System.out.println(" 数字输入错误，请重新输入： ");
			}
		}
		return n;
	}
	// 从键盘读取一个长度不超过 2 位的整数，并将其作为方法的返回值。
	// 如果用户不输入字符而直接回车，方法将以 defaultValue 作为返回值
	public static int readInt(int defaultValue) {
		int n;
		for (;;) {
			String str = readKeyBoard(2, true);
			if (str.equals("")) {
				return defaultValue;
			}

			try {
				n = Integer.parseInt(str);
				break;
			} catch (NumberFormatException e) {
				System.out.println(" 数字输入错误，请重新输入： ");
			}
		}
		return n;
	}

	// 从键盘读取一个长度不超过 limit 的字符串，并将其作为方法的返回值。
	public static String readString(int limit) {
		return readKeyBoard(limit, false);
	}
	// 从键盘读取一个长度不超过 limit 的字符串，并将其作为方法的返回值。
```

```
// 如果用户不输入字符而直接回车，方法将以 defaultValue 作为返回值。
public static String readString(int limit, String defaultValue) {
        String str = readKeyBoard(limit, true);
        return str.equals("") ? defaultValue : str;
}
// 用于确认选择的输入，该方法从键盘读取 'Y' 或 'N'，并将其作为方法的返回值。
public static char readConfirmSelection() {
        char c;
        for (;;) {
                String str = readKeyBoard(1, false).toUpperCase();
                c = str.charAt(0);
                if (c == 'Y' || c == 'N') {
                        break;
                } else {
                        System.out.println(" 选择错误，请重新输入： ");
                }
        }
        return c;
}
public static String readKeyBoard(int limit, boolean
        }
        public static String readKeyBoard(int limit, boolean
blankReturn) {
                String line = "";
                while (scanner.hasNextLine()) {
                        line = scanner.nextLine();
                        if (line.length() == 0) {
                                if (blankReturn)
                                        return line;
                                else
                                        continue;
                        }
                        if (line.length() < 1 || line.length() > limit) {
                                System.out.println("输入长度（不大于" + limit + "）
错误，请重新输入 ");
                                continue;
                        }
                        break;
                }
                return line;
        }
}
```



## Customer 模块

```
package guanli.bean;
public class Customer {
        private String name;
        private char gender;
        private int age;
        private String phone;
        private String email;

        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public char getGender() {
                return gender;
        }
        public void setGender(char gender) {
                this.gender = gender;
        }
        public int getAge() {
                return age;
        }
        public void setAge(int age) {
                this.age = age;
        }
        public String getPhone() {
                return phone;
        }
        public void setPhone(String phone) {
                this.phone = phone;
        }
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
```

```
	public Customer() {
		super();
	}
	public Customer(String name, char gender, int age, String
phone, String email) {
		super();
		this.name = name;
		this.gender = gender;
		this.age = age;
		this.phone = phone;
		this.email = email;
	}

}
```



## CustomerList 模块

```
package guanli.service;

import guanli.bean.Customer;

/**
 * CustomerList 为 Customer 对象的管理模块，内部用数组管理一组
 * Customerdvxd , 并提供相应的添加、修改、删除、遍历方法，提供
 * CustomerView 调用
 * @author Administrator
 */
public class CustomerList {
	private Customer[] customers;// 用来保存客户对象的数组
	private int total = 0;// 记录已保存客户对象的数量

	/*
	 * 用途：构造器 : 用来初始化 customer 数组
	 * 参数：total: 指定 customer 数组的最大空间
	 */
public CustomerList(int totalCustomer) {
	customers = new Customer[totalCustomer];
}
/**
 * 用途：将指定的客户添加到数组中
 * @param customer
 * @return true：添加成功；false：添加失败
 */
public boolean addCustomer(Customer customer) {
	if(total >= customers.length) {
		return false;
	}

	//customers[total++] = customer;
	customers[total] = customer;
	total++;
	return true;
}
```

```
/**
 * 用途：修改指定索引位置的客户信息
 * @param index
 * @param cust
 * @return true：修改成功；false：修改失败
 */
public boolean replaceCustomer(int index, Customer cust) {
        if(index < 0 || index >= total) {
                return false;
        }
        customers[index] = cust;
        return true;
}

/**
 * 用途：删除指定索引位置的客户信息
 * @param index
 * @return true：删除成功；false：删除失败
 */
public boolean deleteCustomer(int index) {
        if(index < 0 || index >= total) {
                return false;
        }
        for(int i = index; i < total - 1; i++) {
                customers[i] = customers[i + 1];
        }
        // 最后位置需要置空
        //customers[--total] = null;
        customers[total - 1] = null;
        total--;
        return true;
}
```



```
/**
 * 用途：获取所有客户的信息
 * @return
 */
public Customer[] getAllCustomer(){
        Customer[] custs = new Customer[total];
        for(int i = 0; i < total ; i++) {
                custs[i] = customers[i];
        }
        return custs;
}
/**
 * 用途：获取指定索引位置上的客户信息
 * @param index
 * @return 如果找到了信息，则返回，如果没找到，则返回 null
 */
public Customer getCustomer(int index) {
        if(index < 0 || index >= total) {
                return null;
        }
        return customers[index];

}

/**
 * 用途：获取存储客户的数量
 * @return
 */
public int getTotal() {
        return total;
}

}
```

## CustomerView 模块

```
package guanli.UI;

import guanli.Util.CMUtility;
import guanli.bean.Customer;
import guanli.service.CustomerList;

public class CustomerView {
	private CustomerList customerList = new CustomerList(10);

	public CustomerView() {
		Customer customer = new Customer(" 张三 ", ' 男 ', 30,
"000-0000000", "email@qq.com");
		customerList.addCustomer(customer);
	}

	/**
	 * 显示《客户信息管理软件》界面的方法
	 */
	public void enterMainMenu() {

		boolean isFlag = true;
		while (isFlag) {

			System.out.println("\n---------------------- 客户信息管理
软件 ----------------------");
			System.out.println("                    1. 添加客户 ");
			System.out.println("                    2. 修改客户 ");
			System.out.println("                    3. 删除客户 ");
			System.out.println("                    4. 客户列表 ");
			System.out.println("                    5. 退    出 \n");
			System.out.print("                   请选择（1-5）：");

			char menu = CMUtility.readMenuSelection();

			switch (menu) {
			case '1':
				addNewCustoner();
				break;
```



```
			case '2':
				modifyCustomer();
				break;
			case '3':
				deleteCustomer();
				break;
			case '4':
				listAllCustomer();
				break;
			case '5':
				System.out.println(" 确认是否退出（Y/N）：");
				char isExit = CMUtility.readConfirmSelection();
				if (isExit == 'Y') {
					isFlag = false;
				}
			}
		}
	}

	/**
	 * 添加客户的操作
	 */
	public void addNewCustoner() {
		System.out.println("---------------------- 添加客户 ----------
--------");
		System.out.print(" 姓名：");
		String name = CMUtility.readString(10);

		System.out.print(" 性别：");
		char gender = CMUtility.readChar();

		System.out.print(" 年龄：");
		int age = CMUtility.readInt();

		System.out.print(" 电话：");
		String phone = CMUtility.readString(13);

		System.out.print(" 邮箱：");
		String email = CMUtility.readString(30);
```

```
            // 将上述数据封装到对象中
            Customer customer = new Customer(name, gender, age,
phone, email);
            boolean isSuccess = customerList addCustomer
(customer);
            if (isSuccess) {
                System.out.println("-------------------- 添加完成 -------
-------------");
            } else {
                System.out.println("-------------------- 添加失败 -------
-------------");
            }
        }

        /**
         * 修改客户的操作
         */
        public void modifyCustomer() {
            System.out.println("-------------------- 修改客户 ------------
--------");
            Customer cust;
            int number;
            for (;;) {
                System.out.println(" 请选择待修改客户编号 (-1 退出): ");
                number = CMUtility.readInt();
                if (number == -1) {
                    return;
                }
                cust = customerList.getCustomer(number - 1);
                if (cust == null) {
                    System.out.println(" 无法找到指定客户！ ");
                } else {
                    break;
                }
            }

            // 修改客户信息
            System.out.print(" 姓名 (" + cust.getName() + "):");
            String name = CMUtility.readString(10, cust.getName());
            System.out.print(" 性别 (" + cust.getGender() + "):");
```



```
            char gender = CMUtility.readChar(cust.getGender());
            System.out.print(" 年龄 (" + cust.getAge() + "):");
            int age = CMUtility.readInt(cust.getAge());
            System.out.print(" 电话 (" + cust.getPhone() + "):");
            String phone = CMUtility.readString(13, cust.getPhone());
            System.out.print(" 邮箱 (" + cust.getEmail() + "):");
            String email = CMUtility.readString(30, cust.getEmail());
            Customer newcust = new Customer(name, gender, age,
phone, email);
            boolean isRepalaced = customerList.
replaceCustomer(number - 1, newcust);

            if (isRepalaced) {
                System.out.println("-------------------- 修改完成 -------
-------------");
            } else {
                System.out.println("-------------------- 修改失败 -------
-------------");
            }
        }

        /**
         * 删除客户的操作
         */
        public void deleteCustomer() {
            System.out.println("-------------------- 删除客户 ------------
--------");
            int number;
            for (;;) {
                System.out.print(" 请选择待删除客户编号（-1 退出）： ");
                number = CMUtility.readInt();
                if (number == -1) {
                    return;
                }

                Customer customer = customerList.
getCustomer(number - 1);
                if (customer == null) {
                    System.out.println(" 无法找到指定客户 !");
                } else {
```

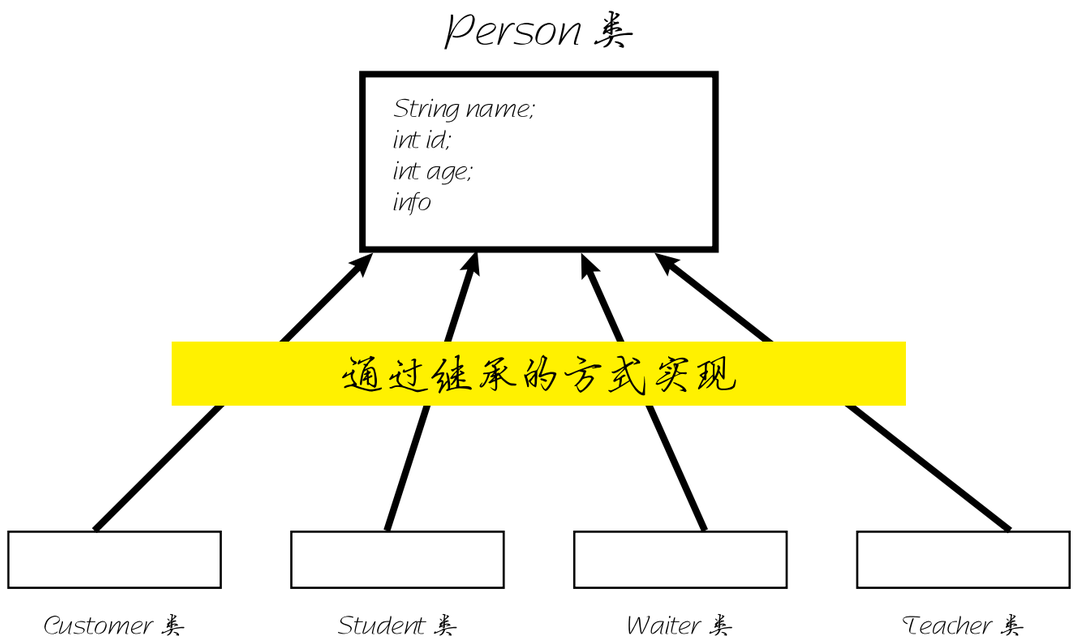
```
                        break;
                    }
                }
                System.out.println(" 请确认是否删除（Y/N）： ");
                char isDelete = CMUtility.readConfirmSelection();
                if (isDelete == 'Y') {
                    boolean deletesuccess = customerList.
deleteCustomer(number - 1);
                    if (deletesuccess) {
                        System.out.println("-------------------- 删除成
功 --------------------");
                    } else {
                        System.out.println("-------------------- 删除失
败 --------------------");
                    }
                } else {
                    return;
                }
        }

        /**
         * 显示客户的操作
         */
        public void listAllCustomer() {
            System.out.println("------------------- 客户列表 ------------
--------");

            int total = customerList.getTotal();
            if (total == 0) {
                System.out.println(" 没有客户记录！ ");
            } else {
                System.out.println(" 编号\t姓名\t性别\t年龄\t电话\t\t
邮箱 ");
                Customer[] custs = customerList.getAllCustomer();
                for (int i = 0; i < custs.length; i++) {
                    Customer cust = custs[i];
                    System.out.println((i + 1) + "\t" + cust.
getName() + "\t" + cust.getGender() + "\t" + cust.getAge()
+ "\t" + cust.getPhone() + "\t" + cust.getEmail() + "\t");
                }
```



```
            }
            System.out.println("------------------ 客户列表完成 ----------
----------");
        }
        public static void main(String[] args) {
            CustomerView view = new CustomerView();
            view.enterMainMenu();
        }
}
```

# 五　面向对象（中）

## 5.1　面向对象特征之二：继承性 (inheritance)




### 一、继承性的好处：

1、减少了代码的冗余，提高的代码的复用性；
2、便于功能扩展；
3、为之后多态性的使用，提供了前提。

### 二、继承性的格式：class A extends B { }

A：子类、派生类、subclass
B：父类、超类、基类、superclass

2.1、体现：一旦子类 A 继承父类 B 以后，子类 A 就获取了父类 B 中声明的所有属性和方法。
特别的，父类中声明为 private 的属性或方法，子类继承父类以后，仍然认为获取了父类中私有的结构。只是因为封装性的影响，使得子类不能直接调用父类的结构而已。

2.2、子类继承父类之后，还可以声明自己特有的属性或方法：实现功能拓展。子类和父类的关系，不同于子集和集合的关系。



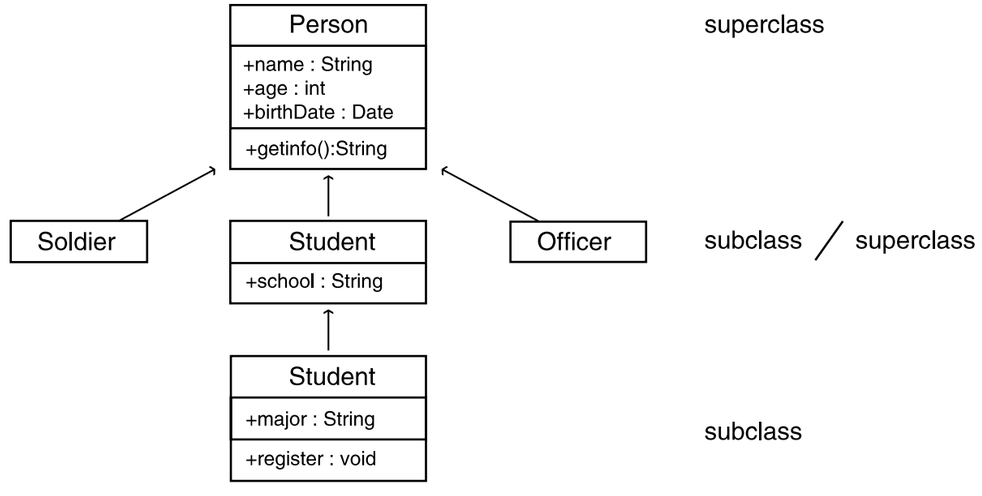
**Java 只支持单继承和多层继承，不允许多重继承。**
**・・一个子类只能有一个父类**
**・・一个父类可以派生出多个子类**

```
class SubDemo extends Demo{}  // ok
class SubDemo extends Demo1，Demo2...{}  // error
```

### 三、Java 中关于继承性的规定：

1、一个类可以被多个子类继承。
2、 Java 中类的单继承性：一个类只能有一个父类。
3、子父类是相对的概念。
4、子类直接继承的父类，称为：直接父类。
　　间接继承的父类称为：间接父类。
5、子类继承父类以后，就获取了直接父类以及所有间接父类中声明的属性和方法。

**四、** 1、如果我们没有显示的声明一个类的父类的话，则此类继承于 Java.lang.Object 类。
2、所有的 Java 类（除 java.long.Object 类之外）都直接或间接的继承于 java.long.Object 类。
3、意味着，所有的 java 类有 java.long.Object 类声明的功能。



## 1、Person 类

```
// 为描述和处理个人信息，定义类 Person
public class Person {
	String name;
	private int age;
	public Person(){
	}

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

	public void eat(){
		System.out.println(" 吃饭 ");
		sleep();
	}
	private void sleep(){
		System.out.println(" 睡觉 ");
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
}
```



## 2、Student 类

```
// 为描述和处理学生信息，定义类 Student
public class Student extends Person {
//	String name;
//	int age;
	String major;
	public Student(){
	}
	public Student(String name,int age,String major){
		this.name = name;
//		this.age = age;
		setAge(age);
		this.major = major;
	}
//	public void eat(){
//		System.out.println(" 吃饭 ");
//	}
//
//	public void sleep(){
//		System.out.println(" 睡觉 ");
//	}
	public void study(){
		System.out.println(" 学习 ");
	}
	public void show(){
		System.out.println("name:" + name + ",age = " + getAge());
	}
}
```

### 3、ExtendsTest 类

```
public class ExtendsTest {
	public static void main(String[] args) {
		Person p1 = new Person();
//		p1.age = 1;
		p1.eat();
		System.out.println("********************");
		Student s1 = new Student();
		s1.eat();
//		s1.sleep();
		s1.name = "Tom";
		s1.setAge(10);
		System.out.println(s1.getAge());
	}
}
```

### 4、Creature 类

```
public class Creature {
	public void brease(){
		System.out.println(" 呼吸 ");
	}
}
```



## 5.2 方法的重写 (ocerride / overwrite)

### 一、重写：

子类继承父类以后，可以对父类中同名同参数的方法，进行覆盖操作。

### 二、应用：

重写以后，当创建子类对象以后，通过子类对象调用子父类中的同名同参数的方法时，实际执行的是子类重写父类的方法。

### 三、重写的规定：

方法的声明：权限修饰符、返回值类型、方法名（形参列表）、throws 异常的类型 {

// 方法体

}

约定俗称：子类中的叫重写的方法，父类中的叫被重写的方法。

1. 子类重写的方法的方法名和形参列表与父类被重写的方法的方法名和形参列表相同；

2. 子类重写的方法的权限修饰符不小于父类被重写的方法的权限修饰符；特殊情况：子类不能重写父类中声明为 private 权限的方法。

3. 返回值类型：

> 父类被重写的方法的返回值类型是 void，则子类重写的方法的返回值类型只能是 void。

> 父类被重写的方法的返回值类型是 A 类型，则字方法重写的方法的返回值类型可是是 A 类或 A 类的子类。

> 父类被重写的方法的返回值类型是基本数据类型（比如 double），则子类重写的方法的返回值类型必须是基本数据类型（也必须是 double）。

4. 子类重写的方法跑出的异常类型不大于父类被重写的方法抛出的异常类型。

子类和父类中的同名同参数的方法要么都声明为非 static 的（考虑重写），要么都声明为 static 的（不是重写）。

## 1、Person 类

```
public class Person {
    String name;
    int age;
    public Person(){
    }
    public Person(String name,int age){
        this.name = name;
        this.age = age;
    }
    public void eat(){
        System.out.println(" 吃饭 ");
    }
    public void walk(int distance){
        System.out.println(" 走路，走的距离是: " + distance + " 公里 ");
        show();
    }
    private void show(){
        System.out.println(" 我是一个人。 ");
    }
    public Object info(){
        return null;
    }
    public double info1(){
        return 1.0;
    }
}
```



## 2、Student 类

```
public class Student extends Person{
    String major;
    public Student(){
    }
    public Student(String major){
        this.major = major;
    }
    public void study(){
        System.out.println(" 学习，专业是 :" + major);
    }
    // 对父类中的 eat() 进行了重写
    public void eat(){
        System.out.println(" 学生应该多吃有营养的。 ");
    }
    public void show(){
        System.out.println(" 我是一个学生。 ");
    }
    public String info(){
        return null;
    }
    // 不是一个类型，所以报错。
//    public int info1(){
//        return 1;
//    }
    // 可以直接将父类的方法的第一行粘过来，直接写方法体
//    public void walk(int distance){
//        System.out.println(" 重写的方法 ");
//    }
    // 直接输入父类的方法名，Alt + /，选择即可生成
    @Override
    public void walk(int distance) {
        System.out.println(" 自动生成 ");
    }
}
```

### 3、PersonTest 类

```
public class PersonTest {
	public static void main(String[] args) {
		Student s = new Student(" 计算机科学与技术 ");
		s.eat();
		s.walk(10);
		s.study();
	}
}
```



# 5.3 关键字：super

## super 关键字的使用

1.super 理解为：父类的。

2.super 可以用来调用：属性、方法、构造器 。

3.super 的使用：

3.1 我们可以在子类的方法或构造器中，通过 "super. 属性 " 或 "super. 方法 " 的方式，显式的调用
父类中声明的属性或方法。但是，通常情况下，我们习惯去省略这个 "super."

3.2 特殊情况：当子类和父类中定义了同名的属性时，我们要想在子类中调用父类中声明的属性，则必须显示的 使用 "super. 属性 " 的方式，表明调用的是父类中声明的属性。

3.3 特殊情况：当子类重写了父类中的方法后，我们想在子类的方法中调用父类中被重写的方法时，必须显示的使用 "super. 方法 " 的方式，表明调用的是父类中被重写的方法。

4.super 调用构造器

4.1 我们可以在子类的构造器中显示的使用 "super( 形参列表 )" 的方式 , 调用父类中声明的指定的构造器。

4.2 "super( 形参列表 )" 的使用，必须声明在子类构造器的首行 ！

4.3 我们在类的构造器中，针对于 "this( 形参列表 )" 或 "super( 形参列表 )" 只能二选一，不能同时出现。

4.4 在构造器的首行，既没有显式的声明 "this( 形参列表 )" 或 "super( 形参列表 )"，则默认的调用的是父类中的空参构造器。super()

4.5 在类的多个构造器中，至少有一个类的构造器使用了 "super( 形参列表 )", 调用父类中的构造器。

## 1、Person 类

```
public class Person {
    String name;
    int age;
    int id = 1003;   // 身份证号
    public Person(){
        System.out.println(" 我无处不在 ");
    }
    public Person(String name){
        this.name = name;
    }
    public Person(String name,int age){
        this(name);
        this.age = age;
    }
    public void eat(){
        System.out.println(" 人，吃饭 ");
    }
    public void walk(){
        System.out.println(" 人，走路 ");
    }
}
```



## 2、Student 类

```
public class Student extends Person{
    String major;
    int id = 1002;   // 学号
    public Student(){
    }
    public Student(String name,int age,String major){
//          this.age = age;
//          this.name = name;
            super(name,age);
            this.major = major;
    }
    public Student(String major){
        this.major = major;
    }
    public void eat(){
        System.out.println(" 学生多吃有营养的食物 ");
    }
    public void Study(){
        System.out.println(" 学生，学习知识。");
        this.eat();
        // 如果，想调用父类中被重写的，不想调用子类中的方法，可以:
        super.eat();
        super.walk();// 子父类中未重写的方法，用 "this." 或 "super." 调用都可以
    }
    public void show(){
        System.out.println("name = " + this.name + ",age = " + super.age);
        System.out.println("id = " + this.id);
        System.out.println("id = " + super.id);
    }
}
```

### 3、SuperTest 类

```
public class SuperTest {
	public static void main(String[] args) {
		Student s = new Student();
		s.show();
		s.Study();
		Student s1 = new Student("Ton",21,"IT" );
		s1.show();
		Student s2 = new Student();
	}
}
```



# 5.4 子类对象实例化过程

## 子类对象实例化的全过程

1. 从结果上看 :

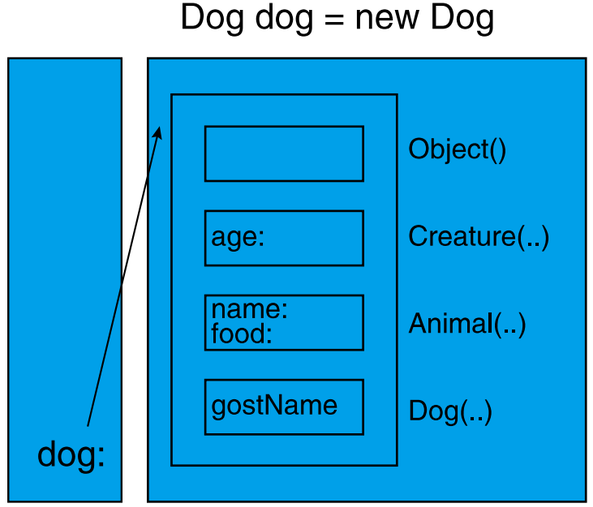
子类继承父类以后，就获取了父类中声明的属性或方法。

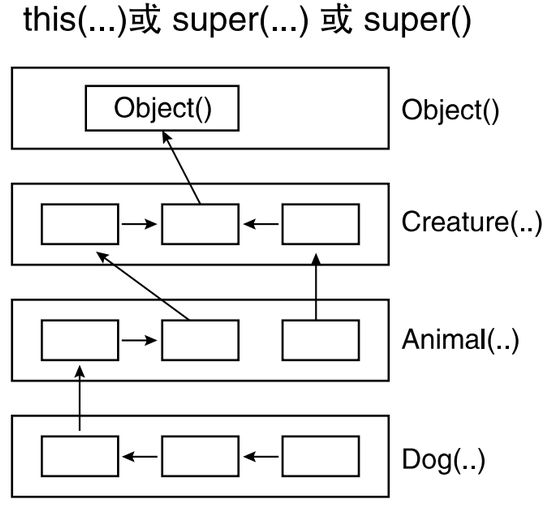
创建子类的对象中，在堆空间中，就会加载所有父类中声明的属性。

2. 从过程上看 :

当我们通过子类的构造器创建子类对象时 , 我们一定会直接或间接的调用其父类构造器，直到调用了 java.lang.Object 类中空参的构造器为止。正因为加载过所有的父类结构，所以才可以看到内存中有父类中的结构，子类对象可以考虑进行调用。

明确 : 虽然创建子类对象时，调用了父类的构造器，但自始至终就创建过一个对象，即为 new 的子类对象。



## 5.5 面向对象特征之三：多态性

1. 理解多态性：可以理解为一个事物的多种态性。

2. 何为多态性：
　　对象的多态性：父类的引用指向子类的对象 ( 或子类的对象赋值给父类的引用 )。

3. 多态的使用：虚拟方法调用
　　有了对象多态性以后，我们在编译期，只能调用父类声明的方法，但在执行期实际执行的是子类重写父类的方法。简称：编译时，看左边；运行时，看右边。
　　若编译时类型和运行时类型不一致，就出现了对象的多态性 (Polymorphism)。
　　多态情况下，
　　“看左边”：看的是父类的引用（父类中不具备子类特有的方法）
　　“看右边”：看的是子类的对象（实际运行的是子类重写父类的方法）。

4. 多态性的使用前提：
　　① 类的继承关系；
　　② 方法的重写。

5. 对象的多态性：只适用于方法，不适用于属性 ( 编译和运行都看左边 )

### 1、Person 类

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

    public void eat(){
        System.out.println(" 人，吃饭 ");
    }
    public void walk(){
        System.out.println(" 人，走路 ");
    }
}
```



### 2、Woman 类

```
public class Woman extends Person{
    boolean isBeauty;

    public void goShopping(){
        System.out.println(" 女人喜欢购物 ");
    }

    public void eat(){
        System.out.println(" 女人少吃，为了减肥。");
    }

    public void walk(){
        System.out.println(" 女人，窈窕的走路。");
    }
}
```

### 3、Man 类

```
public class Man extends Person{
    boolean isSmoking;

    public void earnMoney(){
        System.out.println(" 男人负责工作养家 ");
    }

    public void eat() {
        System.out.println(" 男人多吃肉，长肌肉 ");
    }

    public void walk() {
        System.out.println(" 男人霸气的走路 ");
    }
}
```

## 4、PersonTest 类

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

		Person p1 = new Person();
		p1.eat();

		Man man = new Man();
		man.eat();
		man.age = 25;
		man.earnMoney();

		//*********************************
		// 对象的多态性，父类的引用指向子类的对象
		Person p2 = new Man();
//		Person p3 = new Woman();

		// 多态的使用 : 当调用子父类同名同参数方法时，实际调用的是子类
重写父类的方法 --- 虚拟方法调用
		p2.eat();
		p2.walk();

//		p2.earnMoney();

	}
}
```



## 多态性应用举例

```
public class AnimalTest {
	public static void main(String[] args) {
		AnimalTest test = new AnimalTest();
		test.func(new Dog());
		test.func(new Cat());
	}

	public void func(Animal animal){//Animal animal = new Dog();
		animal.eat();
		animal.shout();
	}
	// 如果没有多态性，就会写很多如下的方法，去调用
	public void func(Dog dog){
		dog.eat();
		dog.shout();
	}
	public void func(Cat cat){
		cat.eat();
		cat.shout();
	}
}

class Animal{
	public void eat(){
		System.out.println(" 动物，进食 ");
	}
	public void shout(){
		System.out.println(" 动物：叫 ");
	}
}

class Dog extends Animal{
	public void eat(){
		System.out.println(" 狗吃骨头 ");
	}
	public void shout() {
		System.out.println(" 汪！汪！汪！ ");
	}
}
```

```
class Cat extends Animal{
        public void eat(){
                System.out.println(" 猫吃鱼 ");
        }

        public void shout() {
                System.out.println(" 喵！喵！喵！  ");
        }
}
```

## 5.5.1　虚拟方法的补充

### 从编译和运行的角度看：

重载，是指允许存在多个同名方法，而这些方法的参数不同。

编译器根据方法不同的参数表，对同名方法的名称做修饰。

对于编译器而言，这些同名方法就成了不同的方法。

它们的调用地址在编译期就绑定了。Java 的重载是可以包括父类和子类的，即子类可以重载父类的同名不同参数的方法。所以：对于重载而言，在方法调用之前，编译器就已经确定了所要调用的方法，这称为“ 早绑定 ”或“ 静态绑定 ”；而对于多态，只有等到方法调用的那一刻，解释运行器才会确定所要调用的具体方法，这称为“ 晚绑定 ”或“ 动态绑定 ”。

引用一句 Bruce Eckel 的话：“ 不要犯傻，如果它不是晚绑定，它就不是多态。 ”



## 5.5.2　向下转型的使用

### 1、Person 类

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

        public void eat(){
                System.out.println(" 人，吃饭 ");
        }
        public void walk(){
                System.out.println(" 人，走路 ");
        }
}
```

### 2、Woman 类

```
public class Woman extends Person{
        boolean isBeauty;

        public void goShopping(){
                System.out.println(" 女人喜欢购物 ");
        }

        public void eat(){
                System.out.println(" 女人少吃，为了减肥。");
        }

        public void walk(){
                System.out.println(" 女人，窈窕的走路。");
        }
}
```

## 3、Man 类

```
public class Man extends Person{
	boolean isSmoking;

	public void earnMoney(){
		System.out.println(" 男人负责工作养家 ");
	}

	public void eat() {
		System.out.println(" 男人多吃肉，长肌肉 ");
	}

	public void walk() {
		System.out.println(" 男人霸气的走路 ");
	}
}
```

## 4、PersonTest 类

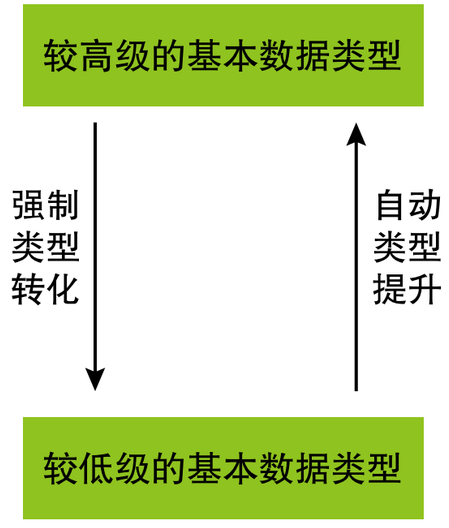
```
public class PersonTest {

	public static void main(String[] args) {
		Person p1 = new Person();
		p1.eat();

		Man man = new Man();
		man.eat();
		man.age = 25;
		man.earnMoney();

		// ***********************************
		System.out.println("************************");
		// 对象的多态性，父类的引用指向子类的对象
		Person p2 = new Man();

		// Person p3 = new Woman();
		// 多态的使用 : 当调用子父类同名同参数方法时，实际调用的是
子类重写父类的方法 --- 虚拟方法调用
		p2.eat();
		p2.walk();
```

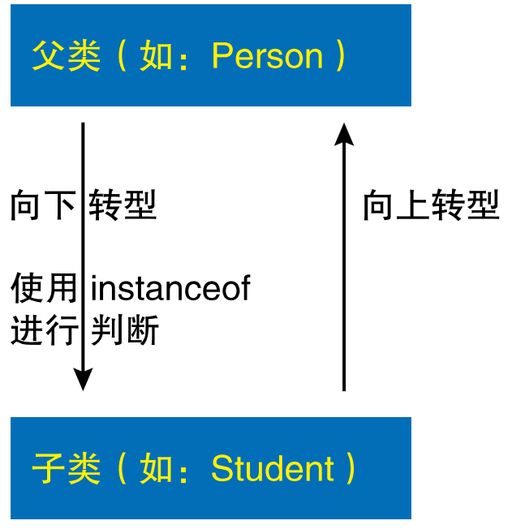


```
		// p2.earnMoney();
		System.out.println("************************");
		// 不能调用子类所特有的方法、属性，编译时，p2 是 Person
类型，
		// p2.earnMoney();

		p2.name = "Tom";
		// p2.isSmoking = true;
		// 有了对象的多态性以后，内存中实际上是加载了子类特有的
属性和方法，但是由于变量声明为父类类型，导致编译时，只能调用父类
中声明的属性和方法。子类的属性和方法不能调用。

		// 如何才能调用子类所特有的属性和方法?
		// 使用强制类型转换符，也可称为 : 向下转型
		Man m1 = (Man) p2;
		m1.earnMoney();
		m1.isSmoking = true;
		// 使用强转时，可能出现 ClassCastException 异常
		// Woman w1 = (Woman)p2;
		// w1.goShopping();
		/*
		 * instanceof 关键字的使用
		 * a instanceof A：判断对象 a 是否是类 A 的实例。如果是，
返回 true，如果不是，返回 false;
		 * 使用情境：为了避免在向下转型时出现 ClassCastException
异常，我们在进行向下转型之前，先进行 instanceof 的判断 , 一旦返回
true, 就进行向下转型。如果返回 false，不进行向下转型。
		 * 如果 a instanceof A 返回 true, 则 a instanceof B 也返回
true。 其中类 B 是类 A 的父类。
		 */

		if (p2 instanceof Woman) {
			Woman w1 = (Woman) p2;
			w1.goShopping();
			System.out.println("**********Woman*********");
		}

		if (p2 instanceof Man) {
			Man m2 = (Man) p2;
			m2.earnMoney();
```

```
            System.out.println("*********Man************");
        }

        if (p2 instanceof Person) {
            System.out.println("***********Person************");
        }

        if (p2 instanceof Object) {
            System.out.println("***********object************");
        }

        // 向下转型的常见问题
        // 练习
        // 问题 1: 编译时通过，运行时不通过
        // 举例一
//      Person p3 = new Woman();
//      Man m3 = (Man)p3;

        // 举例二
        Person p4 = new Person();
        Man m4 = (Man)p4;
        // 问题二 : 编译通过，运行时也通过
        Object obj = new Woman();
        Person p = (Person)obj;

        // 问题三 : 编译不通过
//      Man m5 = new woman();

//      String str = new Date();

//      Object o = new Date();
//      String str1 = (String)o;
    }
}
```





## 练习 1

```
/*
 * 练习 : 子类继承父类
 * 1. 若子类重写了父类方法，就意味着子类里定义的方法彻底覆盖了父类里的同名方法，系统将不可能把父类里的方法转移到子类中。
 *
 * 2. 对于实例变量则不存在这样的现象，即使子类里定义了与父类完全相同的实例变量，这个实例变量依然不可能覆盖父类中定义的实例变量。
 */
public class FieldMethodTest {
	public static void main(String[] args){
		Sub s= new Sub();
		System.out.println(s.count);  //20
		s.display();  //20

		Base b = s;
		//==: 对于引用数据类型来讲，比较的是两个引用数据类型变量的地址值是否一样。
		System.out.println(b == s);  //true
		System.out.println(b.count);  //10
		b.display();
	}
}

class Base {
	int count= 10;
	public void display() {
		System.out.println(this.count);
	}
}

class Sub extends Base {
	int count= 20;
	public void display() {
		System.out.println(this.count);
	}
}
```



# 5.6 Object 类的使用

java.lang.Object 类

1.Object 类是所有 Java 类的根父类 ;

2. 如果在类的声明中未使用 extends 关键字指明其父类，则默认父类为 java.lang.Object 类

3.Object 类中的功能 ( 属性、方法 ) 就具有通用性。

属性：无

方　法：equals() / toString() / getClass() / hashCode() / clone() / finalize() / wait() / notify() / notifyAll()。

4.Object 类只声明了一个空参的构造器。

## 5.6.1 Object 类中的主要结构

| NO. | 方法名称 | 类型 | 描述 |
|---|---|---|---|
| 1 | public　Object() | 构造 | 构造器 |
| 2 | public　boolean　equals(Object　obj) | 普通 | 对象比较 |
| 3 | public　int　hashCode() | 普通 | 取得Hash码 |
| 4 | public　String　toString() | 普通 | 对象打印时调用 |

## 5.6.2 == 操作符与 equals 方法

### == 和 equals 的区别

一、回顾 == 的使用

==：运算符

1. 可以使用在基本数据类型变量和引用数据类型变量中；

2. 如果比较的是基本数据类型变量：比较两个变量保存的数据是否相等。( 不一定类型要相同 )

如果比较的是引用数据类型变量：比较两个对象的地址值是否相同，即两个引用是否指向同一个对象实体；

补充： == 符号使用时，必须保证符号左右两边的变量类型一致。

二、equals() 方法的使用

1. 是一个方法，而非运算符；

2. 只能适用于引用数据类型；

3. Object 类中 equals() 的定义：

```
public boolean equals(Object obj){
    return (this == obj);
}
```

说明：Object 类中定义的 equals() 和 == 的作用是相同的，比较两个对象的地址值是否相同，即两个引用是否指向同一个对象实体。

4. 像 String、Date、File、包装类等都重写了 Object 类中的 equals() 方法 . 两个引用的地址是否相同，而是比较两个对象的 “ 实体内容 ” 是否相同。

5. 通常情况下，我们自定义的类如果使用 equals() 的话，也通常是比较两个对象的 " 实体内容 " 是否相同。那么，我们就需要对 Object 类中的 equals() 进行重写。

重写的原则：比较两个对象的实体内容是否相同。



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

        // 基本数据类型
        int i = 10;
        int j = 10;
        double d = 10.0;
        System.out.println(i == j);//true
        System.out.println(i == d); //true

//      boolean b =true;
//      System.out.println(i == b);

        char c = 10;
        System.out.println(i == c); //true

        char c1 = 'A';
        char c2 = 65;
        System.out.println(c1 == c2); //true

        // 引用数据类型
        Customer cust1 = new Customer("Tom" ,21);
        Customer cust2 = new Customer("Tom" ,21);
        System.out.println(cust1 == cust2); //false

        String str1 = new String("BAT");
        String str2 = new String("BAT");
        System.out.println(str1 == str2); //false
        System.out.println("************************");
        System.out.println(cust1.equals(cust2));  //false
        System.out.println(str1.equals(str2));       //true

        Date date1 = new Date(23432525324L);
        Date date2 = new Date(23432525324L);
        System.out.println(date1.equals(date2)); //true
    }
}
```

## Customer 类

```
public class Customer {
	private String name;
	private int age;
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getAge() {
		return age;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public Customer() {
		super();
	}
	public Customer(String name, int age) {
		super();
		this.name = name;
		this.age = age;
	}


	// 自动生成的 equals()
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Customer other = (Customer) obj;
		if (age != other.age)
			return false;
		if (name == null) {
			if (other.name != null)
				return false;
		} else if (!name.equals(other.name))
			return false;
		return true;
	}

	// 重写原则，比较两个对象的实体内容 ( 即 name 和 age) 是否相同
	// 手动实现 equals() 的重写
//	@Override
//	public boolean equals(Object obj) {
//
//		System.out.println("Customer equals()....");
//		if(this == obj){
//			return true;
//		}
//
//		if(obj instanceof Customer){
//			Customer cust = (Customer)obj;
//			// 比较两个对象的属性是否都相同
//			//if(this.age == cust.age && this.name.equals(cust.
name)){
//				return true;
//			}else{
//				return false;
//			}
//
//			// 或
//			return this.age == cust.age && this.name.equals(cust.
name);
//		}
//
//		return false;
//	}

}
```

## 5.6.3 重写 equals() 方法的原则

对称性：如果 x.equals(y) 返回是“ true ”，那么 y.equals(x) 也应该返回是“ true ”。

自反性：x.equals(x) 必须返回是“ true ”。

传递性：如果 x.equals(y) 返回是“ true ”，而且 y.equals(z) 返回是“ true ”，那么 z.equals(x) 也应该返回是“ true ”。

一致性：如果 x.equals(y) 返回是“ true ”，只要 x 和 y 内容一直不变，不管你重复 x.equals(y) 多少次，返回都是“ true ”。

任何情况下，x.equals(null)，永远返回是“ false ”；x.equals( 和 x 不同类型的对象 ) 永远返回是“ false ”。

## 5.6.4 toString 的使用

### Object 类中 toString() 的使用

1. 当我们输出一个引用对象时，实际上就是调用当前对象的 toString()

2. Object 类中 toString 的定义方法

```
public String toString() {
        return getClass().getName() + "@" + Integer.
toHexString(hashCode());
    }
```

3. 像 String、Date、File、包装类等都重写了 Object 类中的 toString() 方法。使得在调用 toString() 时，返回 " 实体内容 " 信息。

4. 自定义类如果重写 toString() 方法，当调用此方法时，返回对象的 " 实体内容 "。



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

        Customer cust1 = new Customer("Tom" ,21);
        System.out.println(cust1.toString());
        //github4.Customer@15db9742
        System.out.println(cust1);
        //github4.Customer@15db9742 ---> Customer[name =
Tom,age = 21]

        String str = new String("MM");
        System.out.println(str);

        Date date = new Date(45362348664663L);
        System.out.println(date.toString());
        //Wed Jun 24 12:24:24 CST 3407

    }
}
```

# 5.7 包装类 (Wrapper) 的使用

## 5.7.1 单元测试方法的使用

### java 中的 JUnit 单元测试

步骤 :

1. 选中当前项目工程 - -》 右键：build path - -》 add libraries - -》 JUnit 4 - -》 下一步；

2. 创建一个 Java 类进行单元测试。

此时的 Java 类要求：①此类是公共的 ②此类提供一个公共的无参构造器

3. 此类中声明单元测试方法。

此时的单元测试方法：方法的权限是 public, 没有返回值，没有形参。

4. 此单元测试方法上需要声明注解：@Test 并在单元测试类中调用：import org.junit.Test；

5. 声明好单元测试方法以后，就可以在方法体内测试代码。

6. 写好代码后，左键双击单元测试方法名：右键 - -》 run as - -》 JUnit Test；

说明：如果执行结果无错误，则显示是一个绿色进度条，反之，错误即为红色进度条。



```java
public class JUnit {
	int num = 10;
	// 第一个单元测试方法
	@Test
	public void testEquals(){
		String s1 = "MM";
		String s2 = "MM";
		System.out.println(s1.equals(s2));
		//ClassCastException 的异常
//		Object obj = new String("GG");
//		Date date = (Date)obj;

		System.out.println(num);
		show();
	}

	public void show(){
		num = 20;
		System.out.println("show()...");
	}

	// 第二个单元测试方法
	@Test
	public void testToString(){
		String s2 = "MM";
		System.out.println(s2.toString());
	}
}
```

## 5.7.2 包装类的使用

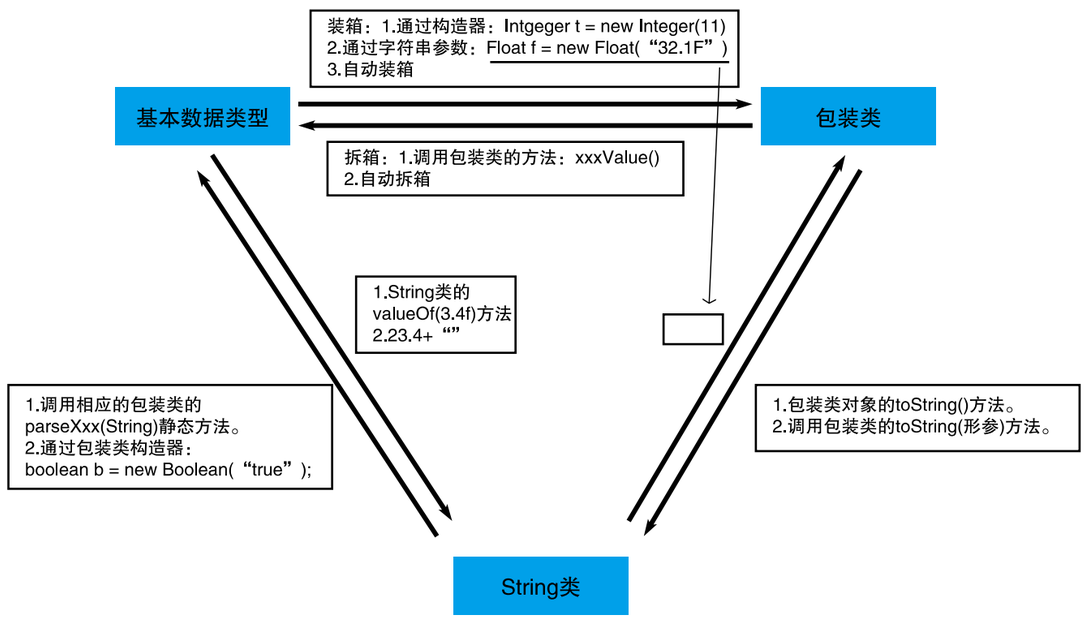
java 提供了 8 种基本数据类型对应的包装类，使得基本数据类型的变量具有类的特征。

| 基本数据类型 | 包装类 |
| --- | --- |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

父类 number

注意：其中 Byte、Short、Integer、Long、Float、Double 的父类是：Number

## 5.7.3 包装类与基本数据类型相互转换






## 基本数据类型、包装类、String 三者之间的相互转换。

```
public class WrapperTest {
    //String 类型 ---> 基本数据类型、包装类 , 调用包装类的 parseXxx()

    @Test
    public void test5(){
        String str1 = "123";
//        String str1 = "123a";

        // 错误的情况，可能会报错
//        int num1 = (int)str1;
//        Integer in1 = (Integer)str1;

        int num2 = Integer.parseInt(str1);
        System.out.println(num2 + 1);//124

        String str2 = "true";
        Boolean b1 = Boolean.parseBoolean(str2);
        System.out.println(b1);    //true
    }

    // 基本数据类型、包装类 ---》String 类型，调用 String 重载的 valueOf(Xxx xxx)
    @Test
    public void test4(){
        int num1 = 10;
        // 方式 1: 连接运算
        String str1 = num1 + "";
        // 方式 2: 调用 String 的 valueOf(Xxx xxx)
        float f1 = 12.3f;
        String str2 = String.valueOf(f1); //"12.3"

        Double d1 = new Double(12.4);
        String str3 = String.valueOf(d1);
        System.out.println(str2);
        System.out.println(str3);  //"12.4"

    }
```

```java
/*
 * JDK 5.0 新特性 : 自动装箱与自动拆箱
 */
@Test
public void test3(){
//        int num1 = 10;
//        // 基本数据类型 --》 包装类的对象
//        method(num1);       //Object obj = num1

        // 自动装箱 : 基本数据类型 --》 包装类
        int num2 = 10;
        Integer in1 = num2;// 自动装箱

        boolean b1 = true;
        Boolean b2 = b1;// 自动装箱

        // 自动拆箱:  包装类 --》 基本数据类型
        System.out.println(in1.toString());

        int num3 = in1;

}

public void method(Object obj){
        System.out.println(obj);
}

// 包装类 --》 基本数据类型 : 调用包装类的 xxxValue()
@Test
public void test2() {
        Integer in1 = new Integer(12);
        int i1 = in1.intValue();
        System.out.println(i1 + 1);

        Float f1 = new Float(12.3f);
        float f2 = f1.floatValue();
        System.out.println(f2 + 1);
}
```



```java
// 基本数据类型 --》包装类 , 调用包装类的构造器
@Test
public void test1() {
        int num1 = 10;
//        System.out.println(num1.toString());

        Integer in1 = new Integer(num1);
        System.out.println(in1.toString());

        Integer in2 = new Integer("123");
        System.out.println(in2.toString());

        // 报异常
//        Integer in3 = new Integer("123abc");
//        System.out.println(in3.toString());

        Float f1 = new Float(12.3f);
        Float f2 = new Float("12.3");
        System.out.println(f1);
        System.out.println(f2);

        Boolean b1 = new Boolean(true);
        Boolean b2 = new Boolean("true");

        Boolean b3 = new Boolean("true123");
        System.out.println(b3); //false

        Order order = new Order();
        System.out.println(order.isMale); //false
        System.out.println(order.isFemale); //null

    }
}

class Order{

        boolean isMale;
        Boolean isFemale;
}
```

# 六　面向对象（下）

## 6.1　关键字：static

### 6.1.1　static 的使用

当我们编写一个类时，其实就是在描述其对象的属性和行为，而并没有产生实质上的对象，只有通过 new 关键字才会产生出对象，这时系统才会分配内存空间给对象，其方法才可以供外部调用。

我们有时候希望无论是否产生了对象或无论产生了多少对象的情况下，某些特定的数据在内存空间里只有一份。

例如所有的中国人都有个国家名称，每一个中国人都共享这个国家名称，不必在每一个中国人的实例对象中都单独分配一个用于代表国家名称的变量。

static 关键字的使用

1.static：静态的。

2.static 可以用来修饰：属性、方法、代码块、内部类。

3. 使用 static 修饰属性：静态变量 ( 或类变量 )。

3.1 属性：是否使用 static 修饰，又分为：静态属性 VS 非静态属性 ( 实例变量 )。

实例变量：我们创建了类的多个对象，每个对象都独立的拥有了一套类中的非静态属性。当修改其中一个非静态属性时，不会导致其他对象中同样的属性值的修饰。

静态变量：我们创建了类的多个对象，多个对象共享同一个静态变量。当通过静态变量去修改某一个变量时，会导致其他对象调用此静态变量时，是修改过的。

3.2 static 修饰属性的其他说明：

① 静态变量随着类的加载而加载。可以通过 " 类 . 静态变量 " 的方式进行调用。

② 静态变量的加载要早于对象的创建。

③ 由于类只会加载一次，则静态变量在内存中也只会存在一次。存在方法区的静态域中。

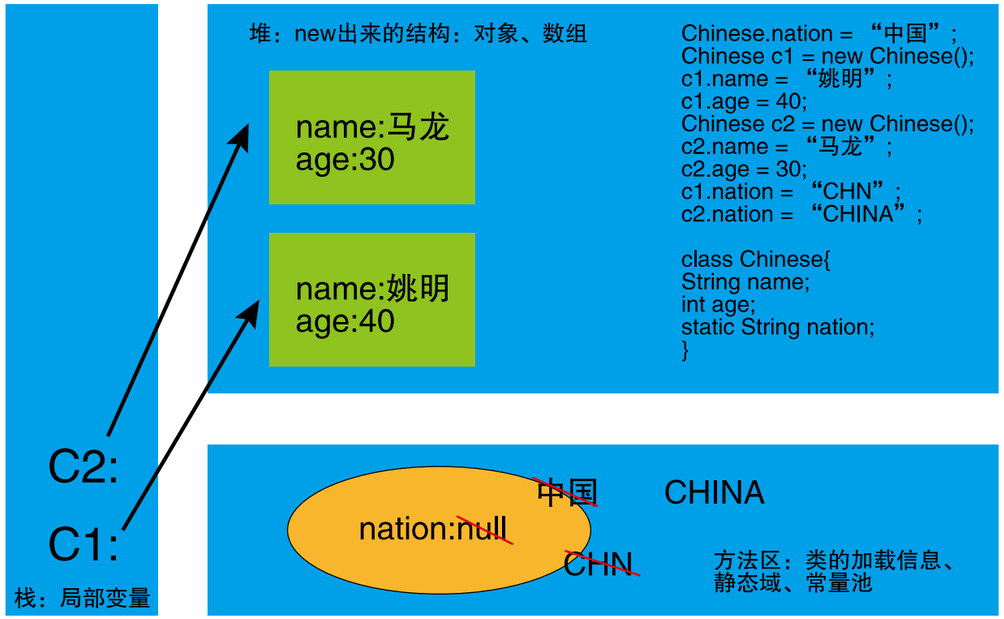
| ④ | 类变量 | 实例变量 |
|---|---|---|
| 类 | yes | no |
| 对象 | yes | yes |

3.3 静态属性举例：System.out.Math.PI；



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

		Chinese.nation = " 中国 ";

		Chinese c1 = new Chinese();
		c1.name = " 姚明 ";
		c1.age = 40;
		c1.nation = "CHN";

		Chinese c2 = new Chinese();
		c2.name = " 马龙 ";
		c2.age = 30;
		c2.nation = "CHINA";

		System.out.println(c1.nation);

		// 编译不通过
//		Chinese.name = " 张继科 ";
	}
}

// 中国人
class Chinese{

	String name;
	int age;
	static String nation;
}
```

## 6.1.2 类变量 vs 实例变量内存解析




## 6.1.3 static 修饰方法

4. 使用 static 修饰方法：静态方法：

① 随着类的加载而加载，可以通过 " 类 . 静态方法 " 的方式调用

| ② | 静态方法 | 非静态方法 |
|---|---|---|
| 类 | yes | no |
| 对象 | yes | yes |

③ 静态方法中，只能调用静态的方法或属性。
非静态的方法中，可以调用所有的方法或属性。

5.static 注意点：

5.1 在静态的方法内，不能使用 this 关键字、super 关键字。

5.2 关于静态属性和静态方法的使用，大家从生命周期的角度去理解。

6. 开发中，如何确定一个属性是否需要声明 static 的？

- - - > 属性是可以被多个对象所共享的，不会随着对象的不同而不同的。

- - - > 类中的常量也常常声明为 static。

开发中，如何确定一个方法是否要声明为 static 的？

- - - > 操作静态属性的方法，通常设置为 static 的；

- - - > 工具类中的方法，习惯上声明为 static 的。

比如：Math、Arrays、Collections



```
public class StaticTest {
	public static void main(String[] args) {
		Chinese.nation = " 中国 ";
		Chinese c1 = new Chinese();
		// 编译不通过
//		Chinese.name = " 张继科 ";
		c1.eat();
		Chinese.show();
		// 编译不通过
//		chinese.eat(); // Chinese.info();
	}
}
// 中国人
class Chinese{
	String name;
	int age;
	static String nation;
	public void eat(){
		System.out.println(" 中国人吃中餐 ");
		// 调用非静态结构
		this.info();
		System.out.println("name : " + name);
		// 调用静态结构
		walk();
		System.out.println("nation : " + Chinese.nation);
	}
	public static void show(){
		System.out.println(" 我是一个中国人！ ");
//		eat();
//		name = "Tom";
		// 可以调用静态的结构
		System.out.println(Chinese.nation);
		walk();
	}
	public void info(){
		System.out.println("name : " + name + ",age : " + age);
	}
	public static void walk(){
	}
}
```

## 6.1.4 自定义 ArrayUtil 的优化

```
// 自定义数组工具类
public class ArrayUtil {

    // 求数组的最大值
    public static int getMax(int[] arr) {
        int maxValue = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (maxValue < arr[i]) {
                maxValue = arr[i];
            }
        }
        return maxValue;
    }

    // 求数组的最小值
    public static int getMin(int[] arr) {
        int minValue = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (minValue > arr[i]) {
                minValue = arr[i];
            }
        }
        return minValue;
    }

    // 求数组总和
    public static int getSum(int[] arr) {
        int sum = 0;
        for (int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
        return sum;
    }
    // 求数组平均值
    public static int getAvg(int[] arr) {
        int avgValue = getSum(arr) / arr.length;
        return avgValue;
    }
```



```
    // 如下两个同名方法构成重载
    // 反转数组
    public static void reverse(int[] arr) {
        for (int i = 0; i < arr.length / 2; i++) {
            int temp = arr[i];
            arr[i] = arr[arr.length - i - 1];
            arr[arr.length - i - 1] = temp;
        }
    }

    public void reverse(String[] arr){

    }

    // 复制数组
    public static int[] copy(int[] arr) {
        int[] arr1 = new int[arr.length];
        for (int i = 0; i < arr1.length; i++) {
            arr1[i] = arr[i];
        }
        return null;
    }

    // 数组排序
    public static void sort(int[] arr) {
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
//                  int temp = arr[j];
//                  arr[j] = arr[j + 1];
//                  arr[j + 1] = temp;
                    // 错误的:
//                  swap(arr[j],arr[j+1]);
                    swap(arr,j ,j+1);
                }
            }
        }
    }
```

```
    // 错误的 : 交换数组中两个指定位置元素的值
//    public void swap(int i,int j){
//        int temp = i;
//        i = j;
//        j = temp;
//    }

    // 正确的:
    private static void swap(int[] arr,int i,int j){
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    // 遍历数组
    public static void print(int[] arr) {
        System.out.print("[");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + ",");
        }
        System.out.println("]");
    }

    // 查找指定元素
    public static int getIndex(int[] arr, int dest) {
        // 线性查找
        for (int i = 0; i < arr.length; i++) {

            if (dest==arr[i]) {
                return i;
            }

        }
        return -1;
    }

}
```



## 6.1.5 static 的应用举例

```
//static 关键字的应用
public class CircleTest {
    public static void main(String[] args) {

        Circle c1 = new Circle();
        Circle c2 = new Circle();
        Circle c3 = new Circle();

        System.out.println("c1 的 ID:" + c1.getId());
        System.out.println("c2 的 ID:" + c2.getId());
        System.out.println("c3 的 ID:" + c3.getId());
        System.out.println(" 创建圆的个数 : " + Circle.getTotal());
    }
}

class Circle{
    private double radius;
    private int id;   // 需要自动赋值

    public Circle(){
        id = init++;
        total++;
    }

    public Circle(double radius){
        this();
        // 或
//        id = init++;
//        total++;
        this.radius = radius;
    }

    private static int total;// 记录创建圆的个数
    private static int init = 1001;//static 声明的属性被所有对象所共享
```

```
public double findArea(){
        return 3.14 * radius * radius;
}

public double getRadius() {
        return radius;
}

public void setRadius(double radius) {
        this.radius = radius;
}

public int getId() {
        return id;
}

public static int getTotal() {
        return total;
}
}
```



## 6.1.6 static 的练习

编写一个类实现银行账户的概念，包含的属性有“帐号”、“密码”、“存款余额”、“利率”、“最小余额”，定义封装这些属性的方法。

账号要自动生成。编写主类，使用银行账户类，输入、输出 3 个储户的上述信息。

考虑：哪些属性可以设计成 static 属性。

```
public class Account {

        private int id;   // 账号
        private String pwd = "000000";       // 密码
        private double balance; // 存款余额

        private static double interestRate; // 利率
        private static double minMoney = 1.0;  // 最小余额
        private static int init = 1001;   // 用于自动生成 id

public Account(){     // 账号自动生成
        id = init++;
}

public Account(String pwd,double balance){
        id = init++;
        this.pwd = pwd;
        this.balance = balance;
}

public String getPwd() {
        return pwd;
}

public void setPwd(String pwd) {
        this.pwd = pwd;
}

public static double getInterestRate() {
        return interestRate;
}
```

```
public static void setInterestRate(double interestRate) {
        Account.interestRate = interestRate;
}

public static double getMinMoney() {
        return minMoney;
}

public static void setMinMoney(double minMoney) {
        Account.minMoney = minMoney;
}

public int getId() {
        return id;
}

public double getBalance() {
        return balance;
}

        @Override
        public String toString() {
                return "Account [id=" + id + ", pwd=" + pwd + ",
balance=" + balance + "]";
        }

}
```



## 6.1.7 单例 (Singleton) 设计模式

设计模式是 ** 在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及解决问题的思考方式。** 设计模免去我们自己再思考和摸索。就像是经典的棋谱，不同的棋局，我们用不同的棋谱。“ 套路 ”

所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例。并且该类只提供一个取得其对象实例的方法。如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的构造器的访问权限设置为 private，这样，就不能用 new 操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。

**单例设计模式 :**

**1. 所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例。**

**2. 如何实现？**

**饿汉式 VS 懒汉式**

**3. 区分饿汉式和懒汉式。**

**饿汉式：坏处：对象加载时间过长。**

**好处：饿汉式是线程安全的。**

**懒汉式：好处：延迟对象的创建。**

**坏处：目前的写法，会线程不安全。**

**---》到多线程内容时，再修改。**

## 1、单例模式的饿汉式

```
public class SingletonTest {
	public static void main(String[] args) {
//		Bank bank1 = new Bank();
//		Bank bank2 = new Bank();

		Bank bank1 = Bank.getInstance();
		Bank bank2 = Bank.getInstance();

		System.out.println(bank1 == bank2);
	}
}

// 单例的饿汉式
class Bank{

	//1. 私有化类的构造器
	private Bank(){

	}

	//2. 内部创见类的对象
	//4. 要求此对象也必须声明为静态的
	private static Bank instance = new Bank();

	//3. 提供公共的静态的方法，返回类的对象。
	public static Bank getInstance(){
		return instance;
	}
}
```



## 2、单例模式的懒汉式

```
public class SingletonTest2 {
	public static void main(String[] args) {

		Order order1 = Order.getInstance();
		Order order2 = Order.getInstance();

		System.out.println(order1 == order2);
	}
}

class Order{
	//1. 私有化类的构造器
	private Order(){

	}

	//2. 声明当前类对象，没有初始化。
	// 此对象也必须声明为 static 的
	private static Order instance = null;

	//3. 声明 public、static 的返回当前类对象的方法
	public static Order getInstance(){
		if(instance == null){
			instance = new Order();
		}
		return instance;
	}
}
```

## 3、单例模式的优点

由于单例模式只生成一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。

## 4、单例 (Singleton) 设计模式 - 应用场景

网站的计数器，一般也是单例模式实现，否则难以同步。

应用程序的日志应用，一般都使用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。

数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。

项目中，读取配置文件的类，一般也只有一个对象。没有必要每次使用配置文件数据，都生成一个对象去读取。

Application 也是单例的典型应用。

Windows 的 **Task Manager ( 任务管理器 )** 就是很典型的单例模式。

Windows 的 **Recycle Bin( 回收站 )** 也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。



# 6.2 理解 main 方法的语法 ( 了解 )

由于 Java 虚拟机需要调用类的 main() 方法，所以该方法的访问权限必须是 public，又因为 Java 虚拟机在执行 main() 方法时不必创建对象，所以该方法必须是 static 的，该方法接收一个 String 类型的数组参数，该数组中保存执行 Java 命令时传递给所运行的类的参数。

又因为 main() 方法是静态的，我们不能直接访问该类中的非静态成员，必须创建该类的一个实例对象后，才能通过这个对象去访问类中的非静态成员，这种情况，我们在之前的例子中多次碰到。

main () 方法的使用说明：
1.main () 方法作为程序的入口；
2.main () 方法也是一个普通的静态方法；
3.main () 方法也可以作为我们与控制台交互的方式。
( 之前，使用 Scanner)

```
public class MainTest {
    public static void main(String[] args) {    // 入口
        Main.main(new String[100]);

        MainTest test = new MainTest();
        test.show();
    }
    public void show(){
    }
}

class Main{
    public static void main(String[] args) {
        args = new String[100];
        for(int i = 0;i < args.length;i++){
            args[i] = "args_" + i;
            System.out.println(args[i]);
        }
    }
}
```

# 6.3 类的成员之四：代码块

## 类的成员之四：代码块（或初始化块）

1. 代码块的作用：用来初始化类、对象的。

2. 代码块如果有修饰的话，只能使用 static。

3. 分类：静态代码块 vs 非静态代码块。

4. 静态代码块：
    > 内部可以有输出语句
    > 随着类的加载而执行，而且只执行一次
    > 作用：初始化类的信息
    > 如果一个类中，定义了多个静态代码块，则按照声明的先后顺序执行
    > 静态代码块的执行，优先于非静态代码块的执行
    > 静态代码块内只能调用静态的属性、静态的方法，不能调用非静态的结构。

5. 非静态代码块
    > 内部可以有输出语句
    > 随着对象的创建而执行
    > 每创建一个对象，就执行一次非静态代码块
    > 作用：可以在创建对象时，对对象的属性等进行初始化
    > 如果一个类中，定义了多个非静态代码块，则按照声明的先后顺序执行
    > 非静态代码块内可以调用静态的属性、静态的方法，或非静态的属性、非静态的方法。

对属性可以赋值的位置：
    ①默认初始化。
    ②显式初始化。
    ③构造器中初始化。
    ④有了对象以后，可以通过 " 对象 . 属性 " 或 " 对象 . 方法 " 的方式，进行赋值。
    ⑤在代码块中赋值。



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

        String desc = Person.desc;
        System.out.println(desc);

        Person p1 = new Person();
        Person p2 = new Person();
        System.out.println(p1.age);

        Person.info();
    }
}

class Person{
    // 属性
    String name;
    int age;
    static String desc = " 我是一个青年 ";

    // 构造器
    public Person(){

    }

    //static 的代码块
    static{
        System.out.println("hello,static block-1");
        // 调用静态结构
        desc = " 我是一个爱小说的人 ";
        info();
        // 不能调用非静态结构
//        eat();
//        name = "Tom";
    }

    static{
        System.out.println("hello,static block-2");
    }
```

```
// 非 static 的代码块
{
        System.out.println("hello,block-2");
}
{
        System.out.println("hello,block-1");
        // 调用非静态结构
        age = 1;
        eat();
        // 调用静态结构
        desc = " 我是一个爱小说的人 1";
        info();
}

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

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

        @Override
        public String toString() {
                return "Person [name=" + name + ", age=" + age + "]";
        }
        public static void info(){
                System.out.println(" 我是一个快乐的人。");
        }

}
```



## 静态初始化块举例 1

```
// 总结 : 由父类到子类，静态先行
class Root{
        static{
                System.out.println("Root 的静态初始化块 ");
        }
        {
                System.out.println("Root 的普通初始化块 ");
        }
        public Root(){
                System.out.println("Root 的无参数的构造器 ");
        }
}

class Mid extends Root{
        static{
                System.out.println("Mid 的静态初始化块 ");
        }
        {
                System.out.println("Mid 的普通初始化块 ");
        }
        public Mid(){
                System.out.println("Mid 的无参数的构造器 ");
        }
        public Mid(String msg){
                // 通过 this 调用同一类中重载的构造器
                this();
                System.out.println("Mid 的带参数构造器，其参数值："
 + msg);
        }
}
```

```
class Leaf extends Mid{
	static{
		System.out.println("Leaf 的静态初始化块 ");
	}
	{
		System.out.println("Leaf 的普通初始化块 ");
	}
	public Leaf(){
		// 通过 super 调用父类中有一个字符串参数的构造器
		super(" 尚硅谷 ");
		System.out.println("Leaf 的构造器 ");
	}
}
public class LeafTest{
	public static void main(String[] args){
		new Leaf();
		//new Leaf();
	}
}
```



## 静态初始化块举例 2

```
class Father {
	static {
		System.out.println("11111111111");
	}
	{
		System.out.println("22222222222");
	}

	public Father() {
		System.out.println("33333333333");
	}
}

public class Son extends Father {
	static {
		System.out.println("44444444444");
	}
	{
		System.out.println("55555555555");
	}
	public Son() {
		System.out.println("66666666666");
	}

	public static void main(String[] args) { // 由父及子 静态先行
		System.out.println("77777777777");
		System.out.println("************************");
		new Son();
		System.out.println("************************");

		new Son();
		System.out.println("************************");
		new Father();
	}
}
```

## 总结：程序中成员变量赋值的执行顺序

对属性可以赋值的位置：
①默认初始化
②显式初始化 / ⑤在代码块中赋值
③构造器中初始化
④有了对象以后，可以通过 " 对象 . 属性 " 或 " 对象 . 方法 " 的方式，进行赋值。

执行的先后顺序：① - ② / ⑤ - ③ - ④

```
public class OrderTest {
	public static void main(String[] args) {
		Order order = new Order();
		System.out.println(order.orderId);
	}
}

class Order{
	int orderId = 3;
	{
		orderId = 4;
	}

}
```



# 6.4 关键字：final

## fina：最终的

1.final 可以用来修饰的结构：类、方法、变量。

2.final 用来修饰一个类：此类不能被其他类所继承。
比如：String 类、System 类、StringBuffer 类。

3.final 修饰一个方法：final 标记的方法不能被子类重写。
比如：Object 类中的 getClass()。

4.final 用来修饰变量：此时的 " 变量 "( 成员变量或局部变量 ) 就是一个常量。名称大写，且只能被赋值一次。
4.1 final 修饰属性，可以考虑赋值的位置有：显式初始化、代码块中初始化、构造器中初始化；
4.2 final 修饰局部变量：
尤其是使用 final 修饰形参时，表明此形参是一个常量。当我们调用此方法时，给常量形参赋一个实参。一旦赋值以后，就只能在方法体内使用此形参，但不能进行重新赋值。

static final 用来修饰：全局常量。

```
public class FinalTest {
	final int WIDTH = 0;
	final int LEFT;
	final int RIGHT;
//	final int DOWN;
	{
		LEFT = 1;
	}
	public FinalTest(){
		RIGHT = 2;
	}
	public FinalTest(int n){
		RIGHT = n;
	}
//	public void setDown(int down){
//		this.DOWN = down;
//	}
```

```
    public void dowidth(){
//          width = 20;     //width cannot be resolved to a variable
    }

    public void show(){
        final int NUM = 10; // 常量
//          num += 20;
    }

    public void show(final int num){
        System.out.println(num);
    }

    public static void main(String[] args) {

        int num = 10;
        num = num + 5;

        FinalTest test = new FinalTest();
//          test.setDown(5);
            test.show(10);
        }
    }

    final class FianlA{
    }

    //class B extends FinalA{     // 错误，不能被继承。
    //
    //}

    //class C extends String{
    //
    //}

    class AA{
        public final void show(){
        }
    }
```



## 6.5 抽象类与抽象方法

**随着继承层次中一个个新子类的定义，类变得越来越具体，而父类则更一般，更通用。类的设计应该保证父类和子类能够共享特征。有时将一个父类设计得非常抽象，以至于它没有具体的实例，这样的类叫做抽象类。**

### abstract 关键字的使用：

1.abstract：抽象的；

2.abstract：可以用来修饰的结构：类、方法；

3.abstract：修饰类：抽象类：
> 此类不能实例化；
> 抽象类中一定有构造器，便于子类实例化时调用 ( 涉及：子类对象实例化全过程 )；
> 开发中，都会提供抽象类的子类，让子类对象实例化，实现相关的操作。

4.abstract 修饰方法：抽象方法：
> 抽象方法，只有方法的声明，没有方法体；
> 包含抽象方法的类，一定是一个抽象类。反之，抽象类中可以没有抽象方法；
> 若子类重写了父类中所有的抽象方法后，此子类方可实例化；
> 若子类没有重写父类中的所有抽象方法，则此子类也是一个抽象类，需要使用 abstract 修饰；

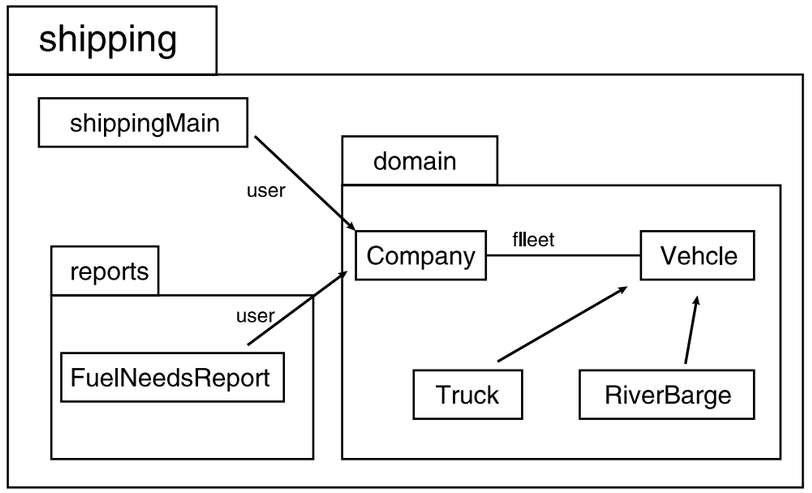
abstract 使用上的注意点：
1.abstract 不能用来修饰变量、代码块、构造器；
2.abstract 不能用来修饰私有方法、静态方法、final 的方法、final 的类。

```
public class AbstractTest {
	public static void main(String[] args) {
		// 一旦 Person 类抽象了，就不可实例化
//		Person p1 = new Person();  //	p1.eat();
	}
}
abstract class Creature{
	public abstract void breath();
}
abstract class Person extends Creature{
	String name;
	int age;
	public Person(){
	}
	public Person(String name,int age){
		this.name = name;
		this.age = age;
	}
	// 不是抽象方法
//	public void eat(){
//		System.out.println(" 人吃饭 ");
//	}
	// 抽象方法
	public abstract void eat();
	public void walk(){
		System.out.println(" 人走路 ");
	}
}
class Student extends Person{
	public Student(String name,int age){
		super(name,age);
	}
	public void eat(){
		System.out.println(" 学生应该多吃有营养的。 ");
	}
	@Override
	public void breath() {
		System.out.println(" 学生应该呼吸新鲜的无雾霾空气 ");
	}
}
```



## 6.5.1 抽象类应用

抽象类是用来模型化那些父类无法确定全部实现，而是由其子类提供具体实现的对象的类。




在航运公司系统中，Vehicle类需要定义两个方法分别计算运输工具的燃料效率和行驶距离。

问题：卡车 (Truck) 和驳船 (RiverBarge) 的燃料效率和行驶距离的计算方法完全不同。Vehicle 类不能提供计算方法，但子类可以。

Java 允许类设计者指定：超类声明一个方法但不提供实现，该方法的实现由子类提 供。这样的方法称为抽象方法。有一个或更多抽象方法的类称为抽象类。Vehicle 是一个抽象类，有两个抽象方法。注意：抽象类不能实例化 new Vihicle() 是非法的。

```
public abstract class Vehicle{
	public abstract double calcFuelEfficiency();
	// 计算燃料效率的抽象方法
	public abstract double calcTripDistance();
	// 计算行驶距离的抽象方法
}

public class Truck extends Vehicle{
	public double calcFuelEfficiency(){
		// 写出计算卡车的燃料效率的具体方法
	}
	public double calcTripDistance(){
		// 写出计算卡车行驶距离的具体方法
	}
}
```

```
public class RiverBarge extends Vehicle{
        public double calcFuelEfficiency() {
                // 写出计算驳船的燃料效率的具体方法
        }
        public double calcTripDistance( ) {
                // 写出计算驳船行驶距离的具体方法
        }
}
```

## 6.5.2 练习

编写一个 Employee 类，声明为抽象类，
包含如下三个属性：name，id，salary。
提供必要的构造器和抽象方法：work()。
对于 Manager 类来说，他既是员工，还具有奖金 (bonus) 的属性。
请使用继承的思想，设计 CommonEmployee 类和 Manager 类，
要求类中提供必要的方法进行属性访问。

```
public abstract class Employee {
        private String name;
        private int id;
        private double salary;

        public Employee(){
                super();
        }

        public Employee(String name, int id, double salary) {
                super();
                this.name = name;
                this.id = id;
                this.salary = salary;
        }

        public abstract void work();
}
```



### Manager 类

对于 Manager 类来说，他既是员工，还具有奖金 (bonus) 的属性。

```
public class Manager extends Employee{
        private double bonus;     // 奖金

        public Manager(double bonus) {
                super();
                this.bonus = bonus;
        }

        public Manager(String name, int id, double salary, double
bonus) {
                super(name, id, salary);
                this.bonus = bonus;
        }
        @Override
        public void work() {
                System.out.println(" 管理员工，提高公司运行效率。");
        }
}
```

### CommonEmployee 类

```
public class CommonEmployee extends Employee {
        @Override
        public void work() {
                System.out.println(" 员工在一线车间生产产品。");
        }
 }
```

## 测试类

请使用继承的思想，设计 CommonEmployee 类和 Manager 类，

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

		Employee manager = new Manager(" 库克 ",
1001,5000,50000);

		manager.work();

		CommonEmployee commonEmployee = new
CommonEmployee();
		commonEmployee.work();
	}
}
```



## 6.5.3 创建抽象类的匿名子类对象

```
public class Num {
}
abstract class Creature{
	public abstract void breath();
}
abstract class Person extends Creature{
	String name;
	int age;
	public Person(){
	}
	public Person(String name,int age){
		this.name = name;
		this.age = age;
	}
	// 不是抽象方法
//	public void eat(){
//		System.out.println(" 人吃饭 ");
//	}
	// 抽象方法
	public abstract void eat();
	public void walk(){
		System.out.println(" 人走路 ");
	}
}
class Student extends Person{
	public Student(String name,int age){
		super(name,age);
	}
	public Student(){
	}
	public void eat(){
		System.out.println(" 学生应该多吃有营养的。 ");
	}
	@Override
	public void breath() {
		System.out.println(" 学生应该呼吸新鲜的无雾霾空气 ");
	}
}
```

## PersonTest 类

## 抽象类的匿名子类

```
public class PersonTest {
	public static void main(String[] args) {
		method(new Student());  // 匿名对象

		Worker worker = new Worker();
		method1(worker);    // 非匿名的类非匿名的对象

		method1(new Worker()); // 非匿名的类匿名的对象

		System.out.println("**********************");

		// 创建了一个匿名子类的对象 :p
		Person p = new Person(){
			@Override
			public void eat() {
				System.out.println(" 吃东西 ");
			}

			@Override
			public void breath() {
				System.out.println(" 呼吸空气 ");
			}
		};
		method1(p);
		System.out.println("**********************");
		// 创建匿名子类的匿名对象
		method1(new Person(){
			@Override
			public void eat() {
				System.out.println(" 吃零食 ");
			}

			@Override
			public void breath() {
				System.out.println(" 云南的空气 ");
			}
		});
	}
	public static void method1(Person p){
		p.eat();
		p.walk();
	}

	public static void method(Student s){

	}
}

class Worker extends Person{

	@Override
	public void eat() {
	}

	@Override
	public void breath() {
	}
}
```

## 6.5.4 多态的应用：模板方法设计模式 (TemplateMethod)

抽象类体现的就是一种模板模式的设计，抽象类作为多个子类的通用模板，子类在抽象类的基础上进行扩展、改造，但子类总体上会保留抽象类的行为方式。

### 解决的问题：

当功能内部一部分实现是确定的，一部分实现是不确定的。这时可以把不确定的部分暴露出去，让子类去实现。

换句话说，在软件开发中实现一个算法时，整体步骤很固定、通用，这些步骤已经在父类中写好了。但是某些部分易变，易变部分可以抽象出来，供不同子类实现。这就是一种模板模式。

### 模板方法设计模式是编程中经常用得到的模式。各个框架、类库中都有他的影子，比如常见的有：

数据库访问的封装
Junit 单元测试
JavaWeb 的 Servlet 中关于 doGet/doPost 方法调用
Hibernate 中模板程序
Spring 中 JDBCTemlate、HibernateTemplate 等



### 例 1

抽象类的应用：模板方法的设计模式

```
public class TemplateTest {
	public static void main(String[] args) {
		SubTemlate t = new SubTemlate();
		t.sendTime();
	}
}

abstract class Template{
	// 计算某段代码执行所需花费的时间
	public void sendTime(){
		long start = System.currentTimeMillis();
		code();	// 不确定部分，易变的部分
		long end = System.currentTimeMillis();
		System.out.println(" 花费的时间为 :" + (end - start));
	}
	public abstract void code();
}

class SubTemlate extends Template{
	@Override
	public void code() {
		for(int i = 2;i <= 1000;i++){
			boolean isFlag = true;
			for(int j = 2;j <= Math.sqrt(i);j++){
				if(i % j == 0){
					isFlag = false;
					break;
				}
			}
			if(isFlag){
				System.out.println(i);
			}
		}
	}
}
```

## 例 2

抽象类的应用：模板方法的设计模式

```
public class TemplateMethodTest {
	public static void main(String[] args) {
		BankTemplateMethod btm = new DrawMoney();
		btm.process();
		BankTemplateMethod btm2 = new ManageMoney();
		btm2.process();
	}
}

abstract class BankTemplateMethod {
	// 具体方法
	public void takeNumber() {
		System.out.println(" 取号排队 ");
	}
	public abstract void transact(); // 办理具体的业务 // 钩子方法
	public void evaluate() {
		System.out.println(" 反馈评分 ");
	}
	// 模板方法，把基本操作组合到一起，子类一般不能重写
	public final void process() {
		this.takeNumber();
		this.transact();
// 像个钩子，具体执行时，挂哪个子类，就执行哪个子类的实现代码
		this.evaluate();
	}
}
class DrawMoney extends BankTemplateMethod {
	public void transact() {
		System.out.println(" 我要取款！！！ ");
	}
}

class ManageMoney extends BankTemplateMethod {
	public void transact() {
		System.out.println(" 我要理财！我这里有 2000 万美元 !!");
	}
}
```



# 6.6 接口 (interface)

## 6.6.1 概述

接口的使用：
1. 接口使用 interface 来定义。

2. 在 Java 中：接口和类是并列的两个结构。

3. 如何去定义两个接口：定义接口中的成员。
	> 3.1 JDK7 及以前：只能定义全局常量和抽象方法。
		> 全局常量：public static final 的，但是书写中，可以省略不写。
		> 抽象方法：public abstract 的。
	> 3.2 JDK8：除了全局常量和抽象方法之外，还可以定义静态方法、默认方法 ( 略 )。

4. 接口中不能定义构造器！意味着接口不可以实例化。

5. Java 开发中，接口通过让类去实现 (implements) 的方式来使用。
	如果实现类覆盖了接口中所有的抽象方法，则此实现类就可以实例化。
	如果实现类没有覆盖接口中所有的抽象方法，则此实现类仍为一个抽象类。

6. Java 类可以实现多个接口 - - - > 弥补了 Java 单继承性的局限性，
	格式：class AA extends BB implements CC,DD,EE。
7. 接口与接口之间是继承，而且可以多继承。

8. 接口的具体使用，体现多态性。

	接口的主要用途就是被实现类实现。（面向接口编程）

9. 接口，实际可以看作是一种规范。

```java
public class InterfaceTest {
	public static void main(String[] args) {
		System.out.println(Flayable.MAX_SPEED);
		System.out.println(Flayable.MIN_SPEED);
	}
}

interface Flayable{
	// 全局变量
	public static final int MAX_SPEED = 7900;
	int MIN_SPEED = 1; // 省略了 public static final

	// 抽象方法
	public abstract void fly();

	void stop(); // 省略了 public abstract
	//Interfaces cannot have constructors
//	public Flayable(){
//
//	}
}
interface Attackable{
	void attack();
}

class Plane implements Flayable{
	@Override
	public void fly() {
		System.out.println(" 飞机通过引擎起飞 ");
	}
	@Override
	public void stop() {
		System.out.println(" 驾驶员减速停止 ");
	}
}
abstract class Kite implements Flayable{
	@Override
	public void fly() {
	}
}
```



```java
class Bullet extends Object implements Flayable,Attackable,CC{

	@Override
	public void attack() {
		// TODO Auto-generated method stub
	}

	@Override
	public void fly() {
		// TODO Auto-generated method stub
	}

	@Override
	public void stop() {
		// TODO Auto-generated method stub
	}

	@Override
	public void method1() {
		// TODO Auto-generated method stub
	}

	@Override
	public void method2() {
		// TODO Auto-generated method stub

	}
}

interface AA{
	void method1();
}
interface BB{
	void method2();
}
interface CC extends AA,BB{

}
```

## 6.6.2 举例

接口的使用
1. 接口使用上也满足多态性。
2. 接口，实际上就是定义了一种规范。
3. 开发中，体会面向接口编程！

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

        Computer com = new Computer();
        //1. 创建了接口的非匿名实现类的非匿名对象
        Flash flash = new Flash();
        com.transferData(flash);
        //2. 创建了接口的非匿名实现类的匿名对象
        com.transferData(new Printer());
        //3. 创建了接口的匿名实现类的非匿名对象
        USB phone = new USB(){
            @Override
            public void start() {
                System.out.println(" 手机开始工作 ");
            }
            @Override
            public void stop() {
                System.out.println(" 手机结束工作 ");
            }
        };
        com.transferData(phone);
        //4. 创建了接口的匿名实现类的匿名对象
        com.transferData(new USB(){
            @Override
            public void start() {
                System.out.println("mp3 开始工作 ");
            }
            @Override
            public void stop() {
                System.out.println("mp3 结束工作 ");
            }
        });
    }
}
```



```
class Computer{

    public void transferData(USB usb){ //USB usb = new Flash();
        usb.start();
        System.out.println(" 具体传输数据的细节 ");
        usb.stop();
    }
}

interface USB{
    // 常量：定义了长、宽、最大、最小的传输速度等
    void start();

    void stop();
}

class Flash implements USB{

    @Override
    public void start() {
        System.out.println("U 盘开始工作 ");
    }

    @Override
    public void stop() {
        System.out.println("U 盘结束工作 ");
    }
}

class Printer implements USB{
    @Override
    public void start() {
        System.out.println(" 打印机开启工作 ");
    }

    @Override
    public void stop() {
        System.out.println(" 打印机结束工作 ");
    }
}
```

## 6.6.3 接口的应用：代理模式 (Proxy)

**代理模式是 Java 开发中使用较多的一种设计模式。代理设计就是为其他对象提供一种代理以控制对这个对象的访问。**

**接口的应用 : 代理模式**

```
public class NetWorkTest {
	public static void main(String[] args) {
		Server server = new Server();
//		server.browse();
		ProxyServer proxyServer = new ProxyServer(server);
		proxyServer.browse();
	}
}
interface NetWork{
	public void browse();
}
// 被代理类
class Server implements NetWork{
	@Override
	public void browse() {
		System.out.println(" 真实的服务器来访问网络 ");
	}
}
// 代理类
class ProxyServer implements NetWork{
	private NetWork work;
	public ProxyServer(NetWork work){
		this.work = work;
	}
	public void check(){
		System.out.println(" 联网前的检查工作 ");
	}
	@Override
	public void browse() {
		check();
		work.browse();
	}	}
```



**应用场景：**

**安全代理：屏蔽对真实角色的直接访问。**
**远程代理：通过代理类处理远程方法调用（RMI）。**
**延迟加载：先加载轻量级的代理对象，真正需要再加载真实对象。**

比如你要开发一个大文档查看软件，大文档中有大的图片，有可能一个图片有 100MB，在打开文件时，不可能将所有的图片都显示出来，这样就可以使用代理模式，当需要查看图片时，用 proxy 来进行大图片的打开。

**分类：**

**静态代理（静态定义代理类）**
**动态代理（动态生成代理类）**
**JDK 自带的动态代理，需要反射等知识**

### 6.6.4 接口的应用：工厂模式

**接口和抽象类之间的对比**

| No. | 区别点 | 抽象类 | 接口 |
|---|---|---|---|
| 1 | 定义 | 包含抽象方法的类 | 主要是抽象方法和全局常量的集合 |
| 2 | 组成 | 构造方法、抽象方法、普通方法、<br>常量、变量 | 常量、抽象方法、<br>(jdk8.0:默认方法、静态方法) |
| 3 | 使用 | 子类继承抽象类(extends) | 子类实现接口(implements) |
| 4 | 关系 | 抽象类可以实现多个接口 | 接口不能继承抽象类，<br>但允许继承多个接口 |
| 5 | 常见设计<br>模式 | 模板方法 | 简单工厂、工厂方法、<br>代理模式 |
| 6 | 对象 | 都通过对象的多态性产生实例化对象 | |
| 7 | 局限 | 抽象类有单继承的局限 | 接口没有此局限 |
| 8 | 实际 | 作为一个模板 | 是作为一个标准或是表示一种能力 |
| 9 | 选择 | 如果抽象类和接口都可以使用的话，<br>优先使用接口，因为避免单继承的局限 | |

**在开发中，常看到一个类不是去继承一个已经实现好的类，而是要么继承抽象类，要么实现接口。**



## 6.7 Java 8 中关于接口的改进

Java 8 中，你可以为接口添加静态方法和默认方法。从技术角度来说，这是完全合法的，只是它看起来违反了接口作为一个抽象定义的理念。

**静态方法：**

使用 static 关键字修饰。可以通过接口直接调用静态方法，并执行其方法体。我们经常在相互一起使用的类中使用静态方法。你可以在标准库中找到像 Collection/Collections 或者 Path/Paths 这样成对的接口和类。

**默认方法：**

默认方法使用 default 关键字修饰。可以通过实现类对象来调用。我们在已有的接口中提供新方法的同时，还保持了与旧版本代码的兼容性。比如：java 8 API 中对 Collection、List、Comparator 等接口提供了丰富的默认方法。

## 6.8 类的成员之五：内部类

1.Java 中允许将一个类 A 声明在另一个类 B 中，则类 A 就是内部类，类 B 就是外部类。
2. 内部类的分类：成员内部类 VS 局部内部类（方法内、代码块内、构造器内）。
3. 成员内部类
> 作为外部类的成员：
- 调用外部类的结构
- 可以被 static 修饰
- 可以被 4 种不同的权限修饰
> 作为一个类：
- 类内可以定义属性、方法、构造器等
- 可以被 final 修饰，表示此类不能被继承
言外之意，不使用 final，就可以被继承
- 可以 abstract 修饰
4. 关注如下的 3 个问题：
> 如何实例化成员内部类的对象
> 如何在成员内部类中区分调用外部类的结构
> 开发中局部内部类的使用
见《InnerClassTest1.java》



# 七 异 常

## 7.1 异常概述与异常体系结构

在使用计算机语言进行项目开发的过程中，即使程序员把代码写得尽善尽美，在系统的运行过程中仍然会遇到一些问题，因为很多问题不是靠代码能够避免的，比如：客户输入数据的格式，读取文件是否存在，网络是否始终保持通畅等等。

### 异常：

在 Java 语言中，将程序执行中发生的不正常情况称为“异常”。（开发过程中的语法错误和逻辑错误不是异常）

### Java 程序在执行过程中所发生的异常事件可分为两类：

Error：Java 虚拟机无法解决的严重问题。如：JVM 系统内部错误、资源耗尽等严重情况。比如：StackOverflowError 和 OOM。一般不编写针对性的代码进行处理。

Java 虚拟机无法解决的严重问题。
如：JVM 系统内部错误、资源耗尽等严重情况。
比如：StackOverflowError 和 OOM。
一般不编写针对性的代码进行处理。

```
public class ErrorTest {
	public static void main(String[] args) {
		//1. 栈溢出：java.lang.StackOverflowError
//		main(args);
		//2. 堆溢出：java.lang.OutOfMemoryError
//		Integer[] arr = new Integer[1024*1024*1024];
	}
}
```

**Exception：其它因编程错误或偶然的外在因素导致的一般性问题，可以使用针对性的代码进行处理。例如：**

空指针访问
试图读取不存在的文件
网络连接中断
数组角标越界

**对于这些错误，一般有两种解决方法：**

一是遇到错误就终止程序的运行。

另一种方法是由程序员在编写程序时，就考虑到错误的检测、错误消息的提示，以及错误的处理。

捕获错误最理想的是在编译期间，但有的错误只有在运行时才会发生。比如：除数为 0，数组下标越界等。

**异常分类：编译时异常和运行时异常**

运行时异常：

是指编译器不要求强制处置的异常。一般是指编程时的逻辑错误，是程序员应该积极避免其出现的异常。java.lang.RuntimeException 类及它的子类都是运行时异常。

对于这类异常，可以不作处理，因为这类异常很普遍，若全处理可能会对程序的可读性和运行效率产生影响。

编译时异常：

是指编译器要求必须处置的异常。即程序在运行时由于外界因素造成的一般性异常。编译器要求 Java 程序必须捕获或声明所有编译时异常。

对于这类异常，如果程序不处理，可能会带来意想不到的结果。



# 7.2 常见异常

## 一、java 异常体系结构

```
java.lang.Throwable
        |----java.lang.Error: 一般不编写针对性的代码进行处理
        |----java.lang.Exception: 可以进行异常处理
                |---- 编译时异常 (checked)
                        |----IOEXception
                                |----FileNotFoundException
                        |----ClassNotFoundException
                |---- 运行时异常 (unchecked, runtimeException)
                        |----NullPointerException
                        |----ArrayIndexOutOfBoundsException
                        |----ClassCaseException
                        |----NumberFormatException
                        |----InputMismatchException
                        |----ArithmaticException
```

```java
public class ExceptionTest {
    // ****************** 以下是编译时异常 ***************************
    @Test
    public void test7() {
//        File file = new File("hello.txt");
//        FileInputStream fis = new FileInputStream(file);
//
//        int data = fis.read();
//        while(data != -1){
//            System.out.print((char)data);
//            data = fis.read();
//        }
//
//        fis.close();
    }

    // ****************** 以下是运行时异常 ***************************
    // ArithmeticException
    @Test
    public void test6() {
        int a = 10;
        int b = 0;
        System.out.println(a / b);
    }
```

```java
// InputMismatchException
@Test
public void test5() {
        Scanner scanner = new Scanner(System.in);
        int score = scanner.nextInt();
        System.out.println(score);

        scanner.close();
}

// NumberFormatException
@Test
public void test4() {
        String str = "123";
        str = "abc";
        int num = Integer.parseInt(str);
}
// ClassCaseException
@Test
public void test3() {
        Object obj = new Date();
        String str = (String)obj;
    }

    // ArrayIndexOutOfBoundsException
    @Test
    public void test2() {
        // int[] arr = new int[10];
        // System.out.println(arr[10]);

        // String str = "abc";
        // System.out.println(str.charAt(3));
    }
    // NullPointerException
    @Test
    public void test1() {
        // int[] arr = null;
        // System.out.println(arr[3]);
        // String str = "abc";
        // str = null;
        // System.out.println(str.charAt(0));
    }
}
```



## 7.3 异常处理机制一：try-catch-finally

**在编写程序时，经常要在可能出现错误的地方加上检测的代码，如进行 x/y 运算时，要检测分母为 0，数据为空，输入的不是数据而是字符等。过多的 if-else 分支会导致程序的代码加长、臃肿，可读性差。因此采用异常处理机制。**

### Java 异常处理：

Java 采用的异常处理机制，是将异常处理的程序代码集中在一起，与正常的程序代码分开，使得程序简洁、优雅，并易于维护。

方式一：try-catch-finally

方式二：throws + 异常类型

### Java 异常处理的方式：try-catch-finally

**try：**

捕获异常的第一步是用 try{…} 语句块选定捕获异常的范围，将可能出现异常的代码放在 try 语句块中。

**catch(Exceptiontypee)：**

在 catch 语句块中是对异常对象进行处理的代码。每个 try 语句块可以伴随一个或多个 catch 语句，用于处理可能产生的不同类型的异常对象。

捕获异常的有关信息：与其它对象一样，可以访问一个异常对象的成员变量或调用它的方法。

getMessage() 获取异常信息，返回字符串。

printStackTrace() 获取异常类名和异常信息，以及异常出现在程序中的位置。返回值 void。

**finally：**

捕获异常的最后一步是通过 finally 语句为异常处理提供一个统一的出口，使得在控制流转到程序的其它部分以前，能够对程序的状态作统一的管理。

不论在 try 代码块中是否发生了异常事件，catch 语句是否执行，catch 语句是否有异常，catch 语句中是否有 return，finally 块中的语句都会被执行。

finally 语句和 catch 语句是任选的。

# 一、异常的处理：抓抛模型

过程一：“ 抛 ”：程序在征程执行过程中，一旦出现异常，就会在异常代码处生成一个对应异常类的对象并将此对象抛出。

一旦抛出对象以后，其后的代码就不再执行。

关于异常对象的产生：

① 系统自动生成的异常对象；

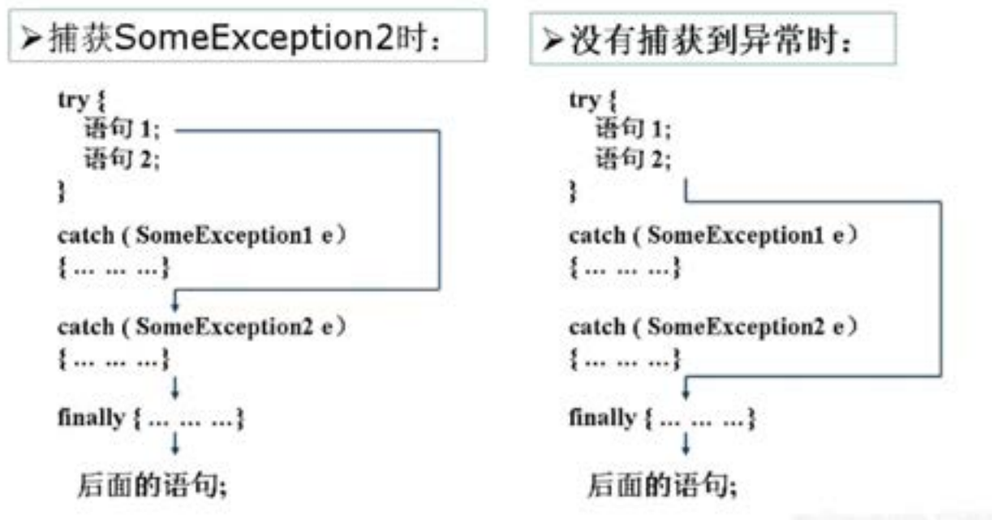
② 手动生成的一个异常对象，并抛出（throw）；

过程二：“ 抓 ”：可以理解为异常的处理方式：

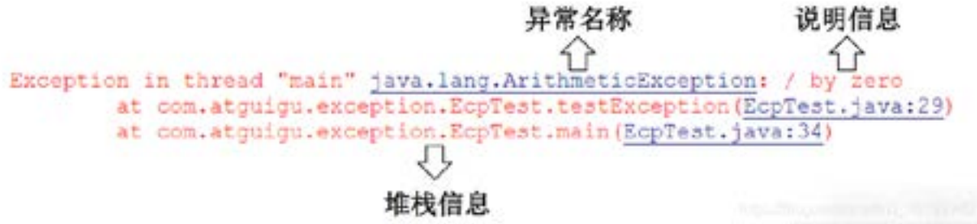
① try - catch - finally　　② throws

# 二、try - catch - finally 的使用

```
try{
        // 可能出现异常的代码
}catch( 异常类型 1 变量名 1){
        // 处理异常的方式 1
}catch( 异常类型 2 变量名 2){
        // 处理异常的方式 2
}catch( 异常类型 3 变量名 3){
        // 处理异常的方式 3
}
...
finally{
        // 一定会执行的代码
}
```






## 说明：

1.finally 是可选的。

2. 使用 try 将可能出现异常代码包装起来，在执行过程中，一旦出现异常，就会生成一个对应异常类的对象，根据此对象的类型，去 catch 中进行匹配。

3. 一旦 try 中的异常对象匹配到某一个 catch 时，就进入 catch 中进行异常的处理。一旦处理完成，就跳出当前的 try - catch 结构（在没有写 finally 的情况）。继续执行其后的代码。

4.catch 中的异常类型如果没有子父类关系，则谁声明在上，谁声明在下无所谓。

catch 中的异常类型如果满足子父类关系，则要求子类一定声明在父类的上面。否则，报错。

5. 常用的异常对象处理的方式：

① String  getMessage()　② printStackTrace()

6. 在 try 结构中声明的变量，再出了 try 结构以后，就不能再被调用，例 65 行 :System.out.println(num)。

7.try - catch - finally 结构可以嵌套。

体会 1：使用 try - catch - finally 处理编译时异常，使得程序在编译时就不再报错，但是运行时仍可能报错。

相当于我们使用 try - catch - finally 将一个编译时可能出现的异常，延迟到运行时出现。

体会 2：开发中，由于运行时异常比较常见，所以我们通常就不针对运行时异常编写 try - catch - finally 了。

针对于编译时异常，我们说一定要考虑异常的处理。

```
public class ExceptionTest1 {
	@Test
	public void test2(){
		try{
			File file = new File("hello.txt");
			FileInputStream fis = new FileInputStream(file);
			int data = fis.read();
			while(data != -1){
				System.out.print((char)data);
				data = fis.read();
			}
			fis.close();
		}catch(FileNotFoundException e){
			e.printStackTrace();
		}catch(IOException e){
			e.printStackTrace();
		}
	}
	@Test
	public void test1(){
		String str = "123";
		str = "abc";
		try{
			int num = Integer.parseInt(str);
			System.out.println("hello-----1");
		}catch(NumberFormatException e){
//			System.out.println(" 出现数值转换异常了，不要着急 ....");
			//String getMessage():
//			System.out.println(e.getMessage());
			//printStackTrace():
			e.printStackTrace();
		}catch(NullPointerException e){
			System.out.println(" 出现空指针异常了，不要着急 ....");
		}catch(Exception e){
			System.out.println(" 出现异常了，不要着急 ....");
		}
//		System.out.println(num);
		System.out.println("hello----2");
	}
}
```



# finally 的使用

## try-catch-finally 中 finally 的使用：

1.finally 是可选的。

2.finally 中声明的是一定会被执行的代码。即使 catch 中又出现异常了，try 中有 return 语句，catch 中有 return 语句等情况。

3. 像数据库连接、输入输出流、网络编程 Socket 等资源，JVM 是不能自动的回收的，我们需要自己手动的进行资源的释放。此时的资源释放，就需要声明在 finally 中。

```
public class FinallyTest {

	@Test
	public void test2() {
		FileInputStream fis = null;
		try {
			File file = new File("hello1.txt");
			// 文件可能不存在，而出现异常
			fis = new FileInputStream(file);

			int data = fis.read();
			while (data != -1) {
				System.out.print((char) data);
				data = fis.read();
			}

		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		} finally {
			try {
				if (fis != null)
					fis.close();
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
	}
```

```java
@Test
public void testMethod() {
    int num = method();
    System.out.println(num);
}

public int method() {

    try {
        int[] arr = new int[10];
        System.out.println(arr[10]);
        return 1;
    } catch (ArrayIndexOutOfBoundsException e) {
        e.printStackTrace();
        return 2;
    } finally {
        System.out.println(" 我一定会被执行 ");
        return 3;
    }
}
```

```java
    @Test
    public void test1() {
        try {
            int a = 10;
            int b = 0;
            System.out.println(a / b);
        } catch (ArithmeticException e) {
            // e.printStackTrace();

            int[] arr = new int[10];
            System.out.println(arr[10]);

        } catch (Exception e) {
            e.printStackTrace();
        }
        // System.out.println(" 我好慢呀 ~~~");
        finally {
            System.out.println(" 我好慢呀 ~~~");
        }
    }
}
```



# 7.4 异常处理机制二：throws

## 声明抛出异常是 Java 中处理异常的第二种方式。

如果一个方法 ( 中的语句执行时 ) 可能生成某种异常，但是并不能确定如何处理这种异常，则此方法应显示地声明抛出异常，表明该方法将不对这些异常进行处理，而由该方法的调用者负责处理。

在方法声明中用 throws 语句可以声明抛出异常的列表，throws 后面的异常类型可以是方法中产生的异常类型，也可以是它的父类。

## 异常处理的方式二：throws + 异常类型

声明抛出异常是 Java 中处理异常的第二种方式。

如果一个方法 ( 中的语句执行时 ) 可能生成某种异常，但是并不能确定如何处理这种异常，则此方法应显示地声明抛出异常，表明该方法将不对这些异常进行处理，而由该方法的调用者负责处理。

在方法声明中用 throws 语句可以声明抛出异常的列表，throws 后面的异常类型可以是方法中产生的异常类型，也可以是它的父类。

1. "throws + 异常类型 " 写在方法的声明处。指明此方法执行时，可能会抛出的异常类型。
一旦当方法体执行时，出现异常，仍会在异常代码处生成一个异常类的对象，此对象满足 throws 后异常类型时，就会被抛出。异常代码后续的代码，就不再执行！
关于异常对象的产生：① 系统自动生成的异常对象
② 手动生成一个异常对象，并抛出 (throw)

2. 体会：try - catch - finally：真正的将异常给处理掉了。
throws 的方式只是将异常抛给了方法的调用者。
并没有真正将异常处理掉。

3. 开发中如何选择使用 try - catch - finally 还是使用 throws ？
3.1 如果父类中被重写的方法没有 throws 方式处理异常，则子类重写的方法也不能使用 throws，意味着如果子类重写的方法中有异常，必须使用 try - catch - finally 方式处理。
3.2 执行的方法 a 中，先后又调用了另外的几个方法，这几个方法是递进关系执行的。我们建议这几个方法使用 throws 的方式进行处理。而执行的方法 a 可以考虑使用 try - catch - finally 方式进行处理。

```
public class ExceptionTest2 {

	public static void main(String[] args){
		try {
			method2();
		} catch (IOException e) {
			e.printStackTrace();
		}

		method3();
	}

	public static void method3(){
		try {
			method2();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	public static void method2() throws IOException{
		method1();
	}

	public static void method1() throws FileNotFoundException,IOE
xception{
		File file = new File("hello1.txt");
		FileInputStream fis = new FileInputStream(file);

		int data = fis.read();
		while(data != -1){
			System.out.print((char)data);
			data = fis.read();
		}

		fis.close();

		System.out.println("hahaha!");
	}
}
```




**重写方法声明抛出异常的原则**

**方法重写的规则之一：**

**子类重写的方法抛出的异常类型不大于父类被重写的方法抛出的异常类型**

```
public class OverrideTest {
	public static void main(String[] args) {
		OverrideTest test = new OverrideTest();
		test.display(new SubClass());
	}
	public void display(SuperClass s){
		try {
			s.method();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
class SuperClass{
	public void method() throws IOException{
	}
}
class SubClass extends SuperClass{
	public void method()throws FileNotFoundException{
	}
}
```

## 7.5 手动抛出异常

Java 异常类对象除在程序执行过程中出现异常时由系统自动生成并抛出，也可根据需要使用人工创建并抛出。

首先要生成异常类对象，然后通过 throw 语句实现抛出操作 ( 提交给 Java 运行环境 )。

可以抛出的异常必须是 Throwable 或其子类的实例。下面的语句在编译时将会产生语法错误：

```
public class StudentTest {
	public static void main(String[] args) {
		try {
			Student s = new Student();
//			s.regist(1001);
			s.regist(-1001);
			System.out.println(s);
		} catch (Exception e) {
//			e.printStackTrace();
			System.out.println(e.getMessage());
		}
	}

	class Student{
		private int id;
		public void regist(int id) throws Exception{
			if(id > 0){
				this.id = id;
			}else{
//				System.out.println(" 您输入的数据非法！ ");
				// 手动抛出异常
//				throw new RuntimeException(" 您输入的数据非法！ ");
				throw new Exception(" 您输入的数据非法！ ");
			}
		}
		@Override
		public String toString() {
			return "Student [id=" + id + "]";
		}
	}
```



## 7.6 用户自定义异常类

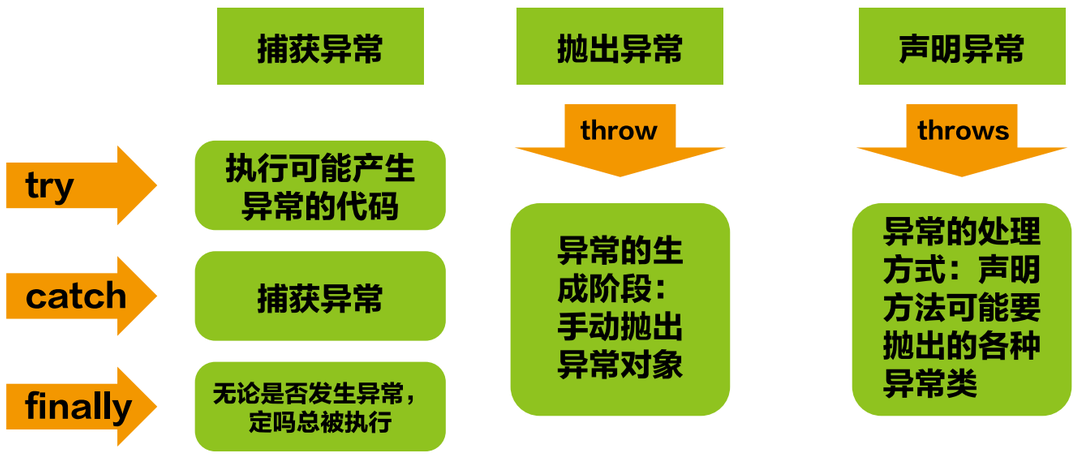
**一般地，用户自定义异常类都是 RuntimeException 的子类。**
**自定义异常类通常需要编写几个重载的构造器。**
**自定义异常需要提供 serialVersionUID。**
**自定义的异常通过 throw 抛出。**
**自定义异常最重要的是异常类的名字，当异常出现时，可以根据名字判断异常类型。**

### 如何自定义异常类？

1. 继承于现有的异常结构：RuntimeException 、Exception
2. 提供全局常量：serialVersionUID
3. 提供重载的构造器

```
public class MyException extends RuntimeException{
	static final long serialVersionUID = -7034897193246939L;

	public MyException(){
	}

	public MyException(String msg){
		super(msg);
	}
}
```

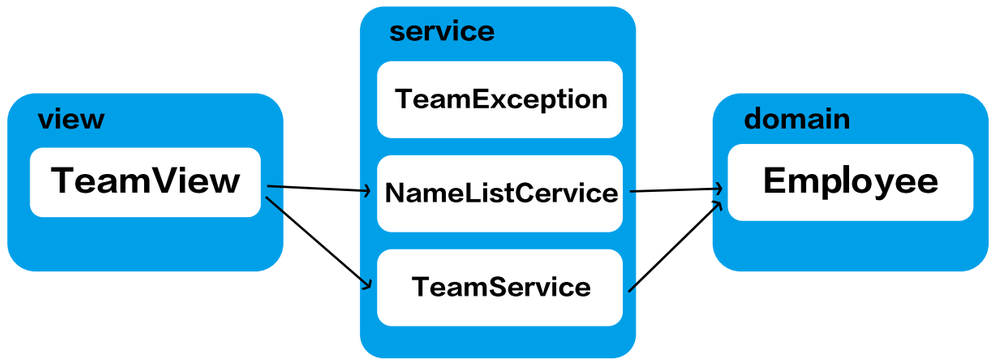
## 7.7 异常总结






# 项目三

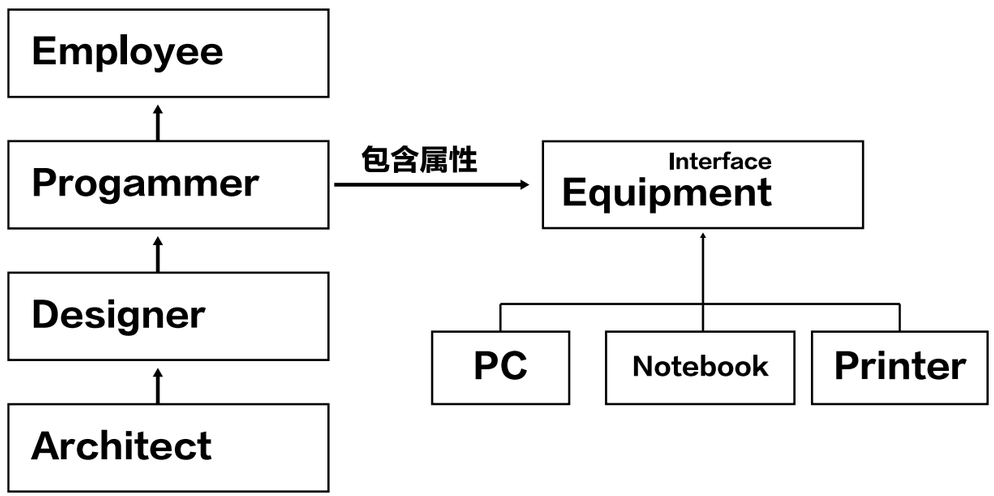
## 软件设计模块




view 模块为主控模块，负责菜单的显示和处理用户操作。
service 模块为实体对象，（Employee 及其子类如程序员等）的管理模块。
NameListService 和 TeamService 类分用格子的数组来管理公司员工和开发团队成员对象。
domain 模块为 Employee 及其子类等 JavaBean 类所在的包。

## 软件设计结构

domain 模块中包含了所有实体类：



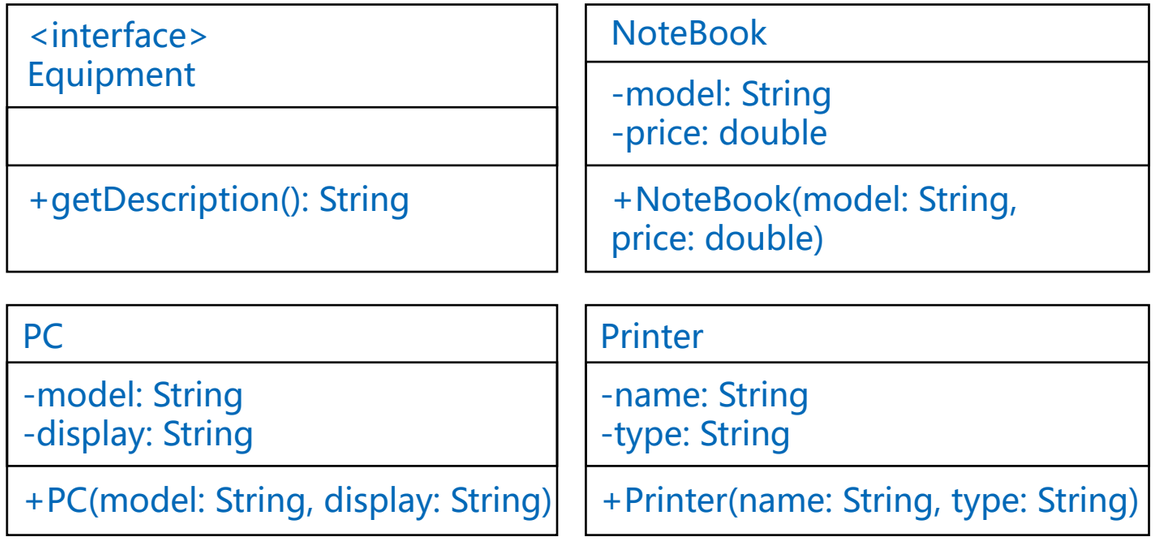

其中程序员（Programmer）及其子类，均会领用某种电子设备（Equipment）

## 第一步：创建项目基本组件

1、完成以下工作：
   1. 创建 TeamSchedule 项目。
   2. 按照设计要求，创建所有包。
   3. 将项目提供的几个类复制到相应的包中。
      （view 包中，TSUtility.java；service 包中：Data.java）

2、按照设计要求，在 domain 包中，创建 Equipment 接口及其各实现子类代码。
3、按照设计要求，在 doamin 包中，创建 Employee 类及其各子类代码。
4、检验代码的正确性。

Equipment 接口及其实现子类的设计

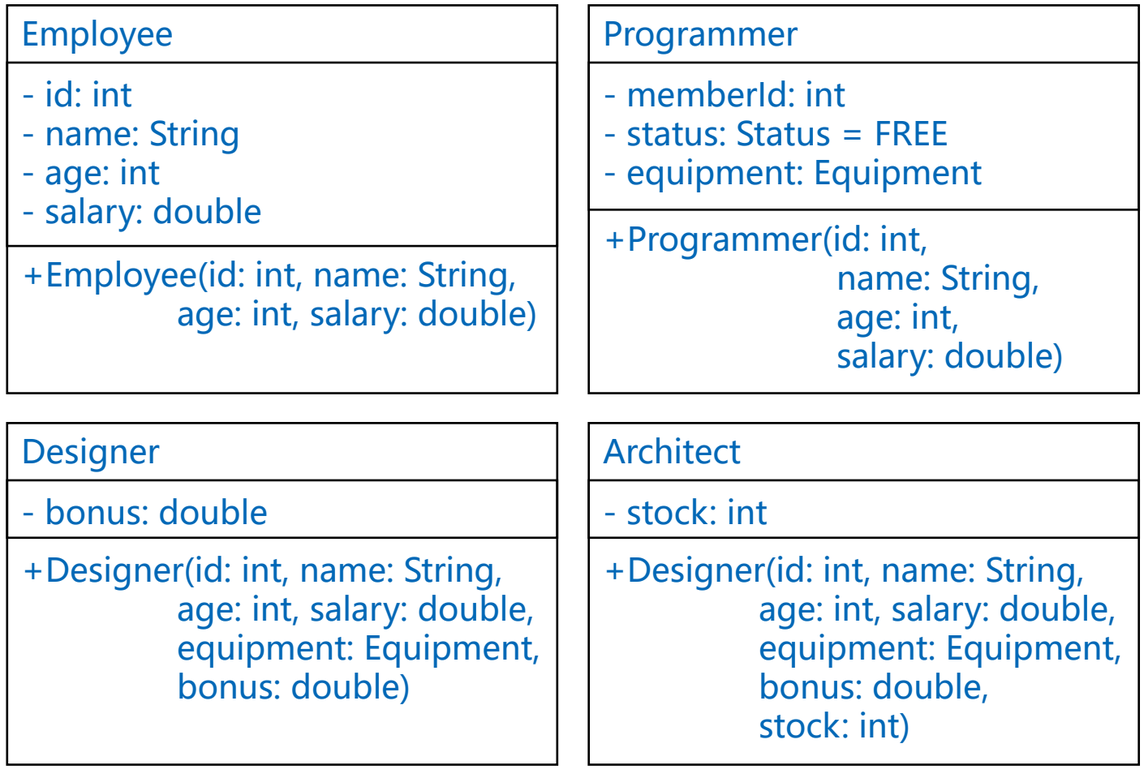



说明：
> model 表示机器的型号。
> display 表示显示器名称。
> type 表示机器的类型。

根据需要提供各属性的 get/set 方法以及重载构造器；
实现类实现接口的方法，返回各自属性的信息。



Employee 类及其子类的设计




说明：
> memberId 用来记录成员加入开发团队后在团队中的 ID。
> Status 是项目 service 包下自定义的类，声明三个对象属性，分别表示成员的状态。
   > FREE - 空闲。
   > BUSY - 已加入开发团队。
   > VACATION - 正在休假。
> equipment 表示该成员领用的设备。
> bonus 表示奖金。
> stock 表示公司奖励的股票数量。

可根据需要为类提供各属性的 get/set 方法以及重载构造器。

## 第二步：实现 service 包中的类

1、按照设计要求编写 NameListService 类。
2、在 NameListService 类中临时添加一个 main 方法中，作为单元测试方法。
3、在方法中创建 NameListService 对象，然后分别用模拟数据调用该对象的各个方法，以测试是否正确。
　　注：测试应细化到包含了所有非正常的情况，以确保方法完全正确。
4、重复 1-3 步，完成 TeamService 类的开发。

NameListService 类的设计

| NameListService |
| --- |
| - employees: Employee[] |
| +NameListService()<br>+getAllEmployees(): Employee[]<br>+getEmployee(int id) throws TeamException: Employee |

功能：负责将 Data 中的数据封装到 Employee[] 数组中，同时提供相关操作 Employee[] 的方法。
说明：
> employees 用来保存公司所有员工对象。
> NameListService() 构造器：
　　> 根据项目提供的 Data 类构造器相应大小的 employees 数组。
　　> 再根据 Data 类中的数据构建不同的对象，包括 Employee、Programmer、Designer、Archietct 对象，以及相关联的 Equipment 子类的对象。
　　> 将对象存于数组中。
　　> Data 类位于 service 包中。
> getAllEmployees() 方法，获取当前所有员工。
　　> 返回：包含所有员工对象的数组。
> getEmployss(id: int) 方法，获取指定 ID 的员工对象。
　　> 参数：指定员工的 ID。
　　> 返回：指定员工对象。
　　> 异常：找不到指定的员工。
在 service 子包下一共自定义异常类：TeamException。
另外，可根据需要自行添加其他方法或重载构造器。



TeamService 类的设计

| TeamService |
| --- |
| - counter: int = 1<br>- MAX_MEMBER: final int = 5<br>- team: Programmer[] = new Programmer[MAX_MEMBER];<br>- total: int = 0 |
| + getTeam(): Programmer[]<br>+ addMember(e: Employee) thores TeamException: void<br>+ removeMember(memberId: int) throws TeamException: void |

功能：关于开发团队成员的管理：添加、删除等。
说明：
> counter 为静态变量，用来为开发团队新增成员自动生成团队中唯一 ID，即 memberId：( 提示：应使用增 1 的方式 )。
> MAX_MEMBER：表示开发团队最大成员数。
> team 数组：用来保存当前团队中的各成员对象。
> total：记录团队成员的实际人数。
> getTeam() 方法：返回当前团队的所有对象。
　　返回：包含所有成员对象的数组，数组大小与成员人数一致。
> addMember(e: Employee) 方法，向团队中添加成员。
　　参数：待添加成员的对象。
　　异常：添加失败，TeamException 中包含了失败的原因。
> removeMember(memberId: int) 方法：从团队中删除成员。
　　参数：待删除成员的 memberId。
　　异常：找不到指定 memberId 的员工，删除失败。
另外，可根据需要自行添加其他方法或重载构造器。

需求说明：如果添加操作因某种原因失败，将显示以下信息：
1 成员已满，无法添加。
2 该成员不是开发人员，无法添加。
3 该员工已在开发团队中。
4 该员工已是某团队成员。
5 该员工正在休假，无法添加。
6 团队中最多只能有一名架构师。
7 团队中最多只能有两名设计师。
8 团队中最多只能有三名程序员。

## 第三步：实现 view 包中类

1、按照设计要求编写 TeamView 类，逐一实现各个方法，并编译。
2、执行 main 方法中，测试软件全部功能。

TeamView 类的设计

| TeamView |
| --- |
| - listSvc: NameListService = new NameListService()<br>- teamSvc: TeamService = new TeamService() |
| + enterMainMenu(): void<br>- listAllEmployees(): void<br>- getTeam(): void<br>- addMember(): void<br>- deleteMember(): void<br>+ main(args: String[]): void |

说明：

> listSvc 和 teamSvc 属性：供类中的方法使用。
> enterMainMenu() 方法：主界面显示及控制方法。
> 以下方法仅供 enterMainMenu() 方法调用：
listAllEmployees() 方法：以表格形式列出公司所有成员。
getTeam() 方法：显示团队成员列表操作。
addMember() 方法：实现添加成员操作。
deleteMember() 方法：实现删除成员操作。



## Data 模块

```
public class Data {
	public static final int EMPLOYEE = 10;
	public static final int PROGRAMMER = 11;
	public static final int DESIGNER = 12;
	public static final int ARCHITECT = 13;
	public static final int PC = 21;
	public static final int NOTEBOOK = 22;
	public static final int PRINTER = 23;

	/**
	 * Employee		:10, name, age, salary
	 * Programmer		:11, name, age, salary
	 * Designer		:12, name, age, salary, bonus
	 * Architect		:13, name, age, salary, bonus, stock
	 */

	public static final String[][] EMPLOYEES = {
			{"10" , "1" , " 马云 ","22" , "3000"},
			{"13" , "2" , " 马化腾 ","32" , "18000" , "15000" , "2000"},
			{"11" , "3" , " 李彦宏 ","23" , "7000"},
			{"11" , "4" , " 刘强东 ","24" , "7300"},
			{"12" , "5" , " 雷军 ","28" , "10000" , "5000"},
			{"11" , "6" , " 任志强 ","22" , "6800"},
			{"12" , "7" , " 柳传志 ","29" , "10800" , "5200"},
			{"13" , "8" , " 杨元庆 ","30" , "19800" , "15000" , "2500"},
			{"12" , "9" , " 史玉柱 ","26" , "9800" , "6600"},
			{"11" , "10" , " 丁磊 ","21" , "6600"},
			{"11" , "11" , " 张朝阳 ","25" , "7100"},
			{"12" , "12" , " 杨致远 ","27" , "9600" , "4800"},
	};

	/**
	 * 如下的 EQUIPMENTS 数组与上面的 EMPLOYEES 数组元素一一对
应
	 * Pc		: 21 ,  model,  display
	 * Notebook	: 22 ,  model,  price
	 * Printer	: 23 ,  name ,  type
	 */
```

```
        public static final String[][] EQUIPMENTS = {
                {},
                {"22" , " 联想 T4" , "6000"},
                {"21" , " 戴尔 " , "NEC 17 寸 "},
                {"21" , " 戴尔 " , " 三星 17 寸 "},
                {"23" , " 佳能 2900" , " 激光 "},
                {"21" , " 华硕 " , " 三星 17 寸 "},
                {"21" , " 华硕 " , " 三星 17 寸 "},
                {"23" , " 爱普生 20K" , " 针式 "},
                {"22" , " 惠普 m6" , "5800"},
                {"21" , " 戴尔 " , "NEC 17 寸 "},
                {"21" , " 华硕 " , " 三星 17 寸 "},
                {"22" , " 惠普 m6" , "5800"},
        };
}
```

## TSUtility 模块

```
import java.util.Scanner;
/**
 * @Description 项目中提供了 TSUtility.java 类，可用来方便的实现键盘访问
 */
public class TSUtility {
        private static Scanner scanner = new Scanner(System.in);
        /**
         * @Description 该方法读取键盘，如果用户输入 "1"-"4" 中任意字符，返回值为用户键入字符
         */
        public static char readMenuSelection() {
                char c;
                for (;;) {
                        String str = readKeyBoard(1, false);
                        c = str.charAt(0);
                        if (c != '1' && c != '2' && c != '3' && c != '4') {
                                System.out.print(" 选择错误，请重新输入 ");
                        } else
                                break;
                }
                return c;
        }
```



```
        /**
         * @Description 该方法提示并等待，直到用户按回车键后返回
         */
        public static void readReturn() {
                System.out.println(" 按回车键继续 ...");
                readKeyBoard(100, true);
        }

        /**
         * @Description 该方法从键盘读取一个长度不超过 2 位的整数，并将其作为该方法的返回值
         */
        public static int readInt() {
                int n;
                for(;;) {
                        String str = readKeyBoard(2,false);
                        try {
                                n = Integer.parseInt(str);
                                break;
                        }catch(NumberFormatException e) {
                                System.out.print(" 数字输入错误，请重新输入： ");
                        }
                }
                return n;
        }

        /**
         * @Description 从键盘读取 'Y' 或 'N'，并将其作为方法的返回值
         */
        public static char readConfirmSelection() {
                char c;
                for (;;) {
                        String str = readKeyBoard(1, false).toUpperCase();
                        c = str.charAt(0);
                        if (c == 'Y' || c == 'N') {
                                break;
                        } else {
                                System.out.println(" 选择错误，请重新输入： ");
                        }
                }
```

```
            return c;
        }
        private static String readKeyBoard(int limit, boolean
blankReturn) {
            String line = "";
            while (scanner.hasNextLine()) {
                line = scanner.nextLine();
                if (line.length() == 0) {
                    if (blankReturn)
                        return line;
                    else
                        continue;
                }

                if (line.length() < 1 || line.length() > limit) {
                    System.out.print(" 输入长度（不大于 " + limit + "）
错误，请重新输入 ");
                    continue;
                }
                break;
            }
            return line;
        }
}
```

## Equipment 模块

```
package domain;

public interface Equipment {
    String getDescription();
}
```



## PC 模块

```
package domain;

public class PC implements Equipment{
    private String model;// 机器型号
    private String display;// 显示器名称

    public PC(String model, String display) {
        super();
        this.model = model;
        this.display = display;
    }

    public PC() {
        super();
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public String getDisplay() {
        return display;
    }

    public void setDisplay(String display) {
        this.display = display;
    }

    @Override
    public String getDescription() {
        return model + "(" + display + ")";
    }

}
```

## NoteBook 模块

```
package domain;

public class Notebook implements Equipment {
    private String model;
    private double price;// 价格

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public Notebook() {
        super();
    }

    public Notebook(String model, double price) {
        super();
        this.model = model;
        this.price = price;
    }

    @Override
    public String getDescription() {
        return model + "(" + price + ")";
    }

}
```



## Printer 模块

```
package domain;

public class Printer implements Equipment{
    private String name;
    private String type;// 机器类型

    public Printer(String name, String type) {
        super();
        this.name = name;
        this.type = type;
    }

    public Printer() {
        super();
    }

    public String getName() {
        return name;
    }

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

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    @Override
    public String getDescription() {
        return name + "(" + type + ")";
    }

}
```

## Employee 模块

```
package domain;
public class Employee {
	private int id;
	private String name;
	private int age;
	private double salary;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

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

	public int getAge() {
		return age;
	}

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

	public double getSalary() {
		return salary;
	}

	public void setSalary(double salary) {
		this.salary = salary;
	}
```



```
	public Employee() {
		super();
	}

	public Employee(int id, String name, int age, double salary) {
		super();
		this.id = id;
		this.name = name;
		this.age = age;
		this.salary = salary;
	}

	public String getDetails() {
		return id + "\t" + name + "\t" + age + "\t" + salary;
	}

	@Override
	public String toString() {
		return getDetails();
	}
}
```

## Programmer 模块

```
package domain;
import service.Status;
public class Programmer extends Employee {
	private int memberId;// 开发团队中的 id
	private Status status = Status.FREE;
	private Equipment equipment;
	public Programmer() {
		super();
	}

	public Programmer(int id, String name, int age, double salary,
Equipment equipment) {
		super(id, name, age, salary);
		this.equipment = equipment;
	}

	public int getMemberId() {
		return memberId;
	}

	public void setMemberId(int memberId) {
		this.memberId = memberId;
	}

	public Status getStatus() {
		return status;
	}

	public void setStatus(Status status) {
		this.status = status;
	}

	public Equipment getEquipment() {
		return equipment;
	}
	public void setEquipment(Equipment equipment) {
		this.equipment = equipment;
	}
	@Override
	public String toString() {
		return getDetails() + "\t 程序员 \t" + status + "\t\t\t" +
equipment.getDescription();
	}
	public String getTeamBaseDetails() {
		return memberId + "/" + getId() + "\t" + getName() + "\t"
+ getAge() + "\t" + getSalary();
	}
	public String getDetailsForTeam() {
		return getTeamBaseDetails() + "\t 程序员 ";
	}
}
```

## Status 模块

```
package service;
// 表示员工状态
public class Status {
	private final String NAME;
	private Status(String name) {
		this.NAME = name;
	}

	public static final Status FREE = new Status("FREE");
	public static final Status BUYS = new Status("BUYS");
	public static final Status VOCATION = new Status("VOCATION");

	public String getNAME() {
		return NAME;
	}
	@Override
	public String toString() {
		return NAME;
	}
}
// 枚举类
//public enum Status {
//	FREE, BUSY, VACATION;
//}
```

## Designer 模块

```
package domain;

public class Designer extends Programmer {
	private double bonus;// 奖金

	public Designer() {
		super();
	}

	public Designer(int id, String name, int age, double salary,
Equipment equipment, double bonus) {
		super(id, name, age, salary, equipment);
		this.bonus = bonus;
	}

	public double getBonus() {
		return bonus;
	}

	public void setBonus(double bonus) {
		this.bonus = bonus;
	}

	@Override
	public String toString() {
		return getDetails() + "\t 设计师 \t" + getStatus() + "\t" +
bonus + "\t\t" + getEquipment().getDescription();
	}

	public String getDetailsForTeam() {
		return getTeamBaseDetails() + "\t 设计师 \t" + getBonus();
	}
}
```



## Architect 模块

```
package domain;

public class Architect extends Designer {
	private int stock;// 股票

	public Architect() {
		super();
	}

	public Architect(int id, String name, int age, double salary,
Equipment equipment, double bonus, int stock) {
		super(id, name, age, salary, equipment, bonus);
		this.stock = stock;
	}

	public int getStock() {
		return stock;
	}

	public void setStock(int stock) {
		this.stock = stock;
	}

	@Override
	public String toString() {
		return getDetails() + "\t 架构师 \t" + getStatus() + "\t" +
getBonus() + "\t" + stock + "\t"+ getEquipment().getDescription();
	}

	public String getDetailsForTeam() {
		return getTeamBaseDetails() + "\t 架构师 \t" + getBonus() +
"\t" + getStock();
	}
}
```

## NameListService 模块

```
package service;

import domain.Architect;
import domain.Designer;
import domain.Employee;
import domain.Equipment;
import domain.Notebook;
import domain.PC;
import domain.Printer;
import domain.Programmer;

import static service.Data.*;
/**
 * @Description 负责将 Data 中的数据封装到 Employee[] 数组中，同时
提供相关操作 Employee[] 的方法
 * @author Administrator
 */
public class NameListService {
	private Employee[] employees;

	/**
	 * @Description 给 employee 级数组元素进行初始化
	 */
	public NameListService() {
		// 根据项目提供的 Data 类构建相应大小的 employees 数组
		// 在根据 Data 类中的数据构建不同的对象，包括 Employee、
Programmer、Designer、Architect 将对象存放于数组中
		employees = new Employee[EMPLOYEES.length];
		for (int i = 0; i < employees.length; i++) {
			// 获取员工的类型
			int type = Integer.parseInt(EMPLOYEES[i][0]);
			// 获取 Employee 的 4 个基本信息
			int id = Integer.parseInt(EMPLOYEES[i][1]);
			String name = EMPLOYEES[i][2];
			int age = Integer.parseInt(EMPLOYEES[i][3]);
			double salary = Double.parseDouble(EMPLOYEES[i]
[4]);
```



```
			Equipment equipment;
			double bonus;
			int stock;

			switch (type) {
			case EMPLOYEE:
				employees[i] = new Employee(id, name, age,
salary);
				break;

			case PROGRAMMER:
				equipment = createEquipment(i);
				employees[i] = new Programmer(id, name, age,
salary, equipment);
				break;

			case DESIGNER:
				equipment = createEquipment(i);
				bonus = Double.parseDouble(EMPLOYEES[i][5]);
				employees[i] = new Designer(id, name, age,
salary, equipment, bonus);
				break;

			case ARCHITECT:
				equipment = createEquipment(i);
				bonus = Double.parseDouble(EMPLOYEES[i][5]);
				stock = Integer.parseInt(EMPLOYEES[i][6]);
				employees[i] = new Architect(id, name, age,
salary, equipment, bonus, stock);
				break;
			}
		}
	}

	/**
	 * @Description 获取指定 index 上的员工设备
	 * @param index
	 * @return
	 */
	private Equipment createEquipment(int index) {
```

```
            int key = Integer.parseInt(EQUIPMENTS[index][0]);
            String ModelOrName = EQUIPMENTS[index][1];
            switch (key) {
            case PC:
                String display = EQUIPMENTS[index][2];
                return new PC(ModelOrName, display);

            case NOTEBOOK:
                double price = Double.
parseDouble(EQUIPMENTS[index][2]);
                return new Notebook(ModelOrName, price);

            case PRINTER:
                String type = EQUIPMENTS[index][2];
                return new Printer(ModelOrName, type);
            }
            return null;
        }
        /**
         * 获取当前所有的员工
         * @return
         */
        public Employee[] getAllEmployees() {
            return employees;
        }
        /**
         * 获取指定 ID 的员工对象
         * @param id
         * @return
         * @throws TeamException
         */
        public Employee getEmployees(int id) throws TeamException {
            for (int i = 0; i < employees.length; i++) {
                if (employees[i].getId() == id) {
                    return employees[i];
                }
            }
            throw new TeamException(" 找不到指定员工 ");
        }
    }
```



## TeamService 模块

```
package service;

import domain.Architect;
import domain.Designer;
import domain.Employee;
import domain.Programmer;
/**
 * 关于开发团队成员的管理：添加删除等
 * @author Administrator
 */
public class TeamService {
    private static int counter = 1;// 给 memberID 赋值
    private final int MAX_MEMBER = 5;// 限制开发团队的人数
    private Programmer[] team = new Programmer[MAX_
MEMBER];// 保存开发团队成员
    private int total;// 记录开发团队中实际的人数
    public TeamService() {
        super();
    }

    public Programmer[] getTeam() {
        Programmer[] team = new Programmer[total];
        for (int i = 0; i < team.length; i++) {
            team[i] = this.team[i];
        }
        return team;
    }
    /*
     * 1 成员已满，无法添加。
     * 2 该成员不是开发人员，无法添加。
     * 3 该员工已在开发团队中。
     * 4 该员工已是某团队成员。
     * 5 该员工正在休假，无法添加。
     * 6 团队中最多只能有一名架构师。
     * 7 团队中最多只能有两名设计师。
     * 8 团队中最多只能有三名程序员。
     * @throws TeamException
     */
```

```
    public void addMember(Employee e) throws TeamException {
        if (total >= MAX_MEMBER) {
            throw new TeamException(" 成员已满，无法添加 ");
        }

        if (!(e instanceof Programmer)) {
            throw new TeamException(" 该成员不是开发人员，无法
添加 ");
        }

        if (isExist(e)) {
            throw new TeamException(" 该员工已在开发团队中 ");
        }

        Programmer p = (Programmer) e;
        // 一定不会出现 ClassCastException
        // if(p.getStatus().getNAME().equals("BUSY"))
        // 减少空指针异常的几率
        if ("BUSY".equals(p.getStatus().getNAME())) {
            throw new TeamException(" 该员工已是某团队成员 ");
        } else if ("VOCATION".equals(p.getStatus().getNAME())) {
            throw new TeamException(" 该员工正在休假，无法添加
");

        }

        // 获取 team 中已有架构师，设计师，程序员的人数
        int numOfArch = 0, numOfDes = 0, numOfPro = 0;
        for (int i = 0; i < total; i++) {
            if (team[i] instanceof Architect) {
                numOfArch++;
            } else if (team[i] instanceof Designer) {
                numOfDes++;

            } else if (team[i] instanceof Programmer) {
                numOfPro++;
            }

        }
```



```
        if (p instanceof Architect) {
            if (numOfArch >= 1) {
                throw new TeamException(" 团队中最多只能有一
名架构师 ");
            }
        } else if (p instanceof Designer) {
            if (numOfDes >= 2) {
                throw new TeamException(" 团队中最多只能有两
名设计师 ");
            }
        } else if (p instanceof Programmer) {
            if (numOfPro >= 3) {
                throw new TeamException(" 团队中最多只能有三
名程序员 ");
            }
        }
        // 将 p 或 e 添加到现有 team 中
        team[total++] = p;
        // p 的属性赋值
        p.setStatus(Status.BUYS);
        p.setMemberId(counter++);
    }

    /**
     * 判断指定的员工是否存在于现有开发团队中
     * @param e
     * @return
     */
    private boolean isExist(Employee e) {
        for (int i = 0; i < total; i++) {
            // return team[i].getID() == e.getID();
            if (team[i].getId() == e.getId()){
                return true;
            }
        }
        return false;
    }

    /**
     * 从团队中删除成员
```

```
     * @param memberID
     * @throws TeamException
     */
    public void remonerMember(int memberID) throws
TeamException {
        int i = 0 ;
        for (; i < total; i++) {
            if (team[i].getMemberId() == memberID) {
                team[i].setStatus(Status.FREE);
                break;
            }
        }
        if(i == total) {
            throw new TeamException(" 找不到指定 memberID 员
工，删除失败 ");
        }
        // 后一个元素覆盖前一个元素，实现删除操作
        for (int j = i + 1; j < total; j++) {
            team[j - 1] = team[j];
        }
        team[--total] = null;
    }
}
```

## TeamException 模块

```
package service;
/**
 * 自定义异常类
 * @author Administrator
 */
public class TeamException extends Exception{
    static final long serialVersionUID = -3387514229948L;

    public TeamException() {
        super();
    }
    public TeamException(String msg) {
        super(msg);
    }
}
```



## NameListServiceTest 模块

```
package team.junit;

import org.junit.Test;

import domain.Employee;
import service.NameListService;
import service.TeamException;

/**
 * 对 NameListService 类的测试
 * @author Administrator
 */
public class NameListServiceTest {
    @Test
    public void testGetAllEmployees() {
        NameListService service = new NameListService();
        Employee[] employees = service.getAllEmployees();
        for (int i = 0; i < employees.length; i++) {
            System.out.println(employees[i].toString());
        }
    }

    @Test
    public void testGetEmployee() {
        NameListService service = new NameListService();
        int id = 1;
        id = 10;
        try {
            Employee employee = service.getEmployees(id);
            System.out.println(employee);
        } catch (TeamException e) {
            System.out.println(e.getMessage());
        }

    }
}
```

## TeamView 模块

```
package view;

import domain.Employee;
import domain.Programmer;
import service.NameListService;
import service.TeamException;
import service.TeamService;

public class TeamView {
	private NameListService listSvc = new NameListService();
	private TeamService teamSvc = new TeamService();

	public void enterMainMenu() {

		boolean loopFlag = true;
		char menu = 0;

		while (loopFlag) {

			if (menu != '1') {
				listAllEmployees();
			}

			System.out.print("1- 团队列表  2- 添加团队成员  3- 删除
团队成员  4- 退出  请选择（1-4）：  ");

			menu = TSUtility.readMenuSelection();
			switch (menu) {

			case '1':
				getTeam();
				break;
			case '2':
				addMember();
				break;
			case '3':
				deleteMember();
				break;
```



```
			case '4':
				System.out.println(" 请确认是否退出 ('Y' 或 'N')：  ");
				char isExit = TSUtility.readConfirmSelection();
				if (isExit == 'Y') {
					loopFlag = false;
				}
				break;
			}
		}
	}

	/**
	 * 显示所有的员工信息
	 */
	private void listAllEmployees() {
		System.out.println("---------------------------------- 开发团
队调度软件 ----------------------------------\n");

		Employee[] employees = listSvc.getAllEmployees();
		if (employees == null || employees.length == 0) {
			System.out.println(" 公司中没有任何员工信息 ");
		} else {
			System.out.println("ID\t 姓名 \t 年龄 \t 工资 \t 职位 \t 状
态 \t 奖金 \t 股票 \t 领用设备 ");
			for (int i = 0; i < employees.length; i++) {
				System.out.println(employees[i]);
			}
		}

		System.out.print("-------------------------------------------
-----------------------------------\n");
	}

	/**
	 * 显示团员列表
	 */
	private void getTeam() {
		System.out.println("---------------------团队成员列表-------
---------------");
		Programmer[] team = teamSvc.getTeam();
```

```
            if (team == null || team.length == 0) {
                System.out.println(" 没有团队开发成员 ");
            } else {
                System.out.println("TID/ID\t 姓名 \t 年龄 \t 工资 \t 职位 \t
状态 \t 奖金 \t 股票 \n");
                for (int i = 0; i < team.length; i++) {
                    System.out.println(team[i].getDetailsForTeam());
                }
            }
            System.out.println("-----------------------------------------
-------------");
        }

        /**
         * 添加成员
         */
        private void addMember() {
            System.out.println("---------------------- 添加成员 -----------
-----------");
            System.out.println(" 请输入添加员工的 ID： ");
            int id = TSUtility.readInt();
            try {
                Employee emp = listSvc.getEmployees(id);
                teamSvc.addMember(emp);
                System.out.println(" 添加成功 ");
            } catch (TeamException e) {
                System.out.println(" 添加失败，原因： " +
e.getMessage());
            }

            // 按回车键继续
            TSUtility.readReturn();
        }

        /**
         * 删除成员
         */
        private void deleteMember() {
            System.out.println("---------------------- 删除成员 -----------
```



```
-----------");
            System.out.println(" 请输入要删除员工的 TID:");
            int memberId = TSUtility.readInt();

            System.out.println(" 确认是否删除 (Y/N):");
            char isDelete = TSUtility.readConfirmSelection();
            if(isDelete == 'N') {
                return;
            }
            try {
                teamSvc.remonerMember(memberId);
                System.out.println(" 删除成功 ");
            }catch(TeamException e){
                System.out.println(" 删除失败，原因 " + e.getMessage());
            }
            // 按回车键继续
            TSUtility.readReturn();
        }

    public static void main(String[] args) {
        TeamView view = new TeamView();
        view.enterMainMenu();
    }
}
```