# Java基础语法

今日内容介绍

* Java开发环境搭建
* HelloWorld案例
* 注释、关键字、标识符
* 数据（数据类型、常量）

# Java开发环境搭建

## Java概述

众所周知Java是一门编程语言，编程语言就是用来编写软件的。那么使用Java到底能用来编写什么软件呢？你所熟知的很多软件都可以用Java来编写，例如：QQ、迅雷、淘宝、京东等等，那么Java比较擅长什么呢？也就是说我们将来从事Java开发工程师工作到底要编写什么类型的软件呢？

答案是：

1. 互联网：电商、P2P等等；
2. 企业级应用：ERP、CRM、BOS、OA等等。

互联网类软件大家都比较熟悉，因为大家生活都可以接触得到。而企业级应用都是为大型企业量身定做的软件，用于工作中，所以大家可能不太了解。现在大家只需对Java开发工程师工作有个大致的了解即可。

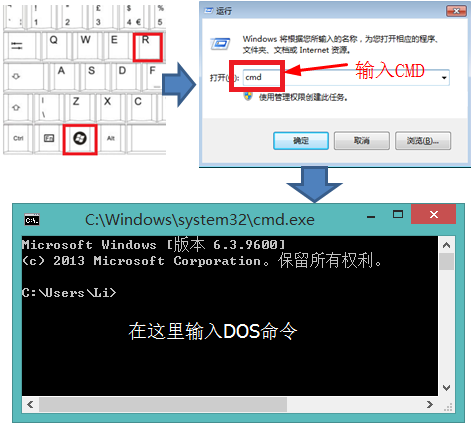
在基础班我们会学习Java的基础知识，即JavaSE（标准版）部分。学习完JavaSE并不能开发大型项目。

在就业班我们会学习Java的高级知识以及整个Java生态圈，即JavaEE（企业版）部分。学习完JavaEE部分就可以开发各种大型项目了。

## 常用DOS命令

编写源文件后就需要开始编译操作了，但在Windows操作系统中编译、运行都需要了解常用DOS命令，所以下面我们来学习一下常用DOS命令（DOS命令是Windows操作系统包含的知识）。

DOS命令是在DOS控制台中使用的命令，弹出DOS控制台的方法是：win+R，然后输出cmd，是不是看起来有点熟悉，没错，就是校验JDK安装是否成功是弹出的小黑屏。



DOS命令有很多，我们不需要学习所有DOS命令，只需要学习我们Java中要用到的常用DOS命令，内容如下：

* 清屏：cls；
* 当DOS控制台内容过多时，可以使用清屏命令，等同与“擦黑板”；
* 例如：C:\ >cls，然后按回车键。
* 切换盘符：:；
* 默认情况下我们是在C盘中，如果想去其他盘符可以使用该命令；
* 例如：C:\ >E:，然后按回车键就到了E盘下。
* 进入指定目录：cd；
* 当我们想进入某个目录时可以使用该命令；
* 例如：D:\>cd develop，这样就进入了D:\develop目录，当然你需要保证这个目录是存在的；
* 例如：D:\>cd develop\Java，这样就进入了D:\develop\Java目录。
* 返回上一级目录：cd..；0
* 当你需要返回到上一级目录时可以使用该命令；
* 例如：D:\develop\Java>cd..，然后按回车，就会返回到D:\develop目录下。
* 返回盘符根目录：cd\；
* 当需要返回到当前盘符根目录下可以使用该命令；
* 例如：D:\develop\Java>cd\，然后按回车，这样就返回到D:/目录下了。
* 显示当前目录下的文件和子目录信息：dir；
* 当需要查看当前目录下所有文件和子目录信息时可以使用该命令；
* 运行应用程序：程序完整名称 参数
* 当需要运行某个应用程序时可以使用该命令；
* 例如要运行记事本程序时：D:\>C:\windows\notepad.exe，这样就打开了记事本程序，这与你到C:\windows目录下双击该程序图标的效果相同，因为该程序支持双击运行；
* 还可以省略“.exe”后缀，例如：D:\>C:\windows\notepad，效果同上；

## Java开发环境概述

在开始学习Java编程之前，我们首先需要安装Java开发环境（JDK：Java开发工具包 Java Development Kit）。

你可能会问：“我们为什么要安装JDK呢”，其实任何事物都需要有他的最基本的生存环境，就像人类也需要最基本的生存环境（天、地、水、空气……）。开发Java程序也不能例外，它也需要最基本的“生存环境”，就是Java开发环境（JDK）。

至于JDK中都包含了什么东西，现在还不是讲解的时候。我们现在的任务就是马上把它下载、安装，并校验安装是否成功。

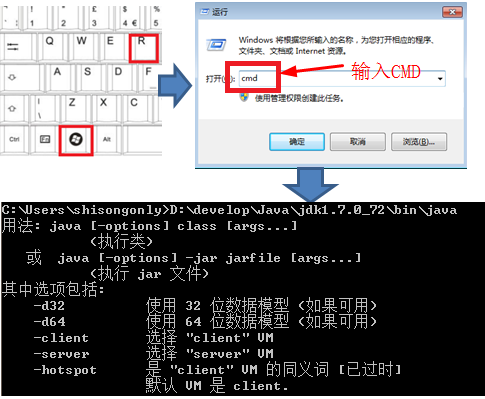
我们课程中使用的JDK版本是JDK7，当前最新版本是JDK8。

## JDK下载&安装&校验

JDK的下载和安装请各位严格按照《JDK下载安装文档0.doc》中的要求操作，如果没有按照要求操作可能会引起意想不到的问题。这里就不再赘述！

下面我们来聊聊如何校验JDK安装是否成功的方法。你无需了解为什么要这么做，只需要严格按照下面的步骤操作即可：

1. 在键盘上按：WIN+R键，这时会弹出运行窗口；
2. 在运行窗口中输入：CMD，按“确定”按钮，这时会弹出DOS窗口；
3. 在DOS窗口中输入：*JDK安装路径*\bin\java.exe。



如果结果与图片中相同，恭喜你，安装成功；否则安装失败！！！下面给出最常见的安装失败样例：

1. 系统找不到指定的路径：可能是因为JDK安装路径输入错误；
2. 不是内部或外部命令，也不是可运行的程序或批处理文件：安装失败。

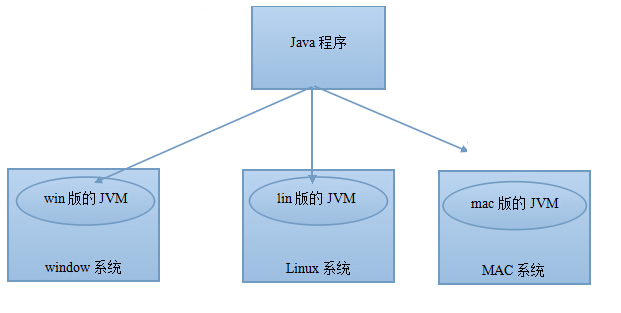
## JDK与JRE

在安装的过程中各位可能注意到了JRE这个名词，下面我们来聊聊JDK和JRE的关系和区别。

* JDK：它是Java开发运行环境，在程序员的电脑上当然要安装JDK；
* JRE：Java Runtime Environment它是Java运行环境，如果你不需要开发只需要运行Java程序，那么你可以安装JRE。例如程序员开发出的程序最终卖给了用户，用户不用开发，只需要运行程序，所以用户在电脑上安装JRE即可。
* JDK包含了JRE。

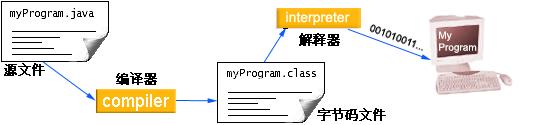
## 跨平台特性

* 平台指的是操作系统 （Windows，Linux，Mac）。
* Java程序可以在任意操作系统上运行，一次编写到处运行
* 实现跨平台需要依赖Java的虚拟机 JVM （Java Virtual Machine）



# Hello World案例

## Java程序介绍



在开始案例之前，我们需要了解一下Java应用程序的编写流程。

通过上图我们可以了解到编写的程序大致如下；

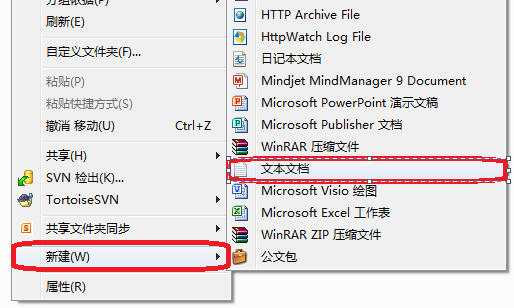
1. 源文件：编写Java源文件（我们也称之为源代码文件），它的扩展名为.java；
2. 编译：然后通过编译器把源文件编译成字节码文件，字节码文件扩展名为.class；
3. 运行：最后使用解释器来运行字节码文件。

编译和运行操作需要使用DOS命令，所以在编写源代码文件之后，我们要学习常用DOS命令，再去学习编译和运行。

## 源代码编写

### 创建HelloWorld.java文件

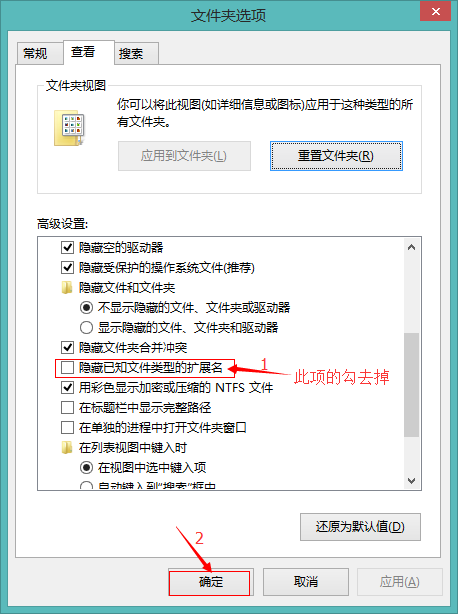
　　Java源代码编写只需要使用最普通的文本编辑器编写即可，例如Windows系统自带的notepad即可。如果你不知道什么是notepad，那么你可以在空白处点击鼠标右键🡪新建🡪文本文档来创建文本文件。



然后修改文件名称为HelloWorld.java，注意文件名称的大小写。以及是否已经修改了文件的扩展名。



如果看不到文件后缀名，可通过窗口上方“查看”菜单中的“文件夹选择”找到“查看”选项卡，在“查看选择卡”的高级设置找到“隐藏已知文件类型的扩展名”选项，去掉前面的勾，点击“确定”即可显示文件后缀名了。如下图



### 编写源程序代码

下面是完整的内容：

HelloWorld.java

**public** **class** HelloWorld {

**public** **static** **void** main(String[] args) {

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

}

}

注意，大家在编写源代码时一定要注意大小写。总而言之，一模一样就对了。

下面我们对源代码进行解析，HelloWorld案例的源代码一共分为三个部分，分别是：**类**、**主方法**，以及**输出语句**。

　　类是最外层的部分，其中public class是必须要写的命令，其中HelloWorld是类名称，要求必须与源文件名称相同。因为我们的源文件名称为HelloWorld.java，所以类名称必须是HelloWorld。类有一对大括号，类的内容存放在这对大括号中。你可能会问什么是类，我的答案是先不要去管什么是类，现在你需要记下来下面的代码，即定义类的语法。

**public** **class** HelloWorld {

}

主方法是类的内容，所以主方法放到类的大括号中。现在我们不需要了解更多，只需要把主方法的内容背下来即可。主方法也有一对大括号，用来存放主方法的内容。

**public** **static** **void** main(String[] args) {

}

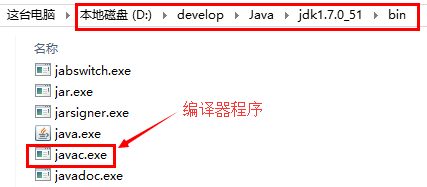
输出语句是主方法的内容，所以输出语句需要放到主方法的大括号中。输出语句也需要背下来。输出语句最后有一对圆括号，在圆括号中有一语话用双引号括起来：Hello World!。在程序运行时会输出它。

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

到这里我们的源代码就已经编写完毕了，还要再次提醒大家编写源代码时要注意大小写。文件名称和文件内容都要注意大小写。

## 编译源代码文件

学习完DOS命令后我们就可以来学习编译源代码文件了。编译操作需要使用编译器来完成，在安装JDK时已经安装了编译器，它属于JDK的一部分。可以到JDK目录下的bin目录下找到它，它的名称为：javac.exe。



注意，javac.exe并不支持双击运行，所以必须使用DOS命令来运行它。

下面我们来介绍一下使用编译器来编译Java源文件的格式：

格式：编译器程序完整名称 源文件完整名称

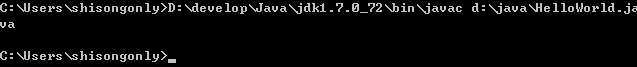
让我们来分析一下：

* 我的JDK安装在D:\develop\Java\jdk1.7.0\_72目录，那么编译器完整名称为：D:\develop\Java\jdk1.7.0\_72\bin\javac.exe；
* 我的源文件存放在：D:\java\HelloWorld.java；
* 编译命令为：D:\develop\Java\jdk1.7.0\_72\bin\javac.exe D:\java\HelloWorld.java。

编译之后会在源文件同目录下生成字节码文件，字节码文件的扩展名为.class。



请注意，在编译时控制台上不会有输出，如果有输出说明源代码有错误，所以大家一旦编译出错，就要去查看源文件中是否存在错误。

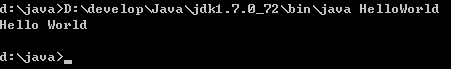


## 运行字节码文件

运行字节码文件需要使用java.exe命令，它与javac.exe在同一目录下。与javac.exe相同，java.exe也不支持双击运行，所以也必须要使用DOS命令来运行。运行字节码文件与编译有点区别：

* 进行字节码文件所在目录：即进入D:\java目录；
* D:\develop\Java\jdk1.7.0\_72\bin\java.exe HelloWord；

注意，在运行HelloWorld.class文件时不能给出“.class”，切记！！！



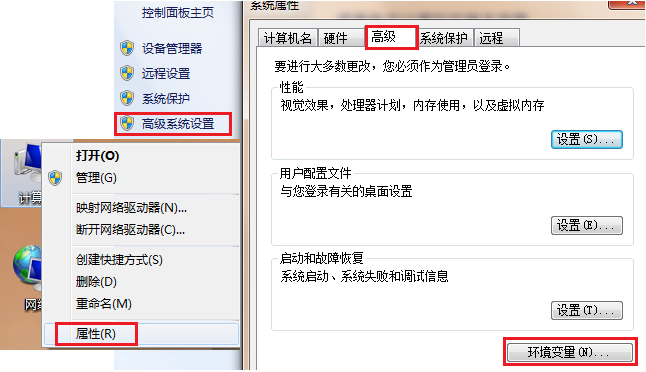
## 简化编译和运行操作

我们发现每次程序编译与运行都要输入D:\develop\Java\jdk1.7.0\_72\bin后，才能使用javac.exe与java.exe，这样的方式很麻烦，那么，能否不输入D:\develop\Java\jdk1.7.0\_72\bin呢？是可以的，通过配置Windows的PATH环境变量来完成。

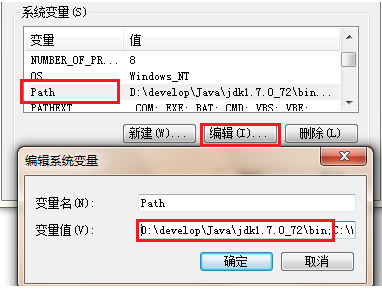
PATH变量中保存着多个路径，当我们在DOS控制台中使用的程序只给出程序名称，而没有给出完整路径时，那么Windows系统会到PATH变量保存的路径中去查找程序。我们如果把“D:\develop\Java\jdk1.7.0\_72\bin”路径保存到PATH变量中，那么在我们使用javac或java时，Windows系统就会自动到“D:\develop\Java\jdk1.7.0\_72\bin”路径下查找javac和java了。

* 配置步骤：

鼠标右键点击计算机 → 属性 → 高级系统设置 → 点击环境变量 → 找到系统变量中的PATH → 点击编辑按钮。



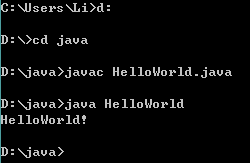
将JDK安装目录\bin路径（我的电脑上的D:\develop\Java\jdk1.7.0\_72\bin）配置到PATH变量中，用英文分号与其他变量隔开。



注意：配置PATH后文件的访问顺序：先访问当前路径，如果当前路径没有该文件，则再访问PATH配置的路径。

接下来我们测试一下：

* 把原有的HelloWorld.class删除；
* 进入HelloWorld.java所在目录：
* 切换盘符到E盘：C:/>D:；
* 进入java目录：D:/>cd java；
* 编译：D:/java/>javac HelloWorld.java；
* 运行：D:/java/>java HelloWorld；



# 注释、关键字与标识符

## 程序注释

通常我们需要在源代码中添加文字用来对进行代码解释说明，但这些文字并不是Java代码的语法，会导致编译出错。这时我们可以使用注释来完成这一事项！

在编译时，编译器会忽略注释的存在，就好像注释内容不存在一样。所以注释并不会导致编译出错，同时注释还方便编写者和其他人阅读源代码，加强对代码的理解。

Java中提供了三种注释方式，分别是：

* 单行注释 //注释内容
* 多行注释 /\*注释内容\*/
* 文档注释 /\*\*注释内容\*/

　　其中文档注释与多行注释作用基本相同，唯一的区别是文档注释可以使用javadoc.exe命令生成API文档。在这里我们不对其进行讲解，等大家对API有一定了解后在学习文档注释生成API文档的相关知识！

HelloWorld.java

/\*

\* 实现步骤：

\* 1.定义一个类 class

\* 2.编写程序执行的入口方法，main主方法

\* 3.通过输出语句System.out.println()将信息”HelloWorld!”打印在控制台上

\*/

// 定义一个类 class

**class** HelloWorld {

// 编写程序执行的入口方法，main主方法

**public** **static** **void** main(String[] args) {

// 通过Java提供的输出语句, 将信息”HelloWorld!”打印在控制台上

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

}

}

建议大家今后在编写代码时先写思路、分析、步骤，然后再写代码。其中思路、分析、步骤都使用注释放在源代码中。

## 关键字

关键字是被Java语言赋予特殊含义，具有专门用途的单词，比如之前接触的class，public，static，void均为Java已经预设好的。你可以先把Java关键字理解为“命令”！  
　　Java中关键字均为小写，下面是Java中所有关键字，你不需要去背，在今后的学习中重要的关键字会不断的出来。



你可以尝试看看HelloWorld案例中哪些是关键字，哪些不是！

## 标识符

学习完了关键字下面我们来学习一下什么是标识符。其实在Java程序中，除了关键字以外基本上都是标识符了。

其实标识符就是名称的意思，所有的名称都统称为标识符。Java中经常要定义类、方法、变量（后面会学习到），在定义他们时总要给它们起名字，这些名字就是标识符。

这里我们要学习的是如何起名字，你可能会说起名字还要学吗？答案是当然要学了！我们要学习的就是标识符的规范。

* 组成元素
  + 英文字符: a-zA-Z
  + 数字: 0-9
  + 符号: \_与$
* 标识符规则
  + 数字不能开头
  + 不可以使用关键字
  + 严格区分大小写，不限制长度起名时，尽量达到见名知意

# 数据

## 程序概述

软件是什么？其实软件就是用来处理数据的程序。可以把软件分为两个部分，一个部分是“数据”；另一个部分是“数据处理的逻辑”。

**程序 = 数据 + 逻辑**

什么是数据？用电商来举例，用户就是数据，商品也是数据。那么什么是数据处理逻辑呢？购买商品的流程就是数据处理逻辑了。

程序开发离不开数据，其实说白了程序就是用来处理数据的。在Java中我们要学习数据的类型及数据量。

通常数据是会发生变化的，而数据的处理逻辑是不发生变化的。例如商品的数量种类是会变化的，用户的数量也是会变化的，但购买的流程通常不会变化。

## 数据类型

在生活中数据是有类型的，例如人的姓名是字符串类型，人的年龄是正整数类型。在Java中数据也是有类型的（任0何数据必然会有类型），下面是Java中数据的类型：



基本数据类型是Java语言中内置的类型，分别是整数类型、小数类型、字符类型、布尔类型。这四类基本类型是最简单、最基础的类型。

引用数据类型是强大的数据类型，它是基于基本数据类型创建的。JavaSE中提供了一个超级类库，类库中包含了近万种引用数据类型。不过现在我们先要学习的是基本类型！

## 常量

Java中的数据量分为常量和变量，我们先来学习常量！

常量就是不变的数据量，例如100就是常量，任何数据量都有其类型，那么100这个常量是什么类型呢？回忆一下基本类型中有四类，分别是整数、小数（浮点数）、字符、布尔。那么100是哪一类型呢？没错，就是整数常量！！！

* 整数类型
  + 十进制表示方式：正常数字 如 13、25等
  + 二进制表示方式：以0b(0B)开头 如0b1011 、0B1001
  + 十六进制表示方式：以0x(0X)开头 数字以0-9及A-F组成 如0x23A2、0xa、0x10
  + 八进制表示方式：以0开头 如01、07、0721
* 小数类型
  + 如1.0、-3.15、3.168等
* 布尔类型
  + true、false
* 字符类型
  + 如'a'，'A', '0', '家'
  + 字符必须使用’’ 包裹，并且其中只能且仅能包含一个字符。
* 字符串类型
  + 字符串String类型是一种引用类型，我们先了解作为常量类型的使用方式
  + 如“我爱Java”，“0123”，“”，“null”
  + 字符串必须使用“”包裹，其中可以包含0~N个字符。

回顾HelloWorld案例，其中System.out.println(“Hello World!”)的圆括号中放的就是一个字符串。那么我们是否可以把圆括号中的字符串换成其他字符串呢？答案是可以的！不只是可以换成其他字符串，还可以把圆括号中的字符串换成其他类型的常量，例如：

我们通过编写代码来实际感受一下： Const.java

/\*

\* 常量的演示

\*/

**public** **class** Const {

**public** **static** **void** main(String[] args) {

//整数类型

//10进制

System.*out*.println(100);

System.*out*.println(12345);

System.*out*.println( 10 );

//2进制

System.*out*.println( 0b1010 );

//8进制

System.*out*.println( 012 );

//16进制

System.*out*.println(0xA);

//小数类型

System.*out*.println(3.15);

//字符类型

System.*out*.println('a');

System.*out*.println('A');

System.*out*.println('0');

System.*out*.println('家');

//布尔类型

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

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

//字符串类型

System.*out*.println("我爱Java");

System.*out*.println("0123");

}

}

# 总结

## Java开发环境搭建

* 下载和安装JDK
* 配置环境变量
  + 计算机->属性->高级系统设置->高级->环境变量->系统变量path->jdk安装目录bin

## 注释

* Java的三种注释写法
* // 单行
* /\*\*/ 多行
* /\*\*\*/ 文档注释

## 关键字

* 定义
  + - 被Java定义好的，赋予特殊含义单词
    - 全部采用小写字母
    - class public static void true false

## 标识符

* 自定义的内容，类名，方法名，变量名
* 包含字母大小写，数字，下划线和美元符，不能数字开头
* 不允许是关键字

## 数据类型和常量

* 基本数据类型
  + - 整数 0 1 2
    - 浮点 0.1 1.0 2.2
    - 布尔 true false
    - 字符 'a'
    - 字符串常量 "" 任意长度

# 第2天 Java基础语法

今日内容介绍

* 变量
* 运算符

# 变量

## 变量概述

前面我们已经学习了常量，接下来我们要学习变量。在Java中变量的应用比常量的应用要多很多。所以变量也是尤为重要的知识点！

什么是变量？变量是一个内存中的小盒子（小容器），容器是什么？生活中也有很多容器，例如水杯是容器，用来装载水；你家里的大衣柜是容器，用来装载衣裤；饭盒是容器，用来装载饭菜。那么变量是装载什么的呢？答案是数据！**结论：变量是内存中装载数据的小盒子，你只能用它来存数据和取数据。**

## 计算机存储单元

变量是内存中的小容器，用来存储数据。那么计算机内存是怎么存储数据的呢？无论是内存还是硬盘，计算机存储设备的**最小信息单元叫“位（bit）”**，我们又称之为“比特位”，通常用小写的字母b表示。而计算机最小的存储单元叫“字节（byte）”，通常用大写字母B表示，字节是由连续的8个位组成。

当程序需要使用存储空间时，操作系统最小会分派给程序1个字节，而不是1个位。你可能会说，如果程序只需要1个位的空间，系统分派不能只分派1个位吗？答案是不能！这就像你只需要1支烟，你到商店去买烟，商店分派的最小单元是1盒（20支），他不可能卖给你1支烟。

你可能会想，1个字节（8位）可以存储很大的数值了，1位最大是9那么8位最大值为99999999。你错了，因为计算机是采用二进行存储的，而不是我们生活中常用的十进制。所以1个字节存储的最大数据是11111111的二进制数。

除了字节外还有一些常用的存储单位，大家可能比较熟悉，我们一起来看看：

1B（字节） = 8bit

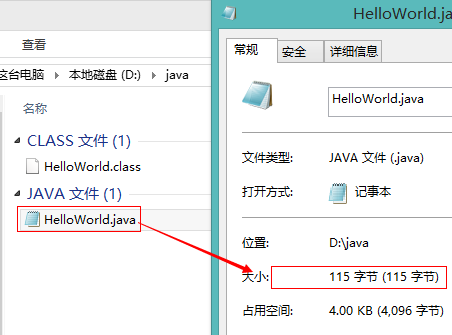
1KB = 1024B

1MB = 1024KB

1GB = 1024MB

1TB = 1024GB

1PB = 1024TB



## 基本类型之4类8种

　　大衣柜不能用来装载水，水杯也不能用来装载衣裤。这说明不同的容器装载不同的物品。变量也是如此，在创建变量时需要指定变量的数据类型，例如整型变量、浮点型变量等等。**结论：变量必须要有明确的类型，什么类型的变量装载什么类型的数据。**

水杯是用来装水的，那么水杯能装多少水呢？一吨？我们知道水杯在创建时不只确定了要装载的是水（数据类型），而且还确定了能装多少水（数据类型的具体种类）。变量也是如此，需要指定变量能装载什么类型的数据，同时也要指定变量能装载多大的数据。

Java中基本类型一共4类，把这4类展开后共8种基本类型。我们今后编写程序时使用的是这8种基本类型而不是4类，这8种基本类型指定了范围。

|  |  |  |  |
| --- | --- | --- | --- |
| **四类** | **八种** | **字节数** | **数据表示范围** |
| **整型** | **byte** | 1 | -128～127 |
| **short** | 2 | -32768～32767 |
| **int** | 4 | -2147483648～2147483648 |
| **long** | 8 | -263～263-1 |
| **浮点型** | **float** | 4 | -3.403E38～3.403E38 |
| **double** | 8 | -1.798E308～1.798E308 |
| **字符型** | **char** | 2 | 表示一个字符，如('a'，'A'，'0'，'家') |
| **布尔型** | **boolean** | 1 | 只有两个值true与false |

## 常量与类型

前面我们说过100是整数常量，但它是byte、short、int、long中的哪一种呢？下面我们来聊聊这一常量类型的问题。

整数常量可以根据所在范围来确定类型，例如100在-128~127之间，所以他是byte类型；500在-32768~32767之间，所以它是short类型；100000在-2147483648~2147483648之间，所以它是int类型。java中默认的整数类型是int类型

你可能会认为12345678901在-263~263-1之间，所以它是long类型。注意了，这是错误的！！！在Java中整数常量如果不在-2147483648~2147483648之间就必须添加“L”后缀（小写的也可以，但建议使用大写），在-2147483648~2147483648之间的也可以添加“L”后缀。也就是说12345678901不在-2147483648~2147483648之间，所以它在Java中是错误的常量，你必须这样写：12345678901L，这才是正确的常量。所以添加了“L”后缀的整数常量都是long类型的，例如：100L、12345678901L都是long类型的常量。

浮点类型的常量也可使用后缀，在Java中所有没有后缀以及使用“D”后缀（小写也可以，但建议使用大写）的小数都是double类型；float类型常量必须添加“F”后缀（小写也可以，但建议使用大写）java中默认的浮点类型是double类型

* 3.14没有后缀，所以它是double类型；
* 5.28D为double类型；
* 1.26F为float类型。

## 定义变量（创建变量）

定义变量的语法格式：

数据类型 变量名 = 数据值；

int a = 100;

其中int是数据类型，指定了变量只能存储整数，而且指定了存储范围为-2147483648～2147483648。

其中a表示变量名，变量名是标识符，这说明只要是合法的标识符都可以用来做变量名。在程序中可以通过变量名来操作变量（内存中的小盒子）。

其中“=100”是给变量赋值，即向a变量中写入100（变量是个小盒子，现在小盒子中保存的是100）。注意，给变量赋的值一定要与类型符合，也就是说int类型只能存储整数，而且必须是在-2147483648～2147483648范围内的整数。100满足了这两个条件，所以是正确的。

练习：

Variabe.java

/\*

变量定义格式：

数据类型 变量名 = 变量值;

\*/

**public** **class** Variable {

**public** **static** **void** main(String[] args) {

**int** a = 10;

**double** b = 3.14;

**char** c = 'z';

String s = "i love java";

a = 20;

System.*out*.println(a);

}

}

## 变量使用的注意事项

我们使用变量的时候需要注意，要满足变量的使用规则才可以使用的，我们来看看都有哪些注意事项。

* 变量使用的注意事项
  + 变量定义后可以不赋值，使用时再赋值。不赋值不能使用。

public static void main(String[] args) {

int x;

x = 20; //为x赋值20

System.out.println(x);//读取x变量中的值，再打印

}

* + 变量使用时有作用域的限制。

public static void main(String[] args) {

int x = 20;

{

int y = 20;

}

System.out.println(x);//读取x变量中的值，再打印

System.out.println(y);//读取y变量中的值失败，失败原因，找不到y变量，因为超出了y变量作用范围，所以不能使用y变量

}

* + 变量不可以重复定义。

public static void main(String[] args){

int x = 10;

double x = 5.5;//编译失败，变量重复定义

}

## 数据类型转换

　　不同类型的变量是否可以在一起运算呢？答案是可以的，但要先进行类型转换再运算。下面我们来学习一下类型转换。

其实，我们所学习的数据，它的表示方式是可以灵活多变的，比如把小数转换成整数的操作

转换的过程中，数据遵循一个原则：

范围小的数据类型值（如byte），可以直接转换为范围大的数据类型值（如int）；

范围大的数据类型值（如int），不可以直接转换为范围小的数据类型值（如byte）

那么，大家还记得每种类型表示数据的范围吗？忘记了不要紧，我来告诉大家，将各种数据类型按照数据范围从小到大依次列出：

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

关于数据类型转换有两种方式，我们来学习一下：

* 自动类型转换

表示范围小的数据类型转换成范围大的数据类型，这种方式称为自动类型转换

自动类型转换格式：

范围大的数据类型 变量 = 范围小的数据类型值；

如：

double d = 1000;

或

int i = 100;

double d2 = i;

* 强制类型转换

表示范围大的数据类型转换成范围小的数据类型，这种方式称为强制类型转换

强制类型转换格式：

范围小的数据类型 变量 = (范围小的数据类型) 范围大的数据类型值;

如：

int i = (int)6.718; //i的值为6

或

double d = 3.14;

int i2 = (int)d; //i2的值为3

# 运算符

## 算术运算符

运算符是用来计算数据的符号。数据可以是常量，也可以是变量。被运算符操作的数我们称为操作数。

算术运算符最常见的操作就是将操作数参与数学计算，具体使用看下图：

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **运算规则** | **范例** | **结果** |
| + | 正号 | +3 | 3 |
| + | 加 | 2+3 | 5 |
| + | 连接字符串 | “中”+“国” | “中国” |
| - | 负号 | int a=3;-a | -3 |
| - | 减 | 3-1 | 2 |
| \* | 乘 | 2\*3 | 6 |
| / | 除 | 5/2 | 2 |
| % | 取模 | 5/2 | 1 |
| ++ | 自增 | int a=1;a++/++a | 2 |
| -- | 自减 | int b=3;a--/--a | 2 |

我们在使用算术运算符时，记得要注意下列事项：

* 加法运算符在连接字符串时要注意，只有直接与字符串相加才会转成字符串。
* 除法“/”当两边为整数时，取整数部分，舍余数。当其中一边为浮点型时，按正常规则相除。
* “%”为整除取余符号，小数取余没有意义。结果符号与被取余符号相同。
* 整数做被除数，0不能做除数，否则报错。

代码演示

/\*

\* 算术运算符

\*/

**public** **class** OperatorDemo1 {

**public** **static** **void** main(String[] args) {

/\*

\* 常量使用算数运算符

\*/

System.*out*.println(10+20);

/\*

\* 变量使用算数运算符

\*/

**int** x = 10;

**int** y = 20;

//"+"作为加法运算使用

**int** z = x + y;

//"+"作为连接字符串使用

System.*out*.println("x="+x);

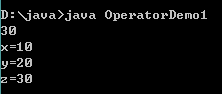
System.*out*.println("y="+y);

System.*out*.println("z="+z);

}

}

运行结果如下图所示。



1. 运行结果

## 算数运算符++、--的使用

算数运算符在前面我们已经学习过了，这里进行一些补充。

在一般情况下，算数运算符不会改变参与计算的变量值。而是在原有变量值不变的情况下，计算出新的值。但是有些操作符会改变参与计算的变量的值，比如++，--。

我们来看一段代码：

int a = 3;

int b = 3;

a++;

b--;

System.out.println(a);

System.out.println(b);

上面代码的输出结果a值为4，b值为2；

这说明a的原有值发生了改变，在原有值的基础上自增1；b的原有值也发生了改变，在原有值的基础上自减1；

* ++运算符，会在原有值的基础上自增1；
* --运算符，会在原有值的基础上自减1。

我们再看一段代码：

int a = 3;

int b = 3;

++a;

--b;

System.out.println(a);

System.out.println(b);

上面代码的输出结果a值为4，b值为2；

这说明++,--运算符单独使用，不参与运算操作时，运算符前后位置导致的运算结果是一致的。

接下来，介绍下++，--运算符参与运算操作时，发生了怎样的变化，我们来看一段代码：

int a = 3;

int b;

b = a++ + 10;

System.out.println(a);

System.out.println(b);

上面代码的输出结果a值为4，b值为13；

这里我要强调一下了，当++，--运算符参与运算操作时，后置++，--的作用：

* ++,--运算符后置时，先使用变量a原有值参与运算操作，运算操作完成后，变量a的值自增1或者自减1；

再介绍下++，--运算符前置时，参与运算操作的变化，我们来看一段代码：

int a = 3;

int b;

b = ++a + 10;

System.out.println(a);

System.out.println(b);

上面代码的输出结果a值为4，b值为14；

这里我强调一下，当++，--运算符参与运算操作时，前置++，--的作用：

* ++，--运算符前置时，先将变量a的值自增1或者自减1，然后使用更新后的新值参与运算操作。

## 赋值运算符

我们来学习一下赋值运算符，赋值运算符就是为变量赋值的符号，赋值运算符的使用看下图：

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **运算规则** | **范例** | **结果** |
| = | 赋值 | int a=2 | 2 |
| += | 加后赋值 | int a=2，a+=2 | 4 |
| -= | 减后赋值 | int a=2，a-=2 | 0 |
| \*= | 乘后赋值 | int a=2，a\*=2 | 4 |
| /= | 整除后赋值 | int a=2，a/=2 | 1 |
| %= | 取模后赋值 | int a=2，a%=2 | 0 |

注意：诸如+=这样形式的赋值运算符，会将结果自动强转成等号左边的数据类型。

写一个代码，我们看一下赋值运算符的使用

/\*

\* 赋值运算符

\* +=, -=, \*=, /=, %= ：

\* 上面的运算符作用：将等号左右两边计算，会将结果自动强转成等号左边的数据类型,再赋值给等号左边的

\* 注意：赋值运算符左边必须是变量

\*/

**public** **class** OperatorDemo2 {

**public** **static** **void** main(String[] args) {

**byte** x = 10;

x += 20;// 相当于 x = (byte)(x+20);

System.*out*.println(x);

}

}

运行结果如下图所示。



1. 运行结果

## 比较运算符

比较运算符，又叫关系运算符，它是用来判断两个操作数的大小关系及是否相等关系的，结果是布尔值true或者false。

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **运算规则** | **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 |

这里需要注意一下：

* 赋值运算符的 = 符号与比较运算符的 == 符号是有区别的，如下：
  + 赋值运算符的 = 符号，是用来将 = 符号右边的值，赋值给 = 符号左边的变量；
  + 比较运算符的 == 符号，是用来判断 == 符号 左右变量的值是否相等的。

我们通过下面的一段代码，我们演示一下这个注意事项：

int a = 3;

int b = 4;

System.out.println( a=b );

System.out.println( a==b );

上面代码输出的结果第一个值为4，第二个值为false。

## 逻辑运算符

逻辑运算符，它是用于布尔值进行运算的，运算的最终结果为布尔值true或false。

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **运算规则** | **范例** | **结果** |
| & | 与 | false&true | False |
| | | 或 | false|true | True |
| ^ | 异或 | true^flase | True |
| ! | 非 | !true | Flase |
| && | 短路与 | false&&true | False |
| || | 短路或 | false||true | True |

看完图后，我们来看一下逻辑运算符的常规使用方式：

* 逻辑运算符通常连接两个其他表达式计算后的布尔值结果
* 当使用短路与或者短路或时，只要能判断出结果则后边的部分就不再判断。

我们通过代码演示一下：

boolean b = 100>10;

boolean b2 = false;

System.out.println(b&&b2); // 打印结果为 false

System.out.println(b||b2); //打印结果为 true

System.out.println(!b2); //打印结果为 true

System.out.println(b && 100>10); //打印结果为 true，本行结果的计算方式稍后讲解运算符优先级时解答

好了，我们来总结一下运算符的结果规律吧：

* 短路与&&:参与运算的两边数据，有false，则运算结果为false；
* 短路或||:参与运算的两边数据，有true，则运算结果为true；
* 逻辑非! : 参与运算的数据，原先是true则变成false，原先是false则变成true。

## 三元运算符

接下来我们要学习的三元运算符与之前的运算符不同。之前学习的均为一元或者二元运算符。元即参与运算的数据。

* 格式：

(条件表达式)？表达式1：表达式2；

* 表达式：通俗的说，即通过使用运算符将操作数联系起来的式子，例如：
  + 3+2，使用算数运算符将操作数联系起来，这种情况，我们称为算数表达式。
  + 3>2，使用比较运算符（也称为条件运算符）将操作数联系起来，这种情况，我们称为条件表达式。
  + 其他表达式，不再一一举例。
* 三元运算符运算规则：

先判断条件表达式的值，若为true，运算结果为表达式1；若为false，运算结果为表达式2。

通过代码演示，我们来学习下三元运算符的使用：

方式一：

System.out.println( 3>2 ? “正确” : “错误” );

// 三元运算符运算后的结果为true，运算结果为表达式1的值“正确”，然后将结果“正确”，在控制台输出打印

方式二：

int a = 3;

int b = 4;

String result = (a==b) ? “相等” : “不相等”;

//三元运算符运算后的结果为false，运算结果为表达式2的值“不相等”，然后将结果赋值给了变量result

方式三：

int n = (3>2 && 4>6) ? 100 : 200;

//三元运算符运算后的结果为false，运算结果为表达式2的值200,然后将结果200赋值给了变量n

## 运算符优先级

在学习运算符的过程中，我们发现，当多个运算符一起使用的时候，容易出现不清晰先后运算顺序的问题，那么，在这里，我们来学习下，运算符之间的运算优先级。

下图是每种运算符的优先级，按照运算先后顺序排序（优先级相同的情况下，按照从左到右的顺序依次运算）

|  |  |  |
| --- | --- | --- |
| **优先级** | **描述** | **运算符** |
| **1** | 括号 | ()、[] |
| **2** | 正负号 | +、- |
| **3** | 自增自减，非 | ++、--、! |
| **4** | 乘除，取余 | \*、/、% |
| **5** | 加减 | +、- |
| **6** | 移位运算 | <<、>>、>>> |
| **7** | 大小关系 | >、>=、<、<= |
| **8** | 相等关系 | ==、!= |
| **9** | 按位与 | & |
| **10** | 按位异或 | ^ |
| **11** | 按位或 | | |
| **12** | 逻辑与 | && |
| **13** | 逻辑或 | || |
| **14** | 条件运算 | ?: |
| **15** | 赋值运算 | =、+=、-=、\*=、/=、%= |
| **16** | 位赋值运算 | &=、|=、<<=、>>=、>>>= |

优先级顺序看完了，我们来通过代码，加强一下：

int n = 3>4 ? 100 : 200;

这行的代码运算执行顺序我们来写一下：

* 1.执行 3>4操作，得到布尔类型false的结果
* 2.通过结果false，将对应的表达式2的结果200，作为运算的最终结果
* 3.把200 赋值给变量n

接下来，我们看一个比较复杂的代码：

int a = 5;

int b = 3;

int c = 1;

int n2 = (a>b && b>c) ? (c++) : (++c);

这段代码运算执行顺序我们也写一下：

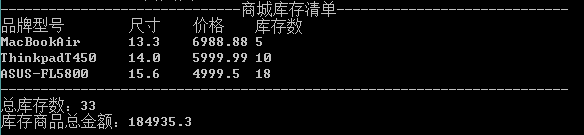
* 1.小括号优先级高，我们先运算第一组小括号中的代码
* 1.1. 比较运算符”>” 优先级大于 逻辑运算符”&&”
  + 先执行 a>b,得到结果true；
  + 再执行 b>c,得到结果true；
  + 最后执行 a>b的结果 && b>c的结果，即 true && true, 结果为true
* 2.三元运算符中的条件判断结果为true，返回表达式1的结果 c++
  + 先将变量c的原有值赋值给变量n2，即n2值为1；
  + 再将变量c的值自增1,更新为2。

运算符我们学到这里就结束了，稍后在“趣味乐园”中，我们可以运用运算符进行练习。

# 商场库存清单案例

## 案例介绍

现在我们来做一个复杂点的案例——商场库存清单案例，这个案例最终会在控制台输出如下结果：



## 案例需求分析

* 观察清单后，可将清单分解为三个部分（清单顶部、清单中部、清单底部）



* 1.清单顶部为固定的数据，直接打印即可
* 2.清单中部为商品，为变化的数据，需要记录商品信息后，打印

经过观察，我们确定一项商品应该有如下几个属性：

品牌型号: 即商品名称，String型

尺寸：物品大小，double型

价格：物品单价，double型

配置：这一项为每种商品的配置信息，String型

库存数：这一项为每种商品的库存个数，int型

* 3.清单底部包含了统计操作，需经过计算后，打印

我们发现两个单独的可变化量

总库存数：所有商品总个数，int型

库存商品总金额：所有商品金额，double型

## 实现代码步骤

一起分析完毕了，我们开始完成案例代码的编写：

* 创建Demo01库存清单.java文件，编写main主方法

**public** **class** Demo01库存清单 {

**public** **static** **void** main(String[] args) {

}

}

* 记录每种库存商品信息

//苹果笔记本电脑

String macBrand = "MacBookAir";

**double** macSize = 13.3;

**double** macPrice = 6988.88;

**int** macCount = 5;

//联想Thinkpad笔记本电脑

String thinkpadBrand = "ThinkpadT450";

**double** thinkpadSize = 14.0;

**double** thinkpadPrice = 5999.99;

**int** thinkpadCount = 10;

//华硕ASUS笔记本电脑

String ASUSBrand = "ASUS-FL5800";

**double** ASUSSize = 15.6;

**double** ASUSPrice = 4999.50;

**int** ASUSCount = 18;

* 统计所有库存商品数量与金额

//统计库存总个数、库存总金额

**int** totalCount = macCount + thinkpadCount + ASUSCount;

**double** totalMoney = (macCount \* macPrice) + (thinkpadCount \* thinkpadPrice) + (ASUSCount \* ASUSPrice);

* 打印库存清单顶部信息

//列表顶部

System.*out*.println("------------------------------商城库存清单-----------------------------");

System.*out*.println("品牌型号 尺寸 价格 库存数");

* 打印库存清单中部信息

//列表中部

System.*out*.println(macBrand+" "+macSize+" "+macPrice+" "+macCount);

System.*out*.println(thinkpadBrand+" "+thinkpadSize+" "+thinkpadPrice+" "+thinkpadCount);

System.*out*.println(ASUSBrand+" "+ASUSSize+" "+ASUSPrice+" "ASUSCount);

* 打印库存清单底部信息

//列表底部

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

System.*out*.println("总库存数："+totalCount);

System.*out*.println("库存商品总金额："+totalMoney);

# 总结

## 知识点总结

* 数据类型转换
  + 数据类型范围从小到大排序（byte < char < short < int < long < float < double），布尔类型Boolean不能参与类型转换；
  + 自动类型转换，范围小的数据类型向范围大的数据类型转换时使用；
  + 强制类型转换，范围大的数据类型向范围小的数据类型转换时使用。
* 算数运算符
  + 用来完成算数运算（如加减乘除计算等）
  + ++，--运算符的使用
    - ++，--前置（如++a），当参与运算时，变量a的值先自增1，然后用自增后的新值再参与运算；
    - ++，--后置（如a++），当参与运算时，变量a先使用原有值参与运算符，变量a的值再自增1。
* 赋值运算符
  + 用来完成数据的赋值（如 int a = 100;）
  + +=,-,\*=,/=这样的赋值运算符包含了一个强制转换的操作，会将左右两边运算后的结果，强制类型转换后赋值给左边

int n = 10;

byte by = 20;

by += n; // 运算完毕后，by的值为byte类型30， 相当于代码 by = (byte)(by + n);

* 比较运算符
  + 用来比较数据的大小（如 3>4），比较运算符也称为条件运算符。
  + 比较后的结果为布尔类型Boolean的值
  + “==”两个等号代表比较是否相等，“=”一个等号代表赋值。
* 逻辑运算符
  + 逻辑与& 和逻辑短路与&&：代表着并且的意思，左右两边都要条件成立，结果才为true；
  + 逻辑或| 和逻辑短路或||：代表着或者的意思，左边两边有一个条件成立，结果就为true；
  + 逻辑非!：代表着相反的意思，原先是false，结果就为true；原先是ture，结果就为false；
  + 逻辑异或^： 左右两边条件结果相同，结果就为false，左右两边条件结果不同，结果就为true；
* 三元运算符
  + 根据判断条件运算结果，执行不同的表达式值；条件为true，执行表达式1，否则，执行表达式2。

第3天 Java基础语法

今日内容介绍

* 引用数据数据类型（Scanner、Random）
* 流程控制语句（if、for、while、dowhile、break、continue）

# 引用数据类型

## Scanner类

我们要学的Scanner类是属于引用数据类型，我们先了解下引用数据类型。

* 引用数据类型的使用

与定义基本数据类型变量不同，引用数据类型的变量定义及赋值有一个相对固定的步骤或格式。

数据类型 变量名 = new 数据类型();

每种引用数据类型都有其功能，我们可以调用该类型实例的功能。

变量名.方法名();

* Scanner类

Scanner类是引用数据类型的一种，我们可以使用该类来完成用户键盘录入，获取到录入的数据。

Scanner使用步骤：

导包：import java.util.Scanner;

创建对象实例：Scanner sc = new Scanner(System.in);

调用方法：

int i = sc.nextInt(); 用来接收控制台录入的数字

String s = sc.next(); 用来接收控制台录入的字符串

了解完Scanner类，我们编写代码来使用下它：ScannerDemo01.java

**import** java.util.Scanner;

**public** **class** ScannerDemo01 {

**public** **static** **void** main(String[] args) {

//创建Scanner引用类型的变量

Scanner sc = **new** Scanner(System.*in*);

//获取数字

System.*out*.println("请输入一个数字");

**int** n = sc.nextInt();

System.*out*.println("n的值为" + n);

//获取字符串

System.*out*.println("请输入一个字符串");

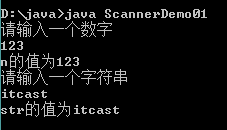
String str = sc.next();

System.*out*.println("str的值为" + str);

}

}

运行结果如下图所示。



1. 运行结果

## 随机数类Random

我们来学习下，用来产生随机数的类Random，它也属于引用数据类型。

这个Random类，它可以产生多种数据类型的随机数，在这里我们主要介绍生成整数与小数的方式。

* 方法简介

public int nextInt(int maxValue) 产生[0,maxValue)范围的随机整数，包含0，不包含maxValue；

public double nextDouble() 产生[0,1)范围的随机小数，包含0.0，不包含1.0。

引用数据类型的使用方式，在学习键盘录入Scanner时，我们已经学习过，在这里，再次回顾一下：

* Random使用方式:
  + import导包：所属包java.util.Random
  + 创建实例格式：Random 变量名 = new Random();

接下来，通过一段代码，一起学习下Random类的使用，RandomDemo.java

**import** java.util.Random;

**public** **class** RandomDemo {

**public** **static** **void** main(String[] args) {

// 创建Random类的实例

Random r = **new** Random();

// 得到0-100范围内的随机整数，将产生的随机整数赋值给i变量

**int** i = r.nextInt(100);

//得到0.0-1.0范围内的随机小数，将产生的随机小数赋值给d变量

**double** d = r.nextDouble();

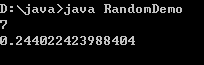
System.*out*.println(i);

System.*out*.println(d);

}

}

运行结果如下图所示：



1. 运行结果

# 流程控制语句

## 选择结构if

接下来要学习的if条件语句分为三种语法格式，每一种格式都有它自身的特点，下面我们分别进行介绍。

### if语句

if语句是指如果满足某种条件，就进行某种处理。例如，小明妈妈跟小明说“如果你考试得了100分，星期天就带你去游乐场玩”。这句话可以通过下面的一段伪代码来描述。

如果小明考试得了100分

妈妈星期天带小明去游乐场

在上面的伪代码中，“如果”相当于Java中的关键字if，“小明考试得了100分”是判断条件，需要用()括起来，“妈妈星期天带小明去游乐场”是执行语句，需要放在{}中。修改后的伪代码如下：

if (小明考试得了100分) {

妈妈星期天带小明去游乐场

}

上面的例子就描述了if语句的用法，在Java中，if语句的具体语法格式如下：

if (条件语句){

执行语句;

……

}

上述格式中，判断条件是一个布尔值，当判断条件为true时，{}中的执行语句才会执行。if语句的执行流程如下图所示。



1. if语句流程图

接下来通过一段代码，学习一下if语句的具体用法，IfDemo01.java

**public** **class** IfDemo01 {

**public** **static** **void** main(String[] args) {

**int** x = 5;

**if** (x < 10) {

x++;

}

System.*out*.println("x=" + x);

}

}

运行结果如下图所示。



1. 运行结果

在上述代码中，定义了一个变量x，其初始值为5。在if语句的判断条件中判断x的值是否小于10，很明显条件成立，{}中的语句会被执行，变量x的值将进行自增。从运行结果可以看出，x的值已由原来的5变成了6。

### if…else语句

if…else语句是指如果满足某种条件，就进行某种处理，否则就进行另一种处理。例如，要判断一个正整数的奇偶，如果该数字能被2整除则是一个偶数，否则该数字就是一个奇数。if…else语句具体语法格式如下：

if (判断条件){

执行语句1

……

}else{

执行语句2

……

}

上述格式中，判断条件是一个布尔值。当判断条件为true时，if后面{}中的执行语句1会执行。当判断条件为false时，else后面{}中的执行语句2会执行。if…else语句的执行流程如下图所示。



1. if…else语句流程图

接下来通过一段代码，来实现判断奇偶数的程序，IfDemo02.java

**public** **class** IfDemo02 {

**public** **static** **void** main(String[] args) {

**int** num = 19;

**if** (num % 2 == 0) {

// 判断条件成立，num被2整除

System.*out*.println("num是一个偶数");

} **else** {

System.*out*.println("num是一个奇数");

}

}

}

运行结果如下图所示。



1. 运行结果

上述代码中，变量num的值为19，模以2的结果为1，不等于0，判断条件不成立。因此会执行else后面{}中的语句，打印“num是一个奇数”。

### if…else if…else语句

if…else if…else语句用于对多个条件进行判断，进行多种不同的处理。例如，对一个学生的考试成绩进行等级的划分，如果分数大于80分等级为优，否则，如果分数大于70分等级为良，否则，如果分数大于60分等级为中，否则，等级为差。if…else if…else语句具体语法格式如下：

if (判断条件1) {

执行语句1

} else if (判断条件2) {

执行语句2

}

...

else if (判断条件n) {

执行语句n

} else {

执行语句n+1

}

上述格式中，判断条件是一个布尔值。当判断条件1为true时，if后面{}中的执行语句1会执行。当判断条件1为false时，会继续执行判断条件2，如果为true则执行语句2，以此类推，如果所有的判断条件都为false，则意味着所有条件均未满足，else后面{}中的执行语句n+1会执行。if…else if…else语句的执行流程如下图所示。



1. if…else if…else语句的流程图

接下来通过一段代码，来实现对学生考试成绩进行等级划分的程序，IfDemo03.java

**public** **class** IfDemo03 {

**public** **static** **void** main(String[] args) {

**int** grade = 75; // 定义学生成绩

**if** (grade > 80) {

// 满足条件 grade > 80

System.*out*.println("该成绩的等级为优");

} **else** **if** (grade > 70) {

// 不满足条件 grade > 80 ，但满足条件 grade > 70

System.*out*.println("该成绩的等级为良");

} **else** **if** (grade > 60) {

// 不满足条件 grade > 70 ，但满足条件 grade > 60

System.*out*.println("该成绩的等级为中");

} **else** {

// 不满足条件 grade > 60

System.*out*.println("该成绩的等级为差");

}

}

}

运行结果如下图所示。



1. 运行结果

上述代码中，定义了学生成绩grade为75。它不满足第一个判断条件grade>80，会执行第二个判断条件grade>70，条件成立，因此会打印“该成绩的等级为良”。

### 选择结构if语句与三元运算转换

三元运算符，它和if-else语句类似，语法如下：

判断条件 ? 表达式1 : 表达式2

三元运算符会得到一个结果，通常用于对某个变量进行赋值，当判断条件成立时，运算结果为表达式1的值，否则结果为表达式2的值。

例如求两个数x、y中的较大者，如果用if…else语句来实现，具体代码如下：

int x = 0;

int y = 1;

int max;

if (x > y) {

max = x;

} else {

max = y;

}

上面的代码运行之后，变量max的值为1。其中3-8行的代码可以使用下面的三元运算来替换。

int max = x > y ? x : y;

## 循环语句while

while循环语句和选择结构if语句有些相似，都是根据条件判断来决定是否执行大括号内的执行语句。区别在于，while语句会反复地进行条件判断，只要条件成立，{}内的执行语句就会执行，直到条件不成立，while循环结束。while循环语句的语法结构如下：

while(循环条件){

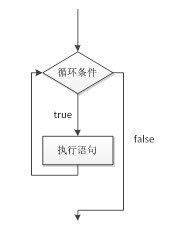
执行语句

………

}

在上面的语法结构中，{}中的执行语句被称作循环体，循环体是否执行取决于循环条件。当循环条件为true时，循环体就会执行。循环体执行完毕时会继续判断循环条件，如条件仍为true则会继续执行，直到循环条件为false时，整个循环过程才会结束。

while循环的执行流程如下图所示。



1. while循环的流程图

接下来通过一段代码，来实现打印1~4之间的自然数，WhileDemo.java

**public** **class** WhileDemo {

**public** **static** **void** main(String[] args) {

**int** x = 1; // 定义变量x，初始值为1

**while** (x <= 4) { // 循环条件

System.*out*.println("x = " + x); // 条件成立，打印x的值

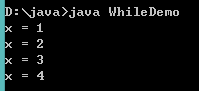
x++; // x进行自增

}

}

}

运行结果如下图所示。



1. 运行结果

在上述代码中，x初始值为1，在满足循环条件x <= 4的情况下，循环体会重复执行，打印x的值并让x进行自增。因此打印结果中x的值分别为1、2、3、4。

大家要注意的是，代码x++用于在每次循环时改变变量x的值，从而达到最终改变循环条件的目的。如果没有这行代码，整个循环会进入无限循环的状态，永远不会结束。

## 循环语句for

for循环语句是最常用的循环语句，一般用在循环次数已知的情况下。for循环语句的语法格式如下：

for（初始化表达式; 循环条件; 操作表达式）{

执行语句

………

}

在上面的语法结构中，for关键字后面()中包括了三部分内容：初始化表达式、循环条件和操作表达式，它们之间用“;”分隔，{}中的执行语句为循环体。

接下来分别用①表示初始化表达式、②表示循环条件、③表示操作表达式、④表示循环体，通过序号来具体分析for循环的执行流程。具体如下：

for（① ; ② ; ③）{

④

}

第一步，执行①

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

第三步，执行④

第四步，执行③，然后重复执行第二步

第五步，退出循环

接下来通过一个案例对自然数1~4进行求和，如下所示。ForDemo01.java

**public** **class** ForDemo01 {

**public** **static** **void** main(String[] args) {

**int** sum = 0; // 定义变量sum，用于记住累加的和

**for** (**int** i = 1; i <= 4; i++) { // i的值会在1~4之间变化

sum += i; // 实现sum与i的累加

}

System.*out*.println("sum = " + sum); // 打印累加的和

}

}

运行结果如下图所示。



1. 运行结果

上述代码中，变量i的初始值为1，在判断条件i<=4为true的情况下，会执行循环体sum+=i，执行完毕后，会执行操作表达式i++，i的值变为2，然后继续进行条件判断，开始下一次循环，直到i=5时，条件i<=4为false，结束循环，执行for循环后面的代码，打印“sum=10”。

为了让初学者能熟悉整个for循环的执行过程，现将上述代码运行期间每次循环中变量sum和i的值通过表2-11罗列出来。

1. sum和i循环中的值

|  |  |  |
| --- | --- | --- |
| **循环次数** | **sum** | **i** |
| 第一次 | 1 | 1 |
| 第二次 | 3 | 2 |
| 第三次 | 6 | 3 |
| 第四次 | 10 | 4 |

## 循环语句do…while

do…while循环语句和while循环语句功能类似，其语法结构如下：

do {

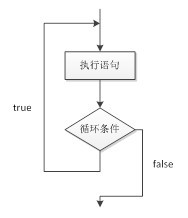
执行语句

………

} while(循环条件);

在上面的语法结构中，关键字do后面{}中的执行语句是循环体。do…while循环语句将循环条件放在了循环体的后面。这也就意味着，循环体会无条件执行一次，然后再根据循环条件来决定是否继续执行。

do…while循环的执行流程如下图所示。



1. do…while循环的执行流程

接下来使用do…while循环语句来实现打印1~4之间的自然数DoWhileDemo.java。

**public** **class** DoWhileDemo {

**public** **static** **void** main(String[] args) {

**int** x = 1; // 定义变量x，初始值为1

**do** {

System.*out*.println("x = " + x); // 打印x的值

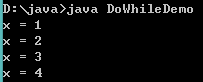
x++; // 将x的值自增

} **while** (x <= 4); // 循环条件

}

}

运行结果如下图所示。



1. 运行结果

我们发现WhileDemo.java与DoWhileDemo.java运行的结果一致，这说明什么呢？这说明do …while循环和while循环能实现同样的功能。然而在程序运行过程中，这两种语句还是有差别的。如果循环条件在循环语句开始时就不成立，那么while循环的循环体一次都不会执行，而do…while循环的循环体还是会执行一次。若将DoWhileDemo.java中的循环条件x<=4改为x < 1，DoWhileDemo.java运行结果会打印x=1，而WhileDemo.java运行结果什么也不会打印。

## 无限循环

最简单无限循环格式：

while(true){}

或

for(;;){}

无限循环存在的原因是并不知道循环多少次，而是根据某些条件，来控制循环。

## 循环嵌套

嵌套循环是指在一个循环语句的循环体中再定义一个循环语句的语法结构。while、do…while、for循环语句都可以进行嵌套，并且它们之间也可以互相嵌套，如最常见的在for循环中嵌套for循环，格式如下：

for(初始化表达式; 循环条件; 操作表达式) {

………

for(初始化表达式; 循环条件; 操作表达式) {

执行语句

………

}

………

}

接下来通过一个练习，来实现使用“\*”打印直角三角形，如下所示。ForForDemo.java

1. **public** **class** ForForDemo {
2. **public** **static** **void** main(String[] args) {
3. **int** i, j; // 定义两个循环变量
4. **for** (i = 1; i <= 9; i++) { // 外层循环
5. **for** (j = 1; j <= i; j++) { // 内层循环
6. System.*out*.print("\*"); // 打印\*
7. }
8. System.*out*.print("\n"); // 换行
9. }
10. }
11. }

运行结果如下所示。



1. 运行结果

在上述代码中定义了两层for循环，分别为外层循环和内层循环，外层循环用于控制打印的行数，内层循环用于打印“\*”，每一行的“\*”个数逐行增加，最后输出一个直角三角形。由于嵌套循环程序比较复杂，下面分步骤进行详细地讲解，具体如下：

第一步，在第3行代码定义了两个循环变量i和j，其中i为外层循环变量，j为内层循环变量。

第二步，在第4行代码将i初始化为1，条件i <= 9为true，首次进入外层循环的循环体。

第三步，在第5行代码将j初始化为1，由于此时i的值为1，条件j <= i为true，首次进入内层循环的循环体，打印一个“\*”。

第四步，执行第5行代码中内层循环的操作表达式j++，将j的值自增为2。

第五步，执行第5行代码中的判断条件j<=i，判断结果为false，内层循环结束。执行后面的代码，打印换行符。

第六步，执行第4行代码中外层循环的操作表达式i++，将i的值自增为2。

第七步，执行第4行代码中的判断条件i<=9，判断结果为true，进入外层循环的循环体，继续执行内层循环。

第八步，由于i的值为2，内层循环会执行两次，即在第2行打印两个“\*”。在内层循环结束时会打印换行符。

第九步，以此类推，在第3行会打印3个“\*”，逐行递增，直到i的值为10时，外层循环的判断条件i <= 9结果为false，外层循环结束，整个程序也就结束了。

## 跳转语句（break、continue）

跳转语句用于实现循环执行过程中程序流程的跳转，在Java中的跳转语句有break语句和continue语句。接下来分别进行详细地讲解。

1. **break语句**

在switch条件语句和循环语句中都可以使用break语句。当它出现在switch条件语句中时，作用是终止某个case并跳出switch结构。当它出现在循环语句中，作用是跳出循环语句，执行后面的代码。关于在switch语句中使用break前面的例程已经用过了，接下来通过下面一段代码，实现将当变量x的值为3时，使用break语句跳出循环，代码如下所示。BreakDemo.java

**public** **class** BreakDemo {

**public** **static** **void** main(String[] args) {

**int** x = 1; // 定义变量x，初始值为1

**while** (x <= 4) { // 循环条件

System.*out*.println("x = " + x); // 条件成立，打印x的值

**if** (x == 3) {

**break**;

}

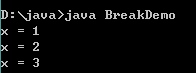
x++; // x进行自增

}

}

}

运行结果如下图所示。



1. 运行结果

在上述带代码中，通过while循环打印x的值，当x的值为3时使用break语句跳出循环。因此打印结果中并没有出现“x=4”。

1. **标记**

当break语句出现在嵌套循环中的内层循环时，它只能跳出内层循环，如果想使用break语句跳出外层循环则需要对外层循环添加标记。接下来将ForForDemo.java稍作修改，控制程序只打印4行“\*”，如下所示。BreakDemo02.java

**public** **class** BreakDemo02 {

**public** **static** **void** main(String[] args) {

**int** i, j; // 定义两个循环变量

itcast: **for** (i = 1; i <= 9; i++) { // 外层循环

**for** (j = 1; j <= i; j++) { // 内层循环

**if** (i > 4) { // 判断i的值是否大于4

**break** itcast; // 跳出外层循环

}

System.*out*.print("\*"); // 打印\*

}

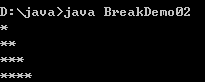
System.*out*.print("\n"); // 换行

}

}

}

运行结果如下图所示。



1. 运行结果

BreakDemo02.java与ForForDemo.java实现原理类似，只是在外层for循环前面增加了标记“itcast”。当i>4时，使用break itcast;语句跳出外层循环。因此程序只打印了4行“\*”。

1. **continue语句**

continue语句用在循环语句中，它的作用是终止本次循环，执行下一次循环。接下来通过一个练习对1~100之内的奇数求和，ContinueDemo.java

**public** **class** ContinueDemo {

**public** **static** **void** main(String[] args) {

**int** sum = 0; // 定义变量sum，用于记住和

**for** (**int** i = 1; i <= 100; i++) {

**if** (i % 2 == 0) { // i是一个偶数，不累加

**continue**; // 结束本次循环

}

sum += i; // 实现sum和i的累加

}

System.*out*.println("sum = " + sum);

}

}

运行结果如下图所示。



1. 运行结果

上述代码中，使用for循环让变量i的值在1~100之间循环，在循环过程中，当i的值为偶数时，将执行continue语句结束本次循环，进入下一次循环。当i的值为奇数时，sum和i进行累加，最终得到1~100之间所有奇数的和，打印“sum = 2500”。

在嵌套循环语句中，continue语句后面也可以通过使用标记的方式结束本次外层循环，用法与break语句相似，在此不再举例说明。

# 猜数字案例

## 案例介绍

我们在日常生活中，经常与朋友们玩猜数字的游戏，非常的有趣。现在我们通过java把这个猜数字游戏编写出来。

猜数字案例是要完成什么样的功能呢？顾名思义，这个游戏就是你出个数字、我来猜。

游戏操作如下：

* 后台预先生成一个1-100之间的随机数，用户键盘录入猜数字
* 如果猜对了，打印“恭喜您，答对了”
* 如果猜错了

猜大了：打印“sorry，您猜大了!”

猜小了：打印“sorry，您猜小了!”

* 直到数字猜到为止

在案例开始之前，我们先要学习如下知识点：

* 随机数类Random
* 选择结构if
* 循环结构while

## 案例需求分析、

猜数字案例的需求如下：

* 后台预先生成一个1-100之间的随机数，用户键盘录入猜数字
* 如果猜对了，打印“恭喜您，答对了”
* 如果猜错了

猜大了：打印“sorry，您猜大了!”

猜小了：打印“sorry，您猜小了!”

* 直到数字猜到为止

我们经过分析，得出如下的分析结果：

* 1.通过Random类中方法nextInt（），生成一个1-100之间的随机数
* 2.输入猜的数字
* 3.通过while循环，进行猜数字对错判断
  + 猜对，跳出循环，游戏结束
  + 猜错了，根据结果，给出提示，接着猜数字，游戏继续
    - 如果猜大了，打印sorry，您猜大了!继续下一次循环
    - 如果猜小了，打印sorry，您猜小了!继续下一次循环

## 实现代码步骤

分析完毕之后，在main()方法中，我们一起来完成代码的编写：GuessNumber.java

**public** **class** GuessNumber {

**public** **static** **void** main(String[] args) {

//1.通过Random类中方法nextInt（），生成一个1-100之间的随机数

**int** randomNumber = **new** Random().nextInt(100);

System.*out*.println("随机数已生成！");

//2.输入猜的数字

System.*out*.println("----请输入您猜的数字：----");

Scanner sc = **new** Scanner(System.*in*);

**int** enterNumber = sc.nextInt();

//3.通过while循环，进行猜数字对错判断

//猜对，跳出循环，游戏结束

**while**(enterNumber != randomNumber){

//猜错了，根据结果，给出提示，接着猜数字，游戏继续

**if**(enterNumber>randomNumber) {

//如果猜大了，打印sorry，您猜大了!继续下一次循环

System.*out*.println("sorry，您猜大了!继续下一次循环");

}**else** {

//如果猜小了，打印sorry，您猜小了!继续下一次循环

System.*out*.println("sorry，您猜小了!继续下一次循环");

}

//输入猜的数字

System.*out*.println("----请输入您猜的数字：----");

enterNumber = sc.nextInt();

}

System.*out*.println("恭喜您，答对了！");

}

}

# 总结

## 知识点总结

* 选择结构 if
  + 方式1:

格式：

if(条件表达式) {

语句体;

}

执行流程：

如果条件表达式值为true, 执行语句体

如果条件表达式值为false,不执行语句体

* + 方式2：

格式:

if(条件表达式) {

语句体1;

} else {

语句体2;

}

执行流程：

如果条件表达式值为true, 执行语句体1

如果条件表达式值为fales,执行语句体2

* + 方式3：

格式：

if(条件表达式1) {

语句体1;

} else if (条件表达式2) {

语句体2;

} else if (条件表达式n...) {

语句体n...

}

...

else {

语句体n+1;

}

执行流程：

哪一个条件表达式成立，执行对应的语句体，

如果所有的条件表达式都不成立，执行最后else中的语句体

* + 注意事项：

1，条件表达式无论是简单还是复杂，结果都是boolean类型的值

2, 如果语句体只有一条语句，大括号可以省略，

如果是多条语句，大括号不能省略，建议什么时候都写上大括号

* 循环语句For
  + 格式：

for (初始化表达式;条件表达式; 控制条件表达式) {

语句体;

}

* + 执行流程：

1，初始化表达式；

2，条件表达式

如果结果为true, 执行语句体

如果结果为flase, 循环结束

3, 当语句体执行后， 执行控制条件表达式

4，接下来，回到第二步, 直到条件表达式为false 循环结束

* 循环语句 while
  + 格式：

初始化表达式;

while (条件表达式) {

语句体;

控制条件表达式;

}

* + 执行流程；

1，初始化表达式；

2，条件表达式

如果结果为true, 执行语句体

如果结果为flase, 循环结束

3, 当语句体执行后， 执行控制条件表达式

4，接下来，回到第二步, 直到条件表达式为false 循环结束跳转语句（break、continue）

* 循环语句 do...while
  + 格式：

初始化表达式;

do {

语句体;

控制条件表达式;

} while (条件表达式);

* + 执行流程:

1，初始化表达式；

2, 执行语句体

3, 执行控制条件表达式

4, 执行条件表达式,

如果结果为true, 继续执行语句体

如果结果为false, 循环结束

* 跳转控制语句

break; 跳出循环

continue; 结束本次循环

* Scanner类
  + 获取键盘录入的数据，对获取数据的具体操作进行了封装，只需要调用方法，即可得到键盘录入的数据。
  + 常用的方法

public int nextInt()：获取键盘录入的数据，返回int类型值

public String next()：获取键盘录入的数据，返回String类型值

* Random类：随机数类，用来产生多种类型的随机数
  + 方法：

public int nextInt(int n) 返回一个，在 0（包括）和指定值（不包括）之间int随机数

第4天 Java基础语法

今日内容介绍

* 流程控制语句（switch）
* 数组

# 流程控制语句

## 选择结构switch

switch 条件语句也是一种很常用的选择语句，它和if条件语句不同，它只能针对某个表达式的值作出判断，从而决定程序执行哪一段代码。例如，在程序中使用数字1~7来表示星期一到星期天，如果想根据某个输入的数字来输出对应中文格式的星期值，可以通过下面的一段伪代码来描述：

用于表示星期的数字

如果等于1，则输出星期一

如果等于2，则输出星期二

如果等于3，则输出星期三

如果等于4，则输出星期四

如果等于5，则输出星期五

如果等于6，则输出星期六

如果等于7，则输出星期天

对于上面一段伪代码的描述，大家可能会立刻想到用刚学过得if…else if…else语句来实现，但是由于判断条件比较多，实现起来代码过长，不便于阅读。Java中提供了一种switch语句来实现这种需求，在switch语句中使用switch关键字来描述一个表达式，使用case关键字来描述和表达式结果比较的目标值，当表达式的值和某个目标值匹配时，会执行对应case下的语句。具体实现代码如下：

switch(用于表示星期的数字) {

case 1 :

输出星期一;

break;

case 2 :

输出星期二;

break;

case 3 :

输出星期三

break;

case 4 :

输出星期四;

break;

case 5 :

输出星期五;

break;

case 6:

输出星期六;

break;

case 7:

输出星期天;

break;

}

* 上面改写后的伪代码便描述了switch语句的基本语法格式，具体如下：

switch (表达式){

case 目标值1:

执行语句1

break;

case 目标值2:

执行语句2

break;

．．．．．．

case 目标值n:

执行语句n

break;

default:

执行语句n+1

break;

}

在上面的格式中，switch语句将表达式的值与每个case中的目标值进行匹配，如果找到了匹配的值，会执行对应case后的语句，如果没找到任何匹配的值，就会执行default后的语句。switch语句中的break关键字将在后面的做具体介绍，此处，我们只需要知道break的作用是跳出switch语句即可。

需要注意的是，在switch语句中的表达式只能是byte、short、char、int类型的值，如果传入其它类型的值，程序会报错。但上述说法并不严谨，实际上在JDK5.0中引入的新特性enum枚举也可以作为switch语句表达式的值，在JDK7.0中也引入了新特性，switch语句可以接收一个String类型的值。

## 选择结构switch练习

接下来通过一个案例演示根据数字来输出中文格式的星期，如下所示。SwitchDemo01.java

**public** **class** SwitchDemo01 {

**public** **static** **void** main(String[] args) {

**int** week = 5;

**switch** (week) {

**case** 1:

System.*out*.println("星期一");

**break**;

**case** 2:

System.*out*.println("星期二");

**break**;

**case** 3:

System.*out*.println("星期三");

**break**;

**case** 4:

System.*out*.println("星期四");

**break**;

**case** 5:

System.*out*.println("星期五");

**break**;

**case** 6:

System.*out*.println("星期六");

**break**;

**case** 7:

System.*out*.println("星期天");

**break**;

**default**:

System.*out*.println("输入的数字不正确...");

**break**;

}

}

}

运行结果如图所示。



1. 运行结果

上述代码中，由于变量week的值为5，整个switch语句判断的结果满足第17行的条件，因此打印“星期五”，例程中的default语句用于处理和前面的case都不匹配的值，将第3行代码替换为int week = 8，再次运行程序，输出结果如下图所示。



1. 运行结果

在使用switch语句的过程中，如果多个case条件后面的执行语句是一样的，则该执行语句只需书写一次即可，这是一种简写的方式。例如，要判断一周中的某一天是否为工作日，同样使用数字1~7来表示星期一到星期天，当输入的数字为1、2、3、4、5时就视为工作日，否则就视为休息日。接下来通过一个案例来实现上面描述的情况，如下所示。SwitchDemo02.java

**public** **class** SwitchDemo02 {

**public** **static** **void** main(String[] args) {

**int** week = 2;

**switch** (week) {

**case** 1:

**case** 2:

**case** 3:

**case** 4:

**case** 5:

// 当 week 满足值 1、2、3、4、5 中任意一个时，处理方式相同

System.*out*.println("今天是工作日");

**break**;

**case** 6:

**case** 7:

// 当 week 满足值 6、7 中任意一个时，处理方式相同

System.*out*.println("今天是休息日");

**break**;

}

}

}

运行结果如下图所示。



1. 运行结果

上述代码中，当变量week值为1、2、3、4、5中任意一个值时，处理方式相同，都会打印“今天是工作日”。同理，当变量week值为6、7中任意一个值时，打印“今天是休息日”。

# 数组

在生活中，我们可能会碰到如下的场景。

现在需要统计某公司员工的工资情况，例如计算平均工资、最高工资等。假设该公司有50名员工，用前面所学的知识完成，那么程序首先需要声明50个变量来分别记住每位员工的工资，这样做会显得很麻烦。

其实在Java中，我们可以使用一个数组来记住这50名员工的工资。数组是指一组数据的集合，数组中的每个数据被称作元素。在数组中可以存放任意类型的元素，但同一个数组里存放的元素类型必须一致。

## 数组的定义

在Java中，可以使用以下格式来定义一个数组。如下

数据类型[] 数组名 = new 数据类型[元素个数或数组长度];

int[] x = new int[100];

上述语句就相当于在内存中定义了100个int类型的变量，第一个变量的名称为x[0]，第二个变量的名称为x[1]，以此类推，第100个变量的名称为x[99]，这些变量的初始值都是0。为了更好地理解数组的这种定义方式，可以将上面的一句代码分成两句来写，具体如下：

int[] x; // 声明一个int[]类型的变量

x = new int[100]; // 创建一个长度为100的数组

接下来，通过两张内存图来详细地说明数组在创建过程中内存的分配情况。

第一行代码 int[] x; 声明了一个变量x，该变量的类型为int[]，即一个int类型的数组。变量x会占用一块内存单元，它没有被分配初始值。内存中的状态如下图所示。



1. 内存状态图

第二行代码 x = new int[100]; 创建了一个数组，将数组的地址赋值给变量x。在程序运行期间可以使用变量x来引用数组，这时内存中的状态会发生变化，如下图所示。



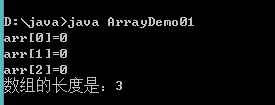
1. 内存状态图

在上图中描述了变量x引用数组的情况。该数组中有100个元素，初始值都为0。数组中的每个元素都有一个索引(也可称为角标)，要想访问数组中的元素可以通过“x[0]、x[1]、……、x[98]、x[99]”的形式。需要注意的是，数组中最小的索引是0，最大的索引是“数组的长度-1”。在Java中，为了方便我们获得数组的长度，提供了一个length属性，在程序中可以通过“数组名.length”的方式来获得数组的长度，即元素的个数。

接下来，通过一个案例来演示如何定义数组以及访问数组中的元素，如下所示。ArrayDemo01.java

1. **public** **class** ArrayDemo01 {
2. **public** **static** **void** main(String[] args) {
3. **int**[] arr; // 声明变量
4. arr = **new** **int**[3]; // 创建数组对象
5. System.*out*.println("arr[0]=" + arr[0]); // 访问数组中的第一个元素
6. System.*out*.println("arr[1]=" + arr[1]); // 访问数组中的第二个元素
7. System.*out*.println("arr[2]=" + arr[2]); // 访问数组中的第三个元素
8. System.*out*.println("数组的长度是：" + arr.length); // 打印数组长度
9. }
10. }

运行结果如下图所示。



1. 运行结果

在上述代码中声明了一个int[]类型变量arr，并将数组在内存中的地址赋值给它。在5~7行代码中通过角标来访问数组中的元素，在第8行代码中通过length属性访问数组中元素的个数。从打印结果可以看出，数组中的三个元素初始值都为0，这是因为当数组被成功创建后，数组中元素会被自动赋予一个默认值，根据元素类型的不同，默认初始化的值也是不一样的。具体如下表所示。

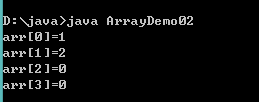
1. 元素默认值

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

如果在使用数组时，不想使用这些默认初始值，也可以显式地为这些元素赋值。接下来通过一个程序来学习如何为数组的元素赋值，如下所示。ArrayDemo02.java

1. **public** **class** ArrayDemo02 {
2. **public** **static** **void** main(String[] args) {
3. **int**[] arr = **new** **int**[4]; // 定义可以存储4个整数的数组
4. arr[0] = 1; // 为第1个元素赋值1
5. arr[1] = 2; // 为第2个元素赋值2
6. // 下面的代码是打印数组中每个元素的值
7. System.*out*.println("arr[0]=" + arr[0]);
8. System.*out*.println("arr[1]=" + arr[1]);
9. System.*out*.println("arr[2]=" + arr[2]);
10. System.*out*.println("arr[3]=" + arr[3]);
11. }
12. }

运行结果如下图所示。



1. 运行结果

在上述代码中，第3行代码定义了一个数组，此时数组中每个元素都为默认初始值0。第2、3行代码通过赋值语句将数组中的元素arr[0]和arr[1]分别赋值为1和2，而元素arr[2]和arr[3]没有赋值，其值仍为0，因此打印结果中四个元素的值依次为1、2、0、0。

在定义数组时只指定数组的长度，由系统自动为元素赋初值的方式称作动态初始化。

在初始化数组时还有一种方式叫做静态初始化，就是在定义数组的同时就为数组的每个元素赋值。数组的静态初始化有两种方式，具体格式如下：

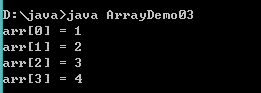
1、类型[] 数组名 = new 类型[]{元素，元素，……};

2、类型[] 数组名 = {元素，元素，元素，……};

上面的两种方式都可以实现数组的静态初始化，但是为了简便，建议采用第二种方式。接下来通过一段代码来演示数组静态初始化的效果，如下所示。ArrayDemo03.java

1. **public** **class** ArrayDemo03 {
2. **public** **static** **void** main(String[] args) {
3. **int**[] arr = { 1, 2, 3, 4 }; // 静态初始化
4. // 下面的代码是依次访问数组中的元素
5. System.*out*.println("arr[0] = " + arr[0]);
6. System.*out*.println("arr[1] = " + arr[1]);
7. System.*out*.println("arr[2] = " + arr[2]);
8. System.*out*.println("arr[3] = " + arr[3]);
9. }
10. }

运行结果如下图所示。



1. 运行结果

上述代码中采用静态初始化的方式为数组每个元素赋予初值，分别是1、2、3、4。需要注意的是，第3行代码千万不可写成int[] arr = new int[4]{1,2,3,4};，这样写编译器会报错。原因在于编译器会认为数组限定的元素个数[4]与实际存储的元素{1,2,3,4}个数有可能不一致，存在一定的安全隐患。

## 数组遍历

在操作数组时，经常需要依次访问数组中的每个元素，这种操作称作数组的遍历。接下来通过一个案例来学习如何使用for循环来遍历数组，如下所示。ArrayDemo04.java

**public** **class** ArrayDemo04 {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 1, 2, 3, 4, 5 }; // 定义数组

// 使用for循环遍历数组的元素

**for** (**int** i = 0; i < arr.length; i++) {

System.*out*.println(arr[i]); // 通过索引访问元素

}

}

}

运行结果如下图所示。



1. 运行结果

上述代码中，定义一个长度为5的数组arr，数组的角标为0~4。由于for循环中定义的变量i的值在循环过程中为0~4，因此可以作为索引，依次去访问数组中的元素，并将元素的值打印出来。

## 数组的常见问题

数组在编写程序时应用非常广泛，灵活地使用数组对实际开发很重要。接下来，本节将针对数组的常见操作进行详细地讲解，如数组的遍历、最值的获取、数组的排序等。

### 数组最值

在操作数组时，经常需要获取数组中元素的最值。接下来通过一个案例来演示如何获取数组中元素的最大值，如下所示。ArrayDemo05.java

**public** **class** ArrayDemo05 {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 4, 1, 6, 3, 9, 8 }; // 定义一个数组

**int** max = arr[0]; // 定义变量max用于记住最大数，首先假设第一个元素为最大值

// 下面通过一个for循环遍历数组中的元素

**for** (**int** x = 1; x < arr.length; x++) {

**if** (arr[x] > max) { // 比较 arr[x]的值是否大于max

max = arr[x]; // 条件成立，将arr[x]的值赋给max

}

}

System.*out*.println("max=" + max); // 打印最大值

}

}

运行结果如下图所示。



1. 运行结果

上述代码中，定义了一个临时变量max，用于记住数组的最大值。通过for 循环获取数组中的最大值，赋值给max变量。

首先假设数组中第一个元素arr[0]为最大值，然后使用for循环对数组进行遍历，在遍历的过程中只要遇到比max值还大的元素，就将该元素赋值给max。这样一来，变量max就能够在循环结束时记住数组中的最大值。需要注意的是，在for循环中的变量i是从1开始的，这样写的原因是程序已经假设第一个元素为最大值，for循环中只需要从第二个元素开始比较，从而提高程序的运行效率。

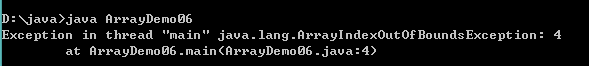
### 数组异常

#### 数组越界异常

每个数组的索引都有一个范围，即0~length-1。在访问数组的元素时，索引不能超出这个范围，否则程序会报错，如下所示。ArrayDemo06.java

1. **public** **class** ArrayDemo06 {
2. **public** **static** **void** main(String[] args) {
3. **int**[] arr = **new** **int**[4]; // 定义一个长度为4的数组
4. System.*out*.println("arr[0]=" + arr[4]); // 通过角标4访问数组元素
5. }
6. }

运行结果如下图所示。



上图运行结果中所提示的错误信息是数组越界异常ArrayIndexOutOfBoundsException，出现这个异常的原因是数组的长度为4，其索引范围为0~3，而上述代码中的第4行代码使用索引4来访问元素时超出了数组的索引范围。

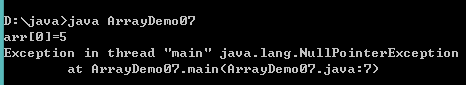
所谓异常指程序中出现的错误，它会报告出错的异常类型、出错的行号以及出错的原因，关于异常在后面的章节会有详细地讲解。

#### 空指针异常

在使用变量引用一个数组时，变量必须指向一个有效的数组对象，如果该变量的值为null，则意味着没有指向任何数组，此时通过该变量访问数组的元素会出现空指针异常，接下来通过一个案例来演示这种异常，如下所示。ArrayDemo07.java

1. **public** **class** ArrayDemo07 {
2. **public** **static** **void** main(String[] args) {
3. **int**[] arr = **new** **int**[3]; // 定义一个长度为3的数组
4. arr[0] = 5; // 为数组的第一个元素赋值
5. System.*out*.println("arr[0]=" + arr[0]); // 访问数组的元素
6. arr = **null**; // 将变量arr置为null
7. System.*out*.println("arr[0]=" + arr[0]); // 访问数组的元素
8. }
9. }

运行结果如下图所示。



1. 运行结果

通过上图所示的运行结果可以看出，上述代码中第4、5行代码都能通过变量arr正常地操作数组。第6行代码将变量置为null，当第7行代码再次访问数组时就出现了空指针异常NullPointerException。

## 二维数组

在程序中可以通过一个数组来保存某个班级学生的考试成绩，试想一下，如果要统计一个学校各个班级学生的考试成绩，又该如何实现呢？这时就需要用到多维数组，多维数组可以简单地理解为在数组中嵌套数组。在程序中比较常见的就是二维数组，接下来针对二维数组进行详细地讲解。

### 二维数组的定义格式

二维数组的定义有很多方式，接下来针对几种常见的方式进行详细地讲解，具体如下：

第一种方式：

int[][] arr = new int[3][4];

上面的代码相当于定义了一个3\*4的二维数组，即二维数组的长度为3，二维数组中的每个元素又是一个长度为4的数组，接下来通过一个图来表示这种情况，如下图所示。

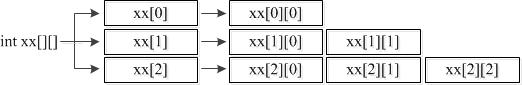


1. 二维数组

第二种方式：

int[][] arr = new int[3][];

第二种方式和第一种类似，只是数组中每个元素的长度不确定，接下来通过一个图来表示这种情况，如下图所示。

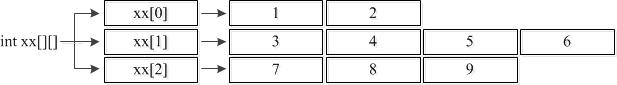


1. 二维数组

第三种方式：

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

上面的二维数组中定义了三个元素，这三个元素都是数组，分别为{1,2}、{3,4,5,6}、{7,8,9}，接下来通过一个图来表示这种情况，如图2-54所示。



1. 二维数组

对二维数组中元素的访问也是通过角标的方式，如需访问二维数组中第一个元素数组的第二个元素，具体代码如下：

arr[0][1];

### 二维数组元素的访问

操作二维数组时，经常需要获取数组中元素的值。接下来通过一个案例来演示如何获取数组中元素值，如下所示。ArrayDemo08.java

class ArrayDemo08 {

public static void main(String[] args){

//定义二维数组的方式

int[][] arr = new int[3][4];

System.out.println( arr );

System.out.println("二维数组的长度: " + arr.length);

//获取二维数组的3个元素

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

System.out.println( arr[1] );

System.out.println( arr[2] );

System.out.println("打印第一个一维数组的元素值");

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

System.out.println( arr[0][1] );//访问的为二维数组中第1个一维数组的第2个元素

System.out.println( arr[0][2] );

System.out.println( arr[0][3] );

System.out.println("打印第二个一维数组的元素值");

System.out.println( arr[1][0] );

System.out.println( arr[1][1] );

System.out.println( arr[1][2] );

System.out.println( arr[1][3] );

System.out.println("打印第三个一维数组的元素值");

System.out.println( arr[2][0] );

System.out.println( arr[2][1] );

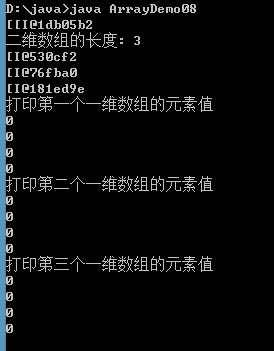
System.out.println( arr[2][2] );

System.out.println( arr[2][3] );

}

}

运行结果如下图所示：



1. 运行结果

### 二维数组元素遍历与数组元素累加和

学习完了数组元素的访问，我们来学习下数组的遍历及数组的元素累加和操作。

class ArrayDemo09 {

public static void main(String[] args){

//一维数组的求累加和并遍历

int[] arr = {10,20,30,40,50};

int sum = 0;

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

//System.out.println(arr[i]);

sum += arr[i];

}

System.out.println("sum= " + sum);

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

//二维数组的求累加和并遍历

int[][] arr2 = { {1,2},{3,4,5},{6,7,8,9,10} };

int sum2 = 0;

for (int i=0; i<arr2.length; i++) {

for (int j=0; j<arr2[i].length; j++) {

//System.out.println(arr2[i][j])

sum2 += arr2[i][j];

}

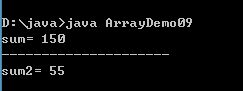
}

System.out.println("sum2= "+ sum2);

}

}

运行结果如下图所示：



1. 运行结果

### 公司销售总和练习

接下来通过一个案例来熟悉二维数组的使用。

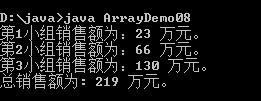
例如要统计一个公司三个销售小组中每个小组的总销售额以及整个公司的销售额。如下所示

* 第一小组销售额为{11, 12}万元
* 第二小组销售额为{21, 22, 23}万元
* 第三小组销售额为{31, 32, 33, 34}万元。

ArrayDemo08.java

1. **public** **class** ArrayDemo10 {
2. **public** **static** **void** main(String[] args) {
3. **int**[][] arr = **new** **int**[3][]; // 定义一个长度为3的二维数组
4. arr[0] = **new** **int**[] { 11, 12 }; // 为数组的元素赋值
5. arr[1] = **new** **int**[] { 21, 22, 23 };
6. arr[2] = **new** **int**[] { 31, 32, 33, 34 };
8. **int** sum = 0; // 定义变量记录总销售额
9. **for** (**int** i = 0; i < arr.length; i++) { // 遍历数组元素
10. **int** groupSum = 0; // 定义变量记录小组销售总额
11. **for** (**int** j = 0; j < arr[i].length; j++) { // 遍历小组内每个人的销售额
12. groupSum = groupSum + arr[i][j];
13. }
14. sum = sum + groupSum; // 累加小组销售额
15. System.*out*.println("第" + (i + 1) + "小组销售额为：" + groupSum + " 万元");
16. }
17. System.*out*.println("总销售额为: " + sum + " 万元");
18. }
19. }

运行结果如下图所示。



1. 运行结果

在上述代码中，第3行代码定义了一个长度为3的二维数组，并在4~6行代码中为数组的每个元素赋值。例程中还定义了两个变量sum和groupSum，其中sum用来记录公司的总销售额，groupSum用来记录每个销售小组的销售额。当通过嵌套for循环统计销售额时，外层循环对三个销售小组进行遍历，内层循环对每个小组员工的销售额进行遍历，内层循环每循环一次就相当于将一个小组员工的销售总额统计完毕，赋值给groupSum，然后把groupSum的值与sum的值相加赋值给sum。当外层循环结束时，三个销售小组的销售总额groupSum都累加到sum中，即统计出了整个公司的销售总额。

# 随机点名器案例

## 案例介绍

随机点名器，即在全班同学中随机的打印出一名同学名字。

要做的随机点名器，它具备以下3个内容：

* 存储所有同学姓名
* 总览全班同学姓名
* 随机点名其中一人，打印到控制台

在编写案例前，我们先来学习下本案例中所涉及到的知识点：

* 数组

## 案例需求分析

在全班同学中随机的打印出一名同学名字。

我们对本案例进行分析，得出如下分析结果：

1.存储全班同学名字

2.打印全班同学每一个人的名字

3.在班级总人数范围内，随机产生一个随机数，查找该随机数所对应的同学名字

在存储同学姓名时，如果对每一个同学都定义一个变量进行姓名存储，则会出现过多孤立的变量，很难一次性将全部数据持有。此时，我们可以使用数组解决多个数据的存储问题。

## 实现代码步骤

编写CallName.java文件，完成程序的编写。

* main方法中进行步骤1、2、3的代码实现

**public** **static** **void** main(String[] args) {

System.*out*.println("--------随机点名器--------");

// 创建一个存储多个同学名字的容器（数组）

String[] students = **new** String[3];

//1.存储全班同学名字

//2.打印全班同学每一个人的名字

//3.获取随机点名到的学生姓名，并打印

}

* 1.存储所有同学姓名

/\*\*

\* 1.存储全班同学名字

\* 创建一个存储多个同学名字的容器（数组）

\* 键盘输入每个同学的名字，存储到容器中（数组）

\*/

//键盘输入多个同学名字存储到容器中

Scanner sc = **new** Scanner(System.*in*);

**for** (**int** i = 0; i < students.length; i++) {

System.*out*.println("存储第"+i+"个名称：");

students[i] = sc.next();

}

* 2.打印全班同学每一个人的名字

/\*\*

\* 2打印全班同学每一个人的名字

\*/

//遍历数组，得到每个同学名字

**for** (**int** i = 0; i < students.length; i++) {

String name = students[i];

//打印同学名字

System.*out*.println("第"+i+"个学生名称：" + name);

}

* 3.获取随机点名到的学生姓名，并打印

/\*\*

\* 3.在班级总人数范围内，随机产生一个随机数

\*/

//根据数组长度，获取随机索引

**int** index = **new** Random().nextInt(students.length);

//通过随机索引从数组中获取名称

String name = students[index];

//返回随机点到的名称

上述代码中，通过随机数类Random产生一个从0到数组长度的随机索引。使用该索引获取students数组中对应的值，便得到了全班同学的随机姓名。

# 总结

## 知识点总结

* 数组
  + 它是一个用来存储同一个数据类型多个元素的一个容器（数组长度是固定的，数组中存储的元素的数据类型要求一致）
  + 格式：

格式1：

数据类型[] 数组名 = new 数据类型[数组长度];

格式2:

数据类型[] 数组名 = new 数据类型[]{元素值1,元素值2,..};

格式3：

数据类型[] 数组名 = {元素值1,元素值2,..};

* + 数组操作的常见问题：

NullPointerException: 空指针异常

ArrayIndexOutOfBoundsException: 数组越界异常

* 二维数组：
  + 它是一个包含多个一维数组的数组
  + 特点：二维数组中的每个元素都是一个一维数组
  + 格式：

格式1：

数据类型[][] 数组名 = new 数据类型[m][n];

m: 代表二维数组中一维数组的个数

n: 代表每个一维数组中元素的个数

格式2：

数据类型[][] 数组名 = new 数据类型[m][];

m: 代表二维数组中一维数组的个数

每一个一维数组通过赋值来确定数组长度

格式3：

数据类型[][] 数组名 = {{元素值1，元素值2,..},{元素值1，元素值2,..},..};

第5天 Java基础语法

今日内容介绍

* 方法

# 方法

## 方法概述

在我们的日常生活中，方法可以理解为要做某件事情，而采取的解决办法。

如：小明同学在路边准备坐车来学校学习。这就面临着一件事情（坐车到学校这件事情）需要解决，解决办法呢？可采用坐公交车或坐出租车的方式来学校，那么，这种解决某件事情的办法，我们就称为方法。

在java中，方法就是用来完成解决某件事情或实现某个功能的办法。

方法实现的过程中，会包含很多条语句用于完成某些有意义的功能——通常是处理文本，控制输入或计算数值。

我们可以通过在程序代码中引用方法名称和所需的参数，实现在该程序中执行（或称调用）该方法。方法，一般都有一个返回值，用来作为事情的处理结果。

## 方法的语法格式

在Java中，声明一个方法的具体语法格式如下：

修饰符 返回值类型 方法名(参数类型 参数名1,参数类型 参数名2,．．．．．．){

执行语句

………

return 返回值;

}

对于上面的语法格式中具体说明如下：

* 修饰符：方法的修饰符比较多，有对访问权限进行限定的，有静态修饰符static，还有最终修饰符final等，这些修饰符在后面的学习过程中会逐步介绍
* 返回值类型：用于限定方法返回值的数据类型
* 参数类型：用于限定调用方法时传入参数的数据类型
* 参数名：是一个变量，用于接收调用方法时传入的数据
* return关键字：用于结束方法以及返回方法指定类型的值
* 返回值：被return语句返回的值，该值会返回给调用者

需要特别注意的是，方法中的“参数类型 参数名1，参数类型 参数名2”被称作参数列表，它用于描述方法在被调用时需要接收的参数，如果方法不需要接收任何参数，则参数列表为空，即()内不写任何内容。方法的返回值必须为方法声明的返回值类型，如果方法中没有返回值，返回值类型要声明为void，此时，方法中return语句可以省略。

接下来通过一个案例来演示方法的定义与使用，如下图所示。MethodDemo01.java

**public** **class** MethodDemo01 {

**public** **static** **void** main(String[] args) {

**int** area = *getArea*(3, 5); // 调用 getArea方法

System.*out*.println(" The area is " + area);

}

// 下面定义了一个求矩形面积的方法，接收两个参数，其中x为高，y为宽

**public** **static** **int** getArea(**int** x, **int** y) {

**int** temp = x \* y; // 使用变量temp记住运算结果

**return** temp; // 将变量temp的值返回

}

}

运行结果如下图所示。

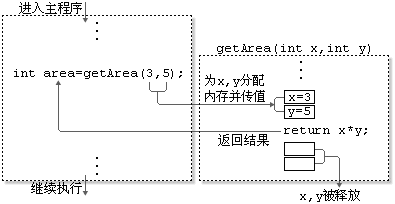


1. 运行结果

在上述代码中，定义了一个getArea()方法用于求矩形的面积，参数x和y分别用于接收调用方法时传入的高和宽，return语句用于返回计算所得的面积。在main()方法中通过调用getArea()方法，获得矩形的面积，并将结果打印。

## 方法调用图解

接下来通过一个图例演示getArea()方法的整个调用过程，如下图所示。



1. getArea()方法的调用过程

从上图中可以看出，在程序运行期间，参数x和y相当于在内存中定义的两个变量。当调用getArea()方法时，传入的参数3和5分别赋值给变量x和y，并将x\*y的结果通过return语句返回，整个方法的调用过程结束，变量x和y被释放。

## 方法定义练习

分别定义如下方法：

定义无返回值无参数方法，如打印3行，每行3个\*号的矩形

定义有返回值无参数方法，如键盘录入得到一个整数

定义无返回值有参数方法，如打印指定M行，每行N个\*号的矩形

定义有返回值有参数方法，如求三个数的平均值

* 无返回值无参数方法，如打印3行，每行3个\*号的矩形

public static void printRect(){

//打印3行星

for (int i=0; i<3; i++) {

//System.out.println("\*\*\*"); 相当于是打印3颗星，换行

//每行打印3颗星

for (int j=0; j<3; j++) {

System.out.print("\*"); // \*\*\*

}

System.out.println();

}

}

* 有返回值无参数方法，如键盘录入得到一个整数

public static int getNumber(){

Scanner sc = new Scanner(System.in);

int number = sc.nextInt();

return number;

}

* 无返回值有参数方法，如打印指定M行，每行N个\*号的矩形

public static void printRect2(int m, int n){

//打印M行星

for (int i=0; i<m; i++) {

//每行中打印N颗星

for (int j=0; j<n; j++) {

System.out.print("\*");

}

System.out.println();

}

}

* 有返回值有参数方法，如求三个数的平均值

public static double getAvg(double a, double b, double c) {

double result = (a+b+c)/3;

return result;

}

## 方法的重载

我们假设要在程序中实现一个对数字求和的方法，由于参与求和数字的个数和类型都不确定，因此要针对不同的情况去设计不同的方法。接下来通过一个案例来实现对两个整数相加、对三个整数相加以及对两个小数相加的功能，具体实现如下所示。MethodDemo02.java

**public** **class** MethodDemo02 {

**public** **static** **void** main(String[] args) {

// 下面是针对求和方法的调用

**int** sum1 = *add01*(1, 2);

**int** sum2 = *add02*(1, 2, 3);

**double** sum3 = *add03*(1.2, 2.3);

// 下面的代码是打印求和的结果

System.*out*.println("sum1=" + sum1);

System.*out*.println("sum2=" + sum2);

System.*out*.println("sum3=" + sum3);

}

// 下面的方法实现了两个整数相加

**public** **static** **int** add01(**int** x, **int** y) {

**return** x + y;

}

// 下面的方法实现了三个整数相加

**public** **static** **int** add02(**int** x, **int** y, **int** z) {

**return** x + y + z;

}

// 下面的方法实现了两个小数相加

**public** **static** **double** add03(**double** x, **double** y) {

**return** x + y;

}

}

运行结果如下图所示。



1. 运行结果

从上述代码不难看出，程序需要针对每一种求和的情况都定义一个方法，如果每个方法的名称都不相同，在调用时就很难分清哪种情况该调用哪个方法。

为了解决这个问题，Java允许在一个类中定义多个名称相同的方法，但是参数的类型或个数必须不同，这就是方法的重载。

下面的三个方法互为重载关系

* public static int add(int x,int y) {逻辑} //两个整数加法
* public static int add(int x,int y,int z) {逻辑} //三个整数加法
* public static int add(double x,double y) {逻辑} //两个小数加法

接下来通过方法重载的方式进行修改，如下所示。MethodDemo03.java

**public** **class** MethodDemo03 {

**public** **static** **void** main(String[] args) {

// 下面是针对求和方法的调用

**int** sum1 = *add*(1, 2);

**int** sum2 = *add*(1, 2, 3);

**double** sum3 = *add*(1.2, 2.3);

// 下面的代码是打印求和的结果

System.*out*.println("sum1=" + sum1);

System.*out*.println("sum2=" + sum2);

System.*out*.println("sum3=" + sum3);

}

// 下面的方法实现了两个整数相加

**public** **static** **int** add(**int** x, **int** y) {

**return** x + y;

}

// 下面的方法实现了三个整数相加

**public** **static** **int** add(**int** x, **int** y, **int** z) {

**return** x + y + z;

}

// 下面的方法实现了两个小数相加

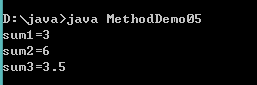
**public** **static** **double** add(**double** x, **double** y) {

**return** x + y;

}

}

MethodDemo02.java的运行结果和MethodDemo03.java一样，如下图所示。



1. 运行结果

上述代码中定义了三个同名的add()方法，它们的参数个数或类型不同，从而形成了方法的重载。

在main()方法中调用add()方法时，通过传入不同的参数便可以确定调用哪个重载的方法，如add(1,2)调用的是两个整数求和的方法。值得注意的是，方法的重载与返回值类型无关，它只有两个条件，一是方法名相同，二是参数个数或参数类型不相同。

### 重载的注意事项

* 重载方法参数必须不同：

参数个数不同，如method(int x)与method(int x,int y)不同

参数类型不同，如method(int x)与method(double x)不同g

参数顺序不同，如method(int x,double y)与method(double x,int y)不同

* 重载只与方法名与参数类型相关与返回值无关

如void method(int x)与int method(int y)不是方法重载，不能同时存在

* 重载与具体的变量标识符无关

如method(int x)与method(int y)不是方法重载，不能同时存在

### 参数传递

参数传递，可以理解当我们要调用一个方法时，我们会把指定的数值，传递给方法中的参数，这样方法中的参数就拥有了这个指定的值，可以使用该值，在方法中运算了。这种传递方式，我们称为参数传递。

* 在这里，定义方法时，参数列表中的变量，我们称为形式参数
* 调用方法时，传入给方法的数值，我们称为实际参数

我们看下面的两段代码，来明确下参数传递的过程：

**public** **class** ArgumentsDemo01 {

**public** **static** **void** main(String[] args) {

**int** a=5;

**int** b=10;

*change*(a, b);//调用方法时，传入的数值称为实际参数

System.*out*.println("a=" + a);

System.*out*.println("b=" + b);

}

**public** **static** **void** change(**int** a, **int** b){//方法中指定的多个参数称为形式参数

a=200;

b=500;

}

}

程序的运行结果如下：



再看另一段代码

**public** **class** ArgumentsDemo02 {

**public** **static** **void** main(String[] args) {

**int**[] arr = { 1, 2, 3 };

*change*(arr);// 调用方法时，传入的数值称为实际参数

**for** (**int** i = 0; i < arr.length; i++) {

System.*out*.println(arr[i]);

}

}

**public** **static** **void** change(**int**[] arr) {// 方法中指定的多个参数称为形式参数

**for** (**int** i = 0; i < arr.length; i++) {

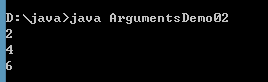
arr[i] \*= 2;

}

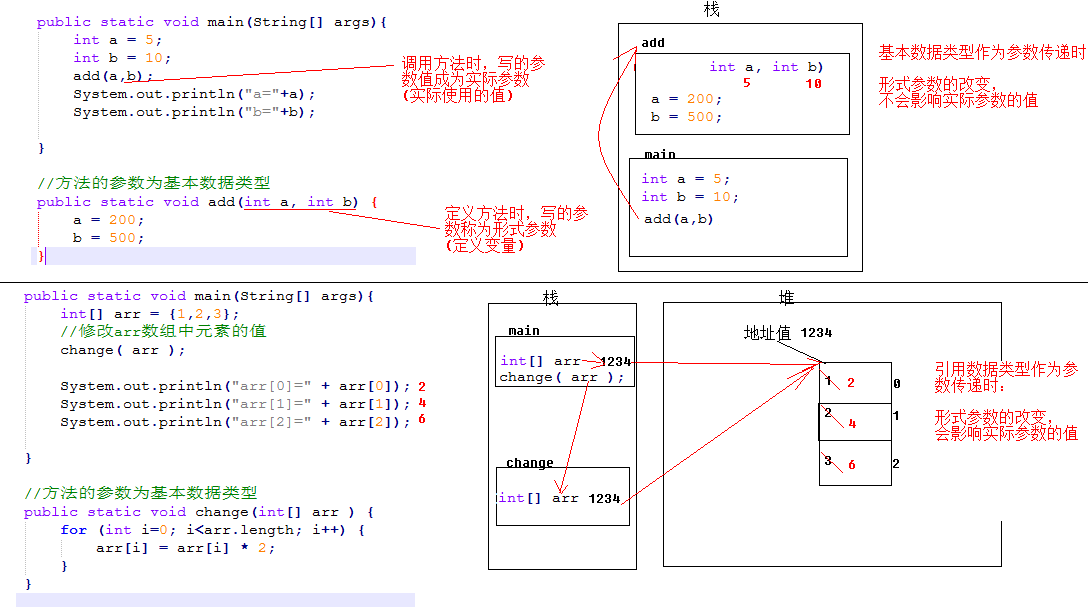
}

}

程序的运行结果如下：



### 参数传递图解与结论



通过上面的两段程序可以得出如下结论：

* 当调用方法时，如果传入的数值为基本数据类型（包含String类型），形式参数的改变对实际参数不影响
* 当调用方法时，如果传入的数值为引用数据类型（String类型除外），形式参数的改变对实际参数有影响

# 随机点名器案例

## 案例介绍

随机点名器，即在全班同学中随机的打印出一名同学名字。

要做的随机点名器，它具备以下3个内容：

* 存储所有同学姓名
* 总览全班同学姓名
* 随机点名其中一人，打印到控制台

## 案例需求分析

在全班同学中随机的打印出一名同学名字。

我们对本案例进行分析，得出如下分析结果：

1.存储全班同学名字

2.打印全班同学每一个人的名字

3.在班级总人数范围内，随机产生一个随机数，查找该随机数所对应的同学名字

该案例须有以下3个内容：

* 存储所有同学姓名
* 总览全班同学姓名
* 随机点名其中一人，打印到控`制台

随机点名器明确地分为了三个功能。如果将多个独立功能的代码写到一起，则代码相对冗长，我们可以针对不同的功能可以将其封装到一个方法中，将完整独立的功能分离出来。

而在存储同学姓名时，如果对每一个同学都定义一个变量进行姓名存储，则会出现过多孤立的变量，很难一次性将全部数据持有。此时，我们可以使用数组解决多个数据的存储问题。

## 实现代码步骤

编写CallName.java文件，完成程序的编写。

* main方法中调用三个独立方法

**public** **static** **void** main(String[] args) {

System.*out*.println("--------随机点名器--------");

// 创建一个存储多个同学名字的容器（数组）

String[] students = **new** String[3];

/\*

\* 1.存储全班同学名字

\*/

addStudentName(students);

/\*

\* 2.打印全班同学每一个人的名字

\*/

printStudentName(students);

/\*

\* 3.获取随机点名到的学生姓名，并打印

\*/

String randomName = *randomStudentName*(students);

System.*out*.println("被点到名的同学是 :" + randomName);

}

* 1.存储所有同学姓名

/\*\*

\* 1.存储全班同学名字

\* 创建一个存储多个同学名字的容器（数组）

\* 键盘输入每个同学的名字，存储到容器中（数组）

\*/

**public** **static** **void** addStudentName(String[] students) {

//键盘输入多个同学名字存储到容器中

Scanner sc = **new** Scanner(System.*in*);

**for** (**int** i = 0; i < students.length; i++) {

System.*out*.println("存储第"+i+"个名称：");

students[i] = sc.next();

}

}

上述方法中，通过键盘录入，完成为指定数组元素赋值。方法定义时，将参数定义为字符串数组，用于接收存放的同学姓名。

* 打印全班同学每一个人的名字

/\*\*

\* 2打印全班同学每一个人的名字

\*/

**public** **static** **void** printStudentName(String[] students) {

//遍历数组，得到每个同学名字

**for** (**int** i = 0; i < students.length; i++) {

String name = students[i];

//打印同学名字

System.*out*.println("第"+i+"个学生名称：" + name);

}

}

上述方法中，方法参数students数组中存储了所有学生的姓名。通过遍历将数组中的每一个元素访问一遍，得到每一个同学名称。

* 3.获取随机点到的学生姓名

/\*\*

\* 3.在班级总人数范围内，随机产生一个随机数,返回随机数位置上的学生姓名

\*/

**public** **static** String randomStudentName(String[] students) {

//根据数组长度，获取随机索引

**int** index = **new** Random().nextInt(students.length);

//通过随机索引从数组中获取名称

String name = students[index];

//返回随机点到的名称

**return** name;

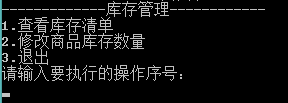
}

上述方法中，通过随机数类Random产生一个从0到数组长度的随机索引。使用该索引获取students数组中对应的值，便得到了全班同学的随机姓名。

# 库存管理案例

## 案例介绍

现在，我们将原有的库存管理案例，进行业务逻辑的封装。



将对下列功能进行方法封装：

* 打印库存清单功能
* 库存商品数量修改功能
* 退出程序功能

## 案例需求分析

管理员能够进行的操作有3项（查看、修改、退出），我们可以采用（switch）菜单的方式来完成。

-------------库存管理------------

1.查看库存清单

2.修改商品库存数量

3.退出

请输入要执行的操作序号：

每一项功能操作，我们采用方法进行封装，这样，可使程序的可读性增强。

选择“1.查看库存清单”功能，则控制台打印库存清单

选择“2.修改商品库存数量”功能，则对每种商品库存数进行更新

选择“3.退出”功能，则退出库存管理，程序结束

## 实现代码步骤

编写代码Demo库存管理.java，完成如下功能：

* 功能菜单

/\*\*

\* 库存管理功能菜单

\* **@return** 管理员键盘输入的功能操作序号

\*/

**public** **static** **int** chooseFunction() {

System.*out*.println("-------------库存管理------------");

System.*out*.println("1.查看库存清单");

System.*out*.println("2.修改商品库存数量");

System.*out*.println("3.退出");

System.*out*.println("请输入要执行的操作序号：");

//接收键盘输入的功能选项序号

Scanner sc = **new** Scanner(System.*in*);

**int** choose = sc.nextInt();

**return** choose;

}

上述方法用来完成库存管理功能菜单的显示、接收管理员选择的功能操作序号。这是完成了案例的第一步。接下来完成“查看、修改、退出”这三项功能。

* 编写main主方法，调用库存管理功能菜单方法，与“查看、修改、退出”这三个方法。

**public** **static** **void** main(String[] args) {

//记录库存商品信息

//品牌型号

String[] brands = **new** String[]{"MacBookAir", "ThinkpadT450"};

//尺寸大小

**double**[] sizes = **new** **double**[]{13.3, 14.0};

//价格

**double**[] prices = **new** **double**[]{6988.88, 5999.99};

//库存个数

**int**[] counts = **new** **int**[]{0, 0};

//通过while循环模拟管理员进行功能重复选择操作

while (true) {

//打印功能菜单操作,接收键盘输入的功能选项序号

**int** choose = *chooseFunction*();

//执行序号对应的功能

**switch** (choose) {

**case** 1://查看库存清单

*printStore*(brands, sizes, prices, counts);

**break**;

**case** 2://修改商品库存数量

*update*(brands, counts);

**break**;

**case** 3://退出

*exit*();

**return**;

default:

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

System.*out*.println("功能选择有误，请输入正确的功能序号!");

**break**;

}

}

}

在主方法中，创建了5个数组，分别用来存储商品的品牌型号、尺寸大小、价格、配置、库存个数，通过接收到的功能选项序号，进行switch语句判断后，调用对应的功能方法。

* 查看库存清单功能

/\*\*

\* 查看库存清单

\* **@param** brands 商品品牌型号

\* **@param** sizes 商品尺寸大小

\* **@param** prices 商品价格

\* **@param** counts 商品库存个数

\*/

**public** **static** **void** printStore(String[] brands, **double**[] sizes, **double**[] prices, **int**[] counts) {

//统计总库存个数、统计库存总金额

**int** totalCount = 0;

**double** totalMoney = 0.0;

**for** (**int** i = 0; i < brands.length; i++) {

totalCount += counts[i];

totalMoney += counts[i] \* prices[i];

}

//列表顶部

System.*out*.println("---------------------------查看库存清单--------------------------");

System.*out*.println("品牌型号 尺寸 价格 库存数");

//列表中部

**for** (**int** i = 0; i < brands.length; i++) {

System.*out*.println(brands[i]+" "+sizes[i]+" "+prices[i]+" "++counts[i]);

}

//列表底部

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

System.*out*.println("总库存数："+totalCount);

System.*out*.println("库存商品总金额："+totalMoney);

}

上述方法用来完成打印库存清单功能，5个方法参数用来打印的库存商品相关信息

* 修改商品库存数量功能

/\*\*

\* 修改商品库存数量

\* **@param** brands 商品品牌型号

\* **@param** counts 商品库存个数

\*/

**public** **static** **void** update(String[] brands, **int**[] counts){

System.*out*.println("------------修改商品库存数量-----------");

**for** (**int** i = 0; i < brands.length; i++) {

System.*out*.println("请输入"+ brands[i] +"商品库存数");

counts[i] = **new** Scanner(System.*in*).nextInt();

}

}

上述方法用来完成修改商品库存数量功能，2个方法参数用来指定所要修改的商品与库存数

* 退出功能

/\*\*

\* 退出

\*/

public static void exit(){

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

System.*out*.println("您已退出系统");

}

上述方法用来完成退出程序的功能

# 总结

## 知识点总结

* 方法
  + 格式：

修饰符 返回值类型 方法名(参数类型 参数名1, 参数类型 参数名2, ...) {

方法体;

return 返回值;

}

* + 方法使用的注意事项：

1，方法不调用，自己不执行

2，方法中不能定义方法， 但是，方法中可以调用方法

3, 方法定义的位置在类中，其他方法的外面

4, 如果方法没有明确的返回值类型，使用'空'类型， void表示

5，void只能在方法返回值类型位置使用，不能作为 普通的数据类型使用

6, 如果方法返回值类型为void类型，可以省略 return ;

* + 方法调用方式：

有明确返回值类型的方法调用方式：

1，单独调用

2，输出调用

3，赋值调用

没有明确返回值类型的方法调用方式：

1. 单独调用
   * 方法重载：

方法重载，在同一个类中，出现了多个同名的方法，他们的参数列表不同 (参数列表的个数不同，参数列表的数据类型不同，参数列表的顺序不同)。

方法重载特点：

与方法的返回值类型无关，与方法的参数名无关，只看方法名与参数列表；

方法重载，是通过JVM来完成同名方法的调用的，通过参数列表来决定调用的是哪一个方法。

第6天 Java基础语法

今日内容介绍

* 自定义类
* ArrayList集合

# 引用数据类型（类）

## 引用数据类型分类

提到引用数据类型（类），其实我们对它并不陌生，如使用过的Scanner类、Random类。

我们可以把类的类型为两种：

* 第一种，Java为我们提供好的类，如Scanner类，Random类等，这些已存在的类中包含了很多的方法与属性，可供我们使用。
* 第二种，我们自己创建的类，按照类的定义标准，可以在类中包含多个方法与属性，来供我们使用。

这里我们主要介绍第二种情况的简单使用。

## 自定义数据类型概述

我们在Java中，将现实生活中的事物抽象成了代码。这时，我们可以使用自定义的数据类型（类）来描述（映射）现实生活中的事物。

类，它是引用数据类型，与之前学习的所有引用数据类型相同，自定义类也是一种数据类型。只是自定义类型并非Java为我们预先提供好的类型，而是我们自己定义的一种引用数据类型用来描述一个事物。

## 类的定义与使用

java代码映射成现实事物的过程就是定义类的过程。

我们就拿一部手机进行分析，它能用来做什么呢？它可以打电话，上网，聊微信等，这些就是手机所提供的功能，也就是方法；手机也有它的特征，如颜色、尺寸大小、品牌型号等，这些就是手机的特征，也就是属性。

目前，我们只关注类中的属性，类中的方法在面向对象部分再进行学习。

### 类的定义格式

* 类的定义格式

创建java文件，与类名相同

public class 类名{

数据类型 属性名称1；

数据类型 属性名称2；

…

}

通过类的定义格式，来进行手机类的描述，如下所示

**public** **class** Phone {

/\*

\* 属性

\*/

String brand;// 品牌型号

String color;// 颜色

**double** size; // 尺寸大小

}

上述代码，就是创建一个类的的过程，类的名称我们给起名为Phone，类中包含了三个属性（brand品牌型号、color颜色、size尺寸大小）。注意，类中定义的属性没有个数要求。

### 类的使用格式

Phone类定义好后，我们就可以使用这个类了，使用方式和使用引用数据类型Scanner类相似。格式如下：

导包：我们将所有的类放到同一个文件夹下，可以避免导包。

创建对象：数据类型 变量名 = new 数据类型();

调用方法：目前我们定义的自定义类不涉及方法，只是属性（自定义类中的方法部分在面向对象部分讲解）

访问属性：变量名.属性 (这是当前的方式，后期会采取调用方法的方式替代掉直接访问的方式来完成对属性的访问。)

当有了Phone数据类型的变量后，我们就可以使用Phone类中的属性了。对属性的访问我们来演示一下，如下所示：

public class Test {

public static void main(String[] args) {

//定义了一个Phone类型的变量p

Phone p = new Phone();

/\*

\* 通过p,使用Phone中的属性

\*/

//访问p中的brand品牌属性

p.brand = "苹果6s";

//访问p中的color颜色属性

p.color = "白色";

//访问p中的size尺寸大小属性

p.size = 5.5;

System.*out*.println("手机品牌为" + p.brand);

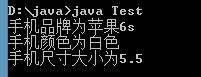
System.*out*.println("手机颜色为" + p.color);

System.*out*.println("手机尺寸大小为" + p.size);

}

}

运行结果如下所示

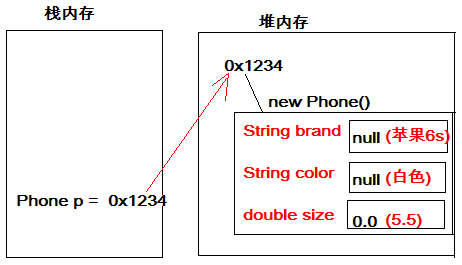


1. 运行结果

### 自定义类型注意事项与内存图

上述代码中，通过类Phone创建出来的变量p，它相当于我们生活中的盒子，里面包含了它能够使用的属性。

* 通过 p.属性名 就可以对属性进行操作
* 与引用类型数组类似，引用类型的自定义类型的变量，直接变量时，结果为对象地址值，这里可以通过内存图简单解释。



## 自定义类型练习

学习了引用数据类型（类）以后，我们就能够使用类描述任何东西了。看几个具体的描述，如下：

* 电饭锅，包含属性（品牌、容量大小、颜色等）
* 汽车，包含属性（品牌、排量、类型等）
* 学生，包含属性（姓名，年龄，性别等）

# ArrayList集合

在前面我们学习了数组，数组可以保存多个元素，但在某些情况下无法确定到底要保存多少个元素，此时数组将不再适用，因为数组的长度不可变。例如，要保存一个学校的学生，由于不停有新生来报道，同时也有学生毕业离开学校，这时学生的数目很难确定。为了保存这些数目不确定的元素，JDK中提供了一系列特殊的类，这些类可以存储任意类型的元素，并且长度可变，统称为集合。在这里，我们先介绍ArrayList集合，其他集合在后续课程中学习。

ArrayList集合是程序中最常见的一种集合，它属于引用数据类型（类）。在ArrayList内部封装了一个长度可变的数组，当存入的元素超过数组长度时，ArrayList会在内存中分配一个更大的数组来存储这些元素，因此可以将ArrayList集合看作一个长度可变的数组。

## 集合的创建

创建集合的常用格式在此说明一下：

导包：import java.util.ArrayList;

创建对象：与其他普通的引用数据类型创建方式完全相同，但是要指定容器中存储的数据类型：

ArrayList<要存储元素的数据类型> 变量名 = new ArrayList<要存储元素的数据类型>();

* 集合中存储的元素，只能为<>括号中指定的数据类型元素；
* “<要存储元素的数据类型>”中的数据类型必须是引用数据类型，不能是基本数据类型；

下面给出8种基本数据类型所对应的引用数据类型表示形式:

|  |  |
| --- | --- |
| **基本数据类型** | **对应的引用数据类型表示形式** |
| **byte** | Byte |
| **short** | Short |
| **Int** | Integer |
| **long** | Long |
| **float** | Float |
| **double** | Double |
| **char** | Character |
| **boolean** | Boolean |

我们通过举几个例子，来明确集合的创建方式：

* 存储String类型的元素

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

* 存储int类型的数据

ArrayList<Integer> list = new ArrayList<Integer>();

* 存储Phone类型的数据

ArrayList<Phone> list = new ArrayList<Phone>();

## 集合中常用方法

接下来，我们来学习下ArrayList集合提供的一些常用方法，如下表：

|  |  |
| --- | --- |
| **方法声明** | **功能描述** |
| **boolean add（Object obj）** | 将指定元素obj追加到集合的末尾 |
| **Object get（int index）** | 返回集合中指定位置上的元素 |
| **int size（）** | 返回集合中的元素个数 |

通过代码演示上述方法的使用。ArrayListDemo01.java

**import** java.util.ArrayList;

**public** **class** ArrayListDemo01 {

**public** **static** **void** main(String[] args) {

// 创建ArrayList集合

ArrayList<String> list = **new** ArrayList<String>();

// 向集合中添加元素

list.add("stu1");

list.add("stu2");

list.add("stu3");

list.add("stu4");

// 获取集合中元素的个数

System.*out*.println("集合的长度：" + list.size());

// 取出并打印指定位置的元素

System.*out*.println("第1个元素是：" + list.get(0));

System.*out*.println("第2个元素是：" + list.get(1));

System.*out*.println("第3个元素是：" + list.get(2));

System.*out*.println("第4个元素是：" + list.get(3));

}

}

强调一点，ArrayList集合相当于是一个长度可变的数组，所以访问集合中的元素也是采用索引方式访问，第一个元素存储在索引0的位置，第二个元素存储在索引1的位置，依次类推。

## 集合的遍历

通过集合遍历，得到集合中每个元素，这是集合中最常见的操作。集合的遍历与数组的遍历很像，都是通过索引的方式，集合遍历方式如下：ArrayListDemo02.java

1. **import** java.util.ArrayList;
2. **public** **class** ArrayListDemo02 {
3. **public** **static** **void** main(String[] args) {
4. //创建ArrayList集合
5. ArrayList<Integer> list = **new** ArrayList<Integer>();
6. //添加元素到集合
7. list.add(13);
8. list.add(15);
9. list.add(22);
10. list.add(29);
11. //遍历集合
12. **for** (**int** i = 0; i < list.size(); i++) {
13. //通过索引，获取到集合中每个元素
14. **int** n = list.get(i);
15. System.*out*.println(n);
16. }
17. }
18. }

上述代码中，第5行定义了一个可以存储int元素的集合；第7-10行，实现将int类型数值存储到集合中；第12-16行，实现遍历集合元素。这里要强调一点，get方法返回值的类型为集合中元素的类型。

## 集合中的常用方法补充

ArrayList集合提供的一些常用方法，如下表：

|  |  |
| --- | --- |
| **方法声明** | **功能描述** |
| **boolean add（int index, Object obj）** | 将指定元素obj插入到集合中指定的位置 |
| **Object remve（int index）** | 从集合中删除指定index处的元素，返回该元素 |
| **void clear（）** | 清空集合中所有元素 |
| **Object set（int index, Object obj）** | 用指定元素obj替代集合中指定位置上的元素 |

* boolean add（int index, Object obj）
  + 功能：在集合中指定index位置，添加新元素obj
  + 功能说明：假设集合list中有元素[“java”,“javaEE”]，当使用add(1，“javaWeb”)后，集合list中的元素为[“java”,“javaWeb”,“JavaEE”]。
* Object set（int index, Object obj）
  + 功能：用指定元素obj替代集合中指定index位置的元素
  + 功能说明：假设集合list中有元素[“java”,“javaEE”]，当使用set(0，“javaWeb”)后，集合list中的元素为[“javaWeb”,“JavaEE”]。
* Object remve（int index）
  + 功能：从集合中删除指定index处的元素，返回该元素
  + 功能说明：假设集合list中有元素[“java”,“javaEE”]，当使用remove(0)后，集合list中的元素为[“JavaEE”]，返回值为“java”。
* void clear（）
  + 功能：清空集合中所有元素
  + 功能说明：假设集合list中有元素[“java”,“javaEE”]，当使用clear()后，集合list中的元素为空[]。

# 随机点名器案例

## 案例介绍

随机点名器，即在全班同学中随机的找出一名同学，打印这名同学的个人信息。

此案例在我们昨天课程学习中，已经介绍，现在我们要做的是对原有的案例进行升级，使用新的技术来实现。

我们来完成随机点名器，它具备以下3个内容：

* 存储所有同学姓名
* 总览全班同学姓名
* 随机点名其中一人，打印到控制台

## 案例需求分析

全班同学中随机的找出一名同学，打印这名同学的个人信息。

我们对本案例进行分析，得出如下分析结果：

1.存储全班同学信息（姓名、年龄）

2.打印全班同学每一个人的信息（姓名、年龄）

3.在班级总人数范围内，随机产生一个随机数，查找该随机数所对应的同学信息（姓名、年龄）

随机点名器明确地分为了三个功能。如果将多个独立功能的代码写到一起，则代码相对冗长，我们可以针对不同的功能可以将其封装到一个方法中，将完整独立的功能分离出来。

而在存储同学姓名时，如果对每一个同学都定义一个变量进行姓名存储，则会出现过多孤立的变量，很难一次性将全部数据持有。此时，我们采用ArrayList集合来解决多个学生信息的存储问题。

## 实现代码步骤

每名学生都拥有多项个人信息，为了方便管理每个人的信息，我们对学生信息进行封装，编写Student.java文件

/\*\*

\* 学生信息类

\*/

**public** **class** Student {

String name; //姓名

**int** age; //年龄

}

上述代码中，对学生信息（姓名、年龄）进行了封装。这样做的好处在于，以后只要找到这名学生，就能够知道他的每项个人信息了。

接下来我们编写CallName.java文件，完成程序的编写。

* main方法中调用三个独立方法

**public** **static** **void** main(String[] args) {

ArrayList<Student> list = **new** ArrayList<Student>(); //1.1创建一个可以存储多个同学名字的容器

/\*

\* 1.存储全班同学信息

\*/

addStudent(list);

/\*

\* 2.打印全班同学每一个人的信息（姓名、年龄）

\*/

printStudent(list);

/\*

\* 3.随机对学生点名，打印学生信息

\*/

randomStudent(list);

}

* 存储所有学生的个人信息

/\*\*

\* 1.存储全班同学名字

\*/

**public** **static** **void** addStudent(ArrayList<Student> list) {

//键盘输入多个同学名字存储到容器中

Scanner sc = **new** Scanner(System.*in*);

**for** (**int** i = 0; i < 3; i++) {

//创建学生

Student s = **new** Student();

System.*out*.println("存储第"+i+"个学生姓名：");

s.name = sc.next();

System.*out*.println("存储第"+i+"个学生年龄：");

s.age = sc.nextInt();

//添加学生到集合

list.add(s);

}

}

上述方法中，方法参数list中用来表示已存储所有学生。通过Scanner，完成新学生信息（姓名，年龄）的录入，并将学生添加到集合中。

* 打印全班同学每一个人的信息

/\*\*

\* 2.打印全班同学每一个人的信息（姓名、年龄）

\*/

**public** **static** **void** printStudent (ArrayList<Student> list) {

**for** (**int** i = 0; i < list.size(); i++) {

Student s = list.get(i);

System.*out*.println("姓名："+s.name +",年龄："+s.age);

}

}

上述方法中，方法参数list中用来表示已存储所有学生。通过遍历集合中的每个元素，得到每个同学信息，并输出打印。

* 随机对学生点名，打印学生信息

/\*\*

\* 3.随机对学生点名，打印学生信息

\*/

**public** **static** **void** randomStudent (ArrayList<Student> list) {

//在班级总人数范围内，随机产生一个随机数

**int** index = **new** Random().nextInt(list.size());

//在容器（ArrayList集合）中，查找该随机数所对应的同学信息（姓名、年龄）

Student s = list.get(index);

System.*out*.println("被随机点名的同学："+s.name + "，年龄:" + s.age);

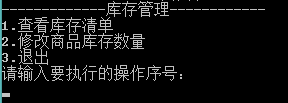
}

上述方法中，通过随机数类Random产生一个从0到集合长度的随机索引。使用该索引获取ArrayList集合中对应的值，便得到了全班同学的随机学生信息并打印。

# 库存管理案例

## 案例介绍

现在，我们将原有的库存管理案例，采用更好的集合方式实现。



将对下列功能进行方法封装：

* 打印库存清单功能
* 库存商品数量修改功能
* 退出程序功能

## 案例需求分析

管理员能够进行的操作有3项（查看、修改、退出），我们可以采用（switch）菜单的方式来完成。

-------------库存管理------------

1.查看库存清单

2.修改商品库存数量

3.退出

请输入要执行的操作序号：

每一项功能操作，我们采用方法进行封装，这样，可使程序的可读性增强。

选择“1.查看库存清单”功能，则控制台打印库存清单

选择“2.修改商品库存数量”功能，则对每种商品库存数进行更新

选择“3.退出”功能，则退出库存管理，程序结束

## 实现代码步骤

每种库存商品都拥有多项商品信息，为了方便管理每种商品的信息，我们对商品信息进行封装，编写Goods.java文件

/\*

\* 库存商品类

\*/

**public** **class** Goods {

String brand; // 商品品牌型号

**double** size; // 商品尺寸大小

**double** price; // 商品价格

**int** count; // 商品库存个数

}

上述代码中，对商品信息（品牌、尺寸、价格、库存数）进行了封装。这样做的好处在于，以后只要找到这个商品，就能够知道该商品的每项信息了。

编写Demo库存管理.java，完成如下功能：

* 功能菜单

/\*\*

\* 库存管理功能菜单

\* **@return** 管理员键盘输入的功能操作序号

\*/

**public** **static** **int** chooseFunction() {

System.*out*.println("-------------库存管理------------");

System.*out*.println("1.查看库存清单");

System.*out*.println("2.修改商品库存数量");

System.*out*.println("3.退出");

System.*out*.println("请输入要执行的操作序号：");

//接收键盘输入的功能选项序号

Scanner sc = **new** Scanner(System.*in*);

**int** choose = sc.nextInt();

**return** choose;

}

上述方法用来完成库存管理功能菜单的显示、接收管理员选择的功能操作序号。这是完成了案例的第一步。接下来完成“查看、修改、退出”这三项功能。

* 编写main主方法，调用库存管理功能菜单方法，与“查看、修改、退出”这三个方法。

**public** **static** **void** main(String[] args) {

//记录库存商品信息

ArrayList<Goods> list = **new** ArrayList<Goods>();

//添加商品到库存

*addStore*(list);

//通过while循环模拟管理员进行功能重复选择操作

while (true) {

//打印功能菜单操作,接收键盘输入的功能选项序号

**int** choose = *chooseFunction*();

//执行序号对应的功能

**switch** (choose) {

**case** 1://查看库存清单

*printStore*(list);

**break**;

**case** 2://修改商品库存数量

*update*(list);

**break**;

**case** 3://退出

*exit*();

**return**;

default:

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

System.*out*.println("功能选择有误，请输入正确的功能序号!");

**break**;

}

}

}

在主方法中，创建了ArrayList集合，用来存储库存商品信息，通过接收到的功能选项序号，进行switch语句判断后，调用对应的功能方法。

* 查看库存清单功能

/\*\*

\* 查看库存清单

\*/

**public** **static** **void** printStore(ArrayList<Goods> list) {

//统计总库存个数、统计库存总金额

**int** totalCount = 0;

**double** totalMoney = 0.0;

//列表顶部

System.*out*.println("---------------------------查看库存清单--------------------------");

System.*out*.println("品牌型号 尺寸 价格 库存数");

//列表中部

**for** (**int** i = 0; i < list.size(); i++) {

Goods item = list.get(i);

System.*out*.println(item.brand+" "+item.size+" "+item.price+" "+ item.count);

//统计总库存个数、统计库存总金额

totalCount += item.count;

totalMoney += item.count \* item.price;

}

//列表底部

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

System.*out*.println("总库存数："+totalCount);

System.*out*.println("库存商品总金额："+totalMoney);

}

上述方法用来完成打印库存清单功能，参数list表示库存商品相关信息集合。

* 修改商品库存数量功能

/\*\*

\* 修改商品库存数量

\*/

**public** **static** **void** update(ArrayList<Goods> list){

System.*out*.println("------------修改商品库存数量-----------");

**for** (**int** i = 0; i < list.size(); i++) {

Goods item = list.get(i);

System.*out*.println("请输入"+ item.brand +"商品库存数量");

item.count = **new** Scanner(System.*in*).nextInt();

list.set(i, item);

}

}

上述方法用来完成修改商品库存数量功能，参数list表示库存商品相关信息集合。

* 退出功能

/\*\*

\* 退出

\*/

public static void exit(){

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

System.*out*.println("您已退出系统");

}

上述方法用来完成退出程序的功能

# 总结

## 知识点总结

* 引用数据类型（类）
  + 类的类型为两种：
    - 第一种，Java为我们提供好的类，如Scanner类，Scanner类等，这些已存在的类中包含了很多的方法与属性，可供我们使用。
    - 第二种，我们自己创建的类，按照类的定义标准，可以在类中包含多个方法与属性，来供我们使用。
  + 创建类的格式

public class 类名 {

//可以定义属性

//也可以定义方法

}

* + 使用类的格式：

类名 变量名 = new 类名();

* + 使用类中的属性与方法格式

使用属性： 变量名.属性

使用方法： 变量名.方法()

* ArrayList集合
  + 它属于引用数据类型（类）。我们可以看作一个长度可变的数组。
  + 创建集合的方式

ArrayList<要存储元素的数据类型> 变量名 = new ArrayList<要存储元素的数据类型>();

* + 集合中的常用方法
    - boolean add（Object obj）
    - Object get（int index）
    - int size（）
    - boolean add（int index, Object obj）
    - Object set（int index, Object obj）
    - Object remve（int index）
    - void clear（）

第7天 Java基础语法

今日内容介绍

* 循环练习
* 数组方法练习

# 循环练习

## 编写程序求 1+3+5+7+……+99 的和值。

题目分析：

通过观察发现，本题目要实现的奇数（范围1-100之间）的累加和。

* 为了记录累加和的值，我们需要定义一个存储累加和的变量
* 我们要获取到1-100范围内的数
* 判断当前数是否为奇数，是奇数，完成累加和操作
* 累加完毕后，最终显示下累加和的值

解题步骤：

1. 定义一个用来记录累加和的变量
2. 使用for循环语句，完成1-100之间每个数的获取
3. 使用if条件语句，判断当前数是否是奇数，是奇数，进行累加和操作
4. 使用输出语句，打印累加和变量的值

代码如下：

public class Test01 {

public static void main(String[] args) {

int sum = 0;

for (int i = 0; i < 100; i++) {

if (i%2==1) {

sum += i;

}

}

System.*out*.println("累加和的值 " + sum);

}

}

100-999

## 输出所有的水仙花数，所谓水仙花数是指一个数3位数，其每位数字立方和等于其本身，如153 = 1\*1\*1 + 3\*3\*3 + 5\*5\*5

题目分析：

通过观察发现，本题目要实现打印符合要求的数字（即水仙花数）。

1. 明确什么样的数就是水仙花数。水仙花数是指一个3位数（100-999之间），其每位数字立方之和等于该3位数本身。如153 = 1\*1\*1 + 3\*3\*3 + 5\*5\*5，

即 3位数本身 = 百位数立方 + 十位数立方 + 个位数立方;

1. 获取水仙花范围内的所有3位数（100-999之间的每个3位数）
2. 判断该3位数是否满足水仙花数，满足，打印该3位数

解题步骤：

1. 使用for循环，得到100-999之间的每个3位数
2. 获取3位数中百位数字、十位数字、个位数字
3. 使用if条件语句，判断该3位数是否满足水仙花数，满足，使用输出语句，打印该3位数

代码如下：

public class Test02 {

public static void main(String[] args) {

for (int i = 100; i < 1000; i++) {

int bai = i/100%10;

int shi = i/10%10;

int ge = i%10;

if (i == bai\*bai\*bai + shi\*shi\*shi + ge\*ge\*ge) {

System.*out*.println(i);

}

}

}

}

## ASCII编码表

American Standard Code for Information Interchange，美国标准信息交换代码.

在计算机中，所有的数据在存储和运算时都要使用二进制数表示, a、b、c、d这样的52个字母（包括大写）、以及0、1等数字还有一些常用的符号, 在计算机中存储时也要使用二进制数来表示，而具体用哪些二进制数字表示哪个符号，当然每个人都可以约定自己的一套（这就叫编码），而大家如果要想互相通信而不造成混乱，那么大家就必须使用相同的编码规则，于是美国有关的标准化组织就出台了ASCII编码，统一规定了上述常用符号用哪些二进制数来表示。



数字0-9对应ASCII编码十进制为48-57, 字母a-z对应ASCII编码十进制为97-122，字母A-Z对应ASCII编码十进制为65-90

## 利用for循环打印ABCDEFG...XYZ，26个大写字母与26个小写字母

题目分析：

通过观察发现，本题目要实现打印26个大写字母、26个小写字母

1. 一共26个大小写字母，那么，可以考虑循环26次。在每次循环中，完成指定字母的大小写打印
2. 找出ABCDEFG…XYZ这些字母之间的变化规律

通过ASCII表发现，后面的字母比它前面的字母，ASCII值大1

下一个字母 = 上一个字母 + 1

如： A B C D

65 66 67 68

3. 在每次循环中打印上一个字母大小写，并指定下一个字母

解题步骤：

1. 定义初始化大写变量，值为’A’； 初始化小写变量，值为’a’
2. 使用for循环，进行26次循环
3. 在每次循环中，打印大写字母、小写字母。

每次打印完成后，更新大写字母值、小写字母值

代码如下：

public class Test04 {

public static void main(String[] args) {

char da = 'A';

char xiao = 'a';

for (int i = 0; i < 26; i++) {

System.*out*.println("大写字母 "+da+" ,小写字母 "+xiao);

da++; //更新大写字母值

xiao++; //更新小写字母值

}

}

}

## 利用for循环打印 9\*9 表?

如：

1\*1=1

1\*2=2 2\*2=4

1\*3=3 2\*3=6 3\*3=9

题目分析：

通过观察发现，如果把1\*1=1这样的内容 看做一颗\*的话，那么打印结果就成了如下效果：

\*

\*\*

\*\*\*

…

这样，就是打印9行星，每行打印星的个数与当前行数相等。

再观察“1\*3=3 2\*3=6 3\*3=9”得出它们如下的变化规律：

每行第n次 +"\*"+ 行号 +"="+ 每行第n次 \* 行号

如: 1 +"\*"+ 2 +"="+ 1\*2; // 相当于1\*2=2

2 +"\*"+ 2 +"="+ 2\*2; // 相当于2\*2=4

解题步骤：

1. 定义一个外层for循环，初始值从1开始，循环9次。用来控制打印的行数
2. 在外层for循环内部，定义一个for循环，初始值从1开始，循环次数与当前行数相等。用来完成每行打印指定次数的乘法公式 如1\*1=1
3. 在内层for循环中，完成每行指定次数的乘法公式打印 如1\*1=1

System.*out*.print(k +"\*"+ j +"="+ j\*k +"\t");

// 变量k代表：每行中的第n次

// 变量j代表：行号

1. 在外循环中，当每行指定次数的乘法公式打印完毕后，通过System.out.println()切换到下一行。这样，再次打印乘法公式时，就在下一行输出打印了

代码如下：

public class Test05 {

public static void main(String[] args) {

for (int j = 1; j < 10; j++) {

for (int k = 1; k <= j; k++) {

System.*out*.print(k +"\*"+ j +"="+ j\*k +"\t");

}

System.*out*.println();

}

}

}

# 数组方法练习

## 定义打印数组元素方法,按照给定的格式打印[11, 33, 44, 22, 55]

题目分析：

通过观察发现，本题目要实现按照指定格式，打印数组元素操作

1. 通过循环，我们可以完成数组中元素的获取，数组名[索引]
2. 观察发现，每个数组元素之间加入了一个逗号”,”进行分隔；并且，整个数组的前后有一对中括号”[]”包裹数组所有元素。

解题步骤：

1. 使用输出语句完成打印 左边的中括号”[”
2. 使用循环，输出数组元素值。输出元素值分为两种情况，如下：
   1. 最后一个数组元素，加上一个右边的中括号”]”
   2. 非最后一个数组元素，加上一个逗号”,”

代码如下：

//打印数组

**public** **static** **void** printArray(**int**[] arr) {

System.***out***.print("[");

**for** (**int** i = 0; i < arr.length; i++) {

**if** (i == arr.length - 1) {

System.***out***.println(arr[i]+"]");

} **else** {

System.***out***.print(arr[i]+", ");

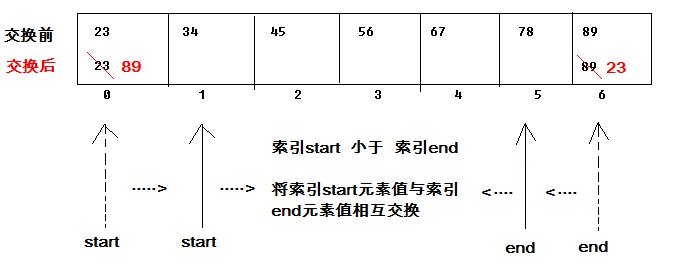
}

}

}

## 数组元素逆序

图解：



题目分析：

通过观察发现，本题目要实现原数组元素倒序存放操作。即原数组存储元素为{11,22,33,44}，逆序后为原数组存储元素变为{44,33,22,11}。

1. 通过图解发现，想完成数组元素逆序，其实就是把数组中索引为start与end的元素进行互换。
2. 每次互换后，start索引位置后移，end索引位置前移，再进行互换
3. 直到start位置超越了end位置，互换结束，此时，数组元素逆序完成。

解题步骤：

1. 定义两个索引变量start值为0，变量end值为数组长度减去1（即数组最后一个元素索引）
2. 使用循环，完成数组索引start位置元素与end位置元素值互换。
3. 在循环换过程中，每次互换结束后，start位置后移1，end位置前移1
4. 在循环换过程中，最先判断start位置是否超越了end位置，若已超越，则跳出循环

代码如下：

//数组元素逆序

**public** **static** **void** receive(**int**[] arr){

**for** (**int** start = 0, end = arr.length-1; start < end; start++,end--) {

**int** temp = arr[start];

arr[start] = arr[end];

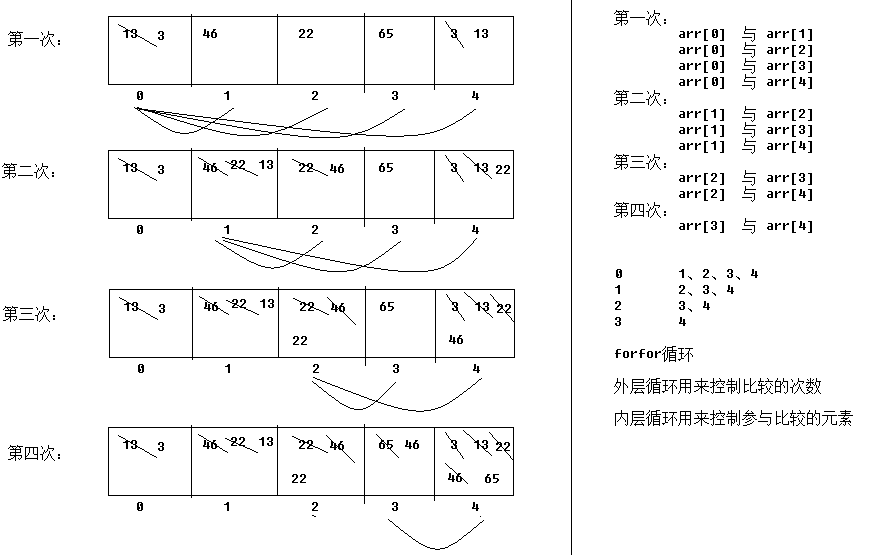
arr[end] = temp;

}

}

## 数组元素选择排序

图解：



题目分析：

通过观察发现，本题目要实现把数组元素{13,46,22,65,3}进行排序

1. 提到数组排序，就要进行元素值大小的比较，通过上图发现，我们想完成排序要经过若干次的比较才能够完成。
2. 上图中用每圈要比较的第一个元素与该元素后面的数组元素依次比较到数组的最后一个元素，把小的值放在第一个数组元素中，数组循环一圈后，则把最小元素值互换到了第一个元素中。
3. 数组再循环一圈后，把第二小的元素值互换到了第二个元素中。按照这种方式，数组循环多圈以后，就完成了数组元素的排序。这种排序方式我们称为选择排序。

解题步骤：

1. 使用for循环（外层循环），指定数组要循环的圈数（通过图解可知，数组循环的圈数为数组长度 - 1）
2. 在每一圈中，通过for循环（内层循环）完成数组要比较的第一个元素与该元素后面的数组元素依次比较到数组的最后一个元素，把小的值放在第一个数组元素中
3. 在每一圈中，要参与比较的第一个元素由第几圈循环来决定。如上图所示
   1. 进行第一圈元素比较时，要比较的第一个元素为数组第一个元素，即索引为0的元素
   2. 进行第二圈元素比较时，要比较的第一个元素为数组第二个元素，即索引为1的元素
   3. 依次类推，得出结论：进行第n圈元素比较时，要比较的第一个元素为数组第n个元素，即数组索引为n-1的元素

代码如下：

//选择排序

**public** **static** **void** selectSort(**int**[] arr) {

//功能

//外层循环用来控制数组循环的圈数

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

//内层循环用来完成元素值比较，把小的元素值互换到要比较的第一个元素中

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

**if** (arr[i] > arr[j]) {

**int** temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

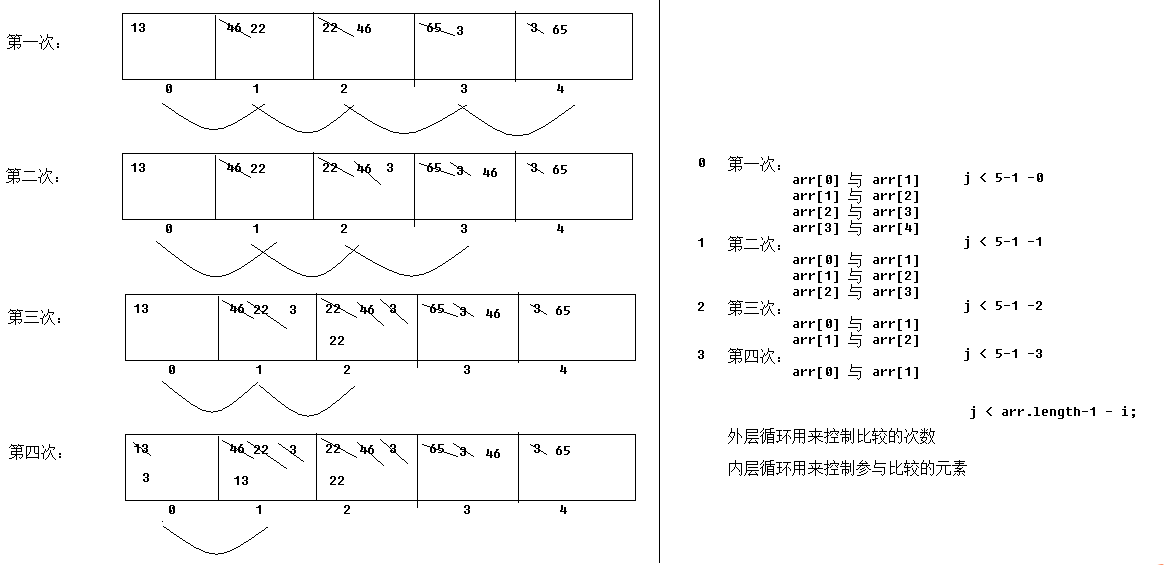
}

}

}

## 数组元素冒泡排序

图解：数组元素{13,46,22,65,3}



题目分析：

通过观察发现，本题目要实现把数组元素{13,46,22,65,3}进行排序

1. 提到数组排序，就要进行元素值大小的比较，通过上图发现，我们想完成排序要经过若干次的比较才能够完成。
2. 上图中相邻的元素值依次比较，把大的值放后面的元素中，数组循环一圈后，则把最大元素值互换到了最后一个元素中。数组再循环一圈后，把第二大的元素值互换到了倒数第二个元素中。按照这种方式，数组循环多圈以后，就完成了数组元素的排序。这种排序方式我们称为冒泡排序。

解题步骤：

1. 使用for循环（外层循环），指定数组要循环的圈数（通过图解可知，数组循环的圈数为数组长度 - 1）
2. 在每一圈中，通过for循环（内层循环）完成相邻的元素值依次比较，把大的值放后面的元素中
3. 每圈内层循环的次数，由第几圈循环来决定。如上图所示
   1. 进行第一圈元素比较时，内层循环次数为数组长度 - 1
   2. 进行第二圈元素比较时，内层循环次数为数组长度 - 2
   3. 依次类推，得出结论：进行第n圈元素比较时，内层循环次数为数组长度 - n

代码如下：

//冒泡排序

**public** **static** **void** bubbleSort(**int**[] arr) {

//功能

//外层循环用来控制数组循环的圈数

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

//j < arr.length-1 为了避免角标越界

//j < arr.length-1-i 为了比较效率,避免重复比较

//内层循环用来完成元素值比较，把大的元素值互换到后面

**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;

}

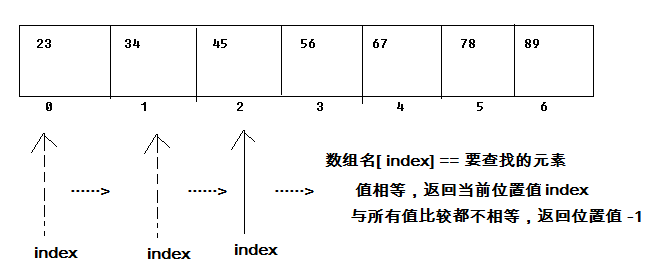
}

}

}

## 数组元素普通查找

图解：



题目分析：

通过观察发现，本题目要实现查找指定数值第一次在数组中存储的位置（索引），返回该位置（索引）。

1. 我们可以通过遍历数组，得到每个数组元素的值
2. 在遍历数组过程中，使用当前数组元素值与要查找的数值进行对比
   1. 数值相等，返回当前数组元素值的索引
   2. 整个循环结束后，比对结果数值没有相等的情况，说明该数组中没有存储要查找的数值，此时，返回一个索引值-1，来表示没有查询到对应的位置。（使用 -1来表示没有查询到，是因为数组的索引没有负数）

解题步骤：

1. 使用for循环，遍历数组，得到每个数组元素值
2. 在每次循环中，使用if条件语句进行当前数组元素值与要查找的数值进行对比，若比较结果相等，直接返回当前数组元素的索引值
3. 若整个循环结束后，比对结果数值没有相等的情况，说明该数组中没有存储要查找的数值，此时，返回一个索引值-1

代码如下：

//普通查找

**public** **static** **int** getArrayIndex(**int**[] arr, **int** number) {

//把数组中的元素依次与指定的数值 进行比较

**for** (**int** i = 0; i < arr.length; i++) {

**if** (arr[i] == number) {

//找到了

return i;

}

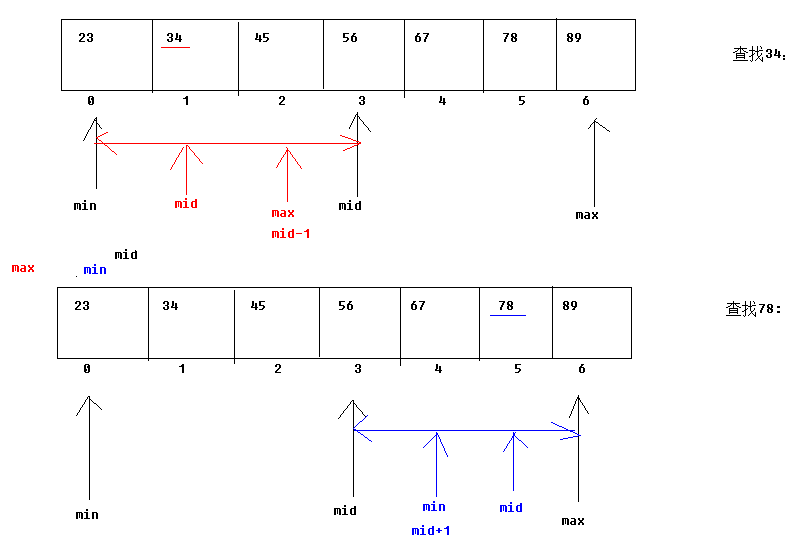
}

**return** -1;

}

## 数组元素二分查找（折半查找）

图解：



题目分析：

通过观察发现，本题目要实现查找指定数值在**元素有序的数组**中存储的位置（索引），返回该位置（索引）。

1. 我们使用数组最中间位置的元素值与要查找的指定数值进行比较，若相等，返回中间元素值的索引
2. 最中间位置的元素值与要查找的指定数值进行比较，若不相等，则根据比较的结果，缩小查询范围为上次数组查询范围的一半；

再根据新的查询范围，更新最中间元素位置，然后使用中间元素值与要查找的指定数值进行比较

* + 比较结果相等，返回中间元素值的索引
  + 比较结果不相等，继续缩小查询范围为上次数组查询范围的一半，更新最中间元素位置，继续比较，依次类推。

1. 当查询范围缩小到小于0个元素时，则指定数值没有查询到，返回索引值-1。

解题步骤：

1. 定义3个用来记录索引值的变量，变量min记录当前范围最小索引值，初始值为0；变量max记录当前范围最大索引值，初始值为数组长度-1；变量mid记录当前当前范围最中间元素的索引值，初始值为(min+max) / 2
2. 使用循环，判断当前范围下，最中间元素值与指定查找的数值是否相等
   * 若相等，结束循环，返回当前范围最中间元素的索引值mid
   * 若不相等，根据比较结果，缩小查询范围为上一次查询范围的一般
     + 中间元素值 比 要查询的数值大，说明要查询的数值在当前范围的最小索引位置与中间索引位置之间，此时，更新查询范围为:

范围最大索引值 = 上一次中间索引位置 -1；

* + - 中间元素值 比 要查询的数值小，说明要查询的数值在当前范围的最大索引位置与中间索引位置之间，此时，更新查询范围为:

范围最小索引值 = 上一次中间索引位置 +1；

* + - 在新的查询范围中，更新中间元素值的位置，再次使用最中间元素值与指定查找的数值是否相等。

中间索引值 = (范围最小索引值 +范围最大索引值) / 2;

1. 每次查询范围缩小一半后，使用if语句判断，查询范围是否小于0个元素，若小于0个元素，则说明指定数值没有查询到，返回索引值-1。

代码如下：

//二分查找法(折半查找法)

**public** **static** **int** halfSearch(**int**[] arr, **int** number) {

//定义3个变量，用来记录min, min, mid的位置

**int** min = 0;

**int** max = arr.length-1;

**int** mid = 0;

**while** (min <= max) {

mid = (min+max)/2;

//没找了， 更新范围，继续比较

//更新范围

**if** (arr[mid] > number) {

//在左边

max = mid-1;

} **else** **if**(arr[i] < number){

//在右边

min = mid+1;

}

**else**{

**return** mid ;

}

**return** -1;

}

第8天 Java基础语法

今日内容介绍

* Eclipse开发工具
* 超市库存管理系统

# Eclipse开发工具

Eclipse是功能强大Java集成开发工具。它可以极大地提升我们的开发效率。可以自动编译，检查错误。在公司中，使用的就是Eclipse进行开发。

## Eclipse的下载、安装、卸载

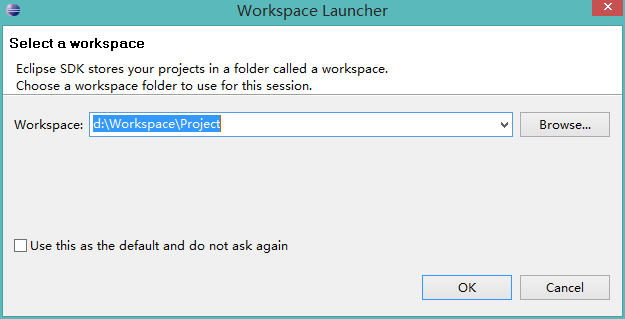
* 下载 <http://www.eclipse.org>
* 安装 (只需要解压后就能使用)
* 卸载（只需要将文件夹删除就可以了）
* 注意： 开发软件的安装目录中，尽量不要出现空格与中文

## Eclipse的使用

在当前阶段，我们只需要完成最基本的Java文件编写，其他功能跟随日常教学慢慢渗透学习。

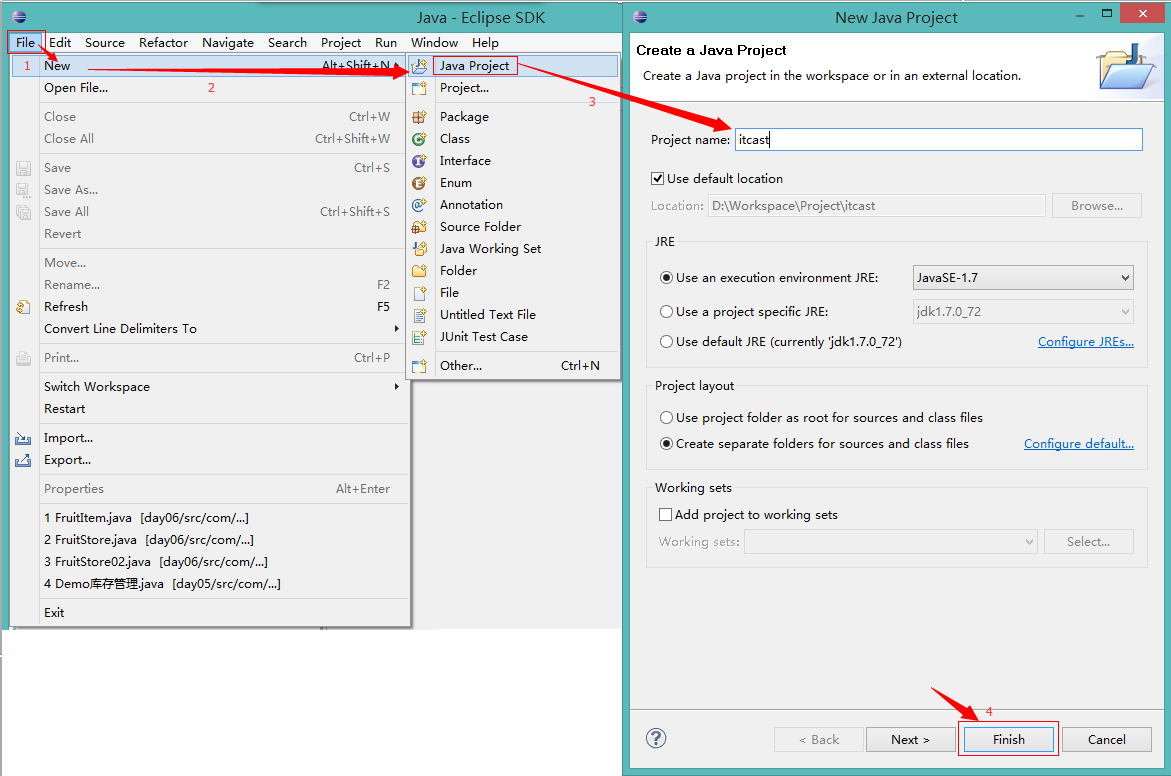
使用步骤：

* 1.开启，选择工作空间，如下图
  + 工作空间：指的是将来编写的java源文件存放的位置
  + 下图中的复选框“□”，如果选中，表示使用当前工作空间为Eclipse的默认工作空间，以后启动Eclipse时，不在提示选择工作空间。



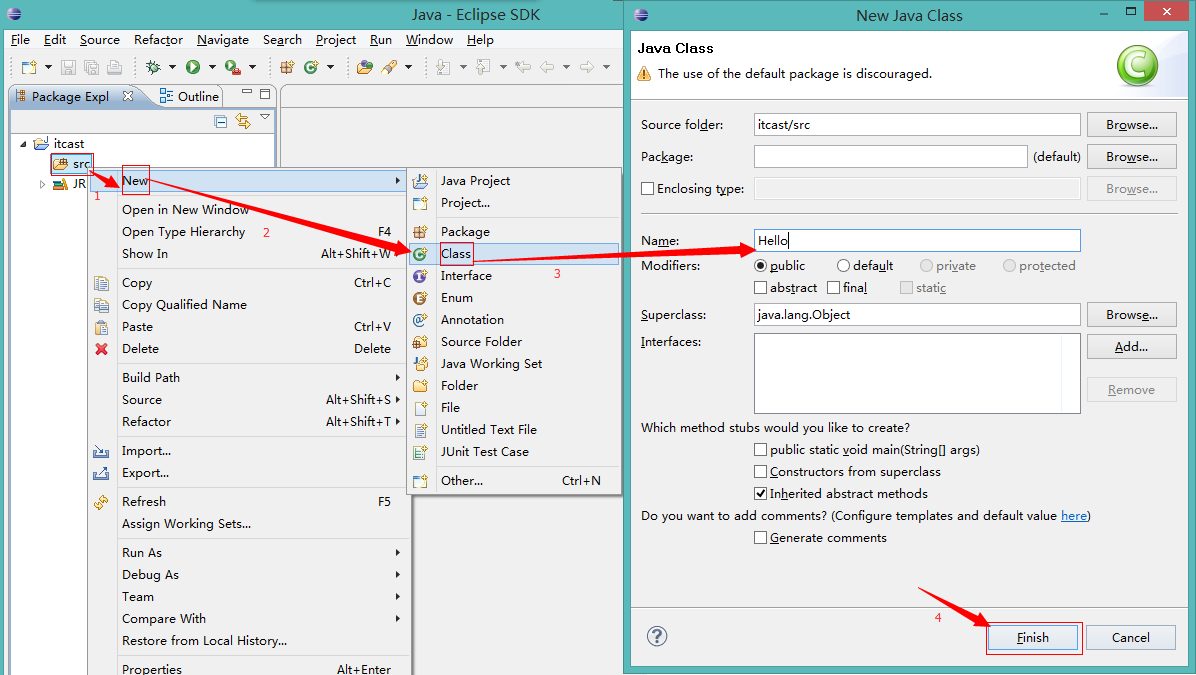
1. 选择工作空间

* 2.在左侧空白区域右键 或者 选中File菜单栏， 选择new--> 创建java项目编写项目名称，点击Finish 完成。如下图



1. 创建java项目

* 3.展开项目，选中源包SRC后，在项目的src目录下右键创建类，指定类的名字 Hello，创建main方法并编写源代码System.out.println("HelloWorld");

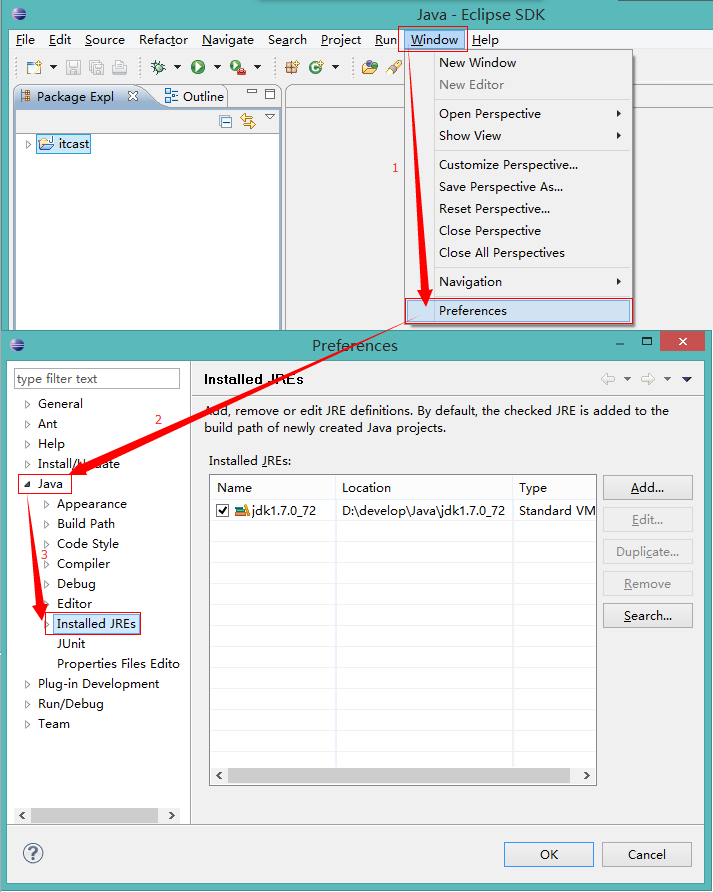


1. 创建类

* 4.编译与执行
  + 编译：自动编译，当java代码保存的时候，自动 编译class文件
  + 执行：
    - 方式1：点击菜单工具栏中的 绿色带有三角形的 run按钮 运行
    - 方式2：点击菜单栏中Run， 点击Run 运行 快捷键是 ctrl+F11
    - 方式3：选中要运行的java文件，或者在编写代码的空白区域，右键选择 Run As --> 运行java程序

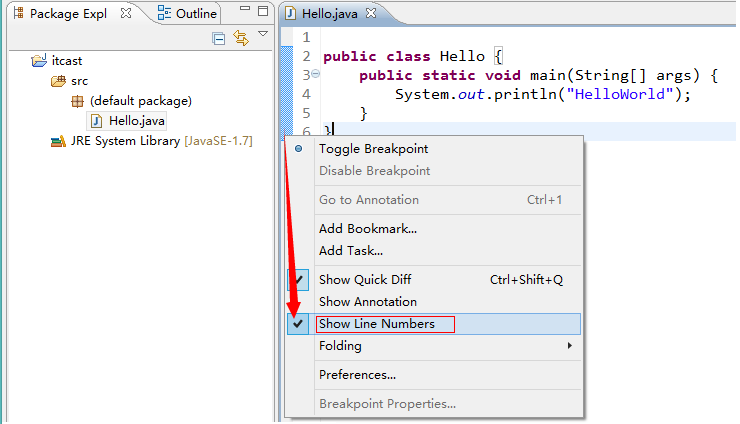
## 针对Eclipse空间的简单配置

* 程序的编译环境和运行环境
  + 编译环境：Window -- Preferences – Java - Compiler
  + 运行环境：Window -- Preferences – Java - Installed JREs



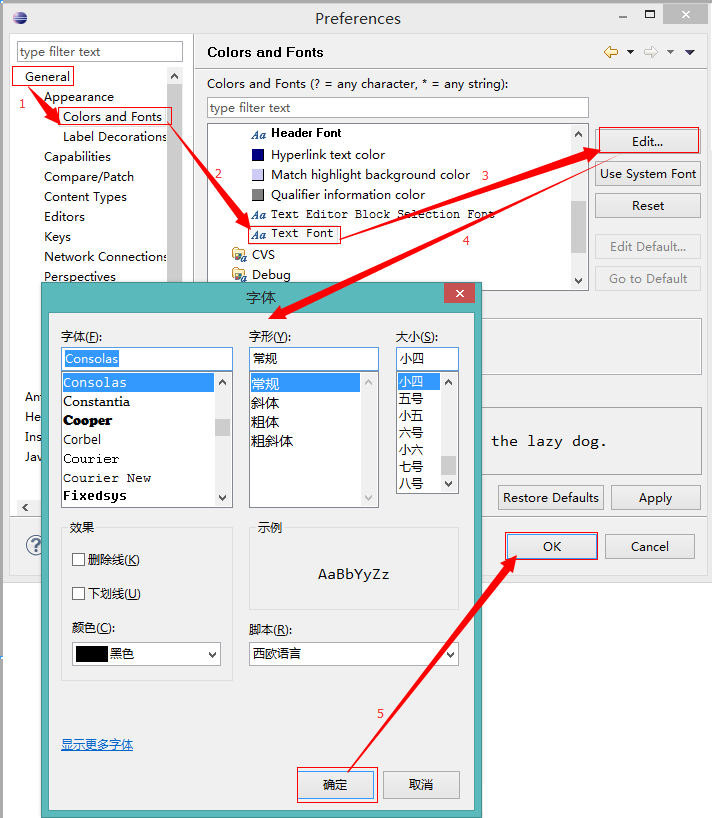
1. 编译环境与运行环境配置

* 如何显示行号与隐藏行号
  + 显示：在代码区域的左边空白区域，右键 -- Show Line Numbers
  + 隐藏：将上面的操作再做一遍



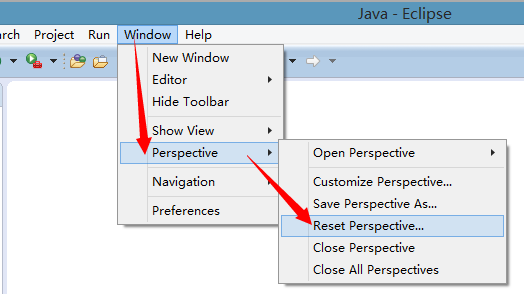
1. 行号显示与隐藏

* 如何更改字体大小与颜色
  + 修改java代码的字体大小与颜色
    - Window -- Preferences -- General -- Appearance -- Colors and Fonts -- Java -- Java Editor Text Font选中后，点击Edit，进行修改
  + 修改其他字体大小与颜色
    - Window -- Preferences -- General -- Appearance -- Colors and Fonts -- Bacic -- Text Font 选中后 点击Edit ,进行修改

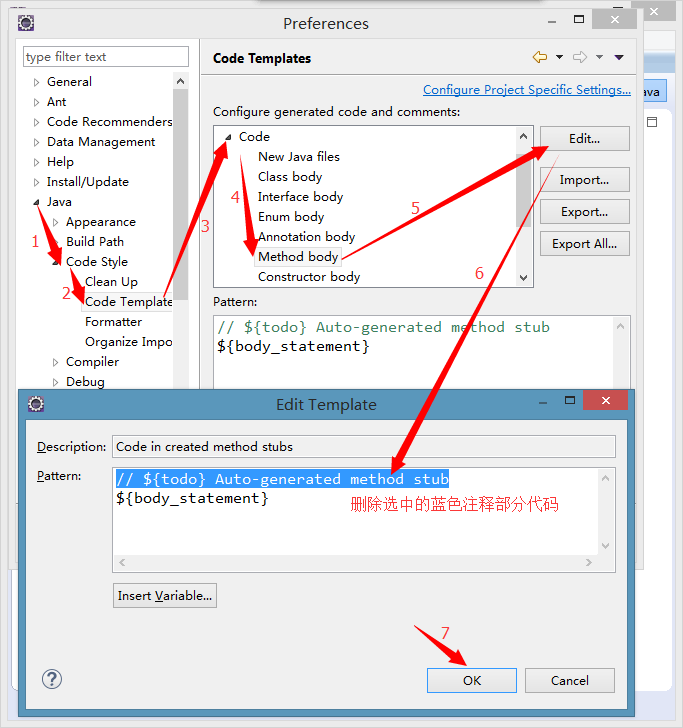


1. 设置字体

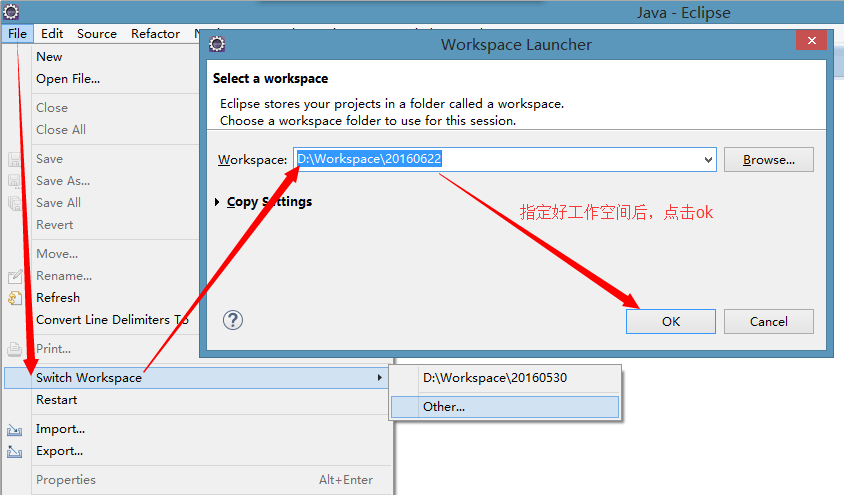
* 显示的窗口乱了，还原默认显示模式
  + Window – Perspective -- Reset Prespective



* 如何去掉默认注释
  + Window -- Preferences -- Java -- Code Style -- Code Templates -- Comments – Methods，点击Edit ，将注释部分删除 (不建议删除)
  + Window -- Preferences -- Java -- Code Style -- Code Templates -- Code -- Method body，点击Edit ，将注释部分删除



* 切换工作空间
  + File – Switch Workspace – 指定工作空间 – ok

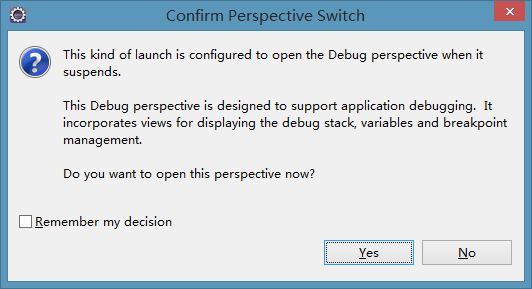


## 快捷键

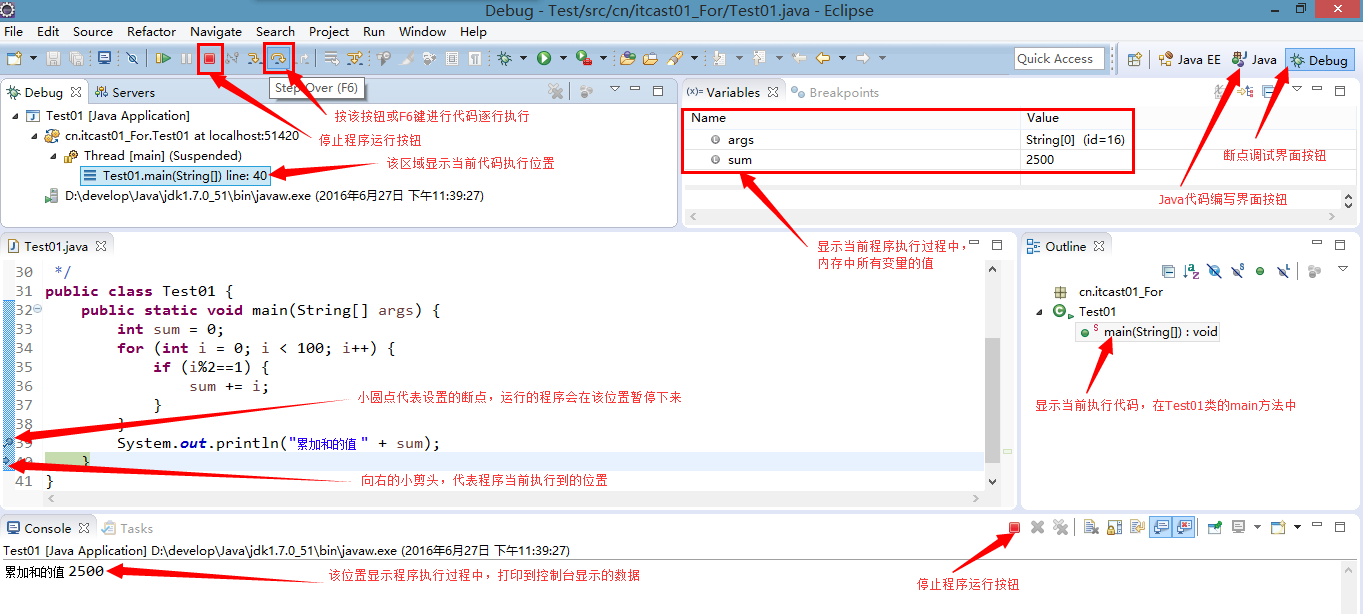
* 内容辅助键 Alt+/
  + 自动补齐main方法 main 然后 Alt+/
  + 自动补齐输出语句 syso 然后 Alt+/
* 格式化Ctrl+Shift+f
  + 代码区域右键 -- Source – Format
* 自动导包： Ctrl+Shift+o
  + 如果当前类在多个包中都存在，这时候，使用Ctrl+shift+o,进行选择一个包导入即可。
* 注释
  + 单行注释
    - 加注释： 先选中需要注释的内容，然后 Ctrl+/
    - 取消注释：先选中需要取消注释的内容， 然后 Ctrl+/
  + 多行注释
    - 加注释： 先选中需要注释的内容，然后 Ctrl+Shift+/
    - 取消注释：先选中需要取消注释的内容， 然后 Ctrl+Shift+\

## 断点调试

* 断点： 就是一个标记
* 如何设置断点
  + 代码区域左边的空白位置 双击即可
* 在哪里加
  + 想看哪个语句的执行结果就在该语句前面加
* 怎么运行断点的程序
  + 代码区域右键 -- Debug as -- Java Application
  + 显示一个页面，提示是否进入调试页面？ 是否记住当前操作？是(Yes)是（勾选复选框）



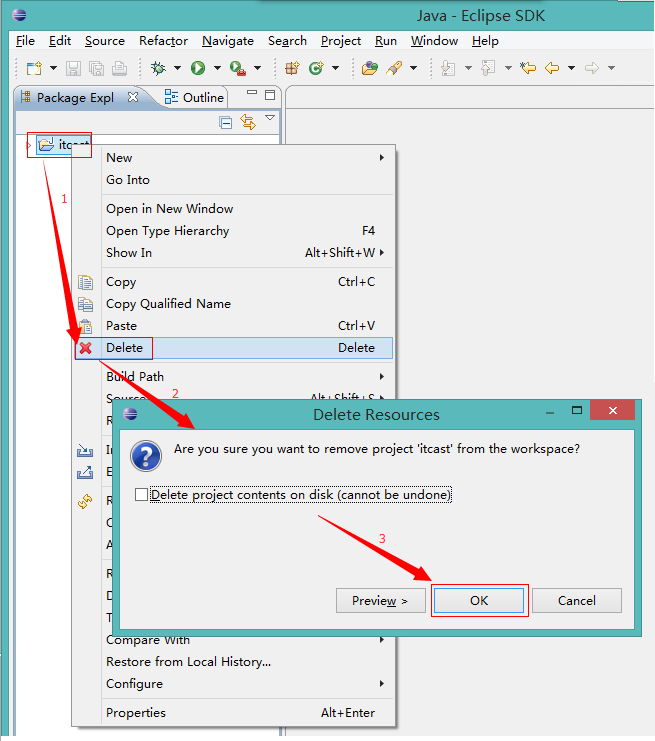
* 弹出了断点页面



* + 点哪里
    - 点击 工具栏中的Step Over (执行下一行)
    - 快捷键F6 看到每一步程序的执行过程
  + 看哪里
    - 看代码区域： 对照看程序执行步骤
    - 看Debug区域： 对照看程序执行步骤
    - 看varable区域： 观察变量的创建、复制、销毁
  + 如何消除断点
    - 把设置断点的方式再执行一遍
    - 切换到Debug视图，选择 Breakpoints -- Remove All Breakpoints 就是2个XX的图标

## 项目的删除

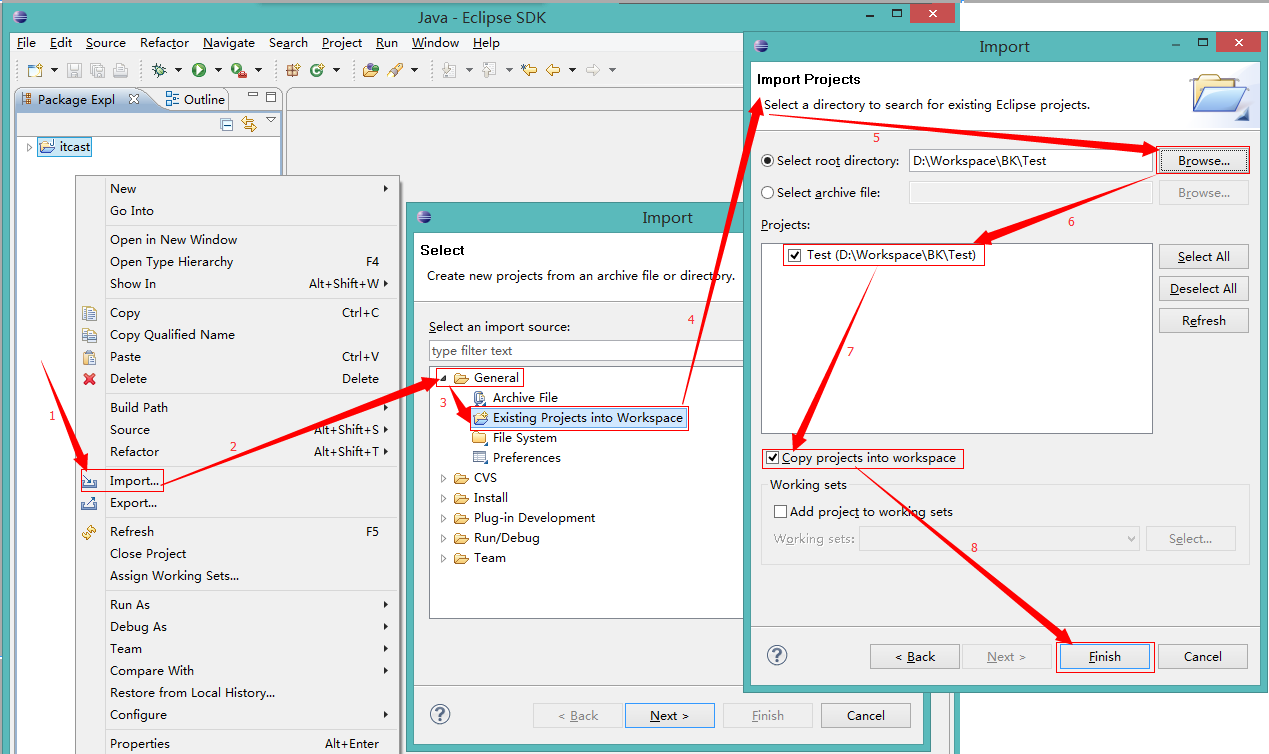
* 在项目区域删除
  + 在左侧PackageExplorer区域对指定项目右键，点击delete删除
* 在硬盘上删除
  + 在左侧PackageExplorer区域对指定项目右键，点击delete删除,同时选中下面的在硬盘上删除(Delete project contents on disk)



1. 项目删除

## 项目的导入

* 在左侧的PackageExplorer区域右键，选择 import
* 选择 General -- Existing Projects into Workspace
* 选择后，点击 next
* 在新的窗口中，点击 Browse 找到需要导入的项目（注意，找到项目的名字位置就可以了），选中要导入的项目，点击Copy复选框，点击Finish完成。
* 注意事项
  + Eclipse中已经存在的同名项目不能再次导入

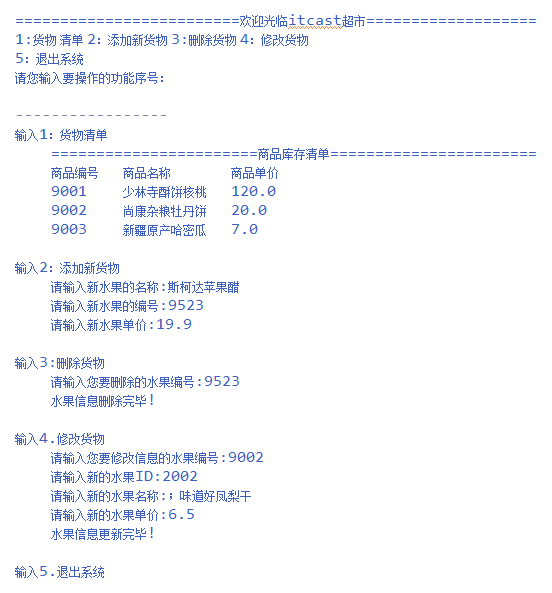


1. 项目导入

# 超市库存管理系统

## 案例介绍

模拟真实的库存管理逻辑，完成超市管理系统的日常功能实现，见下图



## 案例需求分析

根据案例介绍，我们进行分析，首先需要一个功能菜单，然后输入功能序号后，调用序号对应的功能方法，实现想要的操作。分析步骤如下

* 1.完成超市商品初始化。创建商品，将商品添加到集合
* 2.显示来到超市能做的操作，也就是显示主菜单
* 3.根据接收到的功能选项，执行对应的功能
* 3.1.库存货物查询
* 3.2.添加新货物
* 3.3.删除货物
* 3.4.修改货物
* 3.5.退出系统,结束main方法的运行
* 4.循环，回到 2.显示主菜单

## 实现代码步骤

每种库存商品都拥有多项商品信息，为了方便管理每种商品的信息，我们对商品信息进行封装，编写FruitItem.java文件

**public** **class** FruitItem {

**int** ID; //商品编号

String name; //商品名称

**double** price; //商品单价

**double** number; //商品数量

**double** money; //商品金额

}

上述代码中，对商品信息（编号、名称、单价、数量、金额）进行了封装。这样做的好处在于以后只要找到这个商品，就能够知道该商品的每项信息了。

编写FruitStore.java，完成如下功能：

* 编写main主方法

**public** **static** **void** main(String[] args) {

//定义数组,记录每个商品信息

ArrayList<FruitItem> list = **new** ArrayList<FruitItem>();

/\*

\* 1.超市商品初始化

\*/

*init*(list);

while(true){

/\*

\* 2.显示主菜单

\*/

mainMenu();

/\*

\* 3.根据接收到的功能选项，执行对应的功能

\*/

chooseFunction(list);

}

}

上述代码中，通过while(true)循环，完成用户调用不同功能。实现重复功能选择操作。

* 1.超市商品初始化。创建商品，将商品添加到集合

**public** **static** **void** init(ArrayList<FruitItem> list) {

FruitItem item = **new** FruitItem();

item.name = "少林寺酥饼核桃";

item.ID = 9001;

item.price = 120;

FruitItem item2 = **new** FruitItem();

item2.name = "尚康杂粮牡丹饼";

item2.ID = 9002;

item2.price = 20;

FruitItem item3 = **new** FruitItem();

item3.name = "新疆原产哈密瓜";

item3.ID = 3;

item3.price = 9007;

list.add(item);

list.add(item2);

list.add(item3);

}

上述代码中，完成商品集合的初始化操作，添加了3件商品。

* 2.显示来到超市能做的操作，也就是显示主菜单

**public** **static** **void** mainMenu() {

System.*out*.println("=========================欢迎光临itcast超市=========================");

System.*out*.println("1:查询货物 2：添加新货物 3:删除货物 4：修改货物");

System.*out*.println("5：退出系统");

}

上述代码，用来完成用户操作界面的显示。

* 3.根据接收到的功能选项，执行对应的功能

**public** **static** **void** chooseFunction(ArrayList<FruitItem> list) {

System.*out*.println("请您输入要操作的功能序号：");

Scanner sc = **new** Scanner(System.*in*);//键盘输入对象

String choose = sc.next();

//根据接收到的选择，执行对应的功能

**switch** (choose) {

**case** "1": //3.1.库存货物查询

showFruitList(list);

break;

**case** "2": //3.2.添加新货物

addFruitItem(list);

break;

**case** "3": //3.3.删除货物

delFruitItem(list);

break;

**case** "4": //3.4.修改货物

updateFruitItem(list);

break;

**case** "5": //3.5.退出系统,退出JVM

System.*out*.println("退出系统");

return;

default:

System.*out*.println("对不起，没有您输入的功能，请重新选择");

break;

}

}

上述代码中，参数list为库存商品集合。当键盘输入功能序号后，通过switch语句匹配后，调用对应的方法完成对应功能的实现。

* 3.1.库存货物查询

**public** **static** **void** showFruitList(ArrayList<FruitItem> list) {

System.*out*.println("=======================商品库存清单=======================");

System.*out*.println("商品编号\t商品名称\t\t商品单价");

//查询每种库存商品信息

**for** (**int** i = 0; i < list.size(); i++) {

FruitItem item = list.get(i);

System.*out*.println(item.ID + "\t" + item.name + "\t" + item.price);

}

}

上述代码中，用来查询所有库存商品信息。

* 3.2.添加新货物

**public** **static** **void** addFruitItem(ArrayList<FruitItem> list) {

//创建新获取对象

FruitItem newItem = **new** FruitItem();

Scanner sc = **new** Scanner(System.*in*);//键盘输入对象

//提示输入信息

System.*out*.print("请输入新水果的名称:");

newItem.name = sc.next();

System.*out*.print("请输入新水果的编号:");

newItem.ID = sc.nextInt();

System.*out*.print("请输入新水果单价:");

newItem.price = sc.nextDouble();

//向货物集合中添加新的物品项

list.add(newItem);

}

上述代码中，通过键盘录入，将录入的商品信息存储到库存商品集合中。

* 3.3.删除货物

**public** **static** **void** delFruitItem(ArrayList<FruitItem> list) {

System.*out*.print("请输入您要删除的水果编号:");

Scanner sc = **new** Scanner(System.*in*);//键盘输入对象

**int** fruitID = sc.nextInt();

//删除集合元素

**for** (**int** i = 0; i < list.size(); i++) {

FruitItem thisItem = list.get(i);

**if**(thisItem.ID == fruitID) {

list.remove(thisItem);

System.*out*.println("水果信息删除完毕!");

return;

}

}

System.*out*.println("对不起，没有这个编号的水果!");

}

上述代码中，通过键盘录入，在库存集合中查找对应的商品编号，如果找到该商品，将该商品删除，否则，提示没有这个编号的商品。

* 3.4.修改货物

**public** **static** **void** updateFruitItem(ArrayList<FruitItem> list) {

System.*out*.println();

System.*out*.print("请输入您要修改信息的水果编号:");

Scanner sc = **new** Scanner(System.*in*);//键盘输入对象

**int** fruitID = sc.nextInt();

//更新集合元素

**for** (**int** i = 0; i < list.size(); i++) {

FruitItem thisItem = list.get(i);

**if**(thisItem.ID == fruitID) {

System.*out*.print("请输入新的水果ID:");

thisItem.ID = sc.nextInt();

System.*out*.print("请输入新的水果名称:");

thisItem.name = sc.next();

System.*out*.print("请输入新的水果单价:");

thisItem.price = sc.nextDouble();

System.*out*.println("水果信息更新完毕!");

return;

}

}

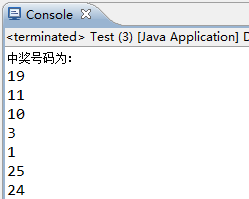
System.*out*.println("对不起，没有这个编号的水果!");

}

上述代码中，通过键盘录入，在库存集合中查找对应的商品编号，如果找到该商品，则键盘录入修改商品信息，否则，提示没有这个编号的商品。

# 趣味乐园

如下图所示，完成对应的双色球摇奖功能：



* 双色球摇奖
  + 随机摇出1-33之间的7个随机数，不能重复
  + 提示：
    - 存储双色球可使用ArrayList集合完成
    - 随机摇出双色球可使用Random类nextInt()方法、ArrayList集合remove()方法完成

第9天 面向对象

今日内容介绍

* 面向对象
* 封装

# 面向对象

## 理解什么是面向过程、面向对象

面向过程与面向对象都是我们编程中，编写程序的一种思维方式。

* 面向过程的程序设计方式，是遇到一件事时，思考“我该怎么做”，然后一步步实现的过程。

例如：公司打扫卫生（擦玻璃、扫地、拖地、倒垃圾等），按照面向过程的程序设计方式会思考“打扫卫生我该怎么做，然后一件件的完成”，最后把公司卫生打扫干净了。

* 面向对象的程序设计方式，是遇到一件事时，思考“我该让谁来做”，然后那个“谁”就是对象，他要怎么做这件事是他自己的事，反正最后一群对象合力能把事就好就行了。

例如，公司打扫卫生（擦玻璃、扫地、拖地、倒垃圾等），按照面向对象的程序设计方式会思考“我该让谁来做，如小明擦玻璃、让小丽扫地、让小郭拖地、让小强倒垃圾等”,这里的“小明、小丽、小郭、小强”就是对象，他们要打扫卫生，怎么打扫是他们自己的事，反正最后一群对象合力把公司卫生打扫干净了。

## 面向对象举例

* 买电脑（组装机）

先使用面向过程说明买电脑这件事：假如我们需要买组装电脑，这时首先会在网上查询具体每一个硬件的参数和报价。然后会去电脑城进行多家询价，接着询价结束后回家根据具体的结果分析出自己比较满意的哪家报价，接着会到这家店里进行组装，组装时还需要进行现场监督，组装完成安装相应的系统，然后电脑抱回家。

分析上述整个过程大体分一下几步：上网查询参数和报价、电脑城询价、现场安装和监督、抱电脑回家。在整个过程中我们参与了每一个细节，并且会感觉相当累。

使用面向对象说明买电脑这件事：假如我们需要买组装机，这时应该找一个懂电脑硬件的人，让他帮我们查看参数和报价，并进行询价和杀价，以及现场组装监督。而我们自己并不需要亲历亲为具体怎么做，只要告诉这个人我们想要的具体需求即可。

分析上述整个过程，发现瞬间变的十分轻松，只要找到懂电脑硬件的这个人，我们的问题都可以解决。并且在这个过程中我们不用那么辛苦。

## 面向对象思维方式的好处

通过生活中的真实场景使用面向对象分析完之后，我们开始分析面向过程和面向对象的差异做出总结：

* 面向对象思维方式是一种更符合人们思考习惯的思想
* 面向过程思维方式中更多的体现的是执行者（自己做事情），面向对象中更多的体现是指挥者（指挥对象做事情）。
* 面向对象思维方式将复杂的问题简单化。

# 类与对象

## 对象在需求中的使用

对面向对象有了了解之后，我们来说说在具体问题中如何使用面向对象去分析问题，和如何使用面向对象。

我们把大象装冰箱为例进行分析。

在针对具体的需求，可以使用名词提炼的办法进行分析，寻找具体的对象。

需求：把大象装冰箱里

对象：大象、冰箱

分三步：

1. 打开冰箱门
2. 将大象装进去
3. 关闭冰箱门

分析发现打开、装、关闭都是冰箱的功能。即冰箱对象具备如下功能：

冰箱打开

冰箱存储

冰箱关闭

用伪代码描述，上述需求中有两个具体的事物 大象 和 冰箱

描述大象：

class 大象

{

}

描述冰箱

class冰箱

{

void 打开(){}

void 存储(大象){}

void 关闭(){}

}

当把具体的事物描述清楚之后，需要使用这些具体的事物，Java使用具体的事物，需要通过new关键字来创建这个事物的具体实例。

使用对象：

1. 创建冰箱的对象

冰箱 bx = new 冰箱();

1. 调用冰箱的功能

对象.功能()；

bx.打开();

bx.存储(new 大象());

bx.关闭();

* 总结：

1. 先按照名词提炼问题领域中的对象
2. 对对象进行描述，其实就是在明确对象中应该具备的属性和功能
3. 通过new的方式就可以创建该事物的具体对象
4. 通过该对象调用它以后的功能。

## 对象在代码中的体现

在分析现实生活中的事物时发现，这些事物都有其具体的特点和功能，这些特点和功能就组成了这个特殊的事物。

描述小汽车。

分析：

事物的特点（属性）：

颜色。

轮胎个数。

事物的(功能)：

运行。

发现：事物其实就是由特点（属性）和行为（功能）组成的。

可以简单理解：属性就是数值，其实就是变量；行为就是功能，就是方法。

小汽车 {

颜色；

轮胎个数；

运行() { }

}

通过计算机语言Java来描述这个事物。

* 定义类的格式

public class 类名 {

//可编写0至n个属性

数据类型 变量名1；

数据类型 变量名2；

//可编写0至n个方法

修饰符 返回值类型 方法名(参数){

执行语句;

}

}

* 汽车类

public class Car {

String color;

int number;

void run() {

System.*out*.println(color + ":" + number);

}

}

通过代码的描述，知道类的真正意义就是在描述事物。属性和功能统称为事物中的成员。

事物的成员分为两种：成员属性和成员功能。

成员属性在代码中的体现就是成员变量

成员功能在代码中的体现就是成员方法

把写好的代码测试一下。需要一个可以独立运行类。

* 创建对象的格式：

类名 对象名 = new 类名();

* 测试类

**public** **class** CarDemo {

**public** **static** **void** main(String[] args) {

/\*

\* 测试：Car类中的run方法。

\*/

// 1,创建Car的对象。给对象起个名字。

Car c = **new** Car();// c是类类型的变量。c指向了一个具体的Car类型的对象。

// 2,通过已有的对象调用该对象的功能。格式：对象.对象成员;

// 3,可以该对象的属性赋值。

c.color = "red";

c.number = 4;

c.run();

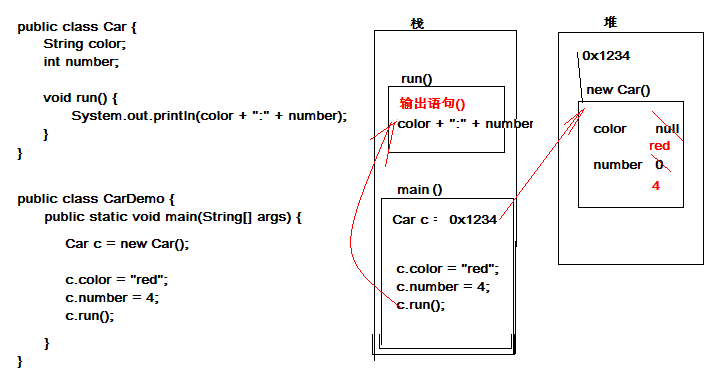
}

}

## 对象的内存图解

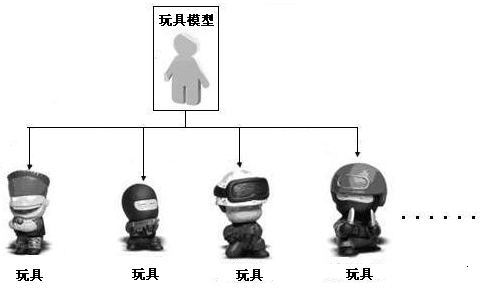
经过上面对小汽车的描述，和Java代码测试，我们虽然可以将生活中的事物使用Java代码描述出来，但是这些代码在内存中是如何执行的，接下来我们需要研究下对象在内存的图解。

接下来就是分析对象在内存中的分配情况。这里需要画图一步一步演示，严格按照画图流程讲解内存对象创建过程。



## 类和对象的区别

面向对象的编程思想力图在程序中对事物的描述与该事物在现实中的形态保持一致。为了做到这一点，面向对象的思想中提出两个概念，即类和对象。其中，类是对某一类事物的抽象描述，而对象用于表示现实中该类事物的个体。接下来通过一个图例来抽象描述类与对象的关系，如下图所示。



1. 类与对象

在上图中，可以将玩具模型看作是一个类，将一个个玩具看作对象，从玩具模型和玩具之间的关系便可以看出类与对象之间的关系。类用于描述多个对象的共同特征，它是对象的模板。对象用于描述现实中的个体，它是类的实例。从上图中可以明显看出对象是根据类创建的，并且一个类可以对应多个对象，接下来分别讲解什么是类和对象。

经过前面几个知识点的学习，基本上掌握了类是用于描述事物的，类中可以定义事物的属性和行为。而对象是通过描述的这个类，使用new关键字创建出来，通过对象就可以调用该对象具体的属性和功能了。

## 局部变量和成员变量区别

理解清楚了类和对象之后，结合前5天的学习知识，发现在描述类的属性和前面学习定义变量差别不大，唯一区别就是位置发生了改变，那么类中定义的变量，和在方法定义的变量有啥差别呢？

回忆以前学习时变量的定义方式，和位置，以及现在定义类中属性的特点。总结下面几点异同

区别一：定义的位置不同

定义在类中的变量是成员变量

定义在方法中或者{}语句里面的变量是局部变量

区别二：在内存中的位置不同

成员变量存储在对内存的对象中

局部变量存储在栈内存的方法中

区别三：声明周期不同

成员变量随着对象的出现而出现在堆中，随着对象的消失而从堆中消失

局部变量随着方法的运行而出现在栈中，随着方法的弹栈而消失

区别四：初始化不同

成员变量因为在堆内存中，所有默认的初始化值

局部变量没有默认的初始化值，必须手动的给其赋值才可以使用。

## 基本类型和引用类型作为参数传递

引用类型数据和基本类型数据作为参数传递有没有差别呢？我们用如下代码进行说明，并配合图解让大家更加清晰

|  |  |
| --- | --- |
| class Demo  {  public static void main(String[] args)  {  int x = 4;  show(x);  System.out.println("x="+x);  }  public static void show(int x)  {  x = 5;    }  } |  |

基本类型作为参数传递时，其实就是将基本类型变量x空间中的值复制了一份传递给调用的方法show()，当在show()方法中x接受到了复制的值，再在show()方法中对x变量进行操作，这时只会影响到show中的x。当show方法执行完成，弹栈后，程序又回到main方法执行，main方法中的x值还是原来的值。

|  |  |
| --- | --- |
| class Demo  {  int x ;  public static void main(String[] args)  {  Demo d = new Demo();  d.x = 5;  show(d);  System.out.println("x="+d.x);  }  public static void show(Demo d)  {  d.x = 6;  }  } |  |

当引用变量作为参数传递时，这时其实是将引用变量空间中的内存地址(引用)复制了一份传递给了show方法的d引用变量。这时会有两个引用同时指向堆中的同一个对象。当执行show方法中的d.x=6时，会根据d所持有的引用找到堆中的对象，并将其x属性的值改为6.show方法弹栈。

由于是两个引用指向同一个对象，不管是哪一个引用改变了引用的所指向的对象的中的值，其他引用再次使用都是改变后的值。

# 封装

## 封装概述

提起封装，大家并不陌生。前面我们学习方法时，就提起过，将具体功能封装到方法中，学习对象时，也提过将方法封装在类中，其实这些都是封装。

封装，它也是面向对象思想的特征之一。面向对象共有三个特征：封装，继承，多态。接下来我们具体学习封装。

* 封装表现：
  + 1、方法就是一个最基本封装体。
  + 2、类其实也是一个封装体。
* 从以上两点得出结论，封装的好处：
  + 1、提高了代码的复用性。
  + 2、隐藏了实现细节，还要对外提供可以访问的方式。便于调用者的使用。这是核心之一，也可以理解为就是封装的概念。
  + 3、提高了安全性。

## 封装举例

机箱：

一台电脑，它是由CPU、主板、显卡、内存、硬盘、电源等部件组长，其实我们将这些部件组装在一起就可以使用电脑了，但是发现这些部件都散落在外面，很容造成不安全因素，于是，使用机箱壳子，把这些部件都装在里面，并在机箱壳上留下一些插口等，若不留插口，大家想想会是什么情况。

总结：机箱其实就是隐藏了办卡设备的细节，对外提供了插口以及开关等访问内部细节的方式。

## 私有private

了解到封装在生活的体现之后，又要回到Java中，细说封装的在Java代码中的体现，先从描述Person说起。

描述人。Person

属性：年龄。

行为：说话：说出自己的年龄。

**class** Person {

**int** age;

String name;

**public** **void** show() {

System.*out*.println("age=" + age + ",name" + name);

}

}

**public** **class** PersonDemo {

**public** **static** **void** main(String[] args) {

// 创建Person对象

Person p = **new** Person();

p.age = -20; // 给Person对象赋值

p.name = "人妖";

p.show(); // 调用Person的show方法

}

}

通过上述代码发现，虽然我们用Java代码把Person描述清楚了，但有个严重的问题，就是Person中的属性的行为可以任意访问和使用。这明显不符合实际需求。

可是怎么才能不让访问呢？需要使用一个Java中的关键字也是一个修饰符 private(私有，权限修饰符)。只要将Person的属性和行为私有起来，这样就无法直接访问。

**class** Person {

**private** **int** age;

**private** String name;

**public** **void** show() {

System.*out*.println("age=" + age + ",name" + name);

}

}

年龄已被私有，错误的值无法赋值，可是正确的值也赋值不了，这样还是不行，那肿么办呢？按照之前所学习的封装的原理，隐藏后，还需要提供访问方式。只要对外提供可以访问的方法，让其他程序访问这些方法。同时在方法中可以对数据进行验证。

一般对成员属性的访问动作：赋值(设置 set)，取值(获取 get)，因此对私有的变量访问的方式可以提供对应的 setXxx或者getXxx的方法。

**class** Person {

// 私有成员变量

**private** **int** age;

**private** String name;

// 对外提供设置成员变量的方法

**public** **void** setAge(**int** a) {

// 由于是设置成员变量的值，这里可以加入数据的验证

**if** (a < 0 || a > 130) {

System.*out*.println(a + "不符合年龄的数据范围");

**return**;

}

age = a;

}

// 对外提供访问成员变量的方法

**public** **void** getAge() {

**return** age;

}

}

* 总结：

类中不需要对外提供的内容都私有化，包括属性和方法。

以后再描述事物，属性都私有化，并提供setXxx getXxx方法对其进行访问。

* 注意：私有仅仅是封装的体现形式而已。

## this关键字

### 成员变量和局部变量同名问题

当在方法中出现了局部变量和成员变量同名的时候，那么在方法中怎么区别局部变量成员变量呢？可以在成员变量名前面加上this.来区别成员变量和局部变量

**class** Person {

**private** **int** age;

**private** String name;

**public** **void** speak() {

**this**.name = "小强";

**this**.age = 18;

System.*out*.println("name=" + **this**.name + ",age=" + **this**.age);

}

}

**class** PersonDemo {

**public** **static** **void** main(String[] args) {

Person p = **new** Person();

p.speak();

}

}

### 对象的内存解释

我们已经学习了如何把生活中的事物使用Java代码描述，接下来我们分析对象在内存中的分配情况。这里需要画图一步一步演示，严格按照画图流程讲解内存对象创建使用过程。

**class** Person {

**private** **int** age;

**public** **int** getAge() {

**return** **this**.age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

}

**public** **class** PersonDemo {

**public** **static** **void** main(String[] args) {

Person p = **new** Person();

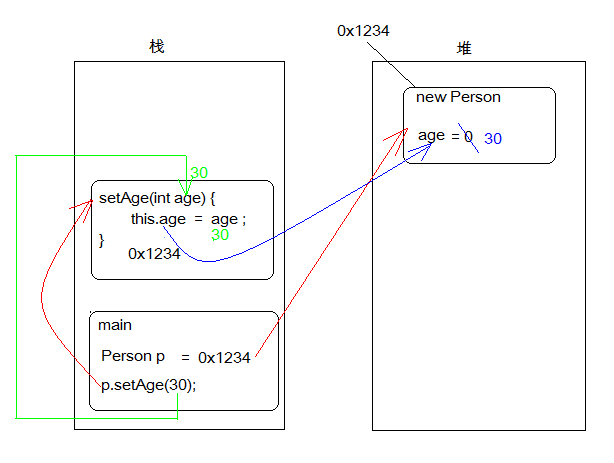
p.setAge(30);

System.*out*.println("大家好，今年我" + p.getAge() + "岁");

}

}

下图为程序中内存对象的创建使用过程。



1. 内存对象创建使用过程

程序执行流程说明：

1. 先执行main方法（压栈），执行其中的 Person p = new Person()；
2. 在堆内存中开辟空间，并为其分配内存地址0x1234，紧接着成员变量默认初始化(age = 0)；将内存地址0x1234赋值给栈内中的Person p 变量
3. 继续执行p.setAge(30)语句，这时会调用setAge(int age)方法，将30赋值为setAge方法中的“age”变量；执行this.age = age语句，将age变量值30 赋值给成员变量this.age为30；
4. setAge()方法执行完毕后（弹栈），回到main()方法，执行输出语句System.out.println()，控制台打印p对象中的age年龄值。

* 注意：
  + this到底代表什么呢？this代表的是对象，具体代表哪个对象呢？哪个对象调用了this所在的方法，this就代表哪个对象。
  + 上述代码中的 p.setAge(30)语句中，setAge(int age)方法中的this代表的就是p对象。

### this的应用

学习this的用法之后，现在做个小小的练习。

需求：在Person类中定义功能，判断两个人是否是同龄人

**class** Person {

**private** **int** age;

**private** String name;

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** speak() {

System.***out***.println("name=" + **this**.name + ",age=" + **this**.age);

}

// 判断是否为同龄人

**public** **boolean** equalsAge(Person p) {

// 使用当前调用该equalsAge方法对象的age和传递进来p的age进行比较

// 由于无法确定具体是哪一个对象调用equalsAge方法，这里就可以使用this来代替

/\*

\* if(this.age == p.age) { return true; } return false;

\*/

**return** **this**.age == p.age;

}

}

# 综合案例---随机点名案例重构

## 案例介绍

随机点名器，即在全班同学中随机的找出一名同学，打印这名同学的个人信息。

此案例在我们前几天课程学习中，已经介绍，现在我们要做的是对原有的案例进行升级，使用新的方式来实现。

我们来完成随机点名器，它具备以下3个内容：

* 存储所有同学姓名
* 总览全班同学姓名
* 随机点名其中一人，打印到控制台

## 案例分析

全班同学中随机的找出一名同学，打印这名同学的个人信息。

我们对本案例进行分析，得出如下分析结果：

1.存储全班同学信息（姓名、年龄）

2.打印全班同学每一个人的信息（姓名、年龄）

3.在班级总人数范围内，随机产生一个随机数，查找该随机数所对应的同学信息（姓名、年龄）并打印

随机点名器明确地分为了三个功能。如果将多个独立功能的代码写到一起，则代码相对冗长，我们可以针对不同的功能可以将其封装到一个方法中，将完整独立的功能分离出来。

而在存储同学姓名时，如果对每一个同学都定义一个变量进行姓名存储，则会出现过多孤立的变量，很难一次性将全部数据持有。此时，我们采用ArrayList集合来解决多个学生信息的存储问题。

## 重构内容分析

将原来使用的简单Student类，封装为包装属性和方法的相对完整的Student类，并将所有访问属性的地方改为通过get/set方法访问。

重构部分已使用红色样色字体表示

## 实现代码步骤

每名学生都拥有多项个人信息，为了方便管理每个人的信息，我们对学生信息进行封装，编写Student.java文件

/\*\*

\* 学生信息类

\*/

**public** **class** Student {

private String name; // 姓名

private **int** age; // 年龄

**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;

}

}

上述代码中，对学生信息（姓名、年龄）进行了封装。这样做的好处在于，以后只要找到这名学生，就能够知道他的每项个人信息了。

接下来我们编写CallName.java文件，完成程序的编写。

* main方法中调用三个独立方法

**public** **static** **void** main(String[] args) {

ArrayList<Student> list = **new** ArrayList<Student>(); //1.1创建一个可以存储多个同学名字的容器

/\*

\* 1.存储全班同学信息

\*/

addStudent(list);

/\*

\* 2.打印全班同学每一个人的信息（姓名、年龄）

\*/

printStudent(list);

/\*

\* 3.随机对学生点名，打印学生信息

\*/

randomStudent(list);

}

* 存储所有学生的个人信息

/\*\*

\* 1.存储全班同学名字

\*/

**public** **static** **void** addStudent(ArrayList<Student> list) {

//键盘输入多个同学名字存储到容器中

Scanner sc = **new** Scanner(System.***in***);

**for** (**int** i = 0; i < 3; i++) {

//创建学生

Student s = **new** Student();

System.***out***.println("存储第"+i+"个学生姓名：");

String name = sc.next();

s.setName(name);

System.***out***.println("存储第"+i+"个学生年龄：");

**int** age = sc.nextInt();

s.setAge(age);

//添加学生到集合

list.add(s);

}

}

上述方法中，方法参数list中用来表示已存储所有学生。通过Scanner，完成新学生信息（姓名，年龄）的录入，并将学生添加到集合中。

* 打印全班同学每一个人的信息

/\*\*

\* 2.打印全班同学每一个人的信息（姓名、年龄）

\*/

**public** **static** **void** printStudent (ArrayList<Student> list) {

**for** (**int** i = 0; i < list.size(); i++) {

Student s = list.get(i);

System.***out***.println("姓名："+s.getName() +",年龄："+s.getAge());

}

}

上述方法中，方法参数list中用来表示已存储所有学生。通过遍历集合中的每个元素，得到每个同学信息，并输出打印。

* 随机对学生点名，打印学生信息

/\*\*

\* 3.随机对学生点名，打印学生信息

\*/

**public** **static** **void** randomStudent (ArrayList<Student> list) {

//在班级总人数范围内，随机产生一个随机数

**int** index = **new** Random().nextInt(list.size());

//在容器（ArrayList集合）中，查找该随机数所对应的同学信息（姓名、年龄）

Student s = list.get(index);

System.***out***.println("被随机点名的同学："+s.getName() + "，年龄:" + s.getAge());

}

上述方法中，通过随机数类Random产生一个从0到集合长度的随机索引。使用该索引获取ArrayList集合中对应的值，便得到了全班同学的随机学生信息并打印。

# 总结

## 知识点总结

* 类与对象
  + 类，用于描述多个对象的共同特征，它是对象的模板。
  + 对象，用于描述现实中的个体，它是类的实例。
  + 类的定义：使用关键字class来定义java中的类
    - 格式：

class 类名 {

//属性

数据类型 变量名;

…

//方法

修饰符 返回值类型 方法名(参数){ }

…

}

* + 创建对象：
    - 格式：

类名 对象名 = new 类名();

* 封装（private关键字）
  + 封装，把对象的属性与方法的实现细节隐藏，仅对外提供一些公共的访问方式
  + 封装的体现：
    - 变量:使用 private 修饰，这就是变量的封装
    - 方法:也是一种封装，封装了多条代码
    - 类： 也是一种封装，封装了多个方法
  + private关键字，私有的意思
    - 它可以用来修饰类中的成员(成员变量，成员方法)
    - private的特点：
      * private修饰的成员只能在当前类中访问，其他类中无法直接访问
* this关键字
  + this关键字，本类对象的引用
    - this是在方法中使用的，哪个对象调用了该方法，那么，this就代表调用该方法的对象引用
    - this什么时候存在的？当创建对象的时候，this存在的
    - this的作用：用来区别同名的成员变量与局部变量（this.成员变量）

public void setName(String name) {

this.name = name;

}

# 第10天面向对象

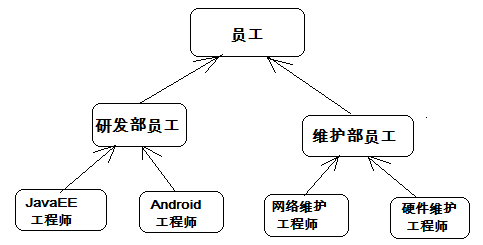
今日内容介绍

* 继承
* 抽象类

# 继承

## 继承的概念

在现实生活中，继承一般指的是子女继承父辈的财产。在程序中，继承描述的是事物之间的所属关系，通过继承可以使多种事物之间形成一种关系体系。例如公司中的研发部员工和维护部员工都属于员工，程序中便可以描述为研发部员工和维护部员工继承自员工，同理，JavaEE工程师和Android工程师继承自研发部员工，而维网络维护工程师和硬件维护工程师继承自维护部员工。这些员工之间会形成一个继承体系，具体如下图所示。



1. 员工继承关系图

在Java中，类的继承是指在一个现有类的基础上去构建一个新的类，构建出来的新类被称作子类，现有类被称作父类，子类会自动拥有父类所有可继承的属性和方法。

## 继承的格式&使用

在程序中，如果想声明一个类继承另一个类，需要使用extends关键字。

格式：

class 子类 extends 父类 {}

接下来通过一个案例来学习子类是如何继承父类的，如下所示。Example01.java

/\*

\* 定义员工类Employee

\*/

**class** Employee {

String name; // 定义name属性

// 定义员工的工作方法

**public** **void** work() {

System.***out***.println("尽心尽力地工作");

}

}

/\*

\* 定义研发部员工类Developer 继承 员工类Employee

\*/

**class** Developer **extends** Employee {

// 定义一个打印name的方法

**public** **void** printName() {

System.***out***.println("name=" + name);

}

}

/\*

\* 定义测试类

\*/

**public** **class** Example01 {

**public** **static** **void** main(String[] args) {

Developer d = **new** Developer(); // 创建一个研发部员工类对象

d.name = "小明"; // 为该员工类的name属性进行赋值

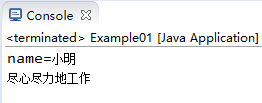
d.printName(); // 调用该员工的printName()方法

d.work(); // 调用Developer类继承来的work()方法

}

}

运行结果如下图所示。



1. 运行结果

在上述代码中，Developer类通过extends关键字继承了Employee类，这样Developer类便是Employee类的子类。从运行结果不难看出，子类虽然没有定义name属性和work()方法，但是却能访问这两个成员。这就说明，子类在继承父类的时候，会自动拥有父类的成员。

## 继承的好处&注意事项

继承的好处：

1、继承的出现提高了代码的复用性，提高软件开发效率。

2、继承的出现让类与类之间产生了关系，提供了多态的前提。

在类的继承中，需要注意一些问题，具体如下：

* 1、在Java中，类只支持单继承，不允许多继承，也就是说一个类只能有一个直接父类，例如下面这种情况是不合法的。

class A{}

class B{}

class C extends A,B{} // C类不可以同时继承A类和B类

* 2、多个类可以继承一个父类，例如下面这种情况是允许的。

class A{}

class B extends A{}

class C extends A{} // 类B和类C都可以继承类A

* 3、在Java中，多层继承是可以的，即一个类的父类可以再去继承另外的父类，例如C类继承自B类，而B类又可以去继承A类，这时，C类也可称作A类的子类。下面这种情况是允许的。

class A{}

class B extends A{} // 类B继承类A，类B是类A的子类

class C extends B{} // 类C继承类B，类C是类B的子类，同时也是类A的子类

* 4、在Java中，子类和父类是一种相对概念，也就是说一个类是某个类父类的同时，也可以是另一个类的子类。例如上面的这种情况中，B类是A类的子类，同时又是C类的父类。

## 继承-子父类中成员变量的特点

了解了继承给我们带来的好处，提高了代码的复用性。继承让类与类或者说对象与对象之间产生了关系。那么，当继承出现后，类的成员之间产生了那些变化呢？

类的成员重点学习成员变量、成员方法的变化。

成员变量：如果子类父类中出现不同名的成员变量，这时的访问是没有任何问题。

看如下代码：

class Fu

{

//Fu中的成员变量。

int num = 5;

}

class Zi extends Fu

{

//Zi中的成员变量

int num2 = 6;

//Zi中的成员方法

public void show()

{

//访问父类中的num

System.out.println("Fu num="+num);

//访问子类中的num2

System.out.println("Zi num2="+num2);

}

}

class Demo

{

public static void main(String[] args)

{

Zi z = new Zi(); //创建子类对象

z.show(); //调用子类中的show方法

}

}

代码说明：Fu类中的成员变量是非私有的，子类中可以直接访问，若Fu类中的成员变量私有了，子类是不能直接访问的。

当子父类中出现了同名成员变量时，在子类中若要访问父类中的成员变量，必须使用关键字super来完成。super用来表示当前对象中包含的父类对象空间的引用。super今天不做具体讲解，在课程第12天会详细讲解。

在子类中，访问父类中的成员变量格式：

super.父类中的成员变量

看如下代码：

class Fu

{

//Fu中的成员变量。

int num = 5;

}

class Zi extends Fu

{

//Zi中的成员变量

int num = 6;

void show()

{

//子父类中出现了同名的成员变量时

//在子类中需要访问父类中非私有成员变量时，需要使用super关键字

//访问父类中的num

System.out.println("Fu num="+super.num);

//访问子类中的num2

System.out.println("Zi num2="+this.num);

}

}

class Demo5

{

public static void main(String[] args)

{

Zi z = new Zi(); //创建子类对象

z.show(); //调用子类中的show方法

}

}

## 继承-子父类中成员方法特点-重写&应用

* 子父类中成员方法的特点

当在程序中通过对象调用方法时，会先在子类中查找有没有对应的方法，若子类中存在就会执行子类中的方法，若子类中不存在就会执行父类中相应的方法。

看如下代码：

class Fu{

public void show(){

System.out.println("Fu类中的show方法执行");

}

}

class Zi extends Fu{

public void show2(){

System.out.println("Zi类中的show2方法执行");

}

}

public class Test{

public static void main(String[] args) {

Zi z = new Zi();

z.show(); //子类中没有show方法，但是可以找到父类方法去执行

z.show2();

}

}

* 成员方法特殊情况——覆盖

子类中出现与父类一模一样的方法时，会出现覆盖操作，也称为override重写、复写或者覆盖。

class Fu

{

public void show()

{

System.out.println("Fu show");

}

}

class Zi extends Fu

{

//子类复写了父类的show方法

public void show()

{

System.out.println("Zi show");

}

}

* 方法重写（覆盖）的应用：

当子类需要父类的功能，而功能主体子类有自己特有内容时，可以重写父类中的方法，这样，即沿袭了父类的功能，又定义了子类特有的内容。

举例：比如手机，当描述一个手机时，它具有发短信，打电话，显示来电号码功能，后期由于手机需要在来电显示功能中增加显示姓名和头像，这时可以重新定义一个类描述智能手机，并继承原有描述手机的类。并在新定义的类中覆盖来电显示功能，在其中增加显示姓名和头像功能。

在子类中，访问父类中的成员方法格式：

super.父类中的成员方法();

看如下代码：

public class Test {

public static void main(String[] args) {

new NewPhone().showNum();

}

}

//手机类

class Phone{

public void sendMessage(){

System.out.println("发短信");

}

public void call(){

System.out.println("打电话");

}

public void showNum(){

System.out.println("来电显示号码");

}

}

//智能手机类

class NewPhone extends Phone{

//覆盖父类的来电显示号码功能，并增加自己的显示姓名和图片功能

public void showNum(){

//调用父类已经存在的功能使用super

super.showNum();

//增加自己特有显示姓名和图片功能

System.out.println("显示来电姓名");

System.out.println("显示头像");

}

}

## 方法重写的注意事项

重写需要注意的细节问题：

* 子类方法覆盖父类方法，必须要保证权限大于等于父类权限。

class Fu(){

void show(){}

public void method(){}

}

class Zi() extends Fu{

public void show(){} //编译运行没问题

void method(){} //编译错误

}

* 写法上稍微注意:必须一模一样:方法的返回值类型 方法名 参数列表都要一样。

总结：当一个类是另一个类中的一种时，可以通过继承，来继承属性与功能。如果父类具备的功能内容需要子类特殊定义时，进行方法重写。

# 抽象类

## 抽象类-产生

当编写一个类时，我们往往会为该类定义一些方法，这些方法是用来描述该类的功能具体实现方式，那么这些方法都有具体的方法体。

但是有的时候，某个父类只是知道子类应该包含怎么样的方法，但是无法准确知道子类如何实现这些方法。比如一个图形类应该有一个求周长的方法，但是不同的图形求周长的算法不一样。那该怎么办呢？

分析事物时，发现了共性内容，就出现向上抽取。会有这样一种特殊情况，就是方法功能声明相同，但方法功能主体不同。那么这时也可以抽取，但只抽取方法声明，不抽取方法主体。那么此方法就是一个抽象方法。

描述JavaEE工程师：行为：工作。

描述Android工程师：行为：工作。

JavaEE工程师和Android工程师之间有共性，可以进行向上抽取。抽取它们的所属共性类型：研发部员工。由于JavaEE工程师和Android工程师都具有工作功能，但是他们具体工作内容却不一样。这时在描述研发部员工时，发现了有些功能（工作）不具体，这些不具体的功能，需要在类中标识出来，通过java中的关键字abstract(抽象)。

当定义了抽象函数的类也必须被abstract关键字修饰，被abstract关键字修饰的类是抽象类。

## 抽象类&抽象方法的定义

抽象方法定义的格式：

public abstract 返回值类型 方法名(参数);

抽象类定义的格式：

abstract class 类名 {

}

看如下代码：

//研发部员工

**abstract** **class** Developer {

**public** **abstract** **void** work();//抽象函数。需要abstract修饰，并分号;结束

}

//JavaEE工程师

**class** JavaEE **extends** Developer{

**public** **void** work() {

System.***out***.println("正在研发淘宝网站");

}

}

//Android工程师

**class** Android **extends** Developer {

**public** **void** work() {

System.***out***.println("正在研发淘宝手机客户端软件");

}

}

## 抽象类的特点：

1、抽象类和抽象方法都需要被abstract修饰。抽象方法一定要定义在抽象类中。

2、抽象类不可以直接创建对象，原因：调用抽象方法没有意义。

3、只有覆盖了抽象类中所有的抽象方法后，其子类才可以创建对象。否则该子类还是一个抽象类。

之所以继承抽象类，更多的是在思想，是面对共性类型操作会更简单。

## 抽象类的细节问题：

1、抽象类一定是个父类？

是的，因为不断抽取而来的。

2、抽象类中是否可以不定义抽象方法。

是可以的，那这个抽象类的存在到底有什么意义呢？不让该类创建对象,方法可以直接让子类去使用

3、抽象关键字abstract不可以和哪些关键字共存？

* 1、private：私有的方法子类是无法继承到的，也不存在覆盖，而abstract和private一起使用修饰方法，abstract既要子类去实现这个方法，而private修饰子类根本无法得到父类这个方法。互相矛盾。
* 2、final，暂时不关注，后面学
* 3、static，暂时不关注，后面学

# 综合案例---员工类系列定义

## 案例介绍

某IT公司有多名员工，按照员工负责的工作不同，进行了部门的划分（研发部员工、维护部员工）。研发部根据所需研发的内容不同，又分为JavaEE工程师、Android工程师；维护部根据所需维护的内容不同，又分为网络维护工程师、硬件维护工程师。

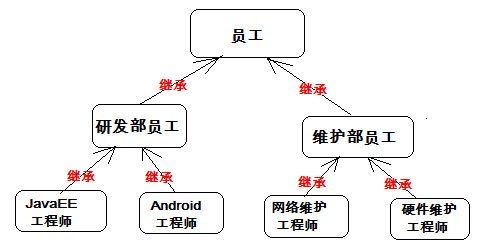
公司的每名员工都有他们自己的员工编号、姓名，并要做它们所负责的工作。

* 工作内容
  + JavaEE工程师：员工号为xxx的 xxx员工，正在研发淘宝网站
  + Android工程师：员工号为xxx的 xxx员工，正在研发淘宝手机客户端软件
  + 网络维护工程师：员工号为xxx的 xxx员工，正在检查网络是否畅通
  + 硬件维护工程师：员工号为xxx的 xxx员工，正在修复打印机

请根据描述，完成员工体系中所有类的定义，并指定类之间的继承关系。进行XX工程师类的对象创建，完成工作方法的调用。

## 案例分析

* 根据上述部门的描述，得出如下的员工体系图



* 根据员工信息的描述，确定每个员工都有员工编号、姓名、要进行工作。则，把这些共同的属性与功能抽取到父类中（员工类），关于工作的内容由具体的工程师来进行指定。
  + 工作内容
    - JavaEE工程师：员工号为xxx的 xxx员工，正在研发淘宝网站
    - Android工程师：员工号为xxx的 xxx员工，正在研发淘宝手机客户端软件
    - 网络维护工程师：员工号为xxx的 xxx员工，正在检查网络是否畅通
    - 硬件维护工程师：员工号为xxx的 xxx员工，正在修复打印机
* 创建JavaEE工程师对象，完成工作方法的调用

## 案例代码实现

* 根据员工体系图，完成类的定义

定义员工类(抽象类)

public abstract class Employee {

**private** String id;// 员工编号

**private** String name; // 员工姓名

**public** String getId() {

**return** id;

}

**public** **void** setId(String id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

//工作方法（抽象方法）

public abstract void work();

}

* 定义研发部员工类Developer 继承 员工类Employee

**public** **abstract** **class** Developer **extends** Employee {

}

* 定义维护部员工类Maintainer 继承 员工类Employee

**public** **abstract** **class** Maintainer **extends** Employee {

}

* 定义JavaEE工程师 继承 研发部员工类，重写工作方法

**public** **class** JavaEE **extends** Developer {

@Override

**public** **void** work() {

System.***out***.println("员工号为 " + getId() + " 的 " + getName() + " 员工，正在研发淘宝网站");

}

}

* 定义Android工程师 继承 研发部员工类，重写工作方法

**public** **class** Android **extends** Developer {

@Override

**public** **void** work() {

System.***out***.println("员工号为 " + getId() + " 的 " + getName() + " 员工，正在研发淘宝手机客户端软件");

}

}

* 定义Network网络维护工程师 继承 维护部员工类，重写工作方法

**public** **class** Network **extends** Maintainer {

@Override

**public** **void** work() {

System.***out***.println("员工号为 " + getId() + " 的 " + getName() + " 员工，正在检查网络是否畅通");

}

}

* 定义Hardware硬件维护工程师 继承 维护部员工类，重写工作方法

**public** **class** Hardware **extends** Maintainer {

@Override

**public** **void** work() {

System.***out***.println("员工号为 " + getId() + " 的 " + getName() + " 员工，正在修复打印机");

}

}

* 在测试类中，创建JavaEE工程师对象，完成工作方法的调用

public class Test {

**public** **static** **void** main(String[] args) {

//创建JavaEE工程师员工对象

JavaEE ee = **new** JavaEE();

//设置该员工的编号

ee.setId("000015");

//设置该员工的姓名

ee.setName("小明");

//调用该员工的工作方法

ee.work();

}

}

# 总结

## 知识点总结

* 继承：是指在一个现有类的基础上去构建一个新的类，构建出来的新类被称作子类，现有类被称作父类，子类会自动拥有父类所有
* 继承的好处：可继承的属性和方法。

提高了代表的可维护性

提高了代码的复用性

让类与类之间产生了继承关系

* + 继承的弊端：

类与类之间的耦合度过高

* + 继承特点：

java中类只能够单继承，不能多继承，可以多层继承

class Yy extends Object {}

class Fu extends Yy{}

class Zi extends Fu {}

所有的类都直接或者间接的继承了 Object类，Object类称为祖宗类

* + 继承的注意事项：

1，使用关键字 extends 让类与类之间 产生继承关系

2, 父类私有的成员，子类不能继承，因为根本看不到

3，不能为了继承某个功能而随意进行继承操作， 必须要符合 is a 的关系

苹果 is a 水果

男人 is a 人

狗 is a 人 ， 这种情况就不能继承了

* + 继承中的成员变量关系：

不同名的变量：

子类直接继承使用

同名的变量：

默认访问的是子类自己的成员变量, 想访问父类中的同名变量，请使用 super.成员变量;

* + 继承中的成员方法关系：

不同名的方法：

子类直接继承使用

同名的方法：

默认访问的是子类自己的成员方法，想访问父类中的同名方法，请使用 super.成员方法();

* + super:用来表示当前对象中包含的父类对象空间的引用

调用父类的成员变量：

super.成员变量;

调用方法的成员方法:

super.成员方法();

* + 方法重写(override)：指 在子父类中，出现了方法声明相同的情况，也叫做方法覆盖，方法复写
    - 方法重写的注意事项：

1. 子类的方法声明要与父类相同

2, 子类要重写方法的方法，方法的权限修饰符不能比父类的更低

* + 3, 父类私有的方法，子类不能够进行方法重写
  + 方法重载(overload)：指 在同一个类中，多个方法名称相同，它们的参数列表不同(个数不同，数据类型不同)
* 抽象
  + 抽象方法： 方法只有声明部分，没有方法体
  + 抽象类： 包含抽象方法的类，一定是抽象类

使用 abstract 修饰的类，是抽象类

* + 抽象类的特点：

1，抽象类与抽象方法都必须使用 abstract来修饰

2，抽象类不能直接创建对象

3，抽象类中可以有抽象方法，也可以没有抽象方法

4，抽象类的子类

a，实现了抽象方法的具体类

b，抽象类

* + 抽象类面试题：

1，抽象类中是否可以没有抽象方法？如果可以，那么，该类还定义成抽象类有意义吗？为什么？

可以没有抽象方法，有意义，不会让其他人直接创建该类对象

# 第11天面向对象

今日内容介绍

* 接口
* 多态

# 接口

## 接口概念

接口是功能的集合，同样可看做是一种数据类型，是比抽象类更为抽象的”类”。

接口只描述所应该具备的方法，并没有具体实现，具体的实现由接口的实现类(相当于接口的子类)来完成。这样将功能的定义与实现分离，优化了程序设计。

请记住：一切事物均有功能，即一切事物均有接口。

## 接口的定义

与定义类的class不同，接口定义时需要使用interface关键字。

定义接口所在的仍为.java文件，虽然声明时使用的为interface关键字的编译后仍然会产生.class文件。这点可以让我们将接口看做是一种只包含了功能声明的特殊类。

定义格式：

public interface 接口名 {

抽象方法1;

抽象方法2;

抽象方法3;

}

使用interface代替了原来的class，其他步骤与定义类相同：

* 接口中的方法均为公共访问的抽象方法
* 接口中无法定义普通的成员变量

## 类实现接口

类与接口的关系为实现关系，即类实现接口。实现的动作类似继承，只是关键字不同，实现使用implements。

其他类(实现类)实现接口后，就相当于声明：”我应该具备这个接口中的功能”。实现类仍然需要重写方法以实现具体的功能。

格式：

class 类 implements 接口 {

重写接口中方法

}

在类实现接口后，该类就会将接口中的抽象方法继承过来，此时该类需要重写该抽象方法，完成具体的逻辑。

* 接口中定义功能，当需要具有该功能时，可以让类实现该接口，只声明了应该具备该方法，是功能的声明。
* 在具体实现类中重写方法，实现功能，是方法的具体实现。

于是，通过以上两个动作将功能的声明与实现便分开了。(此时请重新思考：类是现实事物的描述，接口是功能的集合。)

## 接口中成员的特点

1、接口中可以定义变量，但是变量必须有固定的修饰符修饰，public static final 所以接口中的变量也称之为常量，其值不能改变。后面我们会讲解static与final关键字

2、接口中可以定义方法，方法也有固定的修饰符，public abstract

3、接口不可以创建对象。

4、子类必须覆盖掉接口中所有的抽象方法后，子类才可以实例化。否则子类是一个抽象类。

interface Demo { ///定义一个名称为Demo的接口。

public static final int NUM = 3;// NUM的值不能改变

public abstract void show1();

public abstract void show2();

}

//定义子类去覆盖接口中的方法。类与接口之间的关系是 实现。通过 关键字 implements

class DemoImpl implements Demo { //子类实现Demo接口。

//重写接口中的方法。

public void show1(){}

public void show2(){}

}

## 接口的多实现

了解了接口的特点后，那么想想为什么要定义接口，使用抽象类描述也没有问题，接口到底有啥用呢？

接口最重要的体现：解决多继承的弊端。将多继承这种机制在java中通过多实现完成了。

interface Fu1

{

void show1();

}

interface Fu2

{

void show2();

}

class Zi implements Fu1,Fu2// 多实现。同时实现多个接口。

{

public void show1(){}

public void show2(){}

}

怎么解决多继承的弊端呢？

弊端：多继承时，当多个父类中有相同功能时，子类调用会产生不确定性。

其实核心原因就是在于多继承父类中功能有主体，而导致调用运行时，不确定运行哪个主体内容。

为什么多实现能解决了呢？

因为接口中的功能都没有方法体，由子类来明确。

## 类继承类同时实现接口

接口和类之间可以通过实现产生关系，同时也学习了类与类之间可以通过继承产生关系。当一个类已经继承了一个父类，它又需要扩展额外的功能，这时接口就派上用场了。

子类通过继承父类扩展功能，通过继承扩展的功能都是子类应该具备的基础功能。如果子类想要继续扩展其他类中的功能呢？这时通过实现接口来完成。

class Fu {

public void show(){}

}

interface Inter {

pulbic abstract void show1();

}

class Zi extends Fu implements Inter {

public void show1() {

}

}

接口的出现避免了单继承的局限性。父类中定义的事物的基本功能。接口中定义的事物的扩展功能。

## 接口的多继承

学习类的时候，知道类与类之间可以通过继承产生关系，接口和类之间可以通过实现产生关系，那么接口与接口之间会有什么关系。

多个接口之间可以使用extends进行继承。

interface Fu1{

void show();

}

interface Fu2{

void show1();

}

interface Fu3{

void show2();

}

interface Zi extends Fu1,Fu2,Fu3{

void show3();

}

在开发中如果多个接口中存在相同方法，这时若有个类实现了这些接口，那么就要实现接口中的方法，由于接口中的方法是抽象方法，子类实现后也不会发生调用的不确定性。

## 接口的思想

前面学习了接口的代码体现，现在来学习接口的思想，接下里从生活中的例子进行说明。

**举例：**我们都知道电脑上留有很多个插口，而这些插口可以插入相应的设备，这些设备为什么能插在上面呢？主要原因是这些设备在生产的时候符合了这个插口的使用规则，否则将无法插入接口中，更无法使用。发现这个插口的出现让我们使用更多的设备。

**总结：**接口在开发中的它好处

1. 接口的出现扩展了功能。
2. 接口其实就是暴漏出来的规则。
3. 接口的出现降低了耦合性，即设备与设备之间实现了解耦。

接口的出现方便后期使用和维护，一方是在使用接口（如电脑），一方在实现接口（插在插口上的设备）。例如：笔记本使用这个规则（接口），电脑外围设备实现这个规则（接口）。

## 接口和抽象的区别

明白了接口思想和接口的用法后，接口和抽象类的区别是什么呢？接口在生活体现也基本掌握，那在程序中接口是如何体现的呢？

通过实例进行分析和代码演示抽象类和接口的用法。

1、举例：

犬：

行为：

吼叫；

吃饭；

缉毒犬：

行为：

吼叫；

吃饭；

缉毒；

2、思考：

由于犬分为很多种类，他们吼叫和吃饭的方式不一样，在描述的时候不能具体化，也就是吼叫和吃饭的行为不能明确。当描述行为时，行为的具体动作不能明确，这时，可以将这个行为写为抽象行为，那么这个类也就是抽象类。

可是当缉毒犬有其他额外功能时，而这个功能并不在这个事物的体系中。这时可以让缉毒犬具备犬科自身特点的同时也有其他额外功能，可以将这个额外功能定义接口中。

如下代码演示：

interface 缉毒{

public abstract void 缉毒();

}

//定义犬科的这个提醒的共性功能

abstract class 犬科{

public abstract void 吃饭();

public abstract void 吼叫();

}

// 缉毒犬属于犬科一种，让其继承犬科，获取的犬科的特性，

//由于缉毒犬具有缉毒功能，那么它只要实现缉毒接口即可，这样即保证缉毒犬具备犬科的特性，也拥有了缉毒的功能

class 缉毒犬 extends 犬科 implements 缉毒{

public void 缉毒() {

}

void 吃饭() {

}

void 吼叫() {

}

}

class 缉毒猪 implements 缉毒{

public void 缉毒() {

}

}

3、通过上面的例子总结接口和抽象类的区别：

**相同点:**

* 都位于继承的顶端,用于被其他类实现或继承;
* 都不能直接实例化对象;
* 都包含抽象方法,其子类都必须覆写这些抽象方法;

**区别:**

* 抽象类为部分方法提供实现,避免子类重复实现这些方法,提高代码重用性;接口只能包含抽象方法;
* 一个类只能继承一个直接父类(可能是抽象类),却可以实现多个接口;(接口弥补了Java的单继承)
* 抽象类是这个事物中应该具备的你内容, 继承体系是一种 is..a关系
* 接口是这个事物中的额外内容,继承体系是一种 like..a关系

二者的选用:

* 优先选用接口,尽量少用抽象类;
* 需要定义子类的行为,又要为子类提供共性功能时才选用抽象类;

# 多态

## 多态概述

多态是继封装、继承之后，面向对象的第三大特性。

现实事物经常会体现出多种形态，如学生，学生是人的一种，则一个具体的同学张三既是学生也是人，即出现两种形态。

Java作为面向对象的语言，同样可以描述一个事物的多种形态。如Student类继承了Person类，一个Student的对象便既是Student，又是Person。

Java中多态的代码体现在一个子类对象(实现类对象)既可以给这个子类(实现类对象)引用变量赋值，又可以给这个子类(实现类对象)的父类(接口)变量赋值。

如Student类可以为Person类的子类。那么一个Student对象既可以赋值给一个Student类型的引用，也可以赋值给一个Person类型的引用。

最终多态体现为父类引用变量可以指向子类对象。

多态的前提是必须有子父类关系或者类实现接口关系，否则无法完成多态。

在使用多态后的父类引用变量调用方法时，会调用子类重写后的方法。

## 多态的定义与使用格式

多态的定义格式：就是父类的引用变量指向子类对象

父类类型 变量名 = new 子类类型();

变量名.方法名();

* 普通类多态定义的格式

父类 变量名 = new 子类();

如： class Fu {}

class Zi extends Fu {}

//类的多态使用

Fu f = new Zi();

* 抽象类多态定义的格式

抽象类 变量名 = new 抽象类子类();

如： abstract class Fu {

public abstract void method();

}

class Zi extends Fu {

public void method(){

System.out.println(“重写父类抽象方法”);

}

}

//类的多态使用

Fu fu= new Zi();

* 接口多态定义的格式

接口 变量名 = new 接口实现类();

如： interface Fu {

public abstract void method();

}

class Zi implements Fu {

public void method(){

System.out.println(“重写接口抽象方法”);

}

}

//接口的多态使用

Fu fu = new Zi();

* 注意事项

同一个父类的方法会被不同的子类重写。在调用方法时，调用的为各个子类重写后的方法。

如 Person p1 = new Student();

Person p2 = new Teacher();

p1.work(); //p1会调用Student类中重写的work方法

p2.work(); //p2会调用Teacher类中重写的work方法

当变量名指向不同的子类对象时，由于每个子类重写父类方法的内容不同，所以会调用不同的方法。

## 多态-成员的特点

掌握了多态的基本使用后，那么多态出现后类的成员有啥变化呢？前面学习继承时，我们知道子父类之间成员变量有了自己的特定变化，那么当多态出现后，成员变量在使用上有没有变化呢？

多态出现后会导致子父类中的成员变量有微弱的变化。看如下代码

class Fu {

int num = 4;

}

class Zi extends Fu {

int num = 5;

}

class Demo {

public static void main(String[] args) {

Fu f = new Zi();

System.out.println(f.num);

Zi z = new Zi();

System.out.println(z.num);

}

}

* 多态成员变量

当子父类中出现同名的成员变量时，多态调用该变量时：

编译时期：参考的是引用型变量所属的类中是否有被调用的成员变量。没有，编译失败。

运行时期：也是调用引用型变量所属的类中的成员变量。

简单记：编译和运行都参考等号的左边。编译运行看左边。

多态出现后会导致子父类中的成员方法有微弱的变化。看如下代码

class Fu {

int num = 4;

void show() {

System.out.println("Fu show num");

}

}

class Zi extends Fu {

int num = 5;

void show() {

System.out.println("Zi show num");

}

}

class Demo {

public static void main(String[] args) {

Fu f = new Zi();

f.show();

}

}

* 多态成员方法

编译时期：参考引用变量所属的类，如果没有类中没有调用的方法，编译失败。

运行时期：参考引用变量所指的对象所属的类，并运行对象所属类中的成员方法。

简而言之：编译看左边，运行看右边。

## instanceof关键字

我们可以通过instanceof关键字来判断某个对象是否属于某种数据类型。如学生的对象属于学生类，学生的对象也属于人类。

使用格式：

boolean b = 对象 instanceof 数据类型;

如

Person p1 = new Student(); // 前提条件，学生类已经继承了人类

boolean flag = p1 instanceof Student; //flag结果为true

boolean flag2 = p2 instanceof Teacher; //flag结果为false

## 多态-转型

多态的转型分为向上转型与向下转型两种：

* 向上转型：当有子类对象赋值给一个父类引用时，便是向上转型，多态本身就是向上转型的过程。

使用格式：

父类类型 变量名 = new 子类类型();

如：Person p = new Student();

* 向下转型：一个已经向上转型的子类对象可以使用强制类型转换的格式，将父类引用转为子类引用，这个过程是向下转型。如果是直接创建父类对象，是无法向下转型的！

使用格式：

子类类型 变量名 = (子类类型) 父类类型的变量;

如:Student stu = (Student) p; //变量p 实际上指向Student对象

## 多态的好处与弊端

当父类的引用指向子类对象时，就发生了向上转型，即把子类类型对象转成了父类类型。向上转型的好处是隐藏了子类类型，提高了代码的扩展性。

但向上转型也有弊端，只能使用父类共性的内容，而无法使用子类特有功能，功能有限制。看如下代码

//描述动物类，并抽取共性eat方法

**abstract** **class** Animal {

**abstract** **void** eat();

}

// 描述狗类，继承动物类，重写eat方法，增加lookHome方法

**class** Dog **extends** Animal {

**void** eat() {

System.***out***.println("啃骨头");

}

**void** lookHome() {

System.***out***.println("看家");

}

}

// 描述猫类，继承动物类，重写eat方法，增加catchMouse方法

**class** Cat **extends** Animal {

**void** eat() {

System.***out***.println("吃鱼");

}

**void** catchMouse() {

System.***out***.println("抓老鼠");

}

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

Animal a = **new** Dog(); //多态形式，创建一个狗对象

a.eat(); // 调用对象中的方法，会执行狗类中的eat方法

// a.lookHome();//使用Dog类特有的方法，需要向下转型，不能直接使用

// 为了使用狗类的lookHome方法，需要向下转型

// 向下转型过程中，可能会发生类型转换的错误，即ClassCastException异常

// 那么，在转之前需要做健壮性判断

if( !a instanceof Dog){ // 判断当前对象是否是Dog类型

System.out.println("类型不匹配，不能转换");

return;

}

Dog d = (Dog) a; //向下转型

d.lookHome();//调用狗类的lookHome方法

}

}

我们来总结一下：

* 什么时候使用向上转型：

当不需要面对子类类型时，通过提高扩展性，或者使用父类的功能就能完成相应的操作，这时就可以使用向上转型。

如：Animal a = **new** Dog();

a.eat();

* 什么时候使用向下转型

当要使用子类特有功能时，就需要使用向下转型。

如：Dog d = (Dog) a; //向下转型

d.lookHome();//调用狗类的lookHome方法

* 向下转型的好处：可以使用子类特有功能。
* 弊端是：需要面对具体的子类对象；在向下转型时容易发生ClassCastException类型转换异常。在转换之前必须做类型判断。

如：if( !a instanceof Dog){…}

## 多态-举例

我们明确多态使用，以及多态的细节问题后，接下来练习下多态的应用。

* 毕老师和毕姥爷的故事

/\*

描述毕老师和毕姥爷，

毕老师拥有讲课和看电影功能

毕姥爷拥有讲课和钓鱼功能

\*/

**class** 毕姥爷 {

**void** 讲课() {

System.***out***.println("政治");

}

**void** 钓鱼() {

System.***out***.println("钓鱼");

}

}

// 毕老师继承了毕姥爷，就有拥有了毕姥爷的讲课和钓鱼的功能，

// 但毕老师和毕姥爷的讲课内容不一样，因此毕老师要覆盖毕姥爷的讲课功能

class 毕老师 extends 毕姥爷 {

**void** 讲课() {

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

}

**void** 看电影() {

System.***out***.println("看电影");

}

}

public class Test {

**public** **static** **void** main(String[] args) {

// 多态形式

毕姥爷 a = **new** 毕老师(); // 向上转型

a.讲课(); // 这里表象是毕姥爷，其实真正讲课的仍然是毕老师，因此调用的也是毕老师的讲课功能

a.钓鱼(); // 这里表象是毕姥爷，但对象其实是毕老师，而毕老师继承了毕姥爷，即毕老师也具有钓鱼功能

// 当要调用毕老师特有的看电影功能时，就必须进行类型转换

毕老师 b = (毕老师) a; // 向下转型

b.看电影();

}

}

学习到这里，面向对象的三大特征学习完了。

总结下封装、继承、多态的作用：

* 封装：把对象的属性与方法的实现细节隐藏，仅对外提供一些公共的访问方式
* 继承：子类会自动拥有父类所有可继承的属性和方法。
* 多态：配合继承与方法重写提高了代码的复用性与扩展性；如果没有方法重写，则多态同样没有意义。

# 笔记本电脑案例

## 案例介绍

定义USB接口（具备开启功能、关闭功能），笔记本要使用USB设备，即笔记本在生产时需要预留可以插入USB设备的USB接口，即就是笔记本具备使用USB设备的功能，但具体是什么USB设备，笔记本并不关心，只要符合USB规格的设备都可以。鼠标和键盘要想能在电脑上使用，那么鼠标和键盘也必须遵守USB规范，不然鼠标和键盘的生产出来无法使用

进行描述笔记本类，实现笔记本使用USB鼠标、USB键盘

* USB接口，包含开启功能、关闭功能
* 笔记本类，包含运行功能、关机功能、使用USB设备功能
* 鼠标类，要符合USB接口
* 键盘类，要符合USB接口

## 案例需求分析

阶段一：

使用笔记本，笔记本有运行功能，需要笔记本对象来运行这个功能

阶段二：

想使用一个鼠标，又有一个功能使用鼠标，并多了一个鼠标对象。

阶段三：

还想使用一个键盘 ，又要多一个功能和一个对象

问题：每多一个功能就需要在笔记本对象中定义一个方法，不爽，程序扩展性极差。

降低鼠标、键盘等外围设备和笔记本电脑的耦合性。

## 实现代码步骤

* 定义鼠标、键盘，笔记本三者之间应该遵守的规则

interface USB {

void open();// 开启功能

void close();// 关闭功能

}

* 鼠标实现USB规则

class Mouse implements USB {

public void open() {

System.*out*.println("鼠标开启");

}

public void close() {

System.*out*.println("鼠标关闭");

}

}

* 键盘实现USB规则

class KeyBoard implements USB {

public void open() {

System.*out*.println("键盘开启");

}

public void close() {

System.*out*.println("键盘关闭");

}

}

* 定义笔记本

class NoteBook {

// 笔记本开启运行功能

public void run() {

System.*out*.println("笔记本运行");

}

// 笔记本使用usb设备，这时当笔记本对象调用这个功能时，必须给其传递一个符合USB规则的USB设备

public void useUSB(USB usb) {

// 判断是否有USB设备

if (usb != null) {

usb.open();

usb.close();

}

}

public void shutDown() {

System.*out*.println("笔记本关闭");

}

}

public class Test {

public static void main(String[] args) {

// 创建笔记本实体对象

NoteBook nb = new NoteBook();

// 笔记本开启

nb.run();

// 创建鼠标实体对象

Mouse m = new Mouse();

// 笔记本使用鼠标

nb.useUSB(m);

// 创建键盘实体对象

KeyBoard kb = new KeyBoard();

// 笔记本使用键盘

nb.useUSB(kb);

// 笔记本关闭

nb.shutDown();

}

}

# 总结

## 知识点总结

* 接口：理解为是一个特殊的抽象类，但它不是类，是一个接口
  + 接口的特点：

1，定义一个接口用interface关键字

interface Inter{}

2，一个类实现一个接口，实现implements关键字

class Demo implements Inter{}

3, 接口不能直接创建对象

通过多态的方式，由子类来创建对象，接口多态

* + 接口中的成员特点：

成员变量：

只能是final 修饰的常量

默认修饰符： public static final

构造方法：

无

成员方法：

只能是抽象方法

默认修饰符: public abstract

* + 类与类，类与接口，接口与接口之间的关系

类与类之间：继承关系，单继承，可以是多层继承

类与接口之间: 实现关系，单实现，也可以多实现

接口与接口之间：继承关系，单继承，也可以是多继承

Java中的类可以继承一个父类的同时，实现多个接口

* 多态：理解为同一种物质的多种形态
  + 多态使用的前提：

1，有继承或者实现关系

2，要方法重写

3，父类引用指向子类对象

* + 多态的成员访问特点：

方法的运行看右边，其他都看左边

* + 多态的好处：

提高了程序的扩展性

* + 多态的弊端：

不能访问子类的特有功能

* + 多态的分类
    - 类的多态

abstract class Fu {

public abstract void method();

}

class Zi extends Fu {

public void method(){

System.out.println(“重写父类抽象方法”);

}

}

//类的多态使用

Fu fu= new Zi();

* + - 接口的多态

interface Fu {

public abstract void method();

}

class Zi implements Fu {

public void method(){

System.out.println(“重写接口抽象方法”);

}

}

//接口的多态使用

Fu fu = new Zi();

* instanceof 关键字

格式： 对象名 instanceof 类名

返回值： true, false

作用： 判断指定的对象 是否为 给定类创建的对象

# 第12天面向对象

今日内容介绍

* 构造方法
* this
* super

# 构造方法

我们对封装已经有了基本的了解，接下来我们来看一个新的问题，依然以Person为例，由于Person中的属性都被private了，外界无法直接访问属性，必须对外提供相应的set和get方法。当创建人对象的时候，人对象一创建就要明确其姓名和年龄，那该怎么做呢？

## 构造方法介绍

在开发中经常需要在创建对象的同时明确对象的属性值，比如员工入职公司就要明确他的姓名、年龄等属性信息。

那么，创建对象就要明确属性值，那怎么解决呢？也就是在创建对象的时候就要做的事情，当使用new关键字创建对象时，怎么给对象的属性初始化值呢？这就要学习Java另外一门小技术，构造方法。

那什么是构造方法呢？从字面上理解即为构建创造时用的方法，即就是对象创建时要执行的方法。既然是对象创建时要执行的方法，那么只要在new对象时，知道其执行的构造方法是什么，就可以在执行这个方法的时候给对象进行属性赋值。

* 构造方法的格式：

修饰符 构造方法名(参数列表)

{

}

* 构造方法的体现：
  + 构造方法没有返回值类型。也不需要写返回值。因为它是为构建对象的，对象创建完，方法就执行结束。
  + 构造方法名称必须和类型保持一致。
  + 构造方法没有具体的返回值。
* 构造方法的代码体现：

**class** Person {

// Person的成员属性age和name

private int age;

**private** String name;

// Person的构造方法，拥有参数列表

Person(**int** a, String nm) {

// 接受到创建对象时传递进来的值，将值赋给成员属性

age = a;

name = nm;

}

}

## 构造方法调用和内存图解

理解构造方法的格式和基本功能之后，现在就要研究构造方法是怎么执行的呢？在创建对象的时候是如何初始化的呢？

构造方法是专门用来创建对象的，也就是在new对象时要调用构造方法。现在来看看如何调用构造方法。

**class** Person {

// Person的成员属性age和name

**private** **int** age;

**private** String name;

// Person的构造方法，拥有参数列表

Person(**int** a, String nm) {

// 接受到创建对象时传递进来的值，将值赋给成员属性

age = a;

name = nm;

}

**public** **void** speak() {

System.*out*.println("name=" + name + ",age=" + age);

}

}

**class** PersonDemo {

**public** **static** **void** main(String[] args) {

// 创建Person对象，并明确对象的年龄和姓名

Person p2 = **new** Person(23, "张三");

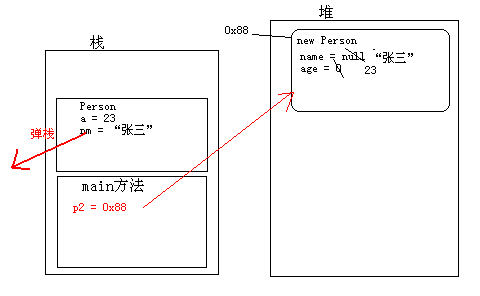
p2.speak();

}

}

上述代码演示了创建对象时构造方法的调用。即在创建对象时，会调用与参数列表对应的构造方法。

上述代码的图解：



* 图解说明：

1. 首先会将main方法压入栈中，执行main方法中的 new Person(23,"张三");
2. 在堆内存中分配一片区域，用来存放创建的Person对象，这片内存区域会有属于自己的内存地址（0x88）。然后给成员变量进行默认初始化（name=null，age=0）。
3. 执行构造方法中的代码（age = a ; name = nm;）,将变量a对应的23赋值给age，将变量nm对应的”张三赋值给name，这段代码执行结束后，成员变量age和name的值已经改变。执行结束之后构造方法弹栈，Person对象创建完成。将Person对象的内存地址0x88赋值给p2。

## 默认构造方法和细节

在没有学习构造方法之前，我们也可以通过new关键字创建对象，并调用相应的方法，同时在描述事物时也没有写构造方法。这是为什么呢？

在之前学习的过程中，描述事物时，并没有显示指定构造方法，当在编译Java文件时，编译器会自动给class文件中添加默认的构造方法。如果在描述类时，我们显示指定了构造方法，那么，当在编译Java源文件时，编译器就不会再给class文件中添加默认构造方法。

class Person {

//如果没有显示指定构造方法，编译会在编译时自动添加默认的构造方法

//Person(){} //空参数的默认构造方法

}

当在描述事物时，要不要在类中写构造方法呢？这时要根据描述事物的特点来确定，当描述的事物在创建其对象时就要明确属性的值，这时就需要在定义类的时候书写带参数的构造方法。若创建对象时不需要明确具体的数据，这时可以不用书写构造方法（不书写也有默认的构造方法）。

* 构造方法的细节：

1. 一个类中可以有多个构造方法，多个构造方法是以重载的形式存在的
2. 构造方法是可以被private修饰的，作用：其他程序无法创建该类的对象。

**class** Person {

**private** **int** age;

**private** String name;

// 私有无参数的构造方法，即外界不能通过new Person();语句创建本类对象

**private** Person() {

}

// 多个构造方法是以重载的形式存在

Person(**int** a) {

age = a;

}

Person(String nm, **int** a) {

name = nm;

age = a;

}

}

## 构造方法和一般方法区别

到目前为止，学习两种方法，分别为构造方法和一般方法，那么他们之间有什么异同呢？

构造方法在对象创建时就执行了，而且只执行一次。

一般方法是在对象创建后，需要使用时才被对象调用，并可以被多次调用。

* 问题：

有了构造方法之后可以对对象的属性进行初始化，那么还需要对应的set和get方法吗？

需要相应的set和get方法，因为对象在创建之后需要修改和访问相应的属性值时，在这时只能通过set或者get方法来操作。

思考，如下代码有问题吗？

**class** Person {

**void** Person() {

}

}

**class** PersonDemo {

**public** **static** **void** main(String[] args) {

Person p = **new** Person();

}

}

# this关键字

在之前学习方法时，我们知道方法之间是可以相互调用的，那么构造方法之间能不能相互调用呢？若可以，怎么调用呢？

## this调用构造方法

在之前学习方法之间调用时，可以通过方法名进行调用。可是针对构造方法，无法通过构造方法名来相互调用。

构造方法之间的调用，可以通过this关键字来完成。

* 构造方法调用格式：

this(参数列表);

* 构造方法的调用

**class** Person {

// Person的成员属性

**private** **int** age;

**private** String name;

// 无参数的构造方法

Person() {

}

// 给姓名初始化的构造方法

Person(String nm) {

name = nm;

}

// 给姓名和年龄初始化的构造方法

Person(String nm, **int** a) {

// 由于已经存在给姓名进行初始化的构造方法 name = nm;因此只需要调用即可

// 调用其他构造方法，需要通过this关键字来调用

**this**(nm);

// 给年龄初始化

age = a;

}

}

## this的原理图解

了解了构造方法之间是可以相互调用，那为什么他们之间通过this就可以调用呢？

通过上面的学习，简单知道使用this可以实现构造方法之间的调用，但是为什么就会知道this调用哪一个构造方法呢？接下来需要图解完成。

**class** Person {

**private** **int** age;

**private** String name;

Person() {

}

Person(String nm) {

name = nm;

}

Person(String nm, **int** a) {

**this**(nm);

age = a;

}

}

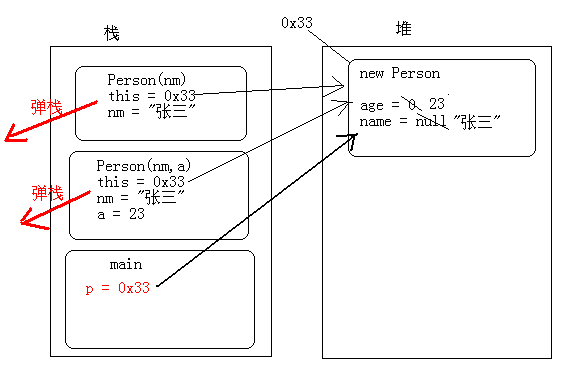
**class** PersonDemo {

**public** **static** **void** main(String[] args) {

Person p = **new** Person("张三", 23);

}

}



* 图列说明：

1. 先执行main方法，main方法压栈，执行其中的new Person(“张三”,23);
2. 堆内存中开辟空间，并为其分配内存地址0x33，，紧接着成员变量默认初始化（name=null age = 0）；
3. 拥有两个参数的构造方法（Person（String nm , int a））压栈，在这个构造方法中有一个隐式的this，因为构造方法是给对象初始化的，那个对象调用到这个构造方法，this就指向堆中的那个对象。
4. 由于Person（String nm , int a）构造方法中使用了this(nm);构造方法Person(String nm)就会压栈，并将“张三”传递给nm。在Person（String nm , int a）构造方法中同样也有隐式的this，this的值同样也为0x33，这时会执行其中name = nm，即把“张三”赋值给成员的name。当赋值结束后Person（String nm , int a）构造方法弹栈。
5. 程序继续执行构造方法（Person（String nm , int a）中的age = a；这时会将23赋值给成员属性age。赋值结束构造方法（Person（String nm , int a）弹栈。
6. 当构造方法（Person（String nm , int a）弹栈结束后，Person对象在内存中创建完成，并将0x33赋值给main方法中的p引用变量。

* 注意：

this到底代表什么呢？this代表的是对象，具体代表哪个对象呢？哪个对象调用了this所在的方法，this就代表哪个对象。

调用其他构造方法的语句必须定义在构造方法的第一行，原因是初始化动作要最先执行。

## 成员变量和局部变量同名问题

通过上面学习，基本明确了对象初始化过程中的细节，也知道了构造方法之间的调用是通过this关键字完成的。但this也有另外一个用途，接下来我们就学习下。

当在方法中出现了局部变量和成员变量同名的时候，那么在方法中怎么区别局部变量成员变量呢？可以在成员变量名前面加上this.来区别成员变量和局部变量

**class** Person {

**private** **int** age;

**private** String name;

// 给姓名和年龄初始化的构造方法

Person(String name, **int** age) {

// 当需要访问成员变量是，只需要在成员变量前面加上this.即可

**this**.name = name;

**this**.age = age;

}

**public** **void** speak() {

System.*out*.println("name=" + **this**.name + ",age=" + **this**.age);

}

}

**class** PersonDemo {

**public** **static** **void** main(String[] args) {

Person p = **new** Person("张三", 23);

p.speak();

}

}

## this的应用

学习完了构造方法、this的用法之后，现在做个小小的练习。

需求：在Person类中定义功能，判断两个人是否是同龄人

**class** Person {

**private** **int** age;

**private** String name;

// 给姓名和年龄初始化的构造方法

Person(String name, **int** age) {

// 当需要访问成员变量是，只需要在成员变量前面加上this.即可

**this**.name = name;

**this**.age = age;

}

**public** **void** speak() {

System.*out*.println("name=" + **this**.name + ",age=" + **this**.age);

}

// 判断是否为同龄人

**public** **boolean** equalsAge(Person p) {

// 使用当前调用该equalsAge方法对象的age和传递进来p的age进行比较

// 由于无法确定具体是哪一个对象调用equalsAge方法，这里就可以使用this来代替

/\*

\* if(this.age == p.age) { return true; } return false;

\*/

**return** **this**.age = p.age;

}

}

# super关键字

## 子父类中构造方法的调用

在创建子类对象时，父类的构造方法会先执行，因为子类中所有构造方法的第一行有默认的隐式super();语句。

格式：

调用本类中的构造方法

this(实参列表);

调用父类中的空参数构造方法

super();

调用父类中的有参数构造方法

super(实参列表);

为什么子类对象创建都要访问父类中的构造方法？因为子类继承了父类的内容，所以创建对象时，必须要先看父类是如何对其内容进行初始化的，看如下程序：

public class Test {

public static void main(String[] args) {

new Zi();

}

}

class Fu{

int num ;

Fu(){

System.out.println("Fu构造方法"+num);

num = 4;

}

}

class Zi extends Fu{

Zi(){

//super(); 调用父类空参数构造方法

System.out.println("Zi构造方法"+num);

}

}

　　执行结果：

　　 Fu构造方法0

　　 Zi构造方法4

通过结果发现，子类构造方法执行时中，调用了父类构造方法，这说明，子类构造方法中有一句super()。

那么，子类中的构造方法为什么会有一句隐式的super()呢？

原因：子类会继承父类中的内容，所以子类在初始化时，必须先到父类中去执行父类的初始化动作。这样，才可以使用父类中的内容。

当父类中没有空参数构造方法时，子类的构造方法必须有显示的super语句，指定要访问的父类有参数构造方法。

## 子类对象创建过程的细节

如果子类的构造方法第一行写了this调用了本类其他构造方法，那么super调用父类的语句还有吗？

这时是没有的，因为this()或者super(),只能定义在构造方法的第一行，因为初始化动作要先执行。

父类构造方法中是否有隐式的super呢？

也是有的。记住：只要是构造方法默认第一行都是super();

父类的父类是谁呢？super调用的到底是谁的构造方法呢？

Java体系在设计，定义了一个所有对象的父类Object

* 注意：

类中的构造方法默认第一行都有隐式的super()语句，在访问父类中的空参数构造方法。所以父类的构造方法既可以给自己的对象初始化，也可以给自己的子类对象初始化。

如果默认的隐式super()语句在父类中没有对应的构造方法，那么必须在构造方法中通过this或者super的形式明确要调用的构造方法。

## super应用

练习：描述学生和工人这两个类，将他们的共性name和age抽取出来存放在父类中，并提供相应的get和set方法，同时需要在创建学生和工人对象就必须明确姓名和年龄

//定义Person类，将Student和Worker共性抽取出来

class Person {

private String name;

private int age;

public Person(String name, int age) {

// super();

this.name = name;

this.age = 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;

}

}

class Student extends Person {

// Student类的构造方法

Student(String name, int age) {

// 使用super关键字调用父类构造方法，进行相应的初始化动作

super(name, age);

}

public void study() {// Studnet中特有的方法

System.out.println(this.getName() + "同学在学习");

}

}

class Worker extends Person {

Worker(String name, int age) {

// 使用super关键字调用父类构造方法，进行相应的初始化动作

super(name, age);

}

public void work() {// Worker 中特有的方法

System.out.println(this.getName() + "工人在工作");

}

}

public class Test {

public static void main(String[] args) {

Student stu = new Student("小明",23);

stu.study();

Worker w = new Worker("小李",45);

w.work();

}

}

# 综合案例---完整的员工类

## 案例介绍

某IT公司有多名员工，按照员工负责的工作不同，进行了部门的划分（研发部员工、维护部员工）。研发部根据所需研发的内容不同，又分为JavaEE工程师、Android工程师；维护部根据所需维护的内容不同，又分为网络维护工程师、硬件维护工程师。

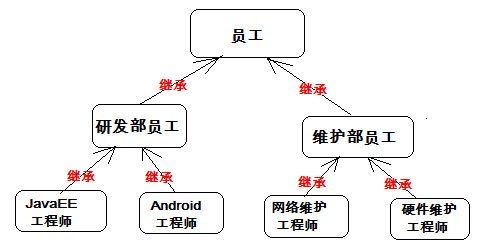
公司的每名员工都有他们自己的员工编号、姓名，并要做它们所负责的工作。

* 工作内容
  + JavaEE工程师：员工号为xxx的 xxx员工，正在研发淘宝网站
  + Android工程师：员工号为xxx的 xxx员工，正在研发淘宝手机客户端软件
  + 网络维护工程师：员工号为xxx的 xxx员工，正在检查网络是否畅通
  + 硬件维护工程师：员工号为xxx的 xxx员工，正在修复打印机

请根据描述，完成员工体系中所有类的定义，并指定类之间的继承关系。进行XX工程师类的对象创建，完成工作方法的调用。

## 案例分析

* 根据上述部门的描述，得出如下的员工体系图



* 根据员工信息的描述，确定每个员工都有员工编号、姓名、要进行工作。则，把这些共同的属性与功能抽取到父类中（员工类），关于工作的内容由具体的工程师来进行指定。
  + 工作内容
    - JavaEE工程师：员工号为xxx的 xxx员工，正在研发淘宝网站
    - Android工程师：员工号为xxx的 xxx员工，正在研发淘宝手机客户端软件
    - 网络维护工程师：员工号为xxx的 xxx员工，正在检查网络是否畅通
    - 硬件维护工程师：员工号为xxx的 xxx员工，正在修复打印机
* 创建JavaEE工程师对象，完成工作方法的调用

## 案例代码实现

* 根据员工体系图，完成类的定义

定义员工类(抽象类)

**public** **abstract** **class** Employee {

**private** String id;// 员工编号

**private** String name; // 员工姓名

//空参数构造方法

**public** Employee() {

**super**();

}

//有参数构造方法

**public** Employee(String id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

**public** String getId() {

**return** id;

}

**public** **void** setId(String id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

//工作方法（抽象方法）

**public** **abstract** **void** work();

}

* 定义研发部员工类Developer 继承 员工类Employee

**public** **abstract** **class** Developer **extends** Employee {

//空参数构造方法

**public** Developer() {

**super**();

}

//有参数构造方法

**public** Developer(String id, String name) {

**super**(id, name);

}

}

* 定义维护部员工类Maintainer 继承 员工类Employee

**public** **abstract** **class** Maintainer **extends** Employee {

//空参数构造方法

**public** Maintainer() {

**super**();

}

//有参数构造方法

**public** Maintainer(String id, String name) {

**super**(id, name);

}

}

* 定义JavaEE工程师 继承 研发部员工类，重写工作方法

**public** **class** JavaEE **extends** Developer {

//空参数构造方法

**public** JavaEE() {

**super**();

}

//有参数构造方法

**public** JavaEE(String id, String name) {

**super**(id, name);

}

@Override

**public** **void** work() {

System.***out***.println("员工号为 " + getId() + " 的 " + getName() + " 员工，正在研发淘宝网站");

}

}

* 定义Android工程师 继承 研发部员工类，重写工作方法

**public** **class** Android **extends** Developer {

//空参数构造方法

**public** Android() {

**super**();

}

//有参数构造方法

**public** Android(String id, String name) {

**super**(id, name);

}

@Override

**public** **void** work() {

System.***out***.println("员工号为 " + getId() + " 的 " + getName() + " 员工，正在研发淘宝手机客户端软件");

}

}

* 定义Network网络维护工程师 继承 维护部员工类，重写工作方法

**public** **class** Network **extends** Maintainer {

//空参数构造方法

**public** Network() {

**super**();

}

//有参数构造方法

**public** Network(String id, String name) {

**super**(id, name);

}

@Override

**public** **void** work() {

System.***out***.println("员工号为 " + getId() + " 的 " + getName() + " 员工，正在检查网络是否畅通");

}

}

* 定义Hardware硬件维护工程师 继承 维护部员工类，重写工作方法

**public** **class** Hardware **extends** Maintainer {

//空参数构造方法

**public** Hardware() {

**super**();

}

//有参数构造方法

**public** Hardware(String id, String name) {

**super**(id, name);

}

@Override

**public** **void** work() {

System.***out***.println("员工号为 " + getId() + " 的 " + getName() + " 员工，正在修复打印机");

}

}

* 在测试类中，创建JavaEE工程师对象，完成工作方法的调用

public class Test {

**public** **static** **void** main(String[] args) {

//创建JavaEE工程师员工对象，该员工的编号000015，员工的姓名 小明

JavaEE ee = **new** JavaEE("000015", "小明");

//调用该员工的工作方法

ee.work();

}

}

# 总结

## 知识点总结

* this关键字
  + this关键字，本类对象的引用
    - this是在方法中使用的，哪个对象调用了该方法，那么，this就代表调用该方法的对象引用
    - this什么时候存在的？当创建对象的时候，this存在的
    - this的作用：用来区别同名的成员变量与局部变量（this.成员变量）

public void setName(String name) {

this.name = name;

}

* 构造方法： 用来给类的成员进行初始化操作
  + 格式：

修饰符 类名 (参数列表) {

...

}

* + 构造方法的特点：
    - 1, 方法名与类名相同
    - 2，没有返回值，也没有返回值类型，连void也没有
  + 构造方法什么时候会被调用执行？

只有在创建对象的时候才可以被调用

* super: 指的是父类的存储空间(理解为父类的引用)

调用父类的成员变量：

super.成员变量;

调用父类的构造方法:

super(参数);

调用方法的成员方法:

super.成员方法();

* 继承中的构造方法注意事项：

1，如果我们手动给出了构造方法，编译器不会在给我们提供默认的空参数构造方法

如果我们没写任何的构造方法，编译器提供给我们一个空参数构造方法

2, 在构造方法中，默认的第一条语句为 super();

它是用来访问父类中的空参数构造方法，进行父类成员的初始化操作

3, 当父类中没有空参数构造方法的时候，怎么办？

a: 通过 super(参数) 访问父类有参数的构造方法

b: 通过 this(参数) 访问本类中其他构造方法

注意:[本类中的其他构造方法已经能够正常访问父类构造方法]

4, super(参数) 与 this(参数) 不能同时在构造方法中存在

# 第13天面向对象

今日内容介绍

* final
* static
* 匿名对象
* 内部类
* 包的声明与访问
* 四种访问修饰符
* 代码块

# final关键字

## final的概念

继承的出现提高了代码的复用性，并方便开发。但随之也有问题，有些类在描述完之后，不想被继承，或者有些类中的部分方法功能是固定的，不想让子类重写。可是当子类继承了这些特殊类之后，就可以对其中的方法进行重写，那怎么解决呢？

要解决上述的这些问题，需要使用到一个关键字final，final的意思为最终，不可变。final是个修饰符，它可以用来修饰类，类的成员，以及局部变量。

## final的特点

* final修饰类不可以被继承，但是可以继承其他类。

**class** Yy {}

**final** **class** Fu **extends** Yy{} //可以继承Yy类

**class** Zi **extends** Fu{} //不能继承Fu类

* final修饰的方法不可以被覆盖,但父类中没有被final修饰方法，子类覆盖后可以加final。

**class** Fu {

// final修饰的方法，不可以被覆盖，但可以继承使用

**public** **final** **void** method1(){}

**public** **void** method2(){}

}

**class** Zi **extends** Fu {

//重写method2方法

**public** **final** **void** method2(){}

}

* final修饰的变量称为常量，这些变量只能赋值一次。

**final** **int** i = 20;

i = 30; //赋值报错，final修饰的变量只能赋值一次

* 引用类型的变量值为对象地址值，地址值不能更改，但是地址内的对象属性值可以修改。

**final** Person p = **new** Person();

Person p2 = **new** Person();

p = p2; //final修饰的变量p，所记录的地址值不能改变

p.name = "小明";//可以更改p对象中name属性值

p不能为别的对象，而p对象中的name或age属性值可更改。

* 修饰成员变量，需要在创建对象前赋值，否则报错。(当没有显式赋值时，多个构造方法的均需要为其赋值。)

**class** Demo {

//直接赋值

**final** **int** m = 100;

//final修饰的成员变量，需要在创建对象前赋值，否则报错。

**final** **int** n;

**public** Demo(){

//可以在创建对象时所调用的构造方法中，为变量n赋值

n = 2016;

}

}

# static关键字

## static概念

当在定义类的时候，类中都会有相应的属性和方法。而属性和方法都是通过创建本类对象调用的。当在调用对象的某个方法时，这个方法没有访问到对象的特有数据时，方法创建这个对象有些多余。可是不创建对象，方法又调用不了，这时就会想，那么我们能不能不创建对象，就可以调用方法呢？

可以的，我们可以通过static关键字来实现。static它是静态修饰符，一般用来修饰类中的成员。

## static特点

* 被static修饰的成员变量属于类，不属于这个类的某个对象。（也就是说，多个对象在访问或修改static修饰的成员变量时，其中一个对象将static成员变量值进行了修改，其他对象中的static成员变量值跟着改变，即多个对象共享同一个static成员变量）

代码演示：

**class** Demo {

**public** **static** **int** *num* = 100;

}

**class** Test {

**public** **static** **void** main(String[] args) {

Demo d1 = **new** Demo();

Demo d2 = **new** Demo();

d1.*num* = 200;

System.***out***.println(d1.*num*); //结果为200

System.***out***.println(d2.*num*); //结果为200

}

}

* 被static修饰的成员可以并且建议通过类名直接访问。

访问静态成员的格式：

类名.静态成员变量名

类名.静态成员方法名(参数)

对象名.静态成员变量名 ------不建议使用该方式，会出现警告

对象名.静态成员方法名(参数) ------不建议使用该方式，会出现警告

代码演示：

**class** Demo {

//静态成员变量

**public** **static** **int** *num* = 100;

//静态方法

**public** **static** **void** method(){

System.***out***.println("静态方法");

}

}

**class** Test {

**public** **static** **void** main(String[] args) {

System.***out***.println(Demo.*num*);

Demo.*method*();

}

}

## static注意事项

* 静态内容是优先于对象存在，只能访问静态，不能使用this/super。静态修饰的内容存于静态区。

**class** Demo {

//成员变量

**public** **int** num = 100;

//静态方法

**public** **static** **void** method(){

//this.num; 不能使用this/super。

System.***out***.println(**this**.num);

}

}

* 同一个类中，静态成员只能访问静态成员

**class** Demo {

//成员变量

**public** **int** num = 100;

//静态成员变量

**public** **static** **int** *count* = 200;

//静态方法

**public** **static** **void** method(){

//System.out.println(num); 静态方法中，只能访问静态成员变量或静态成员方法

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

}

}

* main方法为静态方法仅仅为程序执行入口，它不属于任何一个对象，可以定义在任意类中。

## 定义静态常量

开发中，我们想在类中定义一个静态常量，通常使用public static final修饰的变量来完成定义。此时变量名用全部大写，多个单词使用下划线连接。

定义格式：

public static final 数据类型 变量名 = 值;

如下演示：

**class** Company {

**public** **static** **final** String ***COMPANY\_NAME*** = "传智播客";

**public** **static** **void** method(){

System.***out***.println("一个静态方法");

}

}

当我们想使用类的静态成员时，不需要创建对象，直接使用类名来访问即可。

System.***out***.println(Company.***COMPANY\_NAME***); //打印传智播客

Company.*method*(); // 调用一个静态方法

* 注意：

接口中的每个成员变量都默认使用public static final修饰。

所有接口中的成员变量已是静态常量，由于接口没有构造方法，所以必须显示赋值。可以直接用接口名访问。

interface Inter {

public static final int *COUNT* = 100;

}

访问接口中的静态变量

Inter.COUNT

# 匿名对象

## 匿名对象的概念

匿名对象是指创建对象时，只有创建对象的语句，却没有把对象地址值赋值给某个变量。

如：已经存在的类：

public class Person{

public void eat(){

System.out.println();

}

}

创建一个普通对象

Person p = new Person();

创建一个匿名对象

new Person();

## 匿名对象的特点

* 创建匿名对象直接使用，没有变量名。

new Person().eat() //eat方法被一个没有名字的Person对象调用了。

* 匿名对象在没有指定其引用变量时，只能使用一次。

new Person().eat(); 创建一个匿名对象，调用eat方法

new Person().eat(); 想再次调用eat方法，重新创建了一个匿名对象

* 匿名对象可以作为方法接收的参数、方法返回值使用

**class** Demo {

**public** **static** Person getPerson(){

//普通方式

//Person p = new Person();

//return p;

//匿名对象作为方法返回值

**return** **new** Person();

}

**public** **static** **void** method(Person p){}

}

**class** Test {

**public** **static** **void** main(String[] args) {

//调用getPerson方法，得到一个Person对象

Person person = Demo.*getPerson*();

//调用method方法

Demo.*method*(person);

//匿名对象作为方法接收的参数

Demo.*method*(**new** Person());

}

}

# 内部类

## 内部类概念

* 什么是内部类

将类写在其他类的内部，可以写在其他类的成员位置和局部位置，这时写在其他类内部的类就称为内部类。其他类也称为外部类。

* 什么时候使用内部类

在描述事物时，若一个事物内部还包含其他可能包含的事物，比如在描述汽车时，汽车中还包含这发动机，这时发动机就可以使用内部类来描述。

class 汽车 { //外部类

class 发动机 { //内部类

}

}

* 内部类的分类

内部类分为成员内部类与局部内部类。

我们定义内部类时，就是一个正常定义类的过程，同样包含各种修饰符、继承与实现关系等。在内部类中可以直接访问外部类的所有成员。

## 成员内部类

成员内部类，定义在外部类中的成员位置。与类中的成员变量相似，可通过外部类对象进行访问

* 定义格式

class 外部类 {

修饰符 class 内部类 {

//其他代码

}

}

* 访问方式

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

* 成员内部类代码演示

定义类

class Body {//外部类，身体

private boolean life= true; //生命状态

public class Heart { //内部类，心脏

public void jump() {

System.out.println("心脏噗通噗通的跳")

System.out.println("生命状态" + life); //访问外部类成员变量

}

}

}

访问内部类

public static void main(String[] args) {

//创建内部类对象

Body.Heart bh = new Body().new Heart();

//调用内部类中的方法

bh.jump();

}

## 局部内部类

局部内部类，定义在外部类方法中的局部位置。与访问方法中的局部变量相似，可通过调用方法进行访问

* 定义格式

class 外部类 {

修饰符 返回值类型 方法名(参数) {

class 内部类 {

//其他代码

}

}

}

* 访问方式

在外部类方法中，创建内部类对象，进行访问

* 局部内部类代码演示

定义类

class Party {//外部类，聚会

public void puffBall(){// 吹气球方法

class Ball {// 内部类，气球

public void puff(){

System.out.println("气球膨胀了");

}

}

//创建内部类对象，调用puff方法

new Ball().puff();

}

}

访问内部类

public static void main(String[] args) {

//创建外部类对象

Party p = new Party();

//调用外部类中的puffBall方法

p.puffBall();

}

## 内部类的实际使用——匿名内部类

### 匿名内部类概念

内部类是为了应对更为复杂的类间关系。查看源代码中会涉及到，而在日常业务中很难遇到，这里不做赘述。

最常用到的内部类就是匿名内部类，它是局部内部类的一种。

定义的匿名内部类有两个含义：

* + 临时定义某一指定类型的子类
  + 定义后即刻创建刚刚定义的这个子类的对象

### 定义匿名内部类的作用与格式

**作用：**匿名内部类是创建某个类型子类对象的快捷方式。

**格式：**

new 父类或接口(){

//进行方法重写

};

* 代码演示

//已经存在的父类：

public abstract class Person{

public abstract void eat();

}

//定义并创建该父类的子类对象，并用多态的方式赋值给父类引用变量

Person p = new Person(){

public void eat() {

System.out.println(“我吃了”);

}

};

//调用eat方法

p.eat();

使用匿名对象的方式，将定义子类与创建子类对象两个步骤由一个格式一次完成，。虽然是两个步骤，但是两个步骤是连在一起完成的。

匿名内部类如果不定义变量引用，则也是匿名对象。代码如下：

new Person(){

public void eat() {

System.out.println(“我吃了”);

}

}；

# 包的声明与访问

## 包的概念

java的包，其实就是我们电脑系统中的文件夹，包里存放的是类文件。

当类文件很多的时候，通常我们会采用多个包进行存放管理他们，这种方式称为分包管理。

在项目中，我们将相同功能的类放到一个包中，方便管理。并且日常项目的分工也是以包作为边界。

类中声明的包必须与实际class文件所在的文件夹情况相一致，即类声明在a包下，则生成的.class文件必须在a文件夹下，否则，程序运行时会找不到类。

## 包的声明格式

通常使用公司网址反写，可以有多层包，包名采用全部小写字母，多层包之间用”.”连接

类中包的声明格式：

package 包名.包名.包名…;

如：黑马程序员网址itheima.com那么网址反写就为com.itheima

传智播客 itcast.cn 那么网址反写就为 cn.itcast

* 注意：声明包的语句，必须写在程序有效代码的第一行（注释不算）
* 代码演示：

package cn.itcast; //包的声明，必须在有效代码的第一行

import java.util.Scanner;

import java.util.Random;

public class Demo {}

## 包的访问

在访问类时，为了能够找到该类，必须使用含有包名的类全名（包名.类名）。

包名.包名….类名

如： java.util.Scanner

java.util.Random

cn.itcast.Demo

带有包的类，创建对象格式：包名.类名 变量名 = new包名.类名();

cn.itcast.Demo d = new cn.itcast.Demo();

* 前提：包的访问与访问权限密切相关，这里以一般情况来说，即类用public修饰的情况。
* 类的简化访问

当我们要使用一个类时，这个类与当前程序在同一个包中（即同一个文件夹中），或者这个类是java.lang包中的类时通常可以省略掉包名，直接使用该类。

如：cn.itcast包中有两个类，PersonTest类，与Person类。我们在PersonTest类中，访问Person类时，由于是同一个包下，访问时可以省略包名，即直接通过类名访问 Person。

类名 变量名 = new类名();

Person p = new Person();

* 当我们要使用的类，与当前程序不在同一个包中（即不同文件夹中），要访问的类必须用public修饰才可访问。

package cn.itcst02;

public class Person {}

## import导包

我们每次使用类时，都需要写很长的包名。很麻烦，我们可以通过import导包的方式来简化。

可以通过导包的方式使用该类，可以避免使用全类名编写（即，包类.类名）。

导包的格式：

import 包名.类名;

当程序导入指定的包后，使用类时，就可以简化了。演示如下

//导入包前的方式

//创建对象

java.util.Random r1 = new java.util.Random();

java.util.Random r2 = new java.util.Random();

java.util.Scanner sc1 = new java.util.Scanner(System.in);

java.util.Scanner sc2 = new java.util.Scanner(System.in);

//导入包后的方式

import java.util.Random;

import java.util.Scanner;

//创建对象

Random r1 = new Random();

Random r2 = new Random();

Scanner sc1 = new Scanner(System.in);

Scanner sc2 = new Scanner(System.in);

* import导包代码书写的位置：在声明包package后，定义所有类class前，使用导包import包名.包名.类名;

# 访问修饰符

在Java中提供了四种访问权限，使用不同的访问权限时，被修饰的内容会有不同的访问权限，以下表来说明不同权限的访问能力：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | public | protected | default | private |
| 同一类中 | √ | √ | √ | √ |
| 同一包中(子类与无关类) | √ | √ | √ |  |
| 不同包的子类 | √ | √ |  |  |
| 不同包中的无关类 | √ |  |  |  |

归纳一下：在日常开发过程中，编写的类、方法、成员变量的访问

* 要想仅能在本类中访问使用private修饰；
* 要想本包中的类都可以访问不加修饰符即可；
* 要想本包中的类与其他包中的子类可以访问使用protected修饰
* 要想所有包中的所有类都可以访问使用public修饰。
* 注意：如果类用public修饰，则类名必须与文件名相同。一个文件中只能有一个public修饰的类。

# 代码块

## 局部代码块

局部代码块是定义在方法或语句中

特点：

* 以”{}”划定的代码区域，此时只需要关注作用域的不同即可
* 方法和类都是以代码块的方式划定边界的

class Demo{

public static void main(String[] args) {

{

int x = 1;

System.out.println("普通代码块" + x);

}

int x = 99;

System.out.println("代码块之外" + x);

}

}

结果：

普通代码块1

代码块之外99

## 构造代码块

构造代码块是定义在类中成员位置的代码块

特点：

* 优先于构造方法执行，构造代码块用于执行所有对象均需要的初始化动作
* 每创建一个对象均会执行一次构造代码块。

public class Person {

private String name;

private int age;

//构造代码块

**{**

**System.out.println("**构造代码块执行了**");**

**}**

Person(){

System.out.println("Person无参数的构造函数执行");

}

Person(int age){

this.age = age;

System.out.println("Person（age）参数的构造函数执行");

}

}

class PersonDemo{

public static void main(String[] args) {

Person p = new Person();

Person p1 = new Person(23);

}

}

## 静态代码块

静态代码块是定义在成员位置，使用static修饰的代码块。

特点：

* 它优先于主方法执行、优先于构造代码块执行，当以任意形式第一次使用到该类时执行。
* 该类不管创建多少对象，静态代码块只执行一次。
* 可用于给静态变量赋值，用来给类进行初始化。

public class Person {

private String name;

private int age;

//静态代码块

static{

System.out.println("静态代码块执行了");

}

}

# 总结

## 知识点总结

* final：关键字，最终的意思

final修饰的类：最终的类，不能被继承

final修饰的变量： 相当于是一个常量, 在编译生产.class文件后，该变量变为常量值

final修饰的方法： 最终的方法，子类不能重写，可以继承过来使用

* static : 关键字， 静态的意思

可以用来修饰类中的成员(成员变量，成员方法)

注意： 也可以用来修饰成员内部类

* + 特点：

被静态所修饰的成员，会被所有的对象所共享

被静态所修饰的成员，可以通过类名直接调用，方便

Person.country = "中国";

Person.method();

* + 注意事项：

静态的成员，随着类的加载而加载，优先于对象存在

在静态方法中，没有this关键字

静态方法中，只能调用静态的成员(静态成员变量，静态成员方法

* 匿名对象：一个没有名字的对象
  + 特点：

创建匿名对象直接使用，没有变量名

匿名对象在没有指定其引用变量时，只能使用一次

匿名对象可以作为方法接收的参数、方法返回值使用

* 内部类：在一个类中，定义了一个新类，这个新的类就是内部类

class A {//外部类

class B{// 内部类

}

}

* + 特点：

内部类可以直接访问外部类的成员，包含私有的成员

* 包的声明与访问
  + 类中包的声明格式：

package 包名.包名.包名…;

* + 带有包的类，创建对象格式：包名.类名 变量名 = new包名.类名();

cn.itcast.Demo d = new cn.itcast.Demo();

* + 导包的格式：

import 包名.类名;

* 权限修饰符

public : 公共的

protected: 受保护的

private : 私有的

public protected 默认的 private

在当前类中 Y Y Y Y

同一包中的其他类 Y Y Y

不同包中的子类 Y Y

不同包中的其他类 Y

* 代码块：

局部代码块：定义在方法中的，用来限制变量的作用范围

构造代码块：定义在类中方法外，用来给对象中的成员初始化赋值

静态代码块：定义在类中方法外，用来给类的静态成员初始化赋值

# 第14天面向对象

今日内容介绍

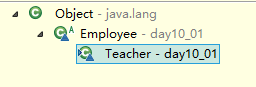
* Eclipse常用快捷键操作
* Eclipse文档注释导出帮助文档
* Eclipse项目的jar包导出与使用jar包
* 不同修饰符混合使用细节
* 辨析何时定义变量为成员变量
* 类、抽象类、接口作为方法参数
* 类、抽象类、接口作为方法返回值

# Eclipse的应用

## 常用快捷操作

* Ctrl+T：查看所选中类的继承树

例如，在下面代码中，选中Teacher类名，然后按Ctrl+T，就会显示出Teacher类的继承关系



//员工

**abstract** **class** Employee{

**public** **abstract** **void** work();

}

//讲师

**class** Teacher **extends** Employee {

**public** **void** work() {

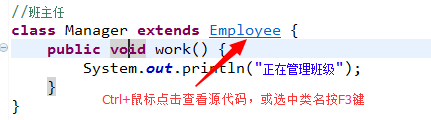
System.***out***.println("正在讲解Java");

}

}

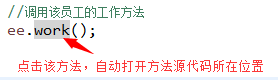
* 查看所选中类的源代码

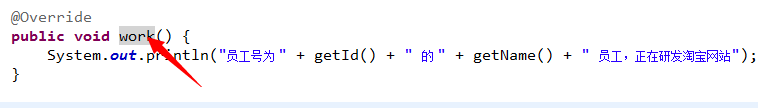
Ctrl+滑动鼠标点击类名，或者选中类名后，按F3键查看所选中类的源代码。



* 查看所选中方法的源代码

Ctrl+滑动鼠标点击方法名，或者选中方法名后，按F3键查看所选中方法的源代码。





* Eclipse中的JRE System Library是默认的Eclipse依赖JRE中的类库。在该位置可以查找到平常使用的String类、Random类、Math类等。

## 文档注释导出帮助文档

在eclipse使用时，可以配合文档注释，导出对类的说明文档，从而供其他人阅读学习与使用。

通过使用文档注释，将类或者方法进行注释用@简单标注基本信息。如@author 作者、@version代码版本、@param方法参数、@return方法返回值等。

**package** cn.itcast;

/\*\*

\* 我的工具类

\* **@author** Li

\* **@version** 1.0版本

\*/

**public** **class** Tool {

/\*\*

\* 返回两个整数的累加和

\* **@param** num1 第一个数

\* **@param** num2 第二个数

\* **@return** 返回累加和

\*/

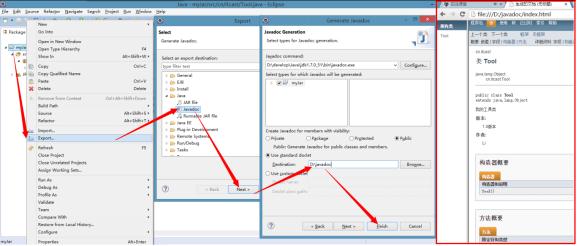
**public** **static** **int** getSum(**int** num1, **int** num2){

**return** num1 + num2;

}

}

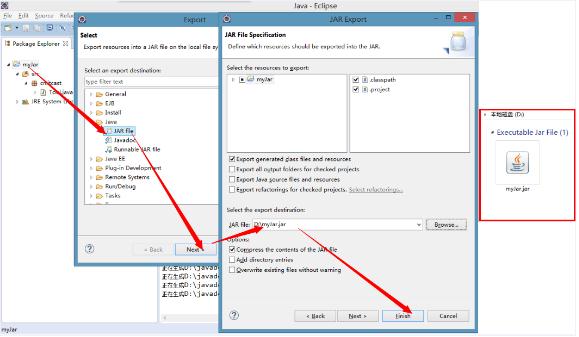
使用Eclipse导出javadoc文档即可，操作步骤如下图：



## 项目的jar包导入与导出

jar包是一个可以包含许多.class文件的压缩文件。我们可以将一个jar包加入到项目的依赖中，从而该项目可以使用该jar下的所有类；也可以把项目中所有的类打包到指定的jar包，提供给其他项目使用。

* **导出jar包：**即把项目中所有类，打包到指定的jar包中，步骤如下图：



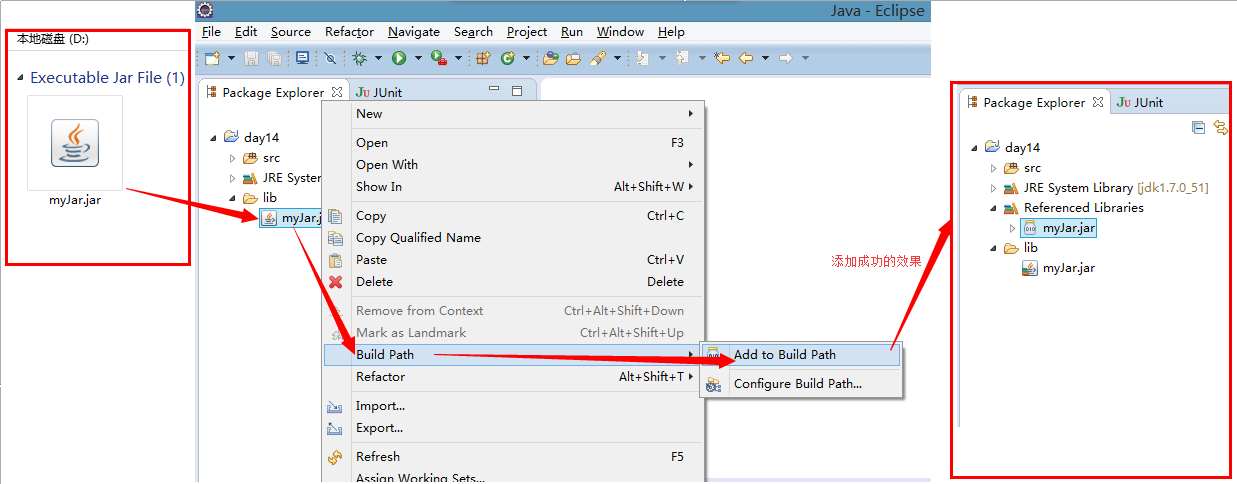
* **导入jar包：**即把指定的jar包，加入到指项目中，提供给项目使用。

导入jar包的过程是将jar包加入到项目的.classpath文件中去，让项目识别，便可以使用jar包中所有的.class文件类。以下是加入步骤：

1：项目根文件夹下创建lib文件夹，用于同一管理所有的jar文件

2：把jar文件复制到lib文件夹中

3：右键点击jar文件，点击Build Path，选择Add to Build Path，此时查看项目根文件夹下的.classpath文件，发现新加入的jar包路径被配置到了该文件中。说明可以使用jar包中所有类了。



* 注意：

Jar包加入后，必须Add to Build Path才能使用

Jar包加入后，加入的类也必须导包，如果加入的类其包名与现有类包名相同，则视作在同一个包下。(不常见)

# 面向对象

## 不同修饰符使用细节

常用来修饰类、方法、变量的修饰符如下：

* public 权限修饰符，公共访问, 类,方法,成员变量
* protected 权限修饰符，受保护访问, 方法,成员变量
* 默认什么也不写 也是一种权限修饰符，默认访问, 类,方法,成员变量
* private 权限修饰符，私有访问, 方法,成员变量
* static 静态修饰符 方法,成员变量
* final 最终修饰符 类,方法,成员变量,局部变量
* abstract 抽象修饰符 类 ,方法

我们编写程序时，权限修饰符一般放于所有修饰符之前，不同的权限修饰符不能同时使用；

同时，abstract与private不能同时使用；

同时，abstract与static不能同时使用；

同时，abstract与final不能同时使用。

* 修饰**类**能够使用的修饰符：

修饰类只能使用public、默认的、final、abstract关键字

使用最多的是 public关键字

**public** **class** Demo {} //最常用的方式

**class** Demo2{}

**public final** **class** Demo3{}

**public abstract** **class** Demo4{}

* 修饰**成员变量**能够使用的修饰符：

public : 公共的

protected : 受保护的

: 默认的

private ：私有的

final : 最终的

static : 静态的

使用最多的是 private

**public** **int** count = 100;

**protected** **int** count2 = 100;

**int** count3 = 100;

**private** **int** count4 = 100; //最常用的方式

**public** **final** **int** count5 = 100;

**public** **static** **int** *count6* = 100;

* 修饰**构造方法**能够使用的修饰符：

public : 公共的

protected : 受保护的

: 默认的

private ：私有的

使用最多的是 public

**public** Demo(){} //最常用的方式

**protected** Demo(){}

Demo(){}

**private** Demo(){}

* 修饰**成员方法**能够使用的修饰符：

public : 公共的

protected : 受保护的

: 默认的

private ：私有的

final : 最终的

static : 静态的

abstract : 抽象的

使用最多的是 public

**public** **void** method1(){}//最常用的方式

**protected** **void** method2(){}

**void** method3(){}

**private** **void** method4(){}

**public** **final** **void** method5(){}

**public** **static** **void** method6(){}//最常用的方式

**public** **abstract** **void** method7();//最常用的方式

# 自定义数据类型的使用

## 辨析成员变量与方法参数的设计定义

* 定义长方形类，包含求周长与求面积的方法
* 定义数学工具类，包含求两个数和的二倍与求两个数积的方法

思考：这两个类的计算方法均需要两个数参与计算，请问两个数定义在成员位置还是形参位置更好，为什么？

如果变量是该类的一部分时，定义成成员变量。

如果变量不应该是类的一部分，而仅仅是功能当中需要参与计算的数，则定义为形参变量。

* 数学工具类

**public** **class** MathTool {

//求两个数的和的二倍

**public** **double** sum2times(**int** number,**int** number2) {

**return** (number+number2)\*2;

}

//求两个数的积

**public** **double** area(**int** number,**int** number2) {

**return** number\*number2;

}

}

* 长方形类

**public** **class** CFX {

//因为长与宽，在现实事物中属于事物的一部分，所以定义成员变量

**private** **int** chang;

**private** **int** kuan;

**public** CFX(**int** chang, **int** kuan) {

**this**.chang = chang;

**this**.kuan = kuan;

}

//求长与宽的周长

**public** **double** zhouChang() {

**return** (chang+kuan)\*2;

}

//求长与宽的面积

**public** **double** mianJi() {

**return** chang\*kuan;

}

**public** **int** getChang() {

**return** chang;

}

**public** **void** setChang(**int** chang) {

**this**.chang = chang;

}

**public** **int** getKuan() {

**return** kuan;

}

**public** **void** setKuan(**int** kuan) {

**this**.kuan = kuan;

}

}

## 类作为方法参数与返回值

* 类作为方法参数

在编写程序中，会经常碰到调用的方法要接收的是一个类类型的情况，那么这时，要向方法中传入该类的对象。如下代码演示：

**class** Person{

**public** **void** show(){

System.***out***.println("show方法执行了");

}

}

//测试类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//创建Person对象

Person p = **new** Person();

//调用method方法

*method*(p);

}

//定义一个方法method，用来接收一个Person对象，在方法中调用Person对象的show方法

**public** **static** **void** method(Person p){

p.show();

}

}

* 类作为方法返回值

写程序调用方法时，我们以后会经常碰到返回一个类类型的返回值，那么这时，该方法要返回一个该类的对象。如下代码演示：

**class** Person{

**public** **void** show(){

System.***out***.println("show方法执行了");

}

}

//测试类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//调用method方法，获取返回的Person对象

Person p = *method*();

//调用p对象中的show方法

p.show();

}

//定义一个方法method，用来获取一个Person对象，在方法中完成Person对象的创建

**public** **static** Person method(){

Person p = **new** Person();

**return** p;

}

}

## 抽象类作为方法参数与返回值

* 抽象类作为方法参数

今后开发中，抽象类作为方法参数的情况也很多见。当遇到方法参数为抽象类类型时，要传入一个实现抽象类所有抽象方法的子类对象。如下代码演示：

//抽象类

**abstract** **class** Person{

**public** **abstract** **void** show();

}

**class** Student **extends** Person{

@Override

**public** **void** show() {

System.***out***.println("重写了show方法");

}

}

//测试类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//通过多态的方式，创建一个Person类型的变量，而这个对象实际是Student

Person p = **new** Student();

//调用method方法

*method*(p);

}

//定义一个方法method，用来接收一个Person类型对象，在方法中调用Person对象的show方法

**public** **static** **void** method(Person p){//抽象类作为参数

//通过p变量调用show方法,这时实际调用的是Student对象中的show方法

p.show();

}

}

* 抽象类作为方法返回值

抽象类作为方法返回值的情况，也是有的，这时需要返回一个实现抽象类所有抽象方法的子类对象。如下代码演示：

//抽象类

**abstract** **class** Person{

**public** **abstract** **void** show();

}

**class** Student **extends** Person{

@Override

**public** **void** show() {

System.***out***.println("重写了show方法");

}

}

//测试类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//调用method方法，获取返回的Person对象

Person p = *method*();

//通过p变量调用show方法,这时实际调用的是Student对象中的show方法

p.show();

}

//定义一个方法method，用来获取一个Person对象，在方法中完成Person对象的创建

**public** **static** Person method(){

Person p = **new** Student();

**return** p;

}

}

## 接口作为方法参数与返回值

* 接口作为方法参数

接口作为方法参数的情况是很常见的，经常会碰到。当遇到方法参数为接口类型时，那么该方法要传入一个接口实现类对象。如下代码演示。

//接口

**interface** Smoke{

**public** **abstract** **void** smoking();

}

**class** Student **implements** Smoke{

@Override

**public** **void** smoking() {

System.***out***.println("课下吸口烟，赛过活神仙");

}

}

//测试类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//通过多态的方式，创建一个Smoke类型的变量，而这个对象实际是Student

Smoke s = **new** Student();

//调用method方法

*method*(s);

}

//定义一个方法method，用来接收一个Smoke类型对象，在方法中调用Smoke对象的show方法

**public** **static** **void** method(Smoke sm){//接口作为参数

//通过sm变量调用smoking方法，这时实际调用的是Student对象中的smoking方法

sm.smoking();

}

}

* 接口作为方法返回值

接口作为方法返回值的情况，在后面的学习中会碰到。当遇到方法返回值是接口类型时，那么该方法需要返回一个接口实现类对象。如下代码演示。

//接口

**interface** Smoke{

**public** **abstract** **void** smoking();

}

**class** Student **implements** Smoke{

@Override

**public** **void** smoking() {

System.***out***.println("课下吸口烟，赛过活神仙");

}

}

//测试类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

//调用method方法，获取返回的会吸烟的对象

Smoke s = *method*();

//通过s变量调用smoking方法,这时实际调用的是Student对象中的smoking方法

s.smoking();

}

//定义一个方法method，用来获取一个具备吸烟功能的对象，并在方法中完成吸烟者的创建

**public** **static** Smoke method(){

Smoke sm = **new** Student();

**return** sm;

}

}

# 星级酒店案例

## 案例介绍

某五星级酒店，资金雄厚，要招聘多名员工（经理、厨师、服务员）。入职的员工需要记录个人信息（姓名、工号、经理特有奖金属性）。他们都有自己的工作要做。

本案例要完成如下需求：

* 获取酒店幸运员工；
* 酒店开设VIP服务，酒店的厨师与服务员可以提供VIP服务。（厨师做菜加量、服务员给顾客倒酒）。
* 编写测试类
  + 向酒店中，增加多名员工（其中包含1名经理，1名厨师、2名服务员）；
  + 调用酒店员工的工作功能
  + 调用酒店员工的VIP服务功能

## 案例需求分析

* 根据“某五星级酒店，资金雄厚……都有自己的工作要做。”分析出，该题目中包含酒店，可以把它封装成类，多名员工）。

class 员工 {

属性：姓名

属性：工号

方法：工作

}

class 厨师 extends 员工{}

class 服务员 extends 员工{}

class 经理 extends 员工 {

属性：奖金

}

员工的类型有经理、厨师、服务员，它们有共同的属性（姓名、工号、），经理额外属性（奖金）。

* 根据“向酒店中，增加多名员工（其中包含1名经理，1名厨师、2名服务员）”。分析出，要创建一个酒店对象，并添加4名员工到酒店对象的员工集合中。

酒店员工集合添加新员工： 经理对象

酒店员工集合添加新员工： 厨师对象

酒店员工集合添加新员工： 服务员对象

酒店员工集合添加新员工： 服务员对象

* 根据“获取酒店幸运员工”。分析出，从酒店员工集合随机得到一名员工对象。

1. 从酒店员工集合长度范围内，随机产生一个随机数

2. 使用该随机数作为集合的索引，返回该索引处对应的员工对象

* 根据“酒店开设VIP服务，酒店的厨师与服务员可以提供VIP服务。（厨师做菜加量、服务员给顾客倒酒）”。分析出，这是要增加一个VIP的接口，接口中提供个VIP服务的方法。让厨师与服务员实现该接口。

interface VIP服务{

抽象方法：服务

}

class 厨师 extends 员工 implements VIP服务{ 重写服务方法 }

class 服务员 extends 员工 implements VIP服务{ 重写服务方法 }

## 实现代码步骤

* VIP服务

public interface VIP {

public abstract void server(); //服务

}

* 员工

/\*

\* 员工：

姓名 String

工号 String

\*/

**public** **abstract** **class** YuanGong {

// 成员变量

**private** String xingMing;

**private** String gongHao;

// 构造方法

**public** YuanGong() {

**super**();

}

**public** YuanGong(String xingMing, String gongHao) {

**super**();

**this**.xingMing = xingMing;

**this**.gongHao = gongHao;

}

// 抽象方法

**public** **abstract** **void** work();

// getters与setters

**public** String getXingMing() {

**return** xingMing;

}

**public** **void** setXingMing(String xingMing) {

**this**.xingMing = xingMing;

}

**public** String getGongHao() {

**return** gongHao;

}

**public** **void** setGongHao(String gongHao) {

**this**.gongHao = gongHao;

}

}

* 服务员

/\*

\* 定义员工的子类 服务员类

\*/

**public** **class** FuWuYuan **extends** YuanGong **implements** VIP {

**public** FuWuYuan() {

**super**();

}

**public** FuWuYuan(String xingMing, String gongHao) {

**super**(xingMing, gongHao);

}

@Override

**public** **void** work() {

System.***out***.println("亲，全身心为您服务，记得给好评哦");

}

@Override

**public** **void** server() {

System.***out***.println("给顾客倒酒");

}

}

* 经理

/\*

\* 经理在员工的基础上，添加了奖金成员

\*/

**public** **class** JingLi **extends** YuanGong {

**private** **double** jiangJin;

**public** JingLi() {

**super**();

}

**public** JingLi(String xingMing, String gongHao, **double** jiangJin) {

**super**(xingMing, gongHao);

**this**.jiangJin = jiangJin;

}

**public** **double** getJiangJin() {

**return** jiangJin;

}

**public** **void** setJiangJin(**double** jiangJin) {

**this**.jiangJin = jiangJin;

}

@Override

**public** **void** work() {

System.***out***.println("哪个员工让顾客不满意，我扣谁钱");

};

}

* 厨师

/\*

\* 定义员工的子类 厨师类

\*/

**public** **class** ChuShi **extends** YuanGong **implements** VIP{

**public** ChuShi() {

**super**();

}

**public** ChuShi(String xingMing, String gongHao) {

**super**(xingMing, gongHao);

}

@Override

**public** **void** work() {

System.***out***.println("我做饭，放心吃吧，包您满意");

}

@Override

**public** **void** server() {

System.***out***.println("做菜加量加料");

}

}

* 测试类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

}

}

# 总结

## 知识点总结

* 不同修饰符的使用
  + 类，最常使用public修饰
  + 成员变量，最常使用private修饰
  + 成员方法，最常使用public修饰
* 自定义数据类型的使用
  + 类作为方法参数时，说明要向方法中传入该类的对象
  + 类作为方法返回值时，说明该方法要返回一个该类的对象。
  + 抽象类作为方法参数时，说明要传入一个实现抽象类所有抽象方法的子类对象。
  + 抽象类作为方法返回值时，说明需要返回一个实现抽象类所有抽象方法的子类对象。
  + 接口作为方法参数时，说明该方法要传入一个接口实现类对象。
  + 接口作为方法返回值时，说明该方法需要返回一个接口实现类对象。

# 第15天常用API

今日内容介绍

* Object
* String
* StringBuilder

# Java的API及Object类

在以前的学习过程中，我们都在学习对象基本特征、对象的使用以及对象的关系。接下来我们开始使用对象做事情，那么在使用对象做事情之前，我们要学习一些API中提供的常用对象。首先在学习API中的Object类之前,先来学习如何使用API。

## Java 的API

Java 的API（API: **A**pplication(应用) **P**rogramming(程序) **I**nterface(接口)）

Java API就是JDK中提供给我们使用的类，这些类将底层的代码实现封装了起来，我们不需要关心这些类是如何实现的，只需要学习这些类如何使用即可。

在JDK安装目录下有个src.zip文件，这个文件解压缩后里面的内容是所有Java类的源文件。可以在其中查看相对应的类的源码。

我们在每次查看类中的方法时，都打开源代码进行查看，这种方式过于麻烦。其实，我们可以通过查帮助文档的方式，来了解Java提供的API如何使用。如下图操作：查找Object类



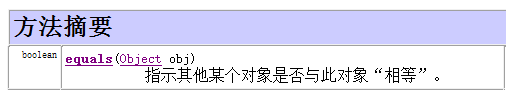
通过帮助文档中类与方法的介绍，我们就能够使用这个类了。

## Object类概述

Object类是Java语言中的根类，即所有类的父类。它中描述的所有方法子类都可以使用。所有类在创建对象的时候，最终找的父类就是Object。

在Object类众多方法中，我们先学习equals方法与toString方法，其他方法后面课程中会陆续学到。

## equals方法



equals方法，用于比较两个对象是否相同，它其实就是使用两个对象的内存地址在比较。Object类中的equals方法内部使用的就是==比较运算符。

在开发中要比较两个对象是否相同，经常会根据对象中的属性值进行比较，也就是在开发经常需要子类重写equals方法根据对象的属性值进行比较。如下代码演示：

/\*

描述人这个类，并定义功能根据年龄判断是否是同龄人

由于要根据指定类的属性进行比较，这时只要覆盖Object中的equals方法

在方法体中根据类的属性值进行比较

\*/

class Person extends Object{

int age ;

//复写父类的equals方法，实现自己的比较方式

public boolean equals(Object obj) {

//判断当前调用equals方法的对象和传递进来的对象是否是同一个

if(this == obj){

return true;

}

//判断传递进来的对象是否是Person类型

if(!(obj instanceof Person)){

return false;

}

//将obj向下转型为Perosn引用，访问其属性

Person p = (Person)obj;

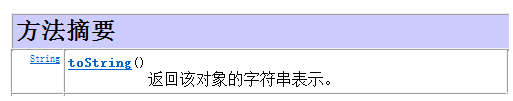
return this.age == p.age;

}

}

**注意**：在复写Object中的equals方法时，一定要注意public boolean equals(Object obj)的参数是Object类型，在调用对象的属性时，一定要进行类型转换，在转换之前必须进行类型判断。

## toString方法



toString方法返回该对象的字符串表示，其实该字符串内容就是对象的类型+@+内存地址值。

由于toString方法返回的结果是内存地址，而在开发中，经常需要按照对象的属性得到相应的字符串表现形式，因此也需要重写它。

class Person extends Object{

int age ;

//根据Person类的属性重写toString方法

public String toString() {

return "Person [age=" + age + "]";

}

}

# String类

## String类的概述

查阅API中的String类的描述，发现String 类代表字符串。Java 程序中的所有字符串字面值（如 "abc" ）都作为此类的实例实现。

//演示字符串

String str = "itcast";

str = "传智播客";

继续查阅API发现说字符串是常量；它们的值在创建之后不能更改，这是什么意思呢？其实就是说一旦这个字符串确定了，那么就会在内存区域中就生成了这个字符串。字符串本身不能改变，但str变量中记录的地址值是可以改变的。

继续查API发现，字符串有大量的重载的构造方法。通过String类的构造方法可以完成字符串对象的创建，那么，通过使用双引号的方式创建对象与new的方式创建对象，有什么不同呢？看如下程序与图解：

String s3 = "abc";

String s4 = **new** String("abc");

System.*out*.println(s3==s4);//false

System.*out*.println(s3.equals(s4));//true,

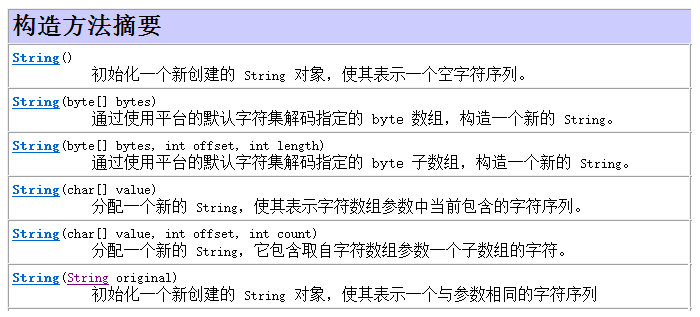
//因为String重写了equals方法，建立了字符串自己的判断相同的依据（通过字符串对象中的字符来判断）

s3和s4的创建方式有什么不同呢？

* s3创建，在内存中只有一个对象。这个对象在字符串常量池中
* s4创建，在内存中有两个对象。一个new的对象在堆中，一个字符串本身对象，在字符串常量池中

## String类构造方法

构造方法是用来完成String对象的创建，下图中给出了一部分构造方法需要在API中找到，并能够使用下列构造方法创建对象。



String s1 = new String(); //创建String对象，字符串中没有内容

byte[] bys = new byte[]{97,98,99,100};

String s2 = new String(bys); // 创建String对象，把数组元素作为字符串的内容

String s3 = new String(bys, 1, 3); //创建String对象，把一部分数组元素作为字符串的内容，参数offset为数组元素的起始索引位置，参数length为要几个元素

char[] chs = new char[]{’a’,’b’,’c’,’d’,’e’};

String s4 = new String(chs); //创建String对象，把数组元素作为字符串的内容

String s5 = new String(chs, 0, 3);//创建String对象，把一部分数组元素作为字符串的内容，参数offset为数组元素的起始索引位置，参数count为要几个元素

String s6 = new String(“abc”); //创建String对象，字符串内容为abc

## String类的方法查找

String类中有很多的常用的方法，我们在学习一个类的时候，不要盲目的把所有的方法尝试去使用一遍，这时我们应该根据这个对象的特点分析这个对象应该具备那些功能，这样大家应用起来更方便。

字符串是一个对象，那么它的方法必然是围绕操作这个对象的数据而定义的。我们想想字符串中有哪些功能呢？

* 字符串中有多少个字符?

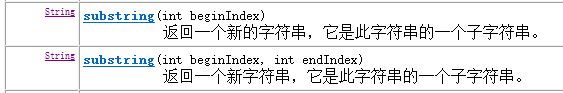


String str = "abcde";

int len = str.length();

System.out.println("len="+len);

* 获取部分字符串。



String str = "abcde";

String s1 = str.substring(1); //返回一个新字符串，内容为指定位置开始到字符串末尾的所有字符

String s2 = str.substring(2, 4);//返回一个新字符串，内容为指定位置开始到指定位置结束所有字符

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

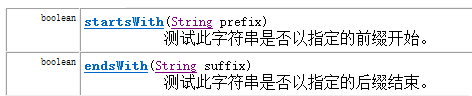
System.out.println("s1="+s1);

System.out.println("s2="+s2);

## String类中方法查找练习

前面给大家简单介绍了几个字符串中常用的方法，这个过程中主要让大家学会如何去查阅API，如何找到自己想用的方法。接下来我们来练习下字符串方法的查找。

* 字符串是否以指定字符串开头。结尾同理。



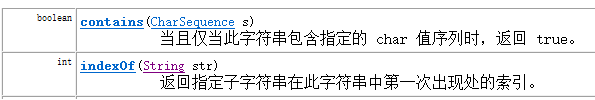
String str = "StringDemo.java";

boolean b1 = str.startsWith("Demo");//判断是否以给定字符串开头

boolean b2 = str.startsWith("String");

boolean b3 = str.endsWith("java");//判断是否以给定字符串结尾

* 字符串中是否包含另一个字符串。

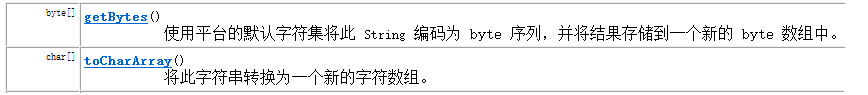


String str = "abcde";

int index = str.indexOf(“bcd”); //判断是否包含指定字符串，包含则返回第一次出现该字符串的索引，不包含则返回-1

boolean b2 = str.contains("bcd");//判断是否包含指定字符串，包含返回true，不包含返回false

* 将字符串转成一个字符数组。或者字节数组。

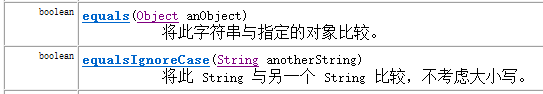


String str = "abcde";

char[] chs = str.toCharArray();

byte[] bytes = str.getBytes();

* 判断两个字符串中的内容是否相同



String str = "abcde";

String str2 = "abcde";

String str3 = "hello";

boolean b1 = str.equals(str2);

boolean b2 = str.equals(str3);

* 获取该字符串对象中的内容



String str = new String("hello");

System.out.println( str.toString() );

System.out.pintln( str );

直接打印引用类型变量时，默认调用该类型进行重写后的toString方法

下面的需求所对应的方法，要求大家自己动手在API中查找，并进行方法使用。

* 判断该字符串的内容是否为空的字符串
* 获取给定的字符，在该字符串中第一次出现的位置
* 获取该字符串中指定位置上的字符
* 把该字符串转换成 小写字符串
* 把该字符串转换成 大写字符串
* 在该字符串中，将给定的旧字符，用新字符替换
* 在该字符串中， 将给定的旧字符串，用新字符串替换
* 去除字符串两端空格，中间的不会去除，返回一个新字符串

## String类方法使用练习

* 题目一：获取指定字符串中，大写字母、小写字母、数字的个数。
* 思路：1.为了统计大写字母、小写字母、数字的个数。创建3个计数的变量。

2.为了获取到字符串中的每个字符，进行字符串的遍历，得到每个字符。

3.对得到的字符进行判断，如果该字符为大写字母，则大写字母个数+1；如果该字符为小写字母，则小写字母个数+1；如果该字符为数字，则数字个数+1。

4.显示大写字母、小写字母、数字的个数

* 代码：

public static void method(String str){

int bigCount = 0; //大写字母的个数

int smallCount = 0; //小写字母的个数

int numberCount = 0; //数字的个数

for (int i=0; i < str.length(); i++) {

char ch = str.charAt(i); //获取指定位置上的字符

if (ch>=’A’ && ch<=’Z’) {

bigCount++;

} else if (ch>=’a’ && ch<=’z’) {

smallCount++;

} else if (ch>=’0’ && ch<=’9’) {

numberCount++;

}

}

System.out.println("大写字母个数："+bigCount);

System.out.println("小写字母个数："+smallCount);

System.out.println("数字个数："+numberCount);

}

* 题目二：将字符串中，第一个字母转换成大写，其他字母转换成小写，并打印改变后的字符串。
* 思路：1.把字符串分为两个部分，第一部分为字符串中第一个字母，第二部分为剩下的字符串。

2.把第一部分字符串转换成大写字母，把第二部分字符串转换成小写字母

3.把两部分字符串连接在一起，得到一个完整的字符串

* 代码：

public static String convert(String str){

//获取第一部分为字符串

String start = str.substring(0,1);

//获取第二部分为字符串

String end = str.substring(1);

//把第一部分字符串转换成大写字母，把第二部分字符串转换成小写字母

String big = start.toUpperCase();

String small = end.toLowerCase();

//把两部分字符串连接在一起，得到一个完整的字符串

return big+small;

}

* 题目三：查询大字符串中，出现指定小字符串的次数。如“hellojava,nihaojava,javazhenbang”中查询出现“java”的次数。
* 思路：1.在大串中，查找小串出现的位置，出现了就次数+1

2.在上次小串出现位置的后面继续查找，需要更改大串的内容为上次未查询到的字符串。

3.回到第一步，继续查找小串出现的位置，直到大串中查询不到小串为止

* 代码：

public static int getCount(String big, String small){

int count = 0; //出现小串的次数

int index = -1;//出现小串的位置

/\*

while的循环条件三步骤：

步骤一. big.indexOf(small) 获取小串在大串中出现的位置

步骤二. 把小串出现的位置，赋值给变量index

步骤三. 判断出现的位置是否为-1， 如果位置等于-1，说明大串中已经查询不到小串了；如果位置不等于-1，那么，进行循环，完成次数累加与修改大串的操作

\*/

while ((index = big.indexOf(small)) != -1 ){

count++;//出现次数+1

//更改大串内容

big = big.substring(index+1);

}

return count;

}

# 字符串缓冲区

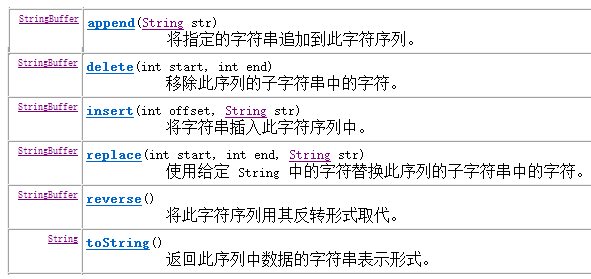
## StringBuffer类

在学习String类时，API中说字符串缓冲区支持可变的字符串，什么是字符串缓冲区呢？接下来我们来研究下字符串缓冲区。

查阅StringBuffer的API，StringBuffer又称为可变字符序列，它是一个类似于 String 的字符串缓冲区，通过某些方法调用可以改变该序列的长度和内容。

原来StringBuffer是个字符串的缓冲区，即就是它是一个容器，容器中可以装很多字符串。并且能够对其中的字符串进行各种操作。

## StringBuffer的方法使用



* 代码演示：

创建一个字符串缓冲区对象。用于存储数据。

StringBuffer sb = new StringBuffer();

sb.append("haha"); //添加字符串

sb.insert(2, "it");//在指定位置插入

sb.delete(1, 4);//删除

sb.replace(1, 4, "cast");//替换指定范围内的内容

String str = sb.toString();

* 注意：append、delete、insert、replace、reverse方法调用后，返回值都是当前对象自己，所以说，StringBuffer它可以改变字符序列的长度和内容。

## StringBuffer类方法查找练习

下面的需求所对应的方法，要求大家自己动手在API中查找，并进行方法使用。

* 从指定位置开始，到末尾结束，截取该字符串缓冲区，返回新字符串
* 在原有字符串缓冲区内容基础上，删除指定位置上的字符

## 对象的方法链式调用

在我们开发中，会遇到调用一个方法后，返回一个对象的情况。然后使用返回的对象继续调用方法。这种时候，我们就可以把代码现在一起，如append方法一样，代码如下：

创建一个字符串缓冲区对象。用于存储数据。

StringBuffer sb = new StringBuffer();

添加数据。不断的添加数据后，要对缓冲区的最后的数据进行操作，必须转成字符串才可以。

String str = sb.append(true).append("hehe").toString();

## StringBuffer练习

练习：int[] arr = {34,12,89,68};将一个int[]中元素转成字符串 格式 [34,12,89,68]

**public** **static** String toString\_2(**int**[] arr) {

StringBuffer sb = **new** StringBuffer();

sb.append("[");

**for** (**int** i = 0; i < arr.length; i++) {

**if**(i!=arr.length-1){

sb.append(arr[i]+",");

}**else**{

sb.append(arr[i]+"]");

}

}

**return** sb.toString();

}

* 无论多少数据，数据是什么类型都不重要，只要最终变成字符串就可以使用StringBuffer这个容器

## StringBuilder类

查阅API发现还有一个StringBuilder类，它也是字符串缓冲区，StringBuilder与它和StringBuffer的有什么不同呢？

我们阅读StringBuilder的API说明发现，它也是一个可变的字符序列。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。

目前，我们还没有涉及到线程与同步，知道结论StringBuilder比StringBuffer快即可。为什么快，我们会在学习线程时讲解。

# 总结

## 知识点总结

* Object: 它是所有类的超类，祖宗类。java中所有的类都直接或间接的继承这个类
  + 方法

public String toString() 返回当前对象中的内容, 对于Object类默认操作来说，返回的对象的类型+@+内存地址值

public boolean equals(Object obj) 比较两个对象内容是否相同，对于Object类默认操作来说,比较的是地址值

* String: 字符串类，字符串是常量；它们的值在创建之后不能更改
  + 方法

boolean equals(Object obj) 判断两个字符串中的内容是否相同

boolean equalsIgnoreCase(String str) 判断两个字符串中的内容是否相同, 忽略大小写

boolean contains(String str) 判断该字符串中 是否包含给定的字符串

boolean startsWith(String str) 判断该字符串 是否以给定的字符串开头

boolean endsWith(String str) 判断该字符串 是否以给定的字符串结尾

boolean isEmpty() 判断该字符串的内容是否为空的字符串 ""

int length() 获取该字符串的长度

char charAt(int index) 获取该字符串中指定位置上的字符

String substring(int start) 从指定位置开始，到末尾结束，截取该字符串，返回新字符串

String substring(int start,int end) 从指定位置开始，到指定位置结束，截取该字符串，返回新字符串

int indexOf(int ch ) 获取给定的字符，在该字符串中第一次出现的位置

int indexOf(String str) 获取给定的字符串，在该字符串中第一次出现的位置

int indexOf(int ch,int fromIndex) 从指定位置开始，获取给定的字符，在该字符

byte[] getBytes() 把该字符串 转换成 字节数组

char[] toCharArray() 把该字符串 转换成 字符数组

String replace(char old,char new) 在该字符串中，将给定的旧字符，用新字符替换

String replace(String old,String new) 在该字符串中， 将给定的旧字符串，用新字符串替换

String trim() 去除字符串两端空格，中间的不会去除，返回一个新字符串

String toLowerCase() 把该字符串转换成 小写字符串

String toUpperCase() 把该字符串转换成 大写字符串

int indexOf(String str,int fromIndex) 从指定位置开始，获取给定的字符串，在该字符串中第一次出现的位置

* StringBuffer/StringBuilder:
  + 方法

public StringBuffer append(String str) 在原有字符串缓冲区内容基础上，在末尾追加新数据

public StringBuffer insert(int offset,String str) 在原有字符串缓冲区内容基础上，在指定位置插入新数据

public StringBuffer deleteCharAt(int index) 在原有字符串缓冲区内容基础上，删除指定位置上的字符

public StringBuffer delete(int start,int end) 在原有字符串缓冲区内容基础上，删除指定范围内的多个字符

public StringBuffer replace(int start,int end,String str)在原有字符串缓冲区内容基础上，将指定范围内的多个字符 用给定的字符串替换

public StringBuffer reverse() 将字符串缓冲区的内容 反转 "abc"----"cba"

public String substring(int start) 从指定位置开始，到末尾结束，截取该字符串缓冲区，返回新字符串

public String substring(int start,int end) 从指定位置开始，到指定位置结束，截取该字符串缓冲区，返回新字符串