



JAVA 学习指南

邵发

配套资料： 教学视频 | 书本 | 在线题库

官网 <http://afanihao.cn> QQ 群 495734195

前 言

本书为 Java 快速入门与进阶 的配套教材，共 27 章。

官网: <http://afanihao.cn>

作者: 邵发

官方 QQ 群: 495734195

本系列教程由 24 篇以上视频教程组成，从入门语法到行业级技术，循序渐近式的全方位教程。内容包含入门语法和高级语法，覆盖 Java 在业界的 3 个应用领域（网站开发、安卓 APP 开发、桌面 GUI 开发）。同时包含专项技术的培训教程，如网络编程基础、数据库开发，FreeMarker, Spring, MyBatis 等。

『Java 学习指南系列』 共有 20 余篇视频课程组成，罗列如下：
（星号课程为主线课程）

- * Java 入门与进阶
- * 3 Swing 入门篇 [必学]
- * 4 Swing 高级篇 [必学]
- * 5 网页基础篇
- * 6 网站入门篇
- * 7 网站中级篇
- * 8 数据库篇(MySQL)
- 9 安卓入门篇
- 10 安卓高级篇
- * 11 FreeMarker 篇
- 12 网站高级篇
- 13 数据结构与算法
- 14 网络通信篇
- 15 Linux/CentOS 篇

- 16 项目应用篇
- 17 二进制篇
- * 18 反射与框架原理
- 19 JavaFX 入门
- 20 JavaFX 高级
- * 21 Spring 入门
- * 22 Spring 高级
- 23 SpringBoot
- 24 MyBatis 框架

以上所列课程，可以从官网 <http://afanihao.cn> 进入学习。

目 录

前 言	1
目 录	3
第 1 章 Hello World	13
本章学习目标	13
1.1 课程概述	13
1.2 开发环境的安装	13
1.3 创建第一个程序	14
1.4 打印, 注释与空白	15
1.4.1 打印	15
1.4.2 注释	16
1.4.3 空白	17
第 2 章 变量	19
本章学习目标	19
2.1 使用变量	19
2.1.1 引例	19
2.1.2 变量	20
2.1.3 变量的命名	21
2.1.4 变量的简单运算	21
2.2 整数, 小数, 字符串	22
2.2.1 整数与小数	22
2.2.2 字符串	23
2.2.3 变量的赋值	24
2.3 布尔类型	24
2.3.1 引例	24
2.3.2 boolean 类型	25
2.3.3 常见问题	26
2.4 其他类型	27
第 3 章 操作符与表达式	28
本章学习目标	28
3.1 算术操作符	28
3.1.1 整型的算术运算	29
3.1.2 浮点型的算术运算	29
3.1.3 练习	30
3.1.4 类型提升	30
3.1.5 混合运算与优先级	30

3.2	关系操作符	31
3.2.1	关系操作符的运算	32
3.2.2	更多练习	32
3.2.3	注意事项	33
3.3	逻辑操作符	34
3.3.1	逻辑与 &&	34
3.3.2	逻辑或 	34
3.3.3	逻辑非 !	35
3.3.4	注意事项	35
3.4	赋值操作符	36
3.4.1	区分初始值与赋值	36
3.4.2	赋值操作的过程	36
3.4.3	组合赋值操作符	37
3.5	自增操作符	37
3.5.1	前置自增与后置自增	38
3.5.2	自减操作符	38
3.6	操作符的优先级	39
3.7	类型转换操作符	39
3.7.1	类型转换时的数据损失	40
3.7.2	显式转换与隐式转换	40
3.7.3	注意事项	41
第4章	语句	42
	本章学习目标	42
4.1	语句的概念	42
4.1.1	空语句	43
4.1.2	复合语句	43
4.2	if 语句	44
4.2.1	分步过程讲解	44
4.2.2	if 语句的两种基本形式	45
4.2.3	if 语句的完全形式	47
4.2.4	常见问题	48
4.3	for 语句	49
4.3.1	引例	49
4.3.2	for 语句	50
4.3.3	更多例子	51
4.3.4	循环变量	51
4.4	for 语句的嵌套	52
4.4.1	例子	53

4.4.2 例子	54
4.5 break 与 continue	55
4.5.1 break 语句	55
4.5.2 continue 语句	56
4.6 for 语句的变形	56
4.6.1 初始化 E1 为空	57
4.6.2 循环条件 E2 为空	58
4.6.3 后置表达式 E3 为空	58
4.7 while 语句	59
4.7.1 例子	59
4.7.2 while 语句的变形	60
第 5 章 数组对象	62
本章学习目标	62
5.1 数组	62
5.1.1 数组的定义	63
5.1.2 数组的遍历	63
5.1.3 数组的初始化	65
5.1.4 数组的长度	65
5.1.5 数组的打印输出	66
5.2 数组的使用	66
5.2.1 数组的应用举例	67
5.3 对象与引用	68
5.3.1 对象与引用的概念	68
5.3.2 空对象 null	69
5.3.3 空指针错误 NullPointerException	70
5.3.4 失去引用的对象	70
第 6 章 类	72
本章学习目标	72
6.1 新建类	72
6.1.1 类	73
6.1.2 创建对象	74
6.1.3 类与对象	74
6.1.4 常见错误	75
6.2 类的属性	75
6.2.1 类的书写步骤	76
6.2.2 编程世界里的类	77
6.2.3 类的嵌套书写	77
6.3 再说对象与引用	78

6.3.1	空对象与空指针异常	79
6.4	属性的默认值	79
第 7 章	类的方法	81
	本章学习目标	81
7.1	方法	81
7.1.1	添加方法	81
7.1.2	方法的调用	82
7.1.3	方法的命名	83
7.2	方法的参数	83
7.2.1	示例 1	83
7.2.2	示例 2	84
7.2.3	理解参数的作用	85
7.3	方法的返回值 (1)	86
7.3.1	示例 1	87
7.3.2	示例 2	88
7.3.3	void 返回值	89
7.4	方法的返回值 (2)	90
7.4.1	示例 1	90
7.4.2	示例 2	92
7.5	方法的返回值 (3)	93
7.5.1	示例 1	94
7.5.2	示例 2	95
7.6	方法名的重载	96
第 8 章	当前对象	98
	本章学习目标	98
8.1	当前对象 this	98
8.1.1	this 参数	100
8.1.2	调用自己的方法	101
8.2	省略与重名	102
8.2.1	省略 this	102
8.2.2	重名	103
8.3	类的设计示范	104
8.3.1	示例 1	104
8.3.2	示例 2	106
8.4	特殊形式的属性	108
第 9 章	访问控制与封装	112
	本章学习目标	112
9.1	访问修饰符	112

9.1.1	私有的	112
9.1.2	可见性	113
9.2	Getter 与 Setter	114
9.2.1	Getter 方法	115
9.2.2	Setter 方法	115
9.2.3	封装	116
第 10 章	对象的创建与销毁	118
	本章学习目标	118
10.1	构造方法	118
10.1.1	添加构造方法	119
10.1.2	构造方法的作用	119
10.1.3	构造方法的重载	120
10.1.4	默认构造方法	121
10.1.5	构造方法的访问控制	122
10.2	对象的销毁	123
第 11 章	继承	125
	本章学习目标	125
11.1	类的继承	125
11.1.1	引例 1	125
11.1.2	引例 2	126
11.1.3	继承 extends	126
11.2	重写	129
11.2.1	部分重写	130
11.3	构造方法的继承	131
11.3.1	显式调用父类构造方法	132
11.4	单根继承	134
11.4.1	Object 类	134
11.4.2	重写 toString 方法	135
11.5	多态	138
11.5.1	父子类型之间的转换	138
11.5.2	方法的多态调用	139
11.6	protected	140
第 12 章	包	142
	本章学习目标	142
12.1	包 package	142
12.1.1	包路径与类路径	143
12.1.2	包的声明	143
12.1.3	包的导入 import	144

第 13 章 静态方法	147
本章学习目标	147
13.1 静态方法	147
13.1.1 添加静态方法	148
13.1.2 静态方法的调用	149
13.2 静态方法的使用	150
13.3 程序的入口	151
第 14 章 常见工具类	153
本章学习目标	153
14.1 字符串	153
14.1.1 字符串的拼接	153
14.1.2 字符串的长度	154
14.1.3 空字符串	154
14.1.4 获取子串	154
14.1.5 判断内容是否相同	154
14.1.6 字典序比较	155
14.1.7 格式化	156
14.1.8 查找子串	156
14.1.9 前缀后缀	157
14.1.10 清除空白	157
14.1.11 分割	158
14.2 包装类	158
14.2.1 包装类	159
14.2.2 装箱拆箱	159
14.2.3 Integer 与 String 转换	160
14.2.4 值的比较	160
14.2.5 包装类的作用	161
14.3 控制台界面	161
14.3.1 控制台类 <code>AfConsole</code>	162
14.4 随机数	163
14.4.1 示例 1	164
14.4.2 示例 2	165
14.4.3 示例 3	165
14.5 字符	166
14.5.1 字符的写法	167
14.5.2 字符编码	167
14.5.3 字符算术	168
14.5.4 字符与字符串	169

14.6 关于 equals	169
第 15 章 链表.....	173
本章学习目标	173
15.1 容器.....	173
15.2 链表.....	174
15.2.1 链表的构造	175
15.2.2 几个概念	176
15.2.3 链表的遍历	176
15.3 插入节点	177
15.3.1 添加到末尾	177
15.3.2 添加到前面	178
15.3.3 添加到指定节点之后	179
15.4 有头链表	179
15.4.1 有头链表的构造	180
15.4.2 有头链表的遍历	180
15.4.3 向有头链表里插入节点	181
15.4.4 从有头链表中删除节点	181
15.5 链表与容器	182
15.5.1 容器的实现	183
15.5.2 容器的使用	184
15.5.3 黑盒设计	185
15.6 ArrayList	185
第 16 章 学生管理系统	187
本章学习目标	187
16.1 系统演示	187
16.2 建立项目	189
16.3 命令行界面	191
16.4 数据记录的管理	193
16.5 查找与删除	197
第 17 章 Java 开发环境	错误!未定义书签。
本章学习目标	错误!未定义书签。
17.1 JDK 的安装与配置	错误!未定义书签。
17.1.1 JDK 的配置	错误!未定义书签。
17.2 Eclipse 与 JDK	错误!未定义书签。
17.3 Java 官方文档	错误!未定义书签。
第 18 章 class 与 jar	错误!未定义书签。
本章学习目标	错误!未定义书签。
18.1 class 文件	错误!未定义书签。

18.2	jar 文件的生成	错误!未定义书签。
18.3	jar 文件的使用	错误!未定义书签。
18.4	Java 系统库	错误!未定义书签。
第 19 章	图形界面	错误!未定义书签。
	本章学习目标	错误!未定义书签。
19.1	窗口程序	错误!未定义书签。
19.2	容器与控件	错误!未定义书签。
19.3	容器的布局	错误!未定义书签。
19.4	自定义的窗口类	错误!未定义书签。
第 20 章	抽象类	错误!未定义书签。
	本章学习目标	错误!未定义书签。
20.1	抽象类的定义	错误!未定义书签。
20.1.1	抽象方法的特点	错误!未定义书签。
20.1.2	抽象类的属性	错误!未定义书签。
20.2	抽象类的使用	错误!未定义书签。
20.2.1	抽象类不能实例化	错误!未定义书签。
20.2.1	抽象类的使用	错误!未定义书签。
第 21 章	接口	错误!未定义书签。
	本章学习目标	错误!未定义书签。
21.1	接口	错误!未定义书签。
21.1.1	接口的特点	错误!未定义书签。
21.1.2	接口的使用	错误!未定义书签。
21.2	接口与继承	错误!未定义书签。
21.3	接口的应用	错误!未定义书签。
21.4	按钮事件处理	错误!未定义书签。
第 22 章	内部类	错误!未定义书签。
	本章学习目标	错误!未定义书签。
22.1	内部类	错误!未定义书签。
22.2	当前外部对象	错误!未定义书签。
22.3	内部类的优势	错误!未定义书签。
22.4	匿名内部类	错误!未定义书签。
22.5	静态内部类	错误!未定义书签。
第 23 章	泛型	错误!未定义书签。
	本章学习目标	错误!未定义书签。
23.1	泛型	错误!未定义书签。
23.1.1	泛型的标准写法	错误!未定义书签。
23.1.2	泛型的简化写法	错误!未定义书签。
23.1.3	要点与细节	错误!未定义书签。

23.2	ArrayList	错误!未定义书签。
23.2.1	ArrayList 的遍历	错误!未定义书签。
23.2.1	ArrayList 的排序	错误!未定义书签。
23.3	Iterator	错误!未定义书签。
23.3.1	Iterator 与 for 循环	错误!未定义书签。
23.4	HashMap	错误!未定义书签。
23.4.1	HashMap 与 ArrayList	错误!未定义书签。
23.4.2	Key 的选择	错误!未定义书签。
23.4.3	HashMap 的更多 API	错误!未定义书签。
第 24 章	异常	错误!未定义书签。
	本章学习目标	错误!未定义书签。
24.1	出错处理	错误!未定义书签。
24.2	异常机制	错误!未定义书签。
24.2.1	抛出异常	错误!未定义书签。
24.2.2	捕获异常	错误!未定义书签。
24.3	自定义异常	错误!未定义书签。
24.3.1	派生 Exception	错误!未定义书签。
24.3.2	抛出自定义异常	错误!未定义书签。
24.3.3	捕获自定义异常	错误!未定义书签。
24.4	异常运行规则	错误!未定义书签。
24.5	退出清理	错误!未定义书签。
24.6	非检查异常	错误!未定义书签。
第 25 章	日期与时间	错误!未定义书签。
	本章学习目标	错误!未定义书签。
25.1	时间的表示	错误!未定义书签。
25.1.1	long 与 Date	错误!未定义书签。
25.2	时间的格式化	错误!未定义书签。
25.3	历法计算	错误!未定义书签。
25.3.1	创建 Calendar 对象	错误!未定义书签。
25.3.2	获取年月日时分秒	错误!未定义书签。
25.3.3	设置年月日时分秒	错误!未定义书签。
25.3.4	往前或往后推算	错误!未定义书签。
25.3.5	与时间值的转换	错误!未定义书签。
25.3.6	差值计算	错误!未定义书签。
25.3.7	计算 2020 年的父亲节	错误!未定义书签。
第 26 章	文件与目录	错误!未定义书签。
	本章学习目标	错误!未定义书签。
26.1	文件操作	错误!未定义书签。

26.2	目录操作	错误!未定义书签。
26.2.1	目录的扫描	错误!未定义书签。
26.2.2	指定过滤器	错误!未定义书签。
26.2.3	递归扫描	错误!未定义书签。
26.3	绝对路径与相对路径	错误!未定义书签。
26.3.1	绝对路径	错误!未定义书签。
26.3.2	相对路径	错误!未定义书签。
26.3.3	工作目录	错误!未定义书签。
26.4	复制与移动	错误!未定义书签。
第 27 章	文本文件	错误!未定义书签。
	本章学习目标	错误!未定义书签。
27.1	了解文件	错误!未定义书签。
27.2	读写数据	错误!未定义书签。
27.2.1	文件的写入	错误!未定义书签。
27.2.2	文件的读取	错误!未定义书签。
27.3	文本的读写	错误!未定义书签。
27.3.1	String 的编解码	错误!未定义书签。
27.3.2	字符串写入	错误!未定义书签。
27.3.3	字符串读取	错误!未定义书签。
27.4	对象的存储	错误!未定义书签。
27.4.1	对象的写入	错误!未定义书签。
27.4.2	对象的读取	错误!未定义书签。
27.4.3	TextFile 工具类	错误!未定义书签。
全 27 章,	已完结!	201

第 1 章 Hello World

本章学习目标

- 准备 Java 的开发环境
- 编写第一个 Java 程序
- 了解打印、注释与空白的用法

本章将安装和配置 Java 所需的开发环境 Eclipse，并在 Eclipse 上创建创建第一个 Java 程序。然后运行程序，观察结果，初步了解 Java 代码的规则。

1.1 课程概述

Java，是一种计算机编程语言。所谓编程语言，是用来编写生成程序的一种工具。使用 Java 可以开发出来各种各样的程序。

计算机程序，在日常使用电脑的时候随处可见。比如，我们电脑上的 QQ、谷歌浏览器等，都叫做应用程序。除此之外，手机上运行的各种 APP 应用也是程序。那么这些程序又是怎么出来的呢？我们即将要学习的编程语言，就是用来编写这些程序的工具。

这就好比，普通人都是可以学会开汽车。但汽车是由谁制造出来的呢？汽车工程师的工作是制造汽车。与此类似，软件工程师的工作则编写程序。

1.2 开发环境的安装

集成开发环境（IDE，Integrated Development Environment），就是一种用于开发程序的工作平台。工欲善其事，必先利其器。对于汽车工程师来说，要造汽车需要有工具、有车床、有车间，需要一整套的工作环境。对于软件工程师来说，也得

有一整套工具。

Java 的主流开发环境有四家平台，分别为 Eclipse, NetBeans, MyEclipse, IntelliJ Idea。本教程选用 Eclipse 作为演示平台。关于 Eclipse 环境的准备，请参照视频教程进行。

1.3 创建第一个程序

打开 Eclipse，创建第一个程序。示例代码如下。

```
package my;

public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("阿发，你好");
    }
}
```

第一个程序非常简单，只有寥寥几行代码。对于初学者来说，暂不需要深究每一个细节的含义。先把整个框架抄下来，运行一遍。以后每一章每一节按照循序渐进的顺序，逐步介绍每一个细节的意思。

在抄写这个例子，需要注意几点规则：

- 若非特殊说明，所有的标点符号都是英文半角的标点；
- 每个标点符号都有语法意义，不要漏掉。比如，末尾的分号。

对于初学者来说，在相当长的一段时间内，都是使用相同的代码框架。请将注意力放在大括号内部的部分代码。如同人类的语言一样，Java 语言的学习亦是一个循序渐进的过程，切不能急于求成。初学者只需照抄这个代码框架，而不要在第一节课就打破砂锅问到底。

在照抄示例成功之后，便可以对示例稍加改动进行尝试。例如，

```
package my;

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

这样的小幅改动，自然是可以的。其实如果出错也没关系，直接退回之前的样子就可以了。按 **CTRL+Z** 键，即可以退回到修改之前的状态。在初学阶段，宜小修小改，不宜有大动作。

1.4 打印，注释与空白

1.4.1 打印

在前面的示例代码中，有一行代码，

```
System.out.println("阿发，你好");
```

其中, `System.out.println()` 的作用是将一段文本输出显示到控制台窗口。运行程序时，可以观察到在 Eclipse 的下面的 **Console** 面板里的输出。如图 1.1 所示。

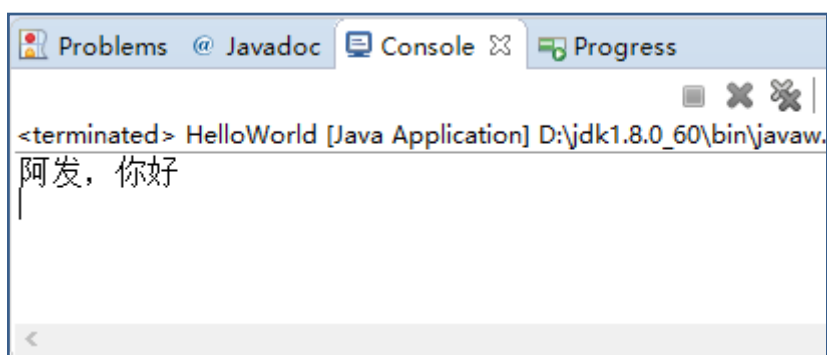


图 1.1 控制台输出显示

可以将多段文字，或者文字和数字拼接起来再输出，示例代码如下。


```
package my;

public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("我是邵发, 今年" + 35 + "岁" );
        System.out.println("分数:" + 99.5 );
        System.out.println("我是邵发, 今年" + 35 + "岁" );
        System.out.println("数学:" + 140 + ", 英语:" + 130 );
    }
}
```

其中，双引号包围的部分称为字符串。可见，使用加号可以把字符串、数字拼接起来，合为一个较长的字符串。

目前在练习的时候，只需要做最简单的练习就足够了，不需要深入。这种写法在后面会反复地出现，大概几天之后就习以为常了。

1.4.2 注释

在 Java 代码里，可以添加一些起注释作用的文字。示例代码如下。

```
// 下面是一个打印输出的练习

System.out.println("我是邵发, 今年" + 35 + "岁" );

// 更练习一个

System.out.println("I am 好人" ); // 哈哈
```

其中，以双斜杠开始到行末尾的部分，称为注释， 或称单行注释。

再看一个例子，

```
/* 下面是一个打印输出的练习

    这是我第 2 天学习了

    我很开心
```

```
*/  
  
System.out.println("我是邵发, 今年" + 35 + "岁" );  
  
System.out.println("I am 好人" ); /* 哈哈 */
```

其中, 以 `/*` 和 `*/` 包含的为多行注释。在本教程中, 一般采用单行注释的写法。

注释有什么用呢? 它是对代码起备注说明作用的, 目的是增强代码的可读性。比如说, 当把代码交给别人看的时候, 他只看代码可能会看得不太明白, 但是结合注释来看就容易多了。因此, 良好的注释可以增加代码的可读性。

那么, 一定要加注释吗? 不是的。一段代码可以没有任何注释, 也可以添加很多注释。无论有没有注释, 对程序本身的功能是没有任何影响的。

1.4.3 空白

在 Java 语言里, 为了让代码更易于阅读, 从美观上考虑, 可以添加空格、换行、与制表符。这三种符号统称为空白。

理论上, 一段代码中可以任意地添加一些空白, 并不影响代码最终的运行结果。但是, 为了代码的可读性, 一般要求代码按照习惯的方式来缩进。一段乱糟糟的代码虽然也可以运行, 但是在实际项目中是要绝对杜绝的。下面给出两段代码, 它们的功能相同, 只是在书写格式上有区别。

下面是一段格式差劲的代码,

```
public class HelloWorld { public  
  
static  
  
void main  
  
(String[] args){ System.out.println("I am 好人" ); }}
```

再来看一段格式美观的代码,

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("I am 好人");
    }
}
```

这两段代码虽然视觉上差异很大，但仔细观察，可以发现只是书写格式上的差异。它们的运行结果完全相同，但是前者格式很乱、无法读懂，后者则清晰易读。通过它们的对比，大家已经明白了代码的可读性是一件多么重要的事情。

『Java 学习指南系列教程』

作者： 邵发

官网： <http://afanihao.cn>

QQ 群： 495734195

本系列教程由 24 篇以上视频教程组成，从入门语法到行业级技术，循序渐近式的全方位教程。内容包含入门语法和高级语法，覆盖 Java 在业界的 3 个应用领域（网站开发、安卓 APP 开发、桌面 GUI 开发）。同时包含专项技术的培训教程，如网络编程基础、数据库开发，FreeMarker, Spring, MyBatis 等。

第 2 章 变量

本章学习目标

- 了解变量的概念
- 学会变量的命名和定义
- 初步学会 `int`、`double` 和 `boolean` 类型的定义

变量 (Variable)，是编程语言里的基本概念。本章先以一个引例来引入变量的概念，然后介绍变量的命名和定义，以及三种基本类型 `int`、`double`、`boolean` 的使用方法。

2.1 使用变量

2.1.1 引例

现在，已经学会了使用 `System.out.println()` 来输出一个值。进一步地，也可以用它来显示简单算术运算的结果。例如，

```
System.out.println("乘积: " + (123 * 456) );
```

其中，小括号中 `123*456` 表示计算 123 与 456 的乘积。此行代码能够将两者的乘积计算出来，并打印输出到控制台窗口。

进一步地，可以再写几行代码，来计算一个数的平方与立方的值。例如，

```
System.out.println("数值: " + 12 );  
System.out.println("平方: " + (12* 12) ); // 平方  
System.out.println("立方: " + (12* 12* 12)); // 立方
```

看起来写程序也不是太难，是不是呢？

但现在存在一个问题：如果求 22 的平方与立方呢？如果不嫌麻烦，可以修改上面的代码，修改成以下的样子。

```
System.out.println("数值: " + 22);  
System.out.println("平方: " + (22* 22) ); // 平方  
System.out.println("立方: " + (22* 22* 22)); // 立方
```

可见，当数值改变时，需要修改 6 处才能够得到新的结果。这样做虽然可以实现，但是显然是有些麻烦的！为了解决这个问题，引入一个新的概念：变量。

2.1.2 变量

变量（Variable），就是“可以变化的量”。

例如，

```
int year =2017;
```

这行代码就定义了一个变量，名字叫 `year`，类型为 `int`，初始值为 2017。其中，`int` 是 `Integer` 的综写，表示一个整数。

再来见识一下更多的写法，

```
int a = 123;  
int year =2017;  
int yue = 7;  
int dongaohuishinanian=2022;  
int bbb333 = 1234590;  
int days_of_2017 = 365;
```

可见，定义一个变量需要指定三个要素：名称、类型、值，如图 2.1 所示。

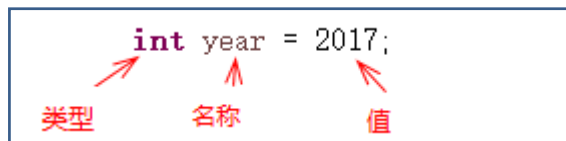


图 2.1 变量的 3 要素

2.1.3 变量的命名

一个变量需要有一个合适的名字。关于命名，需要记住这几条基本的规则：

- 只能包含英文字母、数字、下划线（但不能数字开头）
- 不一定要用英文单词，用拼音也是可以的
- 随便起个名字也可以，但是可读性差
- 区分大小写，`Year` 和 `year` 是不同的变量

下面来看几个变量的定义，观察它们的命名是否正确。示例如下，

```
int 2a = 0; // 错误，不能以数字开头  
int b2 = 1; // 正确，字母开头是没问题的  
int _c3 = 100; // 正确，可以用下划线开头  
int d4-size = 101; // 错误，不可以用横杠
```

2.1.4 变量的简单运算

下面，学习一下变量的简单算术运算：加法，减法、乘法。基本上和小学算术的用法类似，只是在这里用星号来表示乘法。目前暂不考虑除法运算。

- 加法 例 `a = b + 10;`
- 减法 例 `a = b - 10;`
- 乘法 例 `a = b * 10;`

示例代码如下。

```
int b1 = 123;  
int b2 = b1 + 10;  
int b3 = ( b1 + b2 ) * 199;  
System.out.println("b3 的值是" + b3 );
```

看起来没有什么特别的，就是小学算术，一分钟就可以学完了，不是吗？

下面，就可以使用变量的概念来解决引例中的问题，示例代码如下。

```
int a = 12;

int a2 = a * a;    // 平方

int a3 = a * a * a;    // 立方

System.out.println("数值: " + a );

System.out.println("平方: " + a2 );

System.out.println("立方: " + a3 );
```

其中，定义了一个变量 `a`，值为 12。如果要更换计算数值，只需修改一行代码即可。比如，要求计算 22 的平方与立方，则大部分代码不需要修改，只需修改第一行代码。示例如下，

```
int a = 22;    // 修改这一行即可，其他代码不变

int a2 = a * a;

int a3 = a * a * a;

System.out.println("数值: " + a );

System.out.println("平方: " + a2 );

System.out.println("立方: " + a3 );
```

显然，通过变量的定义，简化了问题的实现。当然，变量的存在不仅仅是这一点作用，它的其他用法后续会作进一步的介绍。

2.2 整数，小数，字符串

2.2.1 整数与小数

在 Java 语言里，用 `int` 类型来表示整数。其中，`int` 是 `Integer` 的简称。下面定义几个 `int` 类型的变量，示例代码如下。

```
int a = 12345;

int b = -294;    // 可以表示负数
```

不难发现，使用 `int` 可以表示正整数，也可以表示负整数。

另外，用 `double` 类型表示小数，示例代码如下。

```
double c = 123.456;

double d = -349;    // 相当于-349.0
```

其中，`double` 英文原意为 **Double Float Point**，表示双精度浮点型。浮点型，是一个计算机硬件相关的术语，在这里不必深究其背景意义。作为初学者，只需要知道用 `double` 可以表示一个小数就足够了。

除此之外，还需要知道无论是 `int` 还是 `double`，都只能表示有限的数字范围。

例如，当用一个 `int` 表示一个整数时，是有上限的。

```
int a = 12345; //OK

int b = 111222333444555; // 错误！数字太大，超出范围
```

其中，用 `int` 表示 12345 这么大的整数是可以的，但是表示 111222333444555 这么大的整数就超出了范围。那么，一个 `int` 能表示的整数的范围是多大呢？Java 里规定这个范围是 -2147483648~2147483647，约 21 亿左右。`double` 能表示的小数的范围也是很大的。

关于这个数值范围，有个感性的认识即可，并不需要精确地记忆。在入门教程里，`int` 和 `double` 都是够大的，是足够练习使用的。

2.2.2 字符串

在 Java 语言中，用 `String` 类型来表示一个字符串。

例如，

```
String name = "邵发";
```

其中，定义了一个变量，名称为 `name`，类型为 `String`，值为"邵发"。

现在来做一个小练习：一个同学的名字叫小张，今年 18 岁，体重 67.8kg，生日 1994 年 8 月 3 日。要求用 Java 代码表示这些数值。示例代码如下。


```
String name = "小张";  
  
int age = 18;  
  
double weight = 67.8;  
  
String birthday = "1994-8-3";
```

其中，用 **String** 来表示字符串类型的值，用 **int** 表示整数类型的值，用 **double** 表示小数类型的值。

考虑一下：使用 **String** 不能表示年龄吗？例如，

```
String age = "18";
```

并非不可以，只是说不太合适。至于什么样的数据用什么样的类型来表示，这是个很自然的理解过程，在经过后面稍许的练习之后就自然会明白了。

2.2.3 变量的赋值

前面说过，变量就是可以变化的量。也就是说，一个变量的值是可变的。例如，

```
int a = 0;  
  
System.out.println("a:" + a );  
  
int b = 5;  
  
a = b * b; // 此处 a 的值发生变化  
  
System.out.println("a 变为:" + a);
```

其中，用等号来进行赋值操作。将等号右边的值计算出来，赋值左边的变量。这里，把等号称为“赋值操作符”。关于赋值操作，在下一章里有更详细的讲解。

可以看到，**a** 的初始值为 0，后面通过一个赋值操作修改了 **a** 的值，变成 **b*b** 的结果。这就是变量的赋值操作。

2.3 布尔类型

2.3.1 引例

要求用变量表示以下的值。同学的名字叫小张，具体信息如下。

- 年龄：18 岁，体重：67.8kg
- 生日：1994 年 8 月 3 日
- 性别：男

其中，年龄可以用 `int` 表示，体重用 `double` 类型表示，生日暂时用 `String` 类型表示。难点在于性别这一项。该用什么类型来表示性别呢？

第一种办法：用 `String` 型表示。

```
String sex1 = "male"; // 规定 "male"表示男, "female"表示女
```

第二种办法：用 `int` 型表示。

```
int sex = 1; // 规定 1 表示男, 0 表示女
```

无论是用 `String`，还是用 `int`，都存在一个缺点：可能存在非法的值。例如，如果不小心把 `sex` 设为-1，那就没法理解了。

```
int sex = -1; // 一个非法的值，但编译器不会报错
```

所以，最好有一种新的类型来表示这种非此即彼、二选一的值，就是 `boolean` 类型。

2.3.2 boolean 类型

在 Java 语言里，`boolean` 类型表示“是”、“否”这种二选一的值。一般译作布尔类型。例如，

```
boolean a = true;
boolean b = false;
```

其中，按照变量定义三要素，`boolean` 是变量类型、`a` 是变量名称、`true` 是变量的值。规定 `boolean` 类型的值只能是两种：`true` 或 `false`。

以下写法是错误的，

```
boolean c = 1; // 错误！只能是 true 或 false !
boolean d = "yes"; // 错误！只能是 true 或 false !
```

实际上，`true` 和 `false` 在 Java 语言里都是特殊语义的词，称为关键词。

现在，使用 `int`, `double`, `String`, `boolean` 这四种类型，便足以表示前面引例中规定的问题。示例代码如下。

```
String name = "小张";  
  
int age = 18;  
  
double weight = 67.8;  
  
String birthday = "1994-8-3";  
  
boolean sex = true; // true 表示男性, false 表示女性
```

`boolean` 类型是 Java 初学者的第一道思维门槛，在学习的时候不要追求一撮而就、一步到位。人类的学习过程是曲折的、迭代的，在第一天、第二天的时候可能会懵掉，但是第三天第四天之后可能就会慢慢习惯它。

可以把 `boolean` 和前面学过的 `int` 对比一下，形式上是类同的。示例如下，

```
int a = 1;  
  
int b = 0;  
  
boolean c = true;  
  
boolean d = false;
```

从形式上看，对于 `int` 类型来说，它的值是整数，可以为 1 或 0。而对 `boolean` 来讲，它的值就只能为 `true` 或 `false`。要记住，`true` 和 `false` 是值，而不是字符串！（以双引号包围的才是字符串）。

2.3.3 常见问题

（1）`boolean` 变量能加减乘除吗？

不能。可以类比一下，`String` 类型也不能乘法。

```
String a= "good";  
  
String b= "bad";  
  
String c = a * b ; //????? 这啥意思，完全没意义嘛！
```

（2）`boolean` 变量还能赋第 3 种值吗？

不能。boolean 类型的值要么为 true, 要么为 false

(3) boolean 值如何打印输出?

boolean 类型的值在打印输出时, 显示为 true 或 false

例如,

```
boolean sex = true;  
  
System.out.println("性别: " + sex ); // 性别: true
```

2.4 其他类型

在 Java 语言里, 还存在其他的数据类型, 如 byte, short, float, char 类型。对于初学者, 不要求一次性的全部掌握。在本篇教程中, 用的最多的就是 int, double, String, boolean 这 4 种类型。使用这四种基本类型, 便足够完成语法篇章的学习了。关于其他类型, 在后续的课程里都会逐步介绍的, 不必着急。

第 3 章 操作符与表达式

本章学习目标

- 了解操作符和表达式的概念
- 学会算术操作符的使用
- 学会关系操作符的使用
- 学会逻辑操作符的使用
- 学会赋值操作符的使用
- 理解类型转换与提升操作

操作符（Operator），也称为运算符，比如 `+` `-` `*` `/` 这些符号都是操作符。

表达式（Expression），是一个由数值和操作符组成的式子。把所有的变量的值代入表达式，计算出的最终的值，称为表达式的值。本章介绍 Java 里最主要的几种操作符的使用方法。

3.1 算术操作符

本节学习以下 5 个操作符，用于实现算术运算，如表 3.1 所示。

表 3.1 算术操作符

操作符	名称	作用
<code>+</code>	加	加法运算
<code>-</code>	减	减法运算
<code>*</code>	乘	乘法运算
<code>/</code>	除	除法运算
<code>%</code>	模	模运算

这几个符号其实和小学算术里的内容是基本对应的，有区别的是除法和模运算。

3.1.1 整型的算术运算

假定在以下式子中， a, b 为 `int` 型。

$a + b$ 求和

$a - b$ 求差

$a * b$ 求积

a / b 求商 (结果不保留小数)

$a \% b$ 求余

规定两个 `int` 型在算术运算后，结果仍为 `int` 型。运算规则和小学算术基本一样，下面只需要解释一下除法和模运算。

(1) 整型的除法运算，结果舍去小数部分

例如， $14 / 5$ 的值为 2。

```
System.out.println( "结果为: " + ( 14 / 5 ));
```

(2) 整型的模运算，结果为余数

例如， $14 \% 5$ 的值为 4。

```
System.out.println( "结果为: " + ( 14%5 ));
```

3.1.2 浮点型的算术运算

对于小数的算术运算，规则也是类似的。重点在于它的除法和模运算。

(1) 浮点型的除法运算，结果保留小数部分

例如， $14.0 / 5.0$ 的值为 2.8。

```
System.out.println( "结果为: " + ( 14.0 / 5.0 ));
```

(2) 浮点型的模运算，结果为余数

例如， $14.0 \% 5.0$ 的值为 4.0。

```
System.out.println( "结果为: " + ( 14.0 % 5.0 ) );
```

3.1.3 练习

比较一下，这两个式子的结果为什么不同？

$14 / 5$

$14.0 / 5.0$

其中，

$14 / 5$ ，是两个 `int` 型的运算，所以结果仍然为 `int` 型，值为 2。

$14.0 / 5.0$ ，是两个 `double` 型的运算，所以结果仍然为 `double` 型，值为 2.8。

3.1.4 类型提升

考虑以下式子：

```
int a = 14;

double b = 5.0;

System.out.println( "结果为: " + ( a / b ) );
```

一个 `int` 型与一个 `double` 型进行除法运算，那么结果是 2 还是 2.8 呢？

规定：当 `int` 型与 `double` 型混合运算时，把 `int` 型视为 `double` 型，称为类型的提升。

这是因为，`double` 型是可以兼容整数的表示的，所以都提升为 `double` 型来运算比较好，不会丢失数据。所以，上式 a/b 的结果按 `double/double` 来进行，结果为 2.8。

3.1.5 混合运算与优先级

当一个式子中包含多种符号时，需要考虑优先级问题。

举一个最简单的例子，

$$a * b + c / d$$

在这个式子里有 3 种运算符：乘、加、除。那么，在运算的时候，谁先执行、谁后执行是一个要考虑的问题。和直观理解的一致，乘、除的优先级是略高于加的。

所以，此表达式在运算的时候，是按以下的顺序

- ① $a * b$
- ② c / d
- ③ 上两式结果相加，即 $(a * b) + (c / d)$

通过这个小例子，可以感受到优先级的问题。在 Java 里规定了 * 和 / 的优先级要高于 +，所以才有了上面的结果。关于优先级的问题，本章后面还有进一步的介绍。

3.2 关系操作符

本节学习以下几个操作符，用于计算数值的大小关系，如表 3.2 所示。

表 3.2 关系操作符

操作符	名称	作用
<	小于	
<=	小于或等于	
>	大于	
>=	大于或等于	
==	等于	
!=	不等于	

关系表达式的值为 **boolean** 类型。例如：

```
int a = 10, b=4;
```



```
boolean result = a > b;  
  
System.out.println (" 结果: " + result );
```

其中，将 a,b 的值代入表达式值，计算得 result 的值为 true 。运行以上代码，输出如图 3.1 所示。

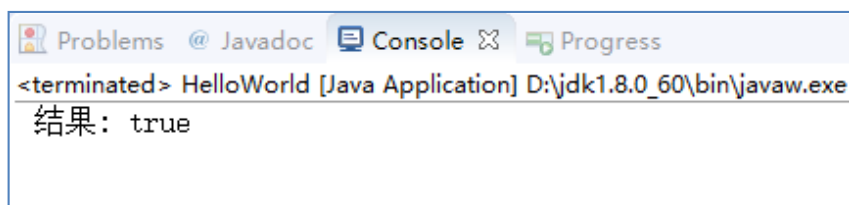


图 3.1 关系操作符输出显示

3.2.1 关系操作符的运算

对于关系表达式，只需要把相应的值代入式子，从逻辑上判断一下。如果是成立的，则值为 true；如果不成立，则值为 false。

例如，(10 > 4)，此式成立，所以其值为 true。

例如，设 a 为 10，则 (a < 5) 不成立，所以其值为 false。

可以直接在 println()里打印输出一下式子的值，示例如下。

```
int a = 10;  
  
System.out.println (" 结果: " + ( a < 5 ) );
```

注意，在 println 里的表达式最好加上小括号，以显式地指定运算的优先级。

3.2.2 更多练习

以下表达式的值是？

4 > 5 false

4 >= 4 true （注: >= 表示大于或等于）

3 < 4 true

`3 != 4` `true`

`3 != 3` `false`

`8 == 7` `false`

`8 == 8` `true`

以下表达式的值是？假设 `int a = 8, b = 7`。

`a > 8` `false`

`a == b` `false`

`a <= b` `false`

`a >= b + 1` `true`

以下表达式的值是？假设 `double a = 11.2, b = 1`。

`a > 8` `true`

`a == b` `false`

`a <= b` `false`

`a >= b + 1` `true`

关系运算符的计算是比较容易理解的。只需要把变量的值代入式子，看成不成立。成立的话就是 `true`，否则就是 `false`。

3.2.3 注意事项

(1) 操作符是一个整体，中间不能加空格

例如 `>=` `<=` `==` `!=`

(2) 对于初学者来说，注意区分 `==` 和 `=`

编程并不是数学，不要把数学的习惯带过来。当判断是否相等时，要用 `==` 来判断。

3.3 逻辑操作符

本节学习以下几个操作符，用于判断逻辑关系，如表 3.3 所示。

表 3.3 逻辑操作符

操作符	名称	作用
&&	逻辑与	并且
	逻辑或	或者
!	逻辑非	非

3.3.1 逻辑与 &&

这个操作符用于表示“并且”的关系。形式如下，

`a && b`

其中，a,b 均为 boolean 类型。当 a 成立并且 b 成立时，结果才成立(true)。否则，结果不成立(false)。

通俗地讲，就是表示日常生活中“并且”的逻辑。比如，一件事情需要 a, b 同时在场才能处理，那就是并且的关系。

练习：求表达式 `(3 > 4) && (4 > 3)` 的结果。

- ① 按优先级先计算 `3>4` 为 false, `4>3` 为 true
- ② 原式即为 `false && true`，所以结果为 false

3.3.2 逻辑或 ||

这个操作符用于表示“或者”的关系。形式如下，

`a || b`

当 a 成立或者成立时，结果成立(true)。否则，结果不成立(false)。

通俗地讲，就是表示日常生活中“或者”的逻辑。比如，一件事情 a, b 均能处理，则只要 a 在场或者 b 在场即可，这就是或者的关系。

练习：求表达式 $3 > 4 \parallel 4 > 3$ 的值

① 按优先级先计算 $3 > 4$ 为 false, $4 > 3$ 为 true

② 原式即为 $\text{false} \parallel \text{true}$ ，所以结果为 true

3.3.3 逻辑非 !

逻辑非表示“否”。形式如下，

!a

其中，如果 a 为 true，则结果为 false；如果 a 为 false，则结果为 true。

练习：求表达式 $\text{!}(3 > 4)$ 的值

① 按优先级先计算 $3 > 4$ 为 false

② 原式即为 !false ，所以结果为 true

3.3.4 注意事项

(1) 操作符是一个整体，中间不能加空格

例如 $\&\& \parallel$

(2) 操作符与操作数之间，可以加空格

比如， !k 也可以

3.4 赋值操作符

在 Java 语言里，等号称为赋值操作符。例如，

```
a = b + 100;
```

其中，不要把 Java 语言理解为数学。在 Java 里，这个等号的作用是“赋值”，就是把右侧的值赋给左边的变量。

注意事项如下：

- 等号左边必须是变量；
- 将等号右边表达式的值，赋给左边的变量。

3.4.1 区分初始值与赋值

在定义一个变量的时候，可以指定初始值。例如，

```
int a = 10; // 定义一个变量 a，初始值为 10  
int b; // 如果未指定初始值，则初始值为 0
```

下面演示一下赋值操作。

```
int a = 10; // 此行的等号为初始化  
int b = 22; // 此行的等号为初始化  
a = b + 100; // 此行的等号表示赋值操作
```

3.4.2 赋值操作的过程

赋值操作分为 2 步运行：

- ① 先计算等号右侧的式子的值；
- ② 再把右侧的值赋给左侧的变量。

例如，

```
int a = 10;
```

```
a = a + 99; // 重点理解这一行的赋值操作
System.out.println("a 的值为 " + a );
```

来分析一下，`a = a + 99` 这一行是分两步来运算的：

- (1) 首先，计算出右侧表达式 `(a+99)` 的值，为 109；
- (2) 然后，将右侧的值赋给左侧的变量，即 `a=109`。

3.4.3 组合赋值操作符

赋值操作符可以和其他操作符组合起来使用，例如，

```
a += b;
```

```
a -= b;
```

```
a *= b;
```

```
a /= b;
```

```
a %= b;
```

规则都是一样的，`a+=b` 相当于 `a = a + b`。

示例：

```
int a = 10;

a += 12; // 相当于 a = a + 12

System.out.println("现在 a 为: " + a); // a 的最终值为 22
```

3.5 自增操作符

在 Java 语言里，`++` 称为自增操作符。例：

```
int a = 10;

a++;

System.out.println("现在 a 为: " + a);
```

其中，`a++` 可以理解为 `a = a + 1`。所以，最终 `a` 的值为 11。

3.5.1 前置自增与后置自增

自增操作符有 2 种写法，写在前面的称为前置自增，写在后面叫后置自增。例如，

后置自增：

```
int b = 10;

a = b++ ;
```

其中，后置自增时，先把 **b** 的值代入式子运算，之后再对 **b** 自增加 1。所以，最终的结果 **a** 为 10, **b** 为 11。

前置自增：

```
int b = 10;

a = ++b ;
```

其中，前置自增时，先对 **b** 自增加 1，然后再代入式子运算。

前置自增与后置自增分辨起来有点烧脑，但它并不是 Java 编程的重点，所以不要在这个知识点上花太多时间。

通常为了增加代码的可读性，可以分成 2 行书写。例如，

```
a = b;

b ++;    // 分成两行来写，以消除可能的误解
```

3.5.2 自减操作符

同样的，还有一套自减操作符 **--**，例如，

```
int a = 0;

int b = 10;

a = b --;    // 先代入运算，后自减。所以 a 变成 10，b 变成 9。
```

再次强调，++和--不是学习的重点，不要过分地研究和练习。

3.6 操作符的优先级

当一个表达式里有多种操作符时，就需要明确操作符的优先级顺序。也就是说，哪些部分先运算，哪些部分后运算。比如，

`a - b * 10 / 2`

这个表达式看起来很简单。之所以会觉得简单，因为事先知道了算术符的优先级。毕竟在小学时代就学过了算术运算的优先级，乘除法的优先级是高于加减法的。

但是，当把多种 Java 操作符综合在一起时，就不那么简单了。比如，

`a > 10 && !ok`

对于这样的式子，含有关系操作符 `>` 和逻辑操作符 `&&` `!`。要进行运算，就必须明确优先级。在这里，关系操作符比逻辑操作符的优先级高，而逻辑非又与逻辑与的优先级高所以，它的运算顺序是这样的：

`((a > 10) && (!ok))`

然而，在 Java 语言里，操作符约有几十种，想要记住它们的优先级顺序是不太现实的，也是不必要的。并不需要强记这些优先级，只要知道几种常用的操作符的优先级就足够了。当表达式较为复杂的时候，建议多加一些小括号，用小括号来明确指定运算的先后顺序。

3.7 类型转换操作符

在整型与浮点型之间，允许类型转换。例如，

```
double a = 12.77;

int b = (int) a; // 将 double 类型转成 int 类型

System.out.println("b 的值:" + b);
```

其中，

`a` 是 `double` 型，

`b` 是 `int` 型，

`b=a` 赋值时，要求左右两侧的类型相同。所以，可以把 `a` 转换成 `int` 型。也就是说，一个 `double` 类型可以转成 `int` 类型。把这对小括号称为类型转换操作符。

反过来，一个 `int` 类型也可以转换成 `double` 类型。例如，

```
int c = 10;

double d = (double) c; // 将 int 类型转成 double 类型
```

3.7.1 类型转换时的数据损失

在 `double -> int` 的转换过程中，数据可能发生损失。例如，

```
double a = 12.77;

int b = (int) a; // 存在数据损失
```

转换后的 `b` 的结果为 12，即小数部分被截断。

反过来，在 `int -> double` 的转换过程中，数据不会损失。例如，

```
int c = 10;

double d = (double) c; // 没有数据损失
```

3.7.2 显式转换与隐式转换

在类型转换的过程中，如果没有风险和数据损失，就可以把类型转换符省略。

例如，

```
int c = 10;

double d = c; // 省略 (double) 转换符
```

在 `d=c` 的过程中，发生了隐式的类型转换，相当于

```
int c = 10;

double d = (double)c ; // 加不加 (double)都可以
```

把这种自然而然的、没有任何风险的转换，称为隐式转换。如果可以隐式转换，就可以把转换符()省略不写。

3.7.3 注意事项

- (1) 小数转整数时，小数部分被截断。注意不是四舍五入！

```
double a = 12.9;

int b = (int) a;    // b 的值是 12，没有四舍五入
```

- (2) 默认其他类型是不能转换的

例如，**boolean** 与 **int** 是不能相互转换的。例如，

```
boolean k = true;

int a = (int) k; // 错！别乱写！
```

第 4 章 语句

本章学习目标

- 了解语句的基本概念
- 学会 if 语句的使用
- 学会 for 语句的使用
- 学会 while 语句的使用

语句 (Statement)，是程序的一条执行单元。所谓计算机程序，本质就是一条条的语句按顺序执行的意思。本章介绍语句的概念，并学会几种常见的逻辑控制语句。

4.1 语句的概念

先来了解一下语句的概念。语句 (Statement)，表示一行可执行的代码。例如，

```
package my;

public class HelloWorld
{
    public static void main(String[] args)
    {
        int a = 10;
        double b = 12.34;
        a = (int) (b * b);
        System.out.println("a=" + a );
    }
}
```

其中，大括号内有 4 条语句。它们的特点是，位置在 main 大括号里面，并且以

分号结束。

并非所有以分号结束的代码都叫语句，比如 `package my;` 这一行代码，由于位置不在 `main{ }` 这个大括号里，所以就不叫语句。

通常把一条语句单独写在一行里，但是也可以把多条语句凑在一行里书写。比如以下的代码，就很不美观，可读性很差。数一数在这段代码里，总共有几条语句呢？

```
a=10; b=11; c = 12;
```

虽然写在同一行里，但是它们是 3 条语句，因为每个分号就算是一条语句。

4.1.1 空语句

一个单独的分号也算是一条语句，称为空语句，也就是说这条语句什么也没干。

例如，

```
int a = 10;  
  
;  
  
;  
  
;  
  
a ++;
```

其中有 5 条语句，3 条是空语句。虽然只有一个分号，也算是一条语句的。

4.1.2 复合语句

可以把多条语句合在一个大括号里，构成一个复合语句。例如，

```
{  
  
    int a = 10;  
  
    a ++;  
  
}
```

其中，大括号内的语句合在一起可以视为一条语句，即复合语句。具体不必了

解太深，有一些印象即可。

4.2 if 语句

在 Java 里，用 if 语句来实现“当满足 XXX 条件时，执行 YYY”这样的逻辑判断。

例如，在使用共享单车时需要检查使用者的年龄。如果在 12 岁以下，则禁止骑行。用 Java 代码可以表示为，

```
int age = 11; // 年龄
if ( age < 12 )
{
    System.out.println("未满 12 岁，不能骑小黄车");
}
```

这里就使用了 if 语句。其基本形式如下，

```
if ( E1 )
{
    S1
}
```

其运行规则为：“当 E1 成立时，执行 S1”。也就是说，当 E1 的值为 true 时执行 S1 语句。

4.2.1 分步过程讲解

为了方便讲解，在每一行代码前面加了一个行号。

(1) 第一组测试

```
① int age = 11 ; // 年龄
② if ( age < 12 )
{
```

```
③      System.out.println("未满 12 岁，不能骑小黄车");  
      }  
④  System.out.println("结束");
```

运行过程如下：

运行 ①：age 的值为 11

运行 ②：先计算 $a < 12$ 的值为 true，所以会进入大括号内部运行

运行 ③：

运行 ④：结束

(2) 第二组测试

现在，我们把初始条件 age 设为 14，再运行一下：

```
①  int age = 14 ; // 年龄  
②  if ( age < 12 )  
    {  
③      System.out.println("未满 12 岁，不能骑小黄车");  
    }  
④  System.out.println("结束");
```

则运行过程如下：

运行 ①：age 的值为 14

运行 ②：先计算 $a < 12$ 的值为 false，所以直接跳出 if

运行 ④：结束

4.2.2 if 语句的两种基本形式

if 语句有两种基本写法，分别介绍一下。

第一种:

```
if ( E1 )
{
    S1
}
```

第二种: 表示“如果,。。。; 否则,。。。 ”这样的逻辑。如果 E1 成立, 就执行 S1; 否则, 就执行 S2。

```
if ( E1 )
{
    S1
}
else
{
    S2
}
```

下面就第二种写法给一个例子。

例如, 一个学生的分数用 `score` 来表示, 如果分数少于 60 分, 则判定为不及格; 否则, 则提示通过、并有奖励。代码如下,

```
int score = 77; // 分数
if ( score < 60)
{
    System.out.println("不及格!得重考啦! ");
}
else
{
    System.out.println("恭喜, 通过了! ");
    System.out.println("有奖励! ");
}
System.out.println("结束");
```

通过这个例子，可以发现大括号里其实是可以添加多条语句的。比如，在 `else{ }` 里，添加了 2 行语句，它们会依次执行。

4.2.3 if 语句的完全形式

用 `if... else if ... else if ... else` 可以表示“如果... 再如果 ... 再如果...否则”这样的多重判断的逻辑。形式如下，

```
if ( E1 )
{
    S1
}
else if ( E2 )
{
    S2
}
else if ( E3 )
{
    S3
}
else
{
    SS
}
```

在运行时，先判断 E1 是否成立；若 E1 不成立，再判断 E2 是否成立；依次判断，一直到最后一个条件。

下面举一个例子来说明这种多重条件的判断。例如，一个学生分数 `score`，若在 90 分以上记为 A，若在 80-90 分记为 B，若在 70-80 记为 C，若在 70 分以下记为 D。示例代码如下。


```

int score = 77;

if ( score >= 90 ) // 90 以上
{
    System.out.println("等级: A");
}

else if (score >= 80) // 80-90
{
    System.out.println("等级: B");
}

else if (score >= 70) // 70-80
{
    System.out.println("等级: C");
}

else // 70 以下
{
    System.out.println("等级: D");
}

System.out.println("结束");

```

其中，可能有人为问，第 2 重条件判断为什么不是以下这种形式呢？

```

else if ( score>=80 && scroe < 90 )
{
    ..... // 省略
}

```

这是因为，前端已经判断过了第一重条件 `score>90`，第一重条件已经不成立，才能走第二重条件这里。所以，后面就不用重复判断了。

4.2.4 常见问题

需要注意的是，在书写 if 语句的时候，末尾不要多加分号。如图 4.1 所示。

```
int score = 77;

if ( score >= 90 ) ;
{
    System.out.println("等级: A");
}
```

错! 不要加分号!

图 4.1 多余的分号

实际上 `if(){}` 是一个整体,如果在中间加个分号,就成了 `if();{}` 显然是不对的。

`if` 语句相对比较简单,容易理解。但对于 Java 编程来说,练习是非常重要的,再简单的知识点也要去练习一下才能有真正的理解。看书百遍,不如自己手写一遍!

4.3 for 语句

4.3.1 引例

下面介绍 `for` 语句。为了便于理解,先给出一个引例。

要求写一段代码,计算从 1 到 100 的平方之和。怎么实现呢?

```
int total = 0;

total += 1 * 1;

total += 2 * 2;

...写 100 行类似的语句 ...

total += 100 * 100;

System.out.println("结果为: " + total);
```

现在要实现这个功能的话,似乎只有老老实实在地、机械地写一百行代码了!

但是这么写的话,看起来就很不科学。Java 语言的设计有这么愚蠢吗?当然不会的。肯定有一种方式,可以快速地实现这种有规律的循环逻辑。那就是下面要介绍的 `for` 语句。

4.3.2 for 语句

在 Java 语言里，使用 for 语句来实现循环逻辑。其基本形式为，

```
for ( E1; E2; E3 )  
{  
    S1  
}
```

其中，

E1: 初始化表达式

E2: 前置表达式（循环条件判断）

E3: 后置表达式

S1: 循环体

其执行的步骤为：

- ① 运行初始化表达式 E1，仅执行一次
- ② 运行 E2（循环条件）：条件若成立，则执行循环；否则退出循环
- ③ 运行 S1（循环体）
- ④ 运行 E3（后置表达式）
- ⑤ 执行下一轮： ② ③ ④

简单的讲，一轮循环就是 E2 >> S1 >> E3，每轮循环之前先判断 E2 是否成立。

E2 为 true 则执行本轮循环，否则退出循环。

现在，用 for 语句来解决引例中的问题。示例代码如下。

```
int total = 0;  
  
int i;  
  
for ( i=1 ; i<=100 ; i++ )  
{  
    total += i * i;  
}
```

```
System.out.println("结果为: " + total);
```

其中，

E1 : i=1 将 i 初始化为 1

E2 : 每次判断 $i \leq 100$ 是否成立

S1 : 循环体为 $total += i * i$

E3 : 在循环体之后执行 $i++$

其中， $E2 \gg S1 \gg E3$ 构造一个循环。

4.3.3 更多例子

再举一个例子。要求计算 1 到 100 以内，能被 3 整除的数 (如 3 6 9 12 ...)。

示例代码如下，

```
int i;
for ( i=1 ; i<=100 ; i++ )
{
    if ( i % 3 == 0 )
    {
        System.out.println("Got: " + i );
    }
}
```

其中，在 for 的循环体里，使用了一个 if 语句来判断 i 的值是否为 3 的倍数。

若 $i \% 3 == 0$ ，则表明 i 就是 3 的倍数。

在这个例子可以看到，在 for 语句里是可以包含其他语句的 (如 if 语句)。

4.3.4 循环变量

通常，可以把用于循环迭代的变量直接定义在 for 的小括号里。例如，

```
for (int i=1; i<=100; i++) // 在小括号里定义循环变量 i
```

```
{  
    // 处理  
}
```

其中，把 `int i=1` 直接定义在初始化变达式里。这种专门用于循环的变量，有时候也可以称为循环变量。

4.4 for 语句的嵌套

在前面的例子里已经看到，`for` 语句里是可以嵌套包含 `if` 语句的。因为在理论上，`if(...){ ... }` 在整体上可以视为一条语句，即复合语句。

各种类型的语句，是可以互相嵌套的，这是很自然的现象。例如这种形式，

```
if ( ... )  
{  
    for (...) { }  
}
```

再复杂一点，像下面这种形式也可以，

```
for (...)  
{  
    for (...)  
    {  
        for(...) {}  
    }  
}
```

这些写法是自然的、也是很常见的，并没有固定的术语。为了方便教学，可以把这种现象通俗地称为嵌套。（嵌套并不是正式术语）。

4.4.1 例子

看一个例子。给定整数 n ，当 n 为偶数时从小到大输出， n 为奇数时从大到小输出。

示例代码如下，

```
int nnn = 11;

if( nnn % 2 == 0)
{
    // nnn 为偶数，从小到大输出
    for (int i=1; i<= nnn;i++)
    {
        System.out.print( i + " ");
    }
}
else
{
    // nnn 为奇数，从大到小输出
    for (int i= nnn; i >0; i--)
    {
        System.out.print( i + " ");
    }
}
```

其中，在 if 语句里，嵌套了 for 语句。因为理论上 for 语句是复合语句，可以视为一条语句。print() 用于输出文本，末尾不换行。而 println() 也用于输出文本，在末尾的时候会附加一个换行。

另外，for 语句也可以按从大到小递减的方式来实现循环逻辑。示例如下，

```
for (int i= nnn; i >0; i--)
{
```

```
..... // 略  
}
```

4.4.2 例子

再看一个例子，要求用*号输出以下形状的图形：

```
*  
  
**  
  
***  
  
****
```

先来分析一下，这样的图形有什么循环规律呢？

第 1 行有 1 个星

第 2 行有 2 个星

... 第 i 行有 i 个星

根据这个规律分析，用一个嵌套的循环（2 重循环）就可以实现这样的功能。

示例代码如下。

```
int N = 5;  
  
for ( int i=0; i<N; i ++ ) // 循环变量 i  
{  
    for(int k=0; k <=i; k++) // 循环变量 k  
    {  
        System.out.print("*");  
    }  
  
    System.out.print("\n"); // 输出换行  
}
```

其中，`print("\n")` 表示输出一个换行。

对于初学者来说，循环变量从 0 开始、还是从 1 开始，是一个问题。比如，以

下两种写法都是可以的，

```
for (int i=0; i<n; i++) // 从 0 开始，是计算机编程的惯例
{
}

for (int i=1; i<=n; i++) // 从 1 开始，也可以
{
}
```

但是，按照计算机编程的惯例，一般采用从 0 开始的写法。这种从 0 开始计数的写法是推荐的写法。初学者开始可能不太适应这种思维，但是在学完下一章后，就会慢慢习惯这种从 0 开始计数的思维方式。

4.5 break 与 continue

在循环体中，**break** 与 **continue** 可以控制循环的运行。其中，**break** 用于中断循环，**continue** 用于跳过本轮循环。

4.5.1 break 语句

先看一个例子。要求打印输出一批数值，当数值超过 10 时，停止打印。

```
for (int i=0; i<100; i++)
{
    if ( i >= 10)
    {
        break; // 运行到 break 时循环中止
    }

    System.out.println( i + " ");
}
```

当运行到 *i* 为 10 时，*i* >= 10 成立，执行 **break** 语句，退出循环。

4.5.2 continue 语句

例如，要求打印所有从 1 到 20 的数值，但是不包括 4 的倍数。

```
for(int i=1; i<=20;i++)
{
    if(i %4 ==0)
    {
        continue; // 跳过本轮，进入下一步
    }

    System.out.print( i + " ");
}
```

其中，当 $i\%4==0$ 时，`continue` 被执行，跳过本轮，进入下一轮。执行的结果如图 4.2 所示。

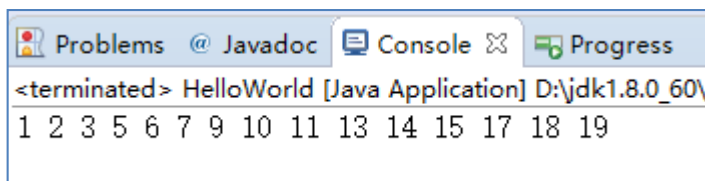


图 4.2 执行的结果

在使用 `break` 和 `continue` 要注意两点：

- `break` 和 `continue` 必须写在循环体里；
- 当存在嵌套时，`break` 中断的是上层循环，不是上上层循环。

4.6 for 语句的变形

下面来了解一下 `for` 语句的几种特殊形式。一般形式下，

```
for (E1 ; E2; E3)
```

```
{
    S1
}
```

在这里，E1, E2, E3 都是可以置空的。

最极端的情形，三者均为空，表示一个无限循环，示例代码如下。

```
for ( ; ; ) // E1,E2,E3 均置空
{
    System.out.println("in loop");
}
```

也就是说，即使小括号的三个部分均为空，也是符合语法规则的。

下面分别按三种情况，来讨论一下 E1,E2,E3 分别为空的三种变形。为了方便讨论，先给出一个标准形式的写法。以下代码，打印了从 0 到 9 的十个数字。

```
for (int i=0; i<10; i++)
{
    ////////// 循环体 //////////
    System.out.print(i + " ");
    //////////////////////////
}
```

4.6.1 初始化 E1 为空

小括号里的第一部分表示初始化，会在第一次循环之前运行。因此，在形式上可以将 E1 表达式提到 for 语句之前，并没有什么影响。

```
int i=0;
for ( ; i<10 ; i++) // E1 为空
{
    ////////// 循环体 //////////
    System.out.print(i + " ");
    //////////////////////////
}
```

```
}
```

其中，小括号里的第一部分初始化，被提到了 `for` 语句之前运行，效果没有差别。

4.6.2 循环条件 E2 为空

E2 如果为空，则默认为 `true`，表示每轮循环都会执行。例如，

```
for ( int i=0 ;      ; i++) // E2 为空
{
    ////////// 循环体 //////////
    System.out.print(i + " ");
}
```

其中，由于 E2 为空，所以这个循环会一直运行，没有休止。可以在循环体中加上条件判断，来控制循环的终止。例如，

```
for ( int i=0 ;      ; i++)
{
    if ( i>= 10)
    {
        break; // 控制循环退出
    }

    ////////// 循环体 //////////
    System.out.print(i + " ");
}
```

4.6.3 后置表达式 E3 为空

后置表达式 E3 在循环体之后运行，也可以置空。示例如下，

```
for ( int i=0 ; i<10 ; ) // E3 为空
{
```

```
////////// 循环体 //////////  
  
System.out.print(i + " ");  
  
i ++; // 放在循环体的下面执行，效果相同  
  
}
```

通过以上 3 种情况的比较，可以发现 for 语句的写法是可以灵活多变的。对于初学者来说，应以标准形式为主。对于这三种变形的情况，稍微了解即可。

4.7 while 语句

在 Java 里，while 语句也可以用于表示“循环”的逻辑。其一般形式为，

```
while ( E1 )  
{  
    S1  
}
```

运行规则为：“当条件 E1 成立的时候，执行循环体 S1；否则退出循环”。

4.7.1 例子

先给出一个简单的例子。要求打印 1 到 10 之间的数值，可以用 while 语句实现。示例代码如下。

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

其中，按照 while 语句的规则，每轮循环之前，要检查循环条件 $i \leq 10$ 是否成立。所以，此循环会执行 10 轮，然后结束退出。

4.7.2 while 语句的变形

while 语句也可以变形。例如，以上例子也可以写成这种形式，

```
int i = 1;

while ( true ) // 循环条件总是为 true
{
    if(i > 10)
    {
        break;
    }

    System.out.println( i);

    i++;
}
```

其中，循环条件总是为 **true**，所以它成了一个无限循环。为了控制循环的退出，在循环体检查当 **if(i>10)** 时，执行 **break** 语句来控制循环的退出。

到此为止，介绍了 Java 的两种最常用的循环语句，分别为 **for** 语句和 **while** 语句。

```
for ( E1; E2; E3)
{
    S1
}

while( E1 )
{
    S1
}
```

除此之外，还有一种 **do { S1 } while(E1)** 的形式的循环逻辑。这种形式不太常用，初学者可以不必了解。

所有的循环的本质原理都是相通的，写法上也是可以互相转换的。也就是说，

一个循环可以用 `for` 实现，也必然可以用 `while` 来实现。使用本章所介绍的 `for` 语句和 `while` 语句，便足够实现所有循环类的控制逻辑。

第 5 章 数组对象

本章学习目标

- 了解数组的概念
- 学会数组的定义
- 学会数组元素的访问
- 了解对象和引用的概念

数组（Array），是指一组相同类型的变量。本章先以一个例子来引入数组的概念，再详细展开介绍数组的定义、数组元素的访问等技术。

5.1 数组

为了方便大家的理解，先给出一个引例。

假设一个班有 30 个学生，为了表示某次考试的成绩，可能需要定义 30 个变量。

例如，

```
int a0 = 98;  
int a1 = 89;  
int a2 = 92;  
... 写 30 行 ...  
int a29 = 94;
```

其中，从 a0 到 a29 共 30 个变量，分别表示这 30 个学生的成绩。显然，这样的定义有点不科学：如果一个学校有一万个学生，难道要定义一万个变量吗？

5.1.1 数组的定义

在 Java 语言里，数组（Array）可以表示一组数字。

例如，

```
int[ ] arr = new int [ 30 ] ;
```

其中，创建了一个数组，长度为 30，存储的数据类型为 int。右侧的关键字 new 表示创建的意思。简单的说，就是创建了一个可以容纳 30 个 int 的数组对象。

数组的定义在形式上有以下几点：

- 对象名称： arr
- 元素类型： int[] （即数组对象）
- 数据长度： 30，即容纳 30 个数

在访问数组元素时，

arr[0] 表示第一个元素

arr[1] 有示第二个元素

...

arr[29] 表示最后一个元素

其中，第 i 个元素用 arr[i] 表示，i 称为 索引 (index) 或 下标 (subscript) 。

注意下标是从 0 开始的，例如 0, 1, 2, ..., 29。

使用下标读取某个元素的值，例如，

```
int s = arr[0] + arr[1] ; // 前 2 个元素的值
```

也可以修改某个元素的值，例如，

```
arr[7] = 99; // 第八个元素的值修改为 99
```

5.1.2 数组的遍历

所谓遍历，是指从头到尾、挨个访问每一个元素。遍历是 Java 编程里的常见操作。

例如，已知一个数组，要求打印出数组中所有元素的数值。示例代码如下。

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

arr[0] = 12;

arr[1] = 98;

arr[2] = 82;

arr[3] = 29;

for (int i = 0; i < 4; i++)
{
    System.out.print(arr[i] + " ");
}
```

使用 for 循环，从头到尾访问每一个元素，这就叫做遍历。

当然，也可以从尾到头反方向遍历，例如，

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

arr[0] = 12;

arr[1] = 98;

arr[2] = 82;

arr[3] = 29;

for (int i = 3; i >= 0 ; i--)
{
    System.out.print(arr[i] + " ");
}
```

再给出一个例子，已知一个数组，求数组中每个元素的和。

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

arr[0] = 12;

arr[1] = 98;

arr[2] = 82;

arr[3] = 29;

int result = 0; //
```

```
for( int i=0; i<4; i++)  
{  
    result += arr[i];  
}  
  
System.out.println("结果为: " + result);
```

5.1.3 数组的初始化

默认地，当用 `new` 来创建一个数组时，所有元素的值为 0。

例如，

```
int[] a1 = new int[4];  
  
double[] a2 = new double[4];
```

则 `a1` 和 `a2` 中所有元素的值都是 0。

另一种情况，在创建数组的时候可以初始化，例如，

```
int[] arr = { 98, 89, 98, 87 } ;
```

则创建了一个长度为 4 的数组，并同时指定了每个元素的值。

5.1.4 数组的长度

已知数组对象，

```
int[] arr = { 98, 89, 98, 87 } ;
```

则数组的长度可以用 `arr.length` 来表示。在遍历时，可以用 `arr.length` 来表示数组的长度。形如，

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

初学者应该强记住这种写法，其具体原理在后面的学习过程中会自然明白。

5.1.5 数组的打印输出

在打印一个数组中的所有元素时，不少初学者会犯以下的错误。例如，

```
int[] arr = { 98, 89, 98, 87 } ;  
  
System.out.println("数组的值: " + arr); // 错误!
```

数组对象是不能直接用于打印输出的。应该改成用 `for` 语句实现。示例代码如下。

```
for (int i = 0; i < 4; i++)  
{  
    System.out.print(arr[i] + " ");  
}
```

5.2 数组的使用

初学者在开始使用数组时，有可能会遇到一个错误：数组越界错误，即 `ArrayIndexOutOfBoundsException`。例如，

```
int[] arr = { 98, 89, 98, 87 };  
  
arr[4] = 128; // 这行有错
```

在运行这段代码时，Eclipse 会提示数组越界错误，如图 5.1 所示。

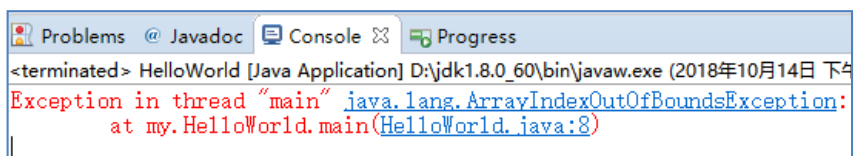


图 5.1 数组越界错误

其原因是，长度为 `N` 的数组，其下标范围应该是 `0,1,..., N-1`。如果在代码中下标超出了这个范围，在运行时就会出错，报告数组下标越界的错误。所以，当 Eclipse 提示 `ArrayIndexOutOfBoundsException` 这个错误时，应该明白它是什么意思。

在上段代码中，数组 `arr` 的长为为 4，所以有效的下标是从 `arr[0]` 到 `arr[3]`。所以

arr[4]是错误的写法。

再看一个在遍历时会常见的错误，示例如下。

```
int[] arr = { 98, 89, 98, 87 };

for ( int i=0; i <= arr.length; i++) // 错误! 是 < 不是 <=
{
    System.out.println(arr[i]);
}
```

在遍历时，如果不小心把 `i<arr.length` 写成了 `i<= arr.length`，那么就会报告数组下标越界的错误。

5.2.1 数组的应用举例

看一个例子。已知以下有四个学生的信息，要求用数组来表示。

姓名	分数
邵	97
王	89
张	94
李	93

分析一下：这里有两组信息，一组为姓名，一组为考试成绩。所以可以定义两个数组，分别表示姓名和分数。示例代码如下。

```
String[] names = {"邵", "王", "张", "李" };

int[] scores = { 97, 89, 94, 93 };

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

5.3 对象与引用

下面介绍一个比较重要的概念：对象与引用。

先看一个例子，

```
① int[] a = { 11, 11, 11, 11 };  
② int[] b = a;  
③ b[3] = 45;  
④ for(int i=0; i<a.length; i++)  
{  
⑤     System.out.print( a[i] + " ");  
}
```

其中，

第①行，创建了一个数组对象 a，

第②行，又定义了另一个 b 对象，

第③行，修改了 b[3]。这里修改的是 b，那么 a 的值会变化吗？

第⑤行，实际输出的是 11 11 11 45，说明对 b 的修改会同时影响 a 的内容。为什么呢？

5.3.1 对象与引用的概念

在 Java 语言里，对象（Object）是一个基础概念。例如，

```
int[] a = new int[4];
```

在此式中，等号右侧创建了一个数组对象，等号左侧的变量 a 称为该对象的引用（Reference）。

一般来说，可以以称作“变量 a 指向了一个对象”，或者简称为“a 是一个对象，其中 a 是对象的名字”。

多个变量可以指向同一个对象，例如，

```
int[] a = new int[4];  
  
int[] b = a; // a,b 指向同一个对象  
  
b[3] = 45;
```

由于 a,b 指向的是一个对象，所以修改 b[3]，就等同于修改了 a[3]。如图 5.2 所示。

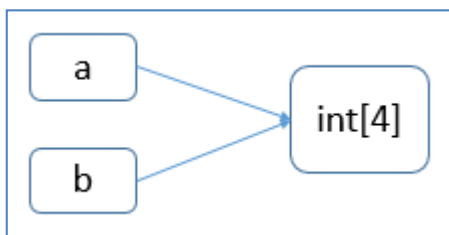


图 5.2 两个引用指向同一个对象

这就好比，“邵发”指的是一个人，“阿发你好”指的是同一个人。一个对象是可以有多个名字的。

5.3.2 空对象 null

在 Java 语言里，还有一种特殊形式的对象，称为“空对象”。例如，

```
int[] a = null;
```

其中，null 表示空对象。当 a 指向一个空对象，其实就是 a 不指向任何对象的意思。

理解以下几行代码：

```
① int[] a = new int[4];  
② int[] b = a;  
③ a = null;
```

其中，

第①行，创建了一个对象，命名为 a

第②行，b 和 a 指向同一个对象

第③行，a 指向 null。此时，a 不指向任何对象，而 b 指向刚才创建的对象。

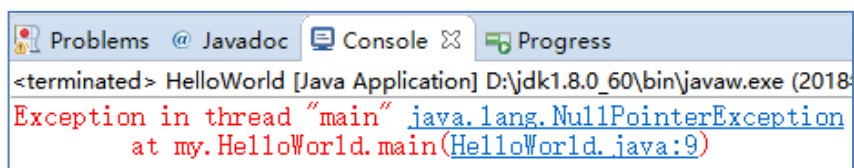
5.3.3 空指针错误 NullPointerException

在编程时，可能经常会遇到一个错误：NullPointerException，即空指针错误。

例如，

```
int[] a = { 11, 11, 11, 11 };  
a = null;  
a [0] = 12; // 出错!!
```

运行这段代码时，将提示出错，如图 5.3 所示。



这是因为，当运行到第 2 行时，a 即成为空对象。所以第 3 行对 a[0] 的访问就是不合逻辑的，因为此时 a 已经不指向任何对象。

5.3.4 失去引用的对象

观察以下代码：

```
① int[] a = { 8, 8, 8 };  
② a = new int[4];  
③ a[0] = a[1] = a[2] = a[3] = 17;
```

其中，

第①行，创建了一个数组对象，内容为{ 8,8,8 }

第②行，创建另一个数组对象，内容{ 0,0,0,0 }

问题是，当执行完第②后，曾经创建的第一个对象就没有任何名字引用它了，

这称为“失去引用”的对象。

当一个对象失去引用后，就不会再被使用，会由系统自动地回收和销毁这个对象。此过程为称垃圾回收（Garbage Collection, GC）。

所以，在 Java 里能看到对象的创建（new 就是创建对象），却看不到对象的销毁，就是因为对象在失去引用之后会被系统自动地销毁。

最后看一个例子体会一下，

```
int [] a = new int[4];  
  
a = new int[5];  
  
a = new int[5];
```

在这段代码里，一共创建了 3 个对象，但前两个对象都失去了引用，会被系统自动回收。关于对象和引用，是一个比较难的话题，初学者不必急于掌握。先试着理解一下，有个印象即可，在随后的课程中还会不断地加强这个印象。

第 6 章 类

本章学习目标

- 了解类的概念
- 学会类的基本定义
- 学会在类里添加属性

类 (Class)，是面向对象设计里的基本概念。本章引入类的概念，介绍类的简单定义，并学会向类里添加属性。在本章的最后，将再次强调对象与引用的概念。

6.1 新建类

先给出一个引例。以下是两组学生的信息，要求用代码表示，如表 6.1 所示。

6.1 两组学生信息的数据

学号	姓名	性别	手机号
20171001	王草	男	18610022345
20171002	李花	女	13820490902

可以用 Java 代码表示这些学生的信息，示例代码如下。

```
// 第一个学生的信息  
String id = "20171001";  
  
String name = "王草";  
  
boolean sex = true;  
  
String cellphone = "18610022345";
```

```
// 第二个学生的信息

String id_2 = "20171002";

String name_2 = "李花";

boolean sex_2 = false;

String cellphone_2 = "13820490902";
```

显然，如果按照这种方式来定义，就需要定义很多的类似的变量。如果有 30 个学生，就得定义 120 个变量，name_1, name_2, name_3... 按照这样的命名方式，似乎不太科学。如果能有一个新的数据类型，专门来表示“学生”这种类型就好了！

下面就引入“类”的概念，来解决这个问题。

6.1.1 类

类（class），在 Java 里表示自定义的数据类型。一个 class 可以是若干基本类型的组合。通过新建 class，可以用来表示自定义的类型。

例如，在 Eclipse 新建一个类 Student.java，其内容如下。

```
package my;

public class Student
{
    public String id;    // 学号
    public String name;  // 姓名
    public boolean sex;  // true: 男 false 女
    public String cellphone; // 手机号
}
```

其中，

- Student 称为类的名称，一般以大写字母开头；
- id, name, sex, cellphone 则称为类的属性，和变量的命名规则相同。

直观的理解就是，Student 表示学生类型，具有 id, name, sex, cellphone 四个属

性，属性可以是 `int`, `double`, `String`, `boolean` 等基本类型。

其中，关键字 `public` 的意思以后再说。初学者先抄下来，不会影响理解。

6.1.2 创建对象

在定义了类型之后，就可以创建该类型的对象。例如，

```
Student s1 = new Student();

s1.id = "20171001";

s1.name = "王草";

s1.sex = true;

s1.cellphone = "18610022345";
```

其中，

- `new Student()` 表示创建一个 `Student` 类型的对象；
- `s1.id` 表示对象 `s1` 的 `id` 属性，中间是一个点号。

以上代码，是创建一个 `s1` 对象，并设置其四个属性的值。其中 `s1.name`，可以读作“对象 `s1` 的 `name` 属性”。

6.1.3 类与对象

什么是类，什么是对象，现在要有一个初步的认识。其实并不复杂，类就是类型的意思，描述了一类事物的特性。而对象则是实例，表示具体的一个东西。

比方说，“人类”是类，描述了一类事物。“小王”是对象，是一个具体的人。

用代码表示为，

```
Human someone = new Human();

someone.name = "小王";
```

其中，`Human` 可以是一个自定义的 `class` 类型（类似于 `Student` 类），`someone` 则是一个具体的对象。

6.1.4 常见错误

以下代码是初学者常犯的一个错误，例如，

```
Student.name = "小王"; // 错误!
```

这在逻辑上是说不通的，因为 **Student** 只是一个类型描述，你不能说“学生”的名字是小王，只能说某个学生的名字是小王。

正确的写法是，

```
Student stu = new Student(); // 先创建对象 stu
stu.name = "小王";           // 再访问对象的属性
```

先创建对象，然后再访问该对象的属性，这是大家要熟练掌握的写法。多写几遍，熟悉一下吧！

6.2 类的属性

类的属性 (Property)，描述这个类里有什么。比如，对于一个 **Student**，它应该有学号 (id)、姓名 (name) 等属性。示例代码如下。

```
public class Student
{
    public String id;
    public String name;
    public boolean sex;
    public String cellphone;
}
```

其中，在类 **Student** 中添加了 4 个属性。

- 属性 **id**，类型为 **String**
- 属性 **name**，类型为 **String**
- 属性 **sex**，类型为 **boolean**
- 属性 **cellphone**，类型为 **String**

实际上，类的属性可以是任何类型。可以是前面所学过的 `int`, `double`, `String`, `boolean` 这些基本类型，也可以是其他还没有学到的类型。

6.2.1 类的书写步骤

下面练习一下类的定义。要求定义一个类，描述图书的信息。

每一本书都有以下信息。

- 书名 (`title`)，`String` 类型
- 作者 (`author`)，`String` 类型
- 出版社 (`press`)，`String` 类型
- 书号 (`ISBN`)，`String` 类型
- 定价 (`price`)，`double` 类型

初学者在定义 `class` 时可能会感觉无从下手，其实这个并不困难，按以下的步骤即可：

① 先给类起一个名字

类名一般应以大写字母开头。要起有意义的名字，"`aaa`", "`abcd123`" 这样的毫无意义的名字是不妥的。名字应该能顾名思义，比如 `Movie` 可以表示电影，而 `ActionMovie` 可以表示动作电影。这样的名字可读性好，让人一眼就能知道它是什么意思。

② 再给类添加属性

属性名字一般以小写开头，如 `fontSize`, `lineHeight`。

属性的类型需要仔细考虑，可以选择 `int`, `double`, `String`, `boolean` 等基本类型，将来还可以选择其他类型。

下面新建一个类叫 **Book**，内容如下，

```
public class Book
{
    public String title;
    public String author;
    public String press;
    public String ISBN;
    public double price;
}
```

在使用时要先创建一个对象，然后再访问这个对象的属性。示例代码如下，

```
Book b = new Book(); // 创建对象
b.title = "C/C++学习指南"; // 设置 b.title 的值
b.price = 49.0; // 把 b 的 price 设置为 49.0
// 其他属性也可以设置一下 ...
```

6.2.2 编程世界里的类

在初学阶段，通常以自然界实际存在的事物来做例子进行讲解。但是需要强调的是，以后将要见到的在编程世界里的类，大多数不对应着自然界的物体。

比如，下面定义一个类，它能够把执行压缩数据的功能。形如以下代码，

```
public class ZipTool
{
}
}
```

显然，对于 **ZipTool** 这样的类，它是不对应自然界中的物体的。

6.2.3 类的嵌套书写

实际上，一个类的属性可以是基本类型(int, double, String, boolean...), 也可以其他任意类型。

比如，每个学生除了有学号、姓名、手机号等基本信息之外，还绑定了一个校园卡、一个银行账号。

首先，定义一个类 `StudentCards` 表示卡号信息，

```
public class StudentCards
{
    public String schoolCardNumber;
    public String bankCardNumber;
}
```

然后，在 `Student` 类中加入 `cards` 属性，

```
public class Student
{
    public String id;
    public String name;
    public boolean sex;
    public String cellphone;
    public StudentCards cards = new StudentCards();
}
```

最后，创建对象并设置属性的值，

```
Student s1 = new Student();
s1.cards.schoolCardNumber = "T92830430";
s1.cards.bankCardNumber = "34989989289988290";
```

其中，`s1.cards.bankCardNumber` 就表示访问 `s1` 的 `cards` 的 `bankCardNumber` 属性。

6.3 再说对象与引用

在上一章里已经初步介绍了对象和引用的概念，这里再加深一下印象。

观察以下代码，

```
① Student s1 = new Student();  
② Student s2 = s1;  
③ s2.name = "邵发";
```

第①行，右侧创建了一个对象，然后用 `s1` 指向了这个对象。这就好比，右边生了一个宝宝，左侧 `s1` 是他/她的名字。

第②行，则是相当于给这个宝宝再起一个名字，一个乳名一个学名。这两个名字指向的是同一个宝宝。

第③行，由于 `s1` 和 `s2` 指向的是同一个对象，所以修改 `s2.name` 就是修改了 `s1.name`。

6.3.1 空对象与空指针异常

当一个引用指向 `null`，其实是说它不指向任何对象。例如，

```
Student s1 = null;  
  
s1.name = "邵发" ;    // 出错!! NullPointerException
```

其中，`s1` 指向了 `null`，就是说它不指向任何对象。所以试图访问 `s1.name` 是错误的。它本身就是个空，当然就不能访问它的属性啦！

6.4 属性的默认值

在 Java 语言里，每一种类型都有其默认值。

- 整数类型 (`long int short byte`): 默认值为 0
- 小数类型 (`double float`): 默认值为 0.0
- 布尔类型 (`boolean`): 默认值为 `false`
- 引用类型 (`String, Student..`): 默认值为 `null`
- 数组类型: 默认为 `null`

本质上，所有的类型默认值都是零的意思。比如说，`false` 和 `null` 本质上就是个 0。

如果不能确定其默认值，那么可以在定义一个变量或属性的时候，显式地指定属性的初始值。例如，

```
public class Student
{
    public String id = "000000000";
    public String name = "未命名";
    public boolean sex = true;
    public String cellphone = "";
}
```

如果属性是 `class` 类型，那么可以 `new` 一个对象，例如，

```
public class Student
{
    public String id = "000000000";
    public StudentCards cards = new StudentCards ();
}
```

也可以先创建对象，然后再设置它的属性。例如，

```
Student stu = new Student();
stu.cards = new StudentCards ();
stu.cards.bankCardNumber = "239230923901234";
```

各种写法，其实殊途同归。不管怎么写，都是要保证在访问 `stu.cards.bankCardNumber` 的时候，`stu.cards` 不能为 `null`，否则就会是空指针错误了！

第 7 章 类的方法

本章学习目标

- 学会在类里添加方法
- 学会定义方法的参数
- 学会定义方法的返回值
- 了解方法名的重载

方法 (Method)，表示一个类能做什么。本章介绍如何在类里添加一个方法，以及如何调用这个方法。然后进一步介绍方法的参数和返回值、方法名重载等语法。

7.1 方法

在 Java 里，方法和属性是对等的术语。在一个类里，不仅可以添加属性，还可以添加方法。可以大致地表示如下：

```
类
{
    属性：描述“我有什么”
    方法：描述“我能做什么”
}
```

7.1.1 添加方法

先看一个例子，在 Student 类里添加一个名字叫 show 的方法，

```
public class Student
{
    public void show ( )
```

```

{
    for(int i= 0; i<10; i++)
    {
        System.out.println("报数: " + (i+1));
    }
}

```

其中，`public void show () { ... }` 这一段就称为方法，而 `show` 是方法名，大括号里的代码称为方法体。如图 7.1 的所示。

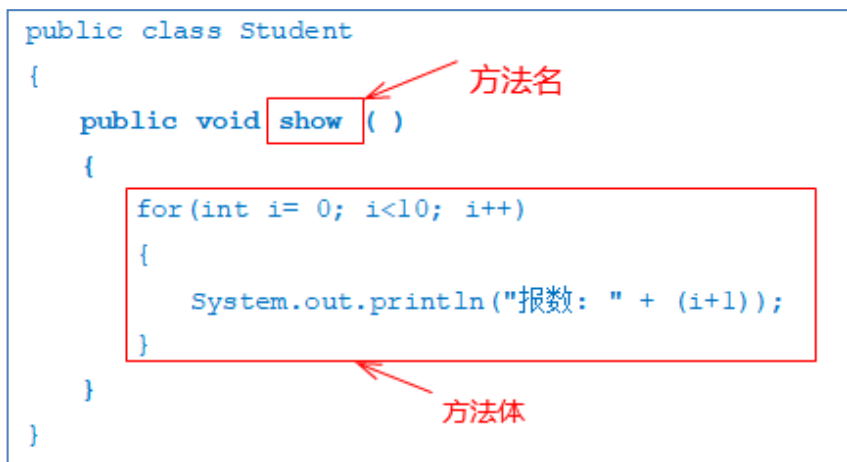


图 7.1 方法名与方法体

初学者只需要先照抄几遍，记住它的形式即可。至于各部分的细节，比如 `public`、`void` 是什么意思，在后面都会有详细讲解的。

7.1.2 方法的调用

现在，已经在 `Student` 类里添加一个 `show()` 方法。那么，怎么使用这个方法呢？

按照面向对象的一般原则，无论是访问属性、还是方法，都要先创建一个对象。

例如，

```
Student s = new Student();
```

```
s.show(); // 调用对象 s 的 show() 方法
```

其中，先创建一个对象 `s`，然后再用 `s.show()` 来调用它的方法。当调用 `s.show()` 时，其方法体里的代码会被执行。

7.1.3 方法的命名

方法的命名规则和属性一样，都是小写字母开头，形如，

```
startService ()
```

```
check ()
```

```
openSafely ()
```

为了保证代码的可读性，一般属性以名词性短语命名，而方法用动词性短语命名（表示一个动作）。

7.2 方法的参数

先来看一下方法的一般形式：

```
修饰符    返回值类型    方法名 ( 参数列表 )
```

```
{
```

```
    方法体
```

```
}
```

其中每一部分的含义，比如什么是参数列表，什么是返回值，接下来会对每一部分详细讲解。

7.2.1 示例 1

先看一个例子。在 `Student` 类里添加一个方法，仍是实现报数的功能，但是要求最大值 `N` 由调用者来指定。示例代码如下。

```
public class Student
```

```

{
    public void show2 ( int maxNumber )
    {
        for(int i= 1; i<maxNumber ; i++)
        {
            System.out.println("报数: " + (i+1));
        }
    }
}

```

其中，`show2()` 方法带了一个参数 `int maxNumber`。可以理解为，此方法用于实现报数的功能，但报数的范围由调用者指定。

下面看一下如何调用这个 `show2` 方法，

```

Student s = new Student();

s.show2(40); // 传入参数值

```

其中，在调用 `s.show2()` 方法时传入了参数的值为 40。当方法带有参数时，在调用的时候要把参数的值传入。

7.2.2 示例 2

进一步修改上面的功能，要求在报数的时候，报数的起点和终点都由调用者决定。

在 `Student` 类里再添加一个方法 `show3`，示例代码如下。

```

public class Student
{
    // 指定报数的上限

    public void show2 ( int maxNumber )
    {

```

```

        for(int i= 1; i<maxNumber ; i++)
        {
            System.out.println("报数: " + (i+1));
        }
    }

    // 指定报数的起点和终点
    public void show3 ( int from, int to )
    {
        for(int i= from; i<= to ; i++)
        {
            System.out.println("报数: " + (i));
        }
    }
}

```

在这段代码里，方法 `show3` 带了 2 个参数，以逗号分隔。

- `int from` ， 表示起点
- `int to` ， 表示终点

下面再来看一下如何调用 `show3` 方法，

```

Student s = new Student();

s.show3( 30, 60); // 从 30 升级到 60

```

也就是说，`show3` 需要传入 2 个参数。

小结一下，参数列表可以由 0..N 个参数组成，以逗号分开。每个参数应指定参数类型和参数名。

7.2.3 理解参数的作用

方法就是描述一个类能够做什么事情，而方法的参数则就是日常生活的“参数”的意思。

比如说，空调可以用 `AirConditioner` 类来表示，空调的致冷功能可以用 `freeze()` 方法来表示。如果 `freeze()` 方法不带参数，在逻辑上就匪夷所思：你买了一台空调，难道连致冷的温度都不能设置吗？形如以下的代码。

```
public class AirConditioner
{
    public void freeze ( )
    {
        // 我能致冷，但我该致冷到多少度？
    }
}
```

显然，空调的致冷温度是由用户来设定的，所以 `freeze()` 方法需要带一个参数来表示目标温度。示例代码如下。

```
public class AirConditioner
{
    public void freeze ( int degree )
    {
        // 启动致冷操作、直到达到目标温度 degree 时
    }
}
```

现在，这个空调就可以正常使用了。示例代码如下。

```
AirConditioner ac = new AirConditioner();

ac.freeze ( 18 ) ; // 致冷到 18 度
```

相信到此为止，大家已经明白了“参数”的作用。

7.3 方法的返回值(1)

再来回顾一下方法的一般形式：

修饰符 返回值类型 方法名 (参数列表)

```
{  
  
    方法体  
  
}
```

目前，方法名、方法体、参数列表的意义我们已经明确，这节课再讲一下“返回值类型”的用法。简单的讲，“方法”就是做一件事情，而“方法的返回值”就是这件事情的结果。

7.3.1 示例 1

已知一个整型的数组，要求出里面的最大值。

```
int[] a1 = { 29, 93, 193 };
```

这个功能对于大家来说，应该没有什么难度。写一个方法，示例代码如下。

```
public int getMax2 ( int[] data)  
{  
    int result = data[0];  
    for ( int i= 0; i<data.length; i++)  
    {  
        if ( data[i] > result)  
        {  
            result = data[i];  
        }  
    }  
    return result;  
}
```

在这一段代码中，要注意两个部分，如图 7.2 所示。

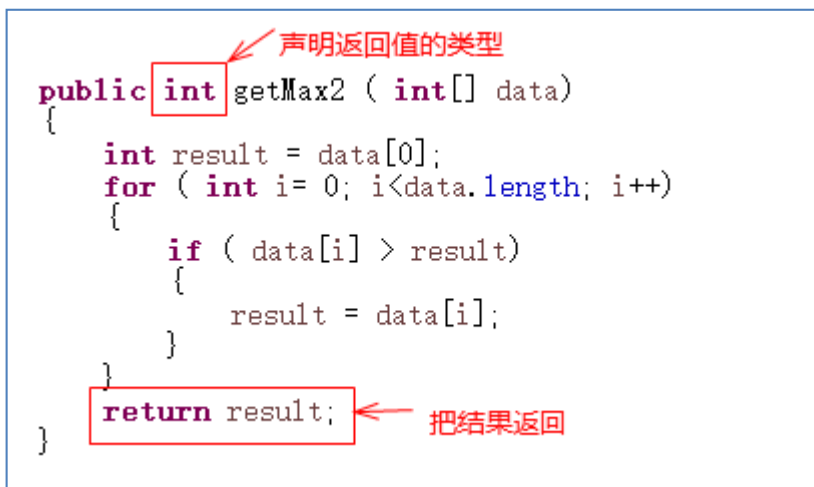


图 7.2 方法的返回值

- 将返回值的类型声明为 `int`，表示这个方法将返回一个整数
- 计算出结果 `result`，然后用 `return` 语句返回

在调用的时候，用变量 `max` 来接收 `m.getMax2()` 的返回值。示例如下，

```
int[] a1 = { 29, 93, 193 };

MyMath m = new MyMath();

int max = m.getMax2 ( a1 );
```

打个比方。让小明去打买冰棍 `xiaoming.buy (money)`，参数就是给了他多少钱 (`money`)，返回值就是他买回来的雪糕。如果给他 1 元钱(`money=1`)，就返回 1 支冰棍；如果给他 5 元钱 (`money=5`)，就返回 5 支冰棍。

7.3.2 示例 2

下面再讲一个例子。现在有两个数组，要求这两个数组中的最大值。

```
① MyMath m = new MyMath();
② int r1 = m.getMax2( a1 );
③ int r2 = m.getMax2( a2 );
④ int max = r1;
⑤ if( r2 > max ) max = r2;
```

```
⑥ System.out.println("结果为: " + max);
```

其中，

第②行，计算数组 a1 里的最大值 r1

第③行，计算数组 a2 里的最大值 r2

第④⑤行，取 r1 和 r2 的最大值

提示：如果 if 语句里只有一行，则可以省略大括号。

```
if ( r2 > max )  
{  
    max = r2;  
}
```

可以简写为，

```
if ( r2 > max )  
    max = r2;
```

不过还是建议初学者，在使用 if 语句时统一使用大括号的形式。

7.3.3 void 返回值

void 的意思是“否、不用”。如果一个方法不需要返回值，则将返回值类型设为 void。打个比方，让小明去看个电影，是不需要返回值的，

```
public void watchMovie( int money)  
{  
}  
}
```

但是，如果让小明去买本书，那么就需要把买的书返回来，

```
public Book buy( int money)  
{  
}  
}
```

所以，所谓的返回值就可以理解为这件事情返回的结果。

7.4 方法的返回值(2)

下面来讨论一下 `return` 语句的用法。`return` 语句必须放在方法体里，起到两个作用：

- 返回一个值；
- 从方法中退出。当 `return` 语句运行时，立即从方法体中退出。

7.4.1 示例 1

给定一个数组，判断它们的和是否大于 100。假设所有元素都是正数。

```
int[] a1 = { 28, 23, 62, 16, 8 };
```

首先，创建一个类 `MyMath` 并添加一个方法 `check()`，示例如下。

```
public class MyMath
{
    public boolean check ( int[] arr )
    {
        // ...
    }
}
```

其中，方法的参数是一个数组 `int[]`。方法的功能是判断各元素的和是否大于 100，所以规定返回值的类型为 `boolean`。如果大于 100 则返回 `true`，否则返回 `false`。

下面把 `check` 的功能实一下，代码如下所示。

```
public boolean check ( int[] arr )
{
    boolean result = false;

    int sum = 0;

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

```

        sum += arr[i];

        if( sum > 100)
        {
            result = true;
            break;
        }
    }

    return result; // 返回结果
}

```

其中，是把结果计算出来，记为 **result**，最后一行使用 **return** 语句返回。

但实际上，**return** 语句不是一定要放在最后的。在得到结果的第一时间，就可以调用 **return** 语句退出方法的运行。可以把代码优化如下，

```

public boolean check ( int[] arr )
{
    int sum = 0;
    for (int i=0; i<arr.length; i++)
    {
        sum += arr[i];

        if( sum > 100)
        {
            return true; // 结果既出，直接返回
        }
    }

    return false;
}

```

注意，在 **return** 后面不需要再加 **break** 语句，因为 **return** 本身有退出方法的功能。当执行 **return** 后，循环自然就被中断，然后从方法中退出。

7.4.2 示例 2

再看一个例子。给定一个 `int` 值，要求打印出一个三角型的形状。形如，

```
1
2 3
4 5 6
7 8 9 10
```

先把方法写出来，代码如下，

```
public void print ( int n )
{
    int rows = 0; // 行数
    int cols = 0; // 列数
    for(int k=1; k<=n; k++)
    {
        System.out.print( k + " ");

        cols ++;

        if( cols > rows )
        {
            System.out.print("\n"); // 换行

            rows ++;

            cols = 0;
        }
    }
}
```

现在考虑：当 `n <= 0` 时怎么办？如果 `n <= 0`，方法是可以提前退出的，再往走就没有意义。所以可以优化一下，写成以下这种形式。

```
public void print ( int n )
{
```

```

if (n<=0)
{
    System.out.println("给定的数<=0!!");
    return; // 退出方法
}

if (n>10)
{
    System.out.println("给定的数太大了!");
    return; // 退出方法
}

int rows = 0; // 行数
int cols = 0; // 列数
for (int k=1; k<=n; k++)
{
    .... 篇幅限制, 省略一些代码 .....
}
}

```

其中, 当执行 **return** 语句后, 方法直接退出, 后面的语句不会继续执行。

同时也注意到, 这个 **print** 方法的返回值类型是 **void**, 所以 **return** 语句里不能加返回值。就是单独的一行 **return** 表示退出方法。截取相关代码如下,

```

if (n<=0)
{
    System.out.println("给定的数<=0!!");
    return; // 注意, 这里的 return 不加返回值
}

```

7.5 方法的返回值 (3)

在使用 **return** 语句来返回一个值时, 可以返回多种类型的值, 包括:

- 基本数据类型: int, double, String, boolean
- 自定义类型: 如 Student, Book
- 数组类型: 如 int[], Student[]

也就是说, 各种类型都是可以返回的。可以返回一个苹果, 也可以返回一个大象, 没有禁忌。随着教程的深入, 后面还会看到返回各种类型的例子。

7.5.1 示例 1

给定一个数组, 求里面的能被 8 整除的数。

```
int[] arr = { 18, 28, 32, 36, 48 };
```

自然地, 还是添加一个类, 然后在类里添加一个方法。代码实现如下,

```
public class MyMath
{
    // 把符合要求的数放在返回值里
    public int[] find8(int[] arr)
    {
        // 创建等大的数组
        int[] temp = new int[arr.length];
        int count = 0;
        for (int i = 0; i < arr.length; i++)
        {
            if (arr[i] % 8 == 0)
            {
                temp[count] = arr[i];
                count++;
            }
        }
        // 拷贝到结果数组里
```

```

        int[] result = new int[count];
        for (int i = 0; i < count; i++)
        {
            result[i] = temp[i];
        }

        return result;
    }
}

```

然后在 `main()` 方法里调用这个方法，

```

public static void main(String[] args)
{
    int[] arr = { 18, 28, 32, 36, 48 };
    MyMath m = new MyMath();
    int[] result = m.find8( arr );
}

```

提示：数组不能直接用 `println` 输出，如果想观察数组 `result` 里的值，可以用 `for` 循环逐个输出，或者用单步调试的方式查看。

7.5.2 示例 2

已知学号，姓名，手机号，创建一个学生对象。

由于没有实际项目的上下文，这个例子对于大家来说可能有点生硬。大家只需要了解这个方法的返回值的形式即可。代码如下，

```

public Student createNew (String id, String name)
{
    Student temp = new Student();
    temp.id = id;
    temp.name = name;
}

```



```
        return temp; // 可以返回一个自定义类型的对象
    }
```

再简单看一下如何调用，

```
int[] arr = { 18, 28, 32, 36, 48 };

MyMath m = new MyMath();

Student stu = m.createNew("2329", "shaofa");
```

通过这个例子，说明返回值的类型也可以是引用类型。

7.6 方法名的重载

重载（OverLoad），是指同名的方法。在 Java 里多个方法可以有相同的名称、不同的参数列表，这种现象称为“方法名重载”。例如，

```
public class Simple
{
    public void test()
    {
        System.out.println("测试 1。。。");
    }

    public void test(int a, int b)
    {
        System.out.println("测试 2: a="+ a + ",b=" + b);
    }

    public void test(int a, String b)
    {
        System.out.println("测试 3: a="+ a + ",b=" + b);
    }
}
```

其中，在 Simple 类中有三个方法，名字都叫 test，但参数列表不同。

第一个 `test` 方法不带参数。第二个 `test` 方法带 2 个参数(`int,int`)，第三个方法带 2 个参数(`int, String`)。这几个方法名称相同、参数列表不同，所以叫方法名重载。

在调用的时候，根据传递的参数来匹配相应的方法，例如，

```
public static void main(String[] args)
{
    Simple s = new Simple();

    s.test();           // 无参
    s.test(10, 12);    // 参数: int, int
    s.test(10, "shaofa");// 参数: int,String
}
```

其中，`s.test()` 时没有传递参数，所以匹配的是无参的 `test()`。`s.test(10,12)` 匹配的是 `test(int, int)`，而 `s.test(10,"shaofa")` 匹配的是 `test(int, String)`。

第 8 章 当前对象

本章学习目标

- 理解什么是当前对象
- 学会 `this` 的使用方法

当前对象 (`this`)，是面向对象编程里的一个重要概念。本章介绍为什么需要当前对象，当指对象指向了哪个对象，以及关于 `this` 的更多使用。

8.1 当前对象 `this`

先给出一个引例。用 `Screen` 类表示一个屏幕，`width`, `height` 表示屏幕的宽和高。要求写一个方法 `pixel()` 来计算它的像素数。像素计算公式： $p = \text{width} * \text{height}$ 。

要实现这个例子并不困难，可以新建一个 `Screen` 类，示例如下。

```
public class Screen
{
    public int width;
    public int height;
    public int pixel ( int w, int h )
    {
        int result = w * h;
        return result;
    }
}
```

然后在 `main()` 方法里按如下的方式调用，

```
Screen s = new Screen();
s.width = 1366;
```

```
s.height = 768;

int p = s.pixel( s.width, s.height );

System.out.println("像素数: " + p );
```

这样是可以实现需求的，但显然也存在一些问题：前面已经设置了 `s.width` 和 `s.height`，后面在调用 `s.pixel()` 方法的时候却又传递了一遍。这就好比，`Screen` 类相当于一个工具，明明已经给它设好了宽高，再让它计算像素的时候，却还是要传递一遍。显然，这样的操作是有一点麻烦的。

下面，对上述代码做一点小改进，示例代码如下。

```
public class Screen
{
    public int width;
    public int height;

    public int pixel ( Screen that ) // 此处改进
    {
        int result = that.width * that.height;

        return result;
    }
}
```

再按如下方式调用，

```
Screen s = new Screen();

s.width = 1366;

s.height = 768;

int p = s.pixel( s ); // 此处改进

System.out.println("像素数: " + p );
```

经此改进之后，在调用 `s.pixel()` 时，只需要对 `s` 对象传递过去就行了。但是像 `s.pixel(s)` 的调用仍然显得有点奇怪：为什么调用对象 `s` 的方法要传递它自己作为参数呢？

8.1.1 this 参数

在 Java 的方法里，默认传递了一个 this 参数，该参数指向对象自身。如图 8.1 所示。

```
public class Screen
{
    public int width;
    public int height;

    public int pixel ()
    {
        int result = this.width * this.height;
        return result;
    }
}
```

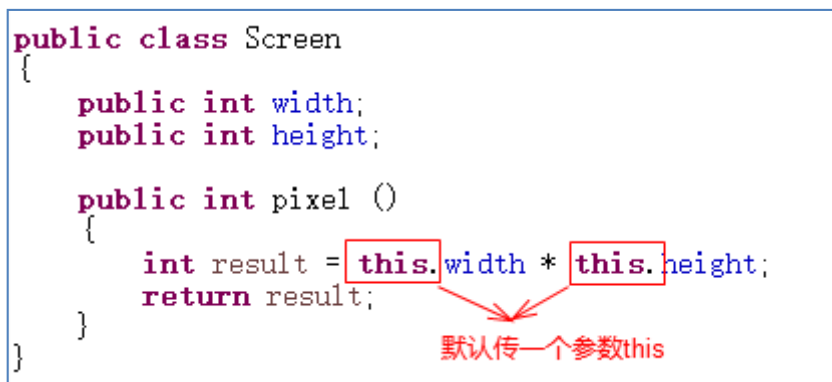


图 8.1 默认的 this 参数

在这里，pixel() 看起来并未带任何参数，但实际上有一个隐含的参数 this，指向了当前对象。那么，什么是当前对象呢？如图 8.2 所示。

```
public static void main(String[] args)
{
    Screen s = new Screen();
    s.width = 1366;
    s.height = 768;

    int p = s.pixel();

    System.out.println("像素数: " + p );
}
```

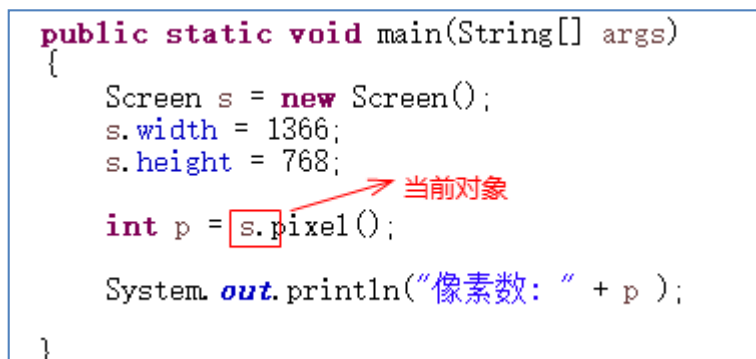


图 8.2 当前对象是哪个对象

当在外面调用 s.pixel() 时，s 即为当前对象。在进入 pixel() 方法，this 所指向的对象，就是当前对象。

8.1.2 调用自己的方法

使用 `this` 不仅可以访问当前对象的属性，也可以调用当前对象的方法。

下面给出一个例子。要求输出 `m,n` 之间所有的质数，例如 400~500 之间所有的质数。所谓质数，是指只能被 1 和自身整除的，如 2, 3, 5, 7, 11。示例代码如下。

```
public class MyMath
{
    // 判断n是否为质数； true,是质数； false, 不是质数
    public boolean isPrime( int n )
    {
        for(int i=2; i<n ; i++)
        {
            if( n % i == 0)
            {
                return false;
            }
        }
        return true;
    }

    // 输出 m,n 之间所有的质数
    public void showPrimes (int m, int n)
    {
        for(int i=m; i<=n; i++)
        {
            if( this.isPrime( i ) ) //
            {
                System.out.println("质数: " + i );
            }
        }
    }
}
```

```
    }  
    }  
}
```

其中, `isPrime(n)` 方法用于判断一个整数 `n` 是否为质数, 如果是质数则返回 `true`; 否则返回 `false`。

另一个方法 `showPrimes(m, n)` 则用于打印在 `m~n` 之间的所有的质数。在 `showPrimes()` 方法里, 间接调用了 `this.isPrime()` 方法来判断是否为质数。这个例子可以说明, 通过 `this` 可以访问当前对象的方法。

8.2 省略与重名

8.2.1 省略 this

在用 `this` 访问当前对象时, `this` 一般是可以省略的。例如,

```
public class Example  
{  
    public int number = 10;  
    public void showNumber()  
    {  
        System.out.println("当前值: " + this.number);  
    }  
}
```

在 `showNumber()` 方法里访问当前对象的属性, 可以写成 `this.number`。但是为了简化书写, 此处的 `this` 可以省略。如图 8.3 所示。

```

public class Example
{
    public int number = 10;

    public void showNumber()
    {
        System.out.println("当前值: " + number);
    }
}

```

省略this

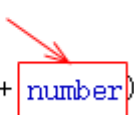


图 8.3 this 可以省略

不过对于初学者来说，应坚持使用 `this`，一直到非常熟练之后再慢慢的把 `this` 省略不写。坚持使用 `this`，有助于加强对当前对象的理解。

8.2.2 重名

关于重名的问题，先看一个例子。

```

① public class Example
② {
③     public int number = 10;
④     public void test()
⑤     {
⑥         int number = 12;
⑦         System.out.println("值为:" + number);
⑧     }
⑨ }

```

在这个例子中，

第 3 行，定义了一个属性 `number`，值为 10

第 6 行，定义了一个变量 `number`，值为 12

第 7 行，输出 `number` 的值。问题是，这个到底是第 3 行的 `number` 还是第 6 行的 `number`？

在 Java 里，把定义在方法中的变量，称为局部变量。所以，第 6 行定义的 `number`

就是一个局部变量。

当局部变量与类的属性名字相同（名字冲突）时，局部变量优先显示。所以，第 7 行中的 **number** 指的是局部变量。此例将输出的值为 12。

在重名的情况下，如果要访问类的属性，则必须指定 **this** 前缀，不能省略。

再举一个例子，代码如下。

```
① public class Example
② {
③     public int number = 10;
④     public void setNumber (int number)
⑤     {
⑥         this.number = number;
⑦     }
⑧ }
```

第 4 行，在参数列表里定义的变量也叫局部变量。

第 6 行，左侧的 **this.number** 指的是属性，右侧的 **number** 指的就是参数里的局部变量。

8.3 类的设计示范

下面通过几个例子，来演示如何以面向对象的思路来设计一个类。

8.3.1 示例 1

有一个换游戏币的机器。可以投 1 元、5 元、10 元的人民币。最后按一下出货按钮，可以吐出游戏币。（假定每个游戏币=1 元人民币）

首先，创建一个类 **Machine** 来表示这种机器，

```
public class Machine {
}
```

然后，这种机器可以投入现金，所以添加一个方法用于投入现金，

```
public class Machine
{
    // 人民币: 1,5,10

    public void insertCash ( int cash )
    {

    }
}
```

当把现金投进去后，应该有一个属性记录一共投了多少现金，

```
public class Machine
{
    public int money = 0; // 机器里投入了多少钱

    public void insertCash ( int cash )
    {

    }
}
```

现在来完成 insertCash() 方法，每投入一些现金后，应该把现金数加到余额上，

```
public void insertCash ( int cash )
{
    this.money += cash;

    System.out.println("当前余额: " + this.money);
}
```

现在，该完成交易了，添加 exchange() 方法，

```
public class Machine
{
    public int money = 0;

    public void insertCash ( int cash )
    {
```

```

        ... 篇幅原因, 省略此处代码 ..
    }

    public int exchange ()
    {
        int numOfCoin = this.money / 1;

        this.money = 0;

        System.out.println("交易完成, 当前余额: " +
this.money);

        return numOfCoin;
    }
}

```

最后看看怎么调用这个类,

```

Machine m = new Machine();

m.insertCash( 5 );

m.insertCash( 10 );

m.insertCash( 50 );

int coins = m.exchange(); // 按一下按钮

System.out.println("拿到了" + coins + "个游戏币");

```

这便是一个最简单的面向对象的设计。可以发现, 类的属性是用于存储数据的, 而类的方法则是用于计算数据的。从这个角度, 可以说类就是属性和方法的综合体。

8.3.2 示例 2

有两个数组, 分别为:

```
int[] a1 = { 123, 38, 103, 89 };
```

```
int[] b1 = { 34, 8, 11, 29 };
```

要求这两个数组中的所有的质数。

首先, 设计一个类 `PrimeFilter`。再添加一个 `put ()` 方法, 将需要处理的数组传

给它，内部检出所有的质数，存在属性 **result** 里。再添加一个 **values()**方法，用于取出最后的结果。示例代码如下。

```
public class PrimeFilter
{
    // 存储
    public int[] result = new int[512];
    public int total= 0;
    // 用户输入：数组 data
    // 把 data 数组里面，所有的质数都放到 result
    public void put ( int[] data)
    {
        for (int i=0; i<data.length; i++)
        {
            if ( this.isPrime( data[i] ))
            {
                this.result[total] = data[i];
                this.total += 1;
            }
        }
    }
    // 取出最终过滤得到 所有的质数
    public int[] values()
    {
        int[] r = new int[total];
        for(int i=0; i< this.total; i++)
        {
            r[i] = this.result[i];
        }
    }
}
```

```

        return r;
    }

    // 判断 n 是否为质数; true,是质数; false, 不是质数
    public boolean isPrime( int n )
    {
        for(int i=2; i<n ; i++)
        {
            if( n % i == 0)
            {
                return false;
            }
        }

        return true;
    }
}

```

再来看一下怎么调用，

```

PrimeFilter filter = new PrimeFilter();

int[]  a1 = { 123, 38, 103, 89 };

int[]  b1 = { 34, 8, 11, 29 };

filter.put (a1);

filter.put (b1);

int[] numbers = filter.values();

```

小结一下，所谓的对象可以视为属性和方法的综合体。对象 = 属性 + 方法。

其中，属性就是数据，方法就是算法。创建一个对象、给它所需的数据、让它工作、取出结果，这就是面向对象的一般设计方法。

8.4 特殊形式的属性

下面，再了解一种相对来说有点奇怪的写法。用 **Human** 表示人类，每个人类都

应该有名字(name)，有一个配偶(mate)。用代码表示如下，

```
public class Human
{
    public String name; // 名字

    public Human mate; // 配偶 (指向另外一个 Human 对象)
}
```

前面说过，类的属性可以是任何类型，包含它的自身类型。在此例中，**Human** 类有一个属性 **mate**，属性的类型不是 **Human**。对于初学者来说，这种写法可能会有点奇怪：一个属性的类型可以是它的同类？

如果觉得奇怪的话，我们可以大胆的改一下，改成我们熟悉的形式，

```
public class Human
{
    public String name;

    public Monkey mate; // Monkey, 猴子
}
```

如此改动，在字面形式上会觉得熟悉一些。但从逻辑上来说，却超出来常理，，一个人类(**Human**)的配偶(**mate**)竟然是一个猴子？

一个 **Human** 的 **mate** 属性应该是另一个 **Human** 对象，这才是符合逻辑的，也是自然的。虽然语法形式上大家会觉得有点生疏，但看多了就习惯了。

下面，再添加一个方法 **merryWith()**，表示与某个人结婚，示例如下，

```
public class Human
{
    public String name; // 名字

    public Human mate; // 配偶 (指向另外一个 Human 对象)
    // 与另一人结婚

    public void merryWith ( Human someone)
    {
```

```

        this.mate = someone; // 我的伴侣是 Ta
        someone.mate = this; // Ta 的伴侣是我
    }

    // 自我介绍
    public void introduce()
    {
        System.out.println("我叫" + this.name
            + ", 我的爱人叫" + mate.name);
    }
}

```

在 `merryWith()` 的时候，需要指定另一个人类对象 `someone` 作为参数。可以发现，方法的参数的类型也可以是任意类型，包含自身类型。在这里，`merryWith()` 的参数就是 `Human` 类型。

下面再来看一下如何使用这个 `Human` 类，示例如下，

```

public class HelloWorld
{
    public static void main(String[] args)
    {
        Human a = new Human(); // 小张
        a.name = "张";

        Human b = new Human("王"); // 小王
        b.name = "王";

        a.merryWith( b ); // a 与 b 结婚

        a.introduce(); // a 自我介绍
        b.introduce(); // b 自我介绍
    }
}

```

显然，要谈婚论嫁，必需得有两个人(a 和 b)。然后才能调用 `a.merryWith(b)` 使

a 和结为夫妻。结婚之后，再自我介绍（introduce）的时候就得把爱人的名字（mate.name）也报一下了！

此例说明了类的属性可以任意类型，包括类的自身类型。方法的参数也可以是任意类型，包括类的自身类型。此例对于初学者来说可能一时难以接受，但是随着学习的深入，慢慢就会习惯的。

第 9 章 访问控制与封装

本章学习目标

- 了解访问修饰符 `public` 和 `private`
- 理解并学会添加 `Getter` 和 `Setter`

访问控制（Access Control），是面向对象编程里的基本设计方法。本章先介绍 `public` 与 `private` 各自的意义，再了解 `Getter` 和 `Setter` 的概念。

9.1 访问修饰符

访问修饰符（Access Modifier），是指在属性和方法前面用于访问控制的关键字。本章介绍两个访问修饰符：`public` 和 `private`。

其中，`public` 是一直常见的，写在属性和方法的前面，如图 9.1 所示。

```
public class Example
{
    public int number = 10;
    public void setNumber (int number)
    {
        this.number = number;
    }
}
```

图 9.1 `public` 修饰符

简单的讲，`public` 表示公开的，`private` 则表示私有的、私密的。

打个比方，有一个家（class），其客厅是 `public` 的，而卧室则是 `private` 的。

9.1.1 私有的

一个属性或方法被声明为 `private`，表示它是私有的。此时，不允许在外部访问

它。

例如，

```
public class Example
{
    private int number = 10; // 声明为私有的
}
```

其中，`number` 前面有个 `private`，意味着在外部无法访问它。例如，

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        Example ex = new Example();
        ex.number = 10; // 错误！不能访问别人私有的属性！
    }
}
```

此时 Eclipse 会提示"The field is not visible" 的错误，意思是不能在 HelloWorld 类里访问 Example 类的 `number` 属性。

强调一下，这里所谓的“外部”，是指 Example 类的外面。比如，在 HelloWorld 里访问 Example 就算是外部。

9.1.2 可见性

可见性（Visibility），也可以称为可访问性。如果一个属性/方法可以访问，就可以称作是可见的（Visible）；反之，就是不可见的（Not Visible）。

`public` 和 `private` 用于控制可见性，区别在于：

- `public`: 外部可以访问的，可见的
- `private`: 外部不可以访问的，不可见的

在 Eclipse 里自动提示时，对于 `private` 的属性/方法会自动过滤，不会显示在候选下拉列表里。这就是可见性的意思。

值得一提的是，在类里的内部访问时，是不受 `private` 限制的。例如，

```
public class Example
{
    private int number = 10;

    public void show ()
    {
        System.out.println("值为" + this.number);
    }
}
```

其中，虽然 `number` 是 `private` 的，但在 `show()` 里可以访问，因为这是在类的内部访问的。这就好比，它们是一家人，家庭内部自然是不受 `private` 限制的。所谓的可见性，是对外人而言的。

9.2 Getter 与 Setter

`public` 和 `private` 的含义其实比较简单，理解起来应该没有太大问题。真正的问题在于，什么时候该用 `public`，什么时候该用 `private`。

其使用原则是：如果仅供内部使用，则设为 `private`；如果是给外部使用的，则用 `public`。

初学者在练习时，如果一时难以决定，则可以统一设为 `private`。当以后需要在外部访问的时候，再改成 `public` 就可以了。例如，

```
public class Example
{
    private int number ; // 将所有属性设为 private
}
```

9.2.1 Getter 方法

如果外部要获取这个 `number` 的值，则可以添加一个方法用于获取属性的值。

例如，

```
public class Example
{
    private int number ;

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

其中，像 `getNumber()` 这样用于获取属性的方法，称为 **Getter**。其名字是一般以 `get` 打头，后面加上属性的名字，并将首字母大写。

其命名规律示例如下，

`number` → `getNumber()`

`name` → `getName()`

`numView` → `getNumView()`

9.2.2 Setter 方法

同理，如果外部想要设置 `number` 的值，就得添加一个 **Setter** 方法。示例如下，

```
public class Example
{
    private int number ;

    public void setNumber(int number)
    {
        this.number = number;
    }
}
```

```
    }  
}
```

Setter 与 Getter 的命名规律基本类似，以 set 打头，并且传递一个同类型的参数。

使用 Getter / Setter 方法，就可以灵活地对一个属性进行访问控制。

- 读/写： 有 Getter，也有 Setter
- 只读： 只有 Getter
- 禁止读/写： 没有 Getter，也没有 Setter

比如，

```
public class Example  
{  
    private int number ;  
    public int getNumber()  
    {  
        return number;  
    }  
}
```

其中，由于 number 属性只有 Getter、没有 Setter，所以它对外是只读的。在外部，可以用 getNumber() 获取属性的值。示例如下，

```
Example e = new Example();  
  
int num = e.getNumber(); // 可以读取属性的值
```

9.2.3 封装

封装是一种设计方法，指的是将实现细节封装到内部，对用户是不可见的。把用户能操作的功能，设为可见的。

打个比方，一台电视机就是一个很好的封装设计。其内部的实现细节是相当复杂的，有各种元器件、线路、线圈。它们被封装在了电视机内部，对用户是不可见的。而暴露在外面的，是显示屏、按钮、信号输入插孔等，这些功能对用户是可见的。如图 9.2 所示，



图 9.2 封装的设计思想

以后在设计类的时候，也应该模仿这种设计思想。把不该让用户看到的东西，封装在类的内部（设为 `private`）；把应该让用户看到的东西，设为 `public`。

第 10 章 对象的创建与销毁

本章学习目标

- 理解构造方法的概念
- 学会构造方法的定义
- 理解构造方法的重载和传参
- 了解对象销毁回收的机制

构造方法（Constructor），是一种特殊的方法。在创建一个对象时，类的构造方法被调用。本章首先介绍构造方法的定义和重载，以及构造方法的调用。最后再了解一下 Java 里的对象回收机制。

10.1 构造方法

当创建一个对象时，如何对它的各个属性进行初始化呢？

例如，定义一个类 `Student`，

```
public class Student
{
    public String id;
    public String name;
    public boolean sex;
}
```

然后创建对象，并设定各个属性的值，

```
Student s = new Student();

s.id = "20180001";

s.name = "邵发";

s.sex = true;
```

这是前面的练习当中所用的初始化的方式。也就先创建对象，然后对它的各个属性逐个赋值。那么，有没有其他办法进行对象的初始化呢？

10.1.1 添加构造方法

构造方法用于对象的初始化。构造就是方法，有方法名和方法体。下面，在类 `Student` 里添加了一个构造方法。示例代码如下。

```
public class Student
{
    public String id;
    public String name;
    public boolean sex;

    // 添加一个构造方法
    public Student(String id,String name,boolean sex)
    {
        this.id = id;
        this.name = name;
        this.sex = sex;
    }
}
```

其中，添加一个构造方法 `public Student(...) {...}`。这种名字和类名相同的方法，就是构造方法，比如类名是 `Student`，方法名也是 `Student`。

构造方法也是方法，但是和普通的方法有着显著的区别。

- 构造方法的名字，必须和类名相同；
- 构造方法没有返回值，即使是 `void` 也不能加。

10.1.2 构造方法的作用

构造方法用于对象的初始化，在 `new` 对象的时候会调用它。例如，


```
Student s = new Student("20180001", "邵发", true);
```

其中，创建了一个 **Student** 对象，同时传入了 3 个参数。在 **new** 运行的时候，其实就是调用了 **Student** 类的构造方法，并把这 3 个参数会传递给构造方法。

也就是说，普通方法是 **s.xxx()** 这种形式调用的，而构造方法是在 **new** 的时候被自动调用的。在 **new** 出对象的时候，内部就会调用它的构造方法。

10.1.3 构造方法的重载

和普通方法一样，构造方法也可以重载。

也就是说，一个类可以有多个构造方法，只要参数列表不同就没问题。例如，

```
public class Student
{
    public String id;
    public String name;
    public boolean sex;
    public Student(String name)
    {
        this.id = "0000";
        this.name = name;
    }
    public Student(String id,String name,boolean sex)
    {
        this.id = id;
        this.name = name;
        this.sex = sex;
    }
}
```

其中，类 **Student** 里定义了 2 个构造方法。第一个构造方法的参数列表为 **(String)**，

第二个构造方法的参数列表为(String, String, boolean)。

所以在创建 **Student** 对象时，可以有 2 种选择，

```
Student s1 = new Student("小李");  
Student s2 = new Student("20180001", "邵发", true);
```

其中，由于参数列表不同，在运行时会自动匹配上相应的构造方法。

10.1.4 默认构造方法

如果一个构造方法没有参数，则称为默认构造方法，或称作无参构造方法。

例如，

```
public class Student  
{  
    public String id;  
    public String name;  
    public boolean sex;  
    public Student() // 默认构造方法，即不带参数的构造方法  
    {  
        this.id = "0000";  
        this.name = "无名";  
        this.sex = true;  
    }  
}
```

在此例中，有一个构造方法没有带参数，称之为 **Student** 类的默认构造方法。

现在有一个问题：如果一个类里没有添加任何构造方法，这种情况怎么理解呢？

在这种情况下，编译器会自动生成一个空的默认构造方法。

举个例子，类 **Example** 中没有构造方法，示例如下。

```
public class Example
{
    public int number;
}
```

由于 **Example** 类没有构造方法，所以在编译时会自动给它生成一个，等效于添加了一个空的构造方法，类似如下的代码。

```
public class Example
{
    public int number;

    public Example()    // 一个空的构造方法
    {
    }
}
```

10.1.5 构造方法的访问控制

一般情况下，构造方法应该设为 **public**。但在某些特殊情况下，它也有可能被设为 **private**。示例代码如下。

```
public class Example
{
    private int number = 10;

    private Example() // 构造方法是 private 的!
    {
    }
}
```

当一个类的构造方法不是 `public` 时,在外面就不能来 `new` 出这个对象了。例如,

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        Example ex = new Example(); // 错! Example 的构造方法
        不可见!
    }
}
```

其中,试图在 `HelloWorld` 类里 `new` 出一个 `Example()` 对象。由于 `Example` 的构造方法是 `private` 的,所以上述代码有语法错误。

这种写法一般用于高级的程序设计,对于初学者来说并不常见。了解一下即可。

10.2 对象的销毁

在 `Java` 语言,程序员只管创建对象,不用管销毁对象。对象的销毁是由系统自动完成的。比如,使用 `new` 可以创建一个对象,

```
Student stu = new Student();
```

创建完了,接着就使用这个对象即可。当对象不再被使用时,会由 `Java` 的垃圾回收机制自动回收。垃圾回收机制,简称 `GC`,即 `Garbage Collection`。

那么,什么情况下对象才是“不再被使用呢”?准确地讲,指的是当一个对象失去引用的时候。在第五章和第六章中已经介绍过什么叫“失去引用”。举一个例子,

```
Student stu = new Student();
stu = new Student();
```

这段代码就创建了两个对象,第一行创建一个 `Student` 对象,第二行又创建了一个 `Student` 对象。但是第一个对象没有人引用它了,它就是“失去引用的对象”。由于已经失去了引用,它将被 `GC` 自动回收和销毁。

关于对象的销毁，并不需要我们了解太深，因为它是一个自动的过程。大家只需要对 GC 这个术语有个印象就够了。

『Java 学习指南系列教程』

作者： 邵发

官网： <http://afanihao.cn>

QQ 群： 495734195

本系列教程由 24 篇以上视频教程组成，从入门语法到行业级技术，循序渐近式的全方位教程。内容包含入门语法和高级语法，覆盖 Java 在业界的 3 个应用领域（网站开发、安卓 APP 开发、桌面 GUI 开发）。同时包含专项技术的培训教程，如网络编程基础、数据库开发，FreeMarker, Spring, MyBatis 等。

第 11 章 继承

本章学习目标

- 了解继承的概念
- 学会方法的重写
- 学会构造方法的继承
- 理解多态和封装的概念
- 了解 `protected` 的含义

继承（Inheritance），用于描述父类型与子类型之间的关系。本章先引入继承的概念，再介绍如何在定义类的时候使用继承的语法。

11.1 类的继承

11.1.1 引例 1

在自然界中，树可以称为一个类，而苹果树也是一个类。它们之间的关系如图 11.1 所示。

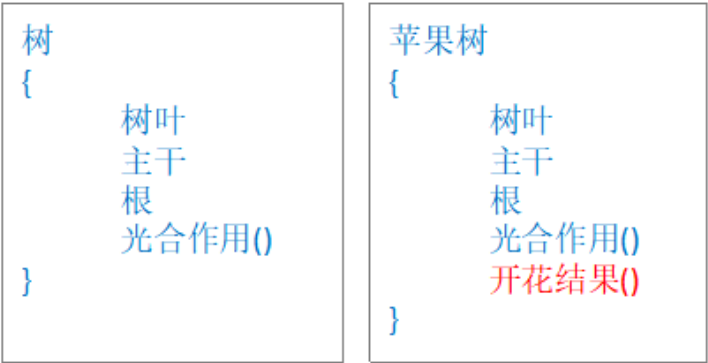


图 11.1 树与苹果树的关系

其中，树作为一个类，具有树叶、主干和根（属性），能进行光合作用（方法）。而苹果树也是一种树，所以也有树叶、主干和树，也能进行光合作用。除此之外，苹果树还有自己的特性：开花结果。

这就是继承关系，苹果树作为一种树，继承了树的所有共性。另外，苹果树也有自己的特性。

11.1.2 引例 2

下面，再给出一个贴近计算机编程的例子，进一步说明继承关系。在计算机上有文件类，而有些文件是视频文件类。这两个类之间就是继承关系，如图 11.2 所示。

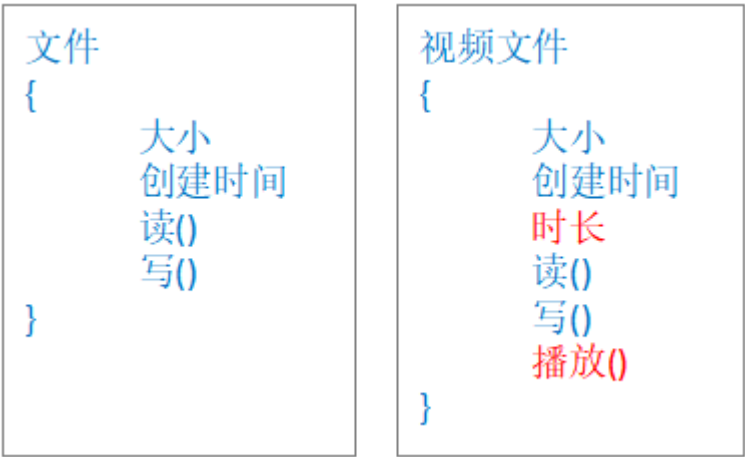


图 11.2 文件与视频文件的关系

所有的文件都有大小、创建时间等属性，都有读操作、写操作等方法。

视频文件作为文件的一种，自然也具备大小、创建时间等属性，也能读写。除此之外，视频文件还具有时长属性，能进行播放操作。可以说，视频文件类继承了文件类的所有属性和方法。

11.1.3 继承 extends

在 Java 语言里，使用 `extends` 表示类与类之间的继承关系。一般形式为，

```
public class B extends A
{
}
```

其中，A,B 是两个类。把 A 称为父类(Superclass)，把 B 称为子类(Subclass)。整体上可以称为 B 继承于 A。

当 B 继承于 A 时，那么父类 A 中的所有 public 的属性和方法，都被子类继承和拥有。还是以视频文件和文件类为例，先定义一个父类 MyFile 表示文件类，

```
public class MyFile
{
    public long size; // 文件大小
    public String name; // 文件名
    public void info() // 显示文件信息
    {
        System.out.println("文件:"+name+",大小:" + size);
    }
}
```

在父类 MyFile 中具有：

- 2 个属性：size, name，表示文件大小和名称
- 1 个方法：info()，用于显示文件的信息

下面，再添加一个 MyVideoFile 表示视频文件类，

```
public class MyVideoFile extends MyFile
{
}
```

其中，使用关键词 extends 表示继承关系，即 MyVideoFile 继承于 MyFile。

作为子类，MyVideoFile 自动地继承了父类的属性和方法。所以共同的东西就不必再写一遍了，只要写上自己特有的东西就可以了。示例代码如下，

```
public class MyVideoFile extends MyFile
```



```

{

    public int duration ; // 时长

    public void play()

    {

        System.out.println("播放视频"+ this.name);

    }

    public void stop()

    {

        System.out.println("停止播放"+ this.name);

    }

}

```

其中，`duration` 是视频文件类特有的属性，`play()`和 `stop()`方法也是视频文件特有的功能。

再来看怎么样使用子类 `MyVideoFile`，例如，

```

MyVideoFile f = new MyVideoFile();

f.size = 1293034; // 继承于父类

f.name = "abc.mp4";// 继承于父类

f.duration = 130;

f.info(); // 继承于父类

f.play();

f.stop();

```

可以看到，在 `MyVideoFile` 里并没有看到 `size`, `name`, `info()`的定义，但却可以直接使用它们。原因就是 `MyVideoFile` 继承于 `MyFile`，在 `extends` 时候已经自动继承了父类 `MyFile` 中的属性和方法。确切 `f` 说，父类里的 `public` 的东西被继承，而 `private` 的东西不被继承。

11.2 重写

重写 (Override): 是指在继承的时候, 如果觉得父类的方法不够好、不够用、不满足需求, 可以把这个方法在子类里重写一遍。

例如, 在先前的例子中, 父类 `MyFile` 表示一般性的文件, 子类 `MyVideoFile` 表示视频文件。在父类中, 有一个 `info()` 方法, 用来打印输出文件的一般信息性息。但对子类来说, 这个 `info()` 方法是不够用的, 因为它没有显示出视频的时长信息。

解决办法是, 在子类 `MyVideoFile` 里把 `info()`方法重写一遍。示例如下,

```
public class MyVideoFile extends MyFile
{
    public int duration ; // 时长

    @Override
    public void info()
    {
        System.out.println("文件名:" + this.name
            + ", 文件大小: " + this.size
            + ", 视频时长: " + this.duration
        );
    }
}
```

其中, `@Override` 是一种特殊语法, 称为注解。`@Override` 用于提示编译器这个方法是重写了父类的方法。关于注解的具体语法含义, 目前不是重点, 可以不用在意。

在子类 `MyVideoFile` 里, 重写了 `info()`方法, 打印输出了视频文件的相关信息。可以发现, 这个重写之后的 `info()`方法是满足要求的。下面再看一下调用,

```
MyVideoFile f = new MyVideoFile();
```

```
f.size = 1293034;

f.name = "abc.mp4";

f.duration = 130;

f.info(); // 子类重写了这个方法
```

在程序运行时执行的是子类里 `info` 方法的定义，如图 11.3 所示。



图 11.3 控制台的输出

11.2.1 部分重写

部分重写，就是在父类方法的基础上，再需补充修改一些功能。

还是以上述场景为例，在父类 `MyFile` 的 `info()`里，已经打印显示了文件名和文件大小。对于子类 `MyVideoFile` 来说，只需要把时长补充打印一下即可。所以，可以把 `MyVideoFile` 类稍做更改，代码如下如示。

```
@Override

public void info()

{

    super.info();

    System.out.println("视频时长"+ this.duration);

}
```

其中，`super.info()` 表示调用父类的 `info()`方法。这样，就在父类的基础上，补充了视频时长输出的功能。

在书写时应注意，重写一个方法时一般要在上面加上一行 `@Override` 注解。原则上可以不加这一行注解，但最好还是加上。关于注解的语法，后面另有专门课程来讲解。

11.3 构造方法的继承

在 Java 语言里，构造方法是自动继承的。这意味着，如果 B 继承于 A，则 A 的构造方法会被自动调用。

例如，先定义一个类 `Parent`，

```
public class Parent
{
    private int a;

    public Parent()
    {
        a = 10;

        System.out.println("父类 Parent 构造...");
    }
}
```

再定义一个 `Child` 类继承于 `Parent`，

```
public class Child extends Parent
{
    public Child()
    {
        System.out.println("子类 Child 构造...");
    }
}
```

显然，在子类中并没有看到父类的构造方法的调用。但是，运行以下代码，

```
public static void main(String[] args)
{
    Child ch = new Child();

    System.out.println("程序退出");
}
```

在这里，创建了一个 `Child` 对象，自然地会调用 `Child` 的构造方法。但在控制台的输出显示如图 11.4 所示。

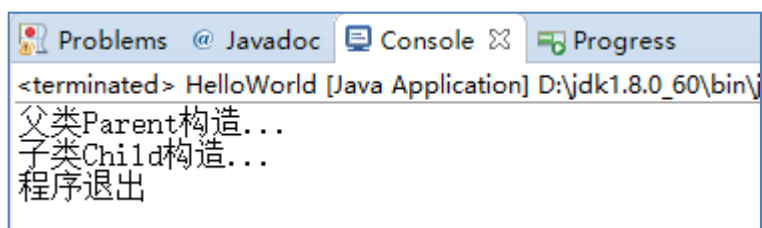


图 11.4 父类和子类的输出

这个显示结果表明，确定是先调用了父类的构造方法，再调用了子类的构造方法。也就是说，在创建子类对象时，父类的构造方法默认会被调用的。

11.3.1 显式调用父类构造方法

可以在子类里显式地调用父类的构造方法。当父类有多个构造方法的时候，就尤其有用。例如，先给父类添加多个构造方法，

```
public class Parent
{
    int a;

    public Parent()
    {
        a = 10;

        System.out.println("父类 Parent 构造...");
    }

    public Parent(int a)
    {
        this.a = a;

        System.out.println("父类 Parent 构造 222...");
    }
}
```

此时父类有 2 个构造方法，那么在子类里就可以显式地指定调用哪一个。例如，

```
public class Child extends Parent
{
    public Child()
    {
        super(12);

        System.out.println("子类 Child 构造...");
    }
}
```

使用 `super` 关键字可以显式指定调用父类的某个构造方法。例如，`super()` 表示调用父类的无参构造方法，而 `super(12)` 表示调用另一个带参的构造方法。其匹配规则在第七章（方法的重载）那一章节已经讲过。

需要注意的是，当使用 `super()` 调用时，`super()` 必须写在构造方法里的第一行。所以像下面的写法是错误的。示例如下，

```
public class Child extends Parent
{
    int number ;
    public Child()
    {
        number = 100;

        super(12); // 错! super() 必须写在构造方法里的第一行

        System.out.println("子类 Child 构造...");
    }
}
```

其中，`super` 的位置有误，应该放在 `number=100` 这一行上面。在 Eclipse 会提示错误 “Constructor call must be the first statement in a constructor”，意思是必须置于第一行。

11.4 单根继承

在 Java 语言里，一个类只能有一个父类，此特性称为单根继承。例如，下面的写法是错误的，

```
public class A extends B, C // 错误的写法!  
{  
}
```

其中，A 不能同时有两个父类 B 和 C，这是语法禁止的。

如果 A 继承于 B，B 继承于 C，C 继承于 D，也就是说 B 是父亲，C 是祖父，D 是曾祖父，可以就形成一根继承链条：

$A \rightarrow B \rightarrow C \rightarrow D$

其中，箭头表示继承关系。链条的最顶端，是顶级父类 D。

11.4.1 Object 类

在 Java 语言里，如果一个类没有显式地指定父类，则默认继承于 Object 类。例如，

```
public class Student  
{  
    public String id;  
    public String name;  
    public boolean sex;  
}
```

这个 Student 类没有指定父类，则默认父类是 Object 类。相当于写成，

```
public class Student extends Object  
{
```

```

    public String id;

    public String name;

    public boolean sex;
}

```

通常情况下，`extends Object` 是可以省略不写的。

在 Java 里，所有类的顶级父类都是 `Object`。或者说，所有的类都是 `Object` 类的子类或孙子类。以前面一节所用的例子进行说明，

```

////////// Parent.java //////////
public class Parent
{
}

////////// Child.java //////////
public class Child extends Parent
{
}

```

由于 `Parent` 的父类是 `Object`，所以最终的继承链如图 11.5 所示。

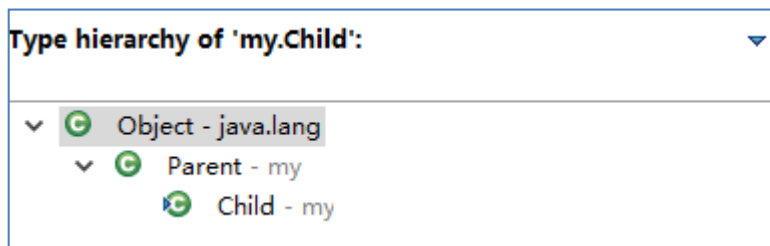


图 11.5 继承链在 Eclipse 中的显示

此图可以在 Eclipse 中得到。右键选中一个类，然后点菜单 `Quick Type Hierarchy` 显示，就能够显示这个继承树。图中可以看到，`Child` 的父类是 `Parent`，而 `Parent` 的父类为 `Object`。

11.4.2 重写 `toString` 方法

在 `Object` 中有一个方法 `toString()`，用于将对象转成字符串显示。这是一个经常

需要重写的方法。

例如，有一个类 `Student`，

```
public class Student
{
    public String id;
    public String name;
    public boolean sex;
}
```

然后在 `main()` 里调用它，

```
public static void main(String[] args)
{
    Student s = new Student();
    s.id = "20180001";
    s.name = "邵发";
    s.sex = true;
    System.out.println("学生信息:" + s);
}
```

然后在 `Eclipse` 里运行程序，在控制台里的输出如图 11.6 所示。

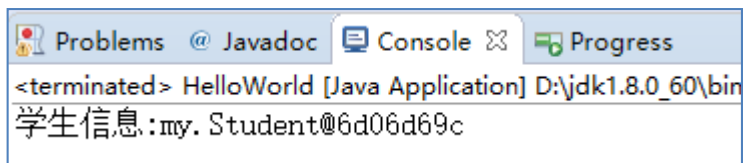


图 11.6 直接打印对象时的输出显示

在前面的章节已经强调过，Java 里的对象是默认不能打印显示的。如果强制以 `println` 打印输出，就可能会出现类似 `my.Student@6d06d69c` 的字样。（类名+对象地址）。

此时应在 `Student` 类里重写一下 `toString()` 方法，以便转成字符串显示。示例如下，

```

public class Student extends Object
{
    public String id;
    public String name;
    public boolean sex;

    @Override
    public String toString()
    {
        String result = id + " / " + name + " / " ;
        if(sex)
            result += "男";
        else
            result += "女";
        return result;
    }
}

```

再运行程序，得到控制台里的输出如图 11.7 所示。

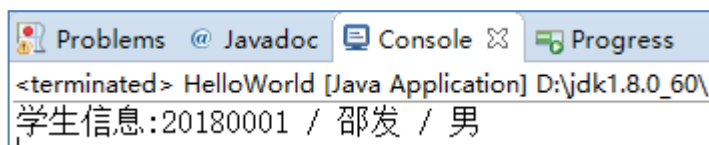


图 11.7 重写 toString() 之后的显示

其中，

```
System.out.println("学生信息:" + s);
```

相当于

```
System.out.println("学生信息:" + s.toString());
```

所以最终显示的是 `s.toString()` 方法返回的字符串。由于 `toString()` 是经常要重写的方法，所以一定要理解它的意思。

11.5 多态

多态 (Polymorphism)，是一个软件设计上的术语。具体地讲，多态在 Java 里体现为以下几种语法现象：

- 重载 (Overload)：多个方法允许有相同的名字
- 重写 (Override)：子类可以重写父类的方法
- 泛型 (Generic Type)：在 23 章中介绍，如 ArrayList, HashMap

方法的重写，就是一种多态的设计。比如说，父类 MyFile 的 info()方法，与子类 MyVideoFile 的 info()方法，两者方法名相同，但是子类把方法重新定义了一遍。同一个名字的方法，却具有两种不同的行为功能，这就是多态的设计。

(注：“多态”这个术语翻译得有点晦涩，不必纠结其字面上的意思)。

11.5.1 父子类型之间的转换

假设有一个类 Pie 表示饼干，另一个类 ApplePie 表示苹果味的饼干，它们具有继承关系。示例代码如下。

```
public class ApplePie extends Pie
{
}
```

子类对象转成父类类型是顺理成章的，例如，

```
ApplePie p1 = new ApplePie();
Pie p2 = (Pie) p1; // 类型转换: ApplePie -> Pie
```

其中，p1 是一个 ApplePie 的对象。由于“苹果味饼干是一种饼干”，所以在逻辑上可以很容易接受 Pie p2 = (Pie) p1，这是自然的、顺理成章的转换。

通常情况下，将子类对象转成父类类型，直接隐式转换就可以，

```
Pie p2 = p1; // 隐式转换即可，没有风险
```

更简洁的，可以写成：

```
Pie p2 = new ApplePie();
```

这是一种常见的写法。右侧为一个 **ApplePie** 对象，被转成 **Pie** 类型的引用。

比如，有一个类 **Baby**，需要传入 **Pie** 对象（表示宝宝想吃饼干），

```
public class Baby
{
    // 宝宝要吃饼干
    public void eat ( Pie p)
    {
    }
}
```

这个 **eat()** 方法表示：宝宝要吃饼干，需传入 **Pie** 对象。那么，现在有一块 **ApplePie**，传给它是不是也可以呢？当然可以。

```
Pie p = new ApplePie();
Baby bb = new Baby();
bb.eat( p );
```

虽然 **eat()** 方法要求传入 **Pie** 类型的对象，但传入 **ApplePie** 类型也是没有问题的，因为 **ApplePie** 就是一种 **Pie**。

11.5.2 方法的多态调用

考虑以下代码，

```
MyFile file = new MyVideoFile();
file.info();
```

那么，**file.info()** 具体执行的是 **MyFile.info()** 还是 **MyVideoFile.info()** 呢？

在做这种判断时，有一个很简单的原则：看对象的真正类型。在这里，**file** 真正指向的是一个 **MyVideoFile** 类型的对象，所以真正执行的是 **MyVideoFile** 里的 **info()** 方法。

也就是说，虽然字面上 **file** 对象是 **MyFile** 类型，但它实际指向的是一个

MyVideoFile 对象。

对于初学者来，这一语法在理解上需要一定时间。初期应强行记住这种 `Parent p = new Child()` 的形式，先记住形式，以后再慢慢理解。

11.6 protected

前面已经学过 `public` 和 `private`，下面再介绍第 3 种访问修饰符：`protected`。字面上可以勉强翻译为“受保护的”。

`protected` 的作用介于 `public` 和 `private` 之间，表示可以被子类继承、但不可以被外部访问。例如，

```
public class Example
{
    protected void show()
    {
        System.out.println(" 简单的例子");
    }
}
```

此此例子中，方法 `show()` 是 `protected`，这意味着不能从外部访问它。例如，试图在 `HelloWorld` 类里访问 `Example` 的 `show()` 方法时，编译器会提示语法错误。示例代码如下，

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        Example e = new Example();
        e.show(); // 语法错误！不能访问 protected 的方法
    }
}
```

另一方面,在定义 `Example` 的子类时, `show()` 方法可以被继承。所以说, `protected` 是介于 `public` 和 `private` 之间的。

对于初学者来说,主要应掌握 `public/private`,对于 `protected` 目前稍微了解即可。

『Java 学习指南系列教程』

作者： 邵发

官网： <http://afanihao.cn>

QQ 群： 495734195

本系列教程由 24 篇以上视频教程组成,从入门语法到行业级技术,循序渐近式的全方位教程。内容包含入门语法和高级语法,覆盖 Java 在业界的 3 个应用领域(网站开发、安卓 APP 开发、桌面 GUI 开发)。同时包含专项技术的培训教程,如网络编程基础、数据库开发,FreeMarker, Spring, MyBatis 等。

第 12 章 包

本章学习目标

- 了解包的概念
- 学会包的声明和导入

包 (Package)，是 Java 语言里用于组织代码的一种形式。可以将不同功能模块的代码分别放在不同的包里，以实现代码的清晰管理。本章介绍包的概念，以及包的相关使用。

12.1 包 package

当一个项目中的代码太多时，就需要分包管理。在 Java 里，把包叫做 package，其实本质上就是对代码的分级目录管理。

下面就一个实际项目的包结构，如图 12.1 所示。

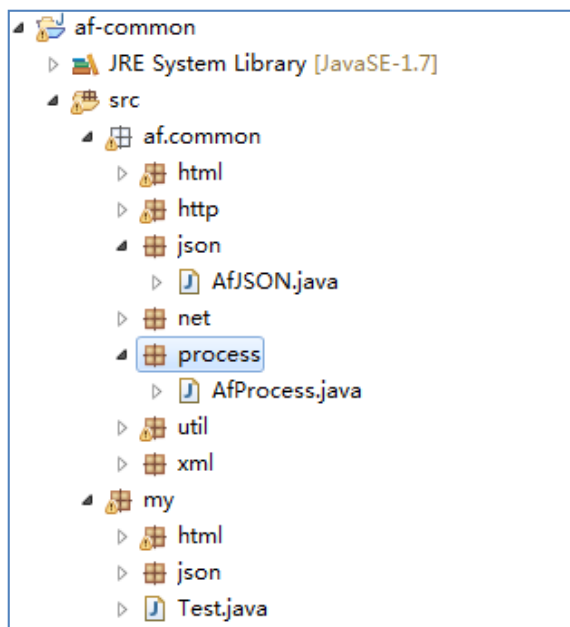


图 12.1 在 Eclipse 里观察包的结构

其中，在项目 `af-common` 下含有几十个源码文件，它们分别放在不同的包路径下。和文件目录树类似，包也是一种树状的结构。在 Eclipse 里，可以视需要创建多个包，具体操作方法请参考配套的视频教程。

包名一般以全小写命名，必要的时候可以加下划线连接多个单词。

12.1.1 包路径与类路径

包的路径用来描述包的分层结构，层次之间用点号隔开。例如，`af.common.json` 表示包 `af` 下有一个子包 `common`，`common` 下又有一个子包 `json`。

包路径其实和 Windows 的目录结构是完全对应的，在 windows 文件资源管理器里，可以观察其目录层次。如图 12.2 所示。

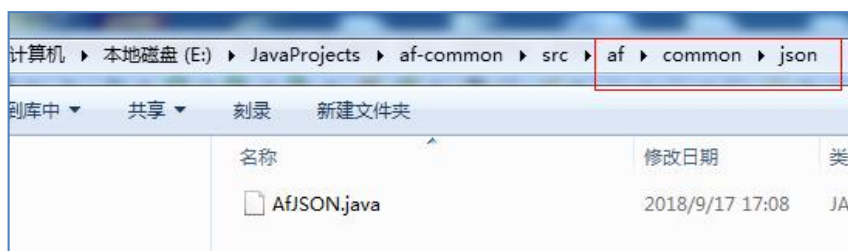


图 12.2 在 Windows 资源管理器的查看包的路径

可以发现，`af.common.json` 实际上就对应了项目源码 `src` 目录下的 `af\common\json\` 这个子目录。目录层次是一一对应的。

以后把 `af.common.json` 称为包路径。类似的，对于包下面的类，把 `af.common.json.AfJSON` 这种形式称为类的完整路径。

12.1.2 包的声明

在添加一个类时，类的第一行必须是包的声明。示例如下，

```
package af.util;
```


其中，使用关键词 `package` 表示本身所在的包路径。要求如下：

- `package` 声明必须放在 `java` 文件的第一行；
- 所声明的包路径，必须与实际路径一致。如图 12.3 所示。

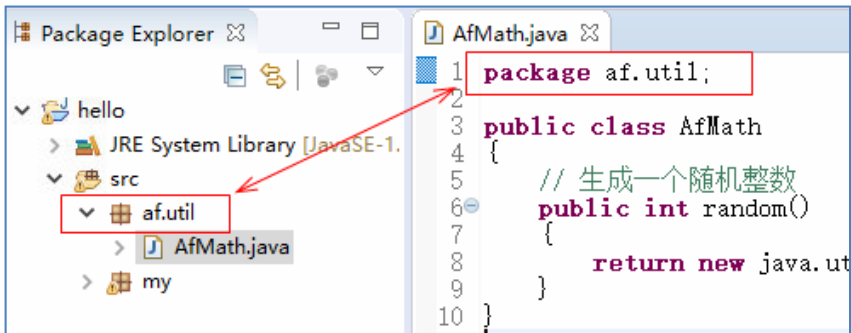


图 12.3 package 声明

一般情况下，在 Eclipse 里添加类时，会自动添加这一行包的声明。

12.1.3 包的导入 import

下面讨论一下，如何使用别的包下面的类。例如，有一个类 `my.HelloWorld`，想调用另一个包下面的类 `af.util.AfMath` 类。项目的目录结构如图 12.4 所示。

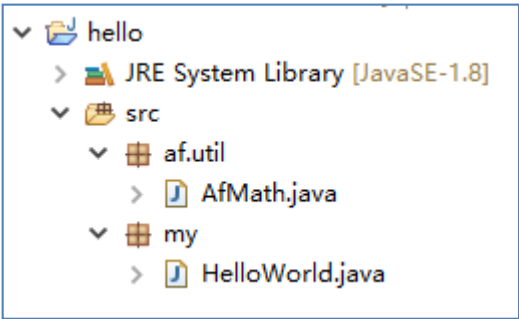
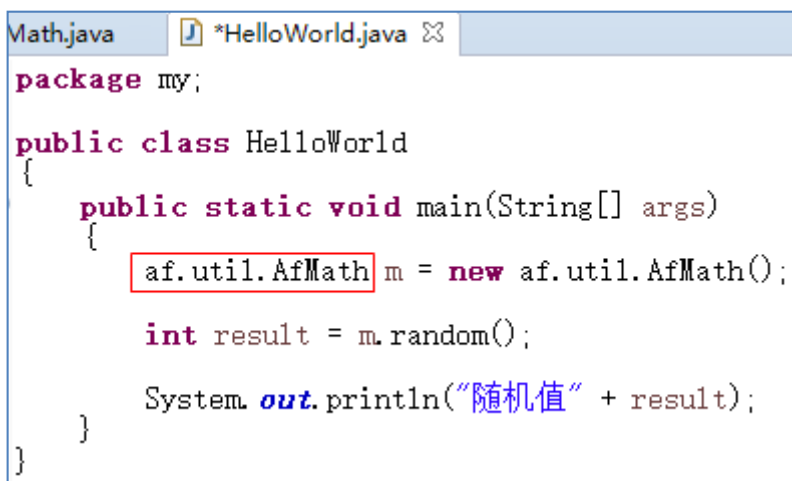


图 12.4 目录结构

可以有两种方式。第一种方式，使用类的全路径 `af.util.AfMath`，如图 12.5 所示。



```

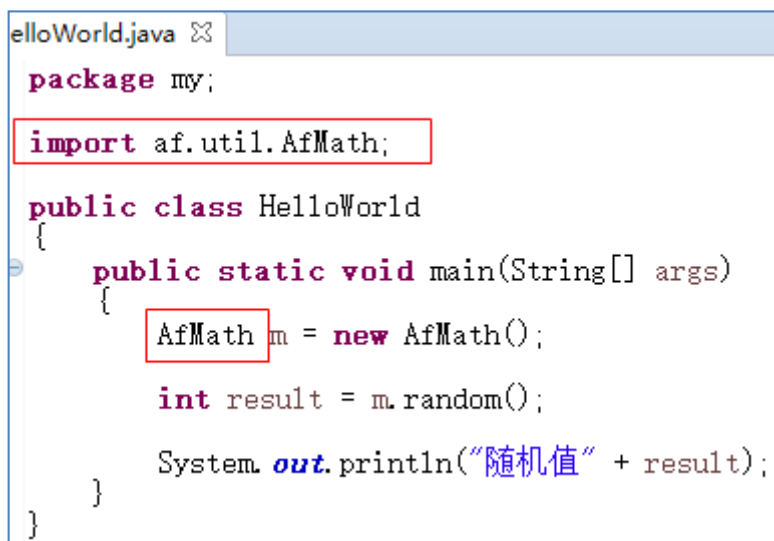
Math.java  *HelloWorld.java
package my;

public class HelloWorld
{
    public static void main(String[] args)
    {
        af.util.AfMath m = new af.util.AfMath();
        int result = m.random();
        System.out.println("随机值" + result);
    }
}

```

图 12.5 全路径的引用方式

另一方式，也是推荐的写法，使用 import 导入其他包的类，如图 12.6 所示。



```

HelloWorld.java
package my;

import af.util.AfMath;

public class HelloWorld
{
    public static void main(String[] args)
    {
        AfMath m = new AfMath();
        int result = m.random();
        System.out.println("随机值" + result);
    }
}

```

图 12.6 使用 import 引入类的路径

在类的上面，先使用 `import af.util.AfMath` 表示将要使用这个类，然后在类里使用时直接使用类的名称即可，不必再写出全路径。

在书写 import 声明时，有两种写法：

- `import af.util.AfMath;` // 导入某个类
- `import af.util.*;` // 星号表示导入该包下的所有类

`import` 是很常用的写法，必须熟练掌握。实际上在书写时，一般使用 Eclipse 自动导入相应的类，不必手写。

第 13 章 静态方法

本章学习目标

- 了解静态方法的概念和定义
- 学会静态方法的调用

静态方法 (Static Method)，在 Java 程序设计里指的是全局方法。本章先介绍静态方法的概念，然后再介绍如何定义一个静态方法并调用它。

13.1 静态方法

目前为止，已经学过的类的方法大概分为两种：一种方法，是与类的关系不大，独立于上下文，随便拷贝到别的地方仍然可以运行；另一种方法，与上下文密切相关，如果拷贝到其他类里则失去意义。

来看一下例子，

```
public class Example
{
    public boolean isPrime( int n )
    {
        for(int i=2; i<n ; i++)
        {
            if( n % i == 0)
            {
                return false;
            }
        }
        return true;
    }
}
```

```
}  
}
```

在这个类中，`isPrime()` 用于判断一个整数 `n` 是否为质数。这个方法是独立的，如果把它拷贝到其他类下，仍然可以照常使用。

再来看另一个类，

```
public class Machine  
{  
    public int money = 0;  
    public void insertCash ( int cash )  
    {  
        this.money += cash;  
        System.out.println("当前余额: " + this.money);  
    }  
}
```

在 `Machine` 类里，`insertCash()`方法不是独立的，如果拷贝到其他类里是没有意义的。因为在 `insertCash()`里要访问类的属性，也就意味着它是上下文相关，只在在 `Machine` 里才有意义。

13.1.1 添加静态方法

当一个方法比较独立、上下文无关时，可以称之为静态方法（或全局方法）。在 Java 语言里，用 `static` 关键字来声明一个静态方法。例如，

```
public class Example  
{  
    public static boolean isPrime( int n )  
    {  
        for(int i=2; i<n ; i++)  
        {
```

```
        if( n % i == 0) return false;
    }

    return true;
}
}
```

其中，`isPrime()`方法被声明为 `static`，则意味着它是一个静态方法。按照前面的理解，静态方法与上下文无关。可以检查一下，如果把 `isPrime()`方法挪到其他类里，也是可以运行的。

在静态方法里不会访问当前对象，否则它就失去了独立性，就不再是静态方法。所以，在 `static` 修饰的方法里，不允许访问当前对象 `this`。也可以这么说，“如果一个方法里访问了 `this`，那么它就不能加 `static`”。

13.1.2 静态方法的调用

静态方法是和当前对象无关的。所以在调用静态方法时，是可以直接调用的。比如，

```
boolean yes = Example.isPrime( 123 );
```

其中，直接以类名 `Example` 来访问该方法，不需要先创建出对象。

以下的写法是错误的，

```
Example ex = new Example();

boolean yes = ex.isPrime( 123 ); // 警告！静态方法应该直接以类
名调用！
```

在 Eclipse 试一下这段代码，可以发现 Eclipse 会给出一个警告。对静态方法的调用，直接 `Example.isPrime()`即可。如果使用 `ex.isPrime()`这种对象调用的方式，反而是不对的。

对于初学者来说，对静态方法的理解要求不高，只需要会调用即可。在初学阶段，在定义自己的方法时，建议不要使用 `static`，因为 `static` 的使用会影响对面向对

象设计的理解。对于大家来说，学习和理解面向对象的设计才是最重要的事情。

13.2 静态方法的使用

下面演示一下静态方法的调用。

在 Java 的类库里，不少类里都有静态方法。以 **Math** 类为例，它里面包含大量数学函数相关的方法，都是静态方法。例如，

- **abs(a)** : 求 a 的绝对值
- **pow(a,b)** : 求 a 的 b 次幂
- **sqrt(a)** : 求 a 的平方根
- **round(a)**: 四舍五入
- **sin/cos/tan** : 正弦, 余弦, 正切

当然，并不是要来一起研究数学题目，毕竟编程和数学是两回事。在这里，只是借助 **Math** 类来演示一下静态方法的使用。

比如，如果想求一个数的平方根，可以用 **Math.sqrt()** 方法。例如，

```
double result = Math.sqrt ( 4.8 );
```

可以看到，静态方法的调用形式是非常简单的。对于别人定义好的静态方法，直接以类名加方法名的形式就可以调用它。

再看一个例子。已经直角三角形的 2 条直接边 **a** 和 **b**，求斜边长度。已经两边求第三边，其实是初中数学的公式。公式：斜边 **c** = 求平方根(**a** 的平方 + **b** 的平方)。

下面用 Java 代码实现如下，

```
double a = 3;

double b = 4;

double c = Math.sqrt( a * a + b * b );

System.out.println("结果: " + c);
```

其中，需要使用 **Math** 类的 **sqrt** 方法来求一个数的平方根。如上所示，直接使

用 `Math.sqrt(value)` 即可。通过演示，是不是觉得 `Math.sqrt()` 方法就是一个工具呢，所以有时也把静态方法称为工具方法。

13.3 程序的入口

程序的入口，即程序运行的起点。当运行一个 Java 程序时，从 `main` 方法的第一行开始运行。`main` 方法的一般形式如下：

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

其中，`main` 方法的形式有着严格的规定，不能随意改变。规定如下：

- 必须为 `public`
- 必须为 `static`
- 返回值必须为 `void`
- 参数必须为 `String[]`

也就是说，`main` 方法是一个约定好的写法，形式是固定的。以下写法都是错误的。

- `public static void main()` // 必须带 `String[]` 参数
- `public static int main(String[] args)` // 必须返回 `void`
- `public void main(String[] args)` // 必须是 `static`

由于 `main()` 方法是静态方法，所以按照前面的介绍，`main` 方法可以写在任何类下面。

另外，在一个 Java 项目里，是允许存在多个 `main` 方法的。比如，在 `A.java` 里有一个 `main`，在 `B.java` 里也可以有一个 `main`，这是没问题的。在启动程序时，需要指定从哪一个文件的 `main` 开始运行。

在 Eclipse 里运行程序时，首先要打开 `main` 方法所在的 Java 文件，然后右键执

行菜单里的 **Run As | Java Application** 便可以启动程序。这一步其实就是指定了以当前类里的 **main** 方法作为程序的入口。

第 14 章 常见工具类

本章学习目标

- 学会字符串 `String` 类的常见操作
- 了解包装类型与基本类型的转换
- 学会控制台界面的输入
- 了解随机数的生成和应用
- 了解字符的概念，字符与字符串的关系

本章介绍几种常见工具类的使用。在程序设计中，不但要学语法规则，还要掌握一些常用的工具，组合在一起才能构成最终有实际用途的程序。

14.1 字符串

`String` 类和 `Math` 类一样，也是 Java 自带的基础类，前面的练习中已经反复使用过。本节介绍 `String` 类的几个常见的用法。

14.1.1 字符串的拼接

使用加号可以直接拼接字符串。例如，

```
String s = "阿发" + "你好";
```

当拼接一个对象时，会自动调用该对象的 `toString()` 方法，例如，

```
Student w = new Student( 201801, "wang", true);  
  
String result = "结果:" + w;
```

相当于

```
String result = "结果:" + w.toString();
```

14.1.2 字符串的长度

在 Java 语言里,无论是中文字符还是英文字符,都只算作一个字符。使用 `length()` 方法可以取得字符串对象的长度,例如,

```
String s = "阿发nihao";  
  
int n = s.length(); // 长度为7
```

其中,在统计字符长度时是不区分中英文的,所以这个字符串的长度为 7。

14.1.3 空字符串

空字符串,是指长度为 0 的字符串。例如,

```
String s = "";  
  
int n = s.length();
```

其中, `s` 是一个正常的字符串对象,其中有 0 个字符。

注意,空字符串和 `null` 是两回事。`null` 表示空对象,即不指向任何对象。

14.1.4 获取子串

使用 `substring()` 方法可以获取一个字符串的子串,需要传入起止位置作为参数。例如,

```
String s = "阿发nihao";  
  
String a1 = s.substring(1); // 从 s[1] 开始到最后  
  
String a2 = s.substring(1, 4); // s[1],s[2],s[3]
```

其中, `substring(startIndex, endIndex)` 是指从 `startIndex` 位置开始,到 `endIndex` 结束。`endIndex` 本身不包含在内。

14.1.5 判断内容是否相同

使用 `equals()` 方法可以判断两个字符串的内容是否相同。例如,

```
String s1 = new String("nihao");  
String s2 = new String("nihao");  
if(s1.equals(s2))  
{  
    System.out.println("相同");  
}
```

可能有人会问，为什么不直接使用==来判断呢，示例如下，

```
if(s1 == s2 ) // 错误的写法!  
{  
    System.out.println("相同");  
}
```

这是初学者的一个常见错误。在 Java 里，== 用于判断 s1 和 s2 是否“同一个对象”，而 equals 方法用于判断 s1 和 s2 是否“内容相同”。

打个比方：小王和小李都是 18 岁，则可以说 wang.equals(li)。但是，小王和小李显然不是同一个对象(wang != li)。

14.1.6 字典序比较

字典序比较，即比较两个字符串的先后顺序。例如，要把一个班上所有同学的姓名排序，就得用 compareTo() 方法。示例如下，

```
String s1 = "liming";  
String s2 = "lilei";  
if( s1.compareTo( s2 ) < 0)  
    System.out.println(s1 + "位于前面");  
else  
    System.out.println(s2 + "位于前面");
```

其中，由于 if 语句后只跟一条语句，所以可以把大括号省略不写。但对于初学者，还是要坚持使用大括号的。

对于 `s1.compareTo (s2)`，如果返回值小于 0，表示 `s1` 在前面。如果大于 0，表示 `s1` 在后面。如果等于 0，表示 `s1` 和 `s2` 内容相等。

需要注意的是，"`Liming`" 和 "`liming`" 不是相等的字符串。按照字典顺序，`Liming` 应排在 `liming` 的前面（即大写字母靠前）。如果想按不区分大小写的方式比较，可以用 `compareToIgnoreCase()` 方法。

14.1.7 格式化

使用 `format()` 可以格式化字符串。`format` 是 `String` 类的静态方法，示例用法如下，

```
String s = String.format(
    "姓名:%s, 年龄: %d , 身高: %.2f"
    , "邵发"
    , 35
    , 1.75 );
```

其中，返回的结果 `s` 的值为 "姓名:邵发, 年龄: 35 , 身高:1.75" 。

`String.format()` 的第一个参数是格式字符串，表示结果的格式。里面的 `%s`，`%d`，`%.2f` 为格式符号，将替换为后面参数的值。

其中，每个格式符号的含义如下：

- `%s`：替换为一个字符串 (`s`, `string`)
- `%d`：替换为一个整数 (`d`, `decimal`)
- `%.2f`：替换为一个小数，并保留 2 位小数点 (`f`, `float point`)

14.1.8 查找子串

使用 `indexOf()` 方法可以在一个字符串里查找一个子串。例如，

```
String s = "China is a great country";
int pos = s.indexOf("great");
```

```
if(pos <0)
{
    System.out.println("没找到");
}
```

其中，`indexOf()` 方法将返回"great"出现的位置。如果找到了，则返回首字符 `g` 的位置；如果没找到，则返回-1。

`indexOf()`还有另一个重载的方法，带一个参数 `fromIndex`，表示从什么位置开始查找。例如，

```
int pos = s.indexOf("great", 10); // 从 s[10]开始找
```

14.1.9 前缀后缀

`String` 类还有两个方法：`startsWith()` 用于判断前缀，`endsWith()` 用于判断后缀。

示例代码如下，

```
String filename = "test.mp4";

if( filename.endsWith(".mp4") ) // 判断是否以.mp4 结尾
{
    System.out.println("支持此视频格式!");
}
```

其中，使用 `endsWith()` 方法为判断 `filename` 是否以".mp4" 结尾。

14.1.10 清除空白

使用 `trim()` 方法，可以清除字符串左右两边的空白字符。例如，

```
String s =" 你好 yes OK "; // 此字符串左右两边存在空白字符

String s2 = s.trim();
```

其中，字符串 `s` 的左右两边的空白字符都会被清除掉，并返回清理后的结果："你好 yes OK"。需要注意的是，字符串里面的空白不会被清除，仅清除两边的空白字符。

14.1.11 分割

使用 `split()` 方法可以将一个字符串分割为多个子串，返回一个字符串数组。例如，将一个字符串按逗号分割，示例如下，

```
String s = "小张, 小王, 小李";  
String[] names = s.split(",");  
for (int i=0; i<names.length; i++)  
{  
    String name = names[i].trim();  
    System.out.println(name);  
}
```

14.2 包装类

在第 11 章中曾经提到，Java 是完全面向对象的语言，所有的类的顶级父类都是 `Object` 类。然而，有一些类型却似乎游离于这个规则之外，那就是基本类型 `byte`, `short`, `int`, `long`, `double`, `float`, `boolean` 和 `char`。

其中的有些类型，如 `int`, `double`, `boolean` 类型，是经常使用的。可以发现，这些基本类型并不是 `class`，因而在将来的某些使用场景下，会存在一些尴尬（第 23 章会介绍）。

为了将所有的类型统一到“类”的范畴，可以将基本类型包装一下。比如，可以设计一个 `MyInteger` 类来表示整数，

```
public class MyInteger  
{  
    public int value;  
    public int getValue()  
    {  
        return this.value;  
    }  
}
```

```

    }

    public void setValue(int value)
    {
        this.value = value;
    }
}

```

这样便得到一个类 **MyInteger**，这个类看起来和 **int** 差不多，但它已经被统一到 **Object** 的子类，以后便可以按类和对象的设计思路来使用。

14.2.1 包装类

在 Java 里，每一种基本类型都对应一个包装类型（**Wrapper Class**）。所列如下，

long	<=>	Long
int	<=>	Integer
short	<=>	Short
byte	<=>	Byte
double	<=>	Double
float	<=>	Float
boolean	<=>	Boolean

可以看到，**int** 对应的包装类型是 **Integer**，它是 Java 自带的基础类。下面就以 **Integer** 类为例，介绍包装类的一般用法。

14.2.2 装箱拆箱

先来看一下，**int** 类型与 **Integer** 类型如何相互转换。

```

Integer a = new Integer(10); // int => Integer

int b = a.intValue(); // Integer => int

```

可以看到，**int** 与 **Integer** 的转换非常的简单。其中，把 **int** 转成 **Integer** 的过程，

称为“装箱”过程；反之，把 `Integer` 转成 `int` 的过程称为“拆箱”。这两个术语其实没有什么高深的地方，就是相互转换一下而已。

进一步地，上述的转换过程还可以写得更简洁一些，示例如下，

```
Integer k = 123;

int m = k;
```

这样的自动转换的过程称为“自动装箱拆箱”。也就是说，`int` 与 `Integer` 之间的转换几乎没什么好注意的，几乎可以随心所欲地使用。

14.2.3 Integer 与 String 转换

下面演示一下 `Integer` 与 `String` 之间的转换。示例如下，

```
String s = String.valueOf( 123 ); // int => String

int a = Integer.valueOf("123"); // String => int
```

其中，`Integer.valueOf()` 是一个静态方法，用于将 `String` 转成 `Integer`。

实际上，包装类 `Integer` 里还提供了多个实用的方法，以方便进行整数相关的操作。

14.2.4 值的比较

对于包装类型的对象，要比较其值的大小，应该用 `equals` 方法，不能用 `==` 或 `!=` 来比较。例如，以下代码是一个初学者容易犯的错误，

```
Integer a = new Integer(123);

Integer b = new Integer(123);

if( a == b ) // 错误!!

    System.out.println("相等");

else

    System.out.println("不相等");
```

应该改为：

```
if( a.equals(b) ) // 正确

    System.out.println("相等");

else

    System.out.println("不相等");
```

原因在 14.1 节里在对字符串进行比较时，已经说过。

在 Java 里，`==` 用于判断 `s1` 和 `s2` 是否“同一个对象”，而 `equals` 方法用于判断 `s1` 和 `s2` 是否“内容相同”。

打个比方：小王和小李都是 18 岁，则可以说 `wang.equals(li)`。但是显然，小王和小李不是同一个对象(`wang != li`)。

14.2.5 包装类的作用

为什么要设计这些包装类呢？主要原因有两条。

其一，包装类里包含了大量的工具方法。刚才所介绍的 `Integer.valueOf()` 就是其中之一。

其二，在设计上，把所有类型统一成“类”的范畴，都是 `Object` 的子类，以方便程序的统一设计。

14.3 控制台界面

控制台（`Console`），是在 20 世纪，在计算机图形界面出现之前的、一种基于文字的用户交互界面。如今的操作系统下一般都还保留有这种古老的操作方式，一般称为命令行窗口或者命令行终端。比如，在 Windows 下就可以找到 DOS 命令行窗口，如图 14.1 所示。

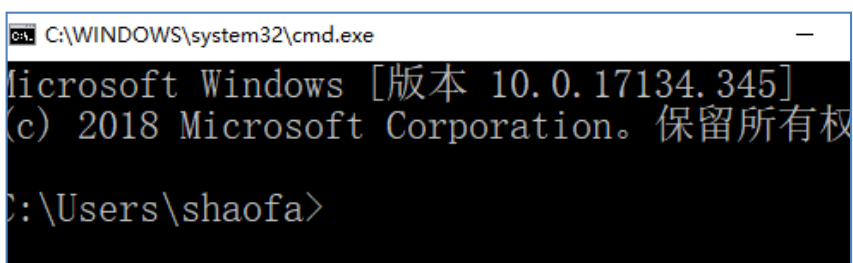


图 14.1 控制台窗口界面

这种黑乎乎的窗口，就称为控制台窗口（Console）。在这个窗口里，只能以输入命令行的方式与程序交互。

现今，这种基于控制台的输入输出方式早已过时，那为什么在这一节里还要介绍这种技术呢？

- 控制台技术在将来的正式开发中（公司里）不会用到；
- 介绍控制台技术，目的仅在于辅助语法练习、模拟用户交互。

14.3.1 控制台类 AfConsole

为了便于初学者的练习，提供一个控制台工具类 AfConsole，使用它可以很方便地实现基于控制台界面的程序。AfConsole 类在课程源码目录里可以找到，直接拷贝添加到项目里即可。如图 14.2 所示。

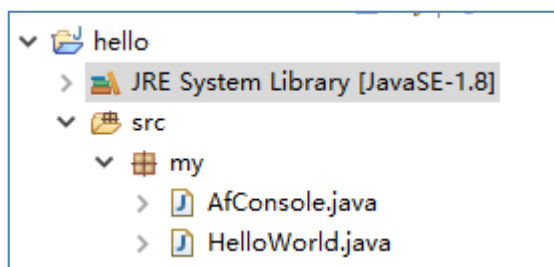


图 14.2 在项目中加入 AfConsole 工具类

下面用一个例子来展示 AfConsole 的使用，如下所示。

```
1 AfConsole c = new AfConsole();  
2 c.print("请输入用户名: ");
```

```
3  String username = c.readString("guest");
4  c.print("请输入密码: ");
5  String password = c.readString("");
6  if(password.equals("123456"))
7  {
8      c.println(username + ",你好! ");
9  }
10 else
11 {
12     c.println("密码错误! ");
13 }
```

其中，

第 1 行，创建一个 AfConsole 对象；

第 3 行，c.readString() 用于从控制台获取用户输入的用户名；

第 5 行，c.readString() 获取用户输入的密码；

第 6 行，判断用户输入的密码是否正确，如果正确，则提示成功登录；否则，提示密码输入错误。

可能有人会问：在其他教程看到的 Scanner 类，为什么在这里不用呢？原因在于，Scanner 这个类用处不大，属于过时的设计，在以后的工作中绝无可能用到。另外 Scanner 类也存在一些问题，不宜使用。所以提供了 AfConsole 工具类，可以更容易地为初学者所用。

14.4 随机数

随机数在工程应用里时而会用到。比如，在设计牌类游戏时，洗牌操作就运用了随机数技术，每次开始游戏之前要把所有的牌随机打乱。又比如，在抽奖、摇号、抽签等应用中也需要使用随机数。

在涉及随机数时，需要使用 Java 提供的一个工具类 `java.util.Random`。例如，

```
Random rand = new Random();  
  
int s = rand.nextInt(1000);
```

其中，`nextInt(1000)` 表示生成一个 0~1000 之间的随机数，结果里不包含 1000。

下面通过几个例子，来演示 `java.util.Random` 这个类的使用方法。

注意，`rand.nextInt(N)` 生成的随机数范围是 `[0.. N-1]`，是不包括 `N` 的。

14.4.1 示例 1

先看一个简单的应用。要求生成 3 个 0~1000 的随机数。

示例代码如下，

```
package my;  
  
import java.util.Random;    // 导入 Random 类  
  
public class HelloWorld  
{  
  
    public static void main(String[] args)  
    {  
  
        Random rand = new Random();  
  
        for(int i=0; i< 3; i++)  
        {  
  
            int s = rand.nextInt(1000); // 结果不包含 1000  
  
            System.out.println("生成随机数: " + s);  
  
        }  
  
    }  
}
```

注意，在使用 `Random` 类时，要记得使用 `import java.util.Random` 导入一下。

运行这段代码，可以发现每次会生成 3 个位于 0-1000 之间的随机数。

14.4.2 示例 2

公司有 96 人，在年会活动上抽奖。要求写一个程序，抽出一个特等奖。特等奖仅 1 名。示例代码如下，

```
Random rand = new Random();  
  
int s = rand.nextInt(96);  
  
System.out.println("恭喜:" + (s+1) + "号员工!");
```

其中，`rand.nextInt(96)` 将返回介于 0-95 之间的随机数。

14.4.3 示例 3

公司有 96 人，在年会活动上抽奖。要求写一个程序，抽出 3 个一等奖。示例代码如下，

```
int[] result = new int[3];  
  
int count = 0;  
  
Random rand = new Random();  
  
while( count < 3)  
{  
  
    // 抽一个幸运员工，s 是它的号码  
  
    int s = rand.nextInt(96);  
  
    // 检查 s 是否已经中过奖了  
  
    boolean exist = false;  
  
    for(int i=0; i<count; i++)  
    {  
  
        if(result[i] == s)  
        {  
  
            exist = true;  
  
            break;  
  
        }  
  
    }  
  
}
```

```

    }

    if(exist)
    {
        continue;
    }

    else
    {
        result[count] = s;

        count ++;
    }
}

// 打印显示中奖结果
for(int i=0; i<result.length; i++)
{
    System.out.println("第" + (result[i] + 1) + "号获
奖!");
}

```

14.5 字符

字符（Character），代表一个文字或符号。在 Java 语言里，用 char 类型表示字符。例如，

```

char c1 = 'A'; // 英文字母
char c2 = '9'; // 数字
char c3 = ' '; // 英文标点（空格）
char c4 = '邵'; // 中文
char c5 = '。'; // 中文标点（句号）
char c6 = 'の'; // 日文
char c7 = '며' ; // 韩文

```

```
char c8 = 'β'; // 希腊字母 c1 = 'A';
```

也就是说，无论是中文、日文、韩文、希腊文，几乎地球上人类的所使用的所有文字和符号，都可以用 `char` 表示。

14.5.1 字符的写法

在书写字符常量时，须用单引号括起来。例如，

```
char c4 = '邵';
```

看起来很简单，但是初学者可能会犯以下错误，

(1) 只能表示一个字符

```
char a = 'Ab'; // 错误! 单引号内只能有一个字符 !
```

(2) 要用英文单引号括起来

```
char a = 'A' ; // 错误! 不能用中文单引号 !
```

(3) 不要乱加空格，因为空格也是一个字符

```
char a = 'A ' ; // 错误! 单引号内不能多加空格!
```

(4) 必须用单引号，而不是双引号

```
char a = "邵"; // 错误! 双引号表示的是字符串!
```

14.5.2 字符编码

实际上，每个字符都对应一个数字，称为字符编码。

例如，

```
char c = 'A';  
  
int n = c;  
  
System.out.println("编码为: " + n);
```

其中，把一个字符 `c` 可以直接转成 `int` 整数。运行这段代码，可以发现字符'A'对应的数字为 65。再来了解几个常见的字符的编码：

- 'A'为 65 , 'B'为 66 ...

- 'a'为 97, 'b'为 98...
- '0' 为 48, '1' 为 49 ...
- 空格字符' '是 32 ..

必须强调的是，并不要求记住每个字符的编码是多少。因为字符的数量很多，不但有英文字符，还是中文、欧洲文字、标点符号等。数量极大，是不可能记住的。程序员只需要了解，每个字符都对应一个数字即可。

地球上所有的语言和文字符号加起来大概有十几万之多，把这些符号统一进行编号，称为 Unicode 编码。例如，字符 '邵' 的 Unicode 编码是 37045。可以用下面的代码来验证一下，

```
char c = '邵';  
  
int n = c;  
  
System.out.println("编码为: " + n);
```

这段代码，就可以打印输出字符'邵' 对应的 Unicode 编码是多少。

在所有的 Unicode 编码序列中，把前面 128 个字符的编码，称为 ASCII 编码。也就是说，ASCII 码是 Unicode 的子集，是很小的一部分。其中包括了小写字母、大小写字母、数字(0-9)、英文标点符号、控制字符等。

14.5.3 字符算术

由于字符对应一个整型数字，所以字符也可以进行简单的加减算术运算。例如，

```
char result = 'A' + 2;
```

由于'A','B','C', ..., 'Z' 序列的编码是递增的，因此可以推算出 result 的值应该是'C'。再例如，

```
int n = 'C' - 'A'; // 值为 2  
  
int k = '2' - '0'; //值为 2
```

需要明确的是，'2' - '0' 其实就是 50 - 48，是它们对应的数字进行减法操作。

14.5.4 字符与字符串

字符串，其实就是一串字符。所以 `char` 和 `String` 之间是可以相互转换的。

`String` → `char` 的转换

```
String str = "afanihao 阿发你好";

char ch = str.charAt(8); // 下标从 0 开始计数

char[] chs = str.toCharArray();
```

`char` → `String` 的转换

```
char[] chs = {'阿', '发', '你', '好'};

String str = new String(chs, 1, 3);

str += '的';
```

再例如，在字符串字符中查找字符，可以用 `String` 的 `indexOf()` 方法。示例代码如下，

```
String str = "afanihao 阿发你好";

int p = str.indexOf('发');

System.out.println("位置: " + p);
```

不过，在 Java 项目经常使用字符串 `String` 进行操作，却很少使用字符 `char`。所以这节内容只需要简单了解即可，不是学习的重点。关于字符类型，只需掌握两点：

- 学会用单引号表示一个字符；
- 了解每个字符都对应一个数字，但不需要记住具体的值。

14.6 关于 equals

在 Java 里，要比较两个对象的内容是否相等，应该使用 `equals` 方法。比如，在比较 `String` 对象和 `Integer` 对象时，都强调了应该使用 `equals` 方法进来比较。

实现上，`equals()`方法是 `Object` 类的一个方法。由于 Java 里所有的类的祖先都是 `Object`，所以所有的类都继承了 `equals` 方法。

默认的 `equals` 方法并不能真正实现对象的比较，子类需要重写 `equals` 方法。比

如，有一个表示学生的类 **Student**，示例代码如下。

```
public class Student
{
    public int id; // 学号
    public String name; // 姓名
    public boolean sex; // 性别
    public Student(int id, String name, boolean sex)
    {
        this.id =id;
        this.name =name;
        this.sex = sex;
    }
}
```

再定义两个 **Student** 对象，

```
Student a = new Student(20190001,"小王", true);
Student b = new Student(20190001,"xiaowang", true);

if(a.equals( b))
    System.out.println("a,b 相等");
else
    System.out.println("a,b 不相等!");
```

这样的两个对象，用 **equals** 方法进行比较，它们是不是相等呢？显然，**a** 和 **b** 的学号相同，但姓名有差异，默认的 **equals** 无法给出想要的比较结果。如果决定使用 **equals** 进行比较，就必须重写 **equals** 方法，在 **equals** 里面决定如何比较。

例如，可以按学号进行比较，只要学号相同就判断为相等。在 **Student.java** 类里添加 **equals** 方法的实现，示例代码如下。

```
public class Student
{
```

```

public int id;

public String name;

public boolean sex;

public Student(int id, String name, boolean sex)
{
    this.id =id;

    this.name =name;

    this.sex = sex;
}

@Override
public boolean equals(Object obj)
{
    Student b = (Student)obj;

    if(this.id == b.id)

        return true;

    return false;
}
}

```

其中，在 `Student` 类里重写了 `equals` 方法，以学号为依据判断两个 `Student` 对象是否相等。

默认的，自定义的类型都是不可以直接用 `equals` 比较的。如果要用 `equals` 比较，就得先确定一下它是否已经正确地重写了 `equals` 方法。

在实际的项目中，一般不会去专门重写 `equals`（有点麻烦），而是直接按需要进行比较。示例代码如下。

```

Student a = new Student(20190001,"小王", true);

Student b = new Student(20190001,"xiaowang", true);

if(a.id == b.id) // 直接比较学号就行，不必专门重写 equals()

```

```
        System.out.println("a,b 相等");  
    else  
        System.out.println("a,b 不相等!");
```

第 15 章 链表

本章学习目标

- 了解容器的概念
- 学会链表的构造和遍历
- 学会链表的插入和删除的操作
- 了解 ArrayList 类的使用

链表 (Linked List)，是一种常见的数据结构，是软件工程师必须理解的基本结构。本章介绍链表的定义和基本使用，并在设计上将链表和容器统一起来。

15.1 容器

容器 (Container)，就是能存放若干对象的东西。容器是一个设计上的术语，并不是语法概念。

比如说，数组就是一个容器，数组里可以容纳多个对象。例如，

```
Student[] ss = new Student[4];
```

则 ss 是一个容器，里面最多可以存放 4 个对象

```
ss[0] = new Student("20170001", "邵发");  
ss[1] = new Student("20170002", "小张");  
ss[2] = new Student("20170003", "小王");  
ss[3] = null;
```

此时容器里有 3 个 Student 对象，还剩 1 个空闲位置。

其实可以形象地理解为，一排 4 个座位，现在已经坐了 3 个人，空一个位置。

邵	张	王	
---	---	---	--

现在考虑数组的插入操作。假设又有一个学生小李，他希望坐在第 2 个位置。代码如下，

```
ss[3] = ss[2]; // “王” 挪一个位置
ss[2] = ss[1]; // “张” 挪一个位置
ss[1] = new Student("20170004", "李");
```

现在，4 个位置已经坐满，如下所示，

邵	李	张	王
---	---	---	---

在上面的操作中，插入一个元素，需要移动两个元素的位置。试想，如果有 100 个元素，那一个插入操作可能就要做 99 次的数据移动，这个代价是很大的。

所以，数组容器有明显的缺点。缺点一，数组的容量是固定的，无法动态调整。缺点二，数组元素的插入和删除都比较复杂。（一人插队，后面所有的人都要挪动位置）

所以希望有一种新的容器类型，既可以存放对象，又能克服数组容器的缺陷，这就是下面所要介绍的：链表。

15.2 链表

链表（Linked List），也是一种容器。在高校里的“数据结构与算法”这门课里，一般都有链表这种数据结构的介绍。

什么叫链表呢，其实就是一种链状的组织结构。比如，现在有 4 猴子(Monkey)，它们目前是孤立存在的对象，互相没有联系。如图 15.1 所示。



图 15.1 4 个孤立的对象

现在，把它们串联起来，形成一个链表结构。怎么串起来呢？让每只猴子握住前一只猴子的尾巴，从而形成一个队伍。如图 15.2 所示。



图 15.2 4 个串联的对象

这种串在一起的逻辑结构就称之为“链表”。

15.2.1 链表的构造

下面来看，在 Java 代码里怎么表示一个链表。首先，定义一个 `Monkey` 类表示猴子。示例代码如下，

```
public class Monkey
{
    public int id;          // 编号
    public String name;     // 名字
    public Monkey next;     // 它后面的猴子
    public Monkey(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
    @Override
    public String toString()
    {
        return String.format("(%s, %s)", name, id);
    }
}
```

其中，`Monkey` 类应重写 `toString()` 方法，以方便调试和打印。关于 `toString()` 的具体写法，可以参考前面的 11.4 节。

`Monkey.next` 表示下一只猴子。如果对这种写法有点生疏，可以回顾一下 8.4 节的内容。

现在，创建 4 个 Monkey 对象，

```
Monkey m1 = new Monkey(1, "圆圆");  
Monkey m2 = new Monkey(2, "方方");  
Monkey m3 = new Monkey(3, "角角");  
Monkey m4 = new Monkey(4, "朱朱");
```

则 m1, m2, m3, m4 是四只猴子，下面把它们依次串起来，

```
m1.next = m2; // m2 握住 m1 的尾巴  
m2.next = m3; // m3 握住 m2 的尾巴  
m3.next = m4; // m4 握住 m3 的尾巴  
m4.next = null; // m4 后面没有猴子
```

像这样，m1 的 next 是 m2，m2 的 next 是 m3，m3 的 next 是 m4，于是这四个对象便有了逻辑上的串联关系，形成一条链状的结构。这种逻辑结构就是链表。

15.2.2 几个概念

链表：这种以“链”状形式串起来的结构。链表不是表，而一条链子。

节点：链表里的每个对象（Node），称为节点。

链表头：最前面的那个节点，称为链表头，或称为头节点。

15.2.3 链表的遍历

链表的头节点在链表中起到至关重要的作用，因为有了头节点，就可以顺着链条一直往下把每个节点遍历出来。

比如，前面构造的链表的头节点是 m1。从 m1 一直往后遍历，m1.next 是 m2，m2.next 是 m3，m3.next 是 m4。m4.next 是 null，说明 m4 是最后一个节点（尾节点）。

以下代码展示了链表的一般遍历方法，

```
Monkey node = m1;    // 从头节点开始

while( node != null)

{

    System.out.println("节点: " + node.name)

    node = node.next;

}
```

也就是说，只要有了头节点，就可以遍历出链表里的每一个节点，直到末尾。

15.3 插入节点

现在来考虑一下，怎么向链表中添加一个节点。

假设现在链表中已经有了 4 只猴子，编号依次为①②③④，如图 15.3 所示。



图 15.3 一个拥有 4 个节点的链表

现在要往这个链表里添加第⑤号猴子。有几种方案，可以把它附加到末尾，也可以插到①②之间，等等。下面分别讨论一下各种方案应该如何实现。

15.3.1 添加到末尾

要把⑤号节点添加链表的末尾，首先需要找到尾节点。

```
// 先找到尾节点

Monkey tail = m1;

while( tail.next != null )

{

    tail = tail.next;

}

// 把 5 号节点附加到末尾
```

```
Monkey m5 = new Monkey(5, "花花");  
tail.next = m5;
```

其中，先用一个遍历操作找到链表的尾节点。然后，`tail.next = m5`，将新节点挂在尾部。

像这样，这个链子上就有了 5 个节点，如图 15.4 所示。



图 15.4 新节点添加在末尾

15.3.2 添加到前面

也可以把⑤号节点添加到头节点之后。示例代码如下，

```
Monkey head = m1;  
m5.next = head.next; // 把②挂到⑤之后  
head.next = m5;      // 把⑤挂到①之后
```

最终结构如图 15.5 所示。



图 15.5 新节点添加在前面

注意在这段代码中，如果改成下面的操作顺序，是错误的。示例如下，

```
Monkey head = m1;  
head.next = m5; // 把 ⑤ 挂到 ① 之后  
m5.next = head.next; // 出错！此时 head.next 是 ⑤
```

这样的代码看起来和上面一样，只颠倒了两行代码的顺序，为什么就不对了呢？大家要仔细梳理清楚。为了便于理解，可以自己在纸上画出来。

15.3.3 添加到指定节点之后

还有一种情形，就是插入到指定的节点之后。比如，要求插入到值为 2 的节点之后。其基本的实现逻辑是：遍历链表，找到目标节点，然后把新节点插入到目标节点之后。

```
Monkey node = m1;

while(node != null) // 遍历每个节点
{
    if(node.id == 2) // 找到目标节点
    {
        // 附加到目标节点之后
        m5.next = node.next;

        node.next = m5;

        break;
    }

    node = node.next;
}
```

小结一下，相对于数组而言，链表有以下优点：

- 可以有无限多个节点，长度不限制
- 插入一个节点，并不需要挪动后面节点的位置。因此插入节点的效率较高。

15.4 有头链表

为了简化链表的相关算法，可以用一种特殊的设计来构造链表，即所谓的“有头链表”。有头链表，就是使用一个假节点来作为链表的头部，其他数据节点挂在假节点的后面。

如图 15.6 所示。



图 15.6 有头链表：有固定的头部节点

在此图中，第一节点不含有有效数据，它是一个假节点。后面 4 个节点都是有效的数据节点。按这种形式构造出来的链表，其插入和删除算法都比较简单。

15.4.1 有头链表的构造

需要创建一个假节点，不含真实数据的节点。例如，创建一只石猴作为头节点，

```
Monkey head = new Monkey(0, "石猴");
```

然后再把 4 个有效节点挂在假节点之后，

```
head.next = m1;  
m1.next = m2;  
m2.next = m3;  
m3.next = m4;  
m4.next = null;
```

如此，便构造了一个含有 5 个节点的链表。头节点是一只石猴，后面 4 个节点是真的猴子。

15.4.2 有头链表的遍历

在遍历时，不应包括头节点，因为头节点里不含有有效数据。代码如下，

```
Monkey m = head.next;  
while( m != null)  
{  
    // 处理这个节点  
    m = m.next;  
}
```

15.4.3 向有头链表里插入节点

最快的方法，是把新节点直接挂在头节点的后面。示例代码如下，

```
m5.next = head.next;  
head.next = m5;
```

插入⑤号节点之后，链表的逻辑结构如图 15.7 所示。



图 15.7 新节点插在头部位置

15.4.4 从有头链表中删除节点

例如，删除 2 号节点，

```
Monkey node = head;  
while( node.next != null )  
{  
    if( node.next.id == 2 ) // 找到目标节点  
    {  
        // 删除该节点  
        node.next = node.next.next;  
        break;  
    }  
    node = node.next;  
}
```

其中，这一行理解起来会有点难度，

`node.next = node.next.next;`

在删除之前，`node` 是①号节点，`node.next` 是②号节点，`node.next.next` 是③号节点。

在删除之后，node 是①号节点，node.next 是③号节点，所以就达到了删除②号节点的效果。

15.5 链表与容器

严格地说，链表还不是容器，它只是实现容器的一种方式。下面将演示怎么用链表这种数据结构来实现一个容器。如图 15.8 所示。

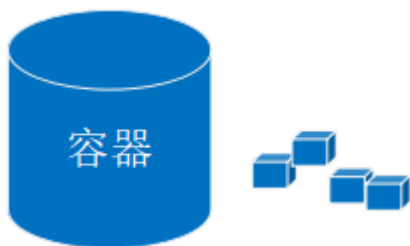


图 15.8 容器示意图

在第 15.1 节里已经讲过，所谓容器，就是一种可以存放对象的东西。创建一个容器之后，就可以把多个对象塞到容器里。如上图所示。

下面，我们将创建一个容器类 `MonkeyList`，用它来存储多个 `Monkey` 对象。该类的大致结构如下：

```
public class MonkeyList
{
    // 添加一个对象
    public void add ( Monkey m )
    {
    }

    // 按编号来查询
    public Monkey get ( int id)
    {
    }

    // 打印输出所有的对象
```

```
public void showAll()
{
}
}
```

在 `MonkeyList` 类里，我们将实现 3 个方法，`add()`方法用于向容器里放入一个 `Monkey` 对象，`get()`方法用于按编号查询，`showAll()`方法用于显示容器里的所有对象。

15.5.1 容器的实现

这样的容器，内部可以用数组结构来实现，也可以使用链表结构来实现。下面使用有头链表的设计来实现这个容器。

首先，添加一个假节点作为头节点。然后，实现 `add()`方法，把新增的节点挂在 `head` 节点的后面。

```
public class MonkeyList
{
    private Monkey head = new Monkey(0, "石猴");
    // 添加一个对象
    public void add ( Monkey m )
    {
        m.next = head.next;
        head.next = m;
    }
    public void showAll()
    {
        Monkey m = head.next;
        while( m != null)
        {
```



```

        System.out.println("容器中的对象: " + m);

        m = m.next;

    }

}

// 其他方法的实现代码省略，参考网盘源码

}

```

15.5.2 容器的使用

有了这个类之后，便可以按如下的方式来使用这个容器，

```

public static void main(String[] args)
{

    // 4 只猴子

    Monkey m1 = new Monkey(101, "圆圆");
    Monkey m2 = new Monkey(102, "方方");
    Monkey m3 = new Monkey(103, "角角");
    Monkey m4 = new Monkey(104, "朱朱");

    MonkeyList monkeys = new MonkeyList();

    monkeys.add( m1 );
    monkeys.add( m2 );
    monkeys.add( m3 );
    monkeys.add( m4 );

    monkeys.showAll();

}

```

此时，对调用者而言，**MonkeyList** 就是一个容器类，可以存放多个 **Monkey** 对象。

而且，这个容器的存储容量似乎是没有上限的，无论有多少只猴子，都可以塞进去。这就是链表这种结构的优点。

15.5.3 黑盒设计

黑盒 (Black Box)，是一种设计术语。所谓黑盒设计，就是在设计一个类或一个模块时，把内部想象成不透明的。

调用者在使用时，只需关心如何使用它，而不必关心（也不能关心）内部的实现。对于调用者而言，无论这个容器内部是丑是美、里面用的是链表还是数组，与我何干？

所以，在上述容器 `MonkeyList` 的实现里，既可以用链表结构来实现，也可以用数组容器来实现。对外部调用者来说，其 `add()`方法和 `showAll()`方法在使用起来感受不到内部的差异。

15.6 ArrayList

`ArrayList`，是 Java 自带的一个基础工具类。兼具 `Array` 和 `List` 的双重特点，可以称为数组链表。`ArrayList` 是一个容器，可以存储任何类型的对象。正如在上节课所说的，对于这种已经写好的容器类，我们首要关心的是如何使用它，而不是关心它的内部实现。

`ArrayList` 提供了丰富的方法操作，比如，`add()`方法可以添加对象，`remove()`方法可以删除对象，等等。

先看一个例子，学习一下它的使用。

比如，有 3 个学生对象，

```
Student s1 = new Student("20170001", "邵发");  
Student s2 = new Student("20170002", "小张");  
Student s3 = new Student("20170003", "小王");
```

现在创建一个容器对象，

```
ArrayList container = new ArrayList();
```

然后把 3 个学生对象添加到容器里，

```
container.add( s1 );  
  
container.add( s2 );  
  
container.add( s3 );
```

接下来，可以像数组一样遍历容器里的所有对象，

```
for(int i=0; i< container.size(); i++)  
{  
    Student s = (Student) container.get(i);  
    System.out.println("遍历得到: " + s );  
}
```

如果要删除某个对象，可以用 `remove()` 方法，示例代码如下，

```
container.remove( 1 );
```

在将来的工程应用中，我们一般不会自己来定义一个链表类，而是直接使用 Java 自带的各种现成的工具类，比如 `ArrayList`、`LinkedList` 类等。关于 `ArrayList` 的更多使用细节，在第 23 章中还有介绍。

第 16 章 学生管理系统

本章学习目标

- 完成一个基于控制台界面的简单管理系统

本章结合 ArrayList 和控制台界面技术，实现一个针对学生信息的管理程序。在此程序中，可以维护若干条学生的信息，并支持增加、列表、查询和删除等管理操作。

16.1 系统演示

这是一个基于控制台界面的程序，允许用户在控制台界面下输入一些命令，通过命令行的方式来管理这个系统。如图 16.1 所示。

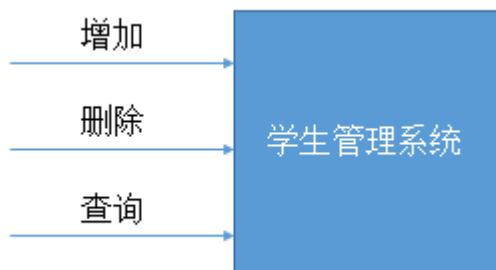


图 16.1 学生管理系统的示意图

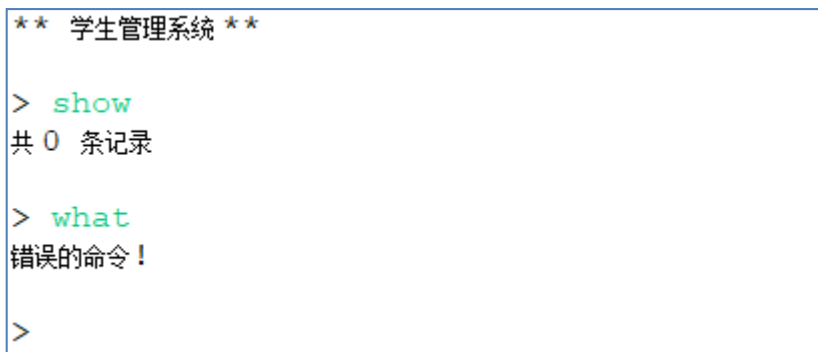
系统支持的命令有：

- add ， 添加一条记录
- show ， 显示所有的记录
- find ， 按姓名查找一条记录
- remove ， 按学号删除一条记录
- exit ， 退出系统

其中，每一条记录包含了一个学生类的信息，学生类 `Student` 的定义如下。

```
public class Student
{
    public int id; // 学号
    public String name; // 姓名
    public boolean sex; // 性别
    public String cellphone; // 手机号
}
```

来看一下系统的演示。当系统运行时，在控制台里会有欢迎提示。在光标闪烁处可以输入一条命令，按回车结束。如图 16.2 所示。



```
★★ 学生管理系统 ★★

> show
共 0 条记录

> what
错误的命令!

>
```

图 16.2 系统运行演示

其中，如果输入的命令正确，则执行这条命令并显示结果。例如，输入 `show` 命令时，将列出所有现有的学生记录。如果输入的命令不支持，则提示“错误的命令”。

当输入 `exit` 命令时，退出程序。如图 16.3 所示。

```
★★ 学生管理系统 ★★

> show
共 0 条记录

> what
错误的命令！

> exit
退出...
Exit
```

图 16.3 输入 exit 命令的演示

16.2 建立项目

下面开始实现这个项目。在 Eclipse 里新建一个 Java 项目，把工具类 AfConsole 加进来，并新建一个 Student 类。如图 16.4 所示。

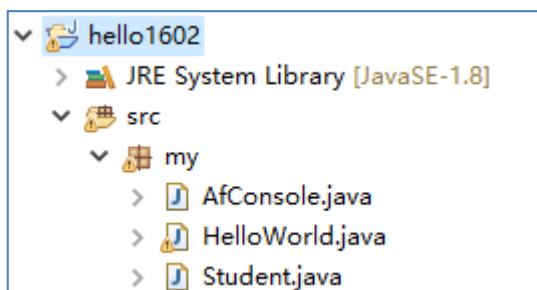


图 16.4 添加一个 Student 类

本项目是基于控制台的程序，所以需要 AfConsole 的支持。比如，让用户从控制台输入一个学生的信息，就需要用 AfConsole 从控制台读取输入。

下面在 HelloWorld 里添加一个方法，用于输入一个学生的信息。示例代码如下。

```
public Student getStudent(AfConsole cons)
{
    Student stu = new Student();

    cons.print("学号: ");
```

```

        stu.id = cons.readInt(0);

        cons.print("姓名: ");
        stu.name = cons.readString("");

        cons.print("性别(1/0): ");
        int nValue = cons.readInt(1);
        stu.sex = nValue > 0;    // 读取一个 int, 转成 boolean

        cons.print("手机号: ");
        stu.cellphone = cons.readString("");
        return stu;
    }

```

其中，用 `readInt()`来读取一个整数输入作为学号，`readString()`读取一个字符串输入作为姓名。

其中，性别字段是一个 `boolean` 类型，无法直接从控制台输入。这里采用的设计是，让用户输入 1 表示男性，0 表示女生。用 `readInt()`读取一个整数，再判断是 1 还是 0，从而转成 `boolean` 类型的值。

在 `main()`方法调用这个 `getStudent()`，示例代码如下。

```

public static void main(String[] args)
{
    HelloWorld world = new HelloWorld();
    AfConsole cons = new AfConsole();
    Student stu = world.getStudent(cons);
    System.out.println("输入了: " + stu);
    System.out.println("Exit");
}

```

运行这个程序，在控制台里输入学生的信息，如图 16.5 所示。

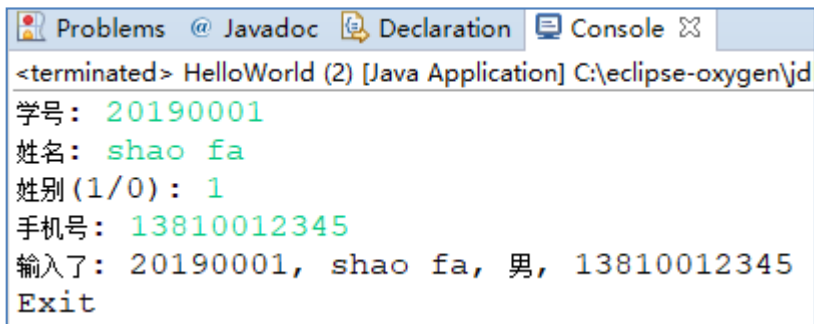


图 16.5 输入学生的信息

观察显示结果，可以发现 `getStudent()`能够成功读取用户的输入，并返回一个 `Student` 对象。

在输入姓名字段时，使用英文或拼音即可，因为在控制台下不方便使用中文输入。目前在语法学习阶段，重点是语法和逻辑，是否支持中文并不重要。后续课程中将引入图形界面编程，到那时候就可以自由地使用中文输入了。

16.3 命令行界面

下面建立命令行界面。所谓命令行界面，就是在控制台下不断地读取用户的输入。当用户输入一条命令并按下回车时，程序里解释执行这一条命令。执行完了之后，再次等待用户输入下一条命名。

在 `HelloWorld` 类里添加一个 `start()`方法，用于实现这种命令行输入的界面逻辑，示例代码如下。

```
public void start()
{
    AfConsole cons = new AfConsole();
    cons.println("** 学生管理系统 **");
    while ( true )
    {
        cons.print("\n> "); // 命令行提示符
        String cmd = cons.readString("");
```



```

        cmd = cmd.trim(); // 清空字符串左右两边的空白

        if(cmd.equals("exit"))
        {
            System.out.println("退出...");
            break;
        }
        else if(cmd.equals("add"))
        {
            Student stu = getStudent(cons);

            System.out.println("\n 添加了: " + stu );
        }
        else
        {
            System.out.println("错误的命令!");
        }
    }
}

```

其中，用一个 `while()` 循环来实现整个命令行框架。每次都是先用 `cmd = cons.readString("")` 来读取一行命令的输入。然后用 `if...else if...else` 来根据不同的命令作出相应的处理。如果命令为 `exit`，则退出循环。如果命令没有匹配到，则提示“错误的命令”。

在 `main` 方法里调用此方法，示例代码如下。

```

public static void main(String[] args)

```

```

{
    HelloWorld world = new HelloWorld();

    world.start();

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

```

其中，执行 `world.start()` 进入到上述的 `while` 循环中，不断读取命令行的输入并处理，直到用户输入 `exit` 时退出循环。

运行这个程序，试着输入一些命令，最后输入 `exit` 退出程序。如图 16.6 所示。

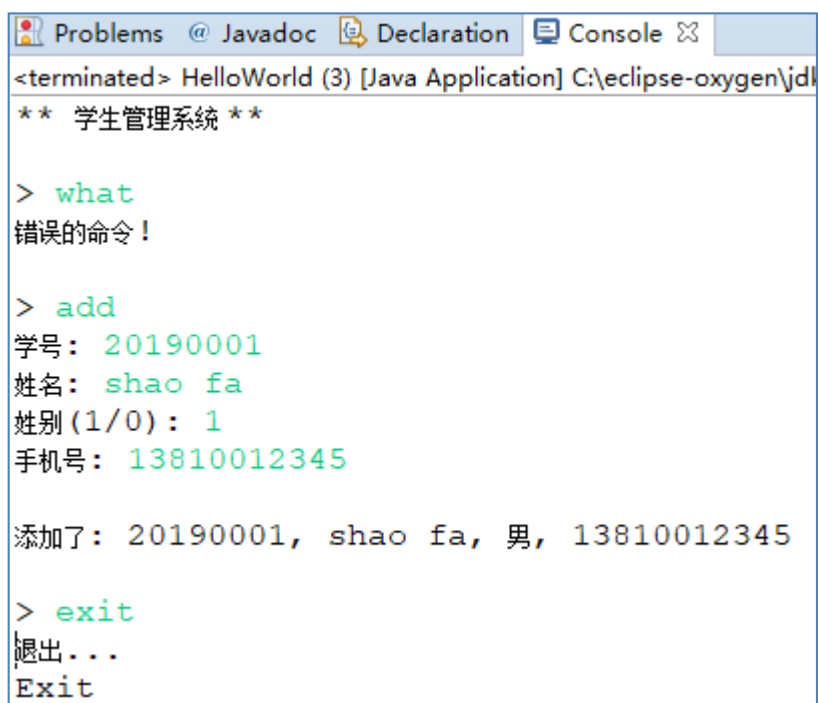


图 16.6 输出若干命令的演示

其中，输入 `what` 命令，此命令不被支持，提示“错误的命令”。输入 `add` 命令，提示输入一个学生的信息。输入 `exit` 命令，退出了程序。

16.4 数据记录的管理

由于系统中将存储多条学生的记录，所以需要有一个类来管理这些 `Student` 对

象。建立一个新的类 `StudentDB`，由它负责管理 `Student` 对象。示例代码如下。

```
public class StudentDB
{
    public ArrayList list = new ArrayList();

    // 添加一条记录
    public void add(Student s)
    {
        list.add( s );
    }

    // 按学号删除一条记录
    public void remove(int id)
    {
        for(int i =0; i<list.size();i++)
        {
            Student s = (Student)list.get(i);
            if(s.id == id )
            {
                list.remove( i );
                break;
            }
        }
    }
}
```

其中，使用一个 `ArrayList` 来存储所有 `Student` 对象。当用户输入一个学生的信息时，将新创建的 `Student` 对象添加到这个 `StudentDB.list` 中。当执行删除操作时，从 `StudentDB.list` 中删除相应的记录。

然后，修改命令行循环，在里面添加一个新的命令 **show**。当用户输入 **show** 命令时，将打印显示系统中所有的学生记录。示例代码如下。

```
AfConsole cons = new AfConsole();

StudentDB db = new StudentDB();

while ( true )

{

    ..... // 省略显示

    else if(cmd.equals("show"))

    {

        System.out.println("共 " + db.list.size()

                            + " 条记录");

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

        {

            Student stu = (Student) db.list.get(i);

            System.out.println(stu);

        }

    }

    ..... //

}
```

再修改原 **add** 命令的实现，当用户输入 **add** 命令时，从控制台里读取一个学生信息的输入，将新创建的 **Student** 对象添加到 **StudentDB.list** 中。示例代码如下。

```
else if(cmd.equals("add"))

{

    Student stu = getStudent(cons);

    System.out.println("\n 添加了: " + stu );

    db.add(stu);

    System.out.println("共 " + db.list.size())
```

```
        + " 条记录");  
    }  
}
```

运行整个程序，测试一下 add 命令和 show 命令的功能。输入 add 命令，添加一个学生的信息。如图 16.7 所示。

```
** 学生管理系统 **  
  
> add  
学号: 20190001  
姓名: shao fa  
性别(1/0): 1  
手机号: 13810012345  
  
添加了: 20190001, shao fa, 男, 13810012345  
共 1 条记录
```

图 16.7 输入 add 命令

命令执行完毕，系统显示当前“共 1 条记录”。继续调用 add 命令，再添加一个学生的信息。如图 16.8 所示。

```
> add  
学号: 20190002  
姓名: li ming  
性别(1/0): 1  
手机号: 18600011122  
  
添加了: 20190002, li ming, 男, 18600011122  
共 2 条记录
```

图 16.8 再输入一个学生记录

命令执行完毕，系统显示当前“共 2 条记录”。然后调用 show 命令查看所有的学生记录。如图 16.9 所示。

```
> show  
共 2 条记录  
20190001, shao fa, 男, 13810012345  
20190002, li ming, 男, 18600011122
```

图 16.9 输入 show 命令显示所有记录

16.5 查找与删除

下面来实现查找与删除功能。添加 `find` 命令用于按姓名查找一条记录，`remove` 命令用于按学号删除一条记录。

首先，修改 `StudentDB` 类，在里面添加 `find` 和 `remove` 方法。先看一个 `StudentDB.remove()`方法的实现，示例代码如下。

```
public void remove(int id)
{
    for(int i =0; i<list.size();i++)
    {
        Student s = (Student)list.get(i);
        if(s.id == id )
        {
            list.remove( i );
            break;
        }
    }
}
```

其中，参数 `id` 表示要移除的学生的学号。通过遍历整个 `list`，逐个比较学号是否匹配，如果匹配则删除这条记录。

再看一下 `StudentDB.find()`方法的实现，示例代码如下。

```
public ArrayList find(String name)
{
    ArrayList result = new ArrayList();
    for(int i =0; i<list.size();i++)
    {
```

```

        Student s = (Student)list.get(i);

        if(s.name.indexOf(name) >= 0)

        {

            result.add( s );

        }

    }

    return result;

}

```

其中，创建了一个 **result** 列表，遍历整个 **list** 并将符合条件的记录添加到 **result** 列表里返回。在遍历匹配时，根据学生姓名进行匹配，只要姓名中含有查询参数子串，则认为匹配。比如，查找含有"shao"的记录，则学生"shao fa" 和 "shao anxin" 均符合条件。

其中，**String.indexOf()** 的用法在 14.1 里有介绍。当含有目标子串时，返回子串的位置索引。当不含目标子串时，返回-1。

在这里，**find()**方法的返回值是一个 **ArrayList** 对象。这意味任何类型的对象都可以作为方法的返回值类型。在 **find()**里，因为要返回多条 **Student** 的记录，所以使用一个 **ArrayList** 作为返回值。

下面，修改命令行循环，添加 **remove** 和 **find** 命令的支持。示例代码如下。

```

        else if(cmd.equals("find"))
        {

            cons.print("输入要查找的姓名: ");

            String name = cons.readString("");

            name = name.trim(); // 去除空白

            ArrayList result = db.find( name );

```

```

        System.out.println("匹配到 " + result.size() +
" 条记录");

        for(int i=0; i<result.size(); i++)
        {
            Student stu = (Student) result.get(i);
            System.out.println(stu);
        }
    }

    else if(cmd.equals("remove"))
    {
        cons.print("输入删除的学号: ");
        int id = cons.readInt(0);

        if(id > 0)
        {
            db.remove( id );

            System.out.println("已删除学生: ID=" + id);
        }
    }
}

```

运行现在的程序，测试 `find` 和 `remove` 命令的功能。当然，开始还是需要使用 `add` 方法先添加几条测试记录，待系统中有几条记录之后，才能执行所谓的查找和删除操作。

调用 `add` 命令，向系统中添加几条学生记录。此过程截图省略。

调用 `show` 命令，显示当前系统中的所有记录。如图 16.10 所示。

```

> show
共 3 条记录
20190001, shao fa, 男, 13810012345
20190002, li ming, 男, 13600011122
20190003, shao anxin, 女, 18833336666

```


图 16.10 显示所有记录

调用 find 命令，查找姓氏为 shao 的学生。如图 16.11 所示。

```
> find
输入要查找的姓名: shao
匹配到 2 条记录
20190001, shao fa, 男, 13810012345
20190003, shao anxin, 女, 18833336666
```

图 16.11 输入 find 按名称查找

调用 remove 命令，删除学号为 20190002 的学生。如图 16.12 所示。

```
> remove
输入删除的学号: 20190002
已删除学生: ID=20190002
```

图 16.12 输入 remove 命令删除

到此为止，系统已经支持 add, show, find, remove, exit 五条命令，完成系统的演示。目前这个程序还比较简单，随着技术的深入，后面还会使用图形界面技术、甚至数据库技术，来创建更复杂的具体实际意义的项目。

全 27 章，已完结！

本电子书共 27 章，完整版在此下载：

<http://afshare.net/resource/view/200321R1.html>

官网： <http://afanihao.cn>

作者： 邵发

官方 QQ 群： 495734195

本系列教程由 24 篇以上视频教程组成，从入门语法到行业级技术，循序渐近式的全方位教程。内容包含入门语法和高级语法，覆盖 Java 在业界的 3 个应用领域（网站开发、安卓 APP 开发、桌面 GUI 开发）。同时包含专项技术的培训教程，如网络编程基础、数据库开发，FreeMarker, Spring, MyBatis 等。