



JAVA基础教程

基于C语言

© 2017-2020,宿宝臣



subaochen@126.com

<http://dz.sdu.edu.cn/blog/subaochen>

<https://github.com/subaochen/java-tutorial>

前言

本书应该是国内第一本开源 (open source) 的 Java8+ 教材，是 Java 系列教程的第一步：初级篇。您可以在<https://github.com/subaochen/java-tutorial>获得本书的所有源代码，包括本书的 lyx 源码、示例代码和图片等。

本书作为为数不多的涉及 Java8+ 的教材，期望能够帮助读者在更高的平台上（本书放弃了 Java5 之前的部分特性）尽快掌握 Java 语言及其编程技能。本书的目标是在尽量短的时间内，比如 48 个学时，帮助 Java 初学者掌握基本的 Java 语法和编程思路。

一般的，理工科院校讲授的第一门计算机程序设计语言是 C 语言，因此大家在学习（自学或者教学）Java 语言时已经具有了一定的 C 语言基础。而目前常见的 Java 语言教材或者辅导材料基本上都是零基础开始讲授 Java 语言程序设计的，完全无视大家的 C 语言基础，导致教与学活动中的很大浪费，也不利于突出 Java 教学活动的重点。因此，本书不是一本零基础的 Java 教程，本书在很多章节将 C 语言和 Java 做了对比，引导读者从 C 语言转换到 Java 语言上来，作者期望读者在 C 语言的基础上能够实现 Java 语言的快速入门。

本书的特点

本书注重培养良好的思维习惯、编程风格，而不是简单的知识传授。很多教材的示例代码过于随意，可能是作者觉得示例代码过于短小，不值得精心雕琢吧。区别于大多数教材的是，本书在编写示例代码时，全部经过作者的一手调试，并尽力保证类名、变量名等命名习惯以及编程风格（包括注释风格）接近工业标准，以帮助读者“先入为主”的了解正确的 Java 编程姿态。如果我们学习 Java 的目的是对接工业化应用，作者认为这样可以事半功倍。

本书注重从实战中学习 Java 的知识点，本书尽力做到每一个概念、每一个知识点都是可以验证的，书中的每一个例子都是完整可运行的，都经过作者的亲自调试。本书的所有示例均可以从<https://github.com/subaochen/java-tutorial> 下载，作者也会根据读者的反馈不断完善和调整示例程序。

本书通过思维导图的方式绘出了 Java 的各个知识点，并针对每个知识点设计了练

习题，帮助读者更好的了解这些知识点及其在实际中的应用方式。

本书的例子均遵循 google 的编码规范（参考附录），希望读者在阅读本书示例代码的同时能够直观的感受 google 编码规范并逐步养成遵守编码规范的好习惯。

如何阅读本书

本书不强调从一开始就掌握 Java 的每个技术细节，而是强调“先跑起来！”，即首先动手写出一个可以运行的 Java 应用程序，并理解这个应用程序为什么能够跑起来，然后再逐步扩充到细节问题。也就是说，在实践中体味和掌握技术细节，在实践中逐步形成良好的编程风格和思维习惯。

作者建议的阅读方式是：尽快、尽早的下载本书配套的示例代码，在阅读到例题的时候，直接用 Idea 打开相应的项目即可查看和运行。同时，建议不要拘泥于作者提供的示例代码，读者完全可以也应该不断尝试修改并测试。编程能力就是在不断尝试中发展起来的。

本书也特别强调循序渐进的学习路线和知识的前后衔接以及连贯性。虽然 Java 语言经常存在一个概念贯穿前后的情况，但是本书强调先掌握这个概念的部分特征，随着学习的深入，将在后续章节逐步展开，并给出恰当的交叉索引将知识点逐步串联起来。

本书的章节标题如果是 * 开头的为较高要求内容，读者可以有选择的阅读，或者作为进阶的阅读材料。

你适合阅读本书吗

本书不是一本零基础的 Java 语言教程。本书假设读者已经有了一些 C 语言的基础，掌握了 C 语言的基本语法。另外，本书在很多方面比较深入的探讨了 Java 编程的理念和最佳实践，也可以作为工程技术人员学习 Java 语言或者进一步理解 Java 语言的材料。

本书的体例

印刷约定

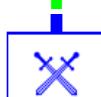
| 字体 | 意义 | 示例 |
|-------------------|-------------|--------------------------------------|
| <i>AbCd123</i> 斜体 | 文件名、路径名、域名等 | <code>ls -l filename</code> |
| AbCd123 加粗 | 在终端输入的命令等 | <code>subaochen_desktop% su</code> |
| 等宽字体 | 示例代码、代码片段等 | <code>public class MyClass...</code> |

图形标识

本书使用了如下的图形标识帮助读者更好的区分和了解知识点所在：



需要注意的知识点。



工程实践中常用的技巧。



容易出错的地方，需要特别小心。

联系作者

您可以通过我的博客获得最新的消息：<http://dz.sdu.edu.cn/blog/subaochen>，或者给我发 Email：subaochen@126.com。本书中的示例代码可以从<http://github.com/subaochen/java-tutorial> 下载，也欢迎读者不吝指教，在 github 提交 PR，或者直接发邮件讨论亦可。您可以在本书的不同章节看到如何获得源代码的相关提示。

本书是如何写成的

本书全部使用开源（Open Source）软件完成：

- Linux，本书所有稿件和源代码均在 Linux 下完成。
- git，本书的写作过程全程通过 git 进行版本控制，也借助于 git 在办公室、家和旅途中实现文档的同步，收益良多。
- Lyx(<http://www.lyx.org>)，优秀的 Latex 前端可视化工具，最新版本（本书写作时是 2.2）配合 xetex、CTeX 可以很好的支持中文处理。
- graphviz，灵活而强大的代码绘图工具，本书部分流程图是使用 graphviz 绘制的。
- Shutter(<http://shutter-project.org>)，Linux 下面优秀的截图工具。
- ArgoUML(<http://argouml.tigris.org>)，优秀的开源 UML 工具。
- umbrello(<https://umbrello.kde.org>)，优秀的开源 UML 工具，本书部分 UML 图是使用 umbrello 绘制的。

- **Dia**, 优秀的开源绘制工具, 堪称 Linux 下面写作绘图的“瑞士军刀”, 本书的大部分流程图、框图和 UML 图都是用 Dia 绘制的。
- **Inkscape**, 优秀的开源矢量图绘制工具, 本书的大部分矢量图是用 Inkscape 绘制的。
- **vym(<http://www.insilmaril.de/vym/>)**, 思维导图绘制工具, 本书所有思维导图都是使用 vym 绘制的。

关于版权

本书遵循 Apache Licence, 您可以在遵循 Apache Licence 和尊重原作者的前提下, 在自己的博客、电子书等以电子文档的形式引用或者使用本书的内容。

但在本书正式出版前, 作者以及本书的其他贡献者一致要求, 您不能将本书内容用于正式出版物, 包括但不限于本书的文字、图片和示例代码。

致谢

感谢山东理工大学电气与电子学院对本书的资助, 感谢各位同仁给予的帮助和指点, 感谢爱妻程玉华承担了大部分家务, 让我有充足的时间专心写作; 感谢实验室的小伙伴们, 尤其是胡安禄同学和徐千惠同学, 帮助画出了部分示例图; 感谢李松阳同学在繁忙的工作之余认真校对, 从标点符号到遣词造句都给出了很多具体的意见和建议。如果您对本书的写作有所贡献, 也会在这里一并感谢。如果有较大的贡献, 本书正式出版时将作为共同作者。期待更多的朋友加入进来!

特别的, 要感谢我的女儿宿佳敏。写作本书时她正在读高中, 作者答应在这个寒假结束时写完这本书送给她。我做到了, 信守一个父亲的诺言。作者坚信, 这是最长情的陪伴和对她学习的最好帮助。

在写作本书过程中, 作者查阅了大量 Java 教材和网络资料, 所引用或依据的精彩论述和案例尽量在文中标出, 以便读者参照和比较, 同时对原作者表示深深的敬意和感谢!

也感谢所有为开源软件作出贡献的人们! 二十年前, 当年懵懂的作者决定彻底拥抱开源软件时, 为了安装一个 Linux 系统曾经不眠不休两天两夜; 回头望, 开源软件蓬勃发展二十年, 值得欣慰, 值得弹冠相庆! 没有 Linux, 没有 Latex, 没有 Lyx, 没有 Dia, 这本书也不可能如此顺利的完稿。致敬, Open Source!

宿宝臣
2017 年 2 月
于山东理工大学 1 号实验楼

目录

| | |
|---|-----------|
| 前言 | i |
| 第一章 初识 Java | 8 |
| 1.1 Java 在程序设计语言中的霸主地位 | 8 |
| 1.2 Java 简史 | 8 |
| 1.2.1 Java 的前身：Oak 的诞生 | 9 |
| 1.2.2 Java 拥抱互联网，横空出世 | 10 |
| 1.2.3 Sun 公司没落，Java 易手 Oracle | 11 |
| 1.3 Java 的特点 | 11 |
| 1.3.1 面向对象 | 11 |
| 1.3.2 跨平台 | 12 |
| 1.3.3 自动垃圾回收 | 13 |
| 1.4 Java 开发环境的搭建 | 14 |
| 1.4.1 Java 版本的选择 | 14 |
| 1.4.2 JDK 的概念 | 14 |
| 1.4.3 基础开发环境 | 15 |
| 1.4.4 IDE 开发环境 | 16 |
| 1.5 初识 Java 的输入输出 | 22 |
| 1.5.1 输出到屏幕 | 22 |
| 1.5.2 从键盘输入 | 23 |
| 第二章 Java 的基础语法 | 26 |
| 2.1 变量和变量的命名 | 26 |
| 2.2 常量和常量的命名 | 27 |
| 2.3 简单数据类型 | 27 |
| 2.4 运算符 | 30 |
| 2.5 表达式 | 30 |
| 2.6 条件判断和分支 | 31 |
| 2.6.1 条件判断 | 31 |

| | |
|---------------------------------------|-----------|
| 2.6.2 Java 加强了的 switch 分支结构 | 32 |
| 2.7 循环 | 34 |
| 2.7.1 三大循环结构 | 34 |
| 2.7.2 Java 加强了的循环结构 | 34 |
| 2.8 方法的不定长参数 | 34 |
| 2.9 Java 的注释 | 34 |
| 2.10 综合应用举例 | 36 |
| 第三章 面向对象编程基础 | 38 |
| 3.1 再说 C 语言的 struct | 38 |
| 3.1.1 struct 的起源 | 38 |
| 3.1.2 struct 的局限性 | 40 |
| 3.2 类和对象的初步概念 | 40 |
| 3.2.1 类是 struct 概念的自然延伸 | 40 |
| 3.2.2 定义 Java 类 | 42 |
| 3.2.3 对象的概念 | 42 |
| 3.3 对象的创建过程 | 46 |
| 3.3.1 构造方法 | 46 |
| 3.3.2 默认的构造方法 | 47 |
| 3.3.3 带参数的构造方法 | 47 |
| 3.3.4 多个构造方法 | 50 |
| 3.3.5 再说对象的初始化 | 54 |
| 3.4 类的组织：包 | 55 |
| 3.4.1 包的概念 | 55 |
| 3.4.2 包的导入和声明 | 56 |
| 3.4.3 包的命名原则 | 57 |
| 3.5 类的继承 | 57 |
| 3.5.1 单继承 | 59 |
| 3.5.2 再说构造方法 | 59 |
| 3.5.3 方法的覆盖和重载 | 62 |
| 3.6 访问控制 | 68 |
| 3.6.1 属性和方法的访问控制 | 68 |
| 3.6.2 类的访问控制 | 77 |
| 3.6.3 内部类 | 77 |
| 3.7 引用类型 | 79 |
| 3.7.1 this | 83 |
| 3.7.2 super | 84 |

| | |
|--|------------|
| 3.8 static | 86 |
| 3.8.1 static 常量 | 87 |
| 3.8.2 static 属性 | 90 |
| 3.8.3 static 方法 | 93 |
| 3.8.4 * 内部类回头看 | 93 |
| 3.8.5 * 内部类的进一步讨论 | 99 |
| 第四章 面向对象工程思想 | 104 |
| 4.1 抽象类和抽象方法 | 104 |
| 4.1.1 抽象类 | 105 |
| 4.1.2 抽象方法 | 107 |
| 4.1.3 抽象类小结 | 108 |
| 4.2 接口 | 108 |
| 4.2.1 定义接口：纯的抽象类 | 108 |
| 4.2.2 实现接口 | 110 |
| 4.2.3 实现多个接口 | 110 |
| 4.2.4 不懂接口的项目经理不是好的项目经理 | 114 |
| 4.2.5 接口上的匿名内部类 | 116 |
| 4.3 多态 | 116 |
| 4.3.1 对象的存储模型 | 121 |
| 4.3.2 方法重载时的情形 | 122 |
| 4.3.3 多态的应用场合 | 123 |
| 4.4 * 向上塑型：面向接口的编程 | 123 |
| 4.4.1 对象的类型 | 123 |
| 第五章 Java 的常用类 | 125 |
| 5.1 字符串类 | 125 |
| 5.1.1 字符串对象的定义和初始化 | 125 |
| 5.1.2 常见字符串处理方法 | 127 |
| 5.2 数字类 | 131 |
| 5.2.1 基本数字类型变量的包裹类 | 131 |
| 5.2.2 不同数字类型之间的转换 | 134 |
| 5.2.3 *BigDecimal | 137 |
| 5.3 日期和时间类 | 141 |
| 5.3.1 日期操作类：LocalDate | 141 |
| 5.3.2 时间操作类：LocalTime | 144 |
| 5.3.3 日期时间类：LocalDateTime | 146 |
| 5.3.4 日期和时间的格式化输出类：DateTimeFormatter | 147 |

| | |
|---|------------|
| 5.3.5 日期和时间的调整 | 150 |
| 第六章 异常处理 | 153 |
| 6.1 异常的概念 | 153 |
| 6.2 C 中的异常处理方式 | 155 |
| 6.3 Java 的异常处理方式 | 155 |
| 6.4 捕获异常 | 156 |
| 6.5 抛出异常 | 161 |
| 6.6 用户自定义异常 | 164 |
| 6.7 finally | 167 |
| 6.8 try with resources ¹ | 167 |
| 第七章 Java 的 IO | 170 |
| 7.1 C 的 IO 回顾 | 170 |
| 7.2 Java 的 IO 体系 | 170 |
| 7.2.1 面向字节的流 | 172 |
| 7.2.2 面向字符的流 | 182 |
| 7.3 从键盘输入数据 ²³ | 182 |
| 7.4 Scanner 类 | 189 |
| 7.5 文件操作 ⁴ | 191 |
| 7.5.1 什么是 Path? | 191 |
| 7.5.2 相对路径和绝对路径 | 192 |
| 7.5.3 Path 类 | 192 |
| 7.6 Files 类 | 198 |
| 7.6.1 检查文件或者目录 | 200 |
| 7.6.2 删除文件或者目录 | 201 |
| 7.6.3 复制文件或者目录 | 201 |
| 7.6.4 移动文件或者目录 | 203 |
| 7.6.5 操作文件或者目录的属性 | 204 |
| 7.6.6 创建、读写文件 | 209 |
| 7.6.7 创建文件和临时文件 | 219 |
| 7.6.8 随机读写文件 | 221 |
| 7.6.9 目录操作 | 226 |
| 7.6.10 遍历目录：FileVisitor 接口 | 229 |

¹本部分示例代码参见：<http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

²借鉴了《Java 程序设计》（谌卫军）的案例，感谢作者谌卫军的精彩阐述！

³本节的完整测试代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/io/src/cn/edu/sdut/softlab/SystemInTest.java>

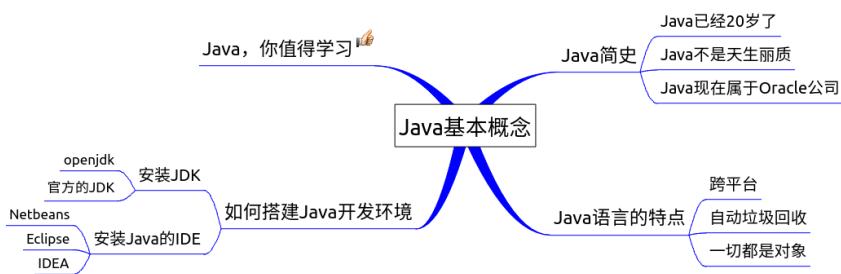
⁴JDK 1.7 引入了新的文件操作 API，即 NIO.2，本节内容着重于 NIO.2，不再涉及旧的文件 API。

| | |
|---------------------------------------|------------|
| 7.7 访问属性文件 | 231 |
| 第八章 Enum | 235 |
| 8.1 C 语言中的 enum 数据类型 | 235 |
| 8.2 Java 的 enum: Enum 的子类 | 235 |
| 8.3 enum 的两个重要属性 | 238 |
| 8.4 enum 的构造方法 | 238 |
| 8.5 switch 中的 enum | 240 |
| 第九章 图形用户界面设计 | 242 |
| 9.1 Java 的图形用户界面 (GUI) 设计概述 | 242 |
| 9.2 Swing 入门 | 243 |
| 9.3 GUI 的顶级容器类 | 255 |
| 9.4 Swing 控件 | 256 |
| 9.4.1 JLabel | 256 |
| 9.4.2 文本控件 | 258 |
| 9.4.3 JButton | 261 |
| 9.4.4 JCheckBox | 263 |
| 9.4.5 JRadioButton | 264 |
| 9.4.6 JComboBox | 266 |
| 9.4.7 Spinner | 266 |
| 9.4.8 Slider | 269 |
| 9.4.9 Tabbed Pane | 269 |
| 9.5 布局管理器 | 269 |
| 9.5.1 BorderLayout | 269 |
| 9.5.2 FlowLayout | 270 |
| 9.5.3 CardLayout | 272 |
| 9.5.4 GridLayout | 273 |
| 9.5.5 GridBagLayout | 275 |
| 9.6 JAVA 图形用户界面的事件机制 | 278 |
| 9.6.1 事件机制的基本原理 | 278 |
| 9.6.2 监听器类的几种情形 | 278 |
| 9.7 Java GUI 综合应用举例 | 281 |
| 第十章 实验 | 286 |
| 附录 A 建议的授课计划 | 320 |

| | |
|--|------------|
| 附录 B Java FAQ | 325 |
| B.1 如何设置命令行参数? | 325 |
| 附录 C Java GUI 控件和事件监听器对照表 | 326 |
| 附录 D JDK 1.5、1.6、1.7、1.8 的差别 | 327 |
| 附录 E 部分练习和思考题参考答案 | 349 |
| E.1 初识 Java | 349 |
| E.2 面向对象编程基础 | 349 |
| E.3 ?? | 349 |
| E.4 Enum | 350 |
| 附录 F Jar 包和 CLASSPATH | 352 |
| F.1 jar 包概念和用法 | 352 |
| F.2 CLASSPATH | 352 |
| F.2.1 CLASSPATH 的默认值 | 352 |
| F.3 war/ear 包 | 355 |
| 附录 G Idea 的常用快捷键 | 356 |
| 附录 H Google 编码规范 | 357 |
| H.1 介绍 | 357 |
| H.1.1 术语说明 | 357 |
| H.1.2 1.2 文档说明 | 357 |
| H.2 源码文件基础 | 357 |
| H.2.1 文件名 | 357 |
| H.2.2 文件编码: UTF-8 | 358 |
| H.2.3 特殊字符 | 358 |
| H.3 源码文件结构 | 359 |
| H.3.1 lincense 或者 copyright 的声明信息。 | 359 |
| H.3.2 包声明 | 359 |
| H.3.3 import 语句 | 359 |
| H.3.4 类声明 | 360 |
| H.4 格式 | 360 |
| H.4.1 花括号 | 360 |
| H.4.2 语句块的缩进: 2 空格 | 361 |
| H.4.3 一行最多只有一句代码 | 362 |
| H.4.4 行长度限制: 80 或 100 | 362 |

| | |
|-------------------------------------|-----|
| H.4.5 长行断行 | 362 |
| H.4.6 空白空间 | 363 |
| H.4.7 分组括号：建议使用 | 364 |
| H.4.8 特殊结构 | 364 |
| H.5 命名 | 367 |
| H.5.1 适用于所有命名标识符的通用规范 | 367 |
| H.5.2 不同类型的标示符规范 | 368 |
| H.5.3 Camel case 的定义 | 369 |
| H.6 编程实践 | 370 |
| H.6.1 @override 都应该使用 | 370 |
| H.6.2 异常捕获不应该被忽略 | 370 |
| H.6.3 静态成员的访问：应该通过类，而不是对象 | 371 |
| H.6.4 不使用 Finalizers 方法 | 371 |
| H.7 Javadoc | 371 |
| H.7.1 格式规范 | 371 |
| H.7.2 摘要片段 | 372 |
| H.7.3 何处应该使用 Javadoc | 372 |

第一章 初识 Java



1.1 Java 在程序设计语言中的霸主地位

自从发明了计算机，程序设计语言就成了一门专门的技艺，至今人类已经发明了上千种程序设计语言 [1]。从权威统计机构 TIOBE¹的最新统计结果可以看出，Java 程序设计语言在最近的十五年几乎牢牢的占据了头名座椅。此后的 10 年、20 年，Java 的霸主地位会动摇吗？根据 ruby 之父松本行弘的自述，ruby 从开始开发到发布用了 3 年左右的时间，在程序员圈子中拥有一定的知名度则又花了 4 年的时间，再到通过 Ruby on Rails 而走红，则又花了 5 年的时间 [2, p109]。可见，程序设计语言的发展和进化是相对缓慢的一个过程，因此可以推断，至少未来 10 年的时间，Java 和 C 仍然会是主流的程序设计语言。因此，笔者认为学好 Java 是 IT 职业生涯的不二敲门砖。

1.2 Java 简史

Java 程序设计语言是不是天生丽质，一开始就大红大紫呢？

让我们穿越到 1990 年，了解一下 Java 的前世和今世，看一看它当年是如何降临人间的 [3]。

¹<http://www.tiobe.com/>

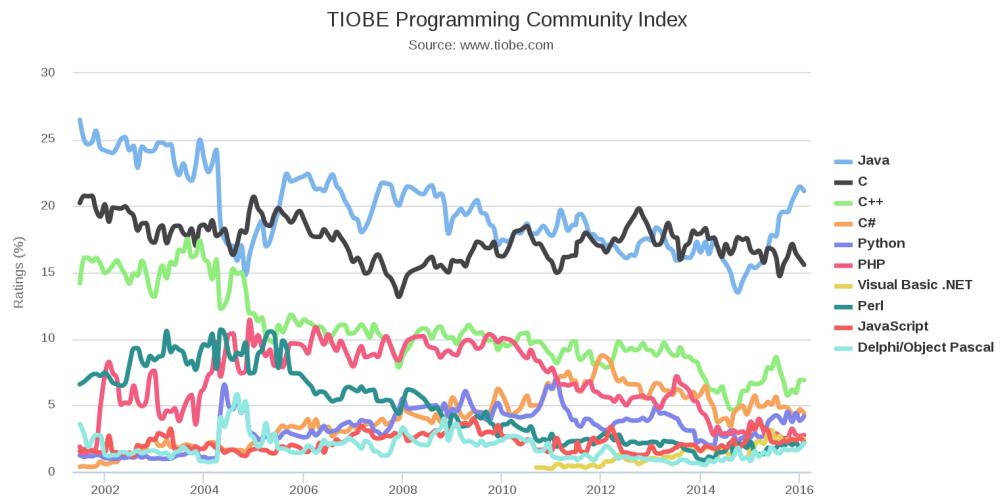


图 1.1: 最近 15 年 top 10 程序设计语言

1.2.1 Java 的前生: Oak 的诞生

1990 年 12 月,当时的 Sun 公司开始了一个叫做“Stealth 计划”的内部项目,由帕特里克·诺顿领衔,起因是公司自己开发的 C++ 和 C 语言编译器很难用,帕特里克被搞的焦头烂额。“Stealth 计划”后来改名为“Green 计划”,詹姆斯·高斯林(人称“Java 之父”,见图1.3)和麦克·舍林丹加入到了帕克里克的工作小组。Sun 公司预计未来的家用电器必将向智能化方向发展,于是“Green 计划”的目标是打造一款智能家电程序设计语言及其平台。

团队最初考虑使用 C++ 语言,但是很多成员包括 Sun 的首席科学家比尔·乔伊,发现 C++ 在嵌入式系统上存在很大问题。因为嵌入式系统可以使用的资源有限,而 C++ 缺乏垃圾回收机制以及可移植的安全性、分布式程序设计和多线程等特性。于是,乔伊决定开发一种集 C 语言和 Mesa 语言大成的新语言。在一份报告上,乔伊把它叫做“未来”。他提议 Sun 公司的工程师应该在 C++ 的基础上,开发一种面向对象的程序设计语言。最初,高斯林试图修改和扩展 C++ 的功能,他自己称这种新语言为 C++ ++ : 多么糟糕的命名啊,高斯林自己后来也放弃了这个名字。直到某一天,高斯林把这种新的语言命名为“Oak”(橡树),这是他办公室外的一棵橡树。

就像很多开发新技术的秘密工程一样,Green 团队没日没夜的工作到了 1992 的夏天,终于能够演示新平台的一部分了,包括 Green 操作系统、Oak 程序设计语言、类库及其硬件。最初的尝试是面向一种类 PDA 设备,被命名为 Star7,这种设备在当时由鲜艳的图形界面和被称为“Duke”(Duke 后来称为 Java 的吉祥物,参见图1.2)的智能代理来帮助用户。

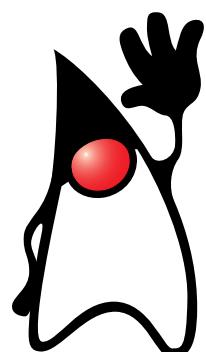


图 1.2: Java 的吉祥物

1992年12月3日，这台Star7设备进行了展示。

同年11月，Green计划的成果转化成为“FirstPerson有限公司”，一个Sun公司的全资子公司。FirstPerson团队对于打造一种高度互动的设备感兴趣，当时华纳发布了一个电视机顶盒的征求提议书时，FirstPerson决定响应时代华纳的提议，提出了一个机顶盒平台方案。但是当时的有线电视界觉得FirstPerson的平台给予了用户过多的控制权，于是最终FirstPerson输给了SGI。FirstPerson与3DO公司的另外一笔关于机顶盒的交易也没有成功，也就是说，FirstPerson公司在电视工业没有取得任何效益，Sun公司不得不收回了FirstPerson公司。幸运的是，Oak语言的命运并没有就此止步。

1.2.2 Java 拥抱互联网，横空出世

1994年6月，在经历了一场历时三天的脑力激荡的讨论后，约翰·盖吉、詹姆斯·高斯林、比尔·乔伊、帕特里克·诺顿、韦恩·罗斯因和埃里克·斯库米，

团队决定再一次做出重大决定，这次他们决定将该技术应用于万维网。他们认为随着Mosaic浏览器²的到来，因特网正在向同样的高度互动的远景演变，而这一远景正是他们在有线电视网中看到的。作为原型，帕特里克·诺顿写了一个小型万维网浏览器，WebRunner，后来改名为HotJava。

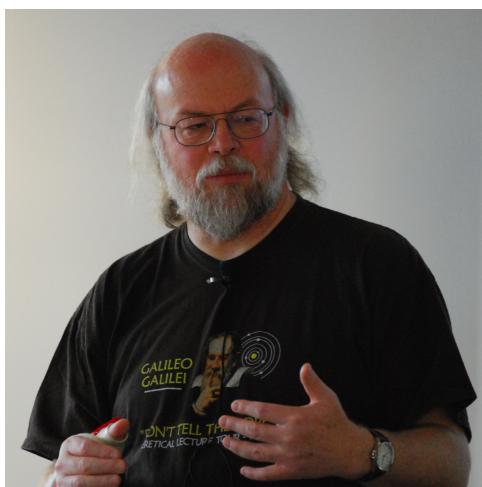


图 1.3: 詹姆斯·高斯林

1994年10月，HotJava和Java平台为公司高层进行演示。1994年，Java 1.0a版本已经可以提供下载，但是Java和HotJava浏览器的第一次公开发布却是在1995年3月23日SunWorld大会上进行的。Sun公司的科学指导约翰·盖吉宣告了Java技术的诞生。这个发布

是与网景公司的执行副总裁马克·安德森的惊人发布一起进行的：网景宣布将在其浏览器中包含对Java的支持，这在当时引起了很大的轰动。1996年1月，Sun公司成立了Java业务集团，专门开发Java技术。这一次，詹姆斯·高斯林等终于扬眉吐气了，Java此后的发展虽然也不是一帆风顺，但是到今天，Java可谓如日中天，詹姆斯·高斯林也被称为“Java之父”，图1.3为詹姆斯·高斯林本尊。

²Mosaic是浏览器的鼻祖，参见[https://en.wikipedia.org/wiki/Mosaic_\(web_browser\)](https://en.wikipedia.org/wiki/Mosaic_(web_browser))

1.2.3 Sun 公司没落，Java 易手 Oracle

然而，天有不测风云，当年像太阳一样耀眼的 Sun 公司，在 2002 年突然衰落³，从此便江河日下，一下从硅谷最值钱的公司沦为人均市值最低的公司。由于种种原因，2009 年 4 月 20 日，Sun 公司以 74 亿美元（要知道，2001 年，Sun 的市值超过 2000 亿美元）整体出售给了 Oracle，从此 Java 易手，Oracle 开启了 Java 发展的新历程。通过图1.4我们可以简要的了解 Java 发展的历程 [4] 和各个版本发布的时间⁴。

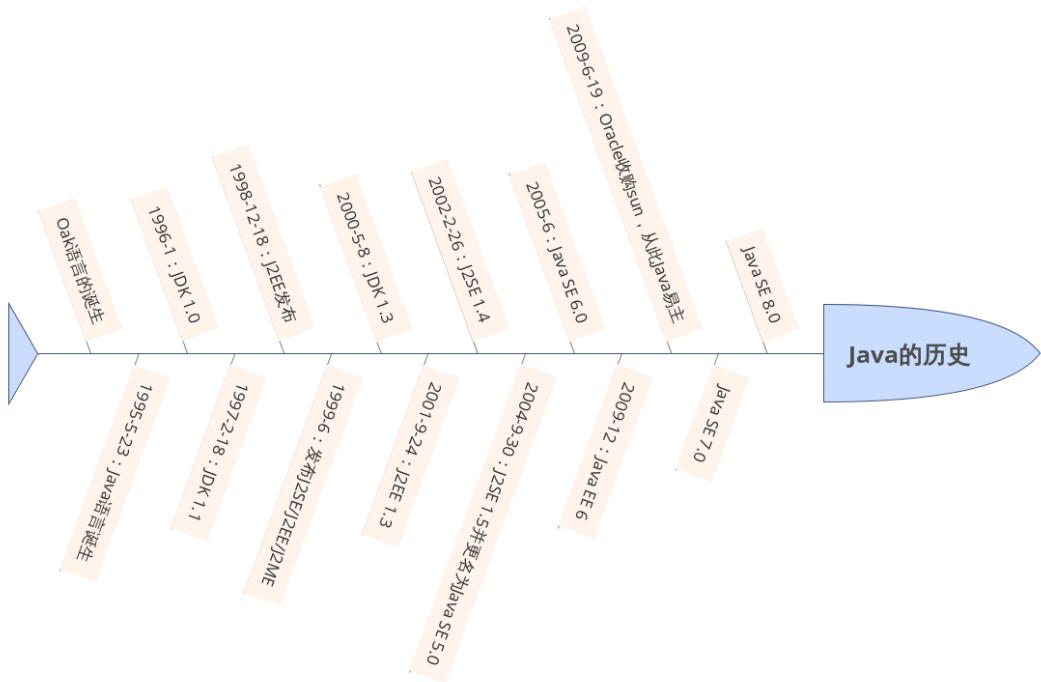


图 1.4: Java 简史

1.3 Java 的特点

1.3.1 面向对象

Java 的特点之一就是面向对象，是程序设计方法的一种。“面向对象程序设计语言”的核心之一就是开发者在设计软件的时候可以使用自定义的类型和关联操作。代码和数

³关于 Sun 衰落的原因，网络上有大量的讨论，比如：<http://tech.sina.com.cn/it/2011-03-04/16485248026.shtml>

⁴如果你对 Java 的历史特别感兴趣，可以参考以下几个站点：

- Oracle 庆祝 Java 20 年：<http://oracle.com.edgesuite.net/timeline/java/>
- Java 20 年：历史与未来：<http://www.infoq.com/cn/news/2015/05/java-20-history-future>

据的实际集合体叫做“对象”。一个对象可以想象成绑定了很多“行为（代码）”和“状态（数据）”的物体。对于数据结构的改变需要和代码进行通信然后操作，反之亦然。面向对象设计让大型软件工程的计划和设计变得更容易管理，能增强工程的健康度，减少失败工程的数量。

1.3.2 跨平台

Java 语言的第二个特性就是跨平台性，也就是说使用 Java 语言编写的程序可以在编译后不用经过任何更改，就能在任何硬件设备条件下运行。这个特性经常被称为“一次编译，到处运行”（write once, run anywhere）。

执行 Java 应用程序必须安装 Java Runtime Environment (JRE)，JRE 内部有一个 Java 虚拟机 (Java Virtual Machine, JVM) 以及一些标准的类库 (Class Library)。通过 JVM 才能在电脑系统执行 Java 应用程序 (Java Application)，这与 .Net Framework 的情况一样，所以电脑上如果没有安装 JVM，Java 应用程序将不能够执行。

实现跨平台性的方法是大多数编译器在进行 Java 语言程序的编码时候会生成一个用字节码写成的“半成品”，这个“半成品”会在 Java 虚拟机（解释层）的帮助下运行，虚拟机会把它转换成当前所处硬件平台的原始代码。之后，Java 虚拟机会打开标准库，进行数据（图片、线程和网络）的访问工作。主要注意的是，尽管已经存在一个进行代码翻译的解释层，有些时候 Java 的字节码代码还是会被 JIT 编译器进行二次编译。

有些编译器，比如 GCJ，可以自动生成原始代码而不需要解释层。但是这些编译器所生成的代码只能应用于特定平台。并且 GCJ 目前只支持部分的 Java API。

Java 语言使用解释层最初是为了轻巧性。所以这些程序的运行效率比 C 语言和 C++ 要低很多，用户也对此颇有微词。很多最近的调查显示 Java 的程序运行速度比几年前要高出许多，有些同样功能的程序的效率甚至超过了 C++ 和 C 语言编写的程序。

Java 语言在最开始应用的时候是没有解释层的，所有需要编译的代码都直接转换成机器的原始代码。这样做的后果就是获得了最佳的性能，但是程序臃肿异常。从 JIT 技术开始，Java 的程序都经过一次转换之后才变成机器码。很多老牌的第三方虚拟机都使用一种叫做“动态编译”的技术，也就是说虚拟机实时监测和分析程序的运行行为，同时选择性地对程序所需要的部分进行编译和优化。所有这些技术都改善了代码的运行速度，但是又不会让程序的体积变得失常。

程序的轻便性事实上是软件编写很难达到的一个目标，Java 虽然成功地实现了“一次编译，到处运行”，但是由于平台和平台之间的差异，所编写的程序在转换代码的时候难免会出现微小的、不可察觉的错误和意外。有些程序员对此非常头疼，他们嘲笑 Java 的程序不是“一次编译，到处运行”，而是“一次编译，到处调试”。以 Java AWT 为例，早期 Java AWT 内提供的按钮、文字区等均是以电脑系统所默认的样式而显示。这令 Java 程序在有些没有提供图案的电脑系统产生错误（在 Microsoft Windows 设有窗

口管理器，在一些 Linux distribution 则没有）。后来 SUN 公司针对 Java AWT 一些问题而推出 Java Swing。

平台无关性让 Java 在服务器端软件领域非常成功。很多服务器端软件都使用 Java 或相关技术建立，比如占据了 WWW 服务器半壁江山的 Apache 服务器⁵，当下火爆的大数据平台 Hadoop⁶等，都是使用 Java 语言开发的。

1.3.3 自动垃圾回收

C++ 语言被用户诟病的原因之一是大多数 C++ 编译器不支持垃圾收集机制。通常使用 C++ 编程的时候，程序员于程序中初始化对象时，会在主机内存堆栈上分配一块内存与地址，当不需要此对象时，进行析构或者删除的时候再释放分配的内存地址。如果对象是在堆栈上分配的，而程序员又忘记进行删除，那么就会造成内存泄漏（Memory Leak）。长此以往，程序运行的时候可能会生成很多不清除的垃圾，浪费了不必要的内存空间。而且如果同一内存地址被删除两次的话，程序会变得不稳定，甚至崩溃。因此有经验的 C++ 程序员都会在删除之后将指针重置为 NULL，然后在删除之前先判断指针是否为 NULL。

C++ 中也可以使用“智能指针”（Smart Pointer）或者使用 C++ 托管扩展编译器的方法来实现自动化内存释放，智能指针可以在标准类库中找到，而 C++ 托管扩展被微软的 Visual C++ 7.0 及以上版本所支持。智能指针的优点是不需引入缓慢的垃圾收集机制，而且可以不考虑线程安全的问题，但是缺点是如果不善使用智能指针的话，性能有可能不如垃圾收集机制，而且不断地分配和释放内存可能造成内存碎片，需要手动对堆进行压缩。除此之外，由于智能指针是一个基于模板的功能，所以没有经验的程序员在需要使用多态特性进行自动清理时也可能束手无策。

Java 语言则不同，上述的情况被自动垃圾收集功能自动处理。对象的创建和放置都是在内存堆栈上面进行的。当一个对象没有任何引用的时候，Java 的自动垃圾收集机制就发挥作用，自动删除这个对象所占用的空间，释放内存以避免内存泄漏。

注意程序员不需要修改 finalize 方法，自动垃圾收集也会发生作用。但是内存泄漏并不是就此避免了，当程序员疏忽大意地忘记解除一个对象不应该有的引用时，内存泄漏仍然不可避免。

不同厂商、不同版本的 JVM 中的内存垃圾回收机制并不完全一样，通常越新版本的内存回收机制越快，IBM、BEA、SUN 等等开发 JVM 的公司都曾宣称过自己制造出了世界上最快的 JVM，JVM 性能的世界纪录也在不断的被打破并提高。

练习 1.1. Java 语言可以跨平台的秘诀是什么？

⁵<http://www.apache.org>

⁶<http://hadoop.apache.org>

1.4 Java 开发环境的搭建

学习一门程序设计语言，最佳实践是边读边练，在一个“全浸”的环境中学习，自然就轻松多了。因此，我们从一开始就搭建一个“舒服”的开发环境，学习中的每一步我们都在这个舒服的开发环境中验证，这样我们走的每一步都是扎实的，串起来就是一个坚实的学习之旅。

Linux 是学习程序设计的绝佳搭档，因此笔者建议大家在 Linux 平台上开始 Java 的学习之旅。Linux 的发布版比较多，根据笔者多年教学经验，建议大家选择 Ubuntu 的最新 LTS 版本即可，下面的 Java 开发环境的搭建均基于 Ubuntu 16.04（写作本书时的最新 LTS 版本）。

1.4.1 Java 版本的选择

Java 分为三个版本：

- Java SE (Java Standard Edition)：Java 的标准版，包含了常见的 Java 应用程序开发和运行所需的环境。本书此后如没有特别说明，Java 都是指 Java SE。
- Java EE (Java Enterprise Edition)：Java 的企业版，包含了 Java 的一些高级特性，如 Servlet, JDBC 等。
- Java ME (Java Micro Edition)：Java 的移动版，包含了开发移动终端应用程序所需的环境。不过，自从 Android 横空出世，Java ME 几乎就没有多大市场了。

1.4.2 JDK 的概念

JDK (Java Development Kit) 是开发 Java 应用程序的必备，提供了开发 Java 应用程序的必须工具，比如：

- **java**：Java 虚拟机，任何 Java 应用程序都是通过 java 这个虚拟机解释执行的。
- **javac**：Java 编译器，将 Java 源代码编译为 java 虚拟机可以理解的字节码文件。
- **javap**：反编译工具，可以粗略的认为，javap 能够将二进制文件还原为 Java 源代码，帮助我们了解 javac 的工作原理。

详细的 JDK 工具列表在下载和安装 JDK（参见节 1.4.3.1 [在对页]）之后转到 bin 目录浏览，在 ubuntu 下面通过以下命令⁷了解 JDK 包含了哪些命令和工具：

```
$ dpkg -L openjdk-8-jdk-headless | more
```

⁷也许你安装的 JDK 版本和这里不一致。可以通过这个命令了解你的电脑上安装的是哪个版本的 JDK：dpkg -l | grep -i "openjdk"

1.4.3 基础开发环境

1.4.3.1 下载和安装 JDK

在 ubuntu 环境下，下载和安装 JDK 可以简单的执行下列命令来完成⁸:

```
$ sudo apt-get install openjdk-8-jdk
```

就这些了！是不是很简单？

验证一下 Java 是不是已经安装好了，执行命令：

```
$ java -version  
openjdk version "1.8.0_91"  
OpenJDK Runtime Environment (build 1.8.0_91-  
3ubuntu1~16.04.1-b14)  
OpenJDK 64-Bit Server VM (build 25.91-b14, mixed mode)  
$ which java  
/usr/bin/java
```

可以看出，我们安装的是 openjdk⁹，不是 Oracle 官方的 JDK。对于我们学习 Java 程序设计而言，openjdk 就足够了。如果你介意的话，也可以去下载 Oracle 官方的 JDK，不再赘述。

现在，我们就可以尝试编写 HelloWord 了！使用任意的文本编辑器编写下面的程序代码并保存为 HelloWorld.java:

例 1.1. 使用 Java 语言向世界问好

代码清单 1.1: HelloWorld.java

```
1 public class HelloWorld {  
2  
3     public static void main(String[] args) {  
4         System.out.println("hello, world!");  
5     }  
6 }
```

打开一个终端，执行下面的命令编译 HelloWorld.java:

```
$ javac HelloWorld.java
```

执行下面的命令欣赏 HelloWorld 吧：

```
$ java HelloWorld
```

```
Hello World!
```

⁸如果你使用的是 windows，请参考<http://jingyan.baidu.com/article/eae07827a4b4a31fec548535.html>了解 windows 下 JDK 的安装和设置方法。由于 Java 是跨平台的，因此 JDK 安装后的用法都是一样的，本书不再强调 windows 和 Linux 中 Java 的用法区别。

⁹<http://openjdk.java.net/>

- 注意在编写 HelloWorld.java 源文件时，文件名 HelloWorld 和源文件中的 HelloWorld 要严格一致的。
- 执行 java 的 class 文件时，java 命令后面的参数不要带 class 字样，即，如果执行 HelloWorld.class，则只输入 java HelloWorld 即可，千万不要画蛇添足变成 java HelloWorld.class。Java 虚拟机会根据 HelloWorld 自动去寻找 HelloWorld.class，如果在终端输入 java HelloWorld.class，java 虚拟机就会去寻找 HelloWorld.class.class 文件，自然就找不到了。

练习 1.2. 请说明 JDK 中，哪个程序是 Java 虚拟机？哪个程序是 Java 编译器？

1.4.4 IDE 开发环境

终端界面（命令行方式）对于编写一个简单的程序很方便，但是终端界面毕竟缺少了一些方便程序设计的功能，比如代码自动提示和补全、文件浏览等，因此在实际的学习和开发中，往往使用所谓的 IDE（Integrated Development Environment）环境。Java 开发中主流的 IDE 有如下三种：

- JetBrain Idea
- eclipse
- NetBeans

笔者推荐大家使用第一种：JetBrain Idea。

1.4.4.1 Idea

JetBrains Idea 是 JetBrains¹⁰ 公司出品的优秀 Java IDE 环境，是很多 Java 高手的最爱，google 公司也将 android 开发的 IDE：android studio 构建在 IDEA 的基础之上（最早的 android 集成开发环境是作为 eclipse 的插件存在的，估计 google 公司对 eclipse 的插件效率和管理方式不满意，于是另起炉灶，在 IDEA 的基础上开发了 android studio）。

IDEA 分为免费的社区版和收费的旗舰版，对于 Java 初学者而言，社区版的功能已经足以应付。

下载和安装 Idea 从 <https://www.jetbrains.com/idea/> 选择 Community 版本下载即可。请根据所使用的操作系统选择不同的压缩格式。作者使用的是 Linux 操作系统，因此选择的是 tar.gz 不带 JDK 的安装包（JDK 已经在节 1.4.2 [在第 14 页]—

¹⁰<https://www.jetbrains.com>

节安装过了)。假设最新的 idea 下载到了`~/downloads`目录下。作者的习惯是在家目录建立一个`devel`目录,然后将所有软件开发相关的工具等都放到`devel`目录下,因此将下载的 idea 安装文件解压缩:

```
$ cd ~/downloads  
$ tar xzvf ideaIC-2016.3.4-no-jdk.tar.gz -C ~/devel  
$ cd ~/devel/idea-IC-163.12024.16/bin  
$ ./idea.sh
```

第一次启动 Idea 的时候需要进行简单的配置,目前阶段全部选择默认选项既可。

创建新项目 下面以 HelloWorld 为例说明如何使用 Idea 创建一个简单的 Java 应用程序。

1. 创建新项目。通过 File->New 菜单打开创建新项目的窗口,如图1.5和图1.6所示。

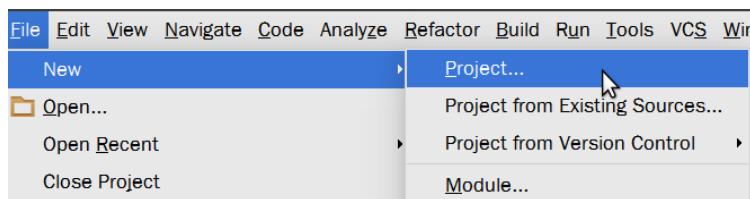


图 1.5: 从菜单新建项目

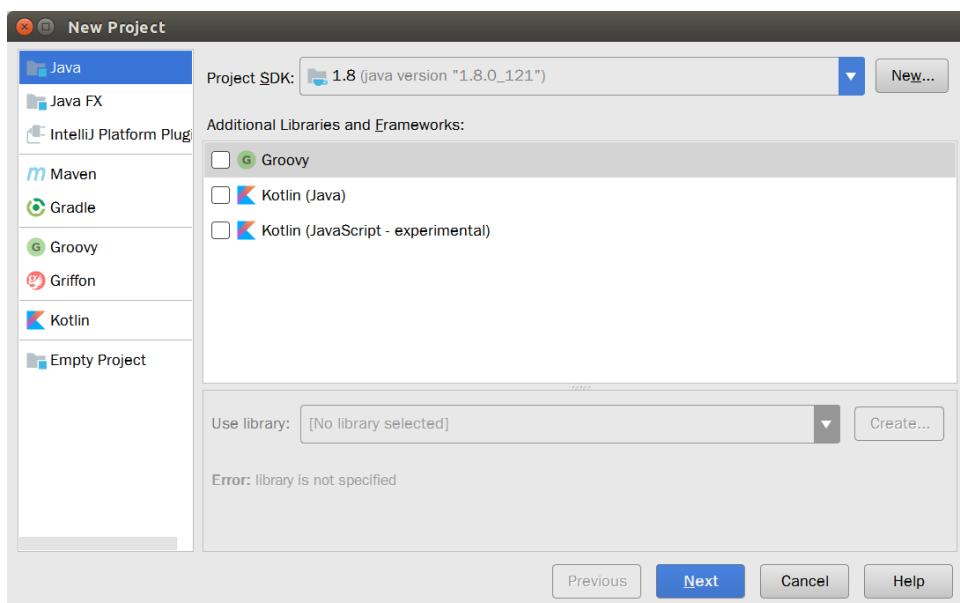


图 1.6: 选择项目类型

2. 接着，如图1.7和图1.8所示，填写项目名称和所在目录，Idea 自动创建项目目录和一个默认的 Main 文件。

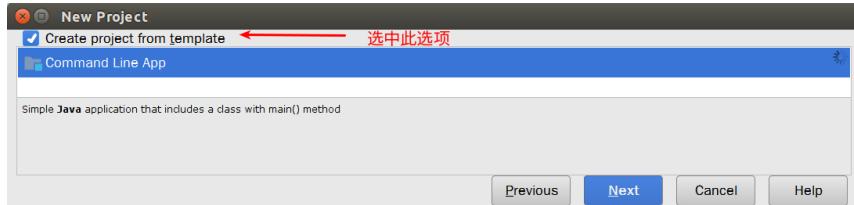


图 1.7: 选择从模板创建项目



图 1.8: 填写项目名称和所在目录

1. 编写源代码。在自动打开的 Main 文件中，编写源代码，如图1.9所示。源代码编

```

1 > public class Main {
2
3 >     public static void main(String[] args) {
4         // write your code here
5         System.out.println("Hello, World!");
6     }
7 }
8

```

图 1.9: 编写源代码

写完毕后，可以将 Main 这个文件改名字为 HelloWorld，如图1.10所示，当光标停在 Main 上面时，两次按 Shift+F6 修改类名。

2. 运行应用程序。如图1.11所示，在右键单击弹出的菜单中选择“Run”运行¹¹即可。

¹¹Ctrl+Shift+F10 是运行应用程序的快捷键。

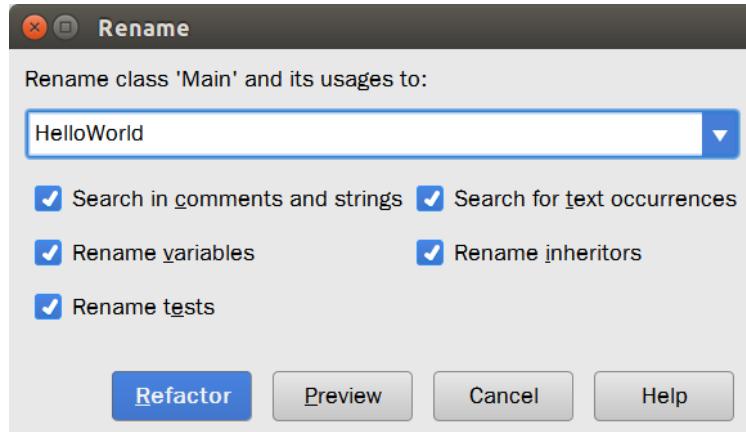


图 1.10: 修改类名

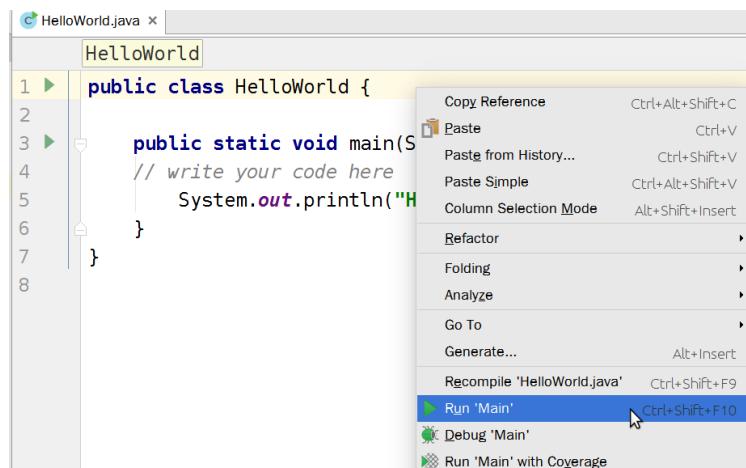


图 1.11: 运行应用程序

建议读者在使用 Idea 完成了第一个 HelloWorld 项目后，使用资源管理器或者命令行工具打开项目的目录看一下 Idea 到底创建了哪些目录，哪些文件？尤其要注意到，Java 应用程序源代码都是使用 `java` 作为文件扩展名的，比如 `HelloWorld.java`，编译后的 Java 字节码文件（即所谓的可执行文件）都是以 `class` 作为文件扩展名的，比如 `HelloWorld.class`。

要特别注意，Java 是大小写敏感的程序设计语言，Java 源代码的文件名和类名必须完全一致！这个规定并非铁板一块，我们将在节 3.2.2 [在第 42 页] 详细讨论 Java 定义类的细节。

1.4.4.2 NetBeans

NetBeans 是一个始于 1997 年的 Xelfi 计划，本身是捷克布拉格查理大学 Charles University 的数学及物理学院的学生计划。此计划延伸而成立了一家公司进而发展成为商用版本的 NetBeans IDE，直到 1999 年 Sun Microsystems 电脑买下此公司。Sun Microsystems 于 2000 年 6 月将 NetBeans IDE 作为开源项目发展。2010 年 1 月，Sun Microsystems 成为甲骨文的子公司。NetBeans IDE 最新版下载量已经超过 18 万次，参与开发人员超过 80 万。NetBeans 项目正在蓬勃发展，并将继续成长。

NetBeans 包括开源的开发环境和应用平台，NetBeans IDE 可以使开发人员利用 Java 平台能够快速创建 Web、企业、桌面以及移动的应用程序，NetBeans IDE 已经支持 PHP、Ruby、JavaScript、Groovy、Grails 和 C/C++ 等开发语言。NetBeans 项目由一个活跃的开发社区提供支持，NetBean 开发环境提供了丰富的产品文档和培训资源以及大量的第三方插件。NetBeans 是开源软件开发集成环境，是一个开放框架，可扩展的开发平台，可以用于 Java、C/C++、PHP 等语言的开发，本身是一个开发平台，可以通过扩展插件来扩展功能。

ubuntu 环境下直接执行下面的命令即可安装 Netbeans：

```
$ sudo apt-get install netbeans
```

上面的命令虽然只是安装了 Netbeans 的基本组件，但是对于我们学习 Java 程序设计语言而言已经足够了！如果你希望尝试最新的 Netbeans 版本，可以从 Netbeans 官网¹² 下载并安装，此处不再赘述。

如果你已经使用过任何一种 IDE 开发环境了，Netbeans 是很容易上手的。一般的开发步骤如下：

1. 创建新项目：通过 File->New Project 打开创建新项目的窗口，填写项目目录即可创建一个新的项目：

¹²<http://www.netbeans.org>

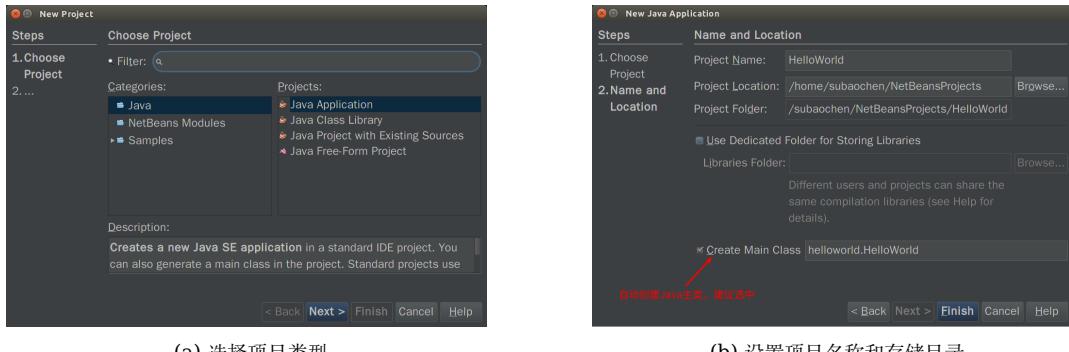


图 1.12: 创建新项目

2. 编写源文件: 在第一步中创建新项目后, 已经自动打开了 HelloWorld.java 文件:

```

1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello World!");
4     }
5 }
```

图 1.13: 编辑 HelloWorld.java



- 巧用 Netbeans 的自动代码提示, 比如打出 System. 之后, 会自动提示 System 类有哪些方法可以调用。
- 如果代码中有语法错误, Netbeans 会给出提示信息和建议的修改方案, 一般情况下选择第一个修正方案即可。

3. 运行: 在 IDE 中运行 Java 应用程序很简单, 在打开的 java 文件中右键可以看到 run file 菜单:

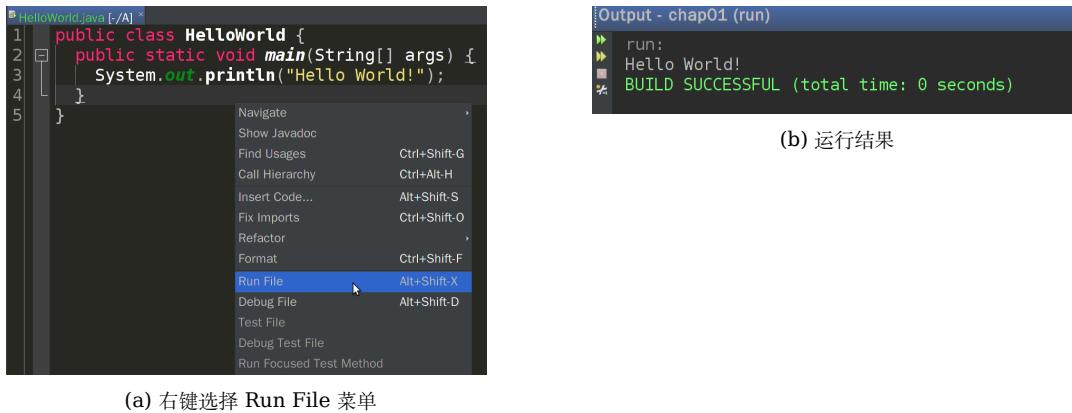


图 1.14: 在 Netbeans 中运行文件

1.4.4.3 Eclipse

Eclispe 是当下非常流行的 Java 开发 IDE 环境，以丰富的插件著称，详情可以访问 Eclipse 的官网：<http://www.eclipse.org>。

安装 Eclipse 可以有两种方式：

1. 直接在 ubuntu 执行命令：
\$ sudo apt-get install eclipse

2. 在 eclipse 的官网下载相应于你的环境（32 位或者 64 位）的合适版本，一般下载最新的稳定版即可。eclipse 是绿色软件，直接解压缩运行其中的 eclipse 文件即可。

eclipse 的具体用法和 Netbeans 大同小异，不再赘述。

1.5 初识 Java 的输入输出

C 语言提供了简单的 printf 和 scanf 处理输出和输入，Java 的输入输出确实要比 C 复杂的多。不过，我们在学习 C 语言的时候也花了不少的时间学习 printf 的格式化输出方式，不是吗？而 Java 的输出虽然形式上复杂了点，但是却不需要我们学习格式化字符串的语法了。为了学习方便，这里先给出最简单的 Java 输入输出方法。

1.5.1 输出到屏幕

在屏幕上输出字符串可以直接调用 **println** 方法（ln 是 line 的缩写）：

```
1 System.out.println( "希望输出的字符串写在这里" );
```

我们先不用理会 System 和 out 具体是什么意思（在节 7.2.1 [在第 172 页] 我们会详细研究 System 以及 out 的用法），`println` 的意思是输出双引号内的字符串到屏幕上，并自动回车换行，这正是我们需要的。

另外，在输出字符串到屏幕时，一般情况下我们可以忘掉 C 中 `printf` 函数的格式化输出，因为 Java 提供了更自然的字符串输出方式：直接使用“+”将多个字符串连接起来即可¹³，比如：

```
1 System.out.println("variable a = " + a + ",variable b = " + b);
```

1.5.2 从键盘输入

从键盘输入的简便方式是使用 `Scanner` 类：

```
1 Scanner console = new Scanner(System.in);
2 int quantity = console.nextInt();
3 double num = console.nextDouble();
4 String str = console.nextLine();
```

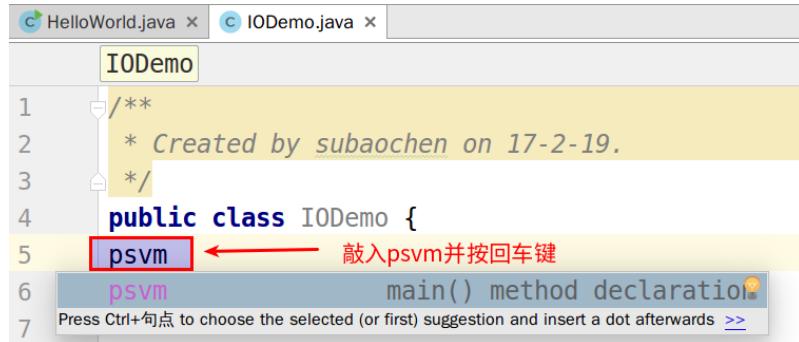
同样的，我们在这里先忽略 `System.in` 的含义，只需要了解 `console.nextInt()` 的意思是读入一个整数，`console.nextDouble()` 的意思是读入一个双精度浮点数即可。`Scanner` 类的常用函数如表 1.15 所示：

| 函数名称 | 功能描述 |
|---------------------------|-------------------------|
| <code>nextInt()</code> | 读入下一个整数 |
| <code>nextShort()</code> | 读入下一个短整数 |
| <code>nextLong()</code> | 读入下一个长整数 |
| <code>nextFloat()</code> | 读入下一个浮点数 |
| <code>nextDouble()</code> | 读入下一个双精度浮点数 |
| <code>next()</code> | 读入下一个 token（空白字符隔开的字符串） |
| <code>nextLine()</code> | 读入下一行 |

图 1.15: `Scanner` 的常用函数

¹³ 在节 5.1.2.4 [在第 129 页] 我们会更具体的讨论如何连接多个字符串。

我们会发现，几乎每个 Java 类都需要创建一个 main 方法，Idea 提供了“模板”机制帮助我们快速创建 main 方法：尝试在一个类的内部敲入 psvm 并按回车键看看，如下图所示：



psvm: 分别是 public static void main 的第一个字母。

例 1.2. Java 的输入输出

参见代码清单1.2，我们从键盘输入什么，就在屏幕上打印出什么。

代码清单 1.2: IODemo.java

```

1 import java.util.Scanner;
2
3 /**
4  * Created by subaochen on 17-2-19.
5 */
6 public class IODemo {
7     public static void main(String[] args) {
8         Scanner console = new Scanner(System.in);
9         String str = console.nextLine();
10        System.out.println("inputed string: " + str);
11    }
12 }
```

练习 1.3. 编写程序，在屏幕打印 10 行 “Hello, Java!”。

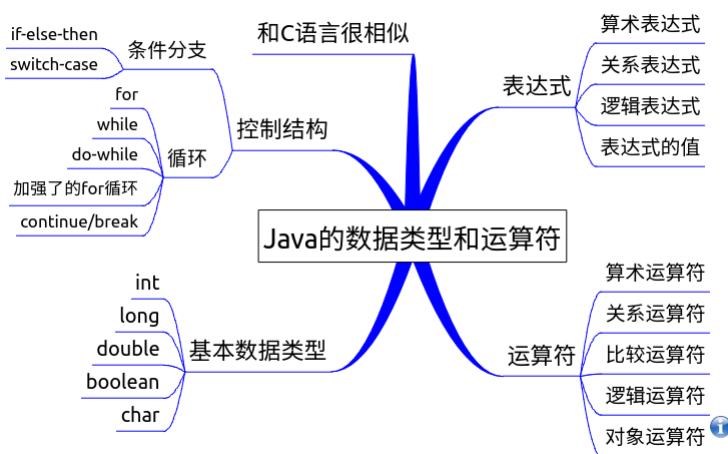
练习 1.4. 编写程序，在屏幕上显示如下的图形[参见解答 1 [在第 349 页]]：

```

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

练习 1.5. 熟悉一下常用的 Idea 快捷键，具体可以参照附录 G [在第 356页]。记住一些常用快捷键可以大大提高开发和学习的效率，因此下些功夫练习一下还是值得的。

第二章 Java 的基础语法



2.1 变量和变量的命名

Java 语言中变量的概念和 C 语言完全一致，这里不再赘述。所不同的是，在 Java 中，我们一般使用“驼峰命名法”¹来给变量起名字，而在 C 语言中一般使用下划线分隔开多个单词，表2.1是常见的 C 语言变量²和 Java 语言变量的对照，也是优秀变量命名的典范：

| C 中的常见变量 | Java 中的变量表示方法 |
|--------------------|-----------------|
| boot_info | bootInfo |
| cmdline_boot_cpuid | cmdlineBootCpid |
| timeAttrs | timeAttrs |
| lookup_flags | lookupFlag |

表 2.1: C 语言的变量和 Java 语言变量对照表

¹ 驼峰命名法：当使用多个单词命名变量时，除第一个单词外，每个单词首字母大写，看起来就像是驼峰一样高低不一。

² 摘自 Linux 内核源代码

合理的变量命名能够加强代码的可读性，减轻团队沟通的压力，软件后期的维护压力也会缓解。从一开始就要有意识的培养合理命名变量的习惯。善于合理的命名变量也是一种能力，这种能力不会从天上掉下来，只有多阅读优秀的代码，注意学习高手的编程思路和变量命名习惯，在平时的练习中坚持给变量合理的命名，才能逐步培养这种能力。本书坚持以合理的变量命名方式给出示例程序，希望通过这种方式让初学者在潜移默化中获得这种能力。

2.2 常量和常量的命名

常量是一种特殊的变量：其值不允许改变的变量。在 C 语言中，我们一般使用全大写字母表示常量，Java 也完全遵守这一通用规则。所不同的是，Java 语言中的常量通常使用 `final static` 进行联合修饰：`final` 表示不可修改，`static` 表示类变量，即所有对象共享一个此常量的备份。显然，对于常量而言，这样的修饰是十分合理和必要的，举例³：

```
1 /* Shutdown state */
2 private static final int RUNNING = 0;
3 private static final int HOOKS = 1;
4 private static final int FINALIZERS = 2;
5 private static final int MAX_SYSTEM_HOOKS = 10;
```

- 请暂时忽略例子中的 `private`，我们将在节 3.6 [在第 68页]深入学习 `private` 的用法。
- 我们将在节 3.8.1 [在第 87页]深入学习 `static` 的用法，这里可以简单理解为和 C 语言中的 `static` 用法类似：在 C 语言中，`static` 变量表示变量位于静态存储区，因此可以在函数之间共享此变量。在 Java 中使用 `static` 修饰变量其实用意很相似：`static` 变量在多个对象中只保存一份，即 `static` 变量的目的是在多个对象中共享变量。

2.3 简单数据类型

Java 的简单数据类型（又称为基本数据类型）和 C 语言几乎完全一致，如表2.2所示。

³ 摘自 openjdk 源代码：<http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/lang/Shutdown.java>

| 类别 | 类型 | 字节数 | 取值范围 | 举例 |
|------|---------|-----|---|--|
| 整数类型 | byte | 1 | -128 ~ 127 (-2 ⁷ ~ 2 ⁷ -1) | byte minor; byte[] image; |
| | short | 2 | -32768 ~ 32767 -2 ¹⁵ ~ 2 ¹⁵ -1 | short state; short index = indexArray[i]; |
| | int | 4 | -2147483648 ~ 2147483647 -2 ³¹ ~ 2 ³¹ -1 | int searchOffset; int nextComma; |
| | long | 8 | - 9223372036854775808 ~ 9223372036854775807 -2 ⁶³ ~ 2 ⁶³ -1 | long timeout; long remaining = 100L; |
| 浮点类型 | float | 4 | $\pm 1.4E - 45f$ ~ $\pm 3.4028235E + 38f$ | float loadFactor; float f = 2.5F; |
| | double | 8 | $\pm 4.9E - 324$ ~ $\pm 1.796931348623157E + 308$ | double scaledFractional; double approxMax; double myNumber = -1234.56; double[] limits = {1,2,3,4,5,6,7}; |
| 字符类型 | char | 2 | \u0000 ~ \uffff (0 ~ 65535) | char c = 'a'; char[] text; char trailChar; |
| 布尔类型 | boolean | 1 | true, false | boolean isDefaultLevel; boolean isInverse; boolean mayAllocateText; boolean first = true; |

表 2.2: Java 的数据类型

代码清单2.1演示了Java中的简单数据类型：

代码清单 2.1: SimpleDataType.java

```
1 public class SimpleDataType {  
2     public static void main(String[] args) {  
3         char c = 'a';  
4         byte b = 20;  
5         int n = 10;  
6         short s = 2;  
7         long l = 100L; // ❶  
8         double d = 2.5D;  
9         float f = 2.5F;  
10      
11    System.out.println("c(char) = " + c + ",size of c = " + Character.SIZE / 8);  
12    System.out.println("b(byte) = " + b + ",size of n = " + Byte.SIZE / 8);  
13    System.out.println("n(int) = " + n + ",size of n = " + Integer.SIZE / 8);  
14    System.out.println("s(short) = " + s + ",size of s = " + Short.SIZE / 8);  
15    System.out.println("l(long) = " + l + ",size of l = " + Long.SIZE / 8);  
16    System.out.println("f(float) = " + f + ",size of f = " + Float.SIZE / 8);  
17    System.out.println("d(double) = " + d + ",size of d = " + Double.SIZE / 8);  
18      
19 }  
20}  
21  
22  
23 //
```

❶ 使用 L 声明 long 型数据是个好习惯

执行结果如下：

```
c(char) = a,size of c = 2  
b(byte) = 20,size of n = 1  
n(int) = 10,size of n = 4  
s(short) = 2,size of s = 2  
l(long) = 100,size of l = 8  
f(float) = 2.5,size of f = 4  
d(double) = 2.5,size of d = 8
```

 C语言中没有字符串数据类型，Java语言中同样没有字符串这种简单数据类型。不过，Java是面向对象的程序设计语言，允许通过定义类来无限扩展数据类型，我们将在节5.1[在第125页]讨论使用JDK中提供的String类来表示字符串以及字符串的相关操作。



实际上，Java 的数据类型分为基本数据类型和引用数据类型，我们在这里只讨论基本数据类型，引用数据类型延迟到节 3.7 [在第 79 页] 讨论。

2.4 运算符

参见表2.3，除了 instanceof 和 new 运算符外，Java 的运算符及其优先级规则和 C 是一样的：

| 类型 | 运算符 |
|-------|---------------------------------------|
| 算术运算符 | +,-,*,/,% , ++ , -- |
| 关系运算符 | <,<=,>,>=,==,!= |
| 逻辑运算符 | &&, ,! |
| 位运算符 | &, ,~,^,<<,>>,>>> |
| 赋值运算符 | =,*=,+=,-=,/=%=,<<=,>>=,>>>=,&=,^=, = |
| 对象运算符 | instanceof |
| | new |

表 2.3: Java 中的运算符

2.5 表达式

我们说“C 语言是一种表达式语言”，可见表达式的概念是多么的重要！实际上，现代的程序设计语言，包括 Java 都可以称作“表达式语言”，表达式由参与运算的元素（称为运算数）和运算符的不同组合而成，即：

定义 2.1. 表达式 = 运算数 + 运算符

要特别强调的是，每一个表达式都一定有一个确定的值，即“表达式的值”。举例说明：

| 表达式类型 | 表达式 | 表达式的值 |
|-------|-----------------|-------|
| 算术表达式 | 1 + 2 - (3 * 4) | -9 |
| 关系表达式 | 1 > 2 | false |
| 逻辑表达式 | 1 && 2 | true |
| 赋值表达式 | a = 3 | true |

图 2.1: 表达式和表达式的值

我们经常在条件判断语句中使用表达式及表达式的值，请参见节 2.6。

例 2.1. 输入圆的半径，求圆的面积

在代码清单 2.2 中，我们将综合运用 Java 的输入输出和算术表达式。也请注意 Java 常量的定义和使用方式。

代码清单 2.2: CircleArea.java

```
1 import java.util.Scanner;
2
3 public class CircleArea {
4
5     // 定义常量PAI
6     public static final double PAI = 3.14; // ①
7
8     /**
9      * main 函数是 Java 程序的执行入口，这一点和 C 语言很相似。
10     *
11     * @param args 命令行参数，和 C 语言的 argv 也很相似
12     */
13    public static void main(String[] args) {
14        Scanner console = new Scanner(System.in);
15        System.out.print("请输入半径: "); // ②
16        float radius = console.nextFloat();
17        double area = PAI * radius * radius; // ③
18        System.out.println("面积=" + area); // ④
19    }
20 }
21 //
```

① 如果去掉 `static` 会怎样？我们将在“类和对象”一章揭开谜底

② `print` 函数只输出内容，不会自动换行

③ 表达式的最终类型是 `double`，和 C 语言的类型转换规则是相同的

④ `println` 输出内容后自动换行。请思考如何使得输出结果只精确到小数点后 3 位？

执行结果如下：

```
请输入半径: 10.3
面积 =333.12261233749405
```

2.6 条件判断和分支

2.6.1 条件判断

Java 中的条件判断语句用法和 C 语言完全一致，不再赘述，通过一个例子看一下在 Java 中条件判断语句的基本用法。

例 2.2. 输入成绩，输出成绩的等级（优秀、良好、及格、不及格）

代码清单 2.3: ShowGrade.java

```

1 import java.util.Scanner;
2
3 public class ShowGrade {
4
5     /**
6      * 根据成绩计算成绩等级.
7      *
8      * @param args 命令行参数
9      */
10    public static void main(String[] args) { // ①
11        Scanner console = new Scanner(System.in);
12        System.out.print("请输入成绩: "); // ②
13        float grade = console.nextFloat();
14        System.out.print("成绩" + grade + "的等级为: ");
15        if (grade >= 85) {
16            System.out.println("优秀");
17        } else if (grade >= 75) {
18            System.out.println("良好");
19        } else if (grade >= 60) {
20            System.out.println("及格");
21        } else {
22            System.out.println("不及格");
23        }
24    }
25 }
26 //
```

① 自动生成 main 函数的小技巧：输入 `psvm` 然后按下 `tab` 键即可

② 在光标闪烁处输入成绩

执行结果为：

```

请输入成绩: 87
成绩 87.0 的等级为: 优秀
```

2.6.2 Java 加强了的 switch 分支结构

Java 中的 switch 语句，不仅可以接受一个整数表达式（即表达式的值为整数），也可以接受一个字符串表达式（即表达式的值为字符串）。

例 2.3. 输入月份的英文缩写，输出月份的天数

代码清单 2.4: Month.java

```
2 import java.util.Scanner;
3
4 public class Month {
5
6     /**
7      * 根据月份的英文缩写输出给定月份的天数.// ①
8      *
9      * @param args 命令行参数
10     */
11    public static void main(String[] args) {
12        int days = 0;
13
14        Scanner console = new Scanner(System.in);
15        System.out.print("请输入月份的英文缩写: ");//②
16        String month = console.next();//③
17        switch (month) {
18            case "FEB": // ④
19                days = 28;
20                break;
21            case "JAN":
22            case "MAR":
23            case "MAY":
24            case "JUL":
25            case "AUG":
26            case "OCT":
27            case "DEC":
28                days = 31;
29                break;//⑤
30            case "APR":
31            case "JUN":
32            case "SEPT":
33            case "NOV":
34                days = 30;
35                break;
36            default:
37                System.out.println("输入错误, 请检查月份英文缩写是否为全部大写");
38        }
39
40        System.out.println("月份" + month + "的天数为" + days);
41    }
42 }
43
44 }
45 //
```

① 注意到这一行最后的小句点: 这是 google 编码规范要求的

② 注意到, 这里需要在紧跟着“.”后面输入月份的英文缩写, 而不是回车再输入

③ 从键盘读入下一个标识符, 使用空格、Tab 或者回车隔开各个标识符

④ 在这里故意忽略了闰年问题, 你能修改一下这个程序, 解决闰年问题吗?

⑤ 这里的 break 不要忘记

2.7 循环

2.7.1 三大循环结构

Java 中的三大循环结构（for 循环、while 循环、do-while 循环）以及 break、continue 的用法和 C 语言完全一致，不再赘述。

2.7.2 Java 加强了的循环结构

Java 提供了一种加强了的 for 循环语句，可以很方便的遍历集合（包括数组），其基本结构为：

for(元素类型 T 循环变量 t: 集合 s) {...}

此加强了的 for 循环会自动从集合 s 中取出每一个元素赋值给 “:” 前面定义的循环变量 t，这样在循环体中就可以使用这个变量 t 了。

例 2.4. 遍历数组

代码清单 2.5: EnhancedFor.java

```

1
2 public class EnhancedFor {
3
4     public static void main(String[] args) {
5         int[] array = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
6         for (int i : array) {
7             System.out.println(i);
8         }
9     }
10 }
11 }
```

2.8 方法的不定长参数

Java 方法的参数可以是不定长的，即任意个数的参数。不定长参数的表示方法是在参数类型和参数名之间使用... 隔开，这样参数名在方法体内就被当做一个数组看待，在方法体内可以遍历这个数组取得每一个参数。

比如将任意多个整数相加的方法，可以如代码清单2.6所示。

2.9 Java 的注释

Java 支持两种方式的注释：

代码清单 2.6: VarargsTest.java

```

1 /**
2  * @author subaochen.
3 */
4 public class VarargsTest {
5     public static void main(String[] args) {
6         System.out.println("1 + 2 + 3 = " + add(1, 2, 3));
7         System.out.println("1 + 2 + 3 + 4 = " + add(1, 2, 3, 4));
8     }
9
10    /**
11     *
12     * @param x
13     * @return
14     */
15    public static int add(int... x) {
16        int result = 0;
17        for (int i = 0; i < x.length; i++)
18            result += x[i];
19
20        return result;
21    }
22 }
```

1. C 语言风格的注释：即`/*...*/`方式的注释。

2. C++ 语言风格的注释：即`//...` 方式的注释。

通常，多行的注释采用 C 语言风格比较方便，单行注释采用 C++ 语言风格比较方便。习惯上，多行的注释使用`/**` 开头，比如：

```

1 /**
2  * 方法的功能描述。
3  * @param param1 参数1的描述
4  * @param param2 参数2的描述
5  * @return 返回值的描述
6 */
7 public String method(String param1, int param2) {...}
```

在上面的例子中，`@param`,`@return` 分别用于描述方法的参数和返回值，`javadoc`⁴可以根据这些标签自动生成 API 文档，建议读者在阅读优秀的源代码时注意学习注释的写法，养成写规范注释的习惯。

⁴`javadoc` 是 Java 提供的 API 文档生成工具，位于 SDK 的 bin 目录下，即和 `java`、`javac` 在同一个目录下。Java SDK 的 API 文档就是通过 `javadoc` 工具生成的，简单的用法是运行：`javadoc` 包名（类名列表），即可在当前目录下生成 HTML 格式的标准 API 文档，读者可自行尝试和体会。详细了解 `javadoc` 可以参考：<https://en.wikipedia.org/wiki/Javadoc> 和 <http://www.oracle.com/technetwork/articles/java/index-137868.html>，使用 Linux 的读者也可以执行 `man javadoc`。



大多数 IDE 都支持自动插入注释框架，比如在 Idea 中，只要在方法（函数）的上一行输入`/**`（注意是两个 *）再按回车，然后在空白处补充描述即可。

2.10 综合应用举例

例 2.5. 利用下面的公式计算 π 的近似值，要求精确到 10^{-6} 为止。

$$\pi/4 \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

问题分析 这是利用级数公式近似计算某个特定值的典型应用，计算结果的精确度取决于累加的项数多少，累加的项数越多，精度越好。

设计说明 在代码清单2.7中，设计了一个计算 π 近似值的方法⁵，这样在 `main` 方法中调用这个方法即可。

代码清单 2.7: CalculatePi.java

```

1 public class CalculatePi {
2
3     public static void main(String[] args) {
4         System.out.printf("pai = %.6f", pi());
5         System.out.println();
6     }
7
8     /**
9      * 计算Pai的近似值。
10     *
11     * @return Pai的近似值
12     */
13    public static double pi() {
14        double result = 0; // 最终结果
15        double item; // 保存每一项的结果
16        int denominator = 1; // 每一项的分母
17        int sign = 1; // 符号状态
18
19        do {
20            item = (double) sign / denominator;
21            result += item;
22            sign = -sign;
23            denominator += 2;
24        } while (Math.abs((double) sign / denominator) >= 1e-6);

```

⁵本书大多数地方把 C 语言中的函数（function）称作“方法”（method），这是面向对象程序设计语言中对函数的称呼。其实，这两者没有差别，只是叫法不同而已。也就是说，在面向过程的语言中，通常称作函数；在面向对象的语言中，通常称作方法。

```
25         return result * 4;
26     }
27 }
28
29 }
```

显示结果 运行例2.5后，输出结果如下：

```
pai = 3.141591
```

程序说明 本例的主要内容是计算 π 近似值的方法 pi()。其设计思路很直接，利用变量 item 保存每个迭代项的数值，然后累加到 result 中即可。注意到 sign 的用法很常见，在每次迭代的时候实现了符号的变换。

思考和练习

练习 2.1. Java 的基本语法结构中，哪些方面和 C 语言一致？Java 增加了哪些新特性？

练习 2.2. 从键盘输入球体的半径，求球体的体积。

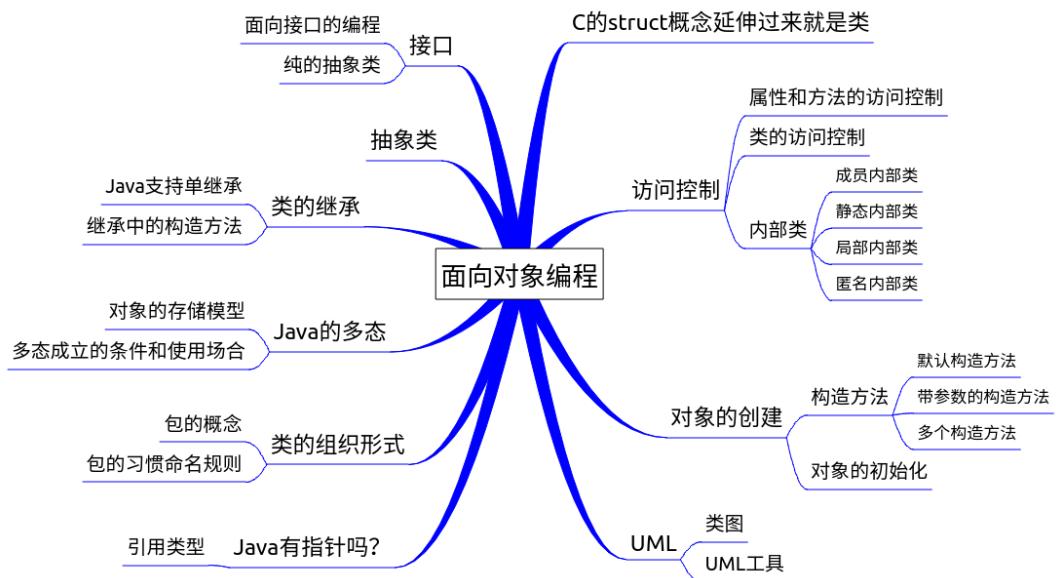
练习 2.3. 编写一个程序，寻找一组整数中的最大值和最小值。输入格式：首先输入 N，表示这组整数的个数，然后依次输入这组整数。

练习 2.4. 编写字符界面版计算器程序，运行时提示输入两个操作数，然后输出加减乘除运行结果。

练习 2.5. 编写一个程序，从键盘输入 x，利用下面公式计算的近似值：

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \dots$$

第三章 面向对象编程基础



3.1 再说 C 语言的 struct

3.1.1 struct 的起源

C 语言以简练、灵活、高效著称，但是 C 语言本身提供的基本数据类型有限，描述现实世界的能力自然也有限。比如要求输入班级同学的英语和数学成绩并按照平均成绩排序，如果我们仅使用 C 的基本数据类型来设计的话，大概能想到的策略有如下几种：

1. 每个同学都用三个变量来表示：zhangsan_name, zhangsan_english, zhangsan_math, wangwu_name, wangwu_english, wangwu_math.....。如果班级有 40 位同学，这需要 120 个变量来表达，不仅容易写错，还不胜其烦！这个思路显然值得商榷，为下策。
2. 使用数组来分别存储同学的名字、英语成绩和数学成绩，比如下面的代码片段：

```
1 char* name[40];
2 float english_score[40];
3 float math_score[40];
```

使用数组极大减少了变量的数量，但是也存在一个问题，我们其实隐含了这样一条规则：`name` 数组的第 i 个元素（同学）的英语成绩是 `english_score` 数组的第 i 个元素，数学成绩是 `math_score` 数组的第 i 个元素。也就是说，这三个数组的对应元素有一一映射的关系。这个一一映射的关系是需要我们程序员自己维护的，也就是说，如果我们破坏了这一隐含的规则，无论有意还是无意，编译器并不能帮助我们发现这个“破坏”，因为从语法上，不符合这个隐含规则也是允许的，这即为程序员带来了负担，也为程序埋下了隐患。可以看出，带来这些问题的关键原因是三个数组的一一映射关系不是语法规级别的映射，编译器无法从语法上保证和检查这三个数组是否遵循了一一映射的关系，这就是 C 语言引入 `struct` 概念的原因：将变量之间的映射关系语法化，我们看使用 `struct` 如何解决这个问题。

3. 使用 `struct` 描述同学们的英语和数学成绩，代码片段如下：

```
1 struct score{
2     char* name;
3     float english_score;
4     float math_score;
5     float avg_score; /*平均成绩*/
6 };
7 struct score stu_scores[40];
```

`struct` 在语法上保证了 `name`、`english_score`、`math_score` 这三个变量是一一映射的，是 `score` 结构体的“成员分量”，因此 `struct` 在语法上保证了名字和成绩不可能张冠李戴，杜绝了可能存在的隐患，也减轻了程序员的思想负担（不需要时刻提醒自己要维护姓名和成绩的映射关系）。所以，`struct` 是 C 语言的重要组成部分，是 C 语言描述现实世界的重武器，在大型项目中大量使用 `struct` 来表达同类事物的共同特征（属性）。比如打开 Linux 内核源代码的头文件目录，我们随处可见大量的 `struct` 定义：

```
1 struct font_desc {
2     int idx;
3     const char *name;
4     int width, height;
5     const void *data;
6     int pref;
7 };
```

因此，我们可以这样定义 `struct`：

定义 3.1. `struct` 是对一类事物公共属性的描述。

3.1.2 struct 的局限性

C 语言通过 struct 大大扩充了 C 的数据类型，每一个 struct 都定义了一种新的数据类型来对应客观世界的某种事物，这样 C 语言不但描述客观世界的能力大大加强了，C 语言程序的可读性、可维护性也大大提高了。

C 语言能够更好的描述客观世界，可以给 struct 记大功一件。但是，我们描述客观世界的是认识世界并进而改造世界，struct 仅仅是方便了描述世界，对于改造世界并没有多大帮助。比如回到我们本章开头的问题：我们不仅仅要输入同学们的成绩，还要按照平均成绩排序。这里至少包含两个方面的“认识世界”和“改造世界”世界的动作：

1. 计算平均成绩：这是属于进一步认识世界的范畴：根据已有的事物属性，我们可以计算和推断出新的属性。
2. 按照平均成绩排序：这是属于改造世界的范畴：对已有事物重新组合排序。

struct 帮助我们很好的描述了同学们的名字和成绩的对应关系，但是计算平均成绩和按照成绩排序这两个“动作”，在 C 语言中是通过独立的函数来实现的，比如下面的代码片段：

```

1 void calculate_avg_score(struct score* score) {
2     /*按照某种权重算法计算平均成绩并保存到score.avg_score中*/
3 }
4 struct score[] sort_score(struct score* scores[]) {
5     /* 将数组scores排序后返回新的struct score数组*/
6 }
```

可以看出，calculate_avg_score 函数和 sort_score 函数虽然都和 struct score 有关系（作为参数或者返回值），但是这种关系是“弱关系”，因为 calculate_avg_score、sort_score 可以没有定义，可以存在于另外的文件中，可以被定义为另外的名称等等。也就是说，在语法上，struct score 和这两个函数并没有必然的联系，编译器没有办法帮助程序员检查这两个函数和 struct score 的映射关系，一切还需要程序员人工的“细心”照料。显然，struct 一定程度上解决了描述世界的问题，但是没有解决认识世界和改造世界的问题。

3.2 类和对象的初步概念

3.2.1 类是 struct 概念的自然延伸

C 中的 struct 实现了对客观事物属性的分组描述，即同一类型的客观事物的属性可以用一个 struct 来表达。正如节 3.1.2 所述，struct 有其局限性，只能描述客观事

物的属性，不能表达对客观事物的进一步认知动作和改造利用的动作，于是在面向对象的程序设计语言中引入“类”的概念弥补 struct 的这一缺陷。

定义 3.2. 类是对一类客观事物公共属性和动作的描述。

对比 struct 我们可以看出，类比 struct 多了“动作”的描述，也就是说，类即可以描述事物的属性，也可以描述我们可以对（用）这类事物“做什么”，是对客观世界的完整的描述。Java 使用 **class** 关键字来定义类，一个简单的只包含属性的类和 struct 非常相似，比如二维坐标中的“点”，一开始可以这样定义（右边作为对照，列出了 C 中 point 结构体的定义）：

代码清单 3.1: Java 的 Point 类定义

```
1 class Point {  
2     int x; // 点的x坐标  
3     int y; // 点的y坐标  
4 }
```

代码清单 3.2: C 的 point 结构体定义

```
1 struct point {  
2     int x; /*点的x坐标*/  
3     int y; /*点的y坐标*/  
4 };
```

但是如果只是使用 class 替换了 struct 关键字，显然没有发挥类描述客观事物的最大威力，因此一个更合理的 Point 类如代码清单3.3所示。

代码清单 3.3: Point.java

```
1 package cn.edu.sdu.tsoftlab.oopbasic.sub321;  
2  
3 /**  
4  * 表达二维坐标点.  
5 */  
6 public class Point {  
7  
8     int x;  
9     int y;  
10  
11    /**  
12     * 设置点的x坐标.  
13     *  
14     * @param newX  
15     */  
16    void setX(int newX) {  
17        x = newX;  
18    }  
19  
20    /**  
21     * 设置点的y坐标.  
22     *  
23     * @param newY  
24     */  
25    void setY(int newY) {  
26        y = newY;
```

```

27 }
28 /**
29 * 移动点(x,y)到(newX, newY).
30 *
31 * @param newX 新的x坐标
32 * @param newY 新的y坐标
33 */
34 void moveTo(int newX, int newY) {
35     setX(newX);
36     setY(newY);
37 }
38 }
39 }
```

可以看出，在这个版本的 Point 类定义中，“点”的属性（x, y）和对点的操作 moveTo 封装在一起了。

3.2.2 定义 Java 类

完整的定义 Java 类的语法是：

[public] class ClassName { ... }

这里有几个要点：

- 大部分情况下需要将类定义为 public 的，参见节 3.6.2 [在第 77页]。
- 类名的常规命名方式驼峰命名法，并且第一个字母是大写的。
- 一个 Java 源代码文件中只能有一个类是 public 的，且这个 public 的类名必须和文件名完全一致，包括大小写。
- 在 Java 源代码文件中可以定义多个类，只要保证只有一个 public 类即可。

3.2.3 对象的概念

类是对一类事物的抽象描述，比如 Point 类是对二维坐标“点”的抽象描述，它表达了屏幕上的一个点所具有的公共属性：x 坐标和 y 坐标，以及我们可以对“点”的操作，比如 moveTo 操作。但是，Point 类并没有和屏幕上的实体点绑在一起，屏幕上的每一个实体点，可以看作 Point 类的一个具体实例，都具有 Point 类给出的一些属性，在这里就是屏幕上的每一个实体点都有相应的 x、y 坐标。我们把实体点叫做类 Point 的实例（instance），或者叫做类 Point 的对象。对象就是实例，实例就是对象。可以看出，一个类可以创建若干实例，所有实例（对象）都具有类中定义

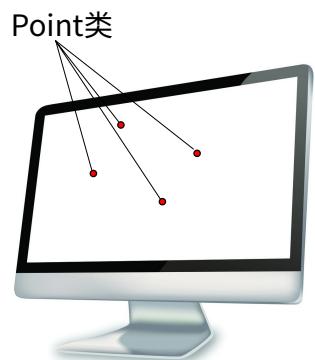


图 3.1: 类和实例的关系

的属性，只是属性的值各不相同而已，一般称作对象的状态不同。比如，点(1,2),(3,5),(10,50)都是Point的实例。

在这里有必要进一步澄清一下常见的几种说法：

- **类 (class)** 就是类型的意思，定义了一个类，就是定义了一个新的数据类型，表达某种事物的公共属性和方法，我们可以根据类创建多个实例（对象）。
- **对象 (object)** 即类的实例。如果把类看作数据类型，则对象可以看作变量。
- **对象的属性 (property)** 即为类的“成员分量”（借用struct的叫法），即类中定义的变量。
- **对象的方法 (method)** 即为类中定义的函数。由于这些函数往往是操作类的属性的某种方法，因此大家习惯上称为对象的方法。

在java中，通过new操作符创建类的实例（对象），比如new Point()创建了一个“点”的实例（对象）。和C语言中访问struct的成员变量类似，访问Java对象的属性和方法是通过.运算符实现的¹。

代码清单 3.4: Draw.java

```
1 package cn.edu.sdu.tsoftlab.oopbasic.sub321;
2
3
4 public class Draw {
5
6     public static void main(String[] args) {
7         Point one = new Point();
8         Point two = new Point();
9
10        one.setX(1);
11        one.setY(1);
12        two.setX(2);
13        two.setY(2);
14
15        printPoint(one, two);
16
17        one.moveTo(3, 3);
18        two.moveTo(5, 0);
19
20        printPoint(one, two);
21    }
22
23    static void printPoint(Point one, Point two) {
24        System.out.println("one.x=" + one.x + ",one.y=" + one.y);
25        System.out.println("two.x=" + two.x + ",two.y=" + two.y);
26    }
}
```

¹和C语言不同的是，由于Java中没有提供指针，在Java中不存在->运算符。

```
27
28 }
```

在程序中，我们通过 new 操作符创建了 2 个 Point 类的对象 one 和 two，并设置了两个对象的状态（即 x、y 坐标），然后打印出两个对象的状态。

例 3.1. 根据已有的 Point 类，编写一个 Rectangle 和 Circle 类，并能够计算 Rectangle 和 Circle 的面积²。

问题分析 矩形（Rectangle）是由一个点和长、宽决定的，圆形（Circle）是由一个点和半径决定的，因此我们可以在 Point 类的基础上定义 Rectangle 和 Circle 类。

设计说明 参见代码清单3.5和代码清单3.6：

代码清单 3.5: Rectangle.java

```
1 package cn.edu.sdut.softlab.oopbasic.sub322;
2
3 public class Rectangle {
4     Point startPoint = new Point(); //
5     int width;
6     int height;
7
8     int area() {
9         return width * height;
10    }
11 }
```

代码清单 3.6: Circle.java

```
1 package cn.edu.sdut.softlab.oopbasic.sub322;
2
3 public class Circle {
4     static final double PI = 3.14159;
5     Point origin = new Point(); //
6     float radius;
7
8     double area() {
9         return PI * radius * radius;
10    }
11 }
```

主类 Draw.java 的设计参见代码清单3.7：

²完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/sub322>

代码清单 3.7: Draw.java

```
1 package cn.edu.sdut.softlab.oopbasic.sub322;
2
3 public class Draw {
4
5     public static void main(String[] args) {
6         Rectangle rect1 = new Rectangle();
7         rect1.startPoint.x = 10;
8         rect1.startPoint.y = 20;
9         rect1.height = 10;
10        rect1.width = 10;
11
12        Circle circle1 = new Circle();
13        circle1.origin.x = 10;
14        circle1.origin.y = 20;
15        circle1.radius = 10;
16
17        System.out.println("rect1's area=" + rect1.area());
18        System.out.println("circle1's area=" + circle1.area());
19
20    }
21
22 }
```

显示结果 执行文件 Draw.java 的结果如下：

```
rect1's area=100
circle1's area=314.159
```

程序说明 注意到我们在 Rectangle 中使用 Point 类定义一个 startPoint 对象的同时初始化了这个对象，即通过 new Point() 创建了一个空的 Point 对象。这是必要的，因为我们在 Draw.java 中创建一个 Rectangle 对象 rect1 后，通过 rect1.startPoint.x 设置 startPoint 的 x 坐标。如果不在 Rectangle 中创建空的 Point 对象，则 rect1.startPoint.x 操作会报告著名的 **NPE** (NullPointerException: 空指针) 异常：

```
Exception in thread "main" java.lang.NullPointerException
at cn.edu.sdut.softlab.oopbasic.sub322.Draw.main(Draw.java:7)
```

3.3 对象的创建过程

当我们使用 `new Point()` 创建一个 `Point` 类的对象时, Java 是如何创建这个对象的呢? 实际上, Java 创建对象的过程如图3.2所示。

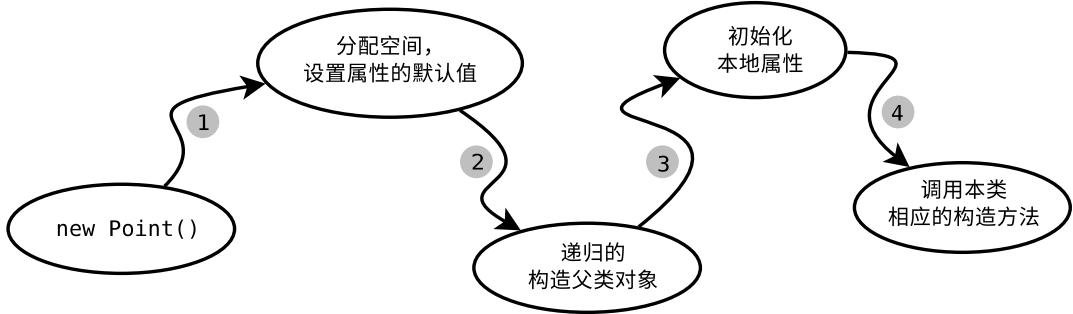


图 3.2: 对象的构造过程

在这里我们暂且不讨论“递归的构造父类对象”这个话题(留待节 3.5.2 [在第 59 页]再讨论), 其他的几个步骤是指:

- 步骤 1: 分配空间。我们已经看到, 相对于简单的数据类型, Java 的对象往往是个复杂的数据结构, 其中除了一些基本的数据类型的属性之外, 还可能包含其他的对象 (比如 `Rectangle` 中包含了 `Point`) 以及方法代码, 因此创建 Java 对象的第一步是为这些数据和方法分配内存空间。分配内存空间后, 类中的基本数据类型属性会被自动设置为默认值: 数字类型的属性默认值为 0 (或者 0.0), `boolean` 类型的默认为 `false`, 字符类型的默认为” (这里是两个单引号, 和 C 语言中的用法一样, 表示空字符)。如果使用类作为属性, 则默认为 `null`。
- 步骤 3: 初始化本类属性。如果本类的属性在定义时由初始化表达式, 则使用该表达式初始化属性。比如

```

1 int x = 10;
2 Point startPoint = new Point();

```

- 步骤 4: 调用本类的构造方法, 这是本节的重点。

3.3.1 构造方法

构造方法的用途, 我们在图 3.2 中已经看到了, 那么如何定义构造方法呢? 简单的说, **构造方法就是和类同名并且没有返回值的方法**。这里有两个关键点: 一是构造方法的方法名必须和类名完全相同, 包括大小写。二是构造方法不需要返回值, 也不允许有返回值。

3.3.2 默认的构造方法

可是，我们在以前的类中，并没有看到构造方法，是怎么回事呢？原来，如果一个类没有定义任何的构造方法，java 编译器会自动增加（插入）一个“默认的构造方法”，即不带任何参数的空的构造方法。也就是说，我们 new Point() 创建一个对象的时候，最后一步会调用 Point 类的无参构造方法。如果 Point 类没有提供这样的构造方法，则调用 Point 类的默认构造方法。

当然，我们也可以“主动”为 Point 编写一个无参的构造方法，这样 new Point() 的最后一步就是调用我们自己编写的构造方法而非默认构造方法了。

 注意无参构造方法和默认构造方法的区别：默认构造方法一定是无参的构造方法，但是无参的构造方法不一定是默认构造方法。当一个类没有显式的定义任何构造方法时，编译器自动添加默认构造方法；当一个类有任何的构造方法时，编译器都不会自动添加默认构造方法。

例 3.2. 编写程序，覆盖 Point 类的默认构造方法，为 Point 对象提供默认的 (x,y) 坐标为 (10,10)³

设计说明 这里只列出 Point.java 和 Draw.java 两个文件，参见代码清单3.8和代码清单3.7，Circle.java 和 Rectangle.java 没有变化。

运行结果 运行 Draw.java 文件结果如下：

```
rect1 的左上角坐标: (10,10),width=10,height=10, 面积 =100  
circle1 的原点坐标: (10,10),radius=10.0 面积 =314.159
```

程序分析 在本例中，Draw.java 创建 Circle 对象 circle1 的时候，并没有设置原点的坐标，但是从运行结果可以看出，原点坐标的值为 (10,10)，这就是 Point 类的无参构造方法的作用：当我们使用 new Point() 创建对象时，最后会调用 Point 的无参构造方法初始化对象的状态。

3.3.3 带参数的构造方法

很多时候，我们希望在创建对象的时候就设置对象的初始状态，比如是否可以通过 new Point(10,20) 这样的方式，创建 Point 对象的同时，设置 Point 对象的 x 坐标为 10,y 坐标为 20 呑？这可以通过带参数的构造方法来实现。也就是说，当我们通过

³完整代码参见<https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/sub332>

代码清单 3.8: Point.java

```
1 package cn.edu.sdut.softlab.oopbasic.sub332;
2
3 /**
4  * 这个例子演示了覆盖默认构造方法时的情形。
5  *
6  * @author SuBaochen:subaochen@126.com
7 */
8 public class Point {
9
10    int x;
11    int y;
12
13    Point() {
14        x = 10;
15        y = 10;
16    }
17
18    void setX(int newX) {
19        x = newX;
20    }
21
22    void setY(int newY) {
23        y = newY;
24    }
25
26    void moveTo(int newX, int newY) {
27        setX(newX);
28        setY(newY);
29    }
30 }
```

代码清单 3.9: Draw.java

`new Point(10,20)` 这样的形式创建 `Point` 对象时，最后会调用 `Point` 类中带两个参数的构造方法。

例 3.3. 编写程序，使用带参数的构造方法创建 `Point` 对象⁴

设计说明 主类 `Draw.java` 没有变化，这里就不列出了。其他类参见代码清单 3.10、代码清单 3.11、代码清单 3.12。

运行结果 运行 `Draw.java` 结果如下：

```
rect1 的左上角坐标: (20,20),width=10,height=10, 面积 =100  
circle1 的原点坐标: (0,0),radius=10.0 面积 =314.159
```

代码分析 当一个类中定义了任何的构造方法后，编译器就不会自动创建默认的构造方法了。比如 `Point` 类中，如果定义了 `Point` 的有参构造方法而没有定义无参构造方法，此时 `Point` 类没有默认构造方法，自然 `new Point()` 方式创建 `Point` 对象时会因为找不到无参的构造方法而报错。如何解决这个问题呢？请参见节 3.3.4。

3.3.4 多个构造方法

Java 允许在类中定义多个构造方法，以便根据不同的情况初始化对象为不同状态。

例 3.4. 编写程序，给 `Point` 类定义两个构造方法⁵

设计说明 主类 `Draw.java` 和 `Rectangle.java` 和例 3.3 相同，这里仅列出 `Point.java` 和 `Circle.java`，参见代码清单 3.13 和代码清单 3.14。

运行结果 运行 `Draw.java` 结果如下：

```
rect1 的左上角坐标: (10,10),width=10,height=10, 面积 =100  
circle1 的原点坐标: (0,0),radius=10.0 面积 =314.159
```

程序说明 `Point` 类现在有两个构造方法，我们在 `Circle.java` 调用了 `Point` 的有参构造方法，在 `Rectangle.java` 中调用了 `Point` 的无参构造方法。

⁴ 完整代码参见 <https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/sub333>

⁵ 完整代码参见 <https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/sub334>

代码清单 3.10: Point.java

```
1 package cn.edu.sdut.softlab.oopbasic.sub333;
2
3 /**
4  * 这个例子演示了覆盖默认构造方法时的情形。
5  *
6  * @author SuBaochen:subaochen@126.com
7 */
8 public class Point {
9
10    int x;
11    int y;
12
13 // Point() {
14 //     x = 10;
15 //     y = 10;
16 // }
17
18    Point(int newX, int newY) {
19        x = newX;
20        y = newY;
21    }
22
23    void setX(int newX) {
24        x = newX;
25    }
26
27    void setY(int newY) {
28        y = newY;
29    }
30
31    void moveTo(int newX, int newY) {
32        setX(newX);
33        setY(newY);
34    }
35 }
```

代码清单 3.11: Circle.java

```
1 package cn.edu.sdut.softlab.oopbasic.sub333;
2
3 public class Circle {
4
5     static final double PI = 3.14159;
6     Point origin = new Point(0, 0); //
7     float radius;
8
9     double area() {
10         return PI * radius * radius;
11     }
12
13 }
```

代码清单 3.12: Rectangle.java

```
1 package cn.edu.sdut.softlab.oopbasic.sub333;
2
3 public class Rectangle {
4
5     Point startPoint = new Point(20, 20); //
6     int width;
7     int height;
8
9     int area() {
10         return width * height;
11     }
12 }
```

代码清单 3.13: Point.java

```
1 package cn.edu.sdut.softlab.oopbasic.sub334;
2
3 /**
4  * 这个例子演示了多个构造方法时的情形.
5  *
6  * @author SuBaochen:subaochen@126.com
7 */
8 public class Point {
9
10    int x;
11    int y;
12
13    Point() {
14        x = 10;
15        y = 10;
16    }
17
18    Point(int newX, int newY) {
19        x = newX;
20        y = newY;
21    }
22
23    void setX(int newX) {
24        x = newX;
25    }
26
27    void setY(int newY) {
28        y = newY;
29    }
30
31    void moveTo(int newX, int newY) {
32        setX(newX);
33        setY(newY);
34    }
35 }
```

代码清单 3.14: Circle.java

```
1 package cn.edu.sdu.tsoftlab.oopbasic.sub334;
2
3 public class Circle {
4
5     static final double PI = 3.14159;
6     Point origin = new Point(0, 0); //
7     float radius;
8
9     double area() {
10        return PI * radius * radius;
11    }
12
13 }
```

3.3.5 再说对象的初始化

通过以上分析我们可以看出，构造方法的目的是为了**创建对象时设置对象的初始状态**，因此一个类需要多少个构造方法并没有一定之规，需要根据实际情况来确定。通常，在一开始设计一个类的时候，建议覆盖默认的构造方法即无参构造方法，提供一个默认的对象状态。随着开发的深入，可以根据实际情况添加更多的构造方法。

除了通过构造方法设置对象的初始状态外，也可以通过以下的方式进行对象的初始化操作：

- 在创建对象后，直接访问对象的属性并设置希望的值。
- 在创建对象后通过 `setter` 方法设置对象的状态。所谓 `setter` 方法，往往是一系列 `setX`, `setY` 方法，其中的 X、Y 为对象的属性。
- 在 JavaEE 环境下，可以提供一个初始化方法，比如 `void init()`，然后通过注解 `@PostConstruct` 的方式实现对象的初始化。

 定义有参构造方法时，要注意参数的个数不要太多，一般不要超过 10 个。太多的参数往往给使用者带来记忆和匹配的负担。如果需要大量的对象初始化操作，建议编写专门的方法初始化对象（比如 `init()` 方法就是一个很好的方法名字），然后在创建对象后调用这个 `init` 方法即可。

3.4 类的组织：包

3.4.1 包的概念

在进行文件管理时，我们使用文件夹来组织不同类型的文件。在 Java 中，我们使用“包”（package）来管理不同类型的 Java 源文件，其实“包”就是文件夹，我们说包 cn.edu.sdut.softlab.oopbasic.sub332 中的 Point.java 时，其实是说，位于目录 cn/edu/sdut/softlab/oopbasic/sub332 目录下的 Point.java。也就是说，把目录的分隔符换成“.”，目录名就变成了包名。图3.3是本章的主要示例代码的目录层级结构：

```
subaochen@subaochen-desktop:~/git/javabook/src/chap03/src$ tree
└── cn
    └── edu
        └── sdut
            └── softlab
                └── chap03
                    ├── sub321
                    │   ├── Draw.java
                    │   └── Point.java
                    ├── sub322
                    │   ├── Circle.java
                    │   ├── Draw.java
                    │   ├── Point.java
                    │   └── Rectangle.java
                    ├── sub33
                    │   ├── Point.java
                    │   └── Rectangle.java
                    ├── sub332
                    │   ├── Circle.java
                    │   ├── Draw.java
                    │   ├── Point.java
                    │   └── Rectangle.java
                    ├── sub333
                    │   ├── Circle.java
                    │   ├── Draw.java
                    │   ├── Point.java
                    │   └── Rectangle.java
                    └── sub334
                        ├── Circle.java
                        ├── Draw.java
                        ├── Point.java
                        └── Rectangle.java
```

图 3.3: 本章示例代码的目录层级结构

3.4.2 包的导入和声明

有了包的概念后，我们就可以使用 **import** 导入其他包中的 Java 源代码，这很像 C 语言中的 **include** 关键字的作用。比如下面的代码片段，我们导入了 `java.util` 包中的 `Date` 类（时间处理类）：

```
1 import java.util.Date;
2
3 public class DateTest {
4     public static void main(String[] args) {
5         Date today = new Date();
6         System.out.println("today is " + today);
7     }
8
9 }
```

关于包的导入 (**import**)，需要注意以下几个方面：

- 如果一个 Java 源代码在一个包中，则这个 Java 源代码应该首先使用 **package** 声明自己所在的包，正如本章所有的例子一样，在代码的开头几乎都一句 **package** 语句声明了包。建议回顾一下本章学过的例子，重点放在包的命名上面，了解如何在 Java 源代码文件中声明包。**只有声明（package）了包的 Java 源代码，才能够被导入进来。**
- 如果 Java 源代码中没有 **package** 声明语句，则编译器认为 Java 文件在**默认包** 中，即 `src` 根目录下，这通常不是一个好的习惯。
- 显然，**import** 应该放在 `class` 的定义前面。
- 在同一个包下的 Java 源代码不需要导入 (**import**)。
- Java 的核心 API，即 `java.lang` 包的类，Java 编译器会自动导入，也不需要我们手工导入。

在 Idea IDE 环境下，在使用到一个其他包的 Java 类时，有两种方式自动导入相对应的包：

- 使用菜单：code->optimize imports，让 Idea 自动导入需要的包。
- 鼠标移动到未识别的类上面，Idea 会给出导入的提示，选择合适的包导入即可。注意，如果 Idea 给出了多个选择（意味着在多个包中都有这个类），此时要认真甄别要导入的包。

 根据 Google 编码规范 (参见节 H.3.3.1 [在第 359 页]), 尽量避免 import package.* 这样的用法。

3.4.3 包的命名原则

我们使用包的目的是把 Java 文件按照逻辑组织起来⁶, 同时也起到了“隔离”的作用, 即不同包中的类即使名字相同也是可以的。或者说, 以后我们提到一个类, 应该连同它的包一起来考虑, 即某某包的某某类。因此, 当我们给包起名字的时候就要谨慎考虑, 最好能够做到你起的包名是全球唯一的, 这样你的包里面的类也就是全球唯一的了。而全球唯一的资源, 很容易就想到了“域名”, 于是通用的和建议的包命名规则⁷是按照域名的反向⁸来命名, 比如你的域名是 lab.company.com, 则包的命名可以是: **com.company.lab**。下面是一些典型的包命名示范:

```
com.google.android  
cn.edu.sdu.java.lesson  
org.apache.log4j
```

 根据 Google 的编码规范 (参见节 H.5.2.1 [在第 368 页]), 包的名字应该都使用小写字母, 并避免使用下划线。

3.5 类的继承

在编程实践中, 我们应该力求消除冗余, 即不存在两片相同的代码。比如下面的代码片段:

```
1 void method1() {  
2     doSomething();  
3     foo();  
4     bar();  
5     foobar();  
6 }  
7  
8 void method2() {  
9     doSomeOtherThing();  
10    foo();  
11    bar();
```

⁶形式上看是这样, 但是包的真正意义和 Java 虚拟机的类加载机制有关, 这里不深入探讨, 有兴趣的读者可以参考 Class-Loader 的相关资料。

⁷参见 JLS: <http://docs.oracle.com/javase/specs/jls/se8/html/jls-6.html#jls-6.1>

⁸为什么要将通常的域名反向排列形成包名呢? 我们阅读的顺序是从左向右的, 当谈到一个组织结构时, 习惯上也是按照从左向右降序排列的, 比如山东省淄博市张店区。因此, 在命名包时把域名的顺序反向过来, 更符合人们的阅读和理解习惯。当然, 这不是 Java 的语法规规定, 只是“约定俗成”, 入乡随俗也体现了一个程序员的素质。

```
12     foobar();  
13 }
```

我们应该这样消除冗余：

```
1 void method1() {  
2     doSomething();  
3     fun();  
4 }  
5  
6 void method2() {  
7     doSomeOtherThing();  
8     fun();  
9 }  
10  
11 void fun() {  
12     foo();  
13     bar();  
14     foobar();  
15 }
```

虽然后一种方式多定义了一个方法 `fun()`，但是好处显而易见：软件的逻辑性（可读性）和可维护性提高了。如果需要对 `fun()` 的流程做任何修改，我们现在只需要在 `fun()` 方法中修改即可。

上面代码消除冗余是建立在方法层面上的，在类的层面上应该如何消除冗余呢？比如有如下的两个类：`BankCard.java` 和 `CreditBankCard.java`。

很明显，`CreditBankCard` 具有 `BankCard` 的大部分属性和方法，从尽力消除冗余的角度看，这样的代码太“丑陋”了！java 提供了“继承”机制解决了这个问题，先看一下代码的实现：`BankCard.java` 的代码没有变化，这里只列出 `CreditCard.java` 的代码，如代码清单3.17所示。

也就是说，`CreditBankCard` 扩展（extends）了 `BankCard` 类：`CreditCard` 自动具有 `BankCard` 的属性和方法，只需要在 `CreditCard` 中定义专属的属性和方法即可。

这就是 Java 等面向对象程序设计语言的“继承”机制。简单的说，**继承机制消除了代码的冗余，提高了代码的复用水平**。继承是面向对象程序设计的重要手段，我们将在节 3.5.1 [在对页] 进一步分析继承在编程实践中的应用和注意事项。

代码清单 3.15: BankCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step1;
2
3 /**
4  * @author SuBaochen:subaochen@126.com
5 */
6 public class BankCard {
7
8     String username;
9     String cardNo;
10    String password;
11
12    void deposit() {
13        // 存钱
14    }
15
16    void withdraw() {
17        // 取钱
18    }
19 }
```

3.5.1 单继承

如图3.4⁹的继承关系中，我们把 BankCard 类称作父类，或者超类，把 CreditCard、DebitCard 类称作子类。CreditCard 和 DebitCard 继承了 BankCard 的属性和方法，并各自扩展了相应的属性和方法（DebitCard 只扩展了属性，并没有扩展方法）。

 Java 只支持单继承，即一个类只允许有一个父类。绕过这个限制的办法是使用接口，请参见节 4.2 [在第 108 页]

3.5.2 再说构造方法

在节 3.3 [在第 46 页] 中，我们已经看到创建对象的第二步是递归的构造父类对象。我们仍然以银行卡为例来看一下所谓的“递归的构造父类对象”的含义。为了清楚期间，我们增加了一个类的层级结构，如图3.5所示。

从代码清单3.18可以看出，Card 类只定义了所有卡都有的基本属性：卡号。另外，显式定义了无参的构造方法，如果被调用的话，只是打印出一条提示信息。

⁹这样表示类层次关系的图叫做 UML 的类图。UML (Universal Model Language, 统一建模语言) 是 IT 行业中通用的描述需求和系统设计的建模工具，其中常用的图形有：类图、用例图、顺序图、活动图等，本书中只涉及到类图，用来表达类的继承和接口的实现。类图的意义应该是一目了然的，箭头表示类继承的方向，比如图3.4中，CreditCard 指向了 BankCard 即表示 CreditCard 是从 BankCard 继承下来的。绘制 UML 图的工具有很多，本书采用的是 umbrello 和

代码清单 3.16: 未使用继承的 CreditCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step1;
2
3 import java.util.Date;
4
5 /**
6  * @author SuBaochen:subaochen@126.com
7 */
8 public class CreditCard {
9
10    String username;
11    String cardNo;
12    String password;
13    Date expired;
14
15    void deposit() {
16        // 存钱
17    }
18
19    void withdraw() {
20        // 取钱
21    }
22
23    void overdraw() {
24        // 透支消费
25    }
26 }
```

代码清单 3.17: 使用了继承的 CreditCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step2;
2
3 import java.util.Date;
4
5 public class CreditCard extends BankCard {
6     Date expired;
7
8     void overdraw() {
9         // 透支消费
10    }
11
12 }
```

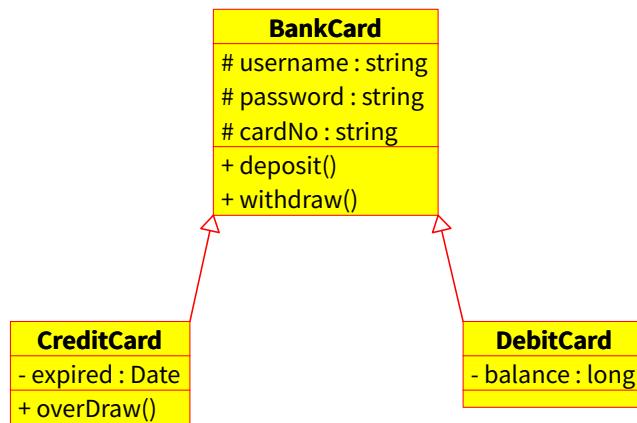


图 3.4: BankCard 的类层次结构

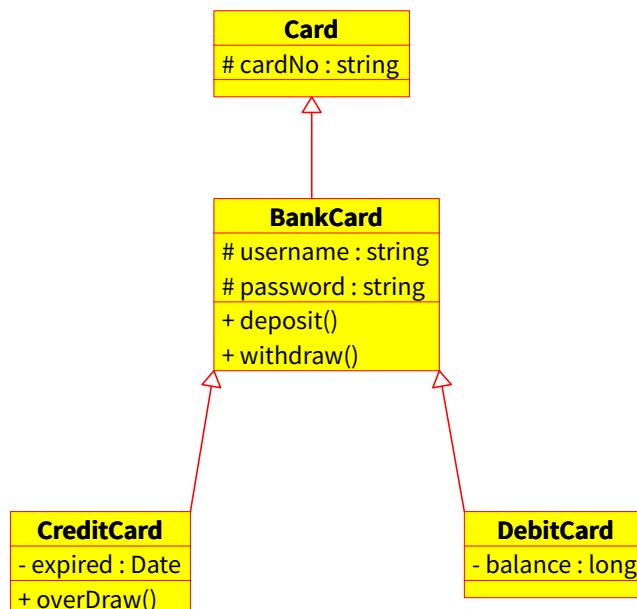


图 3.5: 银行卡的类层级结构

代码清单 3.18: 增加无参构造方法的 Card.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step3;
2
3 /**
4  * @author SuBaochen:subaochen@126.com
5 */
6 public class Card {
7
8     String cardNo;
9
10    Card() {
11        System.out.println("Card constructor called");
12    }
13 }
```

BankCard、CreditCard、DebitCard 的设计思路类似，都是增加了一个无参的构造方法，打印出一条提示信息，如果被调用到的话。在 Client 类中，我们分别创建了 CreditCard 和 DebitCard 对象，注意观察构造方法的调用顺序。

执行 Client.java 的结果如下：

```
Card constructor called
BankCard constructor called
CreditCard constructor called
Card constructor called
BankCard constructor called
DebitCard constuctor called
```

也就是说，所谓的“递归的构造父类对象”是指：**当初始化子类对象时，从上往下依次调用父类的同名构造方法**，如图3.6所示。



在递归构造父类对象的“链条”中，如果其中一环缺失会怎样？请读者自行练习找到此问题的答案：将导致语法错误。

3.5.3 方法的覆盖和重载

方法的覆盖 (override) 和重载 (overload) 是面向对象编程中两个重要且容易混淆的概念，因此这里单独拿出来讨论一下。

dia，参见本书前言部分的说明。

代码清单 3.19: 带有无参构造方法的 BankCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step3;
2
3 public class BankCard extends Card {
4
5     String username;
6     String password;
7
8     BankCard() {
9         System.out.println("BankCard constructor called");
10    }
11
12    void deposit() {
13        // 存钱
14    }
15
16    void withdraw() {
17        // 取钱
18    }
19 }
```

代码清单 3.20: 带有无参构造方法的 CreditCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step3;
2
3 import java.util.Date;
4
5 public class CreditCard extends BankCard {
6
7     Date expired;
8
9     CreditCard() {
10         System.out.println("CreditCard constructor called");
11    }
12
13    void overdraw() {
14        // 透支消费
15    }
16
17 }
```

代码清单 3.21: 带有无参构造方法的 DebitCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step3;
2
3
4 public class DebitCard extends BankCard {
5     float balance;
6
7     DebitCard() {
8         System.out.println("DebitCard constructor called");
9     }
10 }
```

代码清单 3.22: Client.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step3;
2
3 public class Client {
4
5     public static void main(String[] args) {
6         CreditCard card1 = new CreditCard();
7         DebitCard card2 = new DebitCard();
8     }
9
10 }
```

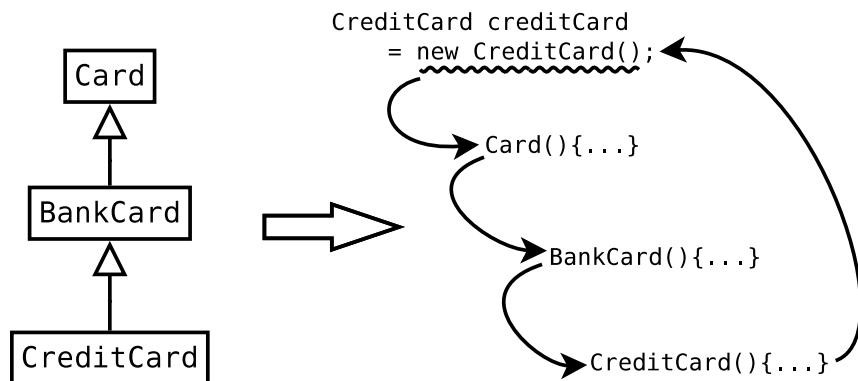


图 3.6: 构造方法的递归调用

3.5.3.1 方法重载

Java 中的方法重载 (overload) 是指在一个类中允许存在多个名字相同但是参数不同的方法。这里所谓的“参数不同”有如下的几种情形：

- 参数个数不同
- 参数类型不同
- 参数个数和类型都不相同

如果我们把方法的参数看做一个控制系统的“输入条件”，那么方法的重载意味着同一个动作在不同的输入条件下的表现如何？

例 3.5. 使用方法重载设计一个计算器类。

代码设计 参见代码清单9.1。

代码清单 3.23: Caculator.java

```
1 package cn.edu.sdut.softlab.oopbasic;
2
3 /**
4  * Created by subaochen on 17-1-14.
5 */
6 public class Caculator {
7     void sum(int a, int b) {
8         System.out.println(a + b);
9     }
10
11    void sum(int a, int b, int c) { // ①
12        System.out.println(a + b + c);
13    }
14
15    void sum(double a, double b) { // ②
16        System.out.println(a + b);
17    }
18
19    // void sum(int x, int y) {} // ③
20
21    void sum(int a, double b) {
22        System.out.println(a + b);
23    }
24
25    void sum(double a, int b) { // ④
26        System.out.println(a + b);
27    }
28
29    public static void main(String[] args) {
30        Caculator cal = new Caculator();
```

```

31     cal.sum(10, 10);
32     cal.sum(10, 10, 10);
33 }
34 } //
```

- ① 参数个数不同
 ② 参数个数相同，但是参数类型不同
 ③ 只是参数名字不同是不可以的！
 ④ 参数个数和类型相同，但是顺序不同是可以的

Java 中典型的方法重载场合：

- 构造方法的重载：我们在 3.3.4 [在第 50 页] 中看到，可以编写类的多个构造方法，实际上就是方法的重载的具体体现。构造方法的名字都是相同的（和类名一致），不同的构造方法（参数不同）表示在不同的条件下如何创建对象。
- 运算符重载：Java 的“+”运算符是被重载了的。我们前面已经看到，“+”运算符在不同的表达式中意义不同，用于算数表达式时表示数学运算的加法，用于两个字符串时表示首尾相接连接两个字符串。
- `String.valueOf` 方法可以接收的参数有 `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long` 等¹⁰。

需要注意的是，只是参数的顺序不同以及参数的名字不同不能构成方法的重载，原因很简单，我们调用方法的时候进行形参和实参的匹配，顺序和名字并不能唯一决定形参和实参的组合。比如方法 `method(String a, String b)` 和 `method(String c, String d)` 不能构成方法的重载，在 Java 编译器看来，这是两个相同的方法，因而会报告编译错误。

举个生活中的例子可能更容易理解方法重载的概念和用途，比如我们知道插座有各种规格以便适用不同的电器，有的电器使用两孔插头，有的电器使用三孔插头，插头还可能分为圆形插头和扁平插头等等。正如图 3.7 所示，生活中我们经常安装一种所谓的“多用插座”，即一个插座可以使用多种不同的插头，这样就免去安装很多个不同的插座的麻烦，也不需要大家记住插头和插座的对应关系，直接拿来用就可以了。这种“多用插座”的设计移植到程序设计上，就叫做“方法重载”（overload），即方法的名字相同（多用插座），但是参数各不相同（适用

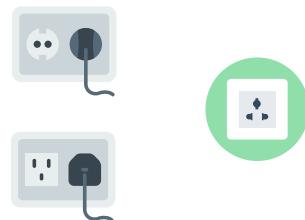


图 3.7：生活中的各种插座和多用插座示例

¹⁰ 我们将在节 5.1 [在第 125 页] 详细讨论 `String` 类的用法，`String.valueOf` 方法的形式可参考 JDK API，这里暂不详细讨论。

 不同的插头), 在具体使用时, 程序员只需要记住少量的方法的名字, 参数类型和个数由 IDE 自动提示即可。

3.5.3.2 方法覆盖

方法覆盖 (override) 是指子类中重新实现了父类中的同名方法, 即子类方法覆盖了父类方法。这样, 当调用子类对象的这个方法时, 就是使用子类重新实现的方法了。否则, 就会调用父类中的方法。因此, 方法覆盖的主要用意是在子类中提供父类方法的加强版, 或者使用不同的逻辑重新实现。

例 3.6. 方法覆盖示例。

代码设计 参见代码清单3.24。

代码清单 3.24: Dog.java

```
1 package cn.edu.sdu.tsoftlab.oopbasic;
2
3 /**
4  * Created by subaochen on 17-1-14.
5 */
6 public class Dog extends Animal {
7     public void hello() {
8         System.out.println("dog say hello.");
9     }
10
11    public static void main(String[] args) {
12        Dog dog = new Dog();
13        dog.hello();
14    }
15 }
16
17
18 class Animal {
19     public void hello() {
20         System.out.println("animal say hello.");
21     }
22 }
```

方法覆盖是 Java 多态的基础, 具体请参见节 4.3 [在第 116 页]。

 注意到从 JDK 1.5 开始, Java 允许子类覆盖父类方法时, 返回值可以是父类方法返回值的子类。比如:

```
1 class A{
2     Father get(){ return null;}
3 }
```

```

4 public class B extends A{
5     @Override
6     Son get(){ return null;}
7 }
8 class Father{}
9 class Son extends Father{}

```

在类 B 中，方法 get 返回值类型是 Son，而其父类 A 的 get 方法的返回值类型是 Father，由于 Son 是 Father 的子类，因此类 B 的方法 get 也属于方法的覆盖（override）。

Java 是如何做到这一点的呢？我们反编译 B.class 会发现：

```

1 // Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
2 // Jad home page: http://www.geocities.com/kpdus/jad.html
3 // Decompiler options: packimports(3)
4 // Source File Name: B.java
5 public class B extends A{
6     public B() {
7     }
8
9     Son get(){
10         return null;
11     }
12
13     volatile Father get(){ // 桥接方法
14         return get();
15     }
16 }

```

可以看出，编译器自动增加了一个所谓的“桥接方法”实现了标准的方法覆盖。

3.6 访问控制

当我们说到面向对象程序设计的封装时，不仅是指把对象的属性和方法“捏合”在了一起，也指从使用者的角度看，访问对象是受到限制的，即对象内部的状态和方法并不总是需要暴露给使用者的。推而广之，甚至对类的访问也可以加以限制。

3.6.1 属性和方法的访问控制

为什么需要对属性和方法进行访问控制呢？从面向对象的设计角度看，一个对象完成了特定的功能，对象的使用者其实并不关心这个特定功能是如何实现的，只要这个功能能够正常工作就可以了，这就是所谓的“松耦合”（loose coupling principle）原则。考虑图3.8所示的情形。

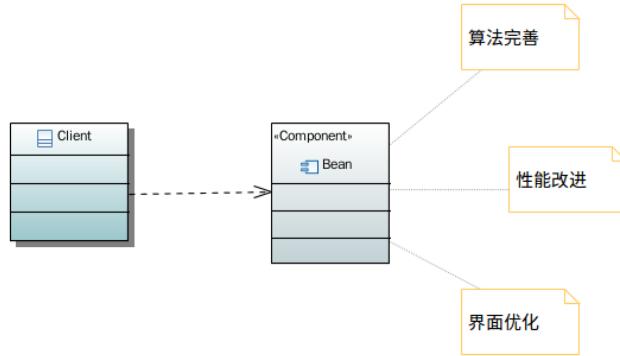


图 3.8: 松耦合原则示意图

当 Client 使用 Bean 组件（对象）提供的功能时，并不需要知晓 Bean 组件内部是如何实现的，也就是说，Bean 组件在后续进行算法完善、性能改进或者界面优化... 时，Client 的代码不需要做任何改动。Bean 组件只需要对外（Client）暴露必须的调用接口（如何暴露，我们会在节 4.2 [在第 108 页] 讨论），其内部实现及只在内部实现中用到的一些对象状态，应该想办法隐藏起来。换句话说，Bean 组件如果对外暴露的调用接口越多，则后续改进时要考虑的因素就越多。

Java 提供了完善的访问控制策略来保护属性和方法，如表所示¹¹（形成了一个三角形排列）。

| 修饰词 | Class (类) | Package (包) | Subclass (子类) | World (其他) |
|-----------|-----------|-------------|---------------|------------|
| public | √ | √ | √ | √ |
| protected | √ | √ | √ | X |
| default | √ | √ | X | X |
| private | √ | X | X | X |

表 3.1: Java 的访问控制修饰符

3.6.1.1 default 修饰符

在之前我们的代码中，属性和方法前面没有使用任何修饰符，即默认修饰符。对照表 3.1 可以看出，default 修饰的（即不使用任何修饰符）的属性和方法对同一个包内的其他类是可见的：即可以读，也可以修改。因此，default 修饰符也被称为“包范围”的修饰符，它以包为界限定了访问权限，这是一种很自然的权限划分，因此被 Java 确定为“default”（默认）情形。

¹¹ 也参见：<http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

代码清单 3.25: 演示 default 修饰符的 BankCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.ac;
2
3 public class BankCard {
4
5     String username;
6     String password;
7
8     BankCard() {
9         System.out.println("BankCard constructor called");
10    }
11
12    void deposit() {
13        // 存钱
14    }
15
16    void withdraw() {
17        // 取钱
18        System.out.println("钱被取走啦!");
19    }
20 }
```

 default 的意思是默认修饰符，即没有在属性和方法前面指定修饰符的情形，并非要显式的指定一个“default”的修饰符。实际上，java 中不存在一个名字叫做“default”的修饰符。因此，下文中的 default 均指：在属性、方法、类前面不使用任何访问控制修饰符。

在代码清单3.25和代码清单3.26中，BankCard 类中的 username 和 password 属性以及 withdraw 方法都是 default 的，因此同一个包中的 Client 类可以随意修改密码，随意取钱：这太不安全了！

执行结果如下：

```
BankCard constructor called
钱被取走啦!
```

 default 修饰符意味着同一个包内的其他类都可以随意操纵你的属性和方法，这是一个危险的设置！设想一下，黑客把一个自己写的破坏性 Java 文件放到你的包里面，然后修改你的银行卡密码，盗取你的钱财... 因此，default 修饰符给软件的安全留了一个后门，并不是一个好的“默认值”！

代码清单 3.26: Client.java

```
1 package cn.edu.sdut.softlab.oopbasic.ac;
2
3 public class Client {
4
5     public static void main(String[] args) {
6         BankCard card = new BankCard();
7         card.username = "zhangsan";
8         card.password = "123456"; // 哇塞，可以直接修改密码啊！
9
10        card.withdraw(); // 取钱畅通无阻！
11        card.deposit(); // 除了爹妈，还有谁往我的卡里存钱呢？！
12    }
13
14 }
```

3.6.1.2 private 修饰符

default 给了同一个包下的其他类“篡改”本类的机会，private 修饰符则完全杜绝了这个问题：从表3.1可以看出，private 修饰过的属性和方法只有本类可以访问，即只有本类可以读取和修改，对同一个包下的其他类、子类以及所有其他的类都不可见。正如 private 的名字所宣示的，这是一个完全隐私的保护性设置，特别适合于本节开头所述的“松耦合原则”中组件（对象）内部属性、方法的表达。实际上，在绝大多数情况下，我们应该将类的属性和方法设置为 **private**，然后根据需要或者设计要求逐步放开限制，以达到对组件（对象）最大程度的封装和保护。

有时我们希望对外一定程度上暴露私有的属性，即不直接暴露私有 (private) 属性，而是通过一个公有 (public) 的方法间接的访问私有属性。这样如果需要的话，我们可以在这个方法中对属性修改做合法性检查，避免任意的篡改私有属性。这些能够读取和设置私有属性的方法往往以 get 和 set 开头，一般称作 **getter/setter** 方法，比如下面的代码片段：

```
1 public class Person {
2     private String username;
3     private String password;
4     private String email;
5
6     public void setUsername(String newUsername) {
7         username = newUsername;
8     }
9
10    public String getUsername() {
11        return username;
12    }
13 }
```

代码清单 3.27: 演示 private 修饰符的 BankCard.java

```
1 package cn.edu.sdu.tsoftlab.oopbasic.acprivate;
2
3 public class BankCard {
4
5     private String username;
6     private String password;
7
8     BankCard() {
9         System.out.println("BankCard constructor called");
10    }
11
12    void deposit() {
13        // 存钱
14    }
15
16    void withdraw() {
17        // 取钱
18        System.out.println("钱被取走啦!");
19    }
20 }
```

```
14     // setPassword/getPassword等方法和getUsername/setUsername类似
15 }
```

 大多数 IDE 开发环境可以帮助我们自动生成这些 getter/setter 方法：在类中右键选择“Insert Code...”，然后选择“Getter and Setter...”，选中希望自动生成 getter/setter 方法的私有属性，可以多选，然后点击“Generate”按钮即可。自动产生的 getter/setter 代码中使用了 this 关键字，我们将在节 4.3.1 [在第 121 页] 讨论它。

例 3.7. 改造节 3.6.1.1 [在第 69 页] 中的例子，使得 BankCard 类中的 username 和 password 都是 private 的。

代码设计 参见代码清单3.27和代码清单3.28。

运行结果 可以看出，我们在 Client 中无法直接修改密码了，否则会报告语法错误。

代码分析 BankCard 类的两个属性使用 private 修饰符后，Client 类无法直接访问这两个属性了。可以通过提供 getter/setter 方法，方便 Client 访问 BankCard 的这

代码清单 3.28: Client.java

```
1 package cn.edu.sdut.softlab.oopbasic.acprivate;
2
3 public class Client {
4
5     public static void main(String[] args) {
6         BankCard card = new BankCard();
7         //card.username = "zhangsan";
8         //card.password = "123456"; // 不可直接修改密码
9
10        card.withdraw(); // 取钱畅通无阻!
11        card.deposit(); // 除了爹妈，还有谁往我的卡里存钱呢？!
12    }
13
14 }
```

两个属性。请读者试着在 `BankCard` 中加上 `getter/setter` 方法，并在 `Client` 中调用 `getter/setter` 修改用户名和密码。

问题 3.1. 构造方法是私有的会怎样？为什么？

我们在 `Client` 类中使用 `new BankCard()` 创建对象时会试图调用 `BankCard` 的构造方法，但是此时 `BankCard` 的构造方法是私有的，意味着对同一个包下面的 `Client` 不可见！因此，如果使用 `private` 修饰构造方法，意味着你不希望别人直接通过 `new` 操作符创建对象。那么这个类还有啥用处呢？实际上，在这种情况下，可以通过提供一个方法来创建对象，比如下面的代码片段：

```
1 public class BankCard {
2     ...
3     private BankCard () {}
4
5     public static BankCard getInstance() {
6         new BankCard();
7     }
}
```

在本类中 `new BankCard()` 当然是没有问题的了，所以通过一个公有的方法 `getInstance()` 来创建对象，则 `Client` 就可以这样使用：

```
1 public class Client {
2     ...
3     BankCard card = BankCard.getInstance();
4 }
```

此种用法常见于工厂模式 [5]、单例模式 [5] 等。

代码清单 3.29: 演示 `protected` 修饰符的 `BankCard.java`

```

1 package cn.edu.sdut.softlab.oopbasic.acprotected;
2
3 public class BankCard {
4
5     protected String username;
6     protected String password;
7
8     public BankCard() {
9         System.out.println("BankCard constructor called");
10    }
11
12    protected void deposit() {
13        // 存钱
14    }
15
16    protected void withdraw() {
17        // 取钱
18        System.out.println("钱被取走啦!");
19    }
20 }
```

3.6.1.3 `protected` 修饰符

`private` 修饰过的属性和方法只有本类可以访问, `default` 修饰过的属性和方法本类和本包其他类可以访问, `protected` 修饰过的属性和方法则进一步放宽了限制: 除了本类、本包之外, 子类也可以访问。或者说, 我们使用 `protected` 修饰属性和方法的唯一目的就是: **子类可见**, 这也是 `protected` 这个修饰符的本意: 受保护的本意大概就是家族内无本质纷争的意思, 所谓“兄弟阋于墙, 外御其辱”, 可以理解为一家人不见外, 不是一家人就对不起了, 不可见。因此, `protected` 修饰的属性和方法本人可见(这个很自然), 同包内的其他类(同胞)可见, 子类(对自己的下一代不隐瞒, 即使下一代不在同一个包内)可见, 其他外族(其他包中的类)则不可见。当然, 节 3.6.1.4 [在第 76 页] 的 `public` 修饰过的属性和方法在子类也是可见的, 但是 `public` 往往太宽泛, 实际应用的时候应该严格限制。

例 3.8. 使用 `protected` 改造银行卡的例子, 观察 `protected` 是如何保护属性和方法的¹²。

代码设计 参见代码清单3.29和代码清单3.30。

¹²完整代码参见<https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/acprotected/other>

代码清单 3.30: 演示 protected 修饰符的 CreditCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.acprotected.other;
2
3 import cn.edu.sdut.softlab.oopbasic.acprotected.BankCard;
4
5 public class CreditCard extends BankCard {
6
7 }
```

代码清单 3.31: Test.java

```
1 package cn.edu.sdut.softlab.oopbasic.acprotected.other;
2
3 import cn.edu.sdut.softlab.oopbasic.acprotected.BankCard;
4
5 public class Test {
6     void test() {
7         BankCard card = new BankCard();
8         // card.password = "123456"; // no way!
9
10    }
11 }
```

运行结果 运行 Client.java 得到如下结果：

BankCard constructor called

钱被取走啦!

代码分析 在 BankCard 中, 我们将两个属性和两个方法均使用 protected 修饰, 因此其子类 CreditCard 能够直接访问这两个属性和方法。注意到 CreditCard 和 BankCard 不在一个包内, 这并不影响 CreditCard 访问父类 BankCard 中 protected 修饰过的属性和方法, 也就是说, Java 首先根据“父子”关系检查访问权限, 然后再根据包内还是包外检查访问权限。

代码清单3.31中创建了一个 BankCard 对象, 但是由于 Test 类和 BankCard 不在同一包内, 也不是 BankCard 的子类, 因此 Test 中创建的 BankCard 对象无法直接访问 protected 修饰过的属性和方法。

3.6.1.4 public 修饰符

public 修饰符当然就是无限制的意思了，在属性和方法的访问控制中，**public** 修饰符应该谨慎使用，只有确定需要公开的情形才使用 **public** 修饰属性和方法，否则和 C 语言有啥区别呢？请记住，**public** 意味着公开的承诺，一旦你使用了 **public** 修饰属性和方法，就意味着你失去了后续修改这部分代码的机会。

虽然在属性和方法上要谨慎使用 **public** 修饰符，但是构造方法往往是 **public** 的，这是因为绝大多数情况下，我们编写了一个类是希望任何人都能够创建相应的对象的。试想，如果构造方法是 **protected**、**default** 甚至是 **private** 的，会怎样？这里列出四种修饰符修饰构造方法的情形：

- **public** 的构造方法：任何人都可以通过 new 创建对象
- **protected** 的构造方法：只有本类、子类和同一个包内的类可以通过 new 创建对象
- **default** 的构造方法：只有本类、同一个包内的类可以通过 new 创建对象
- **private** 的构造方法：只有本类可以通过 new 创建对象

3.6.1.5 属性和方法的访问控制小结

图3.9可以帮助我们理解 **public/protected/private** 修饰符的保护范围。

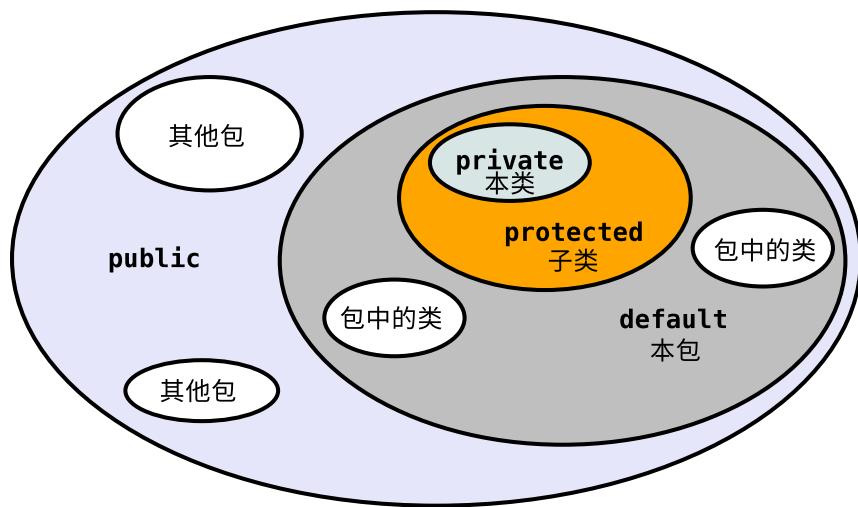


图 3.9：访问控制保护范围示意

在编程实践中，我们往往根据以下的原则来确定使用什么修饰符控制属性和方法的访问权限：

- 首先考虑使用最严格的访问控制策略，即优先考虑使用 `private` 修饰属性和方法，除非你有足够的理由，比如希望子类可见，则选择使用 `protected` 修饰符。
- 尽量避免 `default` 修饰符，即不使用任何修饰符，这是一个危险和不好的编程习惯。
- 尽量避免 `public` 修饰符，这比 `default` 更危险！有两个例外：一是构造方法一般是 `public` 的，理由在节 3.6.1.4 [在对页] 已经讲过了。二是常量一般也是 `public` 的。

3.6.2 类的访问控制

属性和方法的访问控制有 4 个级别，类的访问控制则简单的多，只有两个级别：

1. `public`: `public` 修饰的类，在任何包中都可见。
2. `defualt`: `default` 修饰（即没有任何修饰符）的类，只在本包中可见。

在一个 Java 源文件中，只有一个类可以是 `public` 的，而且这个 `public` 的类的类名和文件名要严格一致。在这个 Java 源文件中，除了这个 `public` 的类之外，依然可以定义其他的 `default` 修饰（即没有使用任何修饰符）的类。但是一般情况下不建议这样做，应该每个类建立单独的 Java 源文件。原因是我们一般是通过浏览这个包（目录）了解这个包中包含了哪些类，如果在一个源文件中包含了多个类，一方面我们无法从文件的目录了解包的内容，另一方面，这些隐藏在其他类文件中的类就只能限于包内访问了。一个例外是内部类，参见节 3.6.3。

3.6.3 内部类

有的时候，我们为了防止命名冲突，或者为了安全起见，定义一个类只希望局限在某个类内使用，这样的类就是内部类（Inner Class）：**定义在一个类内部的类**，比如：

```
1 class OuterClass {  
2     ...  
3     class InnerClass() {  
4         ...  
5     }  
6 }
```

可以把内部类理解为在类内部定义的第三种资源，其他两种是属性和方法，因此类比属性和方法的用法，我们可以这样使用内部类：

```
1 OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

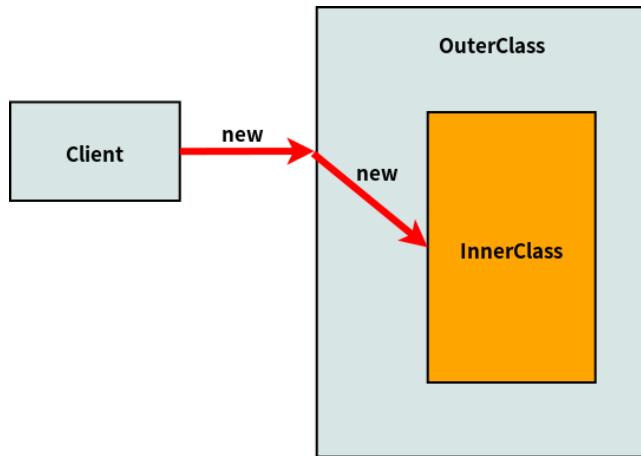


图 3.10: 内部类的创建过程

代码清单 3.32: NotePad.java

```

1 package cn.edu.sdut.softlab.oopbasic.ac.innerclass;
2
3 public class NotePad {
4
5     String content;
6
7     class Editor {
8
9         void parseContent() {
10            System.out.println("parsing:" + content);
11        }
12    }
13 }
  
```

图3.10形象的表示了内部类的创建过程。

例 3.9. 编写 NotePad 类，使用内部类 Editor 实现编辑功能¹³。

代码设计 参见代码清单3.32和代码清单3.33。

运行结果 运行 Client 结果如下：

parsing:test string

¹³完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/ac/innerclass>

代码清单 3.33: Client.java

```
1 package cn.edu.sdu.tsoftlab.oopbasic.ac.innerclass;
2
3 public class Client {
4
5     public static void main(String[] args) {
6         NotePad note = new NotePad();
7         note.content = "test string";
8         NotePad.Editor editor = note.new Editor();
9         editor.parseContent();
10    }
11
12 }
```

代码分析 通过这个例子我们要注意到两点：

- 如何创建一个内部类对象：首先要创建一个外部类对象，然后才能创建一个内部类对象。
- 内部类对象可以访问外部包围类（即内部类是在这个外部包围类中定义的）的属性和方法，即使这个外部类的属性和方法是 `private` 的（请读者试着把 `content` 属性设置为 `private` 看看会怎样？试着把 `parseContent` 方法设置为 `private` 看看会怎样？）。这个很容易理解：内部类和外部类的属性和方法是平等的，就像是本类的方法可以无限制访问本类的属性一样，本类的内部类当然可以无限制访问本类的属性和方法。

也许你会说，`NotePad` 这个例子中，使用内部类 `Editor` 实现 `parseContent` 这个功能太矫情，根本没有必要！是的，在这种简单的情况下确实没有必要绕道使用内部类。但是设想 `NotePad` 不仅仅需要提供文本编辑，还需要文本搜索、格式转换等等其他功能，而且每类功能不止包含一个方法，那么为了逻辑上清晰起见，使用内部类将文本编辑相关的方法组织为内部类 `NotePad.Editor`，将文本搜索相关的方法组织为内部类 `NotePad.Search`，将格式转换相关的方法组织为内部类 `NotePad.Transfer` 就容易理解了。

3.7 引用类型

很多人会说，Java 语言比 C 语言更“进步”或者“安全”的一个原因是 Java 没有指针数据类型，其实这是不确切的。Java 的确没有像 C 语言那样提供“指针”这种基本数据类型，但是 Java 确实使用了指针，而且指针的正确使用同样很重要。Java 中的指针叫做“引用”。

和 C 语言一样，我们可以从变量在内存中的存储来理解 Java 的引用（指针）。Java 的基本数据类型所形成的变量不是引用，变量的名字直接代表了变量所在存储单元的内容，这和 C 语言中的非指针变量是一样的。除此之外，Java 中的数组、对象、Enum 等的名字，都是引用（指针），如图3.11所示。

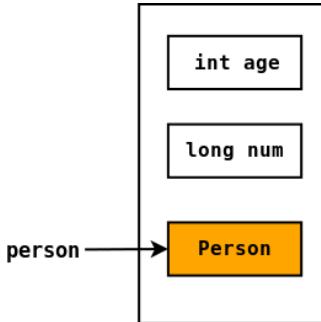


图 3.11: Java 中的引用

比如下面的代码：

```
1 Person person = new Person();
```

之前我们把 `person` 叫做“`Person`”类型的对象，其实这是不严谨的，应该叫做“`Peron`”类型对象的名字。也就是说，`person` 仅仅是一个对象的名字，这个名字其实是指向 `person` 对象的一个引用（指针）。因此下面的代码：

```
1 Person person;
2 person.goHome();
```

就会报“`NullPointerException`”错误，即“空指针异常”。这里只是声明了一个 `person` 类型的引用，但是并没有初始化，也就是说，`person` 并没有指向任何实质的 `Person` 类型的对象。因此在初始化之前，任何对象的引用都是 `null`，这和 C 语言的指针含义完全一样：在 C 语言中，任何指针在初始化之前都是 `null` 指针，直接使用 `null` 指针是有风险的。而在 Java 中，Java 虚拟机会自动侦测空指针，任何对空指针（未初始化的对象引用）都会报告 `NullPointerException` 异常。`NullPointerException` 异常如此常见，通常大家都把 `NullPointException` 异常简称为“**NPE**”。

在 C 语言中，指针作为函数的参数是一个强大的特性，即可以帮助把大量的数据传入函数（输入参数），也可以帮助函数返回大量的数据（输出参数）。在 Java 中，我们也要区分方法的参数为简单数据类型和对象（引用）两种情况。

例 3.10. 使用对象作为方法的参数。

代码设计 我们首先定义一个简单的 `Person` 类，参见代码清单3.34。

测试类 `Client.java` 的代码参见代码清单3.35。

代码清单 3.34: Person.java

```
1 package cn.edu.sdut.softlab.oopbasic.objref;
2
3 public class Person {
4
5     String username;
6     String password;
7     int seniority; // 工龄
8     float salary; // 薪水
9
10    public Person(String username, String password) {
11        this.username = username;
12        this.password = password;
13    }
14
15    public String toString() {
16        return "Person[username=" + username + ",password=" + password
17                + ",seniority = " + seniority + ",salary = " + salary + "]";
18    }
19
20 }
```

运行结果 Client 的运行结果如下：

```
Person[username=zhangsan,password=123456,seniority = 20,salary =
5000.0]
Person[username=zhangsan,password=123456,seniority = 20,salary =
6000.0]
```

代码说明 可以看出, `adjustSalary` 方法中改变了 `person` 对象的属性 `slary` 的值, 这和 C 语言中使用指针作为函数参数的作用如出一辙。

例 3.11. 两个引用指向同一个对象。

代码设计 我们借用例3.10中定义的 Person 类设计一个测试类, 参见代码清单3.36。

代码清单 3.36: Client1.java

```
1 package cn.edu.sdut.softlab.oopbasic.objref;
2
3 /**
4  * Created by subaochen on 17-2-26.
```

代码清单 3.35: Client.java

```
1 package cn.edu.sdut.softlab.oopbasic.objref;
2
3 public class Client {
4
5     public static void main(String[] args) {
6         Person person = new Person("zhangsan", "123456");
7         person.seniority = 20;
8         person.salary = 5000;
9         System.out.println(person);
10
11     adjustSalary(person); // ❶
12     System.out.println(person);
13 }
14
15 public static void adjustSalary(Person person) { // ❷
16     if (person.seniority > 10) {
17         person.salary += 1000; // ❸
18     }
19 }
20 }
21 //
```

❶ 对象作为函数的参数，实际上是将对象的引用（指针）传递给方法

❷ 形参和实参都是对象（引用），指向同一个对象

❸ 这里修改的是形参引用指向的 `person` 对象的 `salary` 属性，但是记得形参和实参指向了同一个对象！

```

5  */
6 public class Client1 {
7     public static void main(String[] args) {
8         Person zhangsan1 = new Person("zhangsan", "123");
9         Person zhangsan2 = new Person("zhangsan", "123");
10        Person zhangsan3 = zhangsan1;
11        System.out.println("zhangsan1 == zhangsan2 ? " + (zhangsan1 == zhangsan2));
12        System.out.println("zhangsan3 == zhangsan1 ? " + (zhangsan3 == zhangsan1));
13    }
14 }

```

运行结果 运行 Client1 的结果如下：

```

zhangsan1 == zhangsan2 ? false
zhangsan3 == zhangsan1 ? true

```

代码说明 在例3.11中，看起来 zhangsan1 和 zhangsan2 两个对象是一样的，有相同的名字和密码设置。但是，这是两个不同的对象，即不同的引用，在内存中有不同的地址。而 zhangsan3 指向了 zhangsan1，即 zhangsan3 和 zhangsan1 指向了同一个对象，因此 zhangsan1 和 zhangsan3 是相等的，如图3.12所示。

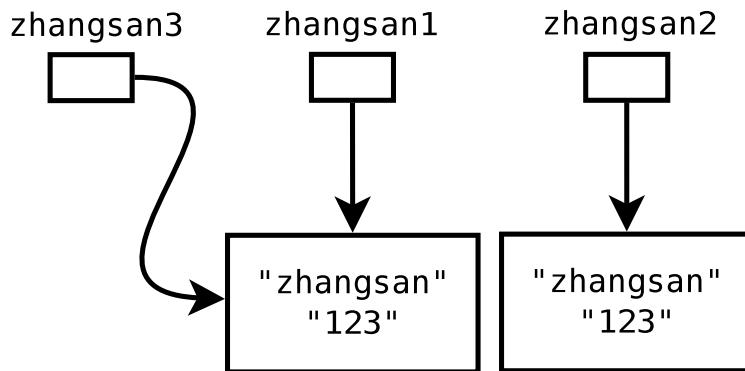


图 3.12: 对象的引用示意图

3.7.1 this

Java 使用 this 关键字表示“对象本身”，即指向当前对象的引用。比如在代码清单3.34中，构造方法是这样写的：

```

1  public Person(String username, String password) {
2      this.username = username;
3      this.password = password;

```

```
4 }
```

其中, `this.username` 即指当前对象的 `username` 属性, 这样就把作为参数的 `username` 和作为对象属性的 `username` 区分开了。

3.7.2 super

我们在节 3.3 [在第 46 页] 中已经看到, 对象在创建时会“递归的构造父类对象”, 也就是说, 递归的调用父类的构造方法, 这并不需要我们在子类的构造方法中做什么。但是, 有时候我们希望能够控制对象的构造过程, 也就是说, 打破 Java 默认的对象构造过程, 在构造对象的链条中加入自定义的成分, 这个时候我们可以使用 `super` 关键字: 指向父类对象的引用。我们通过一个例子来说明 `super` 在对象的构造过程中是如何起作用的。

例 3.12. 使用 `super` 自定义对象的构造过程¹⁴。

代码设计 限于篇幅, 这里只列出了 `CreditCard.java` 的完整代码, 参见代码清单 3.37。

运行结果和分析 我们可以分三种情况运行此例, 分别在 `CreditCard` 类的构造方法 `CreditCard(String cardNo)` 中:

1. 调用 `super()` 时, 输出如下:

```
Card constructor called
BankCard constructor called
CreditCard constructor called,cardNo=123
=====
Card constructor called,cardNo=456
BankCard constructor called,cardNo=456
DebitCard constuctor called,cardNo=456
```

可以看出, 调用 `super()` 会触发“递归的构造父类对象”。

2. 调用 `super(cardNo)` 时, 输出如下:

¹⁴ 完整代码 参见: <https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/inherit/step4>

代码清单 3.37: CreditCard.java

```
1 package cn.edu.sdut.softlab.oopbasic.inherit.step4;
2
3 import java.util.Date;
4
5 public class CreditCard extends BankCard {
6
7     Date expired;
8
9     CreditCard() {
10         System.out.println("CreditCard constructor called");
11     }
12
13     public CreditCard(String cardNo) {
14         super(); // ①
15         // super(cardNo); // ②
16         System.out.println("CreditCard constructor called,cardNo=" + cardNo);
17     }
18
19     void overdraw() {
20         // 透支消费
21     }
22
23 }
24 //
```

- ① 通过 `super` 调用父类的不同构造方法来控制对象的创建过程：调用 `super()` 即调用父类的无参构造方法
② 通过 `super` 调用父类的不同构造方法来控制对象的创建过程：调用 `super(cardNo)` 即调用父类的参数类型和个数相同的构造方法

```
Card constructor called,cardNo=123
BankCard constructor called,cardNo=123
CreditCard constructor called,cardNo=123
=====
Card constructor called,cardNo=456
BankCard constructor called,cardNo=456
DebitCard constuctor called,cardNo=456
```

可以看出，调用 `super(cardNo)` 后，就不会再调用父类的无参构造方法。

3. 不调用父类的构造方法时，输出如下：

```
Card constructor called
BankCard constructor called
CreditCard constructor called,cardNo=123
=====
Card constructor called,cardNo=456
BankCard constructor called,cardNo=456
DebitCard constuctor called,cardNo=456
```

可以看出，不显式调用父类的构造方法，实际上也会默认调用父类的无参构造方法，和显式调用 `super()` 效果是一样的。也就是说，我们在创建子类对象时，要么通过明确调用父类的有参构造方法递归的创建父类对象，要么 `java` 会递归的调用父类的无参构造方法来初始化各级父类对象。

3.8 static

一个电信的面向对象的设计如图3.13所示。

也就是说，我们在抽象出一类事物的共同特性后，编写相应于这类事物的一个类，在这里是 `Book`；然后根据 `Book` 类再创建若干个 `Book` 类型的对象，`book1`、`book2` 等等。有了 `book` 对象后，就可以访问 `book` 对象的属性和方法了，比如调用：`book.edit()`。总之，原则上我们只有持有一个对象后（其实是持有一个指向对象的引用），才能操作这个对象。由图3.13可以看出，不同对象的状态（属性值）往往是不同的。

考虑下面的情形：

- 比如无论书籍的类型、开张等，书籍的“章”一般都定义为“第一章”、“第二章”，因此可以将 `CHAPTER_ONE` 定义为常量，而且不随 `Book` 对象的变化而变化，即无论有多少个 `Book` 对象，也无论 `Book` 对象的其他属性如何，`CHAPTER_ONE`，`CHAPTER_TWO` 的值都不会变化。在这种情况下，每个 `Book` 对象都保存一份

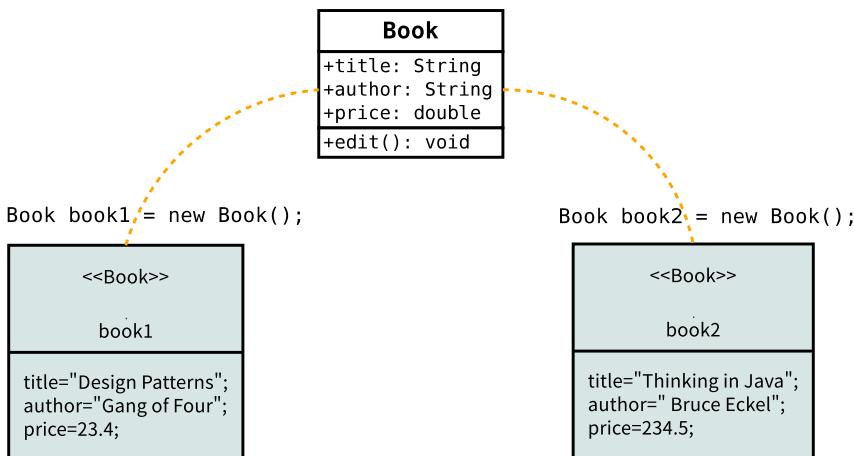


图 3.13: 面向对象编程的基本思路

`CHAPTER_ONE`, `CHAPTER_TWO` 属性是没有必要的。解决这个问题的方法参见节 3.8.1。

- 比如给每个 `Book` 对象一个 id，要求这个 id 能够自动增长，即假设第一个 `Book` 对象的 id 为 1，第二个 `Book` 对象的 id 自动设置为 2，以此类推。解决这个问题的方法参见节 3.8.2 [在第 90 页]。
- 比如我们要设计一个方法 `usage()`，当调用这个方法时打印出 `Book` 类的用法。可以看出这个方法和对象的状态没有任何关系，也就是说，`usage` 方法没有必要读取对象的状态，因此也没有必要每个对象都保存一个 `usage` 方法。解决这个问题的方法参见节 3.8.3 [在第 93 页]。

3.8.1 static 常量

显然，常量没有必要在每个对象中都保存一份，只需要在类的定义中保存一份即可。因此，正如我们在节 2.2 [在第 27 页]中提到的，常量一般的命名方式都是 `public static final` 的，其中的 `static` 表示静态变量，即只在类定义中保存一份的变量，或者说，所有对象共享一份的变量。由于是常量，也通常使用 `final` 修饰符限定此变量不可修改，即常量（常数变量）。由于静态常量只和类有关而和具体对象无关，因此静态常量又称为类常量。

在类中定义常量后，我们有两种方式使用（引用）这个常量：

1. 直接使用类名引用类常量。也就是说，不需要创建一个此类的对象，可以直接通过类名引用类常量。
2. 使用对象引用类常量。如果已经创建了对象，通过对象引用类常量也是可以的。

代码清单 3.38: 使用了 static 常量的 Book.java

```
1 package cn.edu.sduot.softlab.oopbasic.staticconstant;
2
3 /**
4  * 本类示例静态常量的惯用定义方式。
5 *
6 * @author Su Baochen
7 */
8 public class Book {
9     public static final String CHAPTER_ONE = "第一章";
10    public static final String CHAPTER_TWO = "第二章";
11
12    String title;
13    String author;
14    float price;
15
16    @Override
17    public String toString() {
18        return "Book{" + "title=" + title + ", author=" + author + ", price=" + price +
19        '}';
20    }
21 }
```

例 3.13. 在 Book 中定义常量 CHAPTER_ONE, CHAPTER_TWO, 并使用两种方法引用此常量。

代码设计 参见代码清单3.38和代码清单3.39。

运行结果和分析 执行 Client 结果如下：

```
第一章
第一章
```

虽然两种方式都打印出了正确的结果，但是第二种引用 static 常量的方式是不建议的：即不经济，也没必要。

练习 3.1. 在 JDK 源代码中找到一些常量的定义，认真体会：

- 常量的惯用命名方式,。
- 常量的使用（引用）方式。

代码清单 3.39: Client.java

```
1 package cn.edu.sdut.softlab.oopbasic.staticconstant;
2
3 /**
4  * 演示如何使用静态常量.
5  *
6  * @author Su Baochen
7  */
8 public class Client {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        // 常见用法
17        System.out.println(Book.CHAPTER_ONE);
18        // 下面是不建议的用法, 尽量避免
19        Book book = new Book();
20        System.out.println(book.CHAPTER_ONE);
21    }
22
23 }
```

JDK 源代码可以从 <https://github.com/dmlloyd/openjdk> 下载。在 openjdk 的源代码目录下执行如下命令可以找到非常多的常量定义：

```
$ grep -ri "public static final" *
```

3.8.2 static 属性

当我们用 `static` 修饰类的属性时，这个属性就成为“类属性”。相对于“实例属性”（每个对象都有一份独立的拷贝），**类属性在此类的所有对象之间共享一份拷贝**。对于类属性，我们需要注意以下三点：

- 类属性的引用。我们一般采用 `Book.nextId` 的方式，即直接使用类来引用类属性，这也是类属性设计的本意。
- 类属性相当于在对象之间共享的变量，因此一个对象修改了类属性，会影响另外对象的状态，因此使用类属性实际上导致了对象之间的“紧耦合”，造成了系统模块之间的界面不清晰，这是现代软件设计中力图规避的地方。因此，在软件设计实践中要尽量避免使用类属性，除非你有足够的理由。
- 类属性的初始化问题。类属性的默认值和普通属性没有任何区别，但是由于类属性和具体对象没有关联，因此无法在对象的构造方法中初始化。Java 提供了“**static 块**”的方式初始化类属性，比如在 `Book` 中这样初始化 `nextId` 类属性：

```
1 // 初始化静态属性，这里设置id的起点值
2 static {
3     nextId = 100;
4 }
```

和对象的构造方法不同的是，`static` 代码块在整个应用程序运行期间只会执行一次，而对象的构造方法在创建对象时都会执行一次。

例 3.14. 编写一个 `Book` 类，`book` 对象使用自动增长的 `id` 作为对象的标识。

代码设计 参见代码清单3.40和代码清单3.41。

运行结果 运行 `Client` 结果如下：

```
Book{id=100, title=Thinking in Java, author=null, price=0.0}
Book{id=101, title=Design Patterns, author=null, price=0.0}
```

代码清单 3.40: Book.java

```
1 package cn.edu.sdut.softlab.oopbasic.staticvariable;
2
3 /**
4  * 本类示例静态属性的定义方式。
5 *
6  * @author Su Baochen
7 */
8 public class Book {
9
10    int id;
11    String title;
12    String author;
13    float price;
14    static int nextId; //❶
15
16    // 初始化静态属性，这里设置id的起点值
17    static { //❷
18        nextId = 100;
19    }
20
21    @Override
22    public String toString() {
23        return "Book{" + "id=" + id + ", title=" + title + ", author=" + author
24            + ", price=" + price + '}';
25    }
26
27 }
28 //
```

❶ 类属性，在所有共享之间共享的变量

❷ 初始化类属性，在整个应用程序运行期间只会调用一次

代码清单 3.41: Client.java

```
1 package cn.edu.sdut.softlab.oopbasic.staticvariable;
2
3 /**
4  * 演示如何使用静态属性.
5  *
6  * @author Su Baochen
7 */
8 public class Client {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        Book book1 = new Book();
17        book1.title = "Thinking in Java";
18        // 建议的静态属性使用方法
19        book1.id = Book.nextId++;
20        // 不建议的使用静态属性的方法
21        //book1.id = book1.nextId++;
22        System.out.println(book1);
23
24        Book book2 = new Book();
25        book2.title = "Design Patterns";
26        book2.id = Book.nextId++;
27        System.out.println(book2);
28    }
29
30 }
```

3.8.3 static 方法

和类属性¹⁵一样，类方法指使用 `static` 修饰的方法，类方法在所有对象之间共享一个拷贝。类方法通常用于一些工具类的定义，比如 JDK 中的 `Math` 类¹⁶里面的方法都是类方法。

类方法是在对象创建之前就存在的，因此在类方法中不能访问实例属性（对象还没有创建，实例属性根本不存在），只能访问类属性（使用 `static` 修饰的属性），见下面的代码片段：

```

1 public class Test {
2     string name;
3     static int nextId;
4
5     static void testStaticMethod() {
6         nextId++; // 这是可以的
7         //name = "zhangsan"; // 这是不允许的，语法错误
8     }
9 }
```

练习 3.2. 在 `Book` 类中增加一个类方法 `usage()`，调用这个方法显示一段文字，说明这个类的用途。

练习 3.3. 设计一个字符串处理工具类，给出如下的方法实现：

- 能够方便的删除字符串的最后一个字母；
- 能够自动将字符串转化为网址形式，即自动增加“`http://`”前缀；

3.8.4 * 内部类回头看

在内部类中，我们看到内部类就像是定义在类内部的方法一样。类的方法分为静态方法和非静态方法，静态方法只能访问类的静态属性，那么内部类是不是也可以使用 `static` 修饰呢？

实际上，java 的内部类分为四种类型：

1. **成员内部类 (member inner class)**，即我们在内部类中讨论的，和实例方法用法类似的内部类。

¹⁵需要澄清的是，下面两种说法是等价的，我们可能在不同的场合交替使用不同的名称：

- 类属性 = 静态属性
- 类方法 = 静态方法
- 实例属性 = 实例变量 = 非静态属性 = 非静态变量
- 实例方法 = 实例函数 = 对象方法 = 对象函数 = 非静态方法

¹⁶参见：<http://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

2. 静态内部类 (**static inner class**) , 使用 static 修饰的内部类。
3. 局部内部类 (**local inner class**) , 定义在方法内部的类。
4. 匿名内部类 (**anonymous inner class**) , 只使用一次的内部类。

内部类的第一种形式：成员内部类我们已经讨论过了，这里不再赘述，下面我们分别看一下其他三种内部类：

3.8.4.1 静态内部类

静态内部类即使用 static 修饰的内部类，也常叫做“嵌套类”（Nested Class）。静态内部类有两个重要的意义：

1. 正如静态方法一样，使用静态内部类可以限制内部类对外部包围类属性和方法的访问。对比成员内部类，静态内部类只能访问外部包围类的静态属性和静态方法。使用静态内部类的好处是，如果我们在外部包围类中不定义静态属性和方法，实际上就切断了静态内部类和外部包围类的联系，使得静态内部类成为一个独立的模块，符合“松耦合”的现代软件设计理念。因此可以看到在实际的软件开发中，内部类很多都定义为静态内部类，比如 Android 程序设计中，大量使用到静态内部类。
2. 静态内部类对象的创建不需要首先创建外部包围类对象，比如 Outer.Inner innerObj = new Outer.Inner(); 这实际上让静态内部类成为了一个受限的顶级类。

例 3.15. 编写一个静态内部类，演示静态内部类访问外部包围类的静态属性和静态方法¹⁷。

代码设计 参见代码清单3.42和代码清单3.43。

代码清单 3.42: Book.java

```

1 package cn.edu.sdut.softlab.oopbasic.nestedclass;
2
3 /**
4  * 本类演示了静态内部类。
5  *
6  * @author Su Baochen
7 */
8 public class Book {
9
10    private int id;
11    private String title;

```

¹⁷完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/nestedclass>

```
12     private float price;
13     private static final int VERSION = 1;
14
15     public Book(String title) {
16         this.title = title;
17     }
18
19     public void publish() {
20         new Editor("someone").edit(); // ①
21         System.out.println("publish the book[" + title + "]");
22     }
23
24     static class Editor {
25
26         private String editor;
27
28         public Editor(String name) {
29             this.editor = name;
30         }
31
32         // @TODO 尝试修改edit方法的访问控制修饰符看看 ?
33         // private的内部类方法不能在Client中创建的Editor对象调用,
34         // 但是可以在外部包围类中调用
35         private void edit() {
36             //System.out.println(title); // ①
37             System.out.println("edit the book[version=" + VERSION + "] by " + editor); // ②
38         }
39
40         public void usage() {
41             System.out.println("Book.Editor.usage() called");
42         }
43     }
44 }
45
46 }
47 //
```

① 在外部包围类中可以调用静态内部类中的任何方法，包括 `private` 方法

② 不允许访问包围类的非静态属性

③ `VERSION` 在外部包围类中是静态属性，因此在静态内部类中可见

代码清单 3.43: Client.java

```
1 package cn.edu.sdut.softlab.oopbasic.nestedclass;
2
3 /**
4  * 本类演示了静态内部类对象的创建和使用方式 .
5  *
6  * @author Su Baochen
7 */
```

```

8 public class Client {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        Book book = new Book("Learning Java based on C");
17        book.publish();
18        Book.Editor editor = new Book.Editor("somebody");
19        editor.usage();
20
21    }
22 }
```

运行结果 执行 Client 结果如下：

```

edit the book[version=1] by someone
publish the book[Learning Java based on C]
Book.Editor.usage() called
```

3.8.4.2 局部内部类

局部内部类定义在方法内部，可以认为是方法的局部变量。就像方法的局部变量一样，局部内部类的作用范围仅限于方法内部，即在方法外部是无法创建局部内部类的对象的。局部内部类可以使用所在方法中的属性，也可以访问类的实例属性和类属性。

局部内部类的使用场合较少，通常在组织复杂方法代码时使用。但是，由于在方法内部定义类，可能导致方法体臃肿，使得方法的逻辑层次不清晰，在使用局部内部类时要适当取舍。

例 3.16. 编写一个局部内部类，演示局部内部类对方法属性的访问和局部内部类对象的创建方式¹⁸。

代码设计 参见代码清单3.44和代码清单3.45。

运行结果 执行 Client 结果如下：

¹⁸ 完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/oopbasic/src/cn/edu/sdut/softlab/oopbasic/localinnerclass>

代码清单 3.44: Book.java

```
1 package cn.edu.sdut.softlab.oopbasic.localinnerclass;
2
3 /**
4  * 本类演示了局部内部类的用法.
5  *
6  * @author Su Baochen
7  */
8 public class Book {
9
10    int version = 1;
11    static int nextId;
12
13    /**
14     * 发布图书.
15     */
16    public void publish() {
17
18        class Publisher {
19            private void edit() {
20                System.out.println("edit the book before publish");
21            }
22
23            public void publish() {
24                edit();
25                System.out.println("publish book[version=" + version + ",id=" + nextId++);
26            }
27        }
28
29        new Publisher().publish();
30    }
31
32 }
```

代码清单 3.45: Client.java

```

1 package cn.edu.sdut.softlab.oopbasic.localinnerclass;
2
3 /**
4  * 本类演示了局部内部类的用法 .
5 *
6 * @author Su Baochen
7 */
8 public class Client {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        Book book = new Book();
17        book.publish(); // ①
18    }
19 }
20 //
```

① 不能直接创建局部内部类的对象

```

edit the book before publish
publish book[version=1,id=0
```

3.8.4.3 匿名内部类¹⁹

如果一个内部类只使用一次的话，就没有必要非要给这个内部类起个名字了，这就是匿名内部类的使用场合。匿名内部类由于没有名字，必须从父类继承下来（在节 4.2.5 [在第 116 页] 中我们可以看到，也可以实现一个接口）。

例 3.17. 使用匿名内部类

代码设计 参见代码清单3.46、代码清单3.47。

运行结果 执行 Client 结果如下：

¹⁹ 在节 9.6 [在第 278 页]、节 4.2.5 [在第 116 页]、节 ?? [在第 ?? 页] 中，我们还会看到匿名内部类的使用。

代码清单 3.46: Book.java

```
1 package cn.edu.sdut.softlab.oopbasic.annnoymousinnerclass;
2
3 /**
4  * 本类匿名内部类的父类.
5 *
6 * @author Su Baochen
7 */
8 public class Book {
9     private String title;
10
11    public Book(String title) {
12        this.title = title;
13    }
14
15    public void publish() {
16        System.out.println("publish the book[" + title + "]");
17    }
18 }
```

edit the book before publishing
publish the book[Learning Java Based on C]

代码分析 在例3.17中，我们组合使用了匿名对象和匿名内部类，这是由于在这种情况下，创建的匿名内部类对象也只使用一次，就没有必要起个名字了。我们给对象一个名字的目的不就是在重复使用这个对象的时候方便引用吗？

练习 3.4. 还原例3.17为不使用匿名内部类的情形，对比一下说明匿名内部类的优点和缺点。

3.8.5 * 内部类的进一步讨论

在前面的讨论中，内部类都是没有父类的。但是，内部类其实也可以是一个子类²⁰，这样我们在外部包围类中定义多个内部类，每个内部类继承自一个父类，就巧妙的绕过了Java 没有多继承的限制，曲线的实现了多继承。

例 3.18. 编写程序，使用内部类达到多继承的目的。

代码设计 参见代码清单3.48、代码清单3.49、代码清单3.50、代码清单3.51。

²⁰在节 4.2.3 [在第 110页]中，我们可以看到，内部类也可以实现（多个）接口，从而更好的实现 Java 的多继承。

代码清单 3.47: Client.java

```
1 package cn.edu.sdut.softlab.oopbasic.anonymousinnerclass;
2
3 /**
4  * 本类演示匿名内部类的用法 .
5  *
6  * @author Su Baochen
7 */
8 public class Client {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        // 匿名内部类, 是Book的子类
17        // 这里同时使用了匿名对象
18        new Book("Learning Java Based on C") {
19            private void edit() {
20                System.out.println("edit the book before publishing");
21            }
22
23            @Override
24            public void publish() {
25                edit();
26                super.publish();
27            }
28        }.publish();
29
30    }
31
32 }
```

代码清单 3.48: Printer.java

```
1 package cn.edu.sdut.softlab.oopbasic.multiinherit;
2
3 /**
4  * 打印机基类.
5  *
6  * @author Su Baochen
7 */
8 public class Printer {
9
10    public void print() {
11        System.out.println("basic print");
12    }
13
14 }
```

代码清单 3.49: Copier.java

```
1 package cn.edu.sdut.softlab.oopbasic.multiinherit;
2
3 /**
4  * 复印机基类.
5  *
6  * @author Su Baochen
7 */
8 public class Copier {
9
10    public void copy() {
11        System.out.println("basic copy");
12    }
13 }
```

代码清单 3.50: SmartPrinter.java

```
1 package cn.edu.sduot.softlab.oopbasic.multiinherit;
2
3 /**
4  * 多功能一体机类.
5  *
6  * @author Su Baochen
7 */
8 public class SmartPrinter extends Printer {
9
10    @Override
11    public void print() {
12        super.print();
13        System.out.println("SmartPrinter print");
14    }
15
16    public void copy() {
17        new MultiCopier().copy();
18    }
19
20    class MultiCopier extends Copier {
21
22        @Override
23        public void copy() {
24            super.copy();
25            System.out.println("SmartPrinter copy");
26        }
27    }
28
29 }
```

代码清单 3.51: SmartPrinter.java

```
1 package cn.edu.sdut.softlab.oopbasic.multiinherit;
2
3 /**
4  * 本类演示了使用内部类实现多继承的方法 .
5 *
6 * @author Su Baochen
7 */
8 public class SmartPrinterClient {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        SmartPrinter sp = new SmartPrinter();
17        sp.print();
18        sp.copy();
19    }
20
21 }
```

运行结果 执行 Client 结果如下：

```
basic print
SmartPrinter print
basic copy
SmartPrinter copy
```

代码分析 可以看出，在 SmartPrinterTest 中，我们只是使用 SmartPrinter 对象即可同时获得 print 和 copy 两个功能，好像同时从 Printer 和 Copier 继承下来一样。SmartPrinter 类其实是一个 Facade 类 [5]，当我们希望在一个类中融合多项功能时，为了代码逻辑清晰起见，借用内部类实现多继承是常见的策略。

思考和练习

练习 3.5. 编写一个 default 修饰的类，尝试在其他包中创建该类的对象，结果会怎样？

练习 3.6. 考虑下面的代码，指出哪些是类属性，哪些是实例属性？

第四章 面向对象工程思想

不想当将军的士兵不是好士兵，不想当架构师的程序员不是一个好程序员。本章内容不可能涵盖 Java 架构师的所有内容，只是一个菜鸟 Java 程序员进阶的必由之路。



4.1 抽象类和抽象方法

在类的继承关系中，有些情况下我们不希望根据父类创建对象，或者说希望禁止创建父类对象。比如图4.1所示的情形。

如果我们希望禁止创建 Animal 对象¹，即只允许创建 Dog、Cat 等 Animal 的子类对象，该怎么做呢？Java 提供了两种方式：

1. 使用抽象类：将 Animal 定义为抽象类，这样使用 new 创建 Animal 对象将导致语法错误，参见节 4.1.1 [在对页]。

¹为什么有这种需求呢？主要的出发点是强制使用具体的子类对象，因为子类对象比父类对象提供了更具体的属性和方法。

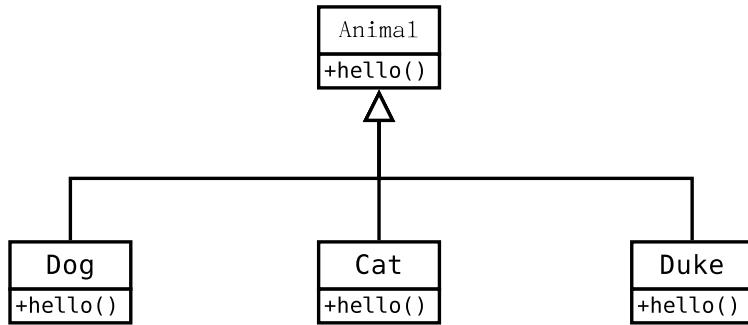


图 4.1: Animal 的类层次关系

代码清单 4.1: Animal.java

```
1 package cn.edu.sdu.tsoftlab.oopm.abstractclass;
2
3 /**
4 *
5 * @author Su Baochen
6 */
7 public abstract class Animal {
8
9     public void hello() {
10         System.out.println("动物在打招呼");
11     }
12 }
```

2. 使用接口: 将 Animal 定义为接口, 这样便不能直接创建 Animal 类型的对象, 只能通过实现 Animal 接口创建具体的对象, 参见节 4.2 [在第 108 页]。

4.1.1 抽象类

只要一个类使用 `abstract` 修饰, 这个类就是一个抽象类。抽象类不允许通过 `new` 操作符创建对象, 抽象类只能作为父类被继承。

例 4.1. 编写一个抽象的 Animal 类。

代码设计 参见代码清单4.1、代码清单4.2、代码清单4.3。

运行结果 运行 Client 结果如下:

```
动物在打招呼
```

代码清单 4.2: Dog.java

```
1 package cn.edu.sdut.softlab.oopm.abstractclass;
2
3 /**
4  * Dog继承了Animal.
5  *
6  * @author Su Baochen
7 */
8 public class Dog extends Animal {
9
10 }
```

代码清单 4.3: Client.java

```
1 package cn.edu.sdut.softlab.oopm.abstractclass;
2
3 /**
4  * 本类演示了抽象类不能实例化 .
5  * @author Su Baochen
6 */
7 public class Client {
8
9 /**
10 * 程序执行入口.
11 * @param args 命令行参数
12 */
13 public static void main(String[] args) {
14     // 抽象类不能实例化!
15     //Animal animal = new Animal();
16     Dog dog = new Dog();
17     dog.hello(); // hello是继承自Animal的
18 }
19
20 }
```

代码清单 4.4: Animal.java

```
1 package cn.edu.sdu.t.softlab.oopm.abstractmethod;
2
3 /**
4  * 本类演示了包含抽象方法的抽象类。
5  *
6  * @author Su Baochen
7  */
8
9 public abstract class Animal {
10
11     public abstract void hello();
12 }
```

代码分析 抽象类是不允许直接实例化的，因此在 Client 中直接创建 Animal 类型的对象会报告语法错误。虽然 Dog 类看起来是一个空的类，但是 Dog 继承自 Animal，因此也继承了 Animal 中已经实现了的公有的 hello 方法。

4.1.2 抽象方法

有的时候，我们希望父类的方法仅仅是个声明，即只是说明了方法的名字、返回值类型、参数的类型和个数及其顺序，方法的具体内容（即方法体）需要父类的子类去实现。也就是说，父类的这个方法是“抽象”的，因为缺少了方法体，直接调用这个抽象的方法也没有意义，因此 Java 规定只要类中有一个抽象方法，这个类就必须定义为抽象类。子类在继承抽象类后，必须实现父类中的所有抽象方法，除非子类也是抽象类。

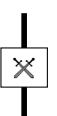
例 4.2. 实现一个包含抽象方法的抽象类。

代码设计 参见代码清单4.4、代码清单4.5，Client 的设计和例4.1一样，这里不再列出。

运行结果 执行 Client 结果如下：

```
汪汪
```

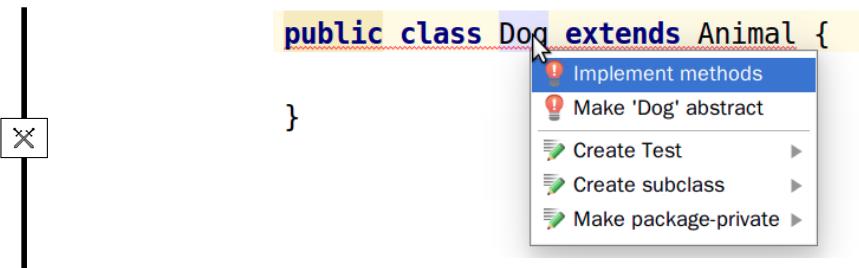
代码分析 可以看出，父类中的抽象方法 hello 在子类中必须给出具体的实现。

 在 Idea 中，如果没有实现父类中的抽象方法，在类名上面会给出提示信息，此时只要在类名上面按 Alt+Enter 键组合即可自动给出默认的方法实现，非常方便：

代码清单 4.5: Dog.java

```

1 package cn.edu.sdu.softlab.oopm.abstractmethod;
2
3 /**
4  * Dog继承了Animal.
5  *
6  * @author Su Baochen
7  */
8 public class Dog extends Animal {
9
10    @Override
11    public void hello() {
12        System.out.println("汪汪");
13    }
14
15 }
```



练习 4.1. 如果 Dog 类也是抽象类，一定需要实现父类 Animal 中的抽象方法吗？试修改例4.2的 Dog 为抽象类，然后增加一个 Hunter 类作为 Dog 的子类。

4.1.3 抽象类小结

抽象类是系统架构的常见手段，有两个重要的作用：

- 强制继承，即必须提供子类才能使用抽象父类的属性和功能。
- 便于合理划分（区分）事物的共性和个性，将个性化的部分设计为抽象方法，强制子类实现父类的抽象方法。

4.2 接口

4.2.1 定义接口：纯的抽象类

如果抽象类中所有的方法都是抽象方法会怎样？比如下面的代码：

代码清单 4.6: Animal.java

```
1 package cn.edu.sdu.tsoftlab.oopm.pureabstract;
2
3 /**
4  * 演示了接口的定义.
5  *
6  * @author Su Baochen
7  */
8 public interface Animal {
9
10    void hello();
11
12    void eat();
13
14    void sleep();
15 }
```

```
1 public abstract class Animal {
2     public abstract void hello();
3     public abstract void eat();
4     public abstract void sleep();
5 }
```

也就是说，此时的 Animal 是一个纯抽象类，Java 提供了 interface（接口）来表达纯的抽象类，如代码清单4.6所示。

接口具有如下的特点：

- 接口一定是抽象的，因此接口类不需要明确声明是抽象的。即，在定义接口时，下面的两种方式都是可以的，一般采用省略 abstract 的写法。

```
1 public abstract interface Animal{...}
2 public interface Animal {}
```

- 接口中的方法一定是 public 的，所以可以省略接口方法的访问控制属性。实际上，在接口方法上使用 private 或者 protected 是语法错误，比如尝试在接口方法上增加 protected：

```
1 public interface Animal {
2     // 语法错误!
3     protected void hello();
4 }
```

正因为接口的以上两个特点，可以认为接口是对“纯的抽象类”的简化写法。

4.2.2 实现接口

接口虽然很像“纯的抽象类”，但是其用法和用意和抽象类有很多不同，参见表4.1

| | 抽象类 | 接口 |
|-------|--------------------------------|------------------------------------|
| 意义 | 概括了一类事物的共同属性和行为特征 | 概括了一类事物的共同行为特征 |
| 方法的实现 | 可以实现部分方法 | 只能声明方法，不允许实现方法 |
| 使用场合 | 用于明确类的层次关系 | 用于明确类的能力范围 |
| 用法 | 子类通过 <code>extends</code> 继承父类 | 通过 <code>implements</code> 实现指定的接口 |

表 4.1：抽象类和接口的关系

例 4.3. 将 Animal 定义为接口，Dog 类实现 Animal 接口。

代码设计 参见代码清单4.6、代码清单4.7、代码清单4.8。

运行结果 执行 Client 结果如下：

```
汪汪
Dog eat method
Dog sleep method
```

代码分析 Dog 类实现了 Animal 接口即意味着 Dog 类要给出 Animal 接口中规定的所有抽象方法的具体实现。

 和实现抽象方法类似，在 Idea 中，通过快捷键 Alt+Enter 可自动补全接口的方法，不再赘述。

4.2.3 实现多个接口

对于抽象类，我们知道需要子类继承抽象的父类并实现父类中的所有抽象方法。但是，Java 只支持单继承，即一个子类只能继承自一个抽象类。接口则不同，一个类可以“实现”若干个接口，这也是接口的强大之处。

现实中，事物总是具有多面性的，比如爱因斯坦不仅仅是物理学家，小提琴也拉得很好。比如大家对毛泽东的评价是伟大的思想家、哲学家、军事家、诗人。通常，我们

代码清单 4.7: Dog.java

```
1 package cn.edu.sdut.softlab.oopm.pureabstract;
2
3 /**
4  * 本类演示了如何实现接口.
5  *
6  * @author Su Baochen
7  */
8 public class Dog implements Animal {
9
10    @Override
11    public void hello() {
12        System.out.println("汪汪");
13    }
14
15    @Override
16    public void eat() {
17        System.out.println("Dog eat method");
18    }
19
20    @Override
21    public void sleep() {
22        System.out.println("Dog sleep method");
23    }
24
25 }
```

代码清单 4.8: Client.java

```
1 package cn.edu.sdut.softlab.oopm.pureabstract;
2
3 /**
4  * 本类演示了接口的用法 .
5  * @author Su Baochen
6  */
7 public class Client {
8
9    /**
10     * 程序执行入口.
11     * @param args 命令行参数
12     */
13    public static void main(String[] args) {
14        Dog dog = new Dog();
15        dog.hello();
16        dog.eat();
17        dog.sleep();
18    }
19 }
```

代码清单 4.9: Printer.java

```
1 package cn.edu.sdut.softlab.oopm.multiinterface;
2
3 /**
4  * 打印机接口.
5  *
6  * @author Su Baochen
7  */
8 public interface Printer {
9
10    void print();
11
12 }
```

代码清单 4.10: Copier.java

```
1 package cn.edu.sdut.softlab.oopm.multiinterface;
2
3 /**
4  * 复印机接口.
5  *
6  * @author Su Baochen
7  */
8 public interface Copier {
9
10    void copy();
11
12 }
```

会把一类事物的共同行为特征归纳为一个接口，那么具有多面性的事物，应该可以拥有多个接口才对，或者说，具有多面性的事物，应该实现多个接口。

例 4.4. 实现多个接口的类。

代码设计 参见代码清单4.9、代码清单4.10、代码清单4.11、代码清单4.12。

运行结果 执行 Client 结果如下：

```
print called
copy called
```

代码清单 4.11: SmartPrinter.java

```
1 package cn.edu.sdut.softlab.oopm.multiinterface;
2
3 /**
4  * 智能打印机类.
5  *
6  * @author Su Baochen
7 */
8 public class SmartPrinter implements Printer, Copier {
9
10    @Override
11    public void print() {
12        System.out.println("print called");
13    }
14
15    @Override
16    public void copy() {
17        System.out.println("copy called");
18    }
19
20 }
```

代码清单 4.12: Client.java

```
1 package cn.edu.sdut.softlab.oopm.multiinterface;
2
3 /**
4  * 本类演示了实现多个接口的类的用法 .
5  *
6  * @author Su Baochen
7 */
8 public class Client {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        SmartPrinter sp = new SmartPrinter();
17        sp.print();
18        sp.copy();
19    }
20
21 }
```

代码分析 SmartPrinter 实现了两个接口 Printer 和 Copier，也就意味着 SmartPrinter 同时具有打印和复印两项功能。有人说，这不是多此一举吗？去掉两个接口的定义，在 SmartPrinter 类中直接实现 copy 和 print 方法就可以。从功能上看，的确可以这样处理，但是从逻辑上看，不使用接口会带来两个方面的问题：

- Print 和 Copy 两个接口定义了功能方法的名称和参数，如果不使用接口，或者不实现规定的接口，那么方法名就失去了参照，很容易造成各自为政的局面，给代码的维护和团队的交流带来麻烦。
- 使用接口带来的另外一个好处是，我们可以通过“向上塑型”加强软件的健壮性，我们将在节 4.4 [在第 123页]展示这一点。

基于以上的分析，其实 SmartPrinter 的更合理的实现应该是代码清单4.13所示。

可以看出，通过内部类的封装，BetterSmartPrinter 类的内部结构更清晰了。

4.2.4 不懂接口的项目经理不是好的项目经理

在节 4.2.3 [在第 110页]中我们已经看到，通过接口可以帮助我们规划模块的调用界面（界限），这在团队协作中非常重要，也就是说，接口为团队协作提供了有力的语法支持，考虑下面的情形：

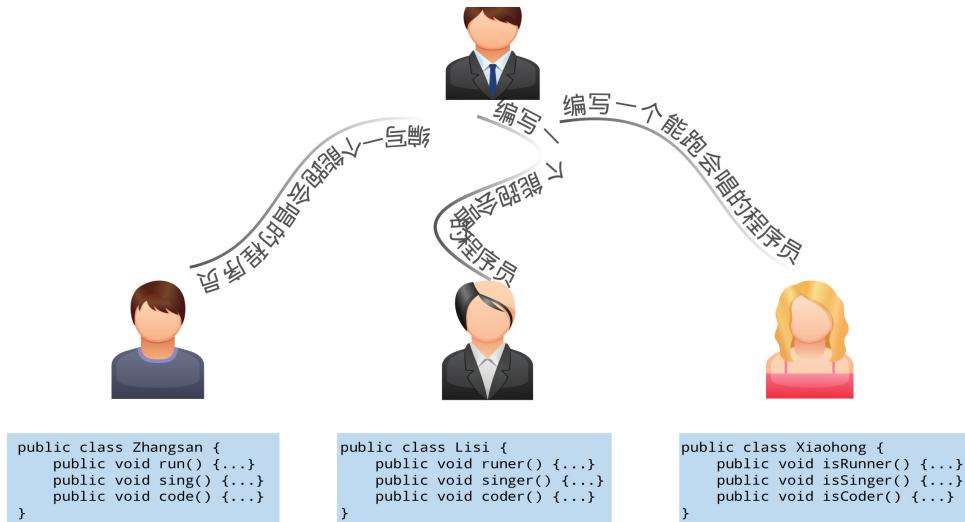


图 4.2: 不引入接口时的团队协作

如果不使用接口，团队成员在功能实现时，在方法命名和方法规划上就失去了约束，这在小型项目中也许还是可以接受的，但是随着项目规模的增大，项目成员的增加和更迭，这种没有约束的编码方式很容易把项目带到沟里，最终失去控制，导致项目失败。因此，作为项目经理或者项目的架构师，接口是一个有力的工具，在项目之初可以通过定义一系列接口的方式规定模块功能和模块之间的交互界面，如图4.3所示。

代码清单 4.13: BetterSmartPrinter.java

```
1 package cn.edu.sdut.softlab.oopm.multiinterface;
2
3 /**
4  * 更符合逻辑的SmartPrinter类.
5  *
6  * @author Su Baochen
7 */
8 public class BetterSmartPrinter {
9
10    public void print() {
11        new MultiPrinter().print();
12    }
13
14    public void copy() {
15        new MultiCopier().copy();
16    }
17
18    class MultiPrinter implements Printer {
19
20        @Override
21        public void print() {
22            System.out.println("multiPrinter print");
23        }
24    }
25
26
27    class MultiCopier implements Copier {
28
29        @Override
30        public void copy() {
31            System.out.println("multiCopier copy");
32        }
33    }
34}
35}
```

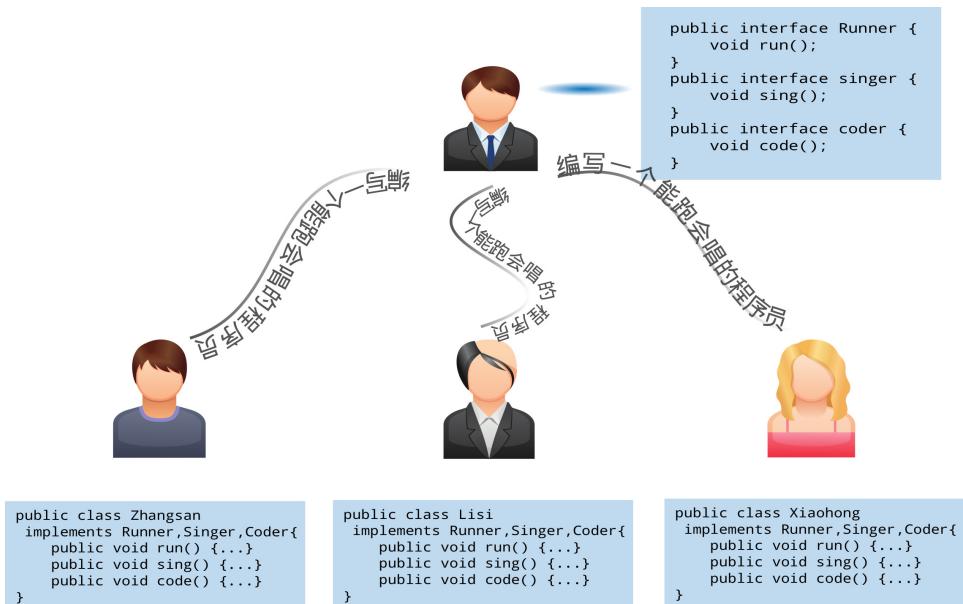


图 4.3: 引入接口后的团队协作

练习 4.2. 分别实现图4.2所示的代码和图4.3所示的代码，认真体会接口在团队协作中的作用。

4.2.5 接口上的匿名内部类

如果我们确定一个对象只临时使用一次，那么这个对象通常可以采用匿名内部类的方式来实现，参见节 3.8.4.3 [在第 98页]。如果这个临时的对象只是实现了某个接口，那么代码还可以进一步简化，直接可以使用“接口上的匿名类”技术来实现，如代码清单4.14所示（Printer 接口和 Copier 接口不变，参见代码清单4.9和代码清单4.10）。

4.3 多态

所谓多态，是指对象在不同阶段或者环境下有不同的行为特征，听起来是不是有点“变色龙”的味道？

先看一个实例，假设有如图4.1所示的类层次结构，也就是说，父类 Animal 有默认的 hello() 方法，三个子类 Dog,Cat,Duke 分别重写(overriding)了父类 Animal 的方法 hello()。代码实现如代码清单4.15、代码清单4.16和代码清单4.17、代码清单4.18、代码清单4.19所示。

在 TestAnimal 中，animals 数组的每个元素是 Animal 类型的，貌似调用 animal 数组元素的 hello 方法应该打印出“动物在打招呼”，但是执行 TestAnimal 会发现输出结果是：

代码清单 4.14: Client.java

```
1 package cn.edu.sdutooplannonymousinnerclass;
2
3 /**
4  * 本类演示了接口上的匿名内部类的用法 .
5  * @author Su Baochen
6  */
7 public class Client {
8
9     /**
10      * 程序执行入口.
11      * @param args 命令行参数
12      */
13     public static void main(String[] args) {
14         new Printer() {
15             @Override
16             public void print() {
17                 System.out.println("print called");
18             }
19
20         }.print();
21
22         new Copier() {
23             @Override
24             public void copy() {
25                 System.out.println("copy called");
26             }
27
28         }.copy();
29
30     }
31
32 }
```

代码清单 4.15: Animal.java

```
1 package cn.edu.sdut.softlab.oopm.polymorphism;
2
3 /**
4  * 演示多态中的父类.
5  *
6  * @author Su Baochen
7 */
8 public class Animal {
9
10    public void hello() {
11        System.out.println("动物在打招呼");
12    }
13
14 }
```

代码清单 4.16: Dog.java

```
1 package cn.edu.sdut.softlab.oopm.polymorphism;
2
3 /**
4  * 演示多态中的子类.
5  *
6  * @author Su Baochen
7 */
8 public class Dog extends Animal {
9
10    public void hello() {
11        System.out.println("汪汪");
12    }
13 }
```

代码清单 4.17: Cat.java

```
1 package cn.edu.sdut.softlab.oopm.polymorphism;
2
3 /**
4  * 演示多态中的子类.
5  *
6  * @author Su Baochen
7 */
8 public class Cat extends Animal {
9
10    public void hello() {
11        System.out.println("喵喵");
12    }
13 }
```

代码清单 4.18: Duke.java

```
1
2 package cn.edu.sdut.softlab.oopm.polymorphism;
3
4 /**
5  * 演示多态中的子类.
6  *
7  * @author Su Baochen
8 */
9 public class Duke extends Animal {
10
11    public void hello() {
12        System.out.println("嘎嘎");
13    }
14 }
```

代码清单 4.19: TestAnimal.java

```
1 package cn.edu.sdut.softlab.oopm.polymorphism;
2
3 /**
4  * 演示多态中的Client类.
5  *
6  * @author Su Baochen
7 */
8 public class TestAnimal {
9
10 /**
11 * 程序执行入口.
12 *
13 * @param args 命令行参数
14 */
15 public static void main(String[] args) {
16     Animal[] animals = new Animal[3];
17     animals[0] = new Dog(); //❶
18     animals[1] = new Cat(); //❷
19     animals[2] = new Duke(); //❸
20     for (Animal animal : animals) {
21         animal.hello();
22     }
23 }
24 }
25 //
```

❶ 向上塑型，将 Dog 对象转换为 Animal 对象

❷ 向上塑型，将 Cat 对象转换为 Animal 对象

❸ 向上塑型，将 Duke 对象转换为 Animal 对象

汪汪
喵喵
嘎嘎

也就是说，虽然我们调用的是父类对象的方法，但是其实真正执行的确的是子类中的方法，这就是多态。但是，同一个对象，在不同的运行时间，怎么会有不同的行为呢？这一点是如何做到的？先从对象的存储说起。

4.3.1 对象的存储模型

在上面的例子中，Dog、Cat、Duke 类的对象在内存中是如何存储的呢？比如当 Dog dog = new Dog() 创建了一个 Dog 类的实例，则在内存中的存储如图4.4所示。

其中，`dog` 是对象的引用，即指针，指向对象在内存的起始地址。对象在内存中的存储分为两个部分：父类的属性和方法存储区域以及子类的属性和方法存储区域，也就是说，子类完全包含了父类的所有信息。因此可以看出，我们得到了一个指向子类的引用 `dog`，这个引用 `dog` 即可以访问子类的方法和属性，也可以访问父类的方法和属性，无非是通过 `dog` 指针不同的偏移量指向不同的区域而已。

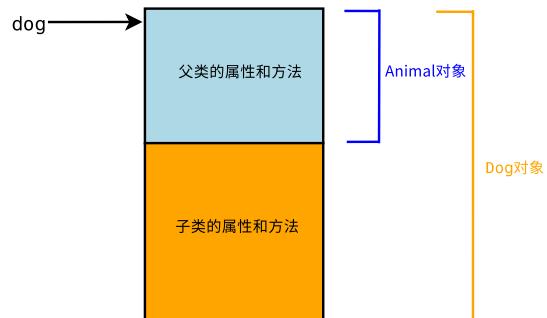


图 4.4: Java 对象在内存中的存储模型

那么 `Animal dog = new Dog()` 是什么意思呢？从图4.4可以看出，此时的 `dog` 指针仍然是指向整个 Dog 对象的指针，但是指针的范围却局限于父类 Animal 的区域，也就是说，此时的 `dog` 指针只能调用父类 Animal 中的属性和方法，而不能调用 Dog 子类扩展了的属性和方法。

如果子类没有重写（overriding）父类的方法，子类和父类的存储是泾渭分明的。如果子类重写（overriding）了父类中的方法，那么父类中被重写的方法同样会发生变化，即变的和子类中重写的方法一模一样，如图4.5所示。

可以看出，当子类重写了父类中的方法时，父类存储区域中被重写的方法就变得和子类中的方法一模一样了，这样就能够理解本文刚开始的例子了：虽然 `animals` 数组的元素都是 `Animal` 类型的对象，但是由于这三个引用分别指向了 `Dog`、`Cat`、`Duke` 的对象，而 `Dog` 等对象重写了 `Animal` 中的 `hello` 方法，致使我们虽然调用的是父类中的方法，但是由于子类重写了父类中的这个方法，父类中的这个方法变得和子类中的方法一模一样了（即所谓的 overriding，重写），最终的效果和直接调用子类中的方法一样，这就是 Java 中的多态，其实也是我们期望发生的事情。

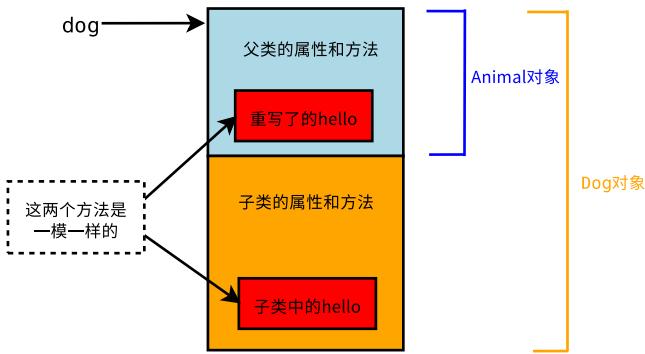


图 4.5: 存在重写情况下的类存储结构

4.3.2 方法重载时的情形

在上例中, 如果子类不是重写 (overriding) 父类的方法, 而是重载 (overloading) 父类的方法, 结果会怎样呢? 比如, Cat 类的 hello 方法改为 (Dog/Duke 以此类推):

```
1 public void hello(String name){  
2     System.out.println("喵喵" + name);  
3 }
```

也就是说, Cat 类的 hello 方法重载了父类的 hello 方法。执行 TestAnimal 的结果是:

```
动物在打招呼  
动物在打招呼  
动物在打招呼
```

根据图4.6可自行分析执行结果。

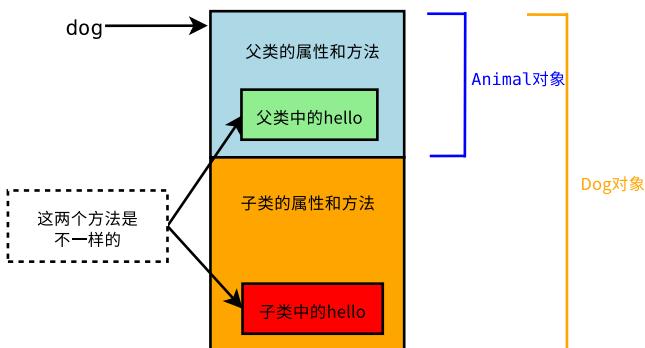


图 4.6: 存在重载情况的类存储结构

4.3.3 多态的应用场合

从以上的分析可以看出，通常会利用方法的重写来达到简化代码的目的，即充分利用 Java 的多态特性。也就是说，在多态的帮助下，我们只需要调用父类的方法就可以了，具体执行的却是子类的对应方法。当然，这里有一个前提，即必须使用子类创建对象的引用，否则多态也就失效了。

练习 4.3. 请说明实现 Java 多态的关键是什么？

4.4 * 向上塑型：面向接口的编程

从节 4.3.1 [在第 121 页]可以看出，子类对象拥有父类对象的所有属性和方法（当然，在访问控制的限制下），反过来说，父类对象的属性和方法是子类对象的一个子集，因此把子类对象转换为一个父类对象是安全的，也就是说，如果我们持有一个子类对象的引用，可以安全的把这个引用转换为父类对象的引用，因为通过这个父类对象的引用，我们访问任何属性和方法都是“可达”的，都不会引起任何错误。

但是反过来是不可行的，即把父类对象转换为子类对象。原因很简单，转换后的子类对象引用仍然指向了父类对象，但是却可能试图访问超出了父类对象范围的属性和方法，显然是不允许的。

这就是 Java 中的“向上塑型”，即对象的引用可以向父类的方向转换，但是不允许向子类的方向转换。在代码清单 4.19 中，我们已经看到了向上塑型的应用，可以说，向上塑型是 Java 多态的基本形态。

在编程实践中，我们也经常按照接口向上塑型，即把对象的类型转换为其实现的某个接口，比如我们重写代码清单 4.12：

```
1 public static void main(String[] args) {  
2     Printer sp = new SmartPrinter(); // 向上塑型为接口类型  
3     sp.print();  
4 }
```

可以看出，通过向上塑型为父类对象或者接口类型，实际上我们获得了一个“受限”的对象引用，即这个对象引用只能访问原对象的部分属性和方法，一定程度上提高了程序的安全性和健壮性：尽量少的对外暴露调用接口。这也符合面向对象编程的封装原则。

4.4.1 对象的类型

讨论如下的代码片段：

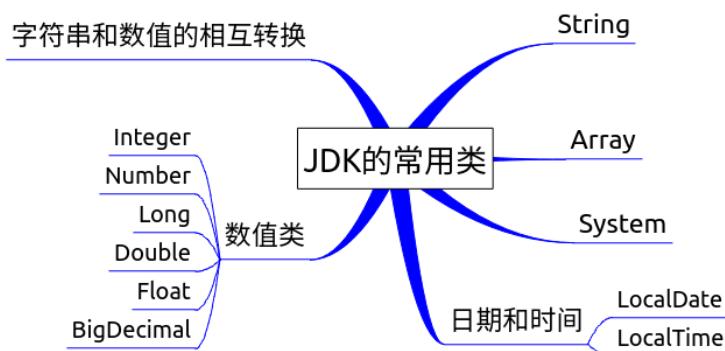
```
1 public class SmartPrinter extends BasePrinter implements Printer, Copier {  
2     ...  
3 }
```

那么以下的用法都是合法的：

```
1 SmartPrinter sp = new SmartPrinter();
2 BasePrinter sp = new SmartPrinter(); // 向上塑型为父类对象
3 Printer sp = new SmartPrinter(); // 向上塑型为接口类型
4 Copier = new SmartPrinter(); // 向上塑型为接口类型
5 Object sp = new SmartPrinter(); // Object是所有类的父类
```

练习 4.4. 一个对象可以有多少种类型？试举例说明。

第五章 Java 的常用类



5.1 字符串类

字符和字符串是程序设计中最常见的数据类型了。在 C 语言中只有字符类型，字符串需要通过字符数组或者指向字符的指针来表达，而 Java 直接提供了 String 类¹来表示字符串，对字符串的处理就方便多了。

5.1.1 字符串对象的定义和初始化

创建字符串对象的最简单方法是直接使用双引号初始化字符串：

```
1 String hello = "Hello, World!";
```

除此之外，JDK8 提供了多达 15 个构造方法可以用来创建字符串对象²，在代码清单5.1：

 注意双引号和单引号的用法区别：双引号用来表示字符串，单引号用来表示字符，比如：

¹建议将 JDK API 的文档常置手边，方便查询。JDK8 API 文档在线版：<http://docs.oracle.com/javase/8/docs/api/index.html>

²参见：<http://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

代码清单 5.1: Client.java

```
1 package cn.edu.sdut.softlab.essential.string;
2
3 /**
4  * 本类演示了String类的常见用法 .
5  *
6  * @author Su Baochen
7  */
8 public class Client {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        String hello = "Hello World!";
17        print(hello);
18        print("Hello Java!");
19
20        char[] data = {'a', 'b', 'c'};
21        String str = new String(data);
22        print(str);
23
24        byte[] bytes = {97, 98, 99}; // ascii码表可查阅http://www.asciitable.com/
25        String byteStr = new String(bytes);
26        print(byteStr);
27    }
28
29    /**
30     * 打印字符串内容.
31     *
32     * @param str 字符串类型的输入参数
33     * @TODO 如何演示hello字符串是不变长度的?
34     */
35    public static void print(String str) {
36        System.out.println(str);
37    }
38
39 }
```

代码清单 5.2: SubStringTest.java

```
1 package cn.edu.sdut.softlab.essentials.string;
2
3 /**
4  * 本类演示了substring方法的用法 .
5 *
6 * @author Su Baochen
7 */
8 public class SubStringTest {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        String str = "java programming language,";
17        System.out.println(str.substring(5)); // 输出programming language
18        System.out.println(str.substring(5, 17)); // 输出programing
19        // 输出java programming language
20        System.out.println(str.substring(0, str.length() - 1));
21    }
22 }
```



String str = "a string"; // 一个字符串对象
char c = 'A'; // 一个字符变量

5.1.2 常见字符串处理方法

5.1.2.1 字符串的长度

获取字符串的长度使用 String 类提供的 length() 方法：

```
1 String str = "a string";
2 System.out.println(str.length());
```

length 方法返回的字符串长度不包括“字符串结束符”³，因此上面的代码片段输出结果为 8。

5.1.2.2 截取字符串

如果把字符串看做字符的有序序列，那么我们可以任意截取这个字符序列的一小段，String 类提供了 substring 方法，参见代码清单 5.2。

³在 C 语言中，每个字符串都有一个字符串结束符，即'\0'，在计算字符串的长度时一般不包括字符串结束符。

一个常见的任务是去掉字符串开头或者末尾的某个特殊字符, 比如字符串 “zhangsan,lisi,”, 我们希望去掉最后的逗号变成 “zhagnsan,lisi”, 可以使用下面的代码:

```
1 String str = "zhangsan,lisi,";
2 System.out.println(str.substring(0,str.length() - 1));
```



注意到, `substring` 方法中, 开始字符串的索引和结束字符串的索引都是从 0 开始的。



如果仅仅是去掉字符串两端的空白字符(包括空格、回车、Tab), 则可以使用 `String` 提供的 `trim` 方法, 这在 WEB 开发中很常见: 用户从浏览器输入的表单中输入的字符串一般需要使用 `trim` 处理一下, 去掉用户无意中输入的空白字符。

5.1.2.3 分割字符串

有的字符串存在明显的分隔符, 比如 linux 的 `passwd` 文件, 使用 “:” 隔开了各个区域:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

如何方便的把这类字符串解析为一个字符串数组呢? `String` 提供了方便的 `split` 方法:

```
1 String str = "root:x:0:0:root:/root:/bin/bash";
2 String[] result = str.split(":");
3 for (String s : result) {
4     System.out.println(s);
5 }
```

注意到, `split` 方法的参数不仅仅可以是一个简单的字符串, 还可以是一个正则表达式⁴, 能够处理复杂的字符串分割的情形。

练习 5.1. 编写一个程序, 解析 Linux 的 `passwd` 文件, 输出用户名和该用户的登录 shell。(关于如何打开文件请参阅节 7 [在第 170 页])。

⁴ 详情请参见: <https://docs.oracle.com/javase/tutorial/essential/regex/>

5.1.2.4 拼接字符串

Java 中拼接两个字符串的方法有以下几种：

- 重载“+”运算符：Java 重载了“+”运算符，允许直接将两个字符串首尾连接起来。注意到如果“+”的两个操作数只要任一个是字符串，另外一个操作数都会自动转换为字符串形式，比如下面的代码片段：

```
1 String str = "Java " + "cool!";
2 int a = 10;
3 System.out.println("variable a=" + a); // 打印出: variable a=10
4 String str2 = a + ""; // 简单的方法把数字a转化为字符串"10"
```

- 使用 `String` 的 `concat` 方法可以达到和使用 + 同样的效果，比如：

```
1 "java".concat(" language"); // 返回 java language
2 "java".concat(" programming").concat(" language");// 返回java programming
language
```

- 使用 `String` 的 `join` 方法⁵，这个方法是 `split` 方法的“反方法”，比如：

```
1 String message = String.join("-", "Java", "is", "cool");
2 // message returned is: "Java-is-cool"
```

- `StringBuilder` 类提供了高效处理字符串的功能，通常涉及到大量字符串拼接时，建议使用 `StringBuilder` 来完成：

```
1 StringBuilder sb = new StringBuilder();
2 sb.append("Java");
3 sb.append("is cool!");
4 String str = sb.toString();
```

5.1.2.5 查找（匹配）字符串

分为以下几种情形：

- 检查是否以特定字符串开头的 `startsWith`
- 检查是否以特定字符串结尾的 `endsWith`

⁵这是一个类方法（静态方法）

代码清单 5.3: StringMatch.java

```
1 package cn.edu.sdu.tsoftlab.essential.string;
2
3 /**
4  * 本类演示了String类的字符串匹配相关方法 .
5  *
6  * @author Su Baochen
7  */
8 public class StringMatchTest {
9
10    /**
11     * 程序执行入口.
12     *
13     * @param args 命令行参数
14     */
15    public static void main(String[] args) {
16        String str = "Java programming language";
17        System.out.println(str.startsWith("Java")); //true
18        System.out.println(str.endsWith("language")); //true
19        System.out.println(str.contains("prog")); //true
20        System.out.println(str.matches("[Jj]ava.*")); //true, 也能匹配java programming
21        language
22    }
23 }
```

- 检查是否包含特定字符串的 contains
- 检查字符串是否满足正则表达式的 matches

例 5.1. 综合示例 String 的字符串匹配方法

代码设计 参见代码清单5.3。

代码分析 这里尤其要注意 matches 方法，其参数是一个正则表达式。

5.1.2.6 其他字符串处理方法

JDK 的 String 类提供了丰富的字符串处理功能，除了上面介绍的常见方法外，还有如下的方法也很常用，请大家自行参考<http://docs.oracle.com/javase/8/docs/api/java/lang/String.html> 学习：

- 根据索引提取字符串中的字符：charAt
- 比较两个字符串：equals

- 比较两个字符串（忽略大小写）`equalsIgnoreCase`
- 判断字符串是否为空：`isEmpty`
- 替换字符串：`replace`
- 转换大小写：`toLowerCase`, `toUpperCase`

Java 面试中经常遇到的问题：`String str = new String("a string")` 创建了几个字符串对象？答案是 2 个：“a string” 作为 `String` 构造方法的参数本身是一个字符串对象，根据“a string”这个字符串使用 `String` 的构造方法又创建了一个字符串对象 `str`，因此这一句创建了 2 个字符串对象。那么 `String str = "a string"` 创建了几个字符串对象呢？显然是 1 个。因此可以看出，使用 `new` 操作符创建字符串对象一般是没有必要的，徒增 Java 的内存占用而已。

5.2 数字类

5.2.1 基本数字类型变量的包裹类

在程序中，当我们用到数字时，一般使用数字的基本数据类型来表达，比如：

```
1 int age = 20;
2 float money = 23456.5f;
3 byte mask = 0xff;
```

如果需要对数字做更深入的处理怎么办呢？Java 提供了各种基本数字类型的包裹类（Wrapper Class），如图5.1所示。

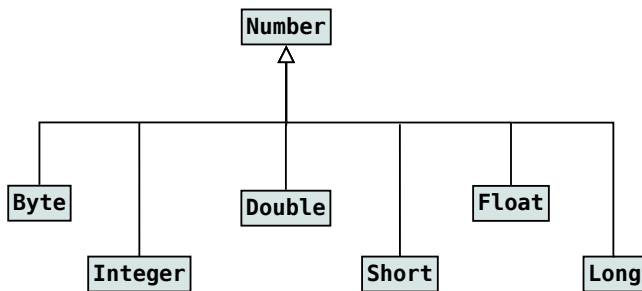


图 5.1: java 中的数字类型包裹类

一般以下情形中，我们需要数字类型的包裹类：

- 有的方法参数明确声明需要一个数字类型的对象，比如：`int toInt(Integer obj)`。

- 使用包裹类中定义的常量，比如 MAX_VALUE, MIN_VALUE, SIZE, TYPE 等。
- 在数字和字符串之间转换或者不同数制之间转换时。

所有 Number 的子类都实现了的方法如表5.1所示。

| 方法名 | 描述 |
|--|---|
| byte byteValue() short shortValue() int intValue() long longValue() float floatValue() double doubleValue() | 返回请求的基本数字类型数值，可能存在损失精度情况，比如一个 Double 类型的对象使用 intValue 时。 |
| int compareTo(Byte anotherByte) int compareTo(Double anotherDouble) int compareTo(Float anotherFloat) int compareTo(Integer anotherInteger) int compareTo(Long anotherLong) int compareTo(Short anotherShort) | 数字类型对象的比较，返回值的意义： 0：两个数字类型对象代表的数字相等 <0：本数字类型对象小于参数中的数字类型对象 >0：本数字类型对象大于参数中的数字类型对象 |
| boolean equals(Object obj) | 覆盖了 Object 类的 equals 方法，仅当两个包裹对象不是 Null 并且代表的数字相等时返回 true。不过有两个例外情况，请参见： http://docs.oracle.com/javase/8/docs/api/java/lang/Number.html |

表 5.1: Number 子类的方法列表

例 5.2. Number 子类的公共方法。

代码设计 参见代码清单5.4。

运行结果 运行 WrapperNumberTest 结果如下：

代码清单 5.4: WrapperNumberTest.java

```
1 package cn.edu.sdu.tsoftlab.essential.number;
2
3 /**
4  * 本类演示了数字类型包裹类的用法 .
5  * @author Su Baochen
6  */
7 public class WrapperNumberTest {
8
9     /**
10      * 程序执行入口 .
11      * @param args 命令行参数
12     */
13    public static void main(String[] args) {
14        System.out.println("Integer max value=" + Integer.MAX_VALUE);
15        System.out.println("Short max value=" + Short.MAX_VALUE);
16        System.out.println("Long max value=" + Long.MAX_VALUE);
17        System.out.println("Float max value=" + Float.MAX_VALUE);
18        System.out.println("Double max value=" + Double.MAX_VALUE);
19
20        Float num = 123.45f; // ❶
21        System.out.println("byteValue=" + num.byteValue());
22        System.out.println("intValue=" + num.intValue());
23        System.out.println("shortValue=" + num.shortValue());
24        System.out.println("longValue=" + num.longValue());
25        System.out.println("floatValue=" + num.floatValue());
26        System.out.println("doubleValue=" + num.doubleValue());
27
28        System.out.println("compareTo 123.45f=" + num.compareTo(123.45f)); // ❷
29        System.out.println("equals 123.45f=" + num.equals(new Float(123.45f))); // ❸
30
31    }
32
33 }
34 //
```

❶ 自动装箱，将数字 123.45 自动转换为一个 `Float` 类型的对象

❷ 自动装箱，将数字 123.45 自动转换为一个 `Float` 类型的对象

❸ 不建议这样写，利用 Java 的自动装箱机制即可

```
Integer max value=2147483647
Short max value=32767
Long max value=9223372036854775807
Float max value=3.4028235E38
Double max value=1.7976931348623157E308
byteValue=123
intValue=123
shortValue=123
longValue=123
floatValue=123.45
doubleValue=123.44999694824219
compareTo 123.45f=0
eqauls 123.45f=true
```

代码分析 可以看出，从 float 到 byte、int、short、long 转换时丢失了精度，在实际编程实践中，要根据具体需求需用合适的 xxxValue 方法获得相应的基本类型数值。

 **自动装箱 (autoboxing) 和自动拆箱 (unboxing)** 是指 Java 编译器自动在基本数据类型和其包裹类之间转换，即在需要一个包裹类的地方，我们只需要给出一个基本类型的数字即可，Java 编译器会自动将这个基本类型的数字转化为一个相应的包裹类的对象，称为“自动装箱”，相反的过程称为“自动拆箱”。比如在代码清单5.4中，我们使用自动装箱机制初始化一个 Float 类型的对象：Float num = 123.45f。

在 Java 编程实践中建议使用自动装箱和卸箱机制来处理基本数字类型变量和对象之间的转换，简化了代码，也提高了代码的可读性。

练习 5.2. 根据例5.2，请写出其他数字类的公共方法的测试类。

5.2.2 不同数字类型之间的转换

这里以 Integer 为例，其他类型的数值之间的转换类似。表5.2列出了 Integer 支持的转换方法（全部是类方法）。

例 5.3. 演示数字之间的转换。

代码设计 参见代码清单5.5

代码清单 5.5: NumberTest.java

```
1 package cn.edu.sdut.softlab.essentials.number;
2
3 /**
4  * 本类以Integer为例，演示了数字之间的相互转换 .
5  * @author Su Baochen
6  */
7 public class NumberTest {
8
9 /**
10 * 程序执行入口。
11 * @param args 命令行参数
12 */
13 public static void main(String[] args) {
14     System.out.println(Integer.decode("10")); // 10进制
15     System.out.println(Integer.decode("0x10")); // 16进制
16     System.out.println(Integer.decode("010")); // 8进制
17     System.out.println(Integer.parseInt("123"));
18 //System.out.println(Integer.parseInt("123.4")); // 格式错误
19     Integer num = 123;
20     System.out.println("num=" + num);
21     System.out.println("num=" + num.toString());
22     System.out.println(Integer.parseInt("10",8));
23     System.out.println(Integer.valueOf(123)); // ①
24     System.out.println(Integer.valueOf("123"));
25 //System.out.println(Integer.valueOf("123.4")); // 格式错误
26     System.out.println(Integer.valueOf("10",2)); // ②
27 }
28
29 }
30 //
```

① 注意这里实际上分两步走：第一步根据数字 123 获得了一个 Integer 类型的对象，第二步调用了 Integer 类型对象的 `toString` 方法输出了“123”字符串

② 根据 2 进制的“10”创建 Integer 类型的对象，整数值为 2

| 方法名 | 描述 |
|---|--------------------------------------|
| static Integer decode(String s) | 将字符串转换为整数，可以接收十进制、八进制和十六进制的表示方式。 |
| static int parseInt(String s) | 将字符串转换为整数，最常见的使用方式，只接受十进制表示方式。 |
| static int parseInt(String s, int radix) | 将字符串转换为整数，radix 为进制形式，可以为 10,2,8,16。 |
| String toString() | 将数字转换为字符串。 |
| static String toString(int i) | 静态方法，将给定的整数转换为字符串。 |
| static Integer valueOf(int i) | 根据给定的整数创建一个整数对象。 |
| static Integer valueOf(String s) | 根据给定的字符串创建一个整数对象。 |
| static Integer valueOf(String s, int radix) | 根据给定的字符串和进制形式创建一个整数对象。 |

图 5.2: Integer 的转换方法

运行结果 执行 NumberTest 结果如下：

```
10
16
8
123
num=123
num=123
8
123
123
2
```

代码分析 首先要注意到不同进制数字的表示方式，可以看出，decode 方法可以自动判别不同进制的数字字符串并转换为一个整数对象。另外，在将字符串转换为整数时，格式不符的字符串会导致 Java 抛出异常。

练习 5.3. 仿照例 5.3，写一个 Double 类型的数字、字符串之间相互转换的测试类 DoubleTest.java。

5.2.3 *BigDecimal

`float` 和 `double` 主要是为了科学计算和工程计算而设计的，其数值的表达和计算结果都是一个近似值，比如 `float` 和 `double` 无法精确的表示 0.1（或者 10 的任何负数次方值），因此在需要精确运算的场合不能使用 `float` 和 `double`，比如常见的货币计算 [6, p190]。

例如，假设你现在有 \$ 10.03，花掉了 \$ 4.36，剩下多少钱呢？如果我们用下面的代码片段计算：

```
1 System.out.println(10.03-4.36);
```

结果不是你想要的 5.67，而是 5.6699999999999999。

是不是四舍五入就能解决问题呢？我们看下面的问题：

假设你有 1 元，货架上的铅笔标价分别为 0.1 元、0.2 元、0.3 元，直到 1 元。你打算从 0.1 元/支的开始，每种买一支，直到买不起为止，请问可以买多少支铅笔，找回多少零头？这个题目似乎可以用下面的简单代码解决：

```
1 public static void main(String[] args) {
2     double funds = 1.00;
3     int itemsBought = 0;
4     for(double price = .10; funds >= price; price += .10) {
5         funds -= price;
6         itemsBought++;
7     }
8
9     System.out.println(itemsBought + " items bought.");
10    System.out.println("Charge: $" + funds);
11
12 }
```

问题很简单，显然应该可以买 4 支铅笔，但是当运行这段代码时，其结果为：

```
3 items bought.  
Charge: $ 0.3999999999999999
```

如何正确的解决这个问题呢？或者说，如何精确的表达计算结果呢？答案是使用 `BigDecimal`，或者 `int`、`long` 进行货币运算。使用 `int`、`long` 进行货币运算需要我们自己控制金额的放大系数（小数点位置），而 `BigDecimal` 则可以帮助我们处理小数点问题，并提供了很多计算功能，比如将上面代码中的 `double` 替换为 `BigDecimal` 的后：

```
1 public static void main(String[] args) {
2     BigDecimal funds = new BigDecimal("1.00");
3     final BigDecimal TEN_CENTS = new BigDecimal(".10");
4     int itemsBought = 0;
```

```

5
6   for (BigDecimal price = TEN_CENTS;
7       funds.compareTo(price) >= 0;
8       price = price.add(TEN_CENTS)) {
9     itemsBought++;
10    funds = funds.subtract(price);
11  }
12
13 System.out.println(itemsBought + " items bought.");
14 System.out.println("Money left over: $" + funds);
15
16 }

```

其运行结果如下：

```

4 items bought.
Money left over: $ 0.00

```

练习 5.4. 编写一个程序，将买铅笔问题使用 int 或者 long 处理。提示：找到货币的最小单位，使用最小单位表示单价和金额。

5.2.3.1 BigDecimal 的常见构造方法

见表5.2，按照使用方便性列出了常见的 BigDecimal 构造方法⁶。

5.2.3.2 BigDecimal 提供的计算方法

BigDecimal 不仅可以任意精度的数字，还提供了如表5.3所示的计算方法。

5.2.3.3 BigDecimal 的舍入处理策略

由于 BigDecimal 提供的是精确的数值计算，我们经常需要根据实际情况进行适当的舍入处理，BigDecimal 提供了多达 8 种舍入处理模式，这 8 种舍入模式在 RoundingMode 中定义，如表5.4所示。

可以看出，HALF_UP 即我们最常用的“四舍五入”模式。

下面的代码片段演示了编程实践中的常见情形：

```

1 // 计算结果四舍五入保留2位小数
2 new BigDecimal("123.456").multiply(new BigDecimal("1.23")).setScale(2, RoundingMode.
HALF_UP);

```

⁶ 完整的构造方法列表参见：<http://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html>

| 构造方法 | 描述 | 示例 |
|-------------------------------------|-----------------------------------|---|
| public BigDecimal(String val) | 使用字符串构造 BigDecimal 对象 | new BigDecimal("123.456"); new BigDecimal("-123.456"); new BigDecimal("1.23E3"); |
| public BigDecimal(int val) | 使用基本的整数构造 BigDecimal 对象 | new BigDecimal(123); |
| public BigDecimal(long val) | 使用基本的 long 构造 BigDecimal 对象 | new BigDecimal(123L); |
| public BigDecimal(double val) | 使用基本的 double 构 造 BigDecimal 对象 | new BigDecimal(123.456D); |
| public BigDecimal(char[] in) | 使用字符序列构造 BigDecimal 对象 | char in = {'1','2','3'}; new BigDecimal(in); |

表 5.2: BigDecimal 的常见构造方法

对于任何需要精确答案的计算任务, 请不要使用 float 或者 double。如果你想让系统来记录十进制小数点, 并且不介意因为不使用基本类型而带来的不便, 请使用 BigDecimal。使用 BigDecimal 还有一些额外的好处, 允许你完全控制舍入。每当一个计算涉及到舍入的时候, BigDecimal 允许你 8 种舍入模式中选择其一。但是 BigDecimal 有一定的性能问题, 即比使用 float 和 double 要慢 (大约上百倍的差距)。如果性能非常关键, 那么使用 int 或者 long 处理精确计算问题。要注意的是, int 最多只能处理 9 位十进制数字, long 最多只能处理 18 位十进制数字。如果数值可能超过 18 位, 就必须使用 BigDecimal。

| 方法名称 | 描述 | 示例 |
|---|--------------------------------------|---|
| public BigDecimal add(BigDecimal augend) | 当前的 BigDecimal 对象和 augend 对象相加 | new BigDecimal("123.456").add(new BigDecimal("234.5432")); |
| public BigDecimal subtract(BigDecimal subtrahend) | 当前的 BigDecimal 对象和 subtrahend 对象相减 | new BigDecimal("123.456").subtract(new BigDecimal("234.5432")); |
| public BigDecimal multiply(BigDecimal multiplicand) | 当前的 BigDecimal 对象和 multiplicand 对象相乘 | new BigDecimal("123.456").multiply(new BigDecimal("234.5432")); |
| public BigDecimal divide(BigDecimal divisor) | 当前的 BigDecimal 对象和 divisor 对象相除 | new BigDecimal("123.456").divide(new BigDecimal("234.5432")); |
| public BigDecimal pow(int n) | 当前对象的 n 次方 | new BigDecimal("123.456").pow(3); |

表 5.3: BigDecimal 的计算方法

| 输入数据 | UP | DOWN | CEIL-ING | FLOOR | HALF_UP | HALF_DOWN | HALF_EVEN | UNNECESSARY |
|------|----|------|----------|-------|---------|-----------|-----------|-------------|
| 5.5 | 6 | 5 | 6 | 5 | 6 | 5 | 6 | 抛出异常 |
| 2.5 | 3 | 2 | 3 | 2 | 3 | 2 | 2 | 抛出异常 |
| 1.6 | 2 | 1 | 2 | 1 | 2 | 2 | 2 | 抛出异常 |
| 1.1 | 2 | 1 | 2 | 1 | 1 | 1 | 1 | 抛出异常 |
| 1.0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| -1.0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1.1 | -2 | -1 | -1 | -2 | -1 | -1 | -1 | 抛出异常 |
| -1.6 | -2 | -1 | -1 | -2 | -2 | -2 | -2 | 抛出异常 |
| -2.5 | -3 | -2 | -2 | -3 | -3 | -2 | -2 | 抛出异常 |
| -5.5 | -6 | -5 | -5 | -6 | -6 | -5 | -6 | 抛出异常 |

表 5.4: RoundingMode 中定义的 8 种舍入模式

5.3 日期和时间类

在 Java 语言中，日期和时间曾经是一个复杂和混乱的问题⁷，Java8 终于给 Java 带来了一致的日期和时间实现！Java8 的日期和时间类是 JSR310⁸的一个具体实现，是 Java8 的重要新特性。本书只介绍 Java8 中新的日期和时间类 API，不再涉及旧的日期和时间 API，也建议读者在编写 Java 应用程序的时候不要再使用旧版本的日期和时间 API。

5.3.1 日期操作类：**LocalDate**

顾名思义，**LocalDate** 的“本意”似乎是“本地日期”，但是遗憾的是，**LocalDate** 的本意应该是“日期”。这是因为 Java8 之前的日期类（旧的日期 API）叫做 Date，因此新的日期 API 只好另起炉灶，就被称作“**LocalDate**”了。**LocalDate** 的常见用法见表5.5。

例 5.4. **LocalDate** 用法示例

⁷关于 Java8 之前的日期和时间 API，在互联网上有大量的吐槽，比如：

- JDK BUG 吗？混乱的日期 API: <http://www.importnew.com/20098.html>
- 为什么我们需要新的 Java 日期/时间 API: <http://www.importnew.com/14140.html>
- What's wrong with Java Date & Time API: <http://stackoverflow.com/questions/1969442/whats-wrong-with-java-date-time-api>

⁸参见：<https://jcp.org/en/jsr/detail?id=310>

| 方法 | 描述 |
|---|-------------------------------|
| public static LocalDate now() | 根据默认时区获得当前日期 |
| public static LocalDate of(int year, int month, int dayOfMonth) | 根据给定的年月日创建 LocalDate 对象 |
| public static LocalDate ofYearDay(int year, int dayOfYear) | 根据给定的年份和天数创建 LocalDate 对象 |
| public static LocalDate parse(CharSequence text) | 根据给定的字符串创建 LocalDate 对象 |
| public static LocalDate parse(CharSequence text, DateTimeFormatter formatter) | 根据给定的字符串及其格式创建 LocalDate 对象 |
| public String format(DateTimeFormatter formatter) | 格式化输出日期（参见节 5.3.4 [在第 147 页]） |
| public boolean isAfter(ChronoLocalDate other) | 判断日期对象是否在 other 日期对象之后 |
| public boolean isBefore(ChronoLocalDate other) | 判断日期对象是否在 other 日期对象之前 |

表 5.5: LocalDate 的常见用法

代码设计 参见代码清单5.6。

代码清单 5.6: LocalDateTest.java

```
1 package cn.edu.sdu.tsoftlab.essential.time;
2
3 import java.time.LocalDate;
4 import java.time.Month;
5 import java.time.ZoneId;
6
7 /**
8 * 本类演示了Java8的LocalDate的用法 .
9 *
10 */
11 * @author http://www.importnew.com/14140.html
12 */
13 public class LocalDateTest {
14
15 /**
16 * 程序执行入口.
17 *
18 * @param args 命令行参数
19 */
20 public static void main(String[] args) {
21
22     LocalDate today = LocalDate.now(); // ❶
23     System.out.println("Current Date=" + today);
24
25     LocalDate firstDay2016 = LocalDate.of(2016, Month.JANUARY, 1); // \longremark{给
26         定年月创建特定日期对象}
27     System.out.println("Specific Date=" + firstDay2016);
28
29     //LocalDate feb29_2014 = LocalDate.of(2014, Month.FEBRUARY, 29); // ❷
30     //Current date in "Asia/Kolkata", you can get it from ZoneId javadoc
31     LocalDate todayKolkata = LocalDate.now(ZoneId.of("Asia/Kolkata")); // ❸
32     System.out.println("Current Date in IST=" + todayKolkata);
33
34
35     //LocalDate todayIST = LocalDate.now(ZoneId.of("IST")); // ❹
36     LocalDate dateFromBase = LocalDate.ofEpochDay(365); // ❺
37     System.out.println("365th day from base date= " + dateFromBase);
38
39     LocalDate hundredDay2016 = LocalDate.ofYearDay(2016, 100); // ❻
40     System.out.println("100th day of 2016=" + hundredDay2016);
41
42     LocalDate one = LocalDate.parse("2016-11-21"); //❼
43     LocalDate two = LocalDate.parse("2016-11-22");
44     System.out.println("2016-11-21 parsed to LocalDate = " + one);
45     System.out.println("2016-11-21 < 2016-11-22 ? " + one.isBefore(two));
46     System.out.println("2016-11-21 < 2016-11-21 ? " + one.isBefore(one));
```

```

47     System.out.println("2016-11-22 > 2016-11-21 ? " + two.isAfter(one));
48 }
49 }
50 //
```

- ① 获取当前日期
 - ① 给定日期不合法
 - ② 根据时区获取当前日期
 - ③ 给定时区不合法
 - ④ 从 1970-1-1 开始计算
 - ⑤ 从给定年份开始计算
 - ⑥ 将字符串解析为 LocalDate 对象
-

运行结果 执行 LocalDateTest 结果如下：

```

Current Date=2016-11-20
Specific Date=2016-01-01
Current Date in IST=2016-11-20
365th day from base date= 1971-01-01
100th day of 2016=2016-04-09
2016-11-21 parsed to LocalDate = 2016-11-21
2016-11-21 < 2016-11-22 ? true
2016-11-21 < 2016-11-21 ? false
2016-11-22 > 2016-11-21 ? true
```

5.3.2 时间操作类：LocalTime

LocalTime 是表示时间的操作类，其 API 设计和 LocalDate 很相似，这里不再详细列出 LocalTime 的方法，具体可以参考 exampename5.5。

例 5.5. LocalTime 类用法示例

代码设计 参见代码清单5.7。

代码清单 5.7: LocalTimeTest. java

```

1
2 package cn.edu.sdut.softlab.essentials.time;
3
4 import java.time.LocalTime;
5 import java.time.ZoneId;
6
7 /**
```

```
8  * 本类演示了LocalTime类的常见用法 .
9  *
10 * @author Su Baochen
11 */
12 public class LocalTimeTest {
13
14     /**
15      * 程序执行入口.
16      *
17      * @param args 命令行参数
18      */
19     public static void main(String[] args) {
20
21         LocalTime time = LocalTime.now(); // ①
22         System.out.println("Current Time=" + time);
23
24         LocalTime specificTime = LocalTime.of(12, 20, 25, 40); // ②
25         System.out.println("Specific Time of Day=" + specificTime);
26
27         LocalTime timeKolkata = LocalTime.now(ZoneId.of("Asia/Kolkata")); // ③
28         System.out.println("Current Time in IST=" + timeKolkata);
29
30         //Getting date from the base date i.e 01/01/1970
31         LocalTime specificSecondTime = LocalTime.ofSecondOfDay(10000); // ④
32         System.out.println("10000th second time= " + specificSecondTime);
33
34         LocalTime one = LocalTime.parse("12:30:55"); //⑤
35         LocalTime two = LocalTime.parse("13:02:15");
36         System.out.println("12:30:55 parsed to LocalDate = " + one);
37         System.out.println("12:30:55 < 13:02:15 ? " + one.isBefore(two));
38         System.out.println("12:30:55 < 12:30:55 ? " + one.isBefore(one));
39         System.out.println("13:02:15 > 12:30:55 ? " + two.isAfter(one));
40     }
41 }
42 //
```

① 获取当前时间

② 根据给定时间创建时间对象

③ 根据给定时区创建时间对象

④ 获得从 1970-1-1 开始计算的时间

⑤ 将字符串解析为 LocalTime 对象

运行结果 执行 LocalTimeTest 结果如下：

```
Current Time=15:07:13.774
Specific Time of Day=12:20:25.0000000040
Current Time in IST=12:37:13.775
10000th second time= 02:46:40
12:30:55 parsed to LocalDate = 12:30:55
12:30:55 < 13:02:15 ? true
12:30:55 < 12:30:55 ? false
13:02:15 > 12:30:55 ? true
```

5.3.3 日期时间类: **LocalDateTime**

LocalDateTime 是表示日期时间的操作类，其 API 设计和 LocalDate 很相似，这里不再详细列出 LocalDateTime 的方法，具体可以参考例5.6。

例 5.6. LocalDateTime 类用法示例

代码设计 参见代码清单5.8。

代码清单 5.8: LocalDateTimeTest.java

```
1
2 package cn.edu.sdut.softlab.essentials.time;
3
4 import java.time.LocalDate;
5 import java.time.LocalDateTime;
6 import java.time.LocalTime;
7 import java.time.Month;
8 import java.time.ZoneId;
9 import java.time.ZoneOffset;
10
11 /**
12  * 本类演示了LocalDateTime类的用法 .
13  *
14  * @author Su Baochen
15  */
16 public class LocalDateTimeTest {
17
18     /**
19      * 程序执行入口.
20      *
21      * @param args 命令行参数
22      */
23     public static void main(String[] args) {
24
25         LocalDateTime today = LocalDateTime.now(); // ❶
```

```
26     System.out.println("Current DateTime=" + today);
27
28     //Current Date using LocalDate and LocalTime
29     today = LocalDateTime.of(LocalDate.now(), LocalTime.now()); // ❶
30     System.out.println("Current DateTime=" + today);
31
32     LocalDateTime specificDate = LocalDateTime.of(2014, Month.JANUARY, 1, 10, 10,
33         30); // ❷
34     System.out.println("Specific Date=" + specificDate);
35
36     LocalDateTime todayKolkata = LocalDateTime.now(ZoneId.of("Asia/Kolkata")); // ❸
37     System.out.println("Current Date in IST=" + todayKolkata);
38
39     LocalDateTime dateFromBase = LocalDateTime.ofEpochSecond(10000, 0, ZoneOffset.
40         UTC); // ❹
41     System.out.println("10000th second time from 01/01/1970= " + dateFromBase);
42 }
43 //
```

- ❶ 获得当前的日期时间对象
- ❷ 根据给定的 `LocalDate` 和 `LocalTime` 创建日期时间对象
- ❸ 根据给定的时区创建日期时间对象
- ❹ 从 `1970-1-1` 开始计算的日期时间对象

运行结果 执行 `LocalDateTimeTest` 结果如下：

```
Current DateTime=2016-11-20T15:08:15.970
Current DateTime=2016-11-20T15:08:15.971
Specific Date=2014-01-01T10:10:30
Current Date in IST=2016-11-20T12:38:15.972
10000th second time from 01/01/1970= 1970-01-01T02:46:40
```

5.3.4 日期和时间的格式化输出类：`DateTimeFormatter`

在大部分情况下，`LocalDate` 和 `LocalTime` 的默认输出即符合预期，参见代码清单5.6和代码清单5.7中 `LocalDate` 和 `LocalTime` 对象的输出。如果需要定制日期和时间的输出格式，`DateTimeFormatter` 提供了丰富的内置格式支持和可定制选项。`DateTimeFormatter` 内置的输出格式如表5.6所示。

`DateTimeFormmater` 的可定制选项请参考 `DateTimeFormmater` 的 API 文档。

例 5.7. 使用 `DateTimeFormmater` 定制日期和时间的输出

| 格式控制选项 | 描述 | 示例 |
|--|--|--|
| ofLocalizedDate(dateStyle) | Formatter with date style from the locale | '2011-12-03' |
| ofLocalizedTime(timeStyle) | Formatter with time style from the locale | '10:15:30' |
| ofLocalizedDateTime(dateTimeStyle) | Formatter with a style for date and time from the locale | '3 Jun 2008 11:05:30' |
| ofLocalizedDateTime(dateStyle,timeStyle) | Formatter with date and time styles from the locale | '3 Jun 2008 11:05' |
| BASIC_ISO_DATE | Basic ISO date | '20111203' |
| ISO_LOCAL_DATE | ISO Local Date | '2011-12-03' |
| ISO_OFFSET_DATE | ISO Date with offset | '2011-12-03+01:00' |
| ISO_DATE | ISO Date with or without offset | '2011-12-03+01:00'; '2011-12-03' |
| ISO_LOCAL_TIME | Time without offset | '10:15:30' |
| ISO_OFFSET_TIME | Time with offset | '10:15:30+01:00' |
| ISO_TIME | Time with or without offset | '10:15:30+01:00'; '10:15:30' |
| ISO_LOCAL_DATE_TIME | ISO Local Date and Time | '2011-12-03T10:15:30' |
| ISO_OFFSET_DATE_TIME | Date Time with Offset | '2011-12-03T10:15:30+01:00' |
| ISO_ZONED_DATE_TIME | Zoned Date Time | '2011-12-03T10:15:30+01:00[Europe/Paris]' |
| ISO_DATE_TIME | Date and time with ZoneId | '2011-12-03T10:15:30+01:00[Europe/Paris]' |
| ISO_ORDINAL_DATE | Year and day of year | '2012-337' |
| ISO_WEEK_DATE | Year and Week | '2012-W48-6' |
| ISO_INSTANT | Date and Time of an Instant | RFC_1123_DATE_TIME RFC 1123 / RFC 822 'Tue, 3 Jun 2008 11:05:30 GMT' |

表 5.6: DateTimeFormatter 的输出格式控制选项

代码设计 参见代码清单5.9。

代码清单 5.9: DateTimeFormatterTest.java

```
1 package cn.edu.sdu.softlab.essentials.time;
2
3
4 import java.time.Instant;
5 import java.time.LocalDate;
6 import java.time.LocalDateTime;
7 import java.time.format.DateTimeFormatter;
8
9 /**
10  * 本类演示了DateTimeFormatter的常见用法 .
11  *
12  * @author Su Baochen
13  */
14 public class DateTimeFormatterTest {
15
16     /**
17      * 程序执行入口.
18      *
19      * @param args 命令行参数
20      */
21     public static void main(String[] args) {
22
23         //Format examples
24         LocalDate date = LocalDate.now();
25         //default format
26         System.out.println("Default format of LocalDate=" + date);
27         //specific format
28         System.out.println(date.format(DateTimeFormatter.ofPattern("d::MMM::uuuu")));
29         System.out.println(date.format(DateTimeFormatter.BASIC_ISO_DATE));
30
31         LocalDateTime dateTime = LocalDateTime.now();
32         //default format
33         System.out.println("Default format of LocalDateTime=" + dateTime);
34         //specific format
35         System.out.println(dateTime.format(DateTimeFormatter.ofPattern("d::MM::uuuu HH
36             ::mm::ss")));
37         System.out.println(dateTime.format(DateTimeFormatter.BASIC_ISO_DATE));
38
39         //Parse examples
40         LocalDateTime dt = LocalDateTime.parse("27::04::2014 21::39::48",
41             DateTimeFormatter.ofPattern("d::MM::uuuu HH::mm::ss"));
42         System.out.println("Default format after parsing = " + dt);
43     }
44 }
```

运行结果 运行 DateTimeFormatterTest 结果如下：

```
Default format of LocalDate=2016-11-20
20:: 十一月::2016
20161120
Default format of LocalDateTime=2016-11-20T15:18:44.241
20:: 十一月::2016 15::18::44
20161120
Default format after parsing = 2014-04-27T21:39:48
```

5.3.5 日期和时间的调整

大多数日期/时间 API 类都实现了一系列工具方法，如：加/减天数、周数、月份数，等等。还有其他的工具方法能够使用 TemporalAdjuster 调整日期，并计算两个日期间的周期。

例 5.8. 使用日期和时间类调整日期和时间

代码设计 参见代码清单5.10。

代码清单 5.10: DateTimeAPITest.java

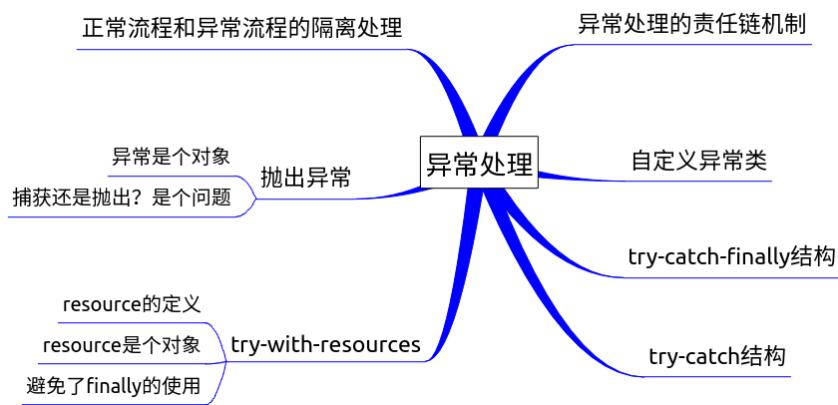
```
1
2 package cn.edu.sdut.softlab.essentials.time;
3
4 import java.time.LocalDate;
5 import java.time.LocalDateTime;
6 import java.time.Period;
7 import java.time.temporal.TemporalAdjusters;
8
9 /**
10  * 本类演示了日期时间类的调整计算方法 .
11  *
12  * @author Su Baochen
13  */
14 public class DateTimeAPITest {
15
16     /**
17      * 程序执行入口.
18      *
19      * @param args 命令行参数
20      */
21     public static void main(String[] args) {
22
23         LocalDate today = LocalDate.now();
24
25         //Get the Year, check if it's leap year
```

```
26     System.out.println("Year " + today.getYear() + " is Leap Year? " + today.  
27         isLeapYear());  
28  
29     //Compare two LocalDate for before and after  
30     System.out.println("Today is before 01/01/2015? " + today.isBefore(LocalDate.of  
31         (2015, 1, 1)));  
32  
33     //Create LocalDateTime from LocalDate  
34     System.out.println("Current Time=" + today.atTime(LocalTime.now()));  
35  
36     //plus and minus operations  
37     System.out.println("10 days after today will be " + today.plusDays(10));  
38     System.out.println("3 weeks after today will be " + today.plusWeeks(3));  
39     System.out.println("20 months after today will be " + today.plusMonths(20));  
40  
41     System.out.println("10 days before today will be " + today.minusDays(10));  
42     System.out.println("3 weeks before today will be " + today.minusWeeks(3));  
43     System.out.println("20 months before today will be " + today.minusMonths(20));  
44  
45     //Temporal adjusters for adjusting the dates  
46     System.out.println("First date of this month= " + today.with(TemporalAdjusters.  
47         firstDayOfMonth()));  
48     LocalDate lastDayOfYear = today.with(TemporalAdjusters.lastDayOfYear());  
49     System.out.println("Last date of this year= " + lastDayOfYear);  
50  
51     Period period = today.until(lastDayOfYear);  
52     System.out.println("Period Format= " + period);  
53     System.out.println("Months remaining in the year= " + period.getMonths());  
54 }
```

运行结果 运行 DateTimeAPITest 结果如下：

```
Year 2016 is Leap Year? true
Today is before 01/01/2015? false
Current Time=2016-11-20T15:26:02.947
10 days after today will be 2016-11-30
3 weeks after today will be 2016-12-11
20 months after today will be 2018-07-20
10 days before today will be 2016-11-10
3 weeks before today will be 2016-10-30
20 months before today will be 2015-03-20
First date of this month= 2016-11-01
Last date of this year= 2016-12-31
Period Format= P1M11D
Months remaining in the year= 1
```

第六章 异常处理



6.1 异常的概念

“人生不如意事十之八九”¹，编程同样道理：世事纷繁复杂，情况千变万化，代码要正确运行，不仅要处理正常的流程，更要处理多样化的非正常流程。比如常见的情形：

- 街头的投币电话，总有人尝试用游戏币看看能不能蒙混过关；
- 长城的古砖墙是为御敌而生，却被人刻上“到此一游”；
- 机动车道上的行人，人行道上的机动车；

回到现实的程序中，在5.3 [在第141页]中，我们看到日期和时间类都定义了一个`parse`方法，能够将字符串解析为`LocalDate`、`LocalTime`或者`LocalDateTime`对象，很直观，很方便，因此在编程实践中很常见。试运行下面的示例：

例 6.1. 从键盘输入字符串创建日期对象

代码设计 参见代码清单6.1。

¹南宋词人辛弃疾词《贺新郎·用前韵再赋》：“叹人生、不如意事，十常八九。”

代码清单 6.1: WhatIfNoException.java

```
1 package cn.edu.sdut.softlab.exception;
2
3 import java.time.LocalDate;
4 import java.util.Scanner;
5
6 public class WhatIfNoException {
7
8     public static void main(String[] args) {
9         Scanner console = new Scanner(System.in);
10        System.out.print("Please input date:");
11        String str = console.nextLine();
12
13        while (!str.equalsIgnoreCase("*")) { // 输入*表示结束
14            System.out.println(LocalDate.parse(str));
15            System.out.print("Please input date:");
16            str = console.nextLine();
17        }
18    }
19 }
```

运行结果 运行 WhatIfNoException 结果如下：

```
Please input date:2016-11-22
2016-11-22
Please input date:2016-13-22
Exception in thread "main" java.time.format.DateTimeParseException:
Text '2016-13-22' could not be parsed: Invalid value for MonthOfYear (valid values 1 - 12): 13
at java.time.format.DateTimeFormatter.createError(DateTimeFormatter.java:1920)
.....
at cn.edu.sdut.softlab.exception.WhatIfNoException.main(WhatIfNoException.java:30)
Caused by: java.time.DateTimeException: Invalid value for MonthOfYear (valid values 1 - 12): 13
at java.time.temporal.ValueRange.checkValidIntValue(ValueRange.java:330)
.....
... 3 more
```

在例6.1中，从键盘输入的字符串如果符合 LocalDate 的格式要求自然一切正常，但是我们不能总是期望人们每次输入都遵循 LocalDate 的格式，无论人们是有意还是无意，总会有破坏规则的时候，比如输入的月份大于 12，天数大于 31 等，显然，错误的

输入是无法获得正确的 LocalDate 对象的，Java 编译器也无法自动纠正这种类型的错误。对于这种情况，Java 友好的给出了错误的原因及其关联的程序代码（stack trace），以帮助程序员快速定位错误代码的位置，更快的修复问题。

实际上，类似例6.1的错误提示方式，对于程序员而言是友好的，但是对于终端用户而言是不友好的：终端用户并不关心程序哪一行出错了。对于终端用户而言，程序应该给出更人性化的提示，比如提示“日期格式有错误，月份应该在 1-12 之间”等等。我们将在 6.3 中介绍如何更人性化的处理。

6.2 C 中的异常处理方式

我们先回顾一下在 C 语言中如何判断用户的输入是否合法，大致应该是如下的代码片段：

```
1 if(month < 1 || month > 12) {  
2     printf("wrong month num:%d",month);  
3     return -1;  
4 }  
5  
6 if(day < 1 || day > 31) { /* 没有考虑二月份情况 */  
7     printf("wrong day num:%d",day);  
8     return -1;  
9 }  
10 ...  
11 /* 用户的输入合法，下面继续处理正常流程 */
```

也就是说，我们只能通过一系列的 if 条件判断来分析用户的输入是否合法，如果可能的异常情形比较多，势必会存在更多的 if 条件判断，造成程序的流程不清晰：哪些是正常流程处理代码，哪些是异常流程处理代码，除非通过注释标识出来，否则很难一眼看清楚。

6.3 Java 的异常处理方式

当然，在 Java 中我们依然可以像 C 语言那样去处理异常。不过，Java 提供了更棒的异常处理机制：将异常情况抽象为异常对象，并采用责任链处理机制处理异常对象。

Java 内置了常见的异常类，如图6.1所示。

当异常情况出现时，Java 虚拟机能够“捕获”（catch）²异常对象，并根据异常对象的类型进行相应的处理。Java 将异常类分为两大类：

²捕获异常的方式？

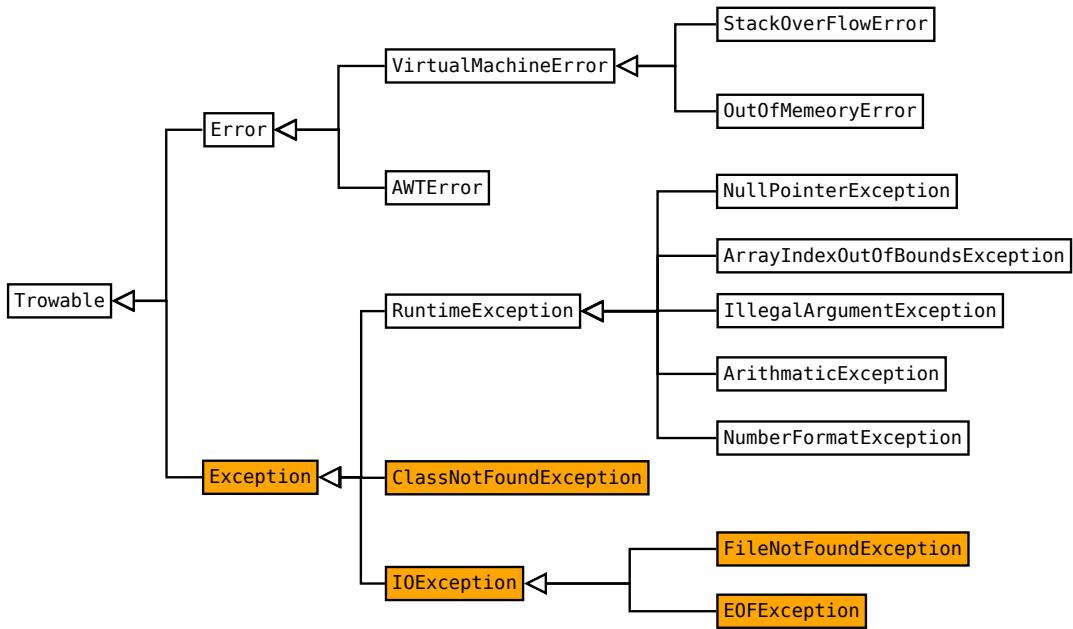


图 6.1: Java 的异常类层次结构

- 非检查型：即 Java 虚拟机能够处理的异常类型，换言之，这种类型的异常不需要在程序中捕获，一旦发生这种类型的异常，Java 虚拟机有一套内置的处理方式，比如除数为零、数组越界、非法参数等异常类型。对于图6.1，`Error` 和 `RuntimeException` 的子类都是非检查型异常。在程序中可以不捕获非检查型异常，也可以捕获非检查型异常。捕获非检查型异常的目的往往是给用户提供更友好的出错提示，单这不是强制的。
- 检查型：即 Java 虚拟机无法自动处理的异常类型，需要在程序中捕获（Catch）。对于图6.1，`ClassNotFoundException` 和 `IOException` 的子类都是检查型异常。要注意的是，检查型异常是必须捕获的，否则会导致语法错误，因此在程序中捕获检查型异常是强制的。

6.4 捕获异常

我们首先简单回顾一下 Java 虚拟机和 Java 应用程序的关系，如图6.2所示。

Java 应用程序编译为 class 文件后，经过 Java 虚拟机的类加载器解析 class 文件在内存中创建相应的存储模型然后执行该应用程序。也就是说，Java 虚拟机完全掌控 Java 应用程序的执行过程，包括应用程序出现异常情况时。Java 通过如下的程序结构捕获异常：

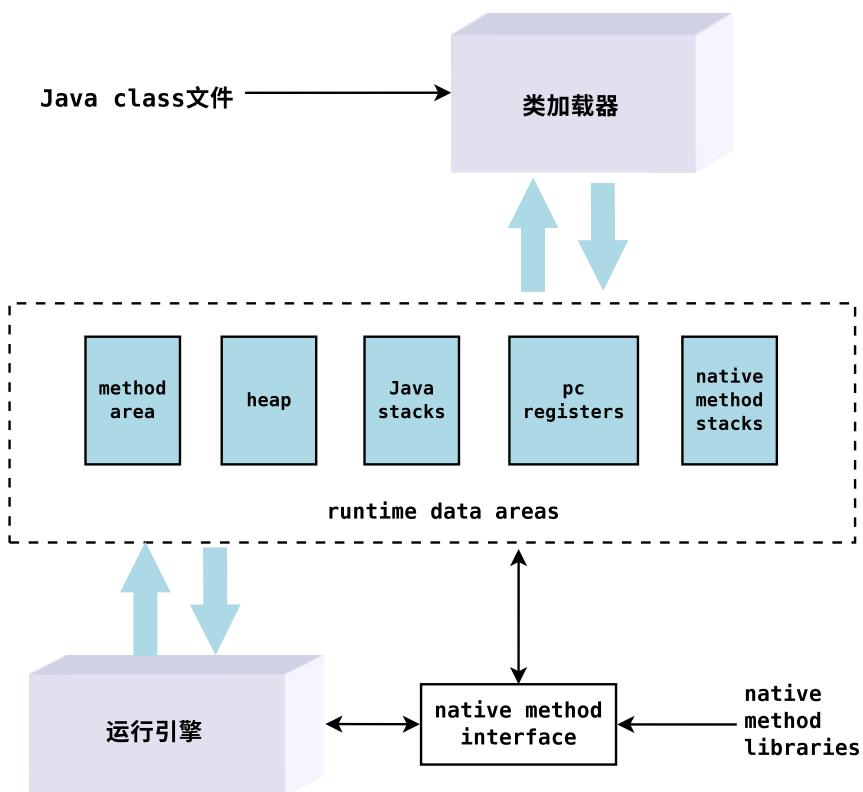


图 6.2: Java 虚拟机和 Java 应用程序的关系

```

2 // 一段可能存在SomeException类型异常的代码
3 } catch (SomeException e) {
4     e.printStackTrace();
5 }

```

在 try 块中的代码就是所谓的“正常流程”，在 catch 块中的代码就是所谓的“异常流程”。如果异常流程有多种，就存在多个 catch 块：

```

1 try {
2     // 一段可能存在SomeException类型异常的代码
3 } catch (SomeException e) {
4     e.printStackTrace();
5 } catch (OtherException e) {
6     e.printStackTrace();
7 } catch (FooException e) {
8     e.printStackTrace();
9 }

```

对比 C 语言中使用 if...else 结构的异常处理方式，Java 的 try...catch 更清晰的区分了正常流程和异常流程：通过不同的异常类也清晰的表达了异常的类型，甚至无需借助注释，只是根据异常类的名字，我们也很容易判断异常的类型及其大致的处理方式。

例 6.2. 捕获异常

代码设计 参见代码清单6.2。

运行结果 此例的运行需要配置命令行参，配置方法参见 B.1 [在第 325 页]

当命令行参数为“24 3”时的输出结果如下：

您输入的两个数相除的结果是：8

修改命令行参数为“24 3.5”可以看到输出变为：

数字格式异常，程序只能接受整数形式的参数

```

java.lang.NumberFormatException: For input string: "3.5"
at java.lang.NumberFormatException.forInputString(NumberFormatException
.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at cn.edu.sdu.softlab.exception.DivTest.main(DivTest.java:34)

```

修改命令行参数为“24 0”，输出结果为：

代码清单 6.2: DivTest.java

```
1 package cn.edu.sdu.t.softlab.exception;
2
3 /**
4  * 本类演示了多个catch块的情形 .
5  * @author 《疯狂Java讲义（第二版）》P350
6 */
7 public class DivTest {
8
9 /**
10 * 程序执行入口.
11 * @param args 命令行参数
12 */
13 public static void main(String[] args) {
14     try {
15         int a = Integer.parseInt(args[0]);
16         int b = Integer.parseInt(args[1]);
17         int c = a / b;
18         System.out.println("您输入的两个数相除的结果是: " + c);
19
20     } catch (ArrayIndexOutOfBoundsException e) {
21         System.out.println("数组越界, 运行程序时输入的参数个数不对。应该输入2个参数, 您输入的参数个数
22             是: " + args.length);
23         e.printStackTrace();
24     } catch (NumberFormatException e) {
25         System.out.println("数字格式异常, 程序只能接受整数形式的参数");
26         e.printStackTrace();
27     } catch (ArithmetricException e) {
28         System.out.println("算数异常, 除数不能为0");
29         e.printStackTrace();
30     } catch (Exception e) {
31         System.out.println("我不知道发生了什么, 总是情况不对");
32         e.printStackTrace();
33     }
34 }
```

算数异常，除数不能为 0

```
java.lang.ArithmetricException: / by zero  
at cn.edu.sdut.softlab.exception.DivTest.main(DivTest.java:35)
```

修改命令行参数为“24”，输出结果为：

数组越界，运行程序时输入的参数个数不对。应该输入 2 个参数，您输入的参数个数是：

1

```
java.lang.ArrayIndexOutOfBoundsException: 1  
at cn.edu.sdut.softlab.exception.DivTest.main(DivTest.java:34)
```

修改命令行参数为“24.5”，输出结果为：

数字格式异常，程序只能接受整数形式的参数

```
java.lang.NumberFormatException: For input string: "24.5"  
at java.lang.NumberFormatException.forInputString(NumberFormatException.  
java:65)  
at java.lang.Integer.parseInt(Integer.java:580)  
at java.lang.Integer.parseInt(Integer.java:615)  
at cn.edu.sdut.softlab.exception.DivTest.main(DivTest.java:33)
```

代码分析 可以看出，Java 遇到异常时即退出整个应用程序。也就是说，尽管我们在代码中可以捕获多个异常，但是当一个异常发生时就会退出整个应用程序。

但是，`ArrayIndexOutOfBoundsException`、`NumberFormatException`、`ArithmetiException` 都是运行时异常 (`RuntimeException`)，即非检查型异常，我们在代码中其实不需要主动捕获，当异常发生时虚拟机会自动处理。虚拟机的一般处理策略是打印出异常发生时的调用栈，供程序员追查和排错。主动捕获运行时异常的好处是可以给终端用户更友好的错误提示。

所有的 Java 异常类都是 `Throwable` 的子类，`Throwable` 类的下列方法能够帮助我们更详细的了解异常的情况：

- `printStackTrace()`: 打印出异常发生时的调用栈 (call stack)，对于程序员排错特别有用，因此在 `catch` 块中经常看到调用 `printStackTrace`。
- `getMessage()`: 返回描述异常的一个字符串，对于终端用户更友好一些。但是，默认情况下，`getMessage` 返回的就是 `stackTrace` 的内容，因此，调用此方法显得更友好的前提条件是，在创建异常对象时设置了描述异常的字



字符串，通常在用户自定义异常中使用，参见：section §6.6。



现代的 IDE，包括 NetBeans，Eclipse 都能够自动检测代码是否应该捕获异常，因此无需记忆哪些代码应该使用 try...catch 结构包围起来，大部分情况下遵从 IDE 的建议即可。但是，有的时候也需要手工组织一下异常处理的代码层次。

6.5 抛出异常

在6.4中，我们捕获（catch）的异常是从哪里来的呢？或者说，我们为什么能够捕获到异常（对象）？异常对象是谁创建的呢？

异常对象当然不是从石头缝里蹦出来的，事实上，我们之所以能够捕获某个异常对象，是因为在 try 代码块中的某个方法调用抛出（throw）了异常对象，即某个方法在运行中探测到发生了异常状况，因此创建了异常对象并抛出。

例 6.3. 抛出异常示例

代码设计 参见代码清单6.3

运行结果 运行 Example 结果如下：

就是这么二！

代码分析 在 add 方法中，我们故意制造了一个异常：当给定的参数是 2 时就抛出异常（throw new Exception()）。所谓抛出异常，就是创建某种类型的异常对象，通过 throw 关键字抛出即可，所以抛出异常的表达方式通常是：

```
1 if(something bad happened) {  
2     throw new SomethingBadException();  
3 }
```

由于在 add 方法中我们可能抛出异常，因此 add 方法必须声明抛出了哪些类型的异常，通过 throws 关键字列出在方法的参数列表后面：

```
1 public void add(int val) throws Exception {...
```

add 方法在 f 方法中被调用，但是 f 方法并没有捕获 add 方法抛出的异常，因此 f 方法也必须在方法中声明抛出异常：

```
1 public void f() throws Exception {...
```

代码清单 6.3: Example.java

```
1 package cn.edu.sdut.softlab.exception;
2
3 /**
4  * 本类演示了throw和throws的用法.
5  * 本例原始版本来自: http://faculty.ycp.edu/~dhovemey/spring2007/cs201/info/exceptionsFileIO.html
6  * @author subaochen
7 */
8 public class Example {
9
10    private int count = 0;
11
12    public static void main(String[] argv) {
13        try {
14            Example ex = new Example();
15            ex.f();
16            System.out.println(ex.count);
17        } catch (Exception e) {
18            System.out.println(e.getMessage());
19        }
20    }
21
22    public void f() throws Exception {
23        add(2);
24        add(3);
25    }
26
27    public void add(int val) throws Exception {
28        if (val == 2) {
29            throw new Exception("就是这么二! ");
30        }
31        count += val;
32    }
33
34    public void mult(int val) {
35        count *= val;
36    }
37 }
```

在 main 方法中我们捕获了 f 方法中抛出的异常（其实是 add 方法抛出的异常），因此 main 方法就不需要抛出异常了。

图6.3形象的表达了抛出异常和捕获异常的联系，图中灰色代码由于异常的关系没有执行到。

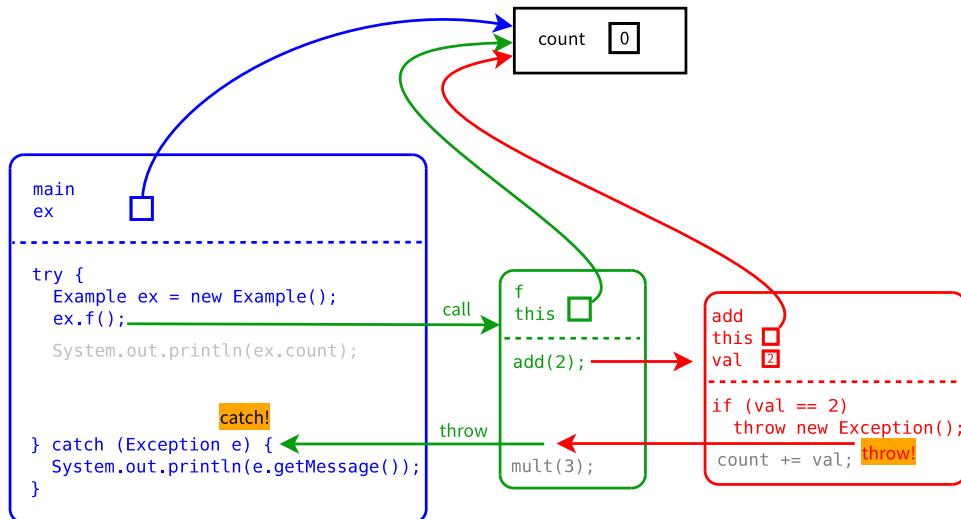


图 6.3: 抛出异常和捕获异常的关系

练习 6.1. 在例6.3中，如果 main 方法中也不捕获异常，需要做怎样的代码变动？

练习 6.2. 在例6.3中，修改 f 方法捕获 add 方法抛出的异常。

练习 6.3. 在例6.3中，修改 add 方法，使得抛出异常部分代码为：

```
1 if( val == 2 ) throw new Exception();
```

则代码的其他部分应该如何修改更合理？

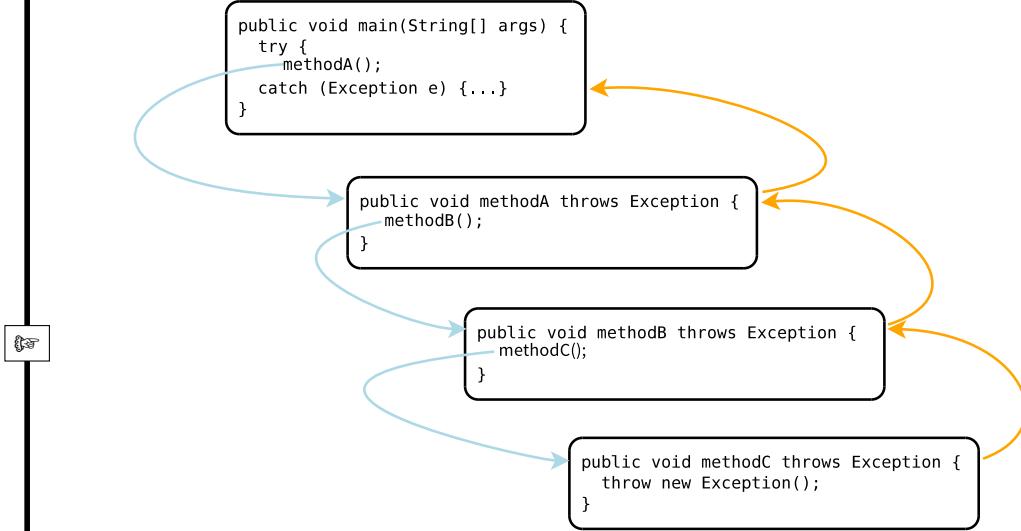
可以看出，Java 的异常处理其实也离不开 if 条件判断，比如在 add 方法中：

```
1 if (val == 2)
2   throw new Exception("就是这么二！");
```

只不过，在 Java 中通过 try...catch 和 throw 机制将杂乱无章的 if 条件判断良好的组织起来，层次更加清晰罢了。

要注意区分 throw 和 throws 的不同用法。throw 用于在代码中创建异常对象并抛出异常，throws 只是在方法中声明本方法可能抛出哪些异常。在方法中声明抛出了某种类型的异常，表明这个方法没有处理异常，将处理异常的责任“抛”给了

方法的调用者。如果方法的调用者也不处理这个异常，则调用者同样需要 throws 这个异常，如此形成了一个异常处理的“责任链”，如下图所示。



可以想见，`main` 方法是我们程序员处理异常的最后“关口”，即，如果我们在 `main` 方法中也不处理（捕获）异常的话，`main` 方法同样需要声明 `throws` 这些异常。`main` 方法抛出的异常就只有 java 虚拟机可以处理了，这通常不是一个好的习惯，因为 Java 虚拟机只能按照默认的异常处理方式打印出调用栈，在终端用户看来，调用栈没有任何价值，徒增抱怨而已。

6.6 用户自定义异常

异常类代表了一种异常的类型，自然我们也可以根据实际的业务逻辑自定义异常类，通常是从 `Exception` 类继承下来即可。比如在电子商务的业务流程中，“下订单”的过程可能会遇到以下的异常情况：

- 库存不足：所订购的商品没有及时付款，提交订单时库存不足了，导致提交失败。我们定义这种异常为 `OutOfInventoryException`。
- 价格变更：提交订单时商品的价格已经改变了，此时应该阻止提交订单。我们定义这种异常为 `PriceNotAvailableException`。

首先我们定义这两个异常类，如代码清单6.4和代码清单6.5所示。

在主类 `CustomerOrder.java` 中，我们

在 NetBeans IDE 中，可以通过“source”菜单方便的自动产生 `OutOfInventoryException` 的构造方法：选择“Insert Code...”，在随后弹出的窗口中选择“constructor”，然后选择重写 `Exception(String ex)` 方法即可。

代码清单 6.4: OutOfInventoryException.java

```
1 /*
2  * Copyright 2016 Su Baochen and individual contributors by the
3  * @authors tag. See the copyright.txt in the distribution for
4  * a full listing of individual contributors.
5  *
6  * Licensed under the Apache License, Version 2.0 (the "License");
7  * you may not use this file except in compliance with the License.
8  * You may obtain a copy of the License at
9  *
10 *     http://www.apache.org/licenses/LICENSE-2.0
11 *
12 * Unless required by applicable law or agreed to in writing, software
13 * distributed under the License is distributed on an "AS IS" BASIS,
14 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 * See the License for the specific language governing permissions and
16 * limitations under the License.
17 */
18
19 package cn.edu.sdut.softlab.exception.eshop;
20
21 /**
22  * 库存不足异常类.
23  * @author Su Baochen
24 */
25 public class OutOfInventoryException extends Exception {
26
27     public OutOfInventoryException(String message) {
28         super(message);
29     }
30 }
```

代码清单 6.5: PriceNotAvailableException.java

```
1 /*
2  * Copyright 2016 Su Baochen and individual contributors by the
3  * @authors tag. See the copyright.txt in the distribution for
4  * a full listing of individual contributors.
5  *
6  * Licensed under the Apache License, Version 2.0 (the "License");
7  * you may not use this file except in compliance with the License.
8  * You may obtain a copy of the License at
9  *
10 *     http://www.apache.org/licenses/LICENSE-2.0
11 *
12 * Unless required by applicable law or agreed to in writing, software
13 * distributed under the License is distributed on an "AS IS" BASIS,
14 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 * See the License for the specific language governing permissions and
16 * limitations under the License.
17 */
18
19 package cn.edu.sdut.softlab.exception.eshop;
20
21 /**
22  * 价格已变更异常类.
23  * @author Su Baochen
24 */
25 public class PriceNotAvailableException extends Exception {
26
27     public PriceNotAvailableException(String message) {
28         super(message);
29     }
30 }
```

6.7 finally

try-catch 结构完美的诠释了异常处理的一般过程, 不过我们考虑一种特殊情况: 如果我们希望在执行完 try 代码块之后总是执行一段代码(不妨称之为“清理代码块”), 即无论 try 代码块是否存在异常, 清理代码块总是要执行的。finally 即为此而设置, 一般结构为:

```

1 try {
2     // 正常流程
3 } catch (...) {
4     // 异常流程1
5 } catch (...) {
6     // 异常流程2
7 } finally {
8     // 清理代码块
9 }
```

在网络通讯和数据库编程中, 这种情形很常见: 无论是否发生异常, 网络链接和数据库链接总是要断开的(`close`), 因此很适合在 finally 代码块中处理断开网络链接和数据库链接。我们在也将在 chapter 7 看到很多 finally 使用的实例。



在 section §6.8 我们可以看到, 使用 `try with resources` 技术可以避免使用 finally, 这样即简化了代码, 也提高了代码的可读性, 值得提倡。

练习 6.4. 编写一个包含 finally 的程序, 说明即使发生了异常, finally 代码块也是会被执行到的。

6.8 try with resources³

在 section §6.7 中我们看到, 对于必须执行的“清理代码块”, 我们可以使用 finally 来保证这一点。Java8 更进了一步: 如果一个对象实现了 `java.lang.AutoCloseable` 接口⁴或者 `Closeable` 接口, 则可以通过 `try-with-resources` 结构来保证代码结束时自动关闭这个对象。实现了 `java.lang.AutoCloseable` 接口(或者 `Closeable` 接口)的对象被称为 `resource`。`try-with-resources` 结构的一般形式如下:

```

1 try (resource1; resource2) {
2     // 正常流程
3 } catch (...) {
4     // 异常流程
5 }
```

³本部分示例代码参见: <http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html>

⁴由于 `AutoCloseable` 接口只有一个抽象方法 `close`, 因此 `AutoCloseable` 接口是一个函数接口, 参见: ??。

比如下面的例子读取文件的第一行：

```

1 static String readFirstLineFromFile(String path) throws IOException {
2     try (BufferedReader br = new BufferedReader(new FileReader(path))) {
3         return br.readLine();
4     }
5 }
```

BufferedReader 是一个实现了 AutoCloseable 接口的 resource (资源)，因此我们把创建 BufferedReader 对象的工作放到了 try-with-resources 结构中，这样无论 br.readLine 是否正常执行，当 try-with-resources 代码块执行完毕后，BufferedReader 对象 br 都会被自动关闭。如果我们不使用 try-with-resources 结构的话，则需要这样编写同样功能的代码：

```

1 static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {
2     BufferedReader br = new BufferedReader(new FileReader(path));
3     try {
4         return br.readLine();
5     } finally {
6         if (br != null) br.close();
7     }
8 }
```

可见，使用 try-with-resources 结构不仅简化了代码，也提高了代码的可读性。

下面的例子在 try-with-resources 中声明了多个 resources：

```

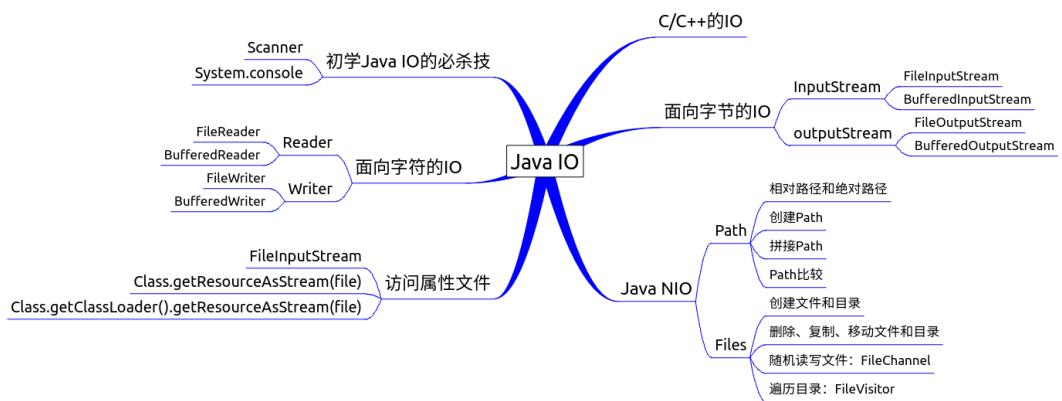
1 public static void writeToFileZipFileContents(String zipFileName,
2                                                 String outputFileName)
3                                                 throws java.io.IOException {
4
5     java.nio.charset.Charset charset =
6         java.nio.charset.StandardCharsets.US_ASCII;
7     java.nio.file.Path outputPath =
8         java.nio.file.Paths.get(outputFileName);
9
10    // Open zip file and create output file with
11    // try-with-resources statement
12
13    try (
14        java.util.zip.ZipFile zf =
15            new java.util.zip.ZipFile(zipFileName);
16        java.io.BufferedWriter writer =
17            java.nio.file.Files.newBufferedWriter(outputPath, charset)
18    ) {
19        // Enumerate each entry
20        for (java.util.Enumeration entries =
21            zf.entries(); entries.hasMoreElements();) {
22            // Get the entry name and write it to the output file
23            String.newLine = System.getProperty("line.separator");
24            String zipEntryName =
```

```
25         ((java.util.zip.ZipEntry)entries.nextElement()).getName() +  
26             newLine;  
27     writer.write(zipEntryName, 0, zipEntryName.length());  
28 }  
29 }  
30 }
```

使用 try-with-resources 结构编写数据库访问程序：

```
1 public static void viewTable(Connection con) throws SQLException {  
2  
3     String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";  
4  
5     try (Statement stmt = con.createStatement()) {  
6         ResultSet rs = stmt.executeQuery(query);  
7  
8         while (rs.next()) {  
9             String coffeeName = rs.getString("COF_NAME");  
10            int supplierID = rs.getInt("SUP_ID");  
11            float price = rs.getFloat("PRICE");  
12            int sales = rs.getInt("SALES");  
13            int total = rs.getInt("TOTAL");  
14  
15            System.out.println(coffeeName + ", " + supplierID + ", " +  
16                                price + ", " + sales + ", " + total);  
17        }  
18    } catch (SQLException e) {  
19        JDBCUtilities.printSQLException(e);  
20    }  
21 }
```

第七章 Java 的 IO



7.1 C 的 IO 回顾

在 C 语言中，输入输出的概念分为两个层面：

- 输入输出的“源”都被看做设备，使用设备描述符来区分不同的输入输出源。
- 从设备输入或者输出的数据通过“流”（stream）模型来处理，参见图7.1和图7.2。

所谓的“流”（stream），是一个仅容一个 bit 数据通过的管道，即数据的有序序列，如图7.3所示。

“流模型”的最大好处是，每个 I/O 函数都尽力做最好的自己即可，通过“流模型”可以将不同的 I/O 函数串联起来协同完成更复杂的 IO 操作。Java 的 IO 体系也借鉴了 C 中的 IO 设计，只不过 C 是面向过程的语言，IO 的处理是通过函数来完成的，而 Java 是面向对象的方式来处理 IO，IO 的操作是通过不同的 IO 操作类来完成的。

7.2 Java 的 IO 体系

Java 继承了 C 对输入输出的基本认识：无论输入输出来自何处（设备），去往何处（设备），一律当做“流”的方式来处理。流就像一个仅容一个 bit 数据通过的管道一样，

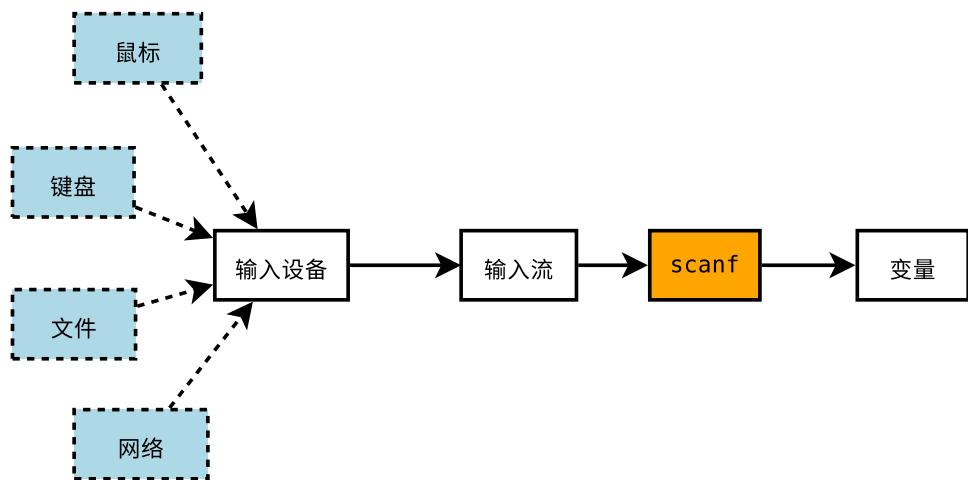


图 7.1: C 的输入模型

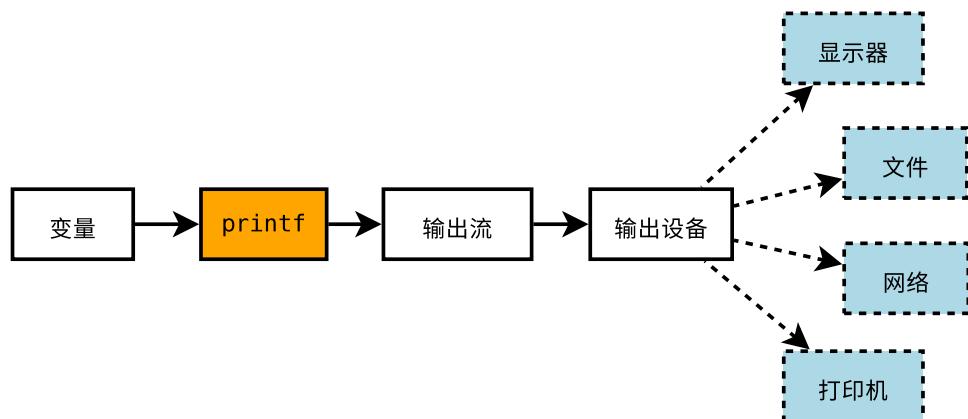


图 7.2: C 的输出模型

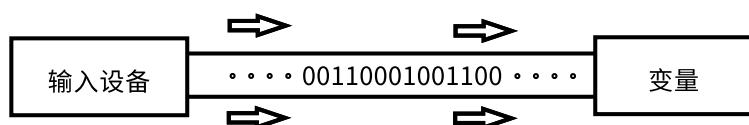


图 7.3: 输入流模型

是一个有序的数据序列。如果我们按照 8 位（1 个字节）来分割解读流中的数据序列，就是“面向字节的流”，有时也简称“字节流”；如果我们以 16 位（2 个字节，即 1 个字符）来分割和解读流中的数据，就是“面向字符的流”，有时也简称“字符流”。其实，字节流和字符流的区分并没有改变流的数据本质，只是我们以不同的视角解读数据罢了，就像一部红楼，经学家看见《易》，道学家看见淫，才子看见缠绵，革命家看见排满，流言家看见宫闱秘事¹。红楼还是那部红楼，不同的人，不同的场合，不同的视角，不同的解读而已。

Java IO 相关的类在包 java.io 中。

7.2.1 面向字节的流

所谓面向字节的流，是指流中的数据按照字节来解释，即每 8 位（1 个字节）为一个解读的单元。Java 提供了如图 7.4 所示的面向字节的流的处理类层次结构。其中的节点流是指直接与输入输出设备打交道处理 I/O 操作的类，处理流是指对原始数据进行二次加工的类。

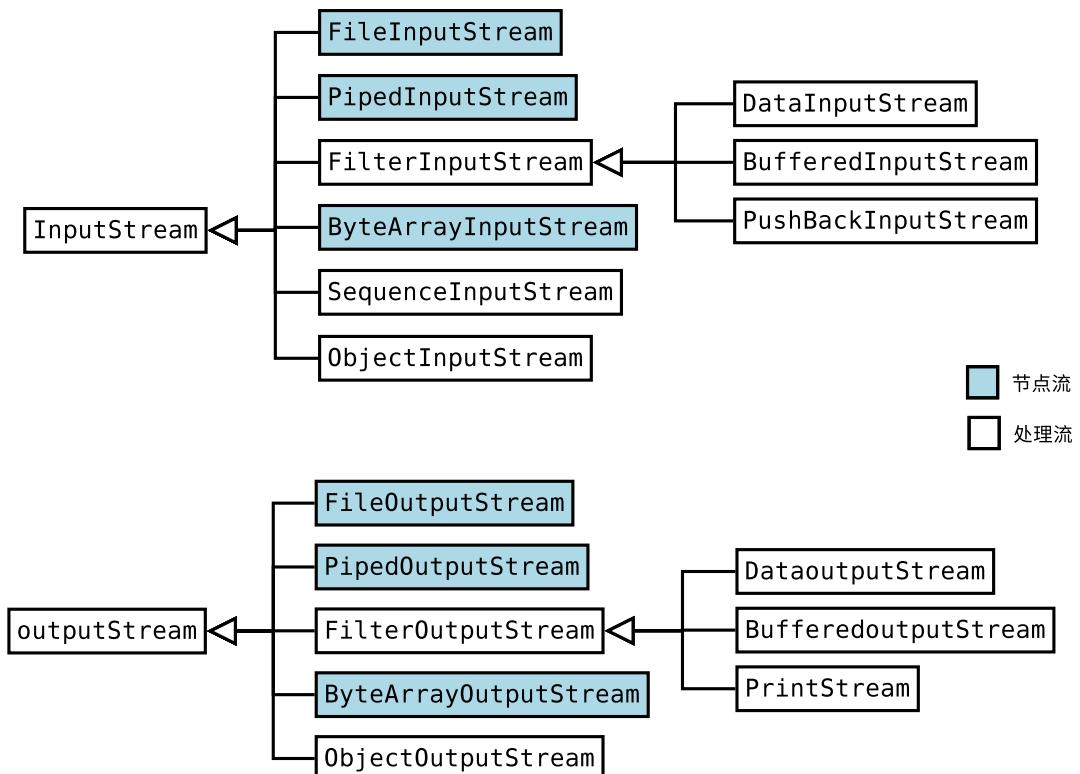


图 7.4: Java 的 `InputStream/OutputStream` 类层次结构

图 7.4 中的 `InputStream/OutputStream` 的子类有各自的应用场景，如表 7.1 所

¹ 出自《鲁迅全集·集外集拾遗补编·<绛洞花主>小引》

示。

InputStream 是个抽象类，是所有字节输入流的父类，其中定义了一些基本的字节输入流的操作方法，如表7.3所示。

Java 内部的数据都是 unicode 编码²的，除 ASCII 码外都是多字节编码方式，因此面向字节的流处理往往用于处理二进制数据，或者用于适合把数据看做二进制的场合。比如：

- 在工业控制领域，我们把接收到的数据按照自己制定的数据格式写入文件（不一定是 Java 语言编写的程序写文件，也许是 C/C++ 写文件），在这种情况下就适合使用面向字节的流打开文件读取数据。在 C 语言中我们也特别强调，为了保证正确读写文件，采用什么方式（主要指面向字节还是面向字符）写入文件，就要采用同样的方式打开文件。
- ASCII 文件可以安全的使用面向字节的流读写，因为 ASCII 字符的长度没有超出 8 位。
- 图片、声音、视频等数据一般是以二进制方式存储的，因此适合使用字节流来处理。

我们之前一直在使用的 System.in，实际上一个 InputStream 类型的对象，System.out 实际上是一个 PrintStream 类型的对象³：

```
1  /**
2   * The "standard" input stream. This stream is already
3   * open and ready to supply input data. Typically this stream
4   * corresponds to keyboard input or another input source specified by
5   * the host environment or user.
6   */
7  public static final InputStream in = null;
8  /**
9   * The "standard" output stream. This stream is already
10  * open and ready to accept output data. Typically this stream
11  * corresponds to display output or another output destination
12  * specified by the host environment or user.
13  * <p>
14  * For simple stand-alone Java applications, a typical way to write
15  * a line of output data is:
16  * <blockquote><pre>
```

² 参见：<http://unicode.org/charts>。Unicode (统一码、万国码、单一码) 是计算机科学领域里的一项业界标准，包括字符集、编码方案等。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。unicode 只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。我们经常说的 UTF-8 编码是 unicode 编码的具体实现（除此之外还有 UTF-16,UTF-32，但是用的不多），UTF-8 最大的一个特点，就是它是一种变长的编码方式。它可以使用 1~4 个字节表示一个符号，根据不同的符号而变化字节长度：对于单字节的符号，字节的第一位设为 0，后面 7 位为这个符号的 unicode 码，因此对于英语字母，UTF-8 编码和 ASCII 码是相同的。对于 n 字节的符号 ($n > 1$)，第一个字节的前 n 位都设为 1，第 $n+1$ 位设为 0，后面字节的前两位一律设为 10。剩下的没有提及的二进制位，全部为这个符号的 unicode 码。

³ 参见 openjdk 的 jdk/src/java.base/share/classes/java/lang/System.java

| 类名 | 描述 |
|----------------------|---|
| FileInputStream | 读取二进制文件，比如图片、音视频等。 |
| PipedInputStream | 提供了管道化操作的具体实现：PipedInputStream 通常和一个 PipedOutputStream 关联在一起，即 PipedOutputStream 的输出送给 PipedInputStream。 |
| ByteArrayInputStream | 将一个字节数组当做一个 InputStream 来处理。 |
| SequenceInputStream | 将多个 InputStream 收尾相接组成一个新的 InputStream。 |
| FilterInputStream | <p>顾名思义，FilterInputStream 接受一个 InputStream 作为参数，然后对这个 InputStream 中的数据做相应的处理后再输出。即，FilterInputStream 通常用来对数据进行筛选、变换、编码等处理，FilterInputStream 的不同子类已经实现了若干的过滤处理，常见的 FilterInputStream 的子类如下：</p> <ul style="list-style-type: none"> • BufferedInputStream：对于任何 InputStream 提供了缓存处理功能，提高了输入处理的效率。 • DataInputStream：如果我们确定 InputStream 中存储的是基本类型数据，则可以借助于 DataInputStream 提供的读取基本类型数据方法简化操作，这些方法直接返回所需要的基本类型变量，比如 readInt。注意到 DataInputStream 也实现了 DataInput 接口，在 DataInput 接口中规范了读取基本类型数据的方法。 • PushBackInputStream：通常，我们从 InputStream 读取一个字节后，这个字节就从 InputStream 移除了，再次从 InputStream 读取数据会从下一个字节开始。PushBackInputStream 的设计目的是把刚刚读取的字节再次送回 InputStream，以便有机会再次读取这个字节的数据。 |
| ObjectInputStream | 用于对基本数据类型和对象的序列化处理，通常和 ObjectOutputStream 联合使用，即 ObjectInputStream 所读取的数据通常是由 ObjectOutputStream 写入的。 |

| 类名 | 描述 |
|-----------------------|---|
| FileOutputStream | 写入二进制文件，比如图片、音视频等。 |
| PipedOutputStream | 通常和一个 PipedInputStream 关联在一起实现管道化操作 |
| ByteArrayOutputStream | 写入到一个字节数组中 |
| FilterOutputStream | 接受一个 OutputStream 作为参数，在数据写入 OutputStream 之前进行一定的处理。常见的 FilterOutputStream 子类如下： <ul style="list-style-type: none">• DataOutputStream：将基本数据类型写入 OutputStream，便于使用 DataInputStream 读入处理。• BufferedOutputStream：对于任何 OutputStream 提供了缓存处理功能，提高了输出处理的效率。• PrintStream：自动刷新缓冲区的 OutputStream。 |
| ObjectOutputStream | 用于对基本数据类型和对象的序列化处理。 |

表 7.2: OutputStream 输出流

| 方法名 | 描述 |
|--|---|
| public abstract int read() throws IOException | 从输入流读取下一个字节，字节值为 0~255。如果输入流不再有数据则返回-1。该方法是一个阻塞方法，直到有数据可读或者数据流结束，或发生异常才返回。 |
| public int read(byte[] b) throws IOException | 从输入流读取一组数据存入缓冲区 b 中，返回所读取字节的个数。如果返回-1 表示数据流结束。该方法相当于 read(b, 0, b.length)。 |
| public int read(byte[] b, int off, int len) throws IOException | 从输入流读取最多 len 字节数据存入缓冲区 b 中，存储位置从 b 的第 off 个位置开始。该方法返回读取的字节数，如果返回-1 表示数据流结束。 |
| public int available() throws IOException | 返回当前输入流可供读取的字节数。 |
| public void mark(int readLimit) | 在输入流中标记当前位置，以后可以调用 reset 方法返回该位置，以便重复读取从该标记位置开始的数据。readLimit 设置调用 mark 方法后可以读取的最大字节数，且保持 mark 标记有效。 |
| public void reset() throws IOException | 重置流的读取位置，回到上次调用 mark 方法标记的位置。 |
| public boolean markSupported() | 检测输入流是否支持 mark 和 reset 方法 |
| public long skip(long n) throws IOException | 从输入流忽略 n 字节的数据，返回被忽略的实际字节数。 |
| public void close() throws IOException | 关闭输入流，释放所占用的系统资源。 |

表 7.3: InputStream 的常用方法

| 方法名 | 描述 |
|--|---|
| public abstract void write(int b) throws IOException | 向输出流写入一个字节。写出字节为整数 b 的低字节，整数 b 的 3 个高字节被忽略。 |
| public void write(byte[] b) throws IOException | 把缓冲区 b 中的全部数据写入输出流 |
| public void write(byte[] b, int off, int len) throws IOException | 把缓冲区 b 从 b[off] 开始的 len 个字节的数据写入输出流 |
| public void flush() throws IOException | 刷新输出流，强制输出缓冲区的数据立即写出 |
| public void close() throws IOException | 关闭输出流 |

表 7.4: OutputStream 的常用方法

```

17   *      System.out.println(data)
18   *  </pre></blockquote>
19   * <p>
20   * See the <code>println</code> methods in class <code>PrintStream</code>.
21   *
22   * @see  java.io.PrintStream#println()
23   * @see  java.io.PrintStream#println(boolean)
24   * @see  java.io.PrintStream#println(char)
25   * @see  java.io.PrintStream#println(char[])
26   * @see  java.io.PrintStream#println(double)
27   * @see  java.io.PrintStream#println(float)
28   * @see  java.io.PrintStream#println(int)
29   * @see  java.io.PrintStream#println(long)
30   * @see  java.io.PrintStream#println(java.lang.Object)
31   * @see  java.io.PrintStream#println(java.lang.String)
32   */
33 public static final PrintStream out = null;

```

输入输出流在使用后为什么要及时关闭呢？这是因为，Java 把所有的输入输出流都抽象为文件的操作，如果我们不及时关闭打开的输入输出流，相当于不及时关闭打开的文件，久而久之可能造成操作系统打开的文件过多，从而拖慢系统运行速度，甚至超出系统允许打开的文件数。因此，输入输出流使用完毕及时关闭是个好习惯。

我们在 6.8 [在第 167 页] 会看到，使用 try-with-resources 可以方便的管理输入输出流的关闭，减轻了程序员的负担。

例 7.1. 复制二进制文件。

代码设计 参见代码清单7.1。

代码清单 7.1: CopyBinary.java

```
1 package cn.edu.sdutoftlab.io;
2
3 import java.io.*;
4
5 /**
6  * 演示复制二进制文件的方法.
7  *
8  * @author Su Baochen
9 */
10 public class CopyBinary {
11     public static void main(String[] args) {
12         copyWithBuffer("test.dat", "another.dat");
13         copyWithoutBuffer("test.dat", "another.dat");
14         copyWithBuffer("test.jpg", "another.jpg");
15         copyWithoutBuffer("test.jpg", "another.jpg");
16     }
17
18     /**
19      * 复制文件, 带缓冲区.
20      * @param orig 源文件路径
21      * @param dest 目标文件路径
22      */
23     public static void copyWithBuffer(String orig, String dest) {
24         try {
25             BufferedInputStream bis = new BufferedInputStream(new FileInputStream(orig));
26             BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(dest
27                     ))
28         ) {
29             int b;
30             while ((b = bis.read()) != -1) {
31                 System.out.print(b);
32                 bos.write(b);
33             }
34             System.out.println();
35         } catch (IOException e) {
36             e.printStackTrace();
37         }
38     }
39
40     /**
41      * 复制文件, 不带缓冲区.
42      * @param orig 源文件路径
43      * @param dest 目标文件路径
44      */
45     public static void copyWithoutBuffer(String orig, String dest) {
```

```
45     try {
46         FileInputStream fis = new FileInputStream(orig);
47         FileOutputStream fos = new FileOutputStream(dest)
48     } {
49         int b;
50         while ((b = fis.read()) != -1) {
51             System.out.print(b);
52             fos.write(b);
53         }
54         System.out.println();
55     } catch (IOException e) {
56         e.printStackTrace();
57     }
58 }
59 }
```

运行结果 在 Idea 中运行 CopyBinary.main() 结果如下：

```
97321151161141051101031049484810504846525310
97321151161141051101031049484810504846525310
```

代码分析和说明 本例我们实现了两种文件复制的方式：

1. 不使用缓冲区的字节流。通过 FileInputStream 读取文件，通过 FileOutputStream 写入文件，没有使用缓冲区，每次读取一个字节。显然当文件比较大时，读写文件的效率是比较低的。
2. 使用缓冲区的字节流。利用 BufferedInputStream 和 BufferedOutputStream 构造带缓冲区的字节流，这是编程实践中最常见的情形。

注意到我们使用了 try-with-resources 的 Java 新语法。如果使用传统的 try-catch 接口则要注意输入字节流和输出字节流在使用完毕后都需要关闭，通常借助于 finally 代码块实现。

例 7.2. FileInputStream/FileOutputStream、BufferedInputStream/BufferedOutputStream、DataInputStream/DataOutputStream 的使用

设计要求 假设一个表示气温的文件 weather.txt 有下列数据⁴，试将这些数据使用 DataOutputStream 重新写入文件 weather.dat，然后使用 DataInputStream 读出 weather.dat 并求温度的平均值 ()。

⁴我们在 section §7.5还会使用文件相关 API 重新设计本例。

17.1 24.2

18.9 22.3

17.3 -2.3 15.6

代码设计 参见代码清单7.2。

代码清单 7.2: Temperature.java

```
1 package cn.edu.sdutoftlab.io;
2
3 import java.io.*;
4
5 /**
6  * Created by subaochen on 16-12-6.
7 */
8 public class Temperature {
9     public static final float[] temperatures = {17.1f, 24.2f, 18.9f, 22.3f, 17.3f, -2.3
10    f, 15.6f};
11
12    public static void main(String[] args) {
13        try (DataOutputStream dos =
14            new DataOutputStream(new BufferedOutputStream(new FileOutputStream("weather.dat")));
15        DataInputStream dis =
16            new DataInputStream(new BufferedInputStream(new FileInputStream("weather.
17            dat")))) {
18            for(int i = 0; i < temperatures.length; i++) {
19                dos.writeFloat(temperatures[i]);
20            }
21            dos.flush(); // 如果不刷新的话，下面会读取不到数据
22
23            double total = 0.0f;
24            int count = 0;
25            try {
26                while (true) {
27                    float current = dis.readFloat();
28                    System.out.println(current);
29                    total += current;
30                    count++;
31                }
32            } catch (EOFException e) {
33                // 这里捕获异常只是为了结束读取文件，因此不需要额外的处理
34            }
35            System.out.println("average temperature = " + total / count);
36        } catch (IOException e) {
37            e.printStackTrace();
38        }
39    }
```

运行结果 在 Idea 中运行 Temperature 结果如下：

```
17.1  
24.2  
18.9  
22.3  
17.3  
-2.3  
15.6  
average temperature = 16.157142809459142
```

代码分析和说明 注意到 dos 的构造方式: DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(new FileOutputStream("weather.dat"))), 可以通过图7.5加深理解。

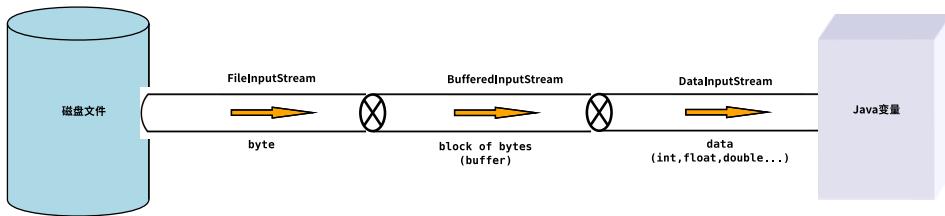


图 7.5: DataStream/BufferedStream/InputStream 的联合使用

注意 weather.txt, weather.dat 文件的位置：目前是在项目的根目录下的，这是因为几乎所有的 IDE 环境，包括 Eclipse、NetBeans、Idea 都把项目的根目录作为字节流的根目录来处理，因此我们在上面的例子中，都是采用了相对路径的方式来读写文件。但是，这种读写文件的方式（主要是对文件路径的定义方式）如果离开了 IDE 环境就失效了，因此在实践中一般不采用此种文件定位方式，一般根据 classpath 定位文件，参见访问属性文件。

无论是读取还是写入文件，字节流的操作分为以下两种方式：

1. 按字节处理：每次读取或者写入一个字节；
2. 按字节数组处理：每次读取或者写入一个字节数组，数组的大小需要事先定义；



在字符流中，我们还会看到按行处理的情形，但是在处理字节流时一般不按行处理，其原因是二进制数据一般不进行换行处理。

练习 7.1. 使用 C 语言通过面向字节的方式打开一个文件写入 “Hello World!”，然后使用 Java 语言读取此文件，看看有什么变化？

练习 7.2. 使用 C 语言通过面向字节的方式打开一个文件写入 “你好，世界！”，然后使用 Java 的面向字节的流读取此文件，看看有什么变化？

7.2.2 面向字符的流

理解了字节流，字符流就不难理解了。字节流是将“流”中的数据按照字节来划分，所谓字符流，只是将“流”中的数据按照 2 个字节（即一个字符）来划分而已。Java IO 提供了如图7.6所示的处理字符流的类层次结构。

和字节流非常类似，Reader 和 Write 是两个抽象类，其中封装了操作字符流的基本方法，如表7.5和表7.6所示。

7.3 从键盘输入数据⁵⁶

对于输入输出而言，使用最多的就是从键盘输入数据，以及在显示器上输出数据。在 Java 语言中，数据的输出很方便，使用 System.out.println

就已经很好用了，无论什么类型的数据，都能够自动转换为字符串输出（在 Java8 以后，甚至包括了 List、Map 类型的数据），这里不再赘述。但是 Java 从键盘输入数据确实不是太方便，比如考虑下面的情形：从键盘输入一个实数 3.125 保存到变量 x 中，该如何完成这个任务呢？如图图实数 3.125 的内存表达所示。我们的目标是获得一个单精度浮点类型变量 x，其值为 3.125，在内存中占用 4 个字节的内存空间，从高到低的 4 个字节分别为 40、48、00 和 00（均为 16 进制数）。

float x = 3.125;

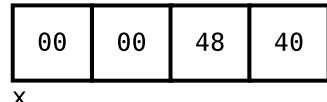


图 7.7：实数 3.125 的内存表达

从键盘输入数据，我们已经知道必须通过 System.in 来完成，下面我们再次回顾一下 System 类的基本内容：

```

1 public final class System {
2     ...
3     public final static InputStream in = null;
4     ...
5     private static void initializeSystemClass() {

```

⁵借鉴了《Java 程序设计》（谌卫军）的案例，感谢作者谌卫军的精彩阐述！

⁶本节的完整测试代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/io/src/cn/edu/sdut/softlab/SystemInTest.java>

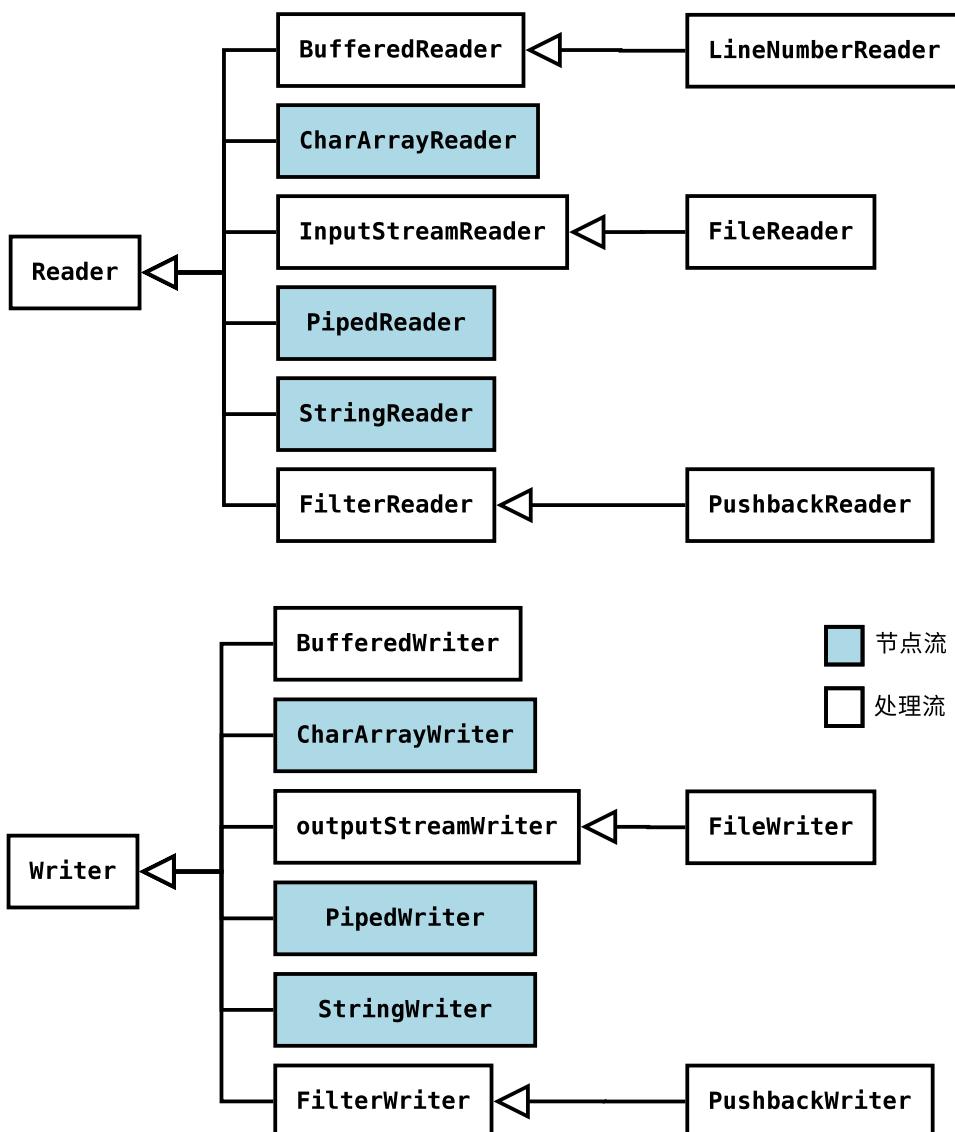


图 7.6: 字符流的类层次结构

| 方法名 | 描述 |
|--|--|
| public int read() throws IOException | 从流读取一个字符并返回，如果没有字符可读则返回-1 |
| public int read(char[] cbuf) throws IOException | 从流读取字符到数组 cbuf 中，返回读取的字符个数 |
| public abstract int read(char[] cbuf, int off, int len) throws IOException | 从流读取字符到数组 cbuf 中，并从 cbuf[off] 开始存储，最多读取 len 个字符。 |
| public long skip(long n) throws IOException | 跳过流中的 n 个字符 |
| public boolean ready() throws IOException | 检测输入字符流是否可读 |
| public void mark(int readAheadLimit) throws IOException | 标记流的当前位置，readAheadLimit 表示在此位置有效期间最多可以读取的字符数 |
| public void reset() throws IOException | 复位标记过的流 |
| public abstract void close() throws IOException | 关闭流 |

表 7.5: Reader 的基本方法

| 方法名 | 描述 |
|--|--|
| public void write(int c) throws IOException | 写一个字符到流 |
| public void write(char[] cbuf) throws IOException | 写字符数组 cbuf 到流 |
| public abstract void write(char[] cbuf, int off, int len) throws IOException | 写字符数组 cbuf 到流，从 cbuf[off] 开始最多写入 len 个字符 |
| public void write(String str) throws IOException | 写字符串 str 到流 |
| public void write(String str, int off, int len) throws IOException | 写字符串 str 到流，从 off 个字符开始，最多写入 len 个字符 |
| public abstract void flush() throws IOException | 刷新流缓冲区 |
| public abstract void close() throws IOException | 关闭流 |

表 7.6: Writer 的基本方法

```

6     ...
7     FileInputStream fdIn = new FileInputStream(FileDescriptor.in);
8     setIn0(new BufferedInputStream(fdIn));
9     ...
10    }
11 }

```

也就是说，`in` 是 `System` 类的静态成员变量，在系统初始化的时候，`in` 初始化为一个带缓冲的文件字节流，即 `in` 是一个从标准输入设备 (`FileDescriptor.in`) 接受二进制数据并实现了缓存处理的字节流。另外也需要注意到，`in` 是 `InputStream` 类型的，但其实在初始化的时候我们看到了，`in` 的真实类型是 `BufferedInputStream` 类型的，这是前面讲过的多态¹¹的概念：子类对象，父类引用。

如果我们直接使用 `System.in` 从键盘读入 3.125，比如保存到一个 `byte` 数组中：

```

1 byte[] b = new byte[20];
2 System.in.read(b);

```

则内存中的数据如图7.8所示。

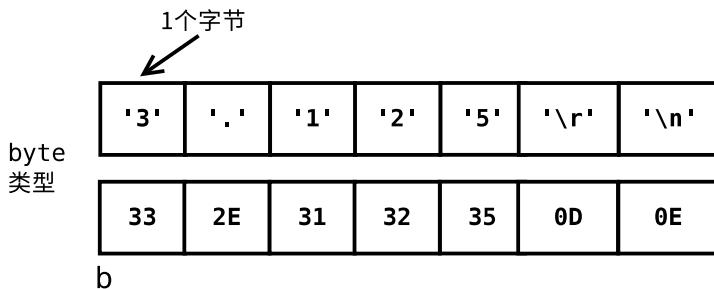


图 7.8: InputStream 的 read 方法

可以看出，当从键盘输入 3.125 时，保存在数组 `b` 中的是它们的 ASCII 值（注意图7.8中使用 16 进制表示内存中的数据），并且包括了回车符和换行符¹²，显然数组 `b` 不符合要求，我们很难直接将数组 `b` 直接转换为一个 `float` 类型的数据。¹³

既然直接使用字节流不容易达成我们的目标，使用字符流 `InputStreamReader` 可以吗？`InputStreamReader` 的构造方法是：

```

1 public InputStreamReader(InputStream in);
2 public InputStreamReader(InputStream in, String enc) throws
UnsupportedEncodingException;

```

¹¹参见 4.3 [在第 116 页]

¹²如果在 Linux 下面进行测试的话，从键盘输入是不包含回车符'\r'的，只有换行符'\n'，即在 Linux 下面通过换行符'\n'表示输入结束。

¹³并非不能，而是比较麻烦，比如可以这样做：

```

byte[] b = new byte[20];
float f = Float.valueOf(new String(b));

```

`InputStreamReader` 的构造方法的参数是 `InputStream` 类型的，也就是说，`InputStreamReader` 的功能是把字节流转换为字符流，于是我们可以尝试这样解决：

```
1 char[] c = new char[20];
2 InputStreamReader sr = new InputStreamReader(System.in);
3 sr.read(c);
```

这段代码的功能是从键盘输入一组数据并保存到字符数组 `c` 中，`c` 在内存中的内容如图7.9所示。

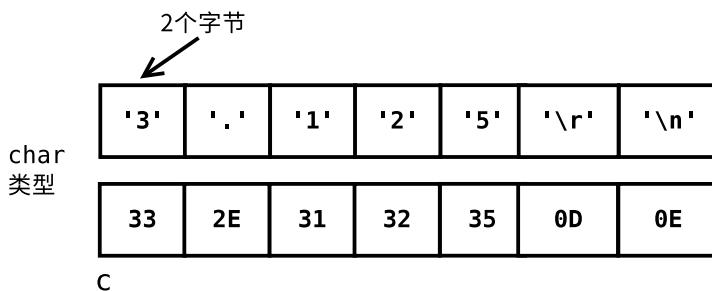


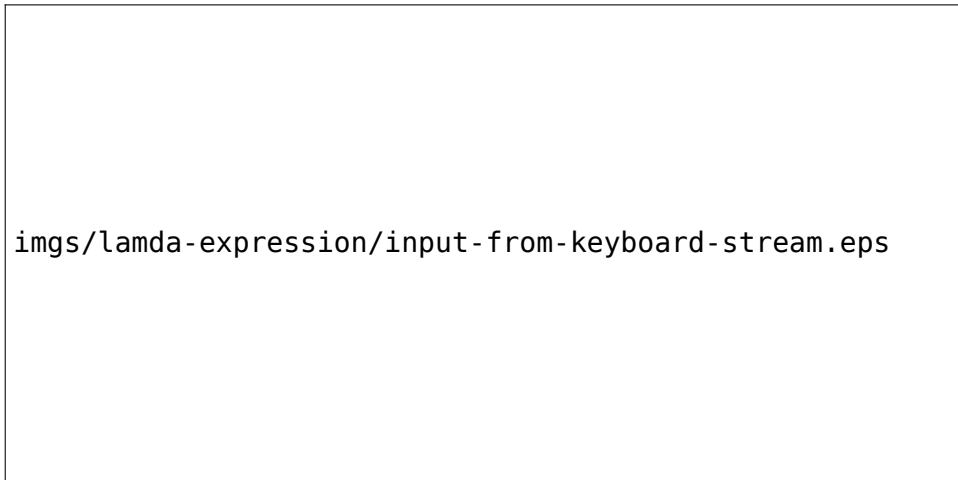
图 7.9: `InputStreamReader` 类的 `read` 方法

可以看出，数组 `c` 存放的数据与图7.8中的数组 `b` 是完全一样的，唯一的区别是数据类型发生了变化，`b` 是一个字节类型的数组，每个数组元素只占一个字节；`c` 是一个字符类型的数组，每个数组元素占两个字节。因此，使用 `InputStreamReader` 和使用 `InputStream` 输入浮点数会遇到相似的问题：都需要将输入的数据（字节数组或者字符数组）转换为字符串对象然后使用 `Float.valueOf(String s)` 转换为浮点数。有没有办法从键盘直接获取字符串呢？`InputStream` 和 `InputStreamReader` 都没有提供这样的功能，即便 `BufferedInputStream` 也没有提供直接从键盘获取字符串的功能，这是容易理解的：`InputStream`、`BufferedInputStream` 的目的是原始的二进制字节，`InputStreamReader` 的目的是为了获取原始的二进制字符，这些都和字符串没有关系。`BufferedReader` 提供了从键盘获取字符串的功能，其中的 `readLine` 方法可以从键盘获取一行字符，并自动删除了末尾的回车换行符，于是我们有了最终的解决方案：

```
1 BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
2 String str = reader.readLine();
3 float x = Float.parseInt(str);
```

在这段代码中，以 `System.in` 对象为参数，创建了一个 `InputStreamReader` 对象，然后以该对象为参数，创建了一个 `BufferedReader` 对象，从而形成了这几个类之间的连接关系¹⁴，如图7.10所示。

¹⁴ 参见 ?? [在第 ??页]



imgs/lambda-expression/input-from-keyboard-stream.eps

图 7.10: 从键盘输入数据相关类的关系

这里的基本思路是: `InputStream` 类负责从键盘读入字节流, 然后 `InputStreamReader` 类将字节流转换为字符流, 接着 `BufferedReader` 进行缓冲并读取一行字符, 即把末尾的回车换行符去掉并将数据转化为字符串, 最后调用 `Float` 类的 `parseFloat` 方法把这个字符串转化为相应的实数。

本节内容中, 我们从观察 java 数据的内存表达深刻理解 java 的输入输出, 常见的 IDE 都可以帮助我们方便的观察 Java 数据在内存中的存储格式, 比如在 Idea 中调试 `SystemInTest` 时可以看到 Idea 即时的给出了各个变量的当前值:

```

10 > public class SystemInTest {
11 >     public static void main(String[] args) { args: {} }
12     try {
13         // 直接使用InputStream
14         byte[] b = new byte[20]; b: {51, 46, 49, 50, 53, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
15         int num = System.in.read(b); num: 6 b: {51, 46, 49, 50, 53, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
16         System.out.println("bytes num read:" + num); num: 6
17         float f = Float.valueOf(new String(b));
18         System.out.println(f);

```

这里设置一个断点
输入3.125后的情形。
很清楚的看出当前数组b的内容。注意这里显示的是10进制数据, 比如10进制的51恰好是16进制的33



如果我们仅仅需要用户从键盘输入一个简单的响应，比如 yes 或 y 的话，可以利用 System.console 方法，代码示例如下：

```
1 static boolean okayToOverwrite(String file) {  
2     String answer = System.console().readLine("overwrite %s (yes/no)? ", file);  
3     return (answer.equalsIgnoreCase("y")) || answer.equalsIgnoreCase("yes"));  
4 }
```

7.4 Scanner 类

从从键盘输入数据¹⁷¹⁸一节的描述可以看出，Java 在处理键盘输入数据时实在不够友好，于是从 Java 1.5 开始增加了一个工具类 Scanner 简化从键盘输入数据的处理。Scanner 类也是从 InputStream 接受数据，但是可以根据模式匹配的方法直接将二进制数据转换为相应的基本数据类型。具体的说，Scanner 会自动根据分隔符（默认为空白字符，包括空格符、回车符、换行符、制表符）从输入数据中分离出一个个字符串（称为 token），并转换为要求的整数、实数或者字符串等，从而方便的实现了在同一行读入多个不同类型的数据。

Scanner 类的常用方法如表7.7所示。

例 7.3. 从键盘输入两个整数，计算其乘积并输出。

代码设计 参见代码清单7.3。

代码清单 7.3: Multiply.java

```
1 package cn.edu.sduto.softlab.io;  
2  
3 import java.util.Scanner;  
4  
5 /**  
6  * @author subaochen.  
7 */  
8 public class MultiplyTest {  
9     public static void main(String[] args) {  
10         Scanner console = new Scanner(System.in);  
11         System.out.println("请输入两个整数: ");  
12         int num1 = console.nextInt();
```

¹⁷借鉴了《Java 程序设计》（谌卫军）的案例，感谢作者谌卫军的精彩阐述！

¹⁸本节的完整测试代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/io/src/cn/edu/sduto/softlab/SystemInTest.java>

| 方法名 | 描述 |
|--|------------------------------|
| public byte nextByte() | 读入下一个字节 (byte) |
| public short nextShort() | 读入下一个短整数 (short) |
| public int nextInt() | 读入下一个整数 (int) |
| public long nextLong() | 读入下一个长整数 (long) |
| public float nextFloat() | 读入下一个单精度浮点数 (float) |
| public double nextDouble() | 读入下一个双精度浮点数 (double) |
| public boolean hasNext() | 是否存在可读的数据? |
| public String next() | 读入下一个 token (即空白字符隔开的独立的字符串) |
| public String nextLine() | 读入下一行 |
| public Scanner useDelimiter(Pattern pattern) | 使用自定义的 token 分隔符 |

表 7.7: Scanner 类的常用方法

```

13     int num2 = console.nextInt();
14     int product = num1 * num2;
15     System.out.println("这两个整数的乘积为: " + product);
16 }
17 }
```

练习 7.3. 使用 BufferedReader 改写例7.3。

练习 7.4. 使用 Scanner 读取文件 “双城记.txt”，统计该小说有多少个单词？

提示：“双城记.txt”文件下载地址：。

7.5 文件操作¹⁹

7.5.1 什么是 Path ?

在新的文件操作 API 中，Path 的概念至关重要，类 Path 是文件操作的入口。

无论是 Windows 操作系统还是 Linux 操作系统，文件系统都是树状结构的。典型的文件系统如图7.11所示。

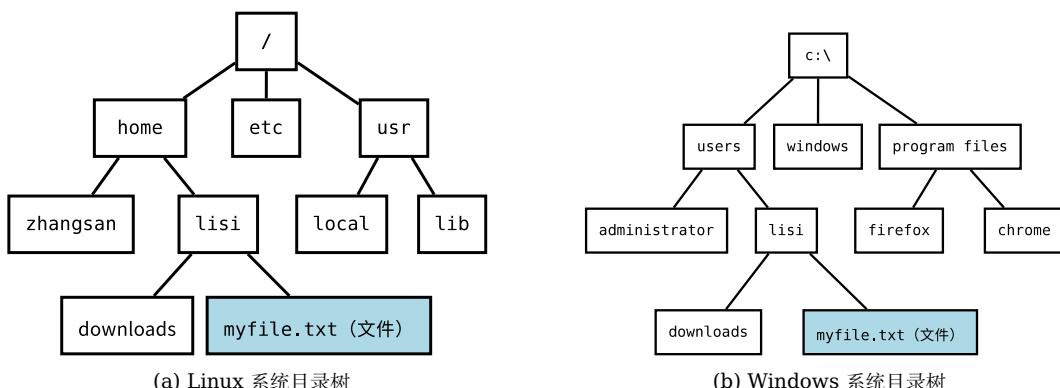


图 7.11: 目录树示意图

myfile.txt 的路径 (Path) 在 Linux 系统表示为: /home/lisi/myfile.txt; 而在 Windows 系统下 myfile.txt 的路径 (Path) 表示为: c:\users\list\myfile.txt。这里要特别注意到目录分隔符的区别: Windows 操作系统使用反斜杠 (\) ²⁰作为目录

¹⁹JDK 1.7 引入了新的文件操作 API，即 NIO.2，本节内容着重于 NIO.2，不再涉及旧的文件 API。

²⁰众所周知，Unix 的历史要比 Windows 久远，Windows 的设计从 Unix 中汲取了很多营养，但是 Windows 的路径分隔符和 Unix 系统的路径分隔符截然相反，给后来的程序设计带来了一些困扰：你必须正确识别和处理不同操作系统的路径分隔符。你一定会想，要是当初微软在设计 Windows 的时候使用和 Unix 相同的路径分隔符，该多好！历史没有如果，现实如此残酷！欲知当初微软选择反斜杠作为文件分隔符的原因，请参考：<https://blogs.msdn.microsoft.com/larryosterman/2005/06/24/why-is-the-dos-path-character/>。简单的说，Windows 操作系统脱胎于 Dos 操作系统，在 Dos 操作系统中反斜杠 (/) 已经作为命令行参数的分隔符了，因此 Windows 只好选用其他的分隔符（反斜杠）作为文件路径分隔符。

分隔符，而其他所有操作系统（包括 Linux、Unix、MacOS）都使用斜杠（/）作为目录分隔符。

注意到，路径（Path）不仅仅是目录的意思。实际上，路径（Path）包含了以下几种情况：

- 纯粹文件名，比如：myfile.txt
- 纯粹目录，比如：/home/sili
- 绝对路径 + 文件名，比如：/home/lisi/myfile.txt
- 相对路径 + 文件名，比如：./lisi/myfile.txt

以上几种情况都是路径（Path）。

7.5.2 相对路径和绝对路径

从形式上看，相对路径和绝对路径很容易区分：以目录分隔符（Linux 系统使用 /，windows 系统使用 \）为起点的路径是绝对路径，其他形式的路径是相对路径。下面是一些示例，都表示文件 myfile.txt：

```
# 假设当前位于/
/home/lisi/myfile.txt # 绝对路径
home/lisi/myfile.txt # 相对路径
./home/lisi/myfile.txt # 相对路径，. 表示当前目录
# 假设当前位于/home
/home/lisi/myfile.txt # 绝对路径
lisi/myfile.txt    # 相对路径
./lisi/myfile.txt # 相对路径
# 假设当前位于/usr
/home/lisi/myfile.txt # 绝对路径
../home/lisi/myfile.txt # 相对路径，.. 表示上一级目录
/home/lisi/../../zhangsan/myfile.txt # 绝对路径
```

7.5.3 Path 类

Path 类是 Java 的新文件 API 的重点和核心，顾名思义，Path 类代表了一个路径，一个 Path 对象包括了文件名和文件所在的目录，因此 Path 类中包含了处理文件和目录的相关方法。

代码清单 7.4: CreatePathTest.java

```
1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.net.URI;
4 import java.nio.file.Path;
5 import java.nio.file.Paths;
6
7 /**
8  * @author subaochen.
9 */
10 public class CreatePathTest {
11     public static void main(String[] args) {
12         Path p1 = Paths.get("/home/subaochen/");
13         Path p2 = Paths.get(args[0]);
14         Path p3 = Paths.get(URI.create("file:///home/subaochen/HelloWorld.java"));
15         Path p4 = Paths.get(System.getProperty("user.home"), "logs", "foo.log");
16     }
17 }
```

7.5.3.1 创建 Path 对象

利用 Paths 类（注意和 Path 类的区别）的 get 方法可以很方便的创建一个 Path 对象，参见代码清单7.4。

在 Idea 中利用调试功能可以方便的观察所创建的 Path 对象，如图7.12所示。

7.5.3.2 获取 Path 信息

Path 对象的主要信息是目录分隔符隔开的一个字符串数组，可以通过 getName 方法返回这个数组的每个元素²²，参见代码清单7.5。

代码清单7.5中涉及的主要方法的用法参见表7.8：

7.5.3.3 Path 转换

Path 转换主要通过表7.9中的 3 个方法实现的。

代码清单7.6演示了 Path 转换的三种情况。

7.5.3.4 拼接 Path

Path 类的 resolve 方法可以拼接路径，如代码清单7.7所示。

²²结合 lambda 表达式操作这个数组更方便，比如：

```
Path path = Paths.get("/home/subaochen/test.txt");
path.forEach(p -> System.out.println(p));
```

代码清单 7.5: PathInfoTest.java

```
1 package cn.edu.sdut.softlab.io;
2
3 import java.net.URI;
4 import java.nio.file.Path;
5 import java.nio.file.Paths;
6
7 /**
8 * @author subaochen.
9 */
10 public class PathInfoTest {
11     public static void main(String[] args) {
12         Path p1 = Paths.get("/home/subaochen/");
13         //Path p2 = Paths.get(args[0]); // 需要传入命令行参数才能正确执行
14         Path p3 = Paths.get(URI.create("file:///home/subaochen/HelloWorld.java"));
15         Path p4 = Paths.get(System.getProperty("user.home"), "logs", "foo.log");
16
17         pathInfo(p1);
18         //pathInfo(p2);
19         pathInfo(p3);
20         pathInfo(p4);
21     }
22
23     // 打印path的信息
24     // 注意, path对象表达的目录和文件不一定存在
25     private static void pathInfo(Path path) {
26         System.out.println("====path info for:" + path + "====");
27         System.out.format("toString: %s%n", path.toString());
28         System.out.format("getFileName: %s%n", path.getFileName());
29         System.out.format("getName(0): %s%n", path.getName(0));
30         System.out.format("getNameCount: %d%n", path.getNameCount());
31         System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
32         System.out.format("getParent: %s%n", path.getParent());
33         System.out.format("getParent: %s%n", path.subpath(0,path.getNameCount() - 1));
34         System.out.format("getRoot: %s%n", path.getRoot());
35         // 使用lambda表达式
36         //path.forEach(p->System.out.println(p));
37     }
38 }
```

| 方法名 | 描述 |
|---|---|
| String toString() | 路径的字符串表达，自动根据不同操作系统使用了不同的目录分隔符 |
| Path getFileName() | 路径中的最“远”对象，即 Path 路径数组的最后一个元素，可能是真实的文件名，也可能是一个子目录名。注意， getFileName 方法的返回值是 Path，因此打印 getFileName 实际上会调用返回的 Path 对象的 toString 方法。 |
| Path getName(int index) | 返回 Path 数组的第一个元素 index。路径中的第一个元素 index 为 0，最后一个元素的 index 为 count-1 |
| int getNameCount() | 返回 Path 中的元素个数 |
| Path subpath(int beginIndex, int endIndex) | 非常类似于 String 类的 subString 的用法，返回 Path 中的一部分信息。beginIndex 和 endIndex 分别指定起始 index 和终止 Index。注意到，endIndex 处的元素不包含在返回的 Path 中。 |
| Path getParent() | 当前 Path 对象的上一级 Path，大部分情况下， getParent 相当于 subpath(0,getNameCount() -1) ，即将 Path 数组最后一个元素去掉即为上一级 Path。 |
| Path getRoot() | 返回根路径。如果是相对路径，则返回 null。 |

表 7.8: Path 信息相关方法

| 方法名 | 描述 |
|---|--|
| URI toUri() | 使用 Uri 方式描述路径。对于本地文件（文件系统）使用 file 协议，因此 URI 的形式如 file:///path-to-file |
| Path toAbsolutePath() | 返回绝对路径，这对于了解文件在文件系统的位置很有帮助 |
| Path toReal-Path(LinkOption... options) throws IOException | 返回真实的文件路径。该方法会检测文件是否存在，也是唯一一个检测文件是否存在的 Path 方法。参数 options 决定了如何处理符号链接。 |

表 7.9: Path 转换方法

代码清单 7.6: PathConversionTest.java

```
1 package cn.edu.sdutoftlab.io;
2
3 import java.io.IOException;
4 import java.net.URI;
5 import java.nio.file.NoSuchFileException;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8
9 /**
10  * @author subaochen.
11 */
12 public class PathConversionTest {
13     public static void main(String[] args) {
14         Path p1 = Paths.get("/home/subaochen/");
15         Path p2 = Paths.get("./test/myfile.txt");
16         Path p4 = Paths.get(System.getProperty("user.home"), "logs", "foo.log");
17
18         // file:///home/subaochen
19         System.out.println("Uri of /home/subaochen/ is: " + p1.toUri());
20         // 转换为绝对路径
21         System.out.println("absolute path of ./test myfile.txt:" + p2.toAbsolutePath())
22             ;
23
24     try {
25         System.out.println("realpath of p1:" + p1.toRealPath());
26         // 抛出NoSuchFileException
27         System.out.println(" realpath of p4:" + p4.toRealPath());
28     } catch (NoSuchFileException e) {
29         System.err.println("file " + e.getFile() + " not exists!");
30     } catch (IOException e) {
31         e.printStackTrace();
32     }
33 }
```

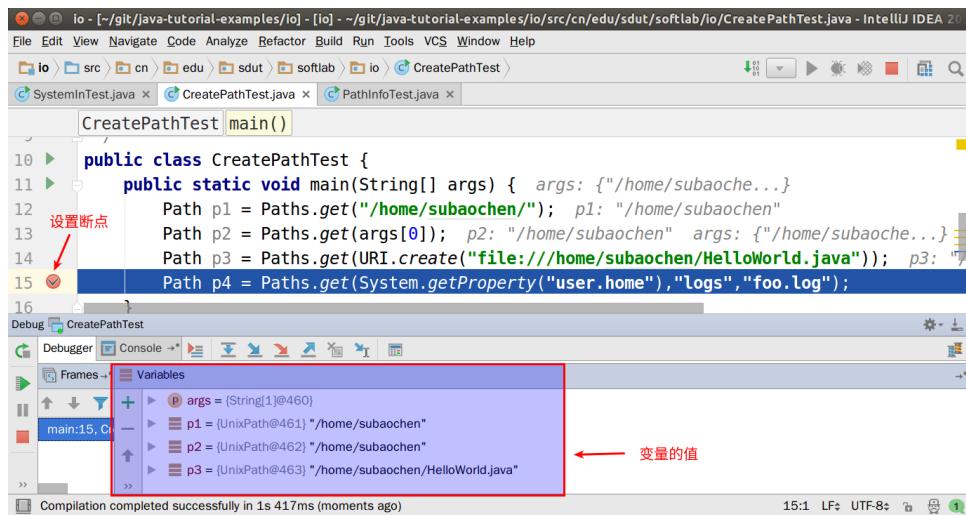


图 7.12: 在 Ideal 中通过调试观察 Path 对象

代码清单 7.7: PathResolveTest.java

```
1 package cn.edu.sdut.softlab.io;
2
3 import java.nio.file.Path;
4 import java.nio.file.Paths;
5
6 /**
7 * @author subaochen.
8 */
9 public class PathResolveTest {
10     public static void main(String[] args) {
11         Path path = Paths.get("/home/subaochen/test");
12         // /home/subaochen/test/myfile.txt
13         System.out.println("myfile.txt resolved:" + path.resolve("myfile.txt"));
14         // /home/subaochen/test/list/myfile.txt
15         System.out.println("myfile.txt resolved:" + path.resolve("list/myfile.txt"));
16         // /myfile.txt
17         System.out.println("myfile.txt resolved:" + path.resolve("/myfile.txt"));
18     }
19 }
```

需要注意的是，如果 `resolve` 的参数是一个绝对路径，则拼接的结果只是返回参数中的绝对路径，因此在 `resolve` 中应该避免传入一个绝对路径。

拼接 Path 的另外一个方法是 `relativize`，请参考 JDK 的相关文档。

7.5.3.5 比较 Path

比较 Path 主要是通过 3 个方法，参见表。

| 方法名 | 描述 |
|--|---|
| <code>boolean equals(Object otherPath)</code> | 重写了 <code>Object.equals</code> 方法，比较两个 Path 对象是否相同：两个 Path 对象代表的路径相同则两个 Path 对象相同 |
| <code>boolean endsWith(Path other)</code> <code>boolean endsWith(String other)</code> | 判断 Path 对象是否以给定的 Path 对象或者路径字符串结尾 |
| <code>boolean startsWith(Path other)</code> <code>boolean startsWith(String other)</code> | 判断 Path 对象是否以给定的 Path 对象或者路径字符串开头 |

表 7.10: 比较两个 Path 的方法

比较 Path 的示例参见代码清单 7.8。

7.6 Files 类

Files²³类是 Java 文件操作新 API (`java.nio.file` 包) 中文件操作的核心类。相对于 Path 类，Files 类聚焦于文件相关的操作。

²³你可能会迷惑，为什么不命名为 File 类呢？很遗憾的是，File 类是旧的（JDK 1.7 之前）的 Java 文件操作 API 中的文件操作类，新的文件操作类只好叫做 Files 了。通常，以单数命名的类用来表示一类事物，复数命名的类是工具辅助类类，比如 Path 表征路径，Paths 是路径的工具辅助类。因此 Files 并不是一个很好的类的命名，这也是 Java API 无奈的选择。

代码清单 7.8: PathCompareTest.java

```
1 package cn.edu.sduot.softlab.io;
2
3 import java.nio.file.Path;
4 import java.nio.file.Paths;
5
6 /**
7  * @author subaochen.
8 */
9 public class PathCompareTest {
10     public static void main(String[] args) {
11         Path p1 = Paths.get("/home/subaochen/myfile.txt");
12         Path p2 = Paths.get(System.getProperty("user.home") + "/myfile.txt");
13         System.out.println("p1 == p2 ? " + p1.equals(p2));
14         // true
15         System.out.println("p1 starts with home? " + p1.startsWith(Paths.get("/home")));
16         // false
17         System.out.println("p1 starts with home? " + p1.startsWith("home")); //❶
18         // false
19         System.out.println("p2 ends with txt? " + p2.endsWith("txt")); //❷
20         // true
21         System.out.println("p2 ends with myfile.txt? " + p2.endsWith("myfile.txt")); //
22         // true
23         System.out.println("p2 ends with myfile.txt? " + p2.endsWith("subaochen/myfile.
24             txt"));
25     }
26 }
```

❶ home 是相对路径, /home 是绝对路径, 是两个不同的路径对象

❷ txt 不是 p2 路径数组中的元素

❸ myfile.txt 是 p2 路径数组中的元素

代码清单 7.9: PathExistTest.java

```

1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.nio.file.Files;
4 import java.nio.file.Path;
5 import java.nio.file.Paths;
6
7 /**
8  * @author subaochen.
9 */
10 public class PathExistTest {
11     public static void main(String[] args) {
12         Path path = Paths.get(System.getProperty("user.home") + "myfile.txt");
13         Path root = Paths.get("/root");
14         System.out.println("myfile.txt exists? " + Files.exists(path));
15         System.out.println("/root exists? " + Files.notExists(root));// ①
16     }
17 }
18 //
```

① 如果是普通用户则无权查看 root 目录

7.6.1 检查文件或者目录

在 7.5.3 [在第 192 页] 中我们看到, Path 类的大多数方法不会检查文件或者目录是否存在, 换句话说, Path 中的大多数方法只是对给定的路径进行语法上的检查和操作。Files 类的 exists 和 notExists 方法可以用来检查一个 Path 对象是否存在于实际的文件系统中, 参见代码清单 7.9。

exists 返回 false 可能存在两种情况:

1. 文件或者目录不存文件或者目录不可见, 即当前用户没有权限查看该文件或者目录

^aJDK 1.7 引入了新的文件操作 API, 即 NIO.2, 本节内容着重于 NIO.2, 不再涉及旧的文件 API。

^bJDK 1.7 引入了新的文件操作 API, 即 NIO.2, 本节内容着重于 NIO.2, 不再涉及旧的文件 API。

^cJDK 1.7 引入了新的文件操作 API, 即 NIO.2, 本节内容着重于 NIO.2, 不再涉及旧的文件 API。

我们可以使用 isReadable、isWritable、isExecutable 进一步检查文件是否可读、可写、可执行, 比如下面的代码片段:

```

1 Path file = ...;
2 boolean isRegularExecutableFile = Files.isRegularFile(file) &
3     Files.isReadable(file) & Files.isExecutable(file);
```

代码清单 7.10: PathDeleteTest.java

```
1 package cn.edu.sdut.softlab.io;
2
3 import java.io.IOException;
4 import java.nio.file.*;
5
6 /**
7  * @author subaochen.
8 */
9 public class PathDeleteTest {
10     public static void main(String[] args) {
11         Path path = Paths.get(System.getProperty("user.home") + "/testfile.txt");
12         try {
13             Files.delete(path);
14         } catch (NoSuchFileException e) {
15             System.err.println("no such file or directory:" + path);
16         } catch (DirectoryNotEmptyException e) {
17             System.err.println("directory is not empty: " + path);
18         } catch (IOException e) {
19             System.err.println(e);
20         }
21     }
22 }
```

7.6.2 删除文件或者目录

Files.delete 方法删除文件或者目录。需要注意的是，如果要删除的是目录，则目录必须是空的，否则删除失败。示例代码参见代码清单7.10。

7.6.3 复制文件或者目录

Files.copy 方法可以复制文件或者目录。默认情况下，copy 方法不会覆盖目的文件，但是可以通过传递 CopyOption 参数影响复制的过程：

- REPLACE_EXISTING：覆盖目的文件。
- COPY_ATTRIBUTES：也复制文件属性到目的文件。如果不设置此选项，只是复制文件本身，文件属性取决于目的目录的设置。
- NOFOLLOW_LINKS：复制符号链接而非符号链接指向的文件。

示例程序参见代码清单7.11。

代码清单 7.11: PathCopyTest.java

```
1 package cn.edu.sdut.softlab.io;
```

```
2
3 import java.io.IOException;
4 import java.nio.file.*;
5
6 import static java.nio.file.LinkOption.NOFOLLOW_LINKS;
7 import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;
8
9 /**
10  * 本类演示了Files.copy的用法.
11 *
12 * @author subaochen.
13 */
14 public class PathCopyTest {
15     public static void main(String[] args) {
16
17         // 不带CopyOption参数复制文件
18         // 请首先在家目录创建一个测试目录test和testdest，并创建测试文件myfile.txt
19         // mkdir test testdest
20         // touch test/myfile.txt
21         Path src = Paths.get(System.getProperty("user.home") + "/test/myfile.txt");
22         Path dest = Paths.get(System.getProperty("user.home") + "/testdest/myfile.txt");
23         // ①
24         try {
25             Files.copy(src, dest);
26         } catch (IOException e) {
27             System.err.println(e);
28         }
29         System.out.println("copy success ? " + Files.exists(dest));
30
31         // 覆盖目的文件
32         try {
33             Files.copy(src, dest, REPLACE_EXISTING); // ②
34         } catch (IOException e) {
35             System.err.println(e);
36         }
37         System.out.println("copy replace existing success ? " + Files.exists(dest));
38
39         // 带目录复制时的情形
40         // 为了防止上面的操作干扰复制目录的效果，先删除testdest目录下的myfile.txt
41         try {
42             Files.delete(dest);
43         } catch (IOException e) {
44             System.err.println(e);
45         }
46         src = Paths.get(System.getProperty("user.home") + "/test");
47         dest = Paths.get(System.getProperty("user.home") + "/testdest");
48         try {
49             Files.copy(src, dest, REPLACE_EXISTING);
50         } catch (IOException e) {
51             System.err.println(e);
52         }
53         // 应该打印出false
```

```

53     System.out.println("copy directory success ? " + Files.exists(dest.resolve("/myfile.txt")));
54
55     // 复制符号链接时的情形
56     // 先在test目录创建一个符号链接: ln -s /etc/passwd test/passwd
57     src = Paths.get(System.getProperty("user.home") + "/test/passwd");
58     dest = Paths.get(System.getProperty("user.home") + "/testdest/passwd");
59     try {
60         Files.copy(src, dest, REPLACE_EXISTING, NOFOLLOW_LINKS); // ②
61     } catch (IOException e) {
62         System.err.println(e);
63     }
64     System.out.println("copy soft link success ? " + Files.exists(dest,
65                         NOFOLLOW_LINKS));
66 }
67 }
68 //

```

- ① 如果目的文件是目录会怎样？比如 `dest` 写为: `Paths.get(System.getProperty("user.home") + "/testdest")`, 请自行测试并思考原因
 ② 注意观察 `REPLACE_EXISTING` 的作用: 到 `testdest` 目录查看是否存在 `myfile.txt` 文件
 ③ 可以尝试去掉 `REPLACE_EXISTING` 和 `NOFOLLOW_LINKS` 观察执行的结果

Files.copy 也支持复制文件到流，或者从流复制到文件：

- `copy(InputStream, Path, CopyOption...options)`
- `copy(Path, OutputStream)`

7.6.4 移动文件或者目录

`Files.move` 可以移动文件或者目录，文件或者目录改名也是要通过 `move` 方法，参见代码清单 7.12。

代码清单 7.12: PathMoveTest.java

```

1 package cn.edu.sdut.softlab.io;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7
8 import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;
9
10 /**
11  * 本类演示了Files.move方法的用法.
12  * @author subaochen.

```

```

13  /*
14  public class PathMoveTest {
15      public static void main(String[] args) {
16          Path src = Paths.get(System.getProperty("user.home") + "myfile.txt");
17          Path dest = Paths.get(System.getProperty("user.home") + "/test/myfile.txt");
18
19          try {
20              // 如果目标文件存在则先删除, 防止干扰move的结果
21              Files.delete(dest);
22              Files.move(src, dest, REPLACE_EXISTING); // ❶
23          } catch (IOException e) {
24              e.printStackTrace();
25          }
26
27          System.out.println("move file success? " + Files.exists(dest));
28      }
29  }

```

❶ 可以尝试去掉 REPLACE_EXISTING 参数看看运行结果有什么不同 ?

7.6.5 操作文件或者目录的属性

我们经常见到元数据 (metadata) 这个说法。简单的说, 元数据是描述数据的数据, 或者说, 一类事物的元数据描述了一类事物的属性。对照面向对象的概念我们可以看出, 元数据非常像类的属性。Java 提供了一组读取或者设置文件属性的方法, 如表7.11所示。

表 7.11: File 操作文件属性的方法

| 属性 | 方法 | 描述 |
|------|---|--|
| 文件大小 | public static long size(Path path) throws IOException | 获取文件的大小 (字节数) |
| 创建时间 | | 没有提供直接的方法获取文件的创建时间, 需要首先获取 BasicFileAttributeView, 然后解析 creationTime, 比如: Path file = ...; BasicFileAttributes attr = Files.readAttributes(file, BasicFileAttributes.class); System.out.println("creationTime: " + attr.creationTime()); |

| | | |
|--------|---|-----------------|
| 最后修改时间 | <pre>public static FileTime getLastModifiedTime(Path path, LinkOption... options) throws IOException public static Path setLastModifiedTime(Path path, FileTime time) throws IOException</pre> | 读取和设置文件的最后修改时间。 |
| 属主 | <pre>public static UserPrincipal getOwner(Path path, LinkOption... options) throws IOException public static Path setOwner(Path path, UserPrincipal owner) throws IOException</pre> | 读取或者设置文件的属主 |
| 所属组 | 没有提供相应的方法 | |
| 访问控制属性 | <pre>public static Set<PosixFilePermission> getPosixFilePermissions(Path path, LinkOption... options) throws IOException public static Path setPosixFilePermissions(Path path, Set<PosixFilePermission> perms) throws IOException</pre> | 读取或者设置文件的访问控制属性 |
| 是否目录 | <pre>public static boolean isDirectory(Path path, LinkOption... options)</pre> | 判断给定的 Path 是否目录 |

| | | |
|--------|--|--|
| 是否普通文件 | <code>public static boolean isRegularFile(Path path, LinkOption... options)</code> | 判断给定的 Path 是否普通文件 |
| 是否符号连接 | <code>public static boolean isSymbolicLink(Path path)</code> | 判断给定的 Path 是否符号链接 |
| 是否隐藏文件 | <code>public static boolean isHidden(Path path) throws IOException</code> | 判断给定的 Path 是否隐藏文件。 注意到，Windows 和 Linux 判断隐藏文件的方法是不同的， Linux 的隐藏文件以“.”开头，而 Windows 的隐藏文件设置了“隐藏”属性。 |

利用表7.11中的方法，一次只能读取或者设置一个文件属性，如果要同时读取或者设置多个文件属性显然比较低效，因此 Java 提供了批量读取或者设置文件属性的方法，参见表7.12。

| 方法 | 描述 |
|---|---|
| <code>public static Map<String, Object> readAttributes(Path path, String attributes, LinkOption... options) throws IOException</code> | 批量读取文件属性，参数 <code>attributes</code> 给出了所要读取的属性列表，* 表示所有属性。属性名称和 <code>FileAttributes</code> 各子类的属性字段定义相同，参见 https://docs.oracle.com/javase/8/docs/api/java/nio/file/attribute/FileAttributeView.html |
| <code>public static <A extends BasicFileAttributes> A readAttributes(Path path, Class<A> type, LinkOption... options) throws IOException</code> | 批量读取文件属性，参数 <code>type</code> 是 <code>BasicFileAttributes</code> 的子类，返回结果和请求参数 <code>type</code> 相同。此方法更加“面向对象”一些，在 IDE 的帮助下不容易出错，因此建议采用此方法批量读取文件属性。 |

表 7.12: 批量读取文件属性的方法

不同操作系统的文件系统存在不小的差别，文件和目录的属性也各不相同，Java 为了能够隐藏这些差异和细节，将文件和目录的属性做了进一步的封装，参见图 7.13 [在对页]。

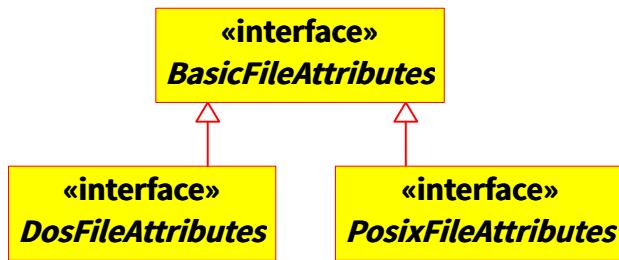


图 7.13: 文件属性接口

BasicFileAttributes 及其子接口只是定义了读取文件属性的方法，如何更新（修改）文件属性呢？Java 进一步封装了在各种情况下操作文件属性的 AttributeView 类，即可以读取文件属性，也可以更新文件属性，同时屏蔽了不同操作系统的差异，如图7.14所示。

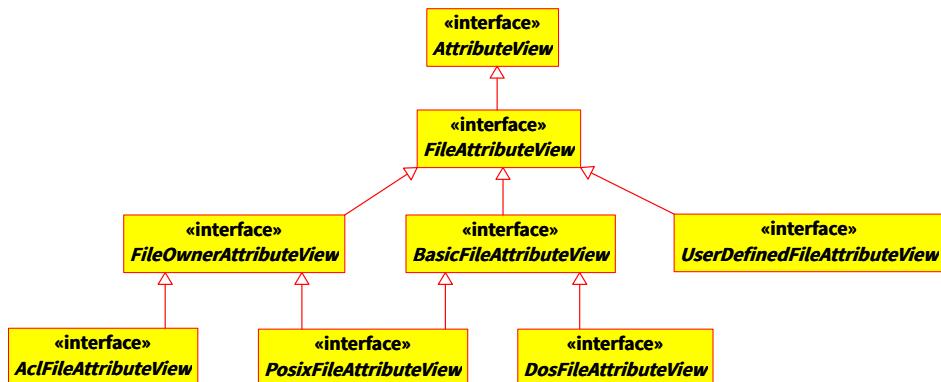


图 7.14: 读取和设置文件属性的 View 接口

例 7.4. 读取文件属性，包括：文件大小、最后修改时间、是否普通文件等。

代码设计 参见代码清单7.13。

代码清单 7.13: PathMetadataTest.java

```

1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7 import java.nio.file.attribute.BasicFileAttributes;
8 import java.nio.file.attribute.FileTime;
9
10 /**

```

```
11  * @author subaochen.
12 */
13 public class PathMetadataTest {
14     public static void main(String[] args) {
15         Path file = Paths.get(System.getProperty("user.home") + "/test/myfile.txt");
16
17         // 批量读取文件的基本属性
18         BasicFileAttributes attr = null;
19         try {
20             attr = Files.readAttributes(file, BasicFileAttributes.class);
21         } catch (IOException e) {
22             System.err.println(e);
23         }
24         System.out.println("creationTime: " + attr.creationTime());
25         System.out.println("lastAccessTime: " + attr.lastAccessTime());
26         System.out.println("lastModifiedTime: " + attr.lastModifiedTime());
27         System.out.println("isDirectory: " + attr.isDirectory());
28         // 不是目录，也不是文件，也不是符号链接的话 (@TODO比如？)
29         System.out.println("isOther: " + attr.isOther());
30         System.out.println("isRegularFile: " + attr.isRegularFile());
31         System.out.println("isSymbolicLink: " + attr.isSymbolicLink());
32         System.out.println("size: " + attr.size());
33
34         // 设置文件的最后修改时间为当前时间
35         long currentTime = System.currentTimeMillis();
36         FileTime ft = FileTime.fromMillis(currentTime);
37         try {
38             Files.setLastModifiedTime(file, ft);
39         } catch (IOException e) {
40             System.err.println(e);
41         }
42
43         // 验证一下最后修改时间修改成功了吗？
44         try {
45             System.out.println("now lastModifiedTime is:" + Files.getLastModifiedTime(
46                 file));
47         } catch (IOException e) {
48             System.err.println(e);
49         }
50     }
51 }
```

运行结果 在 Idea 中运行结果如下：

```

creationTime: 2016-12-12T02:49:04Z
lastAccessTime: 2016-12-12T02:51:47Z
lastModifiedTime: 2016-12-12T02:49:04Z
isDirectory: false
isOther: false
isRegularFile: true
isSymbolicLink: false
size: 0
now lastModifiedTime is:2016-12-13T23:54:11Z

```

代码分析和说明 使用表7.11中的方法一次只能读取或者设置一个文件的属性，因此如果要一次读取或者设置多个文件属性的话，建议使用 `readAttributes` 方法较为高效。

7.6.6 创建、读写文件

7.6.6.1 读写小文件

Files 类为读写小文件²⁴提供了方便的 `write` 和 `readAllBytes` 方法，参见表7.13。可以看出，表7.13中的方法不需要和输入输出流直接打交道。实际上，Files 类的方法是对输入输出流的进一步封装，比如 `readAllBytes` 方法²⁵：

```

1  public static byte[] readAllBytes(Path path) throws IOException {
2      try (SeekableByteChannel sbc = Files.newByteChannel(path);
3           InputStream in = Channels.newInputStream(sbc)) {
4          long size = sbc.size();
5          if (size > (long)MAX_BUFFER_SIZE)
6              throw new OutOfMemoryError("Required array size too large");
7
8          return read(in, (int)size);
9      }
10 }

```

表 7.13：读写小文件的方便方法

| 方法 | 描述 |
|----|----|
|----|----|

²⁴多小的文件算是小文件？因为 `readAllBytes` 是把文件全部内容读到一个 `byte` 数组中，因此文件的尺寸取决于你的内存多少。但是，并不是所有的内存都可以用来保存文件内容的，因此使用这里的方法时要考虑到能够处理的文件的最大尺寸限制。

²⁵参见 openjdk 的 `jdk/src/java.base/share/classes/java/nio/file/Files.java`

```
public static Path  
write(Path path,  
byte[] bytes,  
OpenOption...  
options) throws  
IOException
```

将 bytes 数组写入到文件 path 中。
OpenOption 在 StandardOpenOption 这个 Enum 中定义：

- READ：为读打开文件
- WRITE：为写打开文件
- APPEND：如果可写则追加写入内容到文件末尾
- CREATE：如果文件不存在则创建
- CREATE_NEW：创建新文件；如果文件已经存在则失败，优先级高于 CREATE
- TRUNCATE_EXISTING：如果文件可写并且已经存在则截断文件尺寸为 0
- DELETE_ON_CLOSE：文件操作结束 (close) 后自动删除文件，这对于临时文件很有用
- DSYNC：自动同步文件的内容到存储介质
- SYNC：自动同步文件的内容和原信息到存储介质
- SPARSE：创建稀疏文件²⁶，如果文件系统支持的话

默认是 CREATE, TRUNCATE_EXISTING 和 WRITE，即如果文件不存在则创建，如果文件已存在则截断长度为 0。

²⁶稀疏文件是指创建文件的时候并不真正分配空间，只有真正写入文件的时候才逐步分配空间。

| | |
|--|--|
| public static Path write(Path path, Iterable<? extends CharSequence> lines, Charset cs, OpenOption... options) throws IOException | 将文本 lines 写入文件 path 中。lines 以 System.getProperty("line.separator") 为 行分隔符，并使用给定的 (cs) 编码 ²⁷ 为 byte 数组。 |
| public static byte[] readAllBytes(Path path) throws IOException | 从 path 读取所有内容到 byte 数组。 |
| public static List<String> readAllLines(Path path) throws IOException | 读文件 path 的所有行到 List<String> 中， 行分隔符为\r\n (windows 下) 或\r (Mac 下) 或\n (Linux 下)。 |
| public static List<String> readAllLines(Path path, Charset cs) throws IOException | 读文件 path 的所有行到 List<String> 中， 并使用 cs 编码将 byte 解码为字符串。 |

例 7.5. 读写小文件

代码设计 参见代码清单7.14。

代码清单 7.14: PathCreateTest.java

```

1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7
8 /**
9  * @author subaochen.
10 */

```

²⁷Java 1.7 之后引入了 StandCharsets 类定义了常见的编码方式, 参见:<https://docs.oracle.com/javase/8/docs/api/java/nio/charsets.html>

```

11 public class PathCreateTest {
12     public static void main(String[] args) {
13         Path file = Paths.get(System.getProperty("user.home") + "/test/myfile1.txt");
14         byte[] buf = "test string".getBytes();
15         byte[] fileArray;
16         try {
17             Files.write(file, buf); // ①
18             fileArray = Files.readAllBytes(file); // ②
19             System.out.println(new String(fileArray));
20         } catch (IOException e) {
21             e.printStackTrace();
22         }
23     }
24 }
25 //
```

① 直接写入 byte 数组到文件。比较一下 `InputStream` 中的相关方法可见, `Files.write` 更直观和方便。

② 直接从 `Path` 读 bytes 数组。比较一下 `OutputStream` 中的相关方法可见, `Files.readAllBytes` 更直观和方便。

运行结果 在 Idea 中运行结果如下, 只是简单的输出了文件的内容:

```
test string
```

代码分析和说明

7.6.6.2 带缓存的文本文件处理方法

处理文本文件时, 通常需要借助于缓存提高效率, 我们在 section 7.2.2 中已经讨论过。`Files` 类提供了如表 7.14 的方法更进一步简化了文本文件的处理。

下面的代码片段演示了 `newBufferedReader` 的用法:

```

1 Charset charset = Charset.forName("UTF-8");// or Charset charset = StandardCharsets.UTF
-8;
2 try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
3     String line = null;
4     while ((line = reader.readLine()) != null) {
5         System.out.println(line);
6     }
7 } catch (IOException x) {
8     System.err.format("IOException: %s%n", x);
9 }
```

等价于下面的代码片段 (如果文件是 UTF-8 编码的话):

```

1 try (BufferedReader reader = Files.newBufferedReader(file)) {
2     String line = null;
```

| 方法 | 描述 |
|---|---|
| public static BufferedWriter new- BufferedWriter(Path path, OpenOption... options) throws IOException | 从 path 创建一个 BufferedReader 用于读取 文件内容，使用 UTF-8 解码。 |
| public static BufferedReader new- BufferedReader(Path path, Charset cs) throws IOException | 从 path 创建一个 BufferedReader 用于读取 文件内容，文件内容以 cs 方式解码，默认使用 UTF-8 解码。 |
| public static BufferedWriter new- BufferedWriter(Path path, OpenOption... options) throws IOException | 从 path 创建一个 BufferedWriter 用于写入 文件内容，使用 UTF-8 编码。 |
| public static BufferedWriter new- BufferedWriter(Path path, Charset cs, OpenOption... options) throws IOException | 从 path 创建一个 BufferedWriter 用于写入 文件内容，文件内容以 cs 方式编码，默认使用 UTF-8 编码。 |

表 7.14: Files 中带缓存的文本读写方法

```

3     while ((line = reader.readLine()) != null) {
4         System.out.println(line);
5     }
6 } catch (IOException x) {
7     System.err.format("IOException: %s%n", x);
8 }

```

7.6.6.3 使用不带缓存的输入输出流

Files 同样封装了不带缓存的输入输出流，参见表7.15。

| 方法 | 描述 |
|---|---------------------------------------|
| public static InputStream newInputStream(Path path, OpenOption... options) throws IOException | 从 path 创建一个 InputStream 用于读取文件 内容 |
| public static OutputStream newOutput- Stream(Path path, OpenOption... options) throws IOException | 从 path 创建一个 OutputStream 用于写入文 件内容 |

表 7.15: Files 中不带缓存的输入输出流

例 7.6. Files 对不带缓存的输入流的封装

代码设计 参见代码清单7.15。

代码清单 7.15: PathWithoutBufferTest.java

```

1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.io.*;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7
8 import static java.nio.file.StandardOpenOption.*;

```

```
9
10 /**
11  * @author subaochen.
12 */
13 public class PathWithoutBufferTest {
14     public static void main(String[] args) {
15         Path file = Paths.get(System.getProperty("user.home") + "/test/myfile.txt"); //❶
16
17         // 首先写入文件一部分内容
18         String s = "Hello World! ";
19         // 将字符串转换为byte数组
20         byte data[] = s.getBytes();
21
22         try (OutputStream out = new BufferedOutputStream(
23             Files.newOutputStream(file, CREATE, TRUNCATE_EXISTING))) { //❷
24             out.write(data, 0, data.length);
25         } catch (IOException x) {
26             System.err.println(x);
27         }
28         // 读取文件内容
29         try (InputStream in = Files.newInputStream(file);
30             BufferedReader reader =
31                 new BufferedReader(new InputStreamReader(in))) {
32             String line = null;
33             while ((line = reader.readLine()) != null) {
34                 System.out.println(line);
35             }
36         } catch (IOException x) {
37             System.err.println(x);
38         }
39     }
40 } //
```

❶ 程序执行完可以 `cat myfile.txt` 查看文件内容是否有变化

❷ 在这里可以尝试其他的 `StandOpenOption` 选项，比如 `CREATE_NEW, DELETE_ON_CLOSE`

运行结果 在 Idea 中运行的结果如下（简单的打印出了之前写入的内容）：

```
Hello World!
```

代码分析和说明 `newInputStream` 返回一个不带缓存的输入流，但是这个例子又根据这个不带缓存的输入流构造了一个 `BufferedReader` 以便以行的方式读入文本。你可能会问，为什么不直接使用带缓存的 `newBufferedReader` 方法呢？是的，在这个例子中，使用 `newBufferedReader` 直接返回一个 `BufferedReader` 更方便，本例只是演示如何构造一个不带缓存的输入流。

7.6.6.4 channel 方式读写文件

流 (stream) 是按照字节为单位读写文件的, channel 是按照缓冲区为单位读写文件的, 也就是说, channel 是自然带缓冲的, 可以一次处理一个缓冲区。SeekableByteChannel 内部维护着一个表示当前位置的指针, 通过移动该指针可以实现随机文件读写, 参见 section 7.6.8。

Files 类的 Channel 读写文件方法见表7.16。

| 方法 | 描述 |
|---|--|
| <code>public static SeekableByteChannel newByteChannel(Path path, OpenOption... options) throws IOException</code> | 从 path 创建一个 SeekableByteChannel 用于读写文件内容, 默认是只读打开 Channel 的。 |
| <code>public static SeekableByteChannel newByteChannel(Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs) throws IOException</code> | 从 path 创建一个 SeekableByteChannel 用于读写文件内容, attrs 设置文件的属性。 |

表 7.16: Files 类的 Channel 读写文件方法

例 7.7. 使用 Channel 读写文件

代码设计 参见代码清单7.18。

代码清单 7.16: FileChannelTest.java

```

1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.io.IOException;
4 import java.nio.ByteBuffer;
5 import java.nio.CharBuffer;
6 import java.nio.channels.SeekableByteChannel;
7 import java.nio.charset.Charset;
8 import java.nio.charset.StandardCharsets;
9 import java.nio.file.Files;
10 import java.nio.file.Path;
11 import java.nio.file.Paths;
12
13 /**
14 * 本例演示了Channel方式读取文件的方法.

```

```

15  * 本例借鉴自: http://docs.oracle.com/javase/tutorial/essential/io/file.html
16  * @author subaochen.
17  */
18 public class FileChannelTest {
19     public static void main(String[] args) {
20         Path file = Paths.get(System.getProperty("user.home") + "/test/myfile.txt");
21
22         try (SeekableByteChannel sbc = Files.newByteChannel(file)) { // ①
23             ByteBuffer buf = ByteBuffer.allocate(10); // ②
24
25             String encoding = System.getProperty("file.encoding"); // ③
26             while (sbc.read(buf) > 0) { // ④
27                 buf.flip(); // ⑤
28                 System.out.print(Charset.forName(encoding).decode(buf));
29                 buf.clear(); // ⑥
30             }
31         } catch (IOException x) {
32             System.out.println("caught exception: " + x);
33         }
34     }
35 } //
```

- ① newByteChannel 默认返回只读的 Channel
- ② allocate 创建一个指定字节的 ByteBuffer，本例中，sbc 这个 Channel 每次读取 10 个字节
- ③ 获得当前系统文件编码方式，以便读取文件字节后解码
- ④ 从通道读数据到缓冲区
- ⑤ 切换缓冲区为读模式
- ⑥ 清空缓冲区，准备写入下一轮数据

运行结果 打印出 myfile.txt 的内容，不再列出。

代码分析和说明 和 Stream 不同，使用 Channel 读写文件的要点是一次处理一个 buffer，因此如何使用 Buffer 就称为用好 Channel 的关键。可以借助于 Java 的 Buffer 类，如图7.15所示。

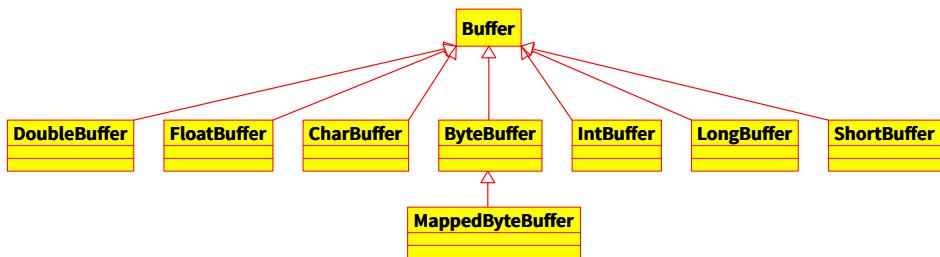


图 7.15: Buffer 的类层次结构

下面的讨论以 ByteBuffer 为例，其他类型的 Buffer 用法类似。

ByteBuffer 实际上是一块可以写入和读取数据的内存，Java 提供了若干方便的方法操作这块内存，参见图7.16，我们假设创建了一个大小为 8 个字节的 ByteBuffer。要理解 ByteBuffer 的用法，需要首先理解 ByteBuffer 的三个基本属性：

- **capacity**: ByteBuffer 的大小，即占用多少个字节的内存空间。每个 ByteBuffer 在创建时需要指定 capacity，一旦设定不允许改变，这里 capacity=8。
- **limit**: 下一次读或者写允许操作的字节数。在写模式下，limit 表示当前最多能够向 Buffer 写多少数据。在读模式下，limit 表示当前最多能够读到多少数据。
- **position**: 下一个可以读或者写的字节的位置索引，position 的最大值为 capacity - 1。在写模式下，position 的初始值为 0，当写入数据（字节）后，position 移动到下一个可写入的位置，比如在图7.16中，在写入 3 个字节的数据后，当前 position=3。如果此时切换这个 ByteBuffer 到读模式，则 position 需要指向下一个可读的字节位置，显然应该将 position 置为 0，limit 置为 3（即当前 position 的值），即从 Buffer 的开头开始能够一次读 3 个字节的数据，如图7.16的“读模式”所示。

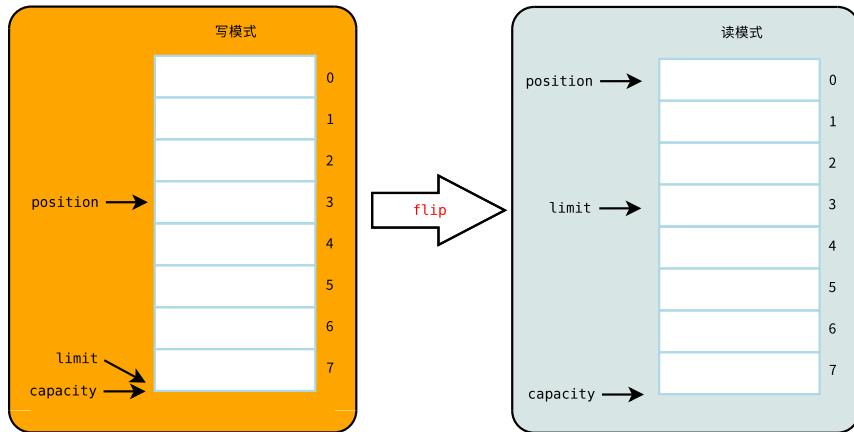
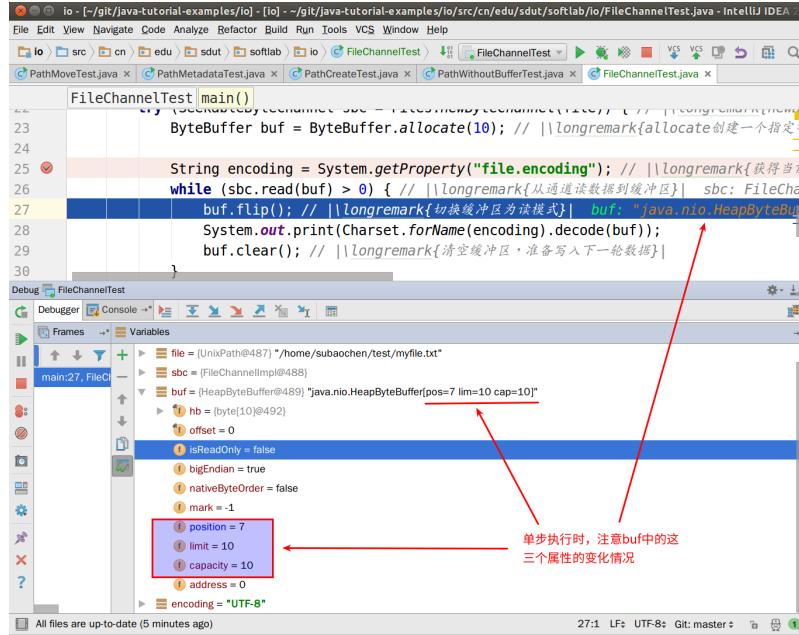


图 7.16: ByteBuffer 的用法

Buffer 通过 flip 方法从写模式切换到读模式，通常对 Buffer 的操作流程为：

1. 写入数据到 Buffer，在这里是通过 Channel 的 read 方法写入数据到 Buffer，也可以通过 Buffer 提供的各种 put 方法写入数据。
2. 调用 flip 方法切换到读模式。
3. 从 Buffer 中读取数据。
4. 调用 clear 或者 compact 方法清空缓冲区。

使用 Idea 可以方便的观察和学习 Buffer 中 capacity、limit、position 的变化情况，如下图所示。



读写文件的方法很多,何时使用 `InputStream/OutputStream/InputStreamReader/OutputStreamWriter`, 何时使用 `Files` 方法读写文件? 一般的原则是什么?

7.6.7 创建文件和临时文件

`Files` 也提供了创建文件和临时文件的方便 API, 见表7.17。`createFile` 和 `createTempFile` 被设计为“原子操作”, 即首先检查文件是否存在(存在则抛出异常), 然后创建一个空文件并设置为指定或者默认的属性, 因此 `createFile` 的安全性要比其他方法高些。

例 7.8. 使用 `createFile` 创建文件

代码设计 参见代码清单7.17。

代码清单 7.17: FileCreateTest.java

```

1 package cn.edu.sdut.softlab.io;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
```

| 方法 | 描述 |
|--|--------------------------|
| <code>public static Path createFile(Path path, FileAttribute<?>... attrs) throws IOException</code> | 根据 path 创建属性为 attrs 的文件 |
| <code>public static Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs) throws IOException</code> | 根据 dir 创建属性为 attrs 的临时文件 |
| <code>public static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs) throws IOException</code> | 根据 dir 创建属性为 attrs 的临时文件 |

表 7.17: Files 创建文件和临时文件的方法

```

6 import java.nio.file.Paths;
7
8 /**
9  * @author subaochen.
10 */
11 public class FileCreateTest {
12     public static void main(String[] args) {
13         Path file = Paths.get(System.getProperty("user.home") + "/test/testcreate.txt");
14         try {
15             Files.createFile(file); // ①
16         } catch (IOException e) {
17             System.err.println(e);
18         }
19     }
20 } //
```

① 重复执行此程序会发现抛出了 `FileNotFoundException`, 即 `createFile` 首先检查文件是否已经存在

运行结果 执行应用程序后检查所创建文件的属性如下:

```

~/test$ ls -l testcreate.txt
-rw-rw-r-- 1 subaochen subaochen 0 12 月 16 08:20 testcreate.txt

```

代码分析和说明 `createFile` 时如果没有给出文件属性则默认创建 664 属性的文件，即用户自己、所属组都可读写，其他所有人均只读。可以通过给出 `FileAttribute` 灵活设置属性，比如下面的代码片段设置为只对用户自己可读写：

```

1 Path file = ...;
2 Set<PosixFilePermission> perms =
3     PosixFilePermissions.fromString("rw-----");
4 FileAttribute<Set<PosixFilePermission>> attr =
5     PosixFilePermissions.asFileAttribute(perms);
6 Files.createFile(file, attr);

```

练习 7.5. 编写程序，在/tmp 创建一个临时文件 `temp.log`，并观察此临时文件的属性

练习 7.6. 编写程序，创建一个对所有人只读的文件，并验证所创建文件的属性

7.6.8 随机读写文件

在 section 7.6.6.1 中其实我们已经看到，`SeekableByteChannel` 已经具有随机读写文件的能力了：`SeekableByteChannel` 的下列方法帮助我们确定读写的起始位置和数量：

| 方法 | 描述 |
|--|--|
| <code>long position() throws IOException</code> | 返回文件的当前位置，即读和写的起始位置 |
| <code>SeekableByteChannel position(long newPosition) throws IOException</code> | 设置 Channel 的当前位置，返回这个 Channel（便于函数式编程） |
| <code>long size() throws IOException</code> | 返回这个 Channel 的大小 |
| <code>SeekableByteChannel truncate(long size) throws IOException</code> | 截断这个 Channel 到给定的大小 |
| <code>int read(ByteBuffer dst) throws IOException</code> | 从 Channel 读数据到 dst |
| <code>int write(ByteBuffer src) throws IOException</code> | 将 dst 的数据写入 Channel |

表 7.18: `SeekableByteChannel` 的随机读写方法

为了更方便的操作文件，JDK 提供了对 `SeekableByteChannel` 的进一步封装：`FileChannel`，除 `SeekableByteChannel` 中的方法外，`FileChannel` 增加了一些

高级特性，比如可以将文件的指定区域映射到内存中以便快速访问，锁定文件的指定区域（不允许其他线程进行操作），从任意位置直接读写文件等。我们有两种方式获取一个 FileChannel：

1. 通过 Path.newByteChannel 获得一个 SeekableByteChannel，然后强制类型转换为 FileChannel。
2. 通过 FileInputStream.getChannel 获取一个 FileChannel。
3. 通过 FileChannel.open 方法直接获得一个 FileChannel。推荐采用此种方式。

例 7.9. 使用 FileChannel 读写文件，假设文件 myfile.txt 原来有如下的内容：

```
Java programming language is good!
```

代码设计 参见代码清单7.18

代码清单 7.18: FileRandomAccessTest.java

```
1 package cn.edu.sduot.softlab.io;
2
3 import java.io.BufferedReader;
4 import java.io.BufferedWriter;
5 import java.io.IOException;
6 import java.nio.ByteBuffer;
7 import java.nio.channels.FileChannel;
8 import java.nio.file.Files;
9 import java.nio.file.Path;
10 import java.nio.file.Paths;
11
12 import static java.nio.file.StandardOpenOption.*;
13
14 /**
15 * 本类演示了随机读写文件的方法。
16 * 借鉴自: http://docs.oracle.com/javase/tutorial/essential/io/rafs.html
17 * @author subaochen.
18 */
19 public class FileRandomAccessTest {
20     public static void main(String[] args) {
21         Path file = Paths.get(System.getProperty("user.home") + "/test/myfile.txt");
22         // 为了保证每次运行结果都一致，首先创建一个实验性的文件
23         try {
24             BufferedWriter bw = Files.newBufferedWriter(file, CREATE, TRUNCATE_EXISTING);
25             bw.write("Java programming language is good!");
26             bw.flush(); // ❶
27         } catch (IOException e) {
28             System.err.println(e);
29         }
30     }
31 }
```

```
30
31     String s = "I was here!\n";
32     byte[] data = s.getBytes();
33
34     ByteBuffer out = ByteBuffer.wrap(data);
35     ByteBuffer copy = ByteBuffer.allocate(12);
36     try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {
37         int nread; //❶
38         do {
39             nread = fc.read(copy);
40         } while (nread != -1 && copy.hasRemaining());
41
42         fc.position(0);
43         while (out.hasRemaining()) //❷
44             fc.write(out);
45
46
47         out.rewind(); // ❸
48
49         long length = fc.size();
50         fc.position(length); //❹
51         copy.flip(); // ❺
52         while (copy.hasRemaining())
53             fc.write(copy); //❻
54         while (out.hasRemaining())
55             fc.write(out); // ❼
56
57         // 打印出文件的内容
58         BufferedReader br = Files.newBufferedReader(file);
59         // 这里可以使用lambda表达式简化代码
60         //br.lines().forEach(p->System.out.println(p));
61         while((s = br.readLine()) != null)
62             System.out.println(s);
63     } catch (IOException x) {
64         System.err.println("I/O Exception: " + x);
65     }
66 }
67 } //
```

- ❶ 如果没有 flush 会怎样？可以测试一下
- ❷ 把文件开头的 12 个字节的数据保存到 copy 中
- ❸ 把 "I was here!" (在变量 out 中) 写入到文件的开头
- ❹ 将 out 的 position 重新重新置为 0，以便下次从头开始操作
- ❺ 将文件的 position 移动到文件末尾，以便从末尾开始追加数据
- ❻ 将变量 copy (其中存储了文件开头的 12 个字节的内容) 的 position 置为 0，以便从头开始读缓冲区内容
- ❼ 把 copy 的内容追加到文件最后
- ❼ 将 "I was here!" 继续追加到文件最后

运行结果 在 Idea 中运行结果如下：

```
I was here!  
ming language is good!Java programI was here!
```

也就是说，在文件的开头写入了“`I was here!\n`”，然后把文件开头原先的“`Java program`”移动到了文件的最后，并在最后追加了“`I was here!\n`”。

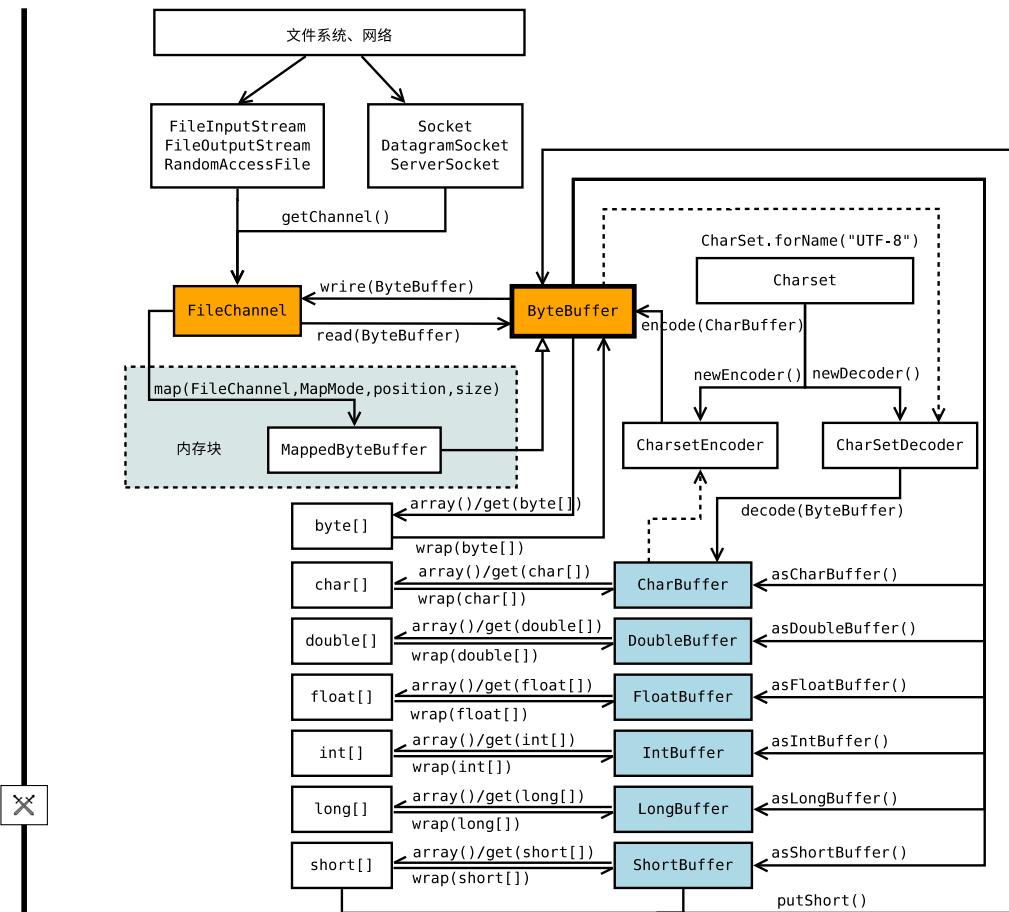
本程序可以反复运行，结果完全一致。

代码分析和说明 FileChannel 提供的文件随机读写概念，重要的是理解两个 position 的用法：

- FileChannel 的 position，用于确定从什么地方开始读写文件；
- ByteBuffer 的 position，用于确定从什么地方开始读写 ByteBuffer；



FileChannel、InputStream/OutputStream、ByteBuffer 之间的关系和用法可以形象的用下图表示：



简单的说，`ByteBuffer` 处于“中心”的位置，无论是从文件系统、网络读入的数据，还是准备写入文件系统、网络的数据，都是通过 `ByteBuffer` 的。因此，其他数据类型如何转换为 `ByteBuffer` 以及 `ByteBuffer` 如何转换为其他数据类型就显得非常重要，上图的大部分篇幅展示了这种相互转换的方法。

我们可以看出，`ByteBuffer` 转换为其他数据类型已经非常方便了，但是从其他数据类型转换为 `ByteBuffer` 的通道却不是很流畅，比如从 `short[]` 转换为一个 `ByteBuffer`，目前没有直接的方法，只能通过遍历这个 `short` 数组，然后将数组的每个元素放入 `ByteBuffer` 这种稍微曲折的方法：

```

1 ByteBuffer bb;
2 ...
3 for(short s:short_array) {
4     bb.putShort(s);
5 }
```

相信 JDK 的未来版本会提供更丰富和合理的 `ByteBuffer` 和其他数据类型的相互转换方法。

7.6.9 目录操作

目录是一种特殊的文件，我们前面介绍的很多 API（所有接受 Path 作为参数的 API）即可以操作文件，也可操作目录。但是毕竟目录的操作有其特殊性，比如下面的情形：

- 创建一个空目录
- 列出目录下的所有子目录
- 列出目录下的所有文件
- 根据给定的规则列出目录下的文件和目录

JDK 的 Files 类同样给出了目录操作的方便方法，分述如下。

7.6.9.1 创建目录

Files 类创建目录主要是两个方法，createDirectory 用于创建一级目录，createDirectories 用于创建多级目录，见表7.19。

| 方法 | 描述 |
|---|--|
| <code>public static Path createDirectory(Path dir, FileAttribute<?>... attrs) throws IOException</code> | 根据 dir 和 attrs 创建一个目录。如果没有 attrs 参数则创建默认属性的目录。如果目录已经存在则抛出 FileAlreadyExistsException 异常。 |
| <code>public static Path createDirectories(Path dir, FileAttribute<?>... attrs) throws IOException</code> | 根据 dir 创建多级目录 |

表 7.19: 创建目录的方法

例 7.10. 创建目录

代码设计 参见代码清单7.19。

代码清单 7.19: CreateDirectoryTest.java

```

1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
```

```
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7
8 /**
9  * @author subaochen.
10 */
11 public class CreateDirectoryTest {
12     public static void main(String[] args) {
13         Path dir = Paths.get(System.getProperty("user.home") + "/test/a");
14
15         try {
16             Files.createDirectory(dir);
17         } catch (IOException e) {
18             System.err.println(e);
19         }
20     }
21 }
```

代码分析和说明 第一次运行时终端没有任何显示，但是当我们去往 test 目录查看时，应该可以看到 test 目录下面多了一个新目录 a。

当第二次运行该程序时，终端显示：

```
java.nio.file.FileAlreadyExistsException: /home/subaochen/test/a
```

也就是说，如果 dir 目录已经存在，则 createDirectory 方法抛出 FileAlreadyExistsException 异常。

我们删除 a 目录，但是创建一个文件名字叫做 a，即在 test 目录执行下列命令：

```
~/test$ rmdir a; touch a
```

再次执行本程序，终端显示：

```
java.nio.file.FileAlreadyExistsException: /home/subaochen/test/a
```

即，如果要创建的目录和文件重名也是不允许的，同样抛出 FileAlreadyExistsException。

练习 7.7. 使用 Files.createDirectories 创建目录 test/a/b/c

7.6.9.2 列出目录

Files 的 newDirectoryStream 方法可以很方便的列出目录下的内容，包括子目录、文件、隐藏文件等，但是需要注意的是，newDirectoryStream 不是一个递归的过程，即不能深入到当前目录的子目录查找内容。

例 7.11. 利用 newDirectoryStream 方法列出当前目录下的文件、子目录等。

代码设计 参见代码清单7.20

代码清单 7.20: ListDirectoryTest.java

```

1 package cn.edu.sdu.t.softlab.io;
2
3 import java.io.IOException;
4 import java.nio.file.DirectoryStream;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8
9 /**
10  * 本类演示了如何遍历一个目录.
11  * @author subaochen.
12 */
13 public class ListDirectoryTest {
14     public static void main(String[] args) {
15         Path dir = Paths.get(System.getProperty("user.home") + "/test");
16
17         try(DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) { //❶
18             for(Path p : stream) {
19                 System.out.println(p);
20             }
21         } catch (IOException e) {
22             System.err.println(e);
23         }
24     }
25 }//

```

❶ 如果不使用 try-with-resources 结构需要调用 stream.close 方法

代码分析和说明 要注意到 newDirectoryStream 返回的是一个 stream, 因此和 InputStream 等用法类似, 一定要记得用完后 close 这个 stream。try-with-resources 结构能够自动关闭 stream, 如果使用 try-catch 结构, 则要在 finally 块中执行 stream.close() 方法关闭 stream。

练习 7.8. 如何利用 newDirectoryStream 方法递归的列出当前目录下的内容, 包括子目录下的文件及其子目录?

7.6.9.3 根据规则列出目录

newDirectoryStream 允许传入过滤规则作为参数, 过滤规则 glob 参见 section §??。

例 7.12. 列出当前目录下的所有 java 源代码和 java class 文件。

代码设计 参见代码清单7.21。

代码清单 7.21: ListDirectoryWithGlobTest.java

```

1 package cn.edu.sdut.softlab.io;
2
3 import java.io.IOException;
4 import java.nio.file.DirectoryStream;
5 import java.nio.file.Files;
6 import java.nio.file.Path;
7 import java.nio.file.Paths;
8
9 /**
10  * 本类演示了如何过滤一个目录.
11  * @author subaochen.
12 */
13 public class ListDirectoryWithGlobTest {
14     public static void main(String[] args) {
15         Path dir = Paths.get(System.getProperty("user.home") + "/test");
16
17         try(DirectoryStream<Path> stream = Files.newDirectoryStream(dir, "*.{java,class}")
18             ) { //❶
19             for(Path p : stream) {
20                 System.out.println(p);
21             }
22         } catch (IOException e) {
23             System.err.println(e);
24         }
25     }
26 }

```

❶ 如果不使用 `try-with-resources` 结构需要调用 `stream.close` 方法

代码设计和分析 和 例7.11相比，我们看到 `newDirectoryStream` 方法只是多了一个参数：表明如何过滤本目录下的内容。

7.6.10 遍历目录：FileVisitor 接口

在 section 7.6.9.2中我们看到了，可以利用 `Files.newDirectoryStream` 列出当前目录下的内容，但是 `newDirectoryStream` 不是递归处理的，因此 `Files` 类提供了遍历目录的另外方式：使用 `FileVisitor` 接口可以更方便的遍历目录。

首先实现 `FileVisitor` 接口，如代码清单7.22所示。

代码清单 7.22: PrintFiles.java

```

1 package cn.edu.sdut.softlab.io;
2
3 import java.io.IOException;

```

```
4 import java.nio.file.FileVisitResult;
5 import java.nio.file.Path;
6 import java.nio.file.SimpleFileVisitor;
7 import java.nio.file.attribute.BasicFileAttributes;
8
9 import static java.nio.file.FileVisitResult.CONTINUE;
10
11 /**
12  * 本类演示了如何使用FileVisitor.
13  * 本类借鉴自: http://docs.oracle.com/javase/tutorial/essential/io/walk.html
14  * @author subaochen.
15 */
16 public class PrintFiles
17     extends SimpleFileVisitor<Path> {
18
19     // Print information about
20     // each type of file.
21     @Override
22     public FileVisitResult visitFile(Path file,
23                                     BasicFileAttributes attr) {
24         if (attr.isSymbolicLink()) {
25             System.out.format("Symbolic link: %s ", file);
26         } else if (attr.isRegularFile()) {
27             System.out.format("Regular file: %s ", file);
28         } else {
29             System.out.format("Other: %s ", file);
30         }
31         System.out.println("(" + attr.size() + "bytes)");
32         return CONTINUE;
33     }
34
35     // Print each directory visited.
36     @Override
37     public FileVisitResult postVisitDirectory(Path dir,
38                                              IOException exc) {
39         System.out.format("Directory: %s%n", dir);
40         return CONTINUE;
41     }
42
43     // If there is some error accessing
44     // the file, let the user know.
45     // If you don't override this method
46     // and an error occurs, an IOException
47     // is thrown.
48     @Override
49     public FileVisitResult visitFileFailed(Path file,
50                                         IOException exc) {
51         System.err.println(exc);
52         return CONTINUE;
53     }
54 }
```

然后可以在主类中使用 `FileVisitor` 接口遍历目录了，如代码清单 7.23 所示。

代码清单 7.23: WalkFileTreeTest.java

```
1 package cn.edu.sdu.tutorial.io.walk;
2
3 import java.io.IOException;
4 import java.nio.file.Files;
5 import java.nio.file.Path;
6 import java.nio.file.Paths;
7
8 /**
9  * 本例演示了如何使用FileVisitor类遍历目录。
10 * 本例借鉴自：http://docs.oracle.com/javase/tutorial/essential/io/walk.html
11 * @author subaochen.
12 */
13 public class WalkFileTreeTest {
14     public static void main(String[] args) {
15         Path startingDir = Paths.get(System.getProperty("user.home") + "/test");
16         PrintFiles pf = new PrintFiles();
17         try {
18             Files.walkFileTree(startingDir, pf); //❶
19         } catch (IOException e) {
20             System.err.println(e);
21         }
22     }
23 } //
```

❶ 递归的遍历目录并根据 `PrintFiles` 类中的相关方法处理每一个目录项

这里的关键是，`walkFileTree` 方法会递归的遍历给定的目录，但是对于递归过程中遇到的每一个项目如何处理呢？我们看到 `walkFileTree` 的第二个参数是一个实现了 `FileVisitor` 接口的对象，这个对象决定了如何处理这些遇到的目录项目，如表 table 7.20 所示。

7.7 访问属性文件

属性文件通常用来保存应用程序的配置信息，这样当应用程序的配置改变时，只需要修改属性文件（配置文件）即可，不需要修改应用程序的源代码，维护比较方便。属性文件的内容通常是以 `key、value` 对的形式出现，即 `key=value` 的形式，比如数据库相关的应用程序中通常将数据库服务器的 IP 地址、用户名、密码等信息保存到如下的属性文件中：

| 方法 | 描述 |
|---|-----------------------------------|
| <code>FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs) throws IOException</code> | 在访问（操作）目录之前的动作，比如准备将目录复制到另外的目录中等。 |
| <code>FileVisitResult postVisitDirectory(T dir, IOException exc) throws IOException</code> | 在访问（操作）目录之后的动作，比如访问目录后可以将目录删除等。 |
| <code>FileVisitResult visitFile(T file, BasicFileAttributes attrs) throws IOException</code> | 访问目录中的文件。 |
| <code>FileVisitResult visitFileFailed(T file, IOException exc) throws IOException</code> | 如果访问文件失败则调用此方法。 |

表 7.20: FileVisitor 接口方法

```
hostname=192.168.1.200
username=postgres
password=password
database=mydb
```

Java 的 Properties 类提供了方便的 API 读写这样的属性文件。

例 7.13. 访问属性文件

代码设计 参见代码清单7.24。

代码清单 7.24: PropertyFileTest.java

```
1 package cn.edu.sdu.softlab.io;
2
3 import java.io.*;
4 import java.time.LocalDateTime;
5 import java.util.Properties;
6
7 /**
8 * 本类演示了访问属性文件的方法。
9 * @author subaochen.
```

```
10  */
11 public class PropertyFileTest {
12     private static final String CONF_FILE = "conf/database-config.properties";
13
14     public static void main(String[] args) {
15         createPropertyFile(CONF_FILE);
16         readPropertyFile(CONF_FILE);
17
18     }
19
20 /**
21 * 读取属性文件.
22 * @param file 属性文件名
23 */
24 private static void readPropertyFile(String file) {
25     Properties pro = new Properties();
26     try (InputStream path = new FileInputStream(CONF_FILE)) {
27         pro.load(path);
28         for(Object key: pro.keySet()) {
29             System.out.println(key.toString() + "=" + pro.getProperty(key.toString()))
30             ;
31         }
32     } catch (IOException e) {
33         System.err.println(e);
34     }
35
36 /**
37 * 创建属性文件.
38 * @param file 属性文件名
39 */
40 public static void createPropertyFile(String file) {
41     Properties pro = new Properties();
42     try (InputStream path = new FileInputStream(CONF_FILE);
43          OutputStream fos = new FileOutputStream(CONF_FILE)) {
44         pro.load(path);
45         pro.setProperty("hostname", "localhost");
46         pro.setProperty("username", "postgres");
47         pro.setProperty("password", "password");
48         pro.setProperty("database", "mydb");
49         pro.store(fos, String.format("modified @ %s", LocalDateTime.now()));
50     } catch (IOException e) {
51         System.err.println(e);
52     }
53 }
54 }
```

运行结果 执行本应用程序结果如下：

```
hostname=localhost  
password=password  
database=mydb  
username=postgres
```

代码设计和分析 本例中，我们使用了 FileInputStream 和 FileOutputStream 构造了一个文件输入输出流， FileInputStream 和 FileOutputStream 的参数是文件名，注意到该文件名是相对于项目根目录的。

实际上，Java 在定位文件时，有如下的几种策略：

- 绝对路径：在 FileInputStream、FileOutputStream 中如果参数的路径使用路径分隔符（windows 下是\，Linux 下是/）开头则是绝对路径。
- 相对于项目的路径：在 FileInputStream、FileOutputStream 中如果参数的路径没有使用路径分隔符（windows 下是\，Linux 下是/）开头则是项目于项目根目录的相对路径。
- 相对于当前 class 文件的路径：如果使用 Class.getResourceAsStream 方法，则获得相对于当前 class 文件的路径，比如：

```
InputStream path = PropertyFileTest.class.getResourceAsStream("database-  
config.properties");
```

path 对象指向的文件 database-config.properties 和 class 文件在同一个目录下，即 database-config.properties 位于 src/cn/edu/sdut/softlab 目录下。但是，如果参数是绝对路径，则获得是相对于包的文件路径，比如：

```
InputStream path = PropertyFileTest.class.getResourceAsStream("/database-  
config.properties");
```

则文件 database-config.properties 位于 src 目录下。

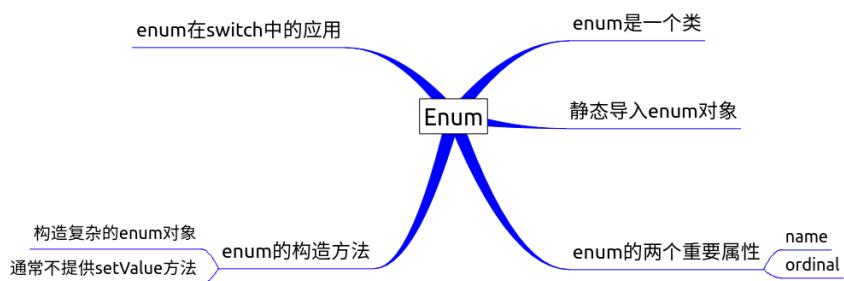
- 相对于包的路径：如果使用 Class.getClassLoader().getResourceAsStream 方法，则获得是相对于包路径的文件，比如：

```
InputStream path = PropertyFileTest.class.getClassLoader().getResourceAsStrea  
config.properties");
```

即文件 database-config.properties 位于 src 目录下。

同时需要注意到，Class.getClassLoader.getResourceAsStream 的参数不允许使用绝对路径，否则返回的 InputStream 为 null。

第八章 Enum



8.1 C 语言中的 enum 数据类型

在 C 语言中，enum 数据类型代表了一系列预定义的常量及其序号，比如一周的定义：

```
1 enum WEEK{SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY};
```

这相当于定义了一种新的数据类型“enum WEEK”，于是我们可以在程序中这样定义变量：

```
1 enum WEEK monday = MONDAY;
2 enum WEEK tuesday;
```

monday 和 tuesday 变量的数据类型是 enum WEEK 类型的，而且 monday 和 tuesday 的数据范围限于 enum week 中定义的常量。monday 的实际数值是整数 1。

8.2 Java 的 enum: Enum 的子类

Java 中枚举类型的定义和 c 语言非常相似，但是要灵活的多。同样一周的定义，我们看在 Java 中是如何定义和使用的。

代码清单 8.1: WeekDay.java

```

1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-20.
5 */
6 public enum WeekDay {
7     SUNDAY,MONDAY,TUESDAY,WENDESDAY,THURSDAY,FRIDAY,SATURDAY;
8 }
```

我们写一个测试类：

代码清单 8.2: WeekDayTest.java

```

1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-20.
5 */
6 public class WeekDayTest {
7     public static void main(String[] args) {
8         WeekDay sunday = WeekDay.SUNDAY;
9
10        System.out.println("sunday is " + sunday);
11        System.out.println("sunday'name is " + sunday.name());
12        System.out.println("sunday'ordinal is " + sunday.ordinal());
13
14        for(WeekDay day : WeekDay.values()) {
15            System.out.println("day[" + day.ordinal() + "] is " + day);
16        }
17    }
18 }
```

可以看出，WeekDay 的用法和类是一样的：根据 WeekDay 创建了一个对象 sunday，可以调用 sunday 对象的方法 name() 或者 ordinal()、values() 等等。

实际上，每一个 enum 类型的对象都自动继承自 Enum 类¹，也就是说，看起来我们是通过关键字 enum 定义了一个新的数据类型 enum WeekDay，其实 Java 会自动的将 enum WeekDay 扩展为一个 Enum 的子类，这就是为什么我们在定义 enum WeekDay 的时候并没有发现 name()、ordinal()、values() 方法，但是却可以调用的原因。

我们可以尝试反编译²一下 WeekDay.class 文件，看一下 enum WeekDay 经过 Java 编译后生成的类：

¹ 参见：<http://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html>

² javap 是 JDK 自带的反编译工具，可以将 class 文件还原为 java 源文件。

```
$ javap WeekDay.class
Compiled from "WeekDay.java"
public final class cn.edu.sdut.softlab.WeekDay extends
java.lang.Enum<cn.edu.sdut.softlab.WeekDay> {
public static final cn.edu.sdut.softlab.WeekDay SUNDAY;
public static final cn.edu.sdut.softlab.WeekDay MONDAY;
public static final cn.edu.sdut.softlab.WeekDay TUESDAY;
public static final cn.edu.sdut.softlab.WeekDay WENDESDAY;
public static final cn.edu.sdut.softlab.WeekDay THURSDAY;
public static final cn.edu.sdut.softlab.WeekDay FRIDAY;
public static final cn.edu.sdut.softlab.WeekDay SATURDAY;
public static cn.edu.sdut.softlab.WeekDay[] values();
public static cn.edu.sdut.softlab.WeekDay valueOf(java.lang.String);
static {};
}
```

也就是说, enum 其实是 Java 提供给我们的“语法糖”,当我们使用 enum 定义一个枚举类型时,Java 在编译时会将枚举类型扩展为一个 Enum 的一个子类。注意到当我们定义一个普通的 Java 类时,编译器会自动从 Object 继承下来的,这也是 enum 枚举类型对象和普通对象的区别。

如果不使用 enum,在 Java 中通常如何表达一周 7 天呢?一般情况下,我们可以在接口中定义常量:

```
1 public interface WeekDay {
2     public static final String SUNDAY = "SUNDAY";
3     public static final String MONDAY = "MONDAY";
4     public static final String TUESDAY = "TUESDAYDAY";
5     public static final String WEDENSDAY = "WEDENSDAY";
6     public static final String THIRSDAY = "THIRSDAY";
7     public static final String FRIDAY = "FRIDAY";
8     public static final String SATURDAY = "SATURDAY";
9 }
```

然后在程序中可以这样引用这些常量:

```
1 String monday = WeekDay.MONDAY;
```

可以看出,使用 enum 表达像一周 7 天这样的常量更有优势:更简洁,并且可以轻松获取常量定义的序号。

8.3 enum 的两个重要属性

任何 enum 默认有两个重要属性（从 Enum 类继承下来的）：

- name：名字，即 enum 的字面意义，通过调用 name() 方法返回这个属性值，但是建议使用 toString() 方法。toString() 默认返回 name，但是我们可以通过覆盖 toString() 方法返回更有意义的 enum 数据描述。
- ordinal：序号，即 enum 在定义中的序号，默认从 0 开始依次递增，通过调用 ordinal() 方法返回这个属性值。通常我们不需要使用 ordinal 这个属性，毕竟我们定义 enum 的主要目的是表达一系列常量，常量的定义位置（序号）通常没有多大意义。

练习 8.1. 定义一个 enum，表达一年的四季。

练习 8.2. 定义一个 enum，表达三基色（红绿蓝）。

8.4 enum 的构造方法

enum 既然是一个 Enum 的子类，自然可以拥有自己的构造方法，比如我们可以这样定义 12 个月份：

代码清单 8.3: Month.java

```

1 package cn.edu.sdu.tsoftlab;
2
3 /**
4  * Created by subaochen on 16-12-22.
5 */
6 public enum Month {
7     JAN(31), FEB(28), MAR(31),
8
9     private int days;
10    Month(int days) {
11        this.days = days;
12    }
13
14    public int days() {
15        return this.days;
16    }
17 }
```

和代码清单 8.1 不同的是，每个 Month 类型的 enum 常量都是通过构造方法 Month(int days) 创建的，比如 JAN(31)。注意到，我们同时提供了 days() 方法以便返回当前 enum 对象的 days 属性值，但是没有提供 setDays() 方法设置 days 属性值：days

属性值只允许创建 enum 对象的时候设置，因此不提供 setDays 方法是合理的。代码清单8.4是 Month 类的一个示例应用。

代码清单 8.4: MonthTest.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-22.
5 */
6 public class MonthTest {
7     public static void main(String[] args) {
8         Month jan = Month.JAN;
9         System.out.println(Month.JAN.toString() + " days is:" + Month.JAN.days());
10    }
11 }
```

例 8.1. 设计一个 enum 表示交通信号灯，并说明交通信号灯的意义。

代码设计 enum 的定义参见代码清单8.5，enum 的使用参见代码清单8.6。

代码清单 8.5: TrafficSignal.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-22.
5 */
6 public enum TrafficSignal {
7     RED("stop"),GREEN("go"),YELLOW("ready");
8
9     private String value;
10    public String value() {
11        return this.value;
12    }
13
14    TrafficSignal(String value) {
15        this.value = value;
16    }
17 }
```

代码清单 8.6: TrafficSignalTest.java

```
1 package cn.edu.sdut.softlab;
2
3 /**
4  * Created by subaochen on 16-12-22.
5 */
6 public class TrafficSignalTest {
```

```

7  public static void main(String[] args) {
8      for(TrafficSignal ts : TrafficSignal.values()) {
9          System.out.println(ts.toString() + " means " + ts.value());
10     }
11 }
12 }
```

运行结果 在 Idea 运行 TrafficSignalTest 结果如下：

```

RED means stop
GREEN means go
YELLOW means ready
```

代码分析和说明 可以看出，enum 增加构造方法的目的是通过构造方法构建更加复杂的 enum 对象，为 enum 对象增加额外的属性（除 name 和 ordinal 之外）。本例为 TrafficSignal 增加了 value 属性以表明不同颜色交通信号灯所代表的意义。

从本例我们也可以看出，使用 enum 后，我们并不需要直接引用字符串 stop、go、ready，这样我们有机会在未来修改交通信号灯的意义，比如当我们将来将 YELLOW 的意义修改为“wait a minute”时，只需要修改 TrafficSignal 类的定义即可，其他引用 TrafficSignal 的类都不需要修改，也就是说，我们的代码可以很好的适用未来的变化：用尽量小的代价应对未来的需求变化，软件的可维护性就大大提高了。enum 是表达常量、范围确定的映射关系的利器！

练习 8.3. 补充完整代码清单 8.3，使之能够表达完整的 12 个月份。

练习 8.4. 定义一个 enum Planet 表示太阳系各行星的名称、质量和半径。

8.5 switch 中的 enum

Java 的 switch 语句比 C 的 switch 语句更强大（参见节 2.6.2 [在第 32 页]）。在 Java 中，switch 的参数不仅可以是整数，也可以是字符串和 enum 对象。比如我们可以这样使用 TrafficSignal：

代码清单 8.7: EnumInSwitchTest.java

```

1 package cn.edu.sdut.softlab;
2
3
4 import static cn.edu.sdut.softlab.TrafficSignal.GREEN;
5
6 /**
```

```
7 * Created by subaochen on 16-12-22.  
8 */  
9 public class EnumInSwitchTest {  
10     public static void main(String[] args) {  
11         TrafficSignal ts = GREEN;//❶  
12         switch(ts) {  
13             case GREEN: // ❷  
14                 System.out.println("green signal");  
15                 break;  
16             case RED:  
17                 System.out.println("red signal");  
18                 break;  
19             case YELLOW:  
20                 System.out.println("yellow signal");  
21                 break;  
22         }  
23     }  
24 } //
```

❶ 如果这里写为: `TrafficSignal ts = GREEN`, 则需要静态导入 `GREEN` 的定义。尝试修改一下看看

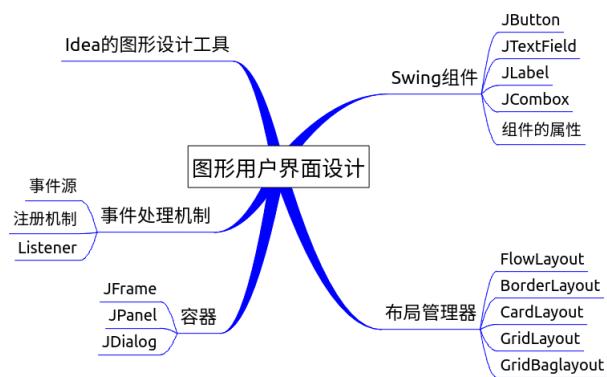
❷ 因为 `switch` 的参数是 `enum` 类型的, `java` 可以推断出这里的 `GREEN` 是 `TrafficSignal.GREEN`

关于 `enum` 的静态导入: 在实际使用 `enum` 对象的时候, 如果我们多次使用了同一个 `enum` 对象, 可以使用静态导入技术适当简化代码, 比如下面这样:

☒

```
1 import static cn.edu.sdut.softlab.TrafficSignal.GREEN;  
2 ...  
3 TrafficSignal ts = RED;  
4 ...
```

第九章 图形用户界面设计



9.1 Java 的图形用户界面 (GUI) 设计概述

Java 的图形用户界面 (GUI) 是由 JFC (Java Foundation Classes) 支撑的, JFC 包括了以下几个方面:

- **Swing GUI 组件**: 包含了图形用户界面设计中的人机交互组件, 从最简单的按钮到复杂的图表等无所不包, 这是我们学习的重点内容。
- 可插拔的 Look-and-Feel 支持: Java 图形用户界面支持可配置的 Look-and-Feel 切换, 可以很容易的实现不同操作系统下的界面统一。
- 辅助 API: 支持读屏幕等操作。
- Java 2D API: 完善的 2D 绘图 API。
- 国际化支持: 秉承 Java 一贯的国际化支持, 能够方便的在 GUI 中处理各种编码的文字。

Swing 是在早期的 Java GUI 应用程序设计接口 AWT 的基础上发展起来的, 目前是 Java GUI 应用程序设计的官方标准。除了 Swing 之外, Eclipse(www.eclipse.org) 组织也发布了一套 Java 图形用户界面的设计库 SWT(Standard Widget Toolkit),

 主要配合 Eclipse 的 RCP (Rich Client Program) 应用程序开发，详情可参考：<https://www.eclipse.org/swt/>

9.2 Swing 入门

本节以 JetBrains Idea 为例说明如何创建和运行简单的 Swing 应用程序。

例 9.1. 使用 Idea 创建一个简单的 Swing 应用程序，将人民币转换为美元。

步骤 1 创建一个新的 Java 项目，见图9.1。

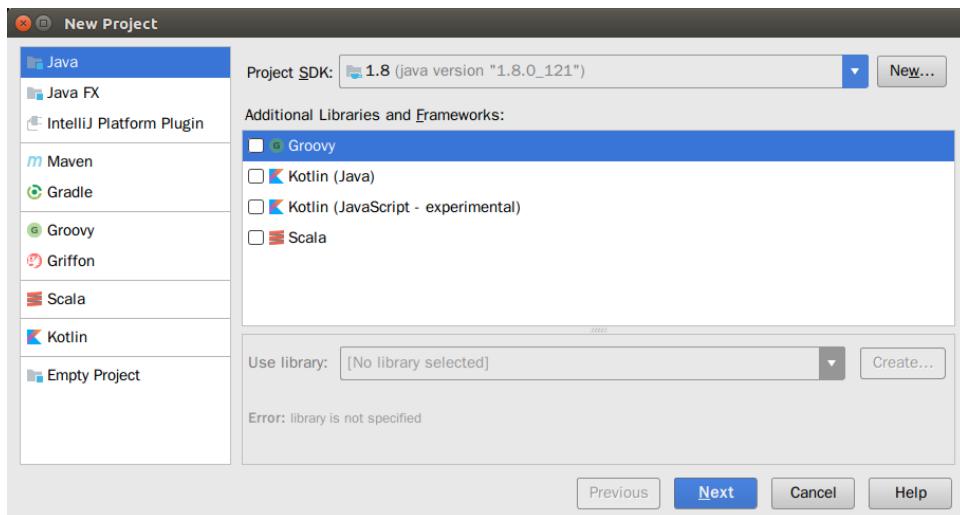


图 9.1：新建一个 Java 项目

步骤 2 在接下来的项目配置窗口中，可以不选择从模板创建项目，因为我们要创建一个 GUI 应用程序，命令行应用的模板没有实质用处，参见图9.2

步骤 3 给项目起个名字，这里叫做“gui”，并选择项目所在的目录，参见图9.3。

步骤 4 在项目的“project 视图”中，右键点击 src，在弹出的菜单中选择 New->GUI Form，新建一个 Form 表单，参见图9.4。

 如果在菜单中没有出现 GUI Form 子菜单，系没有安装 Form Designer 插件所致，请在系统设置中激活 Form Designer 即可，方法是在 File->Settings... 菜单中搜索“UI Designer”插件并安装激活，如图9.5所示。

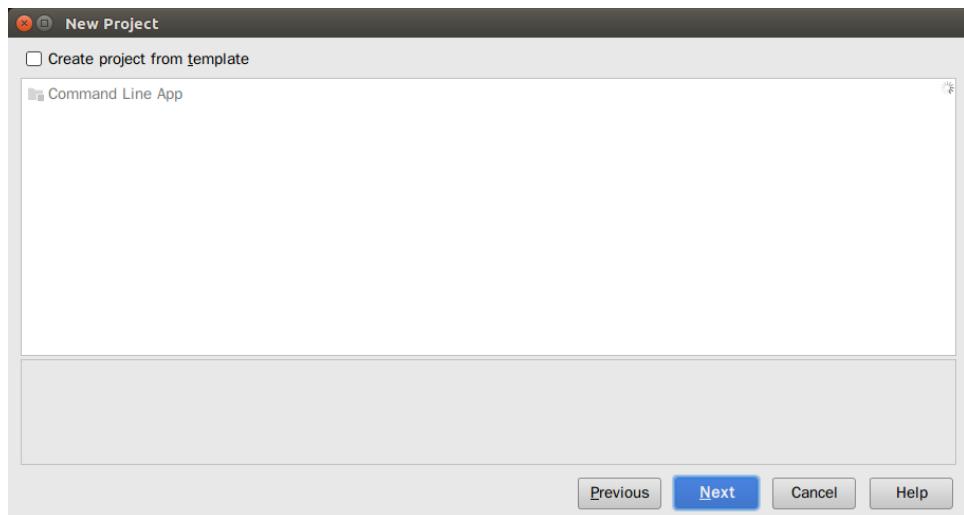


图 9.2: 跳过从模板创建项目

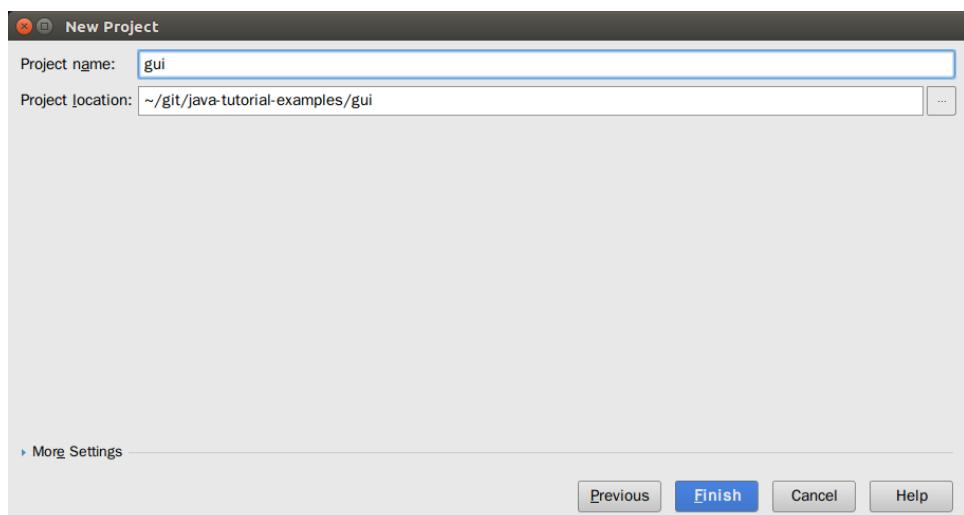


图 9.3: 给项目起个名字

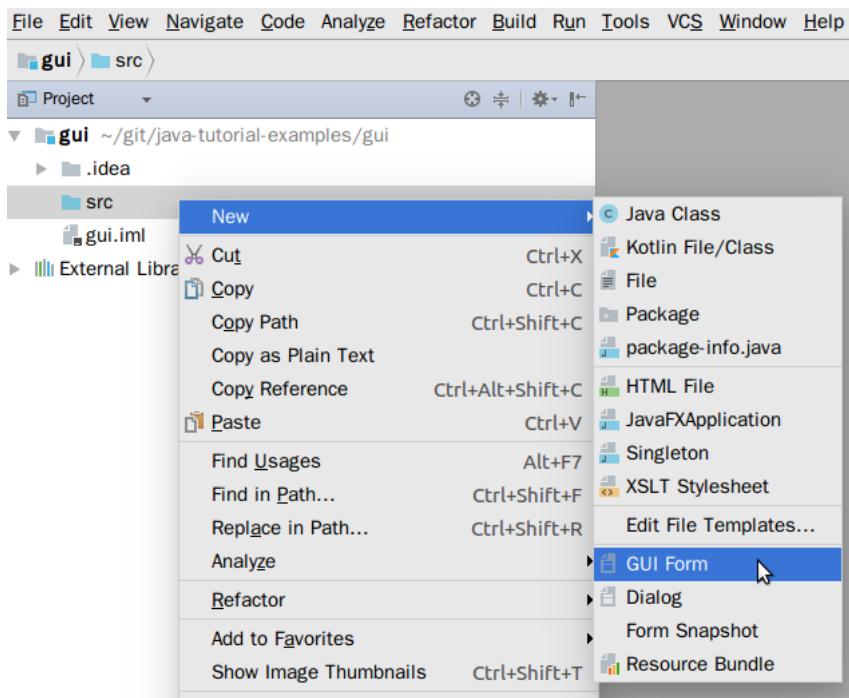


图 9.4: 准备新建一个 Form 表单

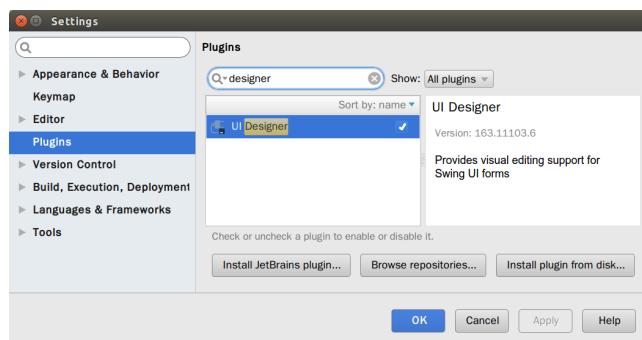


图 9.5: 安装并激活 UI Designer 插件

步骤 5 如图9.6所示,给将要创建的 Form 起个名字,这里叫做“RMB2DollarConverter”,选择布局管理器(此处先认可默认设置,随后可以更改)。注意要勾选(默认已经勾选了)“create bound class”选项,即同时创建和这个 Form 绑定在一起的 Java 类。这个 Java 类我们将来用于显示和操作这个 Form。

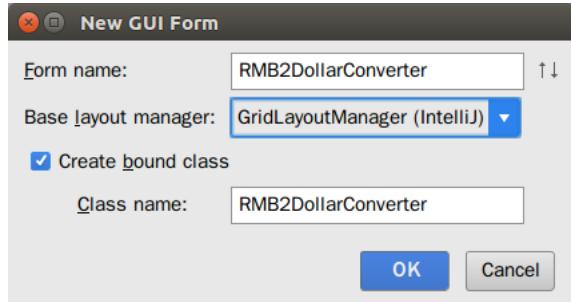


图 9.6: 给 Form 起个名字

步骤 6 如图9.7所示,在新建 Form 后会呈现一个空白的 Form 设计窗口,其主要组成部分为:

- 界面组件关系图:在这个窗口中展示了组件的“父子”关系,目前只有一个 JPanel 组件,随后我们会加入更多组件,可以通过这个窗口直观的查看组件之间的关系。也可以通过这个窗口快捷的选择某个组件进行操作。
- 组件属性窗:当在主设计界面或者界面组件关系窗口中选择了某个组件时,组件属性窗的内容将随之改变。可以通过这个窗口方便的了解界面组件拥有哪些属性,当然,更重要的是,可以设置界面组件的属性值。
- 主设计界面:可以拖放“组件面板”中的组件到主设计界面,主设计界面会根据使用的布局管理器的不同,自动摆放组件。
- 组件面板:列出了各种 Swing 界面组件,可以方便的拖放到主设计界面。

步骤 7 首先在“界面组件关系窗口”中选择目前唯一的组件 Panel,然后在组件属性窗口中编辑“field name”属性,设置其值为“mainPanel”。这里的值是什么并不重要的,重要的是必须给主 Panel 设置一个名字,否则将来无法显示这个 Panel。

在这里,我们也将 mainPanel 的“Layout Manager”(布局管理器)修改为更为简单的“FlowLayout”,我们将在节 9.5 [在第 269页]一节详细的阐述各种布局管理器的用法。

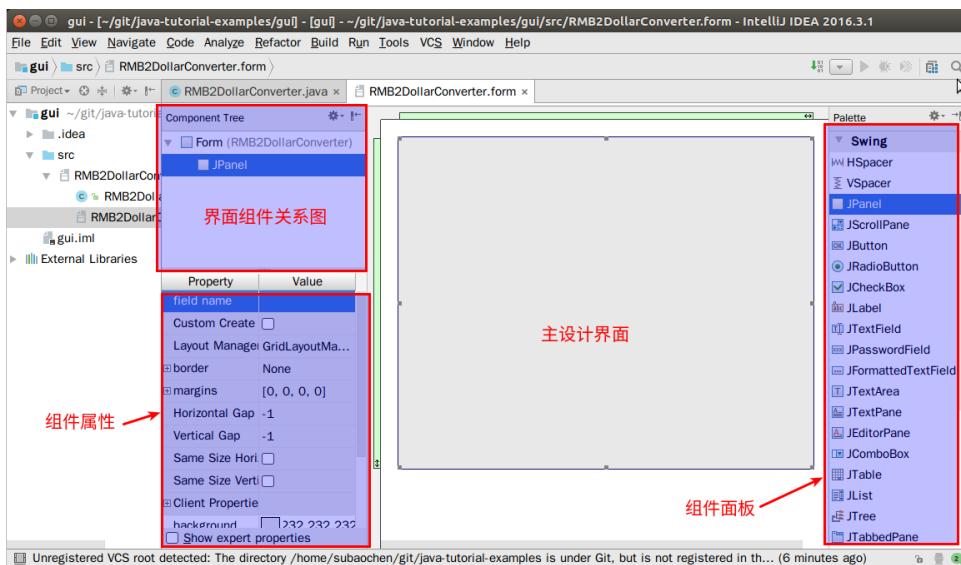


图 9.7: 空白的 Form 设计窗口

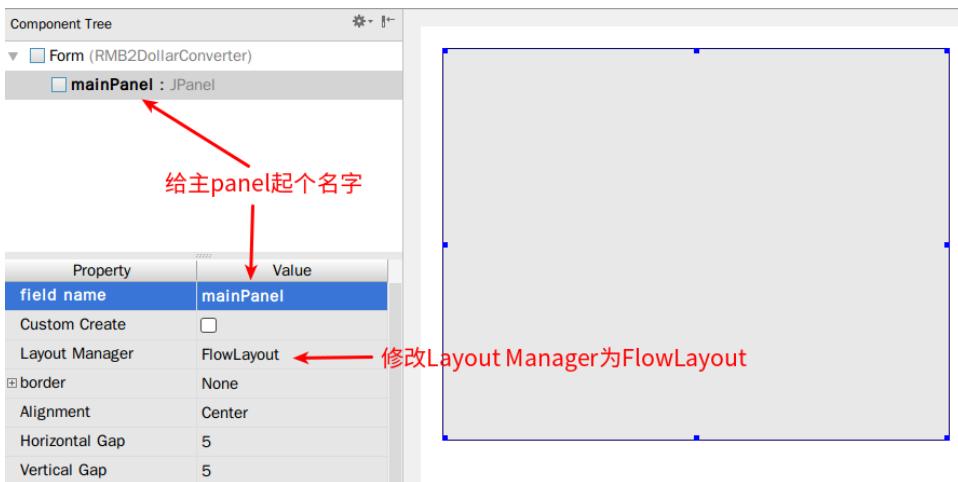


图 9.8: 给主 Panel 起个名字

步骤 8 在“组件面板”窗口选择 JLabel 组件拖放到主设计界面，可以看到在“界面组件关系”窗口中，JLabel 包含在 mainPanel 中。这里只是简单的设置这个 JLabel 的 text 属性为“人民币：”即可。JLabel 的目的是显示一个标签，我们将在 section 9.4.1 一节详细讨论 Jlabel 的用法。

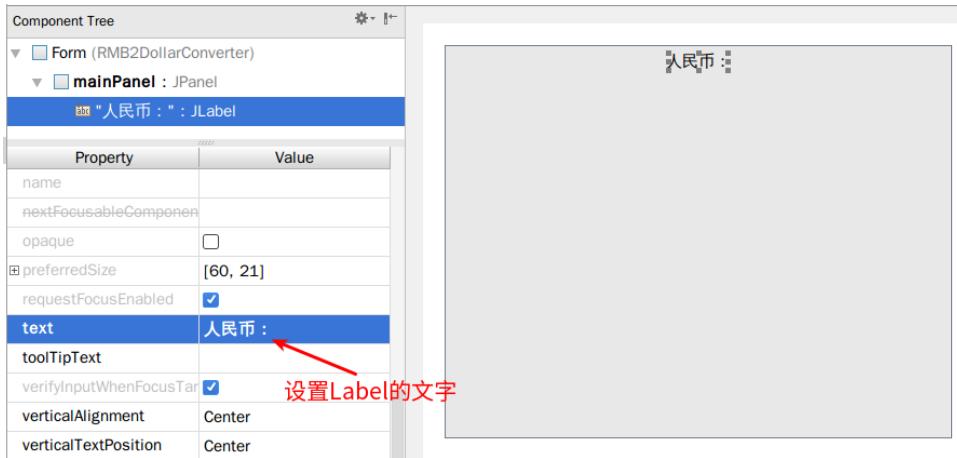


图 9.9: 添加一个简单的 Label

步骤 9 如图9.10所示，接着添加一个 JTextField 组件到主设计界面。这个 JTextField 我们用来输入人民币数额，因此要设置其“field name”属性。为了更好的显示这个输入框，也设置了这个 JTextField 的默认宽度（preferedSize->width）和默认显示的文字（text 属性）。

步骤 10 如图9.11所示，再增加一个“转换”按钮，设置这个按钮的 field name 和 text 属性。

步骤 11 如图9.12所示，在界面中再添加一个用于显示转换结果的 JLabel。由于我们要通过程序设置这个 JLabel 的 text 属性，因此要设置这个 JLabel 的 field name 属性。为了能够更好的显示转换后的结果，也有设置这个 JLabel 的宽度，这里设置为 100（像素值）。

步骤 12 界面的设计工作基本就绪，下面我们为按钮设计处理代码。如图9.13所示，在主设计界面的“转换”按钮上点击右键，在弹出的菜单中选择“create Listener”，准备创建一段点击此按钮的响应代码。

步骤 13 如图9.14所示，在接下来的窗口中选择 Listener 的类型为“ActionListener”。

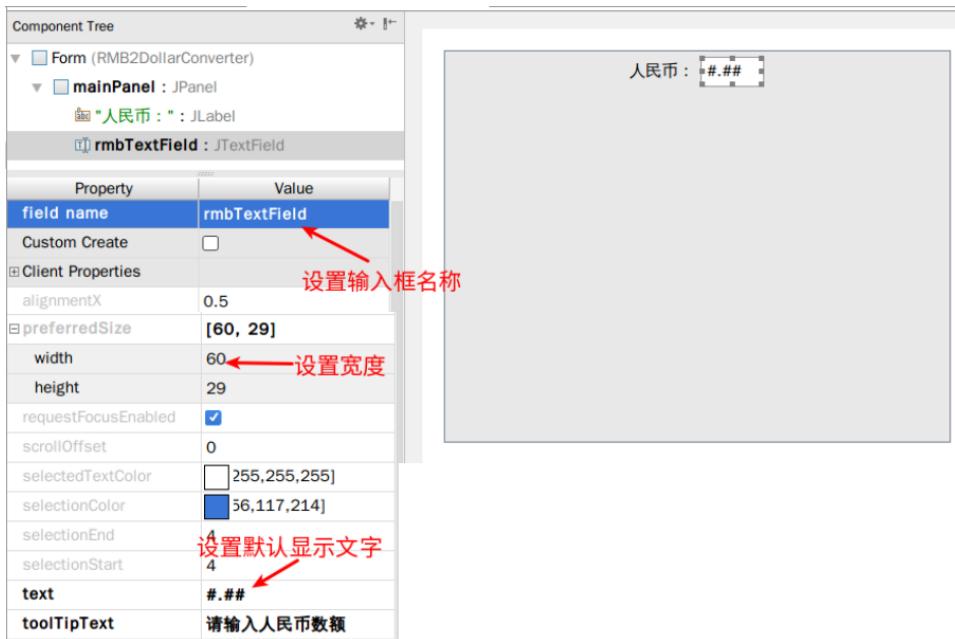


图 9.10: 添加 JTextField 组件

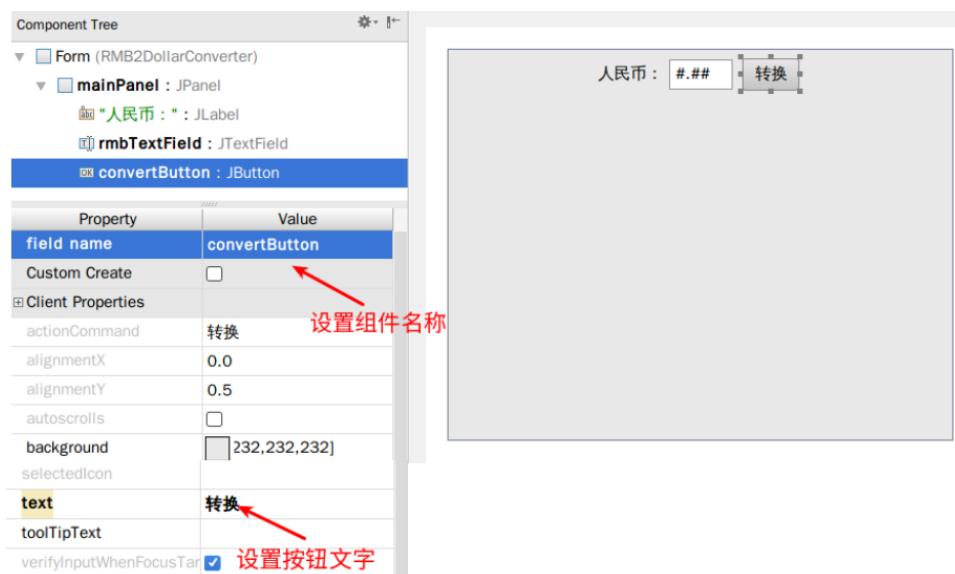


图 9.11: 添加转换按钮

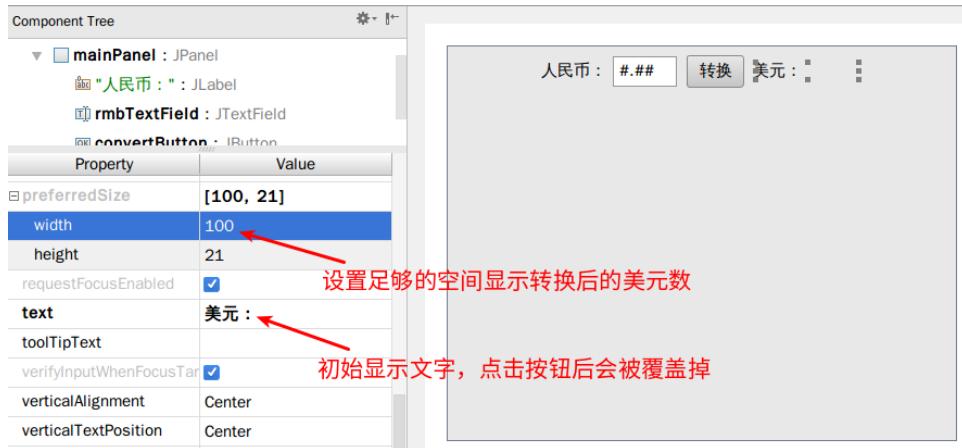


图 9.12: 添加显示转换结果的 JLabel

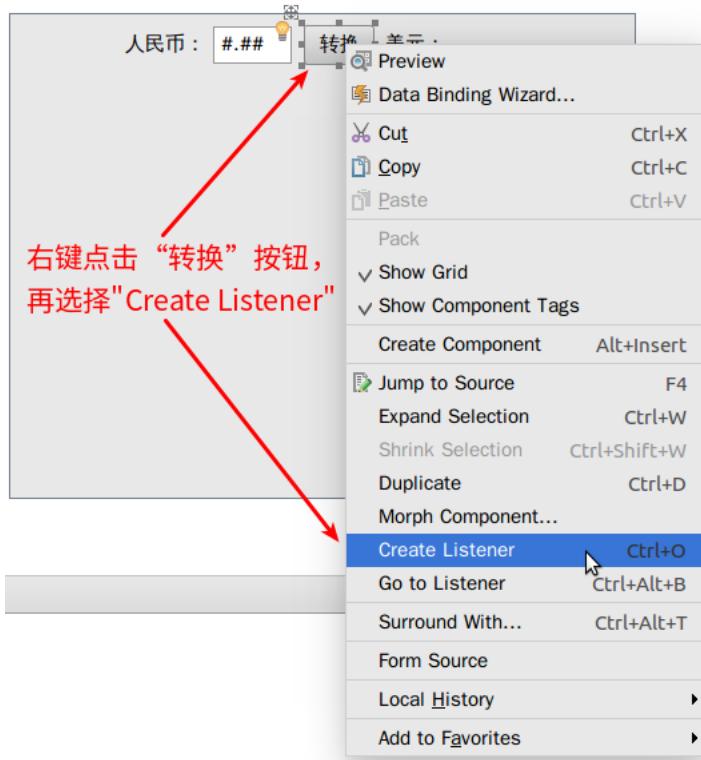


图 9.13: 准备创建 Listener

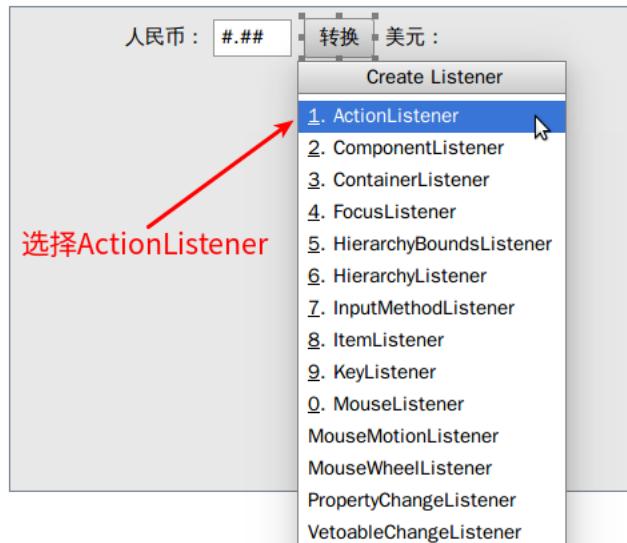


图 9.14: 选择 Listener 的类型

步骤 14 如图9.15所示，在接下来的窗口中，选择 ActionListener 需要覆盖的方法。这里只有一个选项，已经自动选中了。

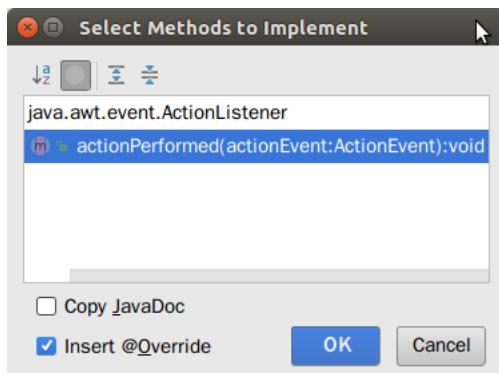


图 9.15: 选择需要覆盖的方法

点击图9.15中的“OK”按钮后，在所绑定的方法中自动添加了一个构造方法，内容如下（行 5-7 是自己编写的）：

```

1  public RMB2DollarConverter() {
2      convertButton.addActionListener(new ActionListener() {
3          @Override
4          public void actionPerformed(ActionEvent actionEvent) {
5              // 获取文本输入框的文字内容并转换为double
6              Double rmb = Double.valueOf(rmbTextField.getText());
7              dollarLabel.setText(String.valueOf(rmb / 7.0)); // 假设当前人民币和美元汇率为7
8          }
9      });

```

```

8         }
9     });
10    }

```

这是 GUI 应用程序事件处理的基本方法，我们将在 section §9.6一节中具体讨论。

步骤 15 最后一步，我们需要在类 RMB2DollarConverter 中增加一个 main 方法并初始化和显示应用程序窗口，JetBrain Idea 提供了自动化的代码生成工具，只需要在代码的合适位置（希望插入 main 方法的位置）按下 Alt+Insert 组合键，在如图9.16所示的窗口中选择“Form main()”即可自动创建 main 方法如下：

```

1  public static void main(String[] args) {
2      JFrame frame = new JFrame("RMB2DollarConverter");
3      frame.setContentPane(new RMB2DollarConverter().mainPanel);
4      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
5      frame.pack();
6      frame.setVisible(true);
7  }

```

```

public class RMB2DollarConverter {
    private JPanel mainPanel;
    private JTextField rmbTextField;
    private JButton convertButton;
    private JLabel dollarLabel;
}

```



图 9.16：准备插入 main 方法

运行结果 经过了以上的 15 个步骤，现在终于可以运行这个应用程序了¹。和运行其他 Java 应用程序的方法一样，运行 RMB2DollarConverter 类结果如图 figure 9.17 所

¹本例的完整代码请参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui>

示。可以在输入框输入人民币数额，然后点击“转换”按钮获得相应的美元数。



图 9.17: RMB2DollarConverter 的运行结果

你可能会问，在类 RMB2DollarConverter 中的几个私有属性（对象）：mainPanel、rmbTextField 等是什么时候初始化的？很好的问题！

实际上，Idea 通过一个 xml 文件 (RMB2DollarConvert.form) 保存了界面中所包含的组件及其属性和相互关系，在编译的时候，Idea 会自动读取这个界面配置文件并生成创建界面中的组件的构造方法，也就是说，mainPanel 等对象其实是 Idea 帮我们创建了的，我们可以直接拿来就用。

可以通过反编译 class 文件看到这一点，在构造方法中的 setupUI 方法创建了各个界面组件对象（部分删减，请自行运行 jad RMB2DollarConverter 获得完整的反编译后的文件）：

```
1 public class RMB2DollarConverter
2 {
3
4     public static void main(String args[])
5     {
6         JFrame frame = new JFrame("RMB2DollarConverter");
7         frame.setContentPane(new RMB2DollarConverter().mainPanel);
8         frame.setDefaultCloseOperation(3);
9         frame.pack();
10        frame.setVisible(true);
11    }
12
13    public RMB2DollarConverter()
14    {
15        setupUI();
16        convertButton.addActionListener(new ActionListener() {
17
18            public void actionPerformed(ActionEvent actionEvent)
19            {
20                Double rmb = Double.valueOf(rmbTextField.getText());
21                dollarLabel.setText(String.valueOf(rmb.doubleValue() / 7D));
22            }
23        });
24    }
25
26    private void setupUI()
27    {
28        JPanel jPanel;
```

```
29     JLabel jLabel;
30     JPanel jPanel = new JPanel();
31     mainPanel = jPanel;
32     jPanel.setLayout(new FlowLayout(1, 5, 5));
33     jLabel = new JLabel();
34     jLabel.setText("\u4EBA\u6C11\u5E01\uFF1A");
35     jPanel.add(jLabel);
36     JTextField jTextField;
37     jTextField = new JTextField();
38     rmbTextField = jTextField;
39     jTextField.setMinimumSize(new Dimension(14, 29));
40     jTextField.setPreferredSize(new Dimension(60, 29));
41     jTextField.setText("#.##");
42     jTextField.setToolTipText("\u8BF7\u8F93\u5165\u4EBA\u6C11\u5E01\u6570\u989D");
43     jPanel.add(jTextField);
44     JButton jButton;
45     jButton = new JButton();
46     convertButton = jButton;
47     jButton.setText("\u8F6C\u6362");
48     jPanel.add(jButton);
49     JLabel jLabel1;
50     jLabel1 = new JLabel();
51     dollarLabel = jLabel1;
52     jLabel1.setPreferredSize(new Dimension(100, 21));
53     jLabel1.setText("\u7F8E\u5143\uFF1A");
54     jPanel.add(jLabel1);
55 }
56
57 private JPanel mainPanel;
58 private JTextField rmbTextField;
59 private JButton convertButton;
60 private JLabel dollarLabel;
61
62
63 }
```

在 Idea 的 Form Designer 中，我们看到可以直接拖放各种控件到应用程序界面中，并且可以直观的通过“属性查看器”查看和修改控件的属性，其中一个重要的属性是 **field name** 属性，即属性的名称。什么时候需要给控件的 **field name** 属性赋值，什么时候不需要理会 **field name** 属性呢？简单的说，如果我们在程序中需要获得这个控件的其他属性，比如输入的文字，或者需要在程序运行期间通过程序改变这个控件的状态，则需要给这个控件的 **field name** 赋值。比如常见的 **JLabel** 控件一般是不需要 **field name** 属性的，而 **JTextField** 则需要 **field name** 属性。

 进一步的观察我们可以发现，只有给控件一个 field name 属性值，Idea 才能在绑定的类中自动创建私有的属性对象（变量）表示这个控件，变量的名字就是 field name 属性的值。

9.3 GUI 的顶级容器类

Java 的图形用户界面也是“面向对象”的，比如图9.18是常见的一个窗口应用程序，我们从 Java 的观点来看，可以分为如下的几部分：

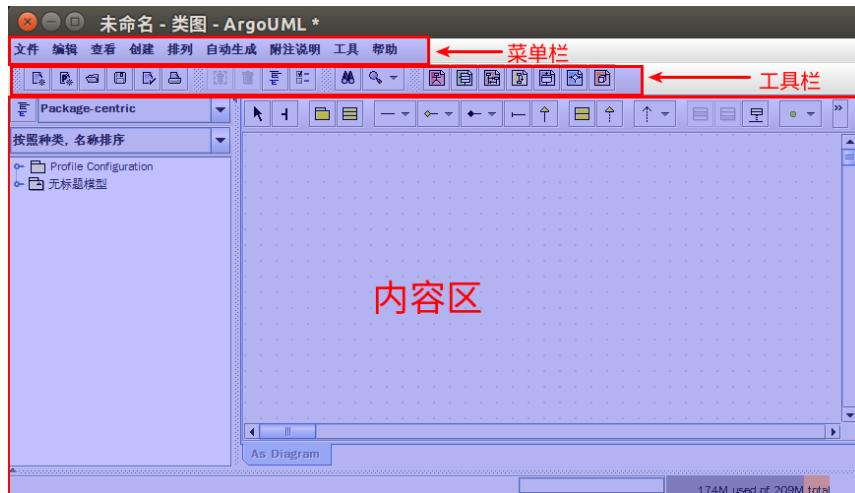


图 9.18: Java 图形用户界面应用程序的大致布局

整个应用程序框架使用 `JFrame` 来表示，在 `JFrame` 内部布置了菜单栏(`JMenuBar`)、工具栏(`JToolBar`)、内容区(`ContentPane`)，我们最经常操作的区域是 `ContentPane`，即几乎所有的界面设计工作集中在 `ContentPane` 上面。

所以，`JFrame` 是一个顶层的容器，其他所有的界面组件都放置在顶层容器之中，如图9.19所示。除了 `JFrame` 之外，Java 也提供了 `JDialog`、`JApplet` 顶层容器，分别用于对话框的设计和 Applet²的设计。本章重点介绍基于 `JFrame` 的 Java 图形用户界面设计，`JDialog` 的用法于此类似。

下面的代码展示了图9.19中的组件间的关系：

```

1 JFrame frame = new JFrame("标题栏内容");
2 frame.getContentPane().add(new JLabel("a label", BorderLayout.TOP));
3 frame.getContentPane().add(new JTextField("##.##", BorderLayout.BOTTOM));
4 JPanel panel = new JPanel();
5 panel.add(new JLabel("another label"));
6 panel.add(new JCheckBox(...));
7 frame.add(panel, BorderLayout.CENTER);

```

²—一种已经没落的技术。

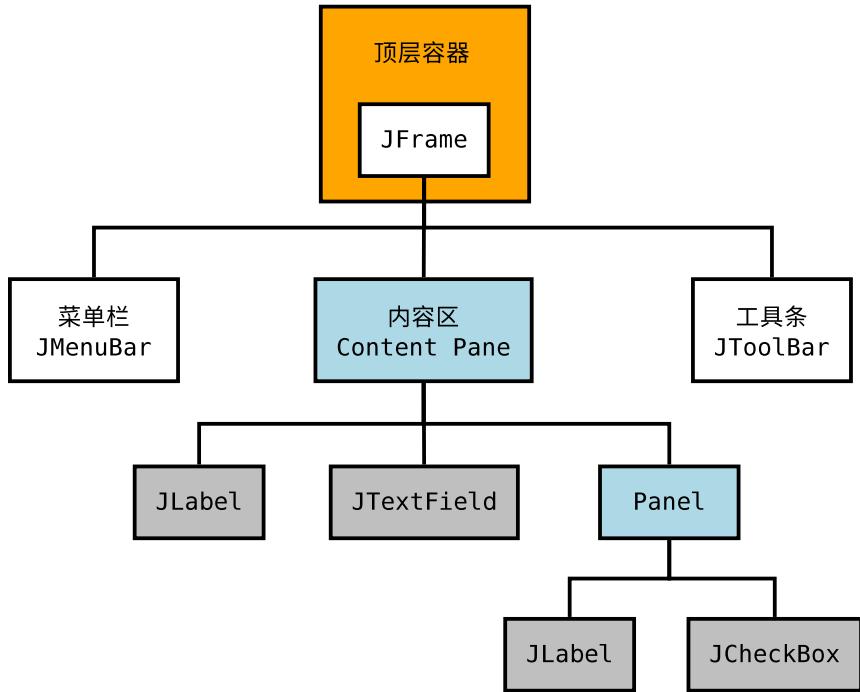


图 9.19: Java GUI 应用程序的基本结构

9.4 Swing 控件

9.4.1 JLabel

Label (文本标签) 也许是最简单的 Swing 控件了, 一般用来表示界面上的几个字符或者一段提示性文字、展示输出结果、图标等。

JLabel 的主要属性如表9.1所示 (每个属性对应一对 get/set 方法), 请参照 Form Designer 的属性对话框了解属性值的可能取值范围。

| 属性 | 说明 |
|------------------------|------------------------|
| icon | Label 所使用的图片 |
| text | Label 显示的文字 |
| horizontalAlignment | Label 的水平对齐方式 |
| verticalAlignment | Label 的纵向对齐方式 |
| horizontalTextPosition | Label 文字的水平位置 |
| verticalTextPosition | Label 文字的垂直位置 |
| textGap | Label 文字和图片之间的间隙 (像素值) |

表 9.1: JLabel 的主要属性

例 9.2. JLabel 示例

代码设计³ 我们在内容区增加了三个 JLabel：最上面的 Label 同时包含了文字和图片，中间的 Label 只包含了文字，最下面的 Label 只包含了图片，其属性对话框的设置分别如图9.20a图9.20b图9.20c所示，其中修改过的属性已经黑体加重标注了，没有使用黑体标注的属性无需设置。

当拖放第一个 JLabel 到内容区的时候，Form Designer 自动在这个 JLabel 的下面增加了一个垂直占位符，以保证这个 JLabel 能够在希望的顶部展示。由于我们还要增加下面的两个 Label，因此这里可以删除这个垂直占位符。

| Property | Value |
|-------------------------|-------------------------------------|
| field name | |
| Custom Create | <input type="checkbox"/> |
| Horizontal Size Policy | Fixed |
| Vertical Size Policy | Fixed |
| Horizontal Align | Center |
| Vertical Align | Center |
| Indent | 0 |
| Minimum Size | [-1, -1] |
| Preferred Size | [-1, -1] |
| Maximum Size | [-1, -1] |
| Client Properties | |
| background | <input type="color"/> 232,232,232 |
| enabled | <input checked="" type="checkbox"/> |
| font | <default> |
| foreground | <input type="color"/> 0,0,0 |
| horizontalAlignment | Trailing |
| horizontalTextPosition | Center |
| icon | |
| labelFor | |
| text | Image and Text |
| toolTipText | |
| verticalAlignment | Center |
| verticalTextPosition | Bottom |

| Property | Value |
|-------------------------|-------------------------------------|
| field name | |
| Custom Create | <input type="checkbox"/> |
| Horizontal Size Policy | Fixed |
| Vertical Size Policy | Fixed |
| Horizontal Align | Left |
| Vertical Align | Center |
| Indent | 0 |
| Minimum Size | [-1, -1] |
| Preferred Size | [-1, -1] |
| Maximum Size | [-1, -1] |
| Client Properties | |
| background | <input type="color"/> 232,232,232 |
| enabled | <input checked="" type="checkbox"/> |
| font | <default> |
| foreground | <input type="color"/> 0,0,0 |
| horizontalAlignment | Leading |
| horizontalTextPosition | Trailing |
| icon | |
| labelFor | |
| text | Text-only Label |
| toolTipText | |
| verticalAlignment | Center |
| verticalTextPosition | Center |

| Property | Value |
|-------------------------|-------------------------------------|
| field name | |
| Custom Create | <input type="checkbox"/> |
| Horizontal Size Policy | Fixed |
| Vertical Size Policy | Fixed |
| Horizontal Align | Center |
| Vertical Align | Center |
| Indent | 0 |
| Minimum Size | [-1, -1] |
| Preferred Size | [-1, -1] |
| Maximum Size | [-1, -1] |
| Client Properties | |
| background | <input type="color"/> 232,232,232 |
| enabled | <input checked="" type="checkbox"/> |
| font | <default> |
| foreground | <input type="color"/> 0,0,0 |
| horizontalAlignment | Leading |
| horizontalTextPosition | Trailing |
| icon | |
| labelFor | |
| text | |
| toolTipText | |
| verticalAlignment | Center |
| verticalTextPosition | Center |

(a) 顶部 Label 属性框设置

(b) 中间 Label 属性框设置

(c) 下面 Label 属性框设置

图 9.20: JLabel 的属性对话框

运行结果 运行此示例，初始状态如图9.21a所示，可以尝试缩放窗口，比如水平放大窗口可以更明显的看出效果，如图9.21b所示。请任意缩放窗口请注意观察这三个 Label 位置的变化，体会水平对齐、垂直对齐等的效果。

代码说明 注意以下几点：

- 这三个 JLabel 都无需设置 field name 属性值。

³ 完整的代码请参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab/>，双击 JLabelDemo.form 即可打开 Form Designer 查看界面设计效果。

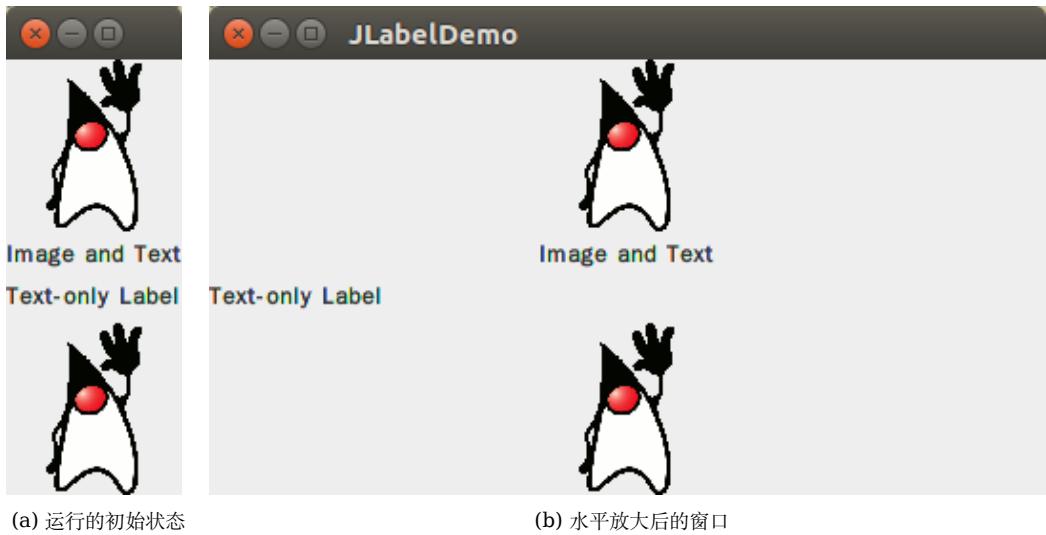


图 9.21: JLabelDemo 运行结果

- 特别注意观察 Horizontal Align 属性的不同效果，可以尝试不同的水平对齐方式观察效果（通过 Idea 的 Form Designer 提供的预览工具更方便，在 Form Designer 界面直接右键选择“preview”即可）。
- 在组合使用文本和图片时，还要注意 horizontalTextPosition 和 verticalTextPosition 这两个属性的效果。

9.4.2 文本控件

Swing 的文本控件是指能够输入文本的控件，比如文本框 (JTextField)、文本域 (JTextArea)、富文本编辑器 (JEditorPane) 等。Swing 的文本控件类的关系如图9.22所示。

- 单行纯文本控件：用于输入一行纯文本字符串。根据场合不同，可以选择使用 JTextField，输入简单的单行字符串；或者 JFormattedTextField，输入格式化的单行字符串，比如表示日期的“2017-3-12”等；或者 JPasswordField，输入密码（默认不显示输入的密码）。
- 多行纯文本控件：用于输入多行纯文本字符串。如果需要输入的字符串比较长，可以采用 JTextArea，会显示一个多元的文本编辑区域供输入。
- 多行富文本控件：用于输入带格式的文本字符串，通常用于输入 HTML 等格式的字符串。

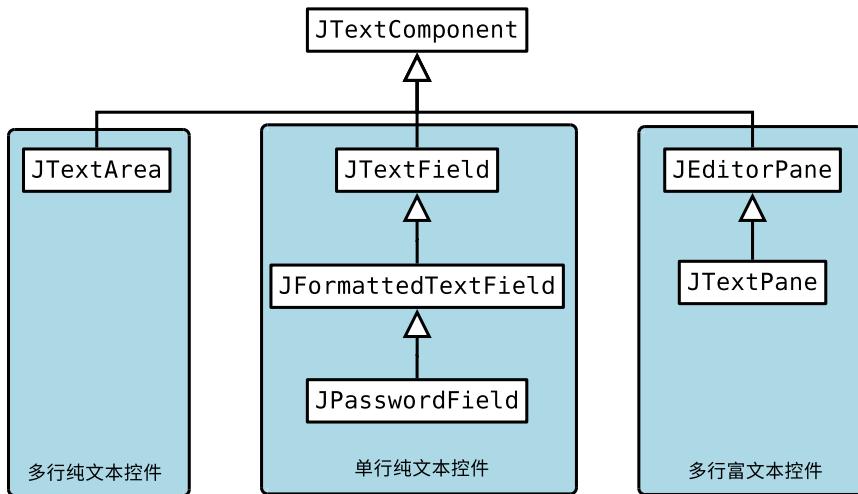


图 9.22: 文本输入控件

文本控件的常见属性如表9.2所示，请参照 Form Designer 中的属性对话框了解这些属性的取值范围。

| 属性 | 说明 |
|---------------------|---|
| text | 文本框的内容 |
| editable | 是否可编辑 |
| columns | 文本框的长度（多少个字符），仅仅作为计算默认显示宽度的依据 |
| horizontalAlignment | 字符在文本框内的对齐方式，可以选择： JTextField.LEADING, JTextField.CENTER, JTextField.TRAILING |

表 9.2: 文本控件的常见属性

例 9.3. 文本控件示例

代码设计 我们设计一个同时展示三种 TextField 的 Panel，如图9.23所示，详情请参见：<https://github.com/subaochen/java-tutorial-examples/blob/master/gui/src/cn/edu/sdut/softlab/TextControlDemo.form>。当在文本框中输入一些字符后（失去焦点后），在最下面的 Label 中显示输入的内容。

运行结果 运行此应用程序的界面如图9.24所示。

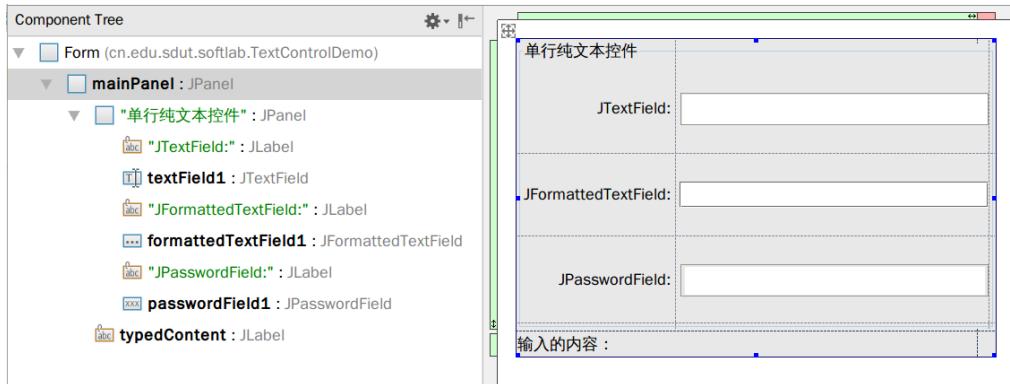


图 9.23: 文本控件示例



图 9.24: 文本控件示例运行结果

代码说明 这个简单的示例我们要注意两点：

- 可以通过 Panel 进一步组织控件，这是一种常见的手段。在例中，我们把三个 TextField 控件放到一个 Panel 中，在把这 Panel 添加到 mainPanel 中。并且，放置三个 TextField 的 Panel 添加了标题 (title)：“单行纯文本控件”，以更清晰的表达这个 Panel 的意义，这也是一种常见的手段。
- 请参照节 9.2 [在第 248页]为三个 TextField 添加 FocusListener，这里我们重点监听了失去焦点的事件。
- 本例也可以监听 ActionListener（当在输入框按下回车键时触发该事件）获得当前输入框的内容，请读者自行完成相关代码并运行测试。

焦点 (Focus) 通常是指文本输入框是否正在接受输入。获得焦点即文本输入框可以输入文字，失去焦点即文本输入框不再能够输入文字，即光标已经离开了此输入框。失去焦点往往意味着用户结束了输入，因此，我们可以通过监听失去焦点事件来获得文本输入框中的内容。

如果焦点的概念用于窗口（应用程序），则获得焦点意味着当前窗口是活动窗

 口，即可操作的窗口；失去焦点意味着其他窗口获得了焦点，当前窗口不可操作。

9.4.3 JButton

按钮 (Button) 是一种常见的交互控件，在 Swing 中最常见的按钮通过 JButton 来描述。按钮用法很直接，通常涉及到以下三个方面：

- 给按钮一个合适的名称，包括显示在按钮上面的文字。
- 有的时候希望在按钮上面也显示图标以更明确的表达按钮的意思，可以通过按钮的 icon 属性来设置图标。
- 给按钮添加鼠标点击的响应代码 (ActionListener 或者 MouseListener)。

JButton 的常见属性如表9.3所示。

| 属性 | 说明 |
|------------------------|-------------|
| enabled | 是否允许点击 |
| icon | 在按钮上面要显示的图标 |
| text | 在按钮上面要显示的文字 |
| horizontalTextPosition | 水平方向的文字位置 |
| verticalTextPosition | 垂直方向的文字位置 |
| horizontalAlignment | 水平方向的对齐方式 |
| verticalAlignment | 垂直方向的对齐方式 |

表 9.3: JButton 的常见属性

例 9.4. JButton 示例

代码设计 我们设计一个包含三个按钮的 Panel，如图9.25所示⁴。

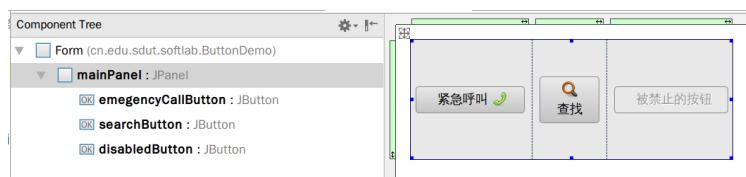


图 9.25: Button 示例 Panel

依然参照9.2为这三个 Button 添加三个 actionPerformed，分别响应鼠标左键单击事件。

⁴ 完整的代码请参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sduto/softlab>

运行结果 在运行时点击各个按钮，观察发生了什么？比如点击“紧急呼叫”按钮，弹出了一个消息框，如图9.26所示。点击“被禁止的按钮”，会怎样？



图 9.26: Button 示例运行结果

代码说明 请仔细查看三个按钮的属性对话框以了解这三个按钮是如何实现这样的效果的：

- 第一个按钮演示了按钮上是可以图文并茂的（设置 icon 属性即可），并且通过设置 horizontalTextPosition 属性为 Leading（在开头），将图片放到了文字的后面。
- 第二个按钮演示了如何将按钮上的文字和图片上下布局：首先让按钮上的内容水平居中（horizontalTextPosition 属性为 Center），然后设置 verticalTextPosition 属性为 Bottom 即可。
- 第三个按钮设置 enabled 为 false 即可，即去掉默认选中的 enabled 选项。

默认的，JButton 的 actionPerformed 是响应鼠标左键单击事件的，比如在

例 9.5. 9.4 中，我们给按钮增加了响应鼠标左键单击的 actionPerformed，那么如何响应鼠标右键、中键的单击事件呢？参见下面的代码：

```
1 // 响应右键单击事件
2 emergencyCallButton.addMouseListener(new MouseAdapter() {
3     @Override
4     public void mouseClicked(MouseEvent mouseEvent) {
5         super.mouseClicked(mouseEvent);
6         if(SwingUtilities.isRightMouseButton(mouseEvent) && mouseEvent.
7             getClickCount() == 1) {
8             JOptionPane.showMessageDialog(null, "右键单击急呼叫按钮");
9         }
10 });

```



9.4.4 JCheckBox

JCheckBox 通常叫做“复选框”，即可以同时打开或关闭多个选项⁵。当选项被选中时，该复选框的“选择”状态为 true，否则为 false。

JCheckBox 的常见属性如表9.4所示。

| 属性 | 说明 |
|------------------------|------------------|
| text | CheckBox 的提示文字 |
| icon | CheckBox 的提示图片 |
| enabled | 是否启用这个控件 |
| selected | CheckBox 默认的选中状态 |
| horizontalAlignment | 文字的水平对齐方式 |
| horizontalTextPosition | 文字的水平位置 |
| verticalAlignment | 文字的垂直对齐方式 |
| verticalTextPosition | 文字的垂直位置 |

表 9.4: JCheckBox 的常见属性

例 9.6. JCheckBox 示例

代码设计 我们设计一个包含 4 个 CheckBox（业余爱好）的 Panel⁶，当鼠标点击某个爱好时显示选中的 CheckBox 的状态；当鼠标移动到某个爱好上面时，同步显示表示这个业余爱好的图片，如图9.28所示。

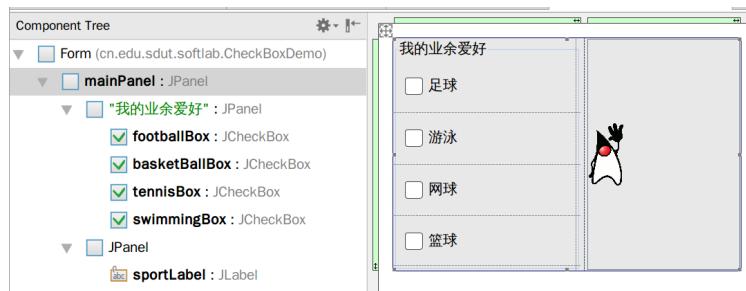


图 9.27: JCheckBox 示例设计

⁵对比一下 JRadioButton（单选框），JRadioButton 只能在一组选项中选中一项。

⁶完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab>

运行结果 在运行时，注意观察鼠标在各个 CheckBox 上面和离开 CheckBox 时界面的不同反应，选中或者反选选项时不同的消息提示，如图9.28所示。



图 9.28: JCheckBox 运行结果

代码说明 我们为每个 JCheckBox 添加了两个事件监听器，一个 ActionListener 用于监听鼠标左键单击选项选择或者反选时产生的事件，一个 MouseListener 用于监听当鼠标进入或者离开选项所在区域时产生的事件。

延伸阅读 其实，JButton, JCheckBox, JRadioButton 等都从 AbstractButton 继承下来的，即 JButton 一族的类层次关系如图9.29所示。

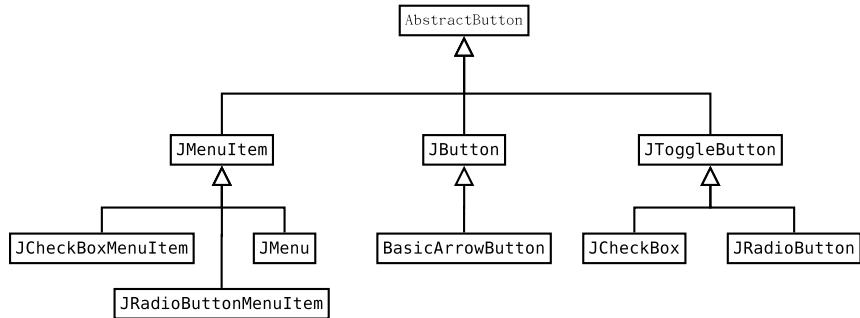


图 9.29: Button 类层次关系

因此，JButtron、JCheckBox、JRadionButton 等的共性就不难理解了。

9.4.5 JRadioButton

`JRadioButton` 通常叫做“单选框”，即通过 `JRadionButton` 可以展示一组选项，但是只能有一个选项处于“选中”状态。当选择另外一个选项时，原先被选中的选项自动失效。

JRadioButton 的常见属性和 JCheckBox 基本相同，只有一点需要注意：JRadioButton 的 buttonGroup 属性是必须设置的，即每个 JRadioButton 都需要设置 buttonGroup 属性，表明这个 JRadioButton 是属于哪个“组”的，以便 Java 根据用户的选择情况设置当前组的 RadioButton 哪个有效，哪个失效。

例 9.7. JRadioButton 示例

代码设计 类似于例9.6，我们设计一个图形用户应用程序⁷，根据选项决定显示哪种运动的图片，如图9.30所示。

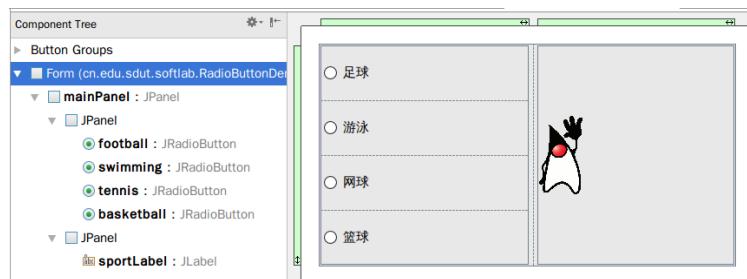


图 9.30: JRadioButton 示例设计

运行结果 点击不同的选项，将显示不同的运动图片，如图9.31所示。



图 9.31: JRadioButton 示例运行结果

代码说明 本例中，我们手工创建了 ActionListener，通常 JRadioButton 的 ActionListener 需要手工创建更合适，因为 JRadioButton 一般需要设置为属于某个 ButtonGroup，整个 ButtonGroup 设计一个 ActionListener 就可以了，没有必要每个 JRadioButton 都独立设计一个 ActionListener，如下列代码所示：

⁷完整的代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab>

```

1  public RadioButtonDemo() {
2      swimming.setActionCommand("swimming");
3      swimming.addActionListener(this);
4      tennis.setActionCommand("tennis");
5      tennis.addActionListener(this);
6      basketball.setActionCommand("basketball");
7      basketball.addActionListener(this);
8      football.setActionCommand("football");
9      football.addActionListener(this);
10 }
11
12 @Override
13 public void actionPerformed(ActionEvent actionEvent) {
14     sportLabel.setIcon(new ImageIcon(RadioButtonDemo.class.getResource("/images/" +
15         + actionEvent.getActionCommand() + ".png")));
16 }

```

在以上代码中，我们在构造方法中为每个 JRadioButton 指定了 actionCommand 以便在监听器代码中获取 actionCommand 拼装图片的路径。

另外需要注意的是，我们需要将这组 JRadioButton 设置一个共同的 ButtonGroup，即在 Form Designer 中如图9.32所示，创建一个新的 ButtonGroup，并将所有 JRadioButton 的 ButtonGroup 属性都设置为这个新建的 ButtonGroup。

9.4.6 JComboBox

JcomboBox 表示“下拉选择框”，通常在空间比较紧张或者选项很多时采用下拉选择框比较合适。

例 9.8. 下拉选择框示例

代码设计 和例9.6类似，我们设计一个下拉选择框来提供运动项目选项⁸，当选中某个运动项目时则显示相应的图片，如图9.33所示。

注意到，下拉选择框 sports 需要设置在其中要显示的条目，可以通过属性对话框的 model 属性来设置，如图9.34所示。

运行结果 该示例的运行结果如图9.35所示。

9.4.7 Spinner

Spinner 是一种便捷填写数字的控件，详情请参见示例代码：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/>

⁸完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab>

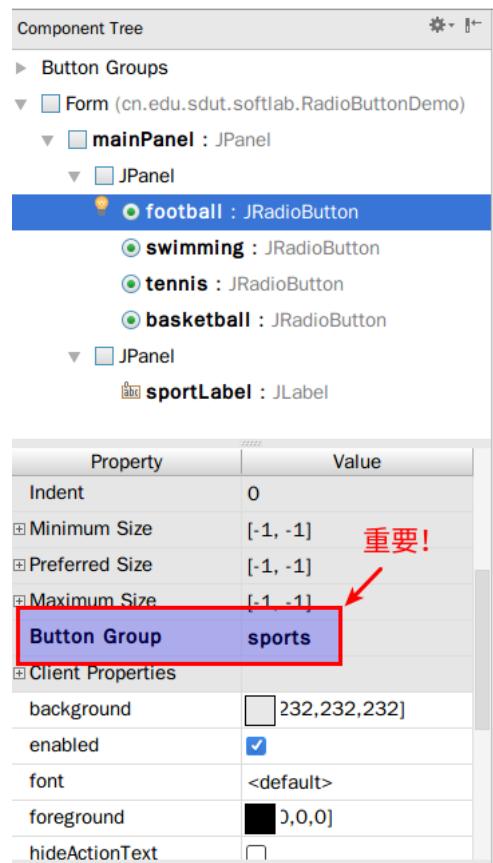


图 9.32: 设置 ButtonGroup

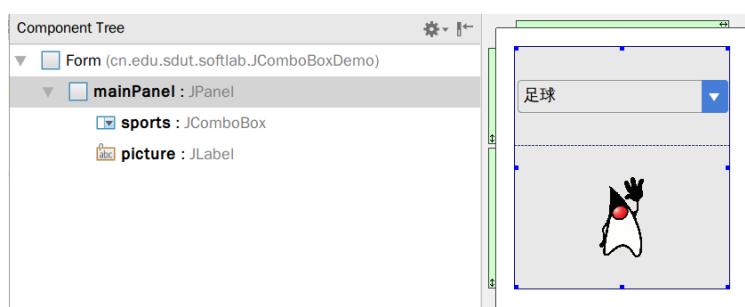


图 9.33: JComboBox 示例设计

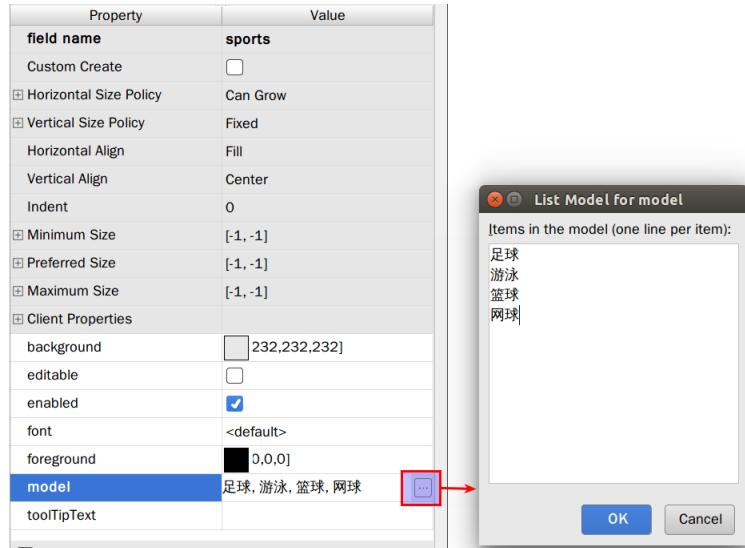


图 9.34: JComboBox 的 model 属性设置



图 9.35: JComboBox 示例的运行结果

sdut/softlab及其注释。

9.4.8 Slider

Slider 是一种通过滑动选择数字的控件, 详情请参见示例代码:<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab>及其注释。

9.4.9 Tabbed Pane

Tabbed Pane 可以实现“标签页”, 详情请参见示例代码:<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab>及其注释。

9.5 布局管理器

在图形用户界面应用程序的设计中, 界面中的控件如何布局总是一个容易让人困扰的问题, 尤其是当窗口大小发生改变的时候, 如何保证界面布局符合当初的设计(期望)呢? 通常有两种常见的界面布局思路: 绝对布局和相对布局。绝对布局是指使用绝对(像素)坐标确定控件在界面的位置, 很显然, 当窗口大小发生改变时, 绝对布局的界面不会随窗口发生改变, 于是导致了不能充分利用放大了的窗口或者无法适应缩小了的窗口, 因此绝对布局在实际的编程中很少用到, 也不建议使用绝对布局, 本节主要阐述各种相对布局的基本思路和方法。

9.5.1 BorderLayout

BorderLayout 很像“麻将桌”, 将界面分为 5 个部分:

- PAGE_START: 界面的顶部
- PAGE_END: 界面的底部
- LINE_START: 界面的左边
- LINE_END: 界面的右边
- CENTER: 界面的中间

BorderLayout 的特点是, 当窗口缩放时, 四周的控件只占用尽可能小的空间, 中间的控件将随窗口占用尽可能多的空间。

例 9.9. BorderLayout 示例

代码设计 如图9.36所示，5个按钮分别位于界面的4边和中间⁹。注意到我们将主Panel的Layout Manager修改为了BorderLayout，而不是默认的GridBagLayout。

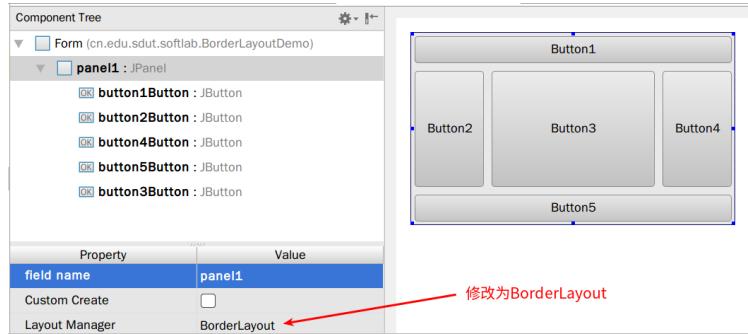


图 9.36: BorderLayout 示例

运行结果 图9.37所示的运行结果是放大窗口后的效果。建议读者在不同的窗口大小情况下观察 BorderLayout 的布局效果，以加深对 BorderLayout 的理解。



图 9.37: BorderLayout 示例运行结果

代码分析 这个示例很简单，我们只是修改了 Panel 的 Layout Manager 为 BorderLayout，然后将 5 个按钮依次放到界面的合适位置即可。

9.5.2 FlowLayout

流式布局（FlowLayout）是一种很自然的布局方式，即各个控件按照加入的顺序在窗口从左向右依次排开，各自只占用尽量小的空间。

例 9.10. FlowLayout 示例

⁹ 完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab>

代码设计 如图9.38所示，设计一个包含若干按钮和 RadioButton 的应用程序¹⁰，控件使用 FlowLayout 布局。

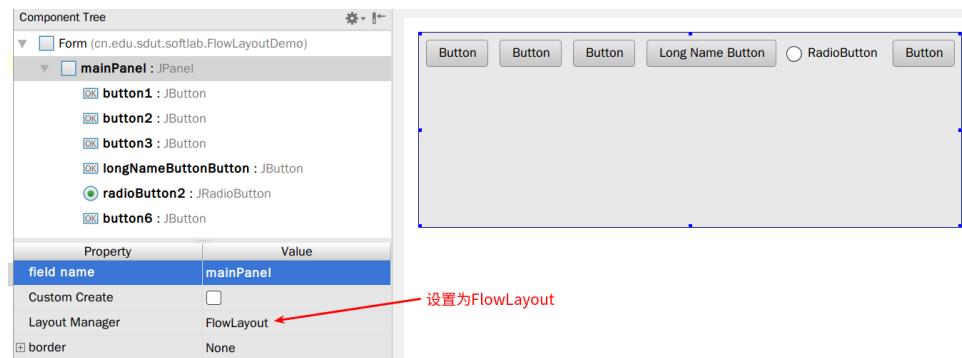


图 9.38: FlowLayout 示例设计

运行结果 默认的运行结果如图9.39所示。请尝试改变窗口的大小观察 FlowLayout 的布局效果。

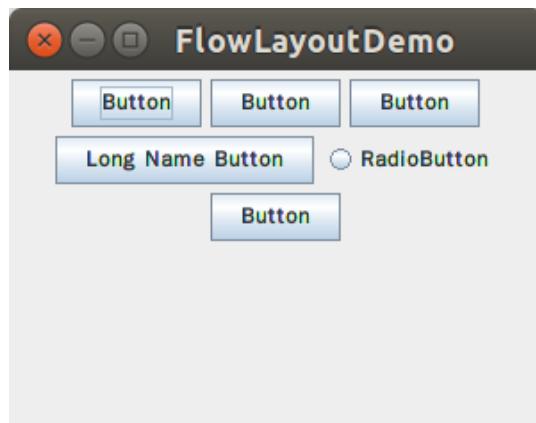


图 9.39: FlowLayout 示例运行结果

代码说明 本例重点有两个：

- mainPanel 采用了 FlowLayout 布局管理器。
- mainPanel 的 preferredSize 修改为 300x200，以便更好的展现 FlowLayout 的布局效果。

¹⁰完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sduto/softlab>

9.5.3 CardLayout

例 9.11. CardLayout 示例

如图9.40所示¹¹，主界面使用默认的 GridLayout，界面的上半部分放置了一个下拉选择框，根据不同的选择展示不同的 card。下半部分是一个 Panel，此 Panel (cardPanel) 使用 CardLayout，在其中添加了两个 card，分别是 buttonPanel 和 textFieldPanel。

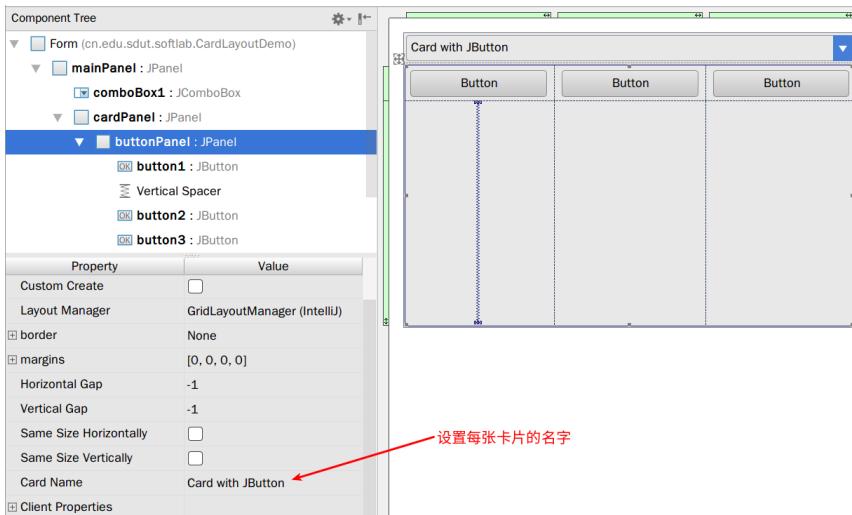


图 9.40: CardLayout 示例设计

运行结果 根据选择的不同，可以看到显示了不同 card 的内容，如图9.41所示。

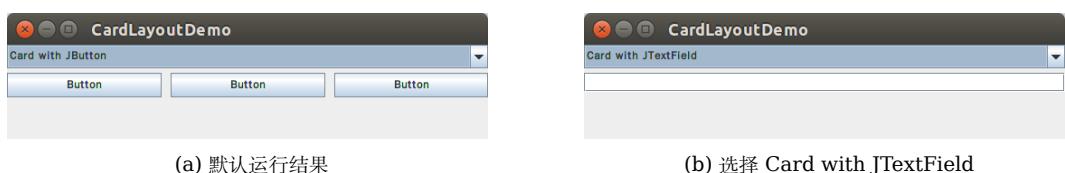


图 9.41: CardLayout 示例运行结果

代码说明 CardLayout 的 show 方法可以有选择的显示指定的 card，注意到 show 方法的第二个参数是指定的 card 的 name (字符串)，因此在 Form Designer 中要设置每个 card 的 name 属性。为了和下拉选择框的设置一致，在本例中 card 的 name 属性设置为了下拉选择框的相应文字。

¹¹ 完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab>

注意到 mainPanel 的布局管理器我们使用了 Idea 默认的 GridBLayout，可以尝试换为 BorderLayout 或者 FlowLayout 看看效果如何？

在本例中我们使用了一个常见的 Java GUI 界面设计技巧：通过 Panel 在局部组织控件，然后对 Panel 使用合适的布局管理器进行管理。也就是说，通常一个 Java GUI 的界面设计是分为两个层面的：第一个层面，对整个界面进行合理的大体的界面划分（使用 Panel）；第二个层面，在每个 Panel 内部进行控件的合理布局。我们在 Java GUI 的综合应用举例中还会看到这种布局策略。

9.5.4 GridLayout

网格布局（GridLayout）是将整个界面划分为表格，表格的每个单元格可以放置一个控件（或者 Panel），从而实现了规整的界面布局。影响 GridLayout 效果的除了表格单元格的个数之外，常见的属性如表9.5所示。

| 属性 | 说明 | 默认值 |
|------------------------|--|---------|
| horizontal gap | 单元格的水平间隔的像素值，-1 代表使用父容器的此设置，或者内置的 10px | -1 |
| vertical gap | 单元格的垂直间隔的像素值，-1 代表使用父容器的此设置，或者内置的 5px | -1 |
| same size horizontally | 如果是 true 的话，所有控件在水平方向大小一致 | false |
| same size vertically | 如果是 true 的话，所有控件在垂直方向大小一致 | false |
| margin | 容器和控件之间的四周间隔大小 | 0,0,0,0 |

表 9.5: GridLayout 的常见属性

例 9.12. GridLayout 示例

代码设计 使用 GridLayout 设计一个 3x2 的表格，其中放置 5 个按钮¹²，如图9.42所示。

运行结果 如图9.43所示，GridLayout 的用法简洁明了，请自行在 Form Designer 中修改相应的参数，比如 verticalGap（水平间隔），horizontalGap（垂直间隔）等观察这些参数对布局的影响。

¹² 完整代码参见：<https://github.com/subaochen/java-tutorial/tree/master/guide/code/gui/src/cn/edu/sdut/softlab>

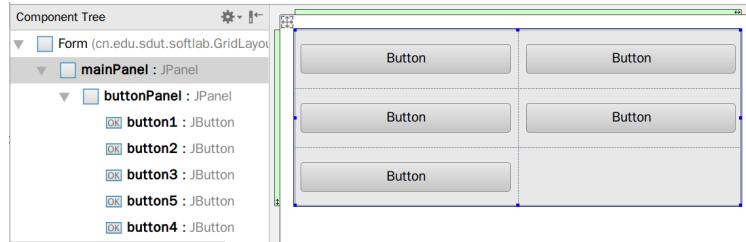


图 9.42: GridLayout 示例设计

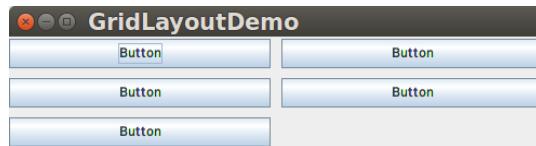
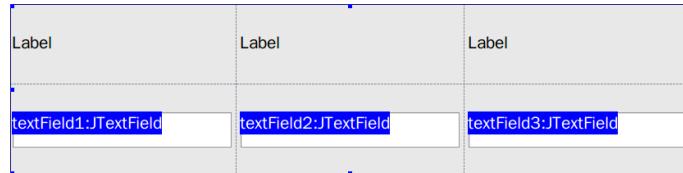


图 9.43: GridLayout 示例运行结果

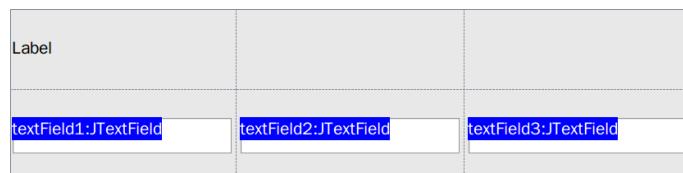
使用 Idea 设计 GridLayout 界面^a的小技巧: Idea 的 Form Designer 通常能够智能的判断应该使用多少格子来放置控件, 但是我们也必须给出明确的“指令”。比如我们希望设计如下的界面:



第一步, 先添加三个 JTextField, 然后在第一个 JTextField 上面放置一个 JLabel, 如下图所示:



第二步, 需要首先将三个 JTextField 调整到一个水平面上, 即如下图所示:



第三步，添加另外的两个 JLabel。

如果没有在第二步首先将三个 JTextField 调整到一个水平面上，则继续添加 JLabel 时 Form Designer 可能会错误的领会我们的意思，从而错误的计算单元格的数量，读者可自行尝试和认真体会。

¹³我们使用了 Idea 提供的 GridLayoutManager，详情请参考：<https://www.formdev.com/jformdesigner/doc/layouts/intellijgridlayout/>

9.5.5 GridBagLayout

网格袋布局（GridBagLayout）显然是 GridLayout 的扩展：GridLayout 只允许控件放置在一个单元格中，而 GridBagLayout 允许一个控件占用多个单元格，因而 GridBagLayout 更加灵活，控制选项（属性）也比较多，常见的属性如表9.6所示¹³。

| 属性 | 说明 | 默认值 |
|------------------|--|---------|
| grid x | 控件所占单元格的起始 x 坐标（左上角） | 0 |
| grid y | 控件所占单元格中的起始 y 坐标（左上角） | 0 |
| grid width | 控件横向占用的单元格数 | 1 |
| grid height | 控件纵向占用的单元格数 | 1 |
| horizontal align | 横向的对齐方式，Fill 意味着充满可能的横向空间（受 weight x 设置的影响） | Fill |
| vertical align | 纵向的对齐方式，Fill 意味着充满可能的纵向空间（受 weight y 设置的影响） | Center |
| weight x | 横向扩展的权重。如果值为 0 则横向不扩展，即控件只占用可能小的空间；通常此值在 0-1 之间设置。 | 0.0 |
| weight y | 纵向扩展的权重。如果值为 0 则纵向不扩展，即控件只占用可能小的空间；通常此值在 0-1 之间设置。 | 0.0 |
| insets | 控件的外边距 | 0,0,0,0 |
| ipad x | 横向的内边距素数，决定了控件距离单元格的边线有多远 | 0 |
| ipad y | 纵向的内边距素数，决定了控件距离单元格的边线有多远 | 0 |

表 9.6: GridBagLayout 的常见属性

¹³本书以 JetBrains Idea 提供的 Form Designer 为蓝本说明 GridBagLayout 的常见属性，和标准的 Java Swing 的 GridBagLayout 的属性略有出入。

在 Idea 的 Form Designer 中, grid x/grid y/grid width/grid height 属性是通过拖放控件自动设置的, 无需手工设置。实际上, Form Designer 的属性对话框没有提供编辑这 4 个属性的功能。

如果熟悉 CSS 的盒子模型的话, insets 相当于 CSS 盒子模型的 margin, ipadx/ipady 相当于 CSS 盒子模型的 padding。

例 9.13. GridBagLayout 示例

代码设计 如图9.44所示, 我们设计一个包含几个按钮的界面, 使用 GridBagLayout 布局管理器。

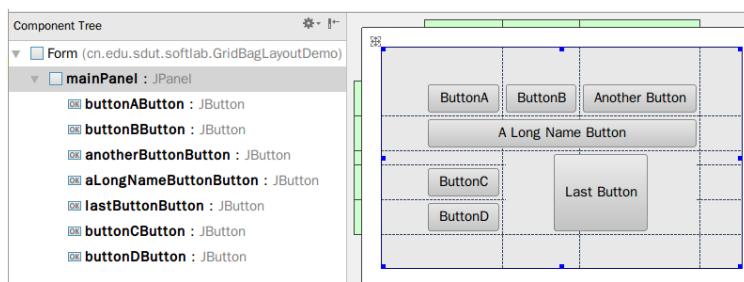


图 9.44: GridBagLayout 示例设计

运行结果 如图9.45所示。



图 9.45: GridBagLayout 示例运行结果

代码分析 本例中各控件的属性如表9.7所示。

请读者尝试表9.8的控件属性组合方式, 观察 GridBagLayout 在界面布局中的灵活性。

| 控件 | 属性设置 |
|--------------------|---|
| ButtonA | grid x = 0, grid y = 0 grid width = 1, grid height = 1 h align = Fill, v align = Center weight x = 0, weight y = 0 |
| ButtonB | grid x = 1, grid y = 0 grid width = 1, grid height = 1 h align = Fill, v align = Center weight x = 0, weight y = 0 |
| Another Button | grid x = 2, grid y = 0 grid width = 1, grid height = 1 h align = Fill, v align = Center weight x = 0, weight y = 0 |
| A Long Name Button | grid x = 0, grid y = 1 grid width = 3, grid height = 1 h align = Fill, v align = Center weight x = 0, weight y = 0 |
| ButtonC | grid x = 0, grid y = 2 grid width = 1, grid height = 1 h align = Fill, v align = Center weight x = 0, weight y = 0 |
| ButtonD | grid x = 0, grid y = 3 grid width = 1, grid height = 1 h align = Fill, v align = Center weight x = 0, weight y = 0 |
| Last Button | grid x = 1, grid y = 2 grid width = 2, grid height = 2 h align = Center, v align = Fill weight x = 0, weight y = 0 |

表 9.7: GridBagLayout 示例应用中的控件属性

| 测试场景 | 控件 | 属性设置 |
|------------------------------|-------------|--|
| 只设置 ButtonA 的属性，其他控件属性不变 | ButtonA | h align = Center, v align = Center weight x = 1, weight y = 0 |
| 只设置 Last Button 的属性，其他控件属性不变 | Last Button | h align = Center, v align = Fill weight x = 1, weight y = 0 |
| 只设置 Last Button 的属性，其他控件属性不变 | Last Button | h align = Fill, v align = Fill weight x = 1, weight y = 0 |
| 只设置 Last Button 的属性，其他控件属性不变 | Last Button | h align = Fill, v align = Fill weight x = 1, weight y = 1 |
| 只设置 Last Button 的属性，其他控件属性不变 | Last Button | h align = Fill, v align = Center weight x = 1, weight y = 1 |

表 9.8: 控件属性的各种组合方式

9.6 JAVA 图形用户界面的事件机制

9.6.1 事件机制的基本原理

事件机制是指代码如何对界面的点击、输入等事件做出响应，我们在前面的例子中已经多次使用了 Swing 的事件机制。Swing 事件机制的三个要点是：

- 事件源 (Event Source): 即触发事件的控件，比如按钮、文本框等。
- 事件对象 (Event Object): 描述事件的封装对象，其中包括了事件源、事件发生事件、事件相关参数（比如鼠标事件包括鼠标点击的坐标、鼠标点击次数等）等。
- 监听器 (Listener): 当事件发生时 Java 会自动调用的方法被称为监听器。

图9.46说明了按钮事件的处理过程，其他类型的事件处理过程与此类似。

常见控件的常见监听器见表9.9，完整的 Java GUI 控件和事件监听器的对照表参见 appendix C。

9.6.2 监听器类的几种情形

注意到注册监听器的方法：src.addListener(listenerObject) 的参数是一个实现了监听器接口的对象，即通过这个方法告诉 Java 虚拟机，当 src (事件源) 对象产生了一个 Xxx 类型的事件时，需要调用 listenerObject 中的相关方法进行事件处理。在实现监听器时，通常有如下的几种策略，以按钮的单击事件 actionEvent 为例：

- 主类直接实现 XxxListener 接口或者扩展 XxxHandler 类，比如：

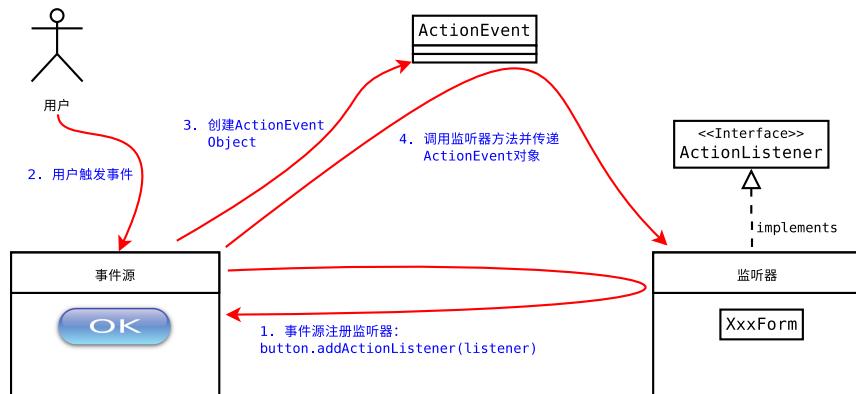


图 9.46: Swing 事件机制 (以按钮为例)

| 控件 | 常用监听器 |
|--------------|--|
| JButton | ActionListener: 响应鼠标左键单击和回车事件 MouseListener: 响应除左键单击外的其他的各种鼠标事件 |
| JTextField | ActionListener: 响应在输入框回车事件 FocusListener: 响应获得和失去焦点事件 |
| JRadioButton | ActionListener: 响应选项选择事件 MouseListener: 响应各种鼠标事件 |
| JCheckBox | actionListener: 响应选项选择事件 MouseListener: 响应各种鼠标事件 |
| JComboBox | actionListener: 响应选项选择事件 |

表 9.9: 控件的常见监听器

```

1 public class FrameDemo implements ActionListener {
2     private JButton button;
3     public FrameDemo () {
4         button = new JButton("OK");
5         button.addActionListener(this); // 注册事件监听器。由于FrameDemo本身实现
6             ActionListener接口，因此这里直接使用this作为实现了事件监听器接口的对象。
7     }
8     public void actionPerformed(ActionEvent event) {
9         // 事件处理代码
10    }
11 }
```

- 使用匿名内部类实现监听器接口，比如：

```

1 public class FrameDemo {
2     private JButton button;
3     public FrameDemo() {
4         button = new JButton("OK");
5         button.addActionListener(new ActionListener() { // 匿名内部类实现
6             ActionListener接口
7             @Override
8             public void actionPerformed(ActionEvent event) {
9                 // 事件处理代码
10            }
11        });
12    }
13 }
```

- 使用独立的类实现监听器接口，比如：

```

1 public class FrameDemo {
2     private JButton button;
3     public FrameDemo() {
4         button = new JButton("OK");
5         button.addActionListener(new ButtonListener());
6     }
7 }
8
9 // 独立的事件监听器类，实现了ActionListener接口
10 class ButtonListener implements ActionListener {
11     @Override
12     public void actionPerformed(ActionEvent event) {
13         // 事件处理代码
14     }
15 }
```

在大多数情况下，我们建议使用匿名内部类实现监听器接口，代码更简洁，封装性更好。

9.7 Java GUI 综合应用举例

例 9.14. 计算器

代码设计 这是一个典型的可以使用 `GridBagLayout` 进行界面设计的应用场合，如图9.47所示。为了更好的适应窗口大小，界面中的所有控件设置了如下属性：

- horizontal align: Fill
- vertical align: Fill
- weight x : 1.0
- weight y : 1.0

完整的代码参见代码清单9.1。

代码清单 9.1: Caculator.java

```
1 package cn.edu.sdu.tsoftlab;
2
3 import javax.script.ScriptEngine;
4 import javax.script.ScriptEngineManager;
5 import javax.script.ScriptException;
6 import javax.swing.*;
7 import java.awt.event.ActionEvent;
8 import java.awt.event.ActionListener;
9
10 /**
11 * Created by subaochen on 17-2-9.
12 */
13 public class Caculator implements ActionListener {
14     private JButton a7Button;
15     private JButton a8Button;
16     private JButton a9Button;
17     private JButton devideBtn;
18     private JTextField result;
19     private JButton a4Button;
20     private JButton a5Button;
21     private JButton a6Button;
22     private JButton multiplyBtn;
23     private JButton a1Button;
24     private JButton a2Button;
25     private JButton a3Button;
26     private JButton subtractBtn;
27     private JButton a0Button;
```

```
28  private JButton addBtn;
29  private JPanel mainPanel;
30  private JButton cButton;
31  private JButton delButton;
32  private JButton x2Button;
33  private JButton equalBtn;
34  private JButton pointBtn;
35  private JButton percentBtn;
36
37  private StringBuilder expression; // 需要计算的表达式
38
39  public Caculator() {
40      expression = new StringBuilder();
41      a0Button.addActionListener(this);
42      a1Button.addActionListener(this);
43      a2Button.addActionListener(this);
44      a3Button.addActionListener(this);
45      a4Button.addActionListener(this);
46      a5Button.addActionListener(this);
47      a6Button.addActionListener(this);
48      a7Button.addActionListener(this);
49      a8Button.addActionListener(this);
50      a9Button.addActionListener(this);
51      pointBtn.addActionListener(this);
52      percentBtn.addActionListener(this);
53      equalBtn.addActionListener(this);
54      addBtn.addActionListener(this);
55      subtractBtn.addActionListener(this);
56      multiplyBtn.addActionListener(this);
57      devideBtn.addActionListener(this);
58      delButton.addActionListener(this);
59      x2Button.addActionListener(this);
60      cButton.addActionListener(this);
61  }
62  @Override
63  public void actionPerformed(ActionEvent actionEvent) {
64      String cmd = actionEvent.getActionCommand();
65      switch (cmd) {
66          case "0":
67          case "1":
68          case "2":
69          case "3":
70          case "4":
71          case "5":
72          case "6":
73          case "7":
74          case "8":
75          case "9":
76          case "+":
77          case "-":
78          case "*":
79          case "/":
```

```
80         case ".":  
81         case "%":  
82             expression.append(cmd);  
83             break;  
84         case "=":  
85             // 计算结果  
86             expression.append("=" + caculate(expression));  
87             break;  
88         case "C":  
89             expression = new StringBuilder();  
90             break;  
91         case "x^2":  
92             expression.append("^2");  
93             break;  
94         case "del":  
95             expression.deleteCharAt(expression.length() - 1);  
96             break;  
97     }  
98  
99     result.setText(expression.toString());  
100 }  
101  
102 private String caculate(StringBuilder expression) {  
103     String result = "";  
104     ScriptEngine engine = new ScriptEngineManager().getEngineByName("JavaScript");  
105     try {  
106         System.out.println("expression=" + expression);  
107         result = engine.eval(expression.toString()).toString();  
108     } catch (ScriptException e) {  
109         e.printStackTrace();  
110     }  
111  
112     return result;  
113 }  
114  
115 public static void main(String[] args) {  
116     JFrame frame = new JFrame("Caculator");  
117     frame.setContentPane(new Caculator().mainPanel);  
118     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
119     frame.pack();  
120     frame.setVisible(true);  
121 }  
122  
123  
124 }
```

运行结果 如图9.48所示。

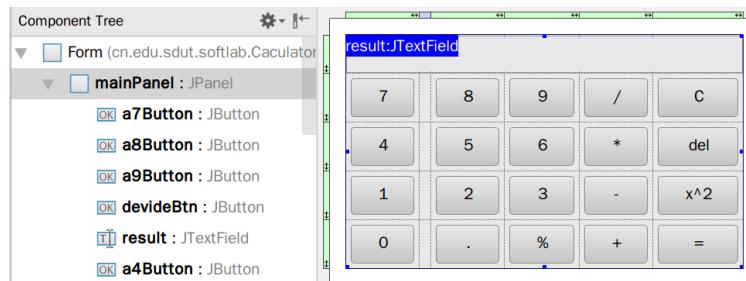


图 9.47: 计算器的界面设计

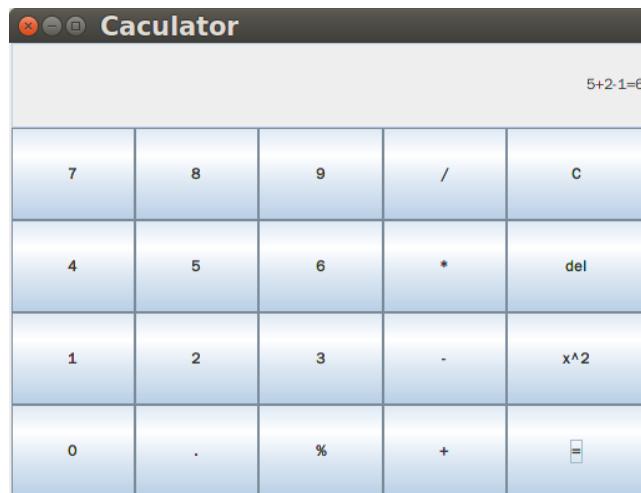


图 9.48: 计算器的运行结果

代码说明 本例为了简化起见，表达式的求值借用了 JDK 内置的 JavaScript 引擎，有兴趣的读者可以自行实现一个表达式求值的引擎或者方法。

练习 9.1. 请尝试使用 GridLayout 重新设计计算器（提示：首先使用 GridLayout 将界面分为上下两部分，上部分是一个 Panel 显示计算结果，使用任意的 Layout 即可；下部分是一个显示计算按钮的 Panel，使用 GridLayout 布局）。

第十章 实验

概述

程序设计是一门实践性很强的技能，Java 语言的程序设计自然也不例外。“纸上得来终觉浅，绝知此事要躬行”，学好 Java 程序设计语言的关键是多动手，重视实验，并认真思考实验中遇到的问题，不断加强对编程思想的认识。记住，**编程不难学，练着练着就会了！**

实验目的和任务

本课程主要给学生讲授 Java 语言的语法知识（包括类的知识点）、面向对象的程序设计思想与算法以及在工程实践中的常见做法，使学生对面向对象编程有有比较深的了解，并具备这方面的能力。Java 语言是完全面向对象的程序设计语言，内容比较抽象，须经过上机的实践，才能领会面向对象的程序设计思想。

教学目的主要是培养学生利用计算机处理问题的思维方式和程序设计的基本方法，启发学生主动将计算机引入到其它基础课和专业课。实验目的是为了提高同学们的动手实践能力，逐渐习惯 Java 面向对象的分析问题和描述问题的方式，掌握使用 Java 语言解决问题的能力。

本课程的实验任务是：

1. 掌握利用 JDK 工具开发一些简单 Java 应用程序的方法，本课程主要使用 JetBrain Idea，读者也可以自行学习 Eclipse、NetBeans 等 IDE。
2. 掌握 JAVA 语言的面向对象的概念及编程思想，在实验过程中不断加深对抽象、封装、多态等面向对象特点的认识。
3. 掌握常见的 Java API 的用法，养成 API 手册长置案头的习惯。
4. 了解 Java 编程的工程实践要求，比如编码规范、最佳实践等。

实验项目及学时分配

| 序号 | 项目名称 | 学时 | 实验要求 | 实验类型 | 仪器设备 |
|----|------------------|----|------|------|--------------|
| 1 | Java 开发环境和简单程序编写 | 2 | 必做 | 验证 | 计算机/SDK/Idea |
| 2 | 面向对象编程 | 2 | 必做 | 验证 | 计算机/SDK/Idea |
| 3 | Java 的异常处理 | 2 | 必做 | 验证 | 计算机/SDK/Idea |
| 4 | Java 的输入输出 | 2 | 必做 | 验证 | 计算机/SDK/Idea |
| 5 | 图形用户界面设计 | 2 | 必做 | 综合 | 计算机/SDK/Idea |
| 6 | 使用集合类 | 2 | 选做 | 综合 | 计算机/SDK/Idea |
| 7 | 并发程序设计 | 2 | 选做 | 综合 | 计算机/SDK/Idea |
| 8 | 网络编程 | 2 | 选做 | 综合 | 计算机/SDK/Idea |

实验报告和考核方式

实验报告 实验报告要求给出程序清单、运行结果、遇到的问题、调试心得。

考核方式 考核方式为操作技能考核、实验报告和面试相结合，实验成绩记入总成绩（占比 20%）。

实验一 Java 开发环境和简单程序编写

实验目的

- 掌握搭建 Java 开发环境的方法。
- 掌握 JetBrains Idea 的基本用法。
- 熟悉基本的 Java 程序设计结构。

实验内容

- 配置 JDK。
- 熟悉 JAVA 开发工具 JetBrains Idea, 编译调试 HelloWorld 程序。

实验要求

- 实验报告中给出配置 JDK 和 Idea 的步骤。
- HelloWorld 程序输出一条简单的问候信息。
- 实验报告中对程序结构做出详细的解释。

实验步骤

1. 下载 JDK 的最新版本并安装，必要的时候设置合适的环境变量。
2. 下载 JetBrains Idea 的最新版本并安装，了解 Idea 的基本用法。
3. 使用 Idea 编写第一个 Java 应用程序：HelloWorld。
4. 编写一个打印 10 行 “Hello, World” 的 Java 程序。参考代码：

```
1 public class HelloWorld {  
2     public static void main(String[] args) {  
3         for(int i = 0; i < 10; i++) {  
4             System.out.println("Hello, World!");  
5         }  
6     }  
7 }
```

5. 使用任意的纯文本编辑工具编写 HelloWorld，并在命令行编译和执行，了解脱离了 IDE 如何编写和运行 Java 应用程序。

实验二面向对象编程

实验目的

- 掌握面向对象的基本思想。
- 掌握 Java 类的概念和写法，掌握访问控制（封装）的技术。
- 掌握 Java 类继承的概念和用法。
- 掌握 Java 接口的概念和用法。

实验内容

- 编写 Java 类和抽象类。
- 编写基类的子类。
- 编写 Java 接口及其实现类。

实验要求

- 自行构思一个实验场景，在实验报告中画出所编写类的 UML 图及其层次关系。
- 给出所编写类的源代码。
- 说明 Java 定义类和接口的不同使用场合。

实验步骤

我们假设一个这样的场景，见图10.1。

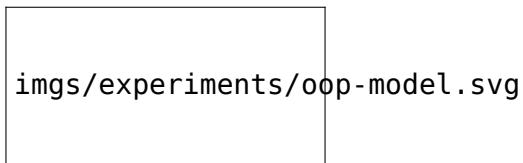


图 10.1: oop 编程实验场景类图

参考代码如下：

代码清单 10.1: Animal.java

```
1 package cn.edu.sdu.tsoftlab.oop;
2
3 /**
4  * Class Animal
5  */
6 abstract public class Animal {
7
8     protected String name;
9     protected float weight;
10    protected float height;
11    protected String category;
12
13    public Animal() {
14    }
15
16    ;
```

```
17
18
19 /**
20  * Set the value of name
21 *
22 * @param newVar the new value of name
23 */
24 protected void setName(String newVar) {
25     name = newVar;
26 }
27
28 /**
29  * Get the value of name
30 *
31 * @return the value of name
32 */
33 protected String getName() {
34     return name;
35 }
36
37 /**
38  * Set the value of weight
39 *
40 * @param newVar the new value of weight
41 */
42 protected void setWeight(float newVar) {
43     weight = newVar;
44 }
45
46 /**
47  * Get the value of weight
48 *
49 * @return the value of weight
50 */
51 protected float getWeight() {
52     return weight;
53 }
54
55 /**
56  * Set the value of height
57 *
58 * @param newVar the new value of height
59 */
60 protected void setHeight(float newVar) {
61     height = newVar;
62 }
63
64 /**
65  * Get the value of height
66 *
67 * @return the value of height
68 */
```

```

69     protected float getHeight() {
70         return height;
71     }
72
73     /**
74      * Set the value of category
75      *
76      * @param newVar the new value of category
77      */
78     protected void setCategory(String newVar) {
79         category = newVar;
80     }
81
82     /**
83      * Get the value of category
84      *
85      * @return the value of category
86      */
87     protected String getCategory() {
88         return category;
89     }
90
91     /**
92      * 奔跑
93      */
94     abstract public void run();
95
96
97     /**
98      * 问候
99      *
100     * @return String
101     */
102    abstract public String sayHello();
103
104
105 }
```

代码清单 10.2: Dog.java

```

1 package cn.edu.sdut.softlaboop;
2
3 /**
4  * Class Dog
5  */
6 public class Dog implements PeopleHelper, MouseKiller {
7
8     protected String color;
9
10    public Dog() {
11    }
```

```
12
13
14     /**
15      * Set the value of color
16      *
17      * @param newVar the new value of color
18      */
19     protected void setColor(String newVar) {
20         color = newVar;
21     }
22
23     /**
24      * Get the value of color
25      *
26      * @return the value of color
27      */
28     protected String getColor() {
29         return color;
30     }
31
32     /**
33      * 和人类一起玩耍
34      */
35     public void playWithPeople() {
36     }
37
38     /**
39      * 杀死耗子
40      */
41     public void killMouse() {
42     }
43
44     /**
45      * 打猎
46      */
47     protected void hunt() {
48     }
49 }
```

代码清单 10.3: Cat.java

```
1 package cn.edu.sduot.softlaboop;
2
3 /**
4  * Class Cat
5  */
6 public class Cat implements PeopleHelper, MouseKiller {
7
8     public Cat() {
9     }
10 }
```

```
11     ;
12
13     /**
14      * 如何玩耍？比如玩毛线球
15      */
16     public void playWithMouse() {
17     }
18
19
20     /**
21      * 和人类一起玩耍
22      */
23     public void playWithPeople() {
24     }
25
26
27     /**
28      * 杀死耗子
29      */
30     public void killMouse() {
31     }
32
33
34 }
```

代码清单 10.4: SheepDog.java

```
1 package cn.edu.sdu.tsoftlab.oop;
2
3 /**
4  * Class SheepDog
5  */
6 public class SheepDog implements PeopleHelper {
7
8     /**
9      * 主人
10     */
11    private String owner;
12
13    public SheepDog() {
14    }
15
16    /**
17     * Set the value of owner
18     * @param newVar the new value of owner
19     */
20    private void setOwner(String newVar) {
21        owner = newVar;
22    }
23
24    /**
```

```
25     * Get the value of owner
26     * @return the value of owner
27     */
28     private String getOwner() {
29         return owner;
30     }
31
32     /**
33     * 看护, 守望羊群
34     */
35     public void watch() {
36     }
37
38     /**
39     * 和人类一起玩耍
40     */
41     public void playWithPeople() {
42     }
43 }
```

代码清单 10.5: MouseKiller.java

```
1 package cn.edu.sdut.softlaboop;
2
3 /**
4  * Interface MouseKiller
5  */
6 public interface MouseKiller {
7
8     /**
9      * 杀死耗子
10     */
11     public void killMouse();
12
13
14 }
```

代码清单 10.6: PeopleHelper.java

```
1 package cn.edu.sdut.softlaboop;
2
3 /**
4  * Interface PeopleHelper
5  * 人类的朋友和助手
6  */
7 public interface PeopleHelper {
8
9     /**
10      * 和人类一起玩耍
11      */
12     public void playWithPeople();
```

```
13 }
```

请读者自行完善其中的方法，并设计一个测试类验证之。

问题 10.1. 如果将 Animal 或者 Dog 的某些属性修改为 private 会怎样？请实验证明。

实验三 Java 的异常处理

实验目的

- 掌握 Java 异常的基本处理思想，了解 Java 常见异常类型。
- 掌握 try/catch/throw/throws 的用法。
- 掌握自定义异常的方法。

实验内容

- 编写一个类，捕获数组越界异常。必须捕获这个异常吗？
- 编写一个自定义异常类。
- 编写一个测试类，抛出自定义异常对象。
- 编写一个测试类，捕获抛出的自定义异常对象。

实验要求

- 给出实验中编写的类的源代码。
- 说明 Java 异常处理的基本原则。

实验步骤

参考代码如下：

代码清单 10.7: MyException.java

```
1 package cn.edu.sduot.softlab.exception;
2
3 /**
4  * Created by subaochen on 17-2-20.
5  */
6 public class MyException extends Exception {
```

```
7     public MyException(String s) {  
8         super(s);  
9     }  
10 }
```

代码清单 10.8: Demo.java

```
1 package cn.edu.sdut.softlab.exception;  
2  
3 /**  
4  * Created by subaochen on 17-2-20.  
5 */  
6 public class Demo {  
7     public void test() throws MyException {  
8         if (true)  
9             throw new MyException("发生了异常, 请及时处理");  
10    }  
11 }
```

代码清单 10.9: Client.java

```
1 package cn.edu.sdut.softlab.exception;  
2  
3 /**  
4  * Created by subaochen on 17-2-20.  
5 */  
6 public class Client {  
7     public static void main(String[] args) {  
8         try {  
9             new Demo().test();  
10        } catch (MyException e) {  
11            e.printStackTrace();  
12        }  
13    }  
14 }
```

实验四 Java 的输入输出

实验目的

- 进一步熟悉 Java 的异常。
- 掌握常见的面向字节和面向字符的 Java IO 类。
- 熟练掌握从键盘输入的方法。

实验内容

- 编写一个程序，统计从键盘输入的字符的个数，按照单词、空白字符、数字分别输出。
- 编写一个程序，将给定的两个文件收尾连接后保存到新文件中，其中给定文件和新文件名从命令行输入。
- 编写一个程序，从键盘输入两个整数，打印出较大的那个。

实验要求

- 给出实验中编写的类的源代码。

实验步骤

参考程序：

代码清单 10.10: StandardIO.java

```

1 package cn.edu.sdut.softlab.io;
2
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6
7 /**
8  * 利用标准输入输出
9 */
10 public class StandardIO {
11     public static void main(String[] args) {
12         //IO操作必须捕获IO异常。
13         try {
14             //先使用System.in构造InputStreamReader, 再构造BufferedReader。
15             BufferedReader stdin =
16                 new BufferedReader(new InputStreamReader(System.in));
17             //读取并输出字符串。
18             System.out.print("Enter a string: ");
19             System.out.println(stdin.readLine());
20
21             //读取并输出整型数据。
22             System.out.print("Enter an integer: ");
23             //将字符串解析为带符号的十进制整数。
24             int number1 = Integer.parseInt(stdin.readLine());
25             System.out.println(number1);
26
27             //读取并输出double类型数据。
28             System.out.print("Enter a double: ");
29             //将字符串解析为带符号的double类型数据。

```

```
30     double number2 = Double.parseDouble(stdin.readLine());
31     System.out.println(number2);
32 } catch (IOException e) {
33     System.err.println("IOException");
34 }
35 }
36 }
```

代码清单 10.11: CatFile.java

```
1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.io.*;
4
5
6 /**
7 * 文件合并
8 */
9 public class CatFile {
10     public static void main(String[] args) {
11         if (args.length != 3) {
12             System.out.println("Usage: java CatFile src-file1 src-file2 src-file3 outFile
13                     ");
14             return;
15         }
16
17         String[] inFiles = new String[args.length - 1];
18         String outFile = args[args.length];
19         for (int i = 0; i < args.length - 1; i++) {
20             inFiles[i] = args[i];
21         }
22
23         try (OutputStream out = new FileOutputStream(outFile);
24 ) {
25             byte[] buf = new byte[1000];
26             for (String file : inFiles) {
27                 InputStream in = new FileInputStream(file);
28                 int b = 0;
29                 while ((b = in.read(buf)) >= 0) {
30                     out.write(buf, 0, b);
31                     out.flush();
32                 }
33             }
34         } catch (FileNotFoundException e) {
35             e.printStackTrace();
36         } catch (IOException e) {
37             e.printStackTrace();
38         }
39     }
40 }
```

41 }

代码清单 10.12: MaxMin.java

```

1 package cn.edu.sdu.tsoftlab.io;
2
3 import java.util.Scanner;
4
5 /**
6  * 输出较大的数
7 */
8 public class MaxMin {
9     public static void main(String[] args) {
10
11         // first value read initialized min and max
12         Scanner stdIn = new Scanner(System.in);
13         int max = stdIn.nextInt();
14         int min = max;
15
16         // read in the data, keep track of min and max
17         while (!stdIn.hasNext()) {
18             int value = stdIn.nextInt();
19             if (value > max) max = value;
20             if (value < min) min = value;
21         }
22
23         // output
24         System.out.println("maximum = " + max + ", minimum = " + min);
25     }
26 }
```

 注意 try-with-resources 结构的用法。如果不使用 try-with-resources 结构，打开的输入输出流需要通过 try-catch-finally 中的 finally 关闭，即在 finally 中调用 `in.close()` 或者 `out.close()`。

实验五图形用户界面设计

实验目的

- 了解 Java 图形用户界面设计的基本方法。
- 熟悉 Java Swing 的各种组件。
- 熟悉 Java 的常见布局管理器。
- 熟悉 Java GUI 的事件处理机制。

实验内容

使用 Idea 实现一个证券交易软件的登录界面（见图10.2），只要实现该界面本身即可，不要求实现真正的登录功能。

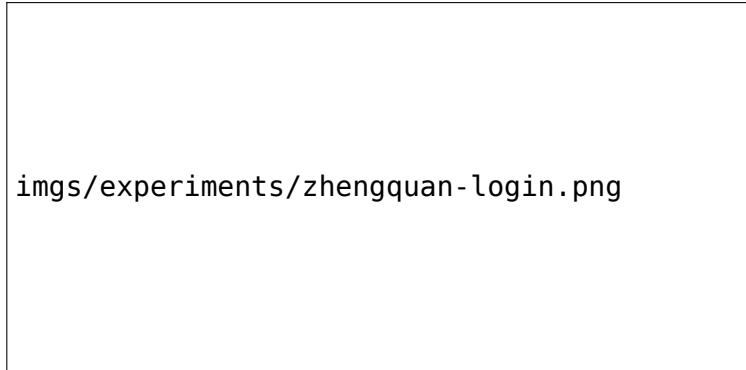


图 10.2: 证券交易软件的登录界面示意图

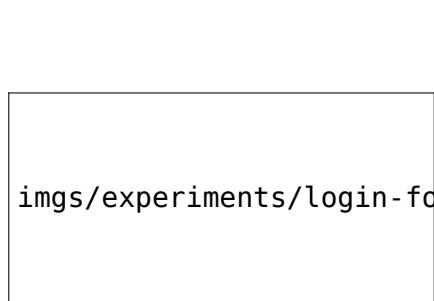
实验要求

给出源代码和 Idea 的设计界面。

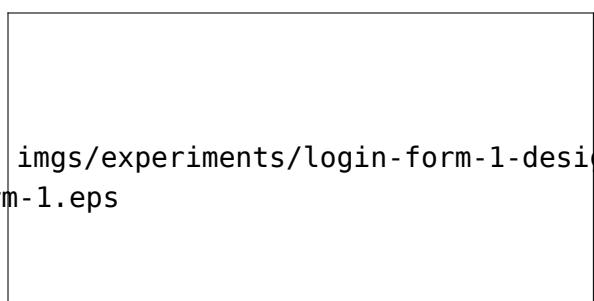
实验步骤

可以分以下几个步骤¹:

1. 使用 GridLayout 将整个界面分为三个部分，如图10.3所示²。



(a) 设计图



(b) 实现图

图 10.3: 界面的第一步划分

¹完整代码参见:<https://github.com/subaochen/java-tutorial/tree/master/guide/code/tests/src/cn/edu/sdut/softlab/gui>

²测试页脚

2. 将主窗口再划分三个部分，如图10.4所示³。

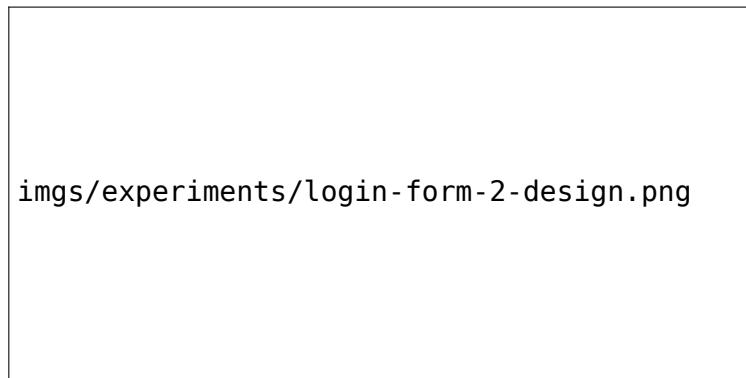


图 10.4: 主窗口的进一步划分

3. 分别在各个部分放置需要的组件，如图10.5所示。



图 10.5: 组织组件

注 10.1. 如何使得按钮的长度适中？

实验六并发应用程序设计

实验目的

- 了解线程的概念。
- 掌握创建 Java 线程的两种方法。

³测试页脚

- 了解 Java 如何通过互斥实现对竞争资源的保护?
- 了解 Java 同步的实现机制。

实验内容

假设有如下的场景：老公（罗密欧）和老婆（朱丽叶）的共同家底（银行余额）是 1000 元，两人同时到银行办理业务：罗密欧去存钱，每笔 100 元，共存了 10 笔；朱丽叶去取钱，每笔 100 元，共取了 10 笔。也就是说，两人业务办理完毕后，两人的共同家底（银行余额）应该依然是 1000 元。试编写一个程序，模拟此操作场景，要求如下：

- 定义 BankAccount 类，包含：
 - 属性：姓名 (name)、帐号 (account)、余额 (balance) 等；
 - 方法：取钱 (withdraw)、存钱 (deposit) 等；
- 定义两个线程类：WithdrawThread(取钱线程) 类和 DepositThread(存钱线程) 类。
- 编写应用程序 BankDemo 类，在 BankDemo 中初始化给定的 BankAccount 对象，启动两个线程类，观察在没有实行同步机制的情况下，BankAccount 类对象中的余额 (balance) 是否具有不确定性；
- 修改 BankAccount 类，利用锁定冲突对象 (`synchronized(this)` 或者 `synchronized(this.account)`) 的方法实行线程的同步。编写程序测试两个线程观察实行同步机制的情况下，BankAccount 类对象中余额 (balance) 的变化。
- 修改 BankAccount 类，通过锁定冲突方法对 withdraw 方法和 deposit 方法实行同步。编写程序测试两个线程，观察实行同步机制的情况下，BankAccount 类对象中余额 (balance) 的变化。

实验要求

- 给出程序的源代码。
- 给出程序运行结果和结果分析。

实验步骤

分三种情况进行。在这三种情况下，主测试程序 BankDemo (参见代码清单 10.13) 和两个线程 (参见代码清单 10.15 和代码清单 10.14) 基本不变 (注意在不同的情形下导入不同包下面的 BankAccount)，只有 BankAccount 类有变化。

代码清单 10.13: BankDemo.java

```

1 package cn.edu.sdut.softlab.concurrency;
2
3 // IMPORTANT! 请根据不同的测试情形打开相应的import语句
4 // 用于无同步保护情形
5 //import cn.edu.sdut.softlab.concurrency.nosync.BankAccount;
6 // 用于同步锁定冲突方法情形
7 //import cn.edu.sdut.softlab.concurrency.syncMethod.BankAccount;
8 // 用于同步锁定对象情形
9 import cn.edu.sdut.softlab.concurrency.syncObject.BankAccount;
10
11 /**
12  * Created by subaochen on 17-2-21.
13 */
14 public class BankDemo {
15     public static void main(String[] args) {
16         BankAccount account = new BankAccount("罗密欧与朱丽叶", "110", 1000);
17
18         Thread withdrawThread = new Thread(new WithdrawThread(account), "罗密欧");
19         Thread depositThread = new Thread(new DepositThread(account), "朱丽叶");
20         withdrawThread.start();
21         depositThread.start();
22
23         try {
24             withdrawThread.join();
25             depositThread.join();
26         } catch (InterruptedException e) {
27             //e.printStackTrace();
28         }
29
30         System.out.println("Finally, balance=" + account.getBalance());
31     }
32 }
```

代码清单 10.14: DepositThread.java

```

1 package cn.edu.sdut.softlab.concurrency;
2
3 // IMPORTANT! 请根据不同的测试情形打开相应的import语句
4 // 用于无同步保护情形
5 //import cn.edu.sdut.softlab.concurrency.nosync.BankAccount;
6 // 用于同步锁定冲突方法情形
7 //import cn.edu.sdut.softlab.concurrency.syncMethod.BankAccount;
8 // 用于同步锁定对象情形
9 import cn.edu.sdut.softlab.concurrency.syncObject.BankAccount;
10
11 /**
12  * 模拟罗密欧存款
13 */
14 public class DepositThread implements Runnable {
15     private BankAccount bankAccount;
```

```

16
17 public DepositThread(BankAccount bankAccount) {
18     this.bankAccount = bankAccount;
19 }
20
21 @Override
22 public void run() {
23     for(int i = 0; i < 10; i++)
24         bankAccount.deposit(10);
25 }
26 }
```

代码清单 10.15: WithdrawThread.java

```

1 package cn.edu.sdut.softlab.concurrency;
2
3 // IMPORTANT! 请根据不同的测试情形打开相应的import语句
4 // 用于无同步保护情形
5 //import cn.edu.sdut.softlab.concurrency.nosync.BankAccount;
6 // 用于同步锁定冲突方法情形
7 //import cn.edu.sdut.softlab.concurrency.syncMethod.BankAccount;
8 // 用于同步锁定对象情形
9 import cn.edu.sdut.softlab.concurrency.syncObject.BankAccount;
10
11 /**
12 * 模拟朱丽叶取钱
13 */
14 public class WithdrawThread implements Runnable{
15     private BankAccount bankAccount;
16
17     public WithdrawThread(BankAccount bankAccount) {
18         this.bankAccount = bankAccount;
19     }
20
21     @Override
22     public void run() {
23         for(int i = 0; i < 10; i++)
24             bankAccount.withdraw(10);
25     }
26 }
```

1. 没有同步保护竞争资源时，BankAccount 代码设计参见代码清单10.16

代码清单 10.16: BankAccount.java

```

1 package cn.edu.sdut.softlab.concurrency.nosync;
2
3 import java.util.Random;
4
5 /**
6 * 模拟银行账户
```

```
7  * @TODO 忽略了余额<0时的处理
8  */
9 public class BankAccount {
10    private String name;
11    private float balance;
12    private String account;
13
14    public BankAccount(String name, String account, float balance) {
15        this.name = name;
16        this.account = account;
17        this.balance = balance;
18    }
19
20    /**
21     * 取钱
22     *
23     * @param money 取钱数额
24     */
25    public void withdraw(float money) {
26        float temp = this.balance;
27        temp -= money;
28        sleep(); // 模拟银行繁复耗时的操作流程，注意在withdraw休眠期间，this.balance可能已被
29        // 经过deposit“悄悄”修改
30        this.balance = temp;
31        System.out.println("-" + Thread.currentThread().getName() + "取钱: " +
32            money);
33    }
34
35    /**
36     * 存钱
37     *
38     * @param money 存钱数额
39     */
40    public void deposit(float money) {
41        float temp = this.balance;
42        temp += money;
43        sleep(); // 模拟银行繁复耗时的操作流程
44        this.balance = temp;
45        System.out.println("+ " + Thread.currentThread().getName() + "存钱: " +
46            money);
47    }
48
49
50    private void sleep() {
51        try {
52            Thread.sleep(new Random().nextInt(500));
53        } catch (InterruptedException e) {
54            //
55        }
56    }
57}
```

```
56 }
57 }
```

2. 锁定冲突对象，只是在代码清单10.16的基础上，在 withdraw 和 deposit 方法上使用了 synchronized 关键字锁定 this 对象，这样 this.balance 的状态就不会因为交错执行而被破坏，或者说，this.balance 的状态能够保证“原子性”：

```
1  public void deposit(float money) {
2      synchronized (this) {
3          float temp = this.balance;
4          temp += money;
5          sleep(); // 模拟银行繁复耗时的操作流程
6          this.balance = temp;
7      }
8      System.out.println("#" + Thread.currentThread().getName() + "存钱: " +
   money);
9  }
```

如果把上面的代码修改为：

```
1  public void deposit(float money) {
2      synchronized (this) {
3          float temp = this.balance;
4          temp += money;
5          sleep(); // 模拟银行繁复耗时的操作流程
6          this.balance = temp;
7          System.out.println("#" + Thread.currentThread().getName() + "存钱: " +
   money);
8      }
9  }
```

试解释执行结果有什么不同？

3. 锁定冲突方法，只是在代码清单10.16的基础上，在 withdraw 和 deposit 方法上使用了 synchronized 关键字，其他的没有变化：

```
1 public synchronized void withdraw(float money) {
2     ...
3 }
4
5 public synchronized void deposit(float money) {
6     ...
7 }
```

实验七网络编程

实验目的

- 了解 TCP/IP 网络的基本概念。
- 掌握 URL 的概念及其读取方法。
- 掌握 Socket 的概念和创建方法。
- 掌握 Socket 编程的基本思路。
- 了解 Apache mina 的基本用法。

实验内容

使用 Socket 编写一个应用程序，分为服务器端和客户端两部分，要求为：

服务器端 每次收到客户端的一个数据包即追加保存到指定的文件中。

客户端 从给定的 URL 读取网页内容，按行发送到服务器端。每次发送一行，直到发送完毕为止。

实验要求

- 给出所编写的应用程序的源代码。
- 给出运行结果，并解释之。

实验步骤

服务器端代码参考代码清单 10.17，客户端参考代码清单 10.18。注意首先运行服务器端，然后再运行客户端。

代码清单 10.17: ServerDemo.java

```
1 package cn.edu.sdu.tsoftlab.network;
2
3 import java.io.FileOutputStream;
4 import java.io.IOException;
5 import java.io.PrintWriter;
6 import java.net.ServerSocket;
7 import java.net.Socket;
8 import java.util.Scanner;
```

```
9
10 /**
11  * Created by subaochen on 17-2-21.
12 */
13 public class ServerDemo {
14     public static void main(String[] args) {
15         try{
16             ServerSocket serverSocket = new ServerSocket(2000);
17             Socket clientSocket = serverSocket.accept();
18             //PrintWriter out = new PrintWriter(clientSocket.getOutputStream());
19             FileOutputStream fis = new FileOutputStream("out.txt");
20             Scanner in = new Scanner(clientSocket.getInputStream());
21         } {
22             String inputLine;
23             while((inputLine = in.nextLine()) != null) {
24                 System.out.println("接收到的数据: " + inputLine);
25                 // 写入文件
26                 fis.write(inputLine.getBytes());
27             }
28
29         } catch (IOException e) {
30             e.printStackTrace();
31         }
32     }
33 }
```

代码清单 10.18: ClientDemo.java

```
1 package cn.edu.sdut.softlab.network;
2
3 import java.io.IOException;
4 import java.io.InputStream;
5 import java.io.OutputStream;
6 import java.net.Socket;
7 import java.net.URL;
8 import java.net.UnknownHostException;
9 import java.util.Scanner;
10
11 /**
12  * Created by subaochen on 17-2-22.
13 */
14 public class ClientDemo {
15     public static void main(String[] args) {
16         try (Socket socket = new Socket("localhost", Integer.parseInt("2000"));
17             OutputStream out = socket.getOutputStream();
18         ) {
19             URL url = new URL("http://www.baidu.com");
20             Scanner console = new Scanner(url.openStream());
21             String inputLine;
22             while(console.hasNext()) {
23                 inputLine = console.nextLine();
```

```

24         System.out.println("准备发往服务器的数据: " + inputLine);
25         out.write(inputLine.getBytes());
26     }
27
28 } catch (UnknownHostException e) {
29     e.printStackTrace();
30 } catch (IOException e) {
31     e.printStackTrace();
32 }
33 }
34 }
```

读者也可以自行将 ?? [在第 ??页]一节的示例程序录入验证和学习。

实验八使用集合类

实验目的

了解和掌握 ArrayList 和 LinkedList 的不同特点和适用场合。

实验内容

编写程序，考察 ArrayList 和 LinkedList 在随机访问和频繁读写情况下的性能对比。

实验要求

在随机访问和频繁读写两种情况下，给出 ArrayList 和 LinkedList 的性能对比数据并绘制图表，给出 ArrayList 和 LinkedList 各自的适用场合的结论。

在不同的读写次数下，ArrayList 和 LinkedList 的对比数据如何变化？

进一步的，如何模拟随机读写情况，即读出和写入的位置使用随机数而不是顺序产生的位置索引数据，再次对比和绘制图表。

实验步骤

参考程序：

代码清单 10.19: ArrayListVsLinkedList.java

```

1 package cn.edu.sdu.tsoftlab.collections;
2
3 import java.util.ArrayList;
4 import java.util.LinkedList;
5 import java.util.List;
```

```
6
7 /**
8  * Created by subaochen on 17-1-22.
9 */
10 public class ArrayListVsLinkedList {
11     public static final int NUM_ADD = 200000; // add测试次数
12     public static final int NUM_GET = 10000; // get测试次数
13     public static final int NUM_REMOVE = 1000; // remove测试次数
14
15     public static void main(String[] args) {
16         List<Integer> arrayList = new ArrayList<>();
17         List<Integer> linkedList = new LinkedList<>();
18
19         // ArrayList add
20         long startTime = System.nanoTime();
21
22         for (int i = 0; i < NUM_ADD; i++) {
23             arrayList.add(i);
24         }
25         long endTime = System.nanoTime();
26         long duration = endTime - startTime;
27         System.out.println("ArrayList add: " + duration);
28
29         // LinkedList add
30         startTime = System.nanoTime();
31
32         for (int i = 0; i < NUM_ADD; i++) {
33             linkedList.add(i);
34         }
35         endTime = System.nanoTime();
36         duration = endTime - startTime;
37         System.out.println("LinkedList add: " + duration);
38
39         // ArrayList get
40         startTime = System.nanoTime();
41
42         for (int i = 0; i < NUM_GET; i++) {
43             arrayList.get(i);
44         }
45         endTime = System.nanoTime();
46         duration = endTime - startTime;
47         System.out.println("ArrayList get: " + duration);
48
49         // LinkedList get
50         startTime = System.nanoTime();
51
52         for (int i = 0; i < NUM_GET; i++) {
53             linkedList.get(i);
54         }
55         endTime = System.nanoTime();
56         duration = endTime - startTime;
57         System.out.println("LinkedList get: " + duration);
```

```
58
59
60     // ArrayList remove
61     startTime = System.nanoTime();
62
63     for (int i = NUM_REMOVE; i >= 0; i--) {
64         arrayList.remove(i);
65     }
66     endTime = System.nanoTime();
67     duration = endTime - startTime;
68     System.out.println("ArrayList remove: " + duration);
69
70
71     // LinkedList remove
72     startTime = System.nanoTime();
73
74     for (int i = NUM_REMOVE; i >= 0; i--) {
75         linkedList.remove(i);
76     }
77     endTime = System.nanoTime();
78     duration = endTime - startTime;
79     System.out.println("LinkedList remove: " + duration);
80 }
81 }
```

参考图表：



参考资料

- ArrayList 和 LinkedList 的性能比较: <http://javaeye-mao.iteye.com/blog/157977>
- 5 Difference Between ArrayList And LinkedList In Java With Example: <http://javahungry.blogspot.com/2015/04/difference-between-arraylist->

and-linkedlist-in-java-example.html

表 目 录

| | |
|---|-----|
| 2.1 C 语言的变量和 Java 语言变量对照表 | 26 |
| 2.2 Java 的数据类型 | 28 |
| 2.3 Java 中的运算符 | 30 |
| 3.1 Java 的访问控制修饰符 | 69 |
| 4.1 抽象类和接口的关系 | 110 |
| 5.1 Number 子类的方法列表 | 132 |
| 5.2 BigDecimal 的常见构造方法 | 139 |
| 5.3 BigDecimal 的计算方法 | 140 |
| 5.4 RoundingMode 中定义的 8 种舍入模式 | 141 |
| 5.5 LocalDate 的常见用法 | 142 |
| 5.6 DateTimeFormatter 的输出格式控制选项 | 148 |
| 7.1 InputStream 输入流 | 174 |
| 7.2 OutputStream 输出流 | 175 |
| 7.3 InputStream 的常用方法 | 176 |
| 7.4 OutputStream 的常用方法 | 177 |
| 7.5 Reader 的基本方法 | 184 |
| 7.6 Writer 的基本方法 | 185 |
| 7.7 Scanner 类的常用方法 | 190 |
| 7.8 Path 信息相关方法 | 195 |
| 7.9 Path 转换方法 | 195 |
| 7.10 比较两个 Path 的方法 | 198 |
| 7.11 File 操作文件属性的方法 | 204 |
| 7.12 批量读取文件属性的方法 | 206 |
| 7.13 读写小文件的方便方法 | 209 |
| 7.14 Files 中带缓存的文本读写方法 | 213 |

| | |
|--|-----|
| 7.15 Files 中不带缓存的输入输出流 | 214 |
| 7.16 Files 类的 Channel 读写文件方法 | 216 |
| 7.17 Files 创建文件和临时文件的方法 | 220 |
| 7.18 SeekableByteChannel 的随机读写方法 | 221 |
| 7.19 创建目录的方法 | 226 |
| 7.20 FileVisitor 接口方法 | 232 |
| | |
| 9.1 JLabel 的主要属性 | 256 |
| 9.2 文本控件的常见属性 | 259 |
| 9.3 JButton 的常见属性 | 261 |
| 9.4 JCheckBox 的常见属性 | 263 |
| 9.5 GridLayout 的常见属性 | 273 |
| 9.6 GridBagLayout 的常见属性 | 275 |
| 9.7 GridBagLayout 示例应用中的控件属性 | 277 |
| 9.8 控件属性的各种组合方式 | 278 |
| 9.9 控件的常见监听器 | 279 |

示例代码列表

| | | |
|------|---------------------------|----|
| 1.1 | HelloWorld.java | 15 |
| 1.2 | IODemo.java | 24 |
| 2.1 | SimpleDataType.java | 29 |
| 2.2 | CircleArea.java | 31 |
| 2.3 | ShowGrade.java | 32 |
| 2.4 | Month.java | 32 |
| 2.5 | EnhancedFor.java | 34 |
| 2.6 | VarargsTest.java | 35 |
| 2.7 | CalculatePi.java | 36 |
| 3.1 | Java 的 Point 类定义 | 41 |
| 3.2 | C 的 point 结构体定义 | 41 |
| 3.3 | Point.java | 41 |
| 3.4 | Draw.java | 43 |
| 3.5 | Rectangle.java | 44 |
| 3.6 | Circle.java | 44 |
| 3.7 | Draw.java | 45 |
| 3.8 | Point.java | 48 |
| 3.9 | Draw.java | 49 |
| 3.10 | Point.java | 51 |
| 3.11 | Circle.java | 52 |
| 3.12 | Rectangle.java | 52 |
| 3.13 | Point.java | 53 |
| 3.14 | Circle.java | 54 |
| 3.15 | BankCard.java | 59 |
| 3.16 | 未使用继承的 CreditCard.java | 60 |
| 3.17 | 使用了继承的 CreditCard.java | 60 |
| 3.18 | 增加无参构造方法的 Card.java | 62 |
| 3.19 | 带有无参构造方法的 BankCard.java | 63 |
| 3.20 | 带有无参构造方法的 CreditCard.java | 63 |

| | |
|--|-----|
| 3.21 带有无参构造方法的 DebitCard.java | 64 |
| 3.22 Client.java | 64 |
| 3.23 Caculator.java | 65 |
| 3.24 Dog.java | 67 |
| 3.25 演示 default 修饰符的 BankCard.java | 70 |
| 3.26 Client.java | 71 |
| 3.27 演示 private 修饰符的 BankCard.java | 72 |
| 3.28 Client.java | 73 |
| 3.29 演示 protected 修饰符的 BankCard.java | 74 |
| 3.30 演示 protected 修饰符的 CreditCard.java | 75 |
| 3.31 Test.java | 75 |
| 3.32 NotePad.java | 78 |
| 3.33 Client.java | 79 |
| 3.34 Person.java | 81 |
| 3.36 Client1.java | 81 |
| 3.35 Client.java | 82 |
| 3.37 CreditCard.java | 85 |
| 3.38 使用了 static 常量的 Book.java | 88 |
| 3.39 Client.java | 89 |
| 3.40 Book.java | 91 |
| 3.41 Client.java | 92 |
| 3.42 Book.java | 94 |
| 3.43 Client.java | 95 |
| 3.44 Book.java | 97 |
| 3.45 Client.java | 98 |
| 3.46 Book.java | 99 |
| 3.47 Client.java | 100 |
| 3.48 Printer.java | 101 |
| 3.49 Copier.java | 101 |
| 3.50 SmartPrinter.java | 102 |
| 3.51 SmartPrinter.java | 103 |
| 4.1 Animal.java | 105 |
| 4.2 Dog.java | 106 |
| 4.3 Client.java | 106 |
| 4.4 Animal.java | 107 |
| 4.5 Dog.java | 108 |
| 4.6 Animal.java | 109 |

| | |
|---|-----|
| 4.7 Dog.java | 111 |
| 4.8 Client.java | 111 |
| 4.9 Printer.java | 112 |
| 4.10 Copier.java | 112 |
| 4.11 SmartPrinter.java | 113 |
| 4.12 Client.java | 113 |
| 4.13 BetterSmartPrinter.java | 115 |
| 4.14 Client.java | 117 |
| 4.15 Animal.java | 118 |
| 4.16 Dog.java | 118 |
| 4.17 Cat.java | 119 |
| 4.18 Duke.java | 119 |
| 4.19 TestAnimal.java | 120 |
| 5.1 Client.java | 126 |
| 5.2 SubStringTest.java | 127 |
| 5.3 StringMatch.java | 130 |
| 5.4 WrapperNumberTest.java | 133 |
| 5.5 NumberTest.java | 135 |
| 5.6 LocalDateTest.java | 143 |
| 5.7 LocalTimeTest.java | 144 |
| 5.8 LocalDateTimeTest.java | 146 |
| 5.9 DateTimeFormatterTest.java | 149 |
| 5.10 DateTimeAPITest.java | 150 |
| 6.1 WhatIfNoException.java | 154 |
| 6.2 DivTest.java | 159 |
| 6.3 Example.java | 162 |
| 6.4 OutOfInventoryException.java | 165 |
| 6.5 PriceNotAvailableException.java | 166 |
| 7.1 CopyBinary.java | 178 |
| 7.2 Temperature.java | 180 |
| 7.3 Multiply.java | 189 |
| 7.4 CreatePathTest.java | 193 |
| 7.5 PathInfoTest.java | 194 |
| 7.6 PathConversionTest.java | 196 |
| 7.7 PathResolveTest.java | 197 |
| 7.8 PathCompareTest.java | 199 |
| 7.9 PathExistTest.java | 200 |

| | |
|---|-----|
| 7.10 PathDeleteTest.java | 201 |
| 7.11 PathCopyTest.java | 201 |
| 7.12 PathMoveTest.java | 203 |
| 7.13 PathMetadataTest.java | 207 |
| 7.14 PathCreateTest.java | 211 |
| 7.15 PathWithoutBufferTest.java | 214 |
| 7.16 FileChannelTest.java | 216 |
| 7.17 FileCreateTest.java | 219 |
| 7.18 FileRandomAccessTest.java | 222 |
| 7.19 CreateDirectoryTest.java | 226 |
| 7.20 ListDirectoryTest.java | 228 |
| 7.21 ListDirectoryWithGlobTest.java | 229 |
| 7.22 PrintFiles.java | 229 |
| 7.23 WalkFileTreeTest.java | 231 |
| 7.24 PropertyFileTest.java | 232 |
| 8.1 WeekDay.java | 235 |
| 8.2 WeekDayTest.java | 236 |
| 8.3 Month.java | 238 |
| 8.4 MonthTest.java | 239 |
| 8.5 TrafficSignal.java | 239 |
| 8.6 TrafficSignalTest.java | 239 |
| 8.7 EnumInSwitchTest.java | 240 |
| 9.1 Caculator.java | 281 |
| 10.1 Animal.java | 289 |
| 10.2 Dog.java | 291 |
| 10.3 Cat.java | 292 |
| 10.4 SheepDog.java | 293 |
| 10.5 MouseKiller.java | 294 |
| 10.6 PeopleHelper.java | 294 |
| 10.7 MyException.java | 295 |
| 10.8 Demo.java | 296 |
| 10.9 Client.java | 296 |
| 10.10 StandardIO.java | 297 |
| 10.11 CatFile.java | 298 |
| 10.12 MaxMin.java | 299 |
| 10.13 BankDemo.java | 303 |
| 10.14 DepositThread.java | 303 |

| | |
|--|-----|
| 10.15 WithdrawThread.java | 304 |
| 10.16 BankAccount.java | 304 |
| 10.17 ServerDemo.java | 307 |
| 10.18 ClientDemo.java | 308 |
| 10.19 ArrayListVsLinkedList.java | 309 |

附录 A 建议的授课计划

| 序号 | 主要内容 | 说明 |
|----|--|---|
| 1 | <ul style="list-style-type: none">为什么要学习 Java 程序设计语言?Java 语言的特点JDK 的概念和安装使用 Idea 搭建 Java 开发环境, HelloWorld 演示基于终端的 Java 开发 | 从一开始就强调读书和练习相结合, 避免眼高手低 |
| 2 | <ul style="list-style-type: none">迅速回顾 C 的基本语法规则, 包括变量、运算符、表达式、控制结构重点讲述 Java 语言在基本语法规则方面和 C 语言的差异<ul style="list-style-type: none">- Java 变量命名常用驼峰命名法- Java 特有的运算符: new, instanceof- Java 加强了的 for 循环- Java 加强了的 switch 分支结构- 不定长方法参数 | 由于 Java 语言和 C 语言在基础语法规则上 90% 是相同的, 因此本部分可以快速通过, 这样避免了学生过多的纠缠于语法细节, 也减少了学习 Java 语言的恐惧感 (被繁杂的语法吓倒)。 |

| | | |
|---|---|--|
| 3 | <ul style="list-style-type: none"> • 类和对象的概念 • Java 如何创建类? 如何创建对象? • 构造方法的用法 • 类的组织: 包的用法 • 类的层次结构: 继承 | <p>由于大部分读者是第一次接触 OOP, 此部分可以稍微放慢授课的速度, 通过大量的实例以及和生活中的现象类比掌握 OOP 的基本概念和用法。</p> |
| 4 | <ul style="list-style-type: none"> • Java 的访问控制 • 理解引用类型, 尤其是方法参数中的引用类型 • static 的几种情形 | <p>建议通过反复的演示和练习使读者充分理解本部分内容。</p> |
| 5 | <ul style="list-style-type: none"> • 抽象类等的概念及其用法 • 接口是纯的抽象类, 接口的设计目的 • 多态的概念, 多态的实现方法 • 面向接口的编程 | <p>本部分内容比较抽象, 建议重点放到抽象类和接口的概念和使用。</p> |
| 6 | <ul style="list-style-type: none"> • <code>String</code> 类的常见方法 • 数字的表达, 字符串和数字的相互转化方法 • 时间和日期的表达和常见处理方法 | |

| | | |
|----|---|---|
| 7 | <ul style="list-style-type: none"> • Java 异常的概念，注意和 C 语言的比较 • 异常的分类，异常类的层次结构 • try-catch 结构 • throw 和 throws 的区别 • 如何编写自定义异常类？ • try-with-resources 结构的使用场合 | <p>理解异常的责任链机制是掌握 Java 异常处理的关键</p> |
| 8 | <p>流的概念（C 语言中就有） 面向字节和面向字符的流 从键盘输入的方法 文件的读写：NIO（Path, Files）</p> | <p>本部分内容多，重点是 InputStream/OutputStream/Reader/Writer 的用法，NIO 可以作为较高要求处理</p> |
| 9 | <ul style="list-style-type: none"> • Enum 的概念（和 C 语言的对比） • Java 中的 Enum 是面向对象的 • 工程实践中 Enum 的重要性 • switch 中的 Enum | |
| 10 | <ul style="list-style-type: none"> • 窗口的基本概念 • Swing GUI 设计常见组件：Panel, JLabel 等 • 布局管理器 | <p>重点是学生要通过练习掌握起来，本部分不要求学生了解 GUI 的很多底层知识，学会使用 Idea 进行 GUI 界面设计即可。</p> |

| | | |
|----|---|------------------------------|
| 11 | <ul style="list-style-type: none"> • Swing 的事件机制（在上节课中其实已经在不断使用，这里总结一下） • 匿名类在事件处理中的使用 • Java GUI 综合示例 | |
| 12 | 习题课 | 建议在这里插入一次习题课作为“中场休息”，巩固已学知识。 |
| 13 | <ul style="list-style-type: none"> • 为什么要引入泛型？ • 泛型的常见形态（用法） | 要注意到讲述这部分内容时还没有学习“集合类” |
| 14 | <ul style="list-style-type: none"> • 泛型中的通配符 • 泛型的类型擦除 | 本部分内容可以作为较高要求 |
| 15 | 集合类框架 常见集合类：List、Set、Map、Queue 的用法 | 通过实例，可结合泛型演示 |
| 16 | 总结集合类中的泛型 集合类中的 PECS 原则 | 本部分内容可作为较高要求 |
| 17 | <ul style="list-style-type: none"> • lambda 表达式基本概念：为什么引入 lambda 表达式？ • lambda 表达式的常见形式 • 集合类的流式处理 | 本部分可以作为较高要求内容 |

| | | |
|----|---|---------------------------------|
| 18 | <ul style="list-style-type: none">• 线程的概念• Java 创建线程的两种方法• 竞争资源的概念• 线程的同步和互斥处理方法• synchronized 和 wait()/notifyAll() 的联合使用举例 | 重点是线程的概念和 synchronized 如何保护竞争资源 |
| 19 | <ul style="list-style-type: none">• 网络基础知识简单回顾• URL 的表达和内容读取• Socket 的概念• Java Socket 服务器端和客户端编程• Apache mina 简介 | Apache mina 可以简单一提，要求有兴趣的学生自学。 |
| 20 | <ul style="list-style-type: none">• 注解的基本概念• 注解的基本写法• 什么是 DI?• 注解在 DI 中的应用 | DI 可作为较高要求 |

附录 B Java FAQ

B.1 如何设置命令行参数？

分以下 2 种情况：

1. 如果是从命令行执行，直接在 Java 应用程序后面设置命令行参数即可，比如：

```
java Client arg-list
```

2. 如果使用 IDE，一般的 IDE 在 run 菜单都有一个“configuration....”菜单，通过此菜单可以设置应用程序的命令行参数。比如 Idea 中，通过如图B.1的方式设置命令行参数。

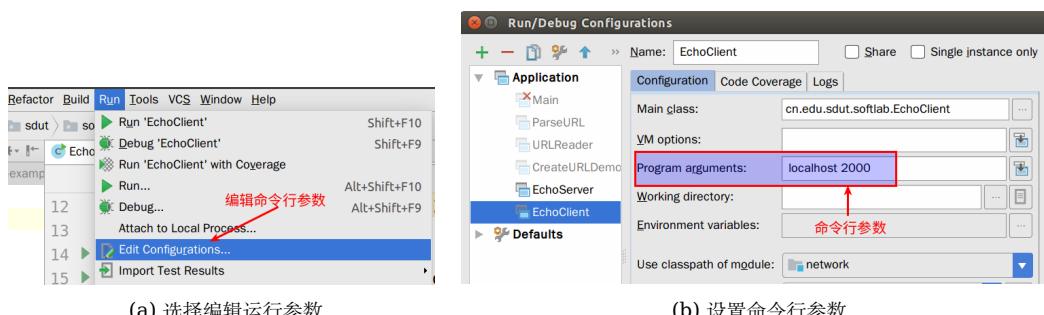


图 B.1: Idea 的命令行参数设置方法

附录 C Java GUI 控件和事件监听器 对照表

参见:<http://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html>

附录 D JDK 1.5、1.6、1.7、1.8 的差別

对于很多刚接触 java 语言的初学者来说，要了解一门语言，最好的方式就是要能从基础的版本进行了解，升级的过程，以及升级的新特性，这样才能循序渐进的学好一门语言。今天先为大家介绍一下 JDK1.5 版本到 JDK1.7 版本的特性。希望能给予帮助。

JDK1.5 新特性：1. 自动装箱与拆箱：

自动装箱的过程：每当需要一种类型的对象时，这种基本类型就自动地封装到与它相同类型的包装中。

自动拆箱的过程：每当需要一个值时，被装箱对象中的值就被自动地提取出来，没必要再去调用 intValue 和 doubleValue 方法。

自动装箱，只需将该值赋给一个类型包装器引用，java 会自动创建一个对象。

自动拆箱，只需将该对象值赋给一个基本类型即可。

java——类的包装器

类型包装器有：Double,Float,Long,Integer,Short,Character 和 Boolean

2. 枚举

把集合里的对象元素一个一个提取出来。枚举类型使代码更具可读性，理解清晰，易于维护。枚举类型是强类型的，从而保证了系统安全性。而以类的静态字段实现的类似替代模型，不具有枚举的简单性和类型安全性。

简单的用法：JavaEnum 简单的用法一般用于代表一组常用常量，可用来代表一类相同类型的常量值。

复杂用法：Java 为枚举类型提供了一些内置的方法，同事枚举常量还可以有自己的方法。可以很方便的遍历枚举对象。

3. 静态导入

通过使用 import static，就可以不用指定 Constants 类名而直接使用静态成员，包括静态方法。

import xxxx 和 import static xxxx 的区别是前者一般导入的是类文件如 import java.util.Scanner; 后者一般是导入静态的方法，import static java.lang.System.out。

4. 可变参数（Varargs）

可变参数的简单语法格式为：

```
methodName([argumentList], dataType...argumentName);
```

5. 内省 (Introspector)

是 Java 语言对 Bean 类属性、事件的一种缺省处理方法。例如类 A 中有属性 name, 那我们可以通过 getName, setName 来得到其值或者设置新的值。通过 getName/setName 来访问 name 属性, 这就是默认的规则。Java 中提供了一套 API 用来访问某个属性的 getter /setter 方法, 通过这些 API 可以使你不需要了解这个规则(但你最好还是要搞清楚), 这些 API 存放于包 java.beans 中。

一般的做法是通过类 Introspector 来获取某个对象的 BeanInfo 信息, 然后通过 BeanInfo 来获取属性的描述器 (PropertyDescriptor), 通过这个属性描述器就可以获取某个属性对应的 getter/setter 方法, 然后我们就可以通过反射机制来调用这些方法。

6. 泛型 (Generic)

C++ 通过模板技术可以指定集合的元素类型, 而 Java 在 1.5 之前一直没有相对的功能。一个集合可以放任何类型的对象, 相应地从集合里面拿对象的时候我们也不得不对他们进行强制得类型转换。猛虎引入了泛型, 它允许指定集合里元素的类型, 这样你可以得到强类型在编译时刻进行类型检查的好处。

7. For-Each 循环

For-Each 循环得加入简化了集合的遍历。假设我们要遍历一个集合对其中的元素进行一些处理。

JDK 1.6 新特性

有关 JDK1.6 的新特性 reamerit 的博客文章已经说的很详细了。

1. Desktop 类和 SystemTray 类

在 JDK6 中, AWT 新增加了两个类: Desktop 和 SystemTray。

前者可以用来打开系统默认浏览器浏览指定的 URL, 打开系统默认邮件客户端给指定的邮箱发邮件, 用默认应用程序打开或编辑文件 (比如, 用记事本打开以 txt 为后缀名的文件), 用系统默认的打印机打印文档; 后者可以用来在系统托盘区创建一个托盘程序.

2. 使用 JAXB2 来实现对象与 XML 之间的映射

JAXB 是 Java Architecture for XML Binding 的缩写, 可以将一个 Java 对象转变成为 XML 格式, 反之亦然。

我们把对象与关系数据库之间的映射称为 ORM, 其实也可以把对象与 XML 之间的映射称为 OXM(Object XML Mapping). 原来 JAXB 是 Java EE 的一部分, 在 JDK6 中, SUN 将其放到了 Java SE 中, 这也是 SUN 的一贯做法。JDK6 中自带的这个 JAXB 版本是 2.0, 比起 1.0(JSR 31) 来, JAXB2(JSR 222) 用 JDK5 的新特性 Annotation 来标识要作绑定的类和属性等, 这就极大简化了开发的工作量。

实际上, 在 Java EE 5.0 中, EJB 和 Web Services 也通过 Annotation 来简化开发工作。另外,JAXB2 在底层是用 StAX(JSR 173) 来处理 XML 文档。除了 JAXB

之外，我们还可以通过 XMLBeans 和 Castor 等来实现同样的功能。

3. 理解 StAX

StAX(JSR 173) 是 JDK6.0 中除了 DOM 和 SAX 之外的又一种处理 XML 文档的 API。

StAX 的来历：在 JAXP1.3(JSR 206) 有两种处理 XML 文档的方法：DOM(Document Object Model) 和 SAX(Simple API for XML)。

由于 JDK6.0 中的 JAXB2(JSR 222) 和 JAX-WS 2.0(JSR 224) 都会用到 StAX 所以 Sun 决定把 StAX 加入到 JAXP 家族当中来，并将 JAXP 的版本升级到 1.4(JAXP1.4 是 JAXP1.3 的维护版本)。JDK6 里面 JAXP 的版本就是 1.4.

StAX 是 The Streaming API for XML 的缩写，一种利用拉模式解析(pull-parsing)XML 文档的 API。StAX 通过提供一种基于事件迭代器(Iterator)的 API 让程序员去控制 xml 文档解析过程，程序遍历这个事件迭代器去处理每一个解析事件，解析事件可以看做是程序拉出来的，也就是程序促使解析器产生一个解析事件然后处理该事件，之后又促使解析器产生下一个解析事件，如此循环直到碰到文档结束符；

SAX 也是基于事件处理 xml 文档，但却是用推模式解析，解析器解析完整个 xml 文档后，才产生解析事件，然后推给程序去处理这些事件；DOM 采用的方式是将整个 xml 文档映射到一颗内存树，这样就可以很容易地得到父节点和子结点以及兄弟节点的数据，但如果文档很大，将会严重影响性能。

4. 使用 Compiler API

现在我们可以用 JDK6 的 Compiler API(JSR 199) 去动态编译 Java 源文件，Compiler API 结合反射功能就可以实现动态的产生 Java 代码并编译执行这些代码，有点动态语言的特征。

这个特性对于某些需要用到动态编译的应用程序相当有用，比如 JSP Web Server，当我们手动修改 JSP 后，是不希望需要重启 Web Server 才可以看到效果的，这时候我们就可以用 Compiler API 来实现动态编译 JSP 文件，当然，现在的 JSP Web Server 也是支持 JSP 热部署的，现在的 JSP Web Server 通过在运行期间通过 Runtime.exec 或 ProcessBuilder 来调用 javac 来编译代码，这种方式需要我们产生另一个进程去做编译工作，不够优雅而且容易使代码依赖于特定的操作系统；Compiler API 通过一套易用的标准的 API 提供了更加丰富的方式去做动态编译，而且是跨平台的。

5. 轻量级 Http Server API

JDK6 提供了一个简单的 Http Server API，据此我们可以构建自己的嵌入式 Http Server，它支持 Http 和 Https 协议，提供了 HTTP1.1 的部分实现，没有被实现的部分可以通过扩展已有的 Http Server API 来实现，程序员必须自己实现 HttpHandler 接口，HttpServer 会调用 HttpHandler 实现类的回调方法来处理客户端请求，在这里，我们把一个 Http 请求和它的响应称为一个交换，包装成 HttpExchange 类，HttpServer 负责将 HttpExchange 传给 HttpHandler 实现类的回调方法。

6. 插入式注解处理 API(Pluggable Annotation Processing API)

插入式注解处理 API(JSR 269) 提供一套标准 API 来处理 Annotations(JSR 175)

实际上 JSR 269 不仅仅用来处理 Annotation, 我觉得更强大的功能是它建立了 Java 语言本身的一个模型, 它把 method, package, constructor, type, variable, enum, annotation 等 Java 语言元素映射为 Types 和 Elements(两者有什么区别?), 从而将 Java 语言的语义映射成为对象, 我们可以在 javax.lang.model 包下面可以看到这些类. 所以我们可以利用 JSR 269 提供的 API 来构建一个功能丰富的元编程 (metaprogramming) 环境.

JSR 269 用 Annotation Processor 在编译期间而不是运行期间处理 Annotation, Annotation Processor 相当于编译器的一个插件, 所以称为插入式注解处理. 如果 Annotation Processor 处理 Annotation 时 (执行 process 方法) 产生了新的 Java 代码, 编译器会再调用一次 Annotation Processor, 如果第二次处理还有新代码产生, 就会接着调用 Annotation Processor, 直到没有新代码产生为止. 每执行一次 process 方法被称为一个“round”, 这样整个 Annotation processing 过程可以看作是一个 round 的序列.

JSR 269 主要被设计成为针对 Tools 或者容器的 API. 举个例子, 我们想建立一套基于 Annotation 的单元测试框架 (如 TestNG), 在测试类里面用 Annotation 来标识测试期间需要执行的测试方法。

7. 用 Console 开发控制台程序

JDK6 中提供了 java.io.Console 类专用来访问基于字符的控制台设备. 你的程序如果要与 Windows 下的 cmd 或者 Linux 下的 Terminal 交互, 就可以用 Console 类代劳. 但我们不总是能得到可用的 Console, 一个 JVM 是否有可用的 Console 依赖于底层平台和 JVM 如何被调用. 如果 JVM 是在交互式命令行 (比如 Windows 的 cmd) 中启动的, 并且输入输出没有重定向到另外的地方, 那么就可以得到一个可用的 Console 实例.

8. 对脚本语言的支持如: ruby, groovy, javascript.

9. Common Annotations

Common annotations 原本是 Java EE 5.0(JSR 244) 规范的一部分, 现在 SUN 把它的一部分放到了 Java SE 6.0 中.

随着 Annotation 元数据功能 (JSR 175) 加入到 Java SE 5.0 里面, 很多 Java 技术 (比如 EJB, Web Services) 都会用 Annotation 部分代替 XML 文件来配置运行参数 (或者说是支持声明式编程, 如 EJB 的声明式事务), 如果这些技术为通用目的都单独定义了自己的 Annotations, 显然有点重复建设, 所以, 为其他相关的 Java 技术定义一套公共的 Annotation 是有价值的, 可以避免重复建设的同时, 也保证 Java SE 和 Java EE 各种技术的一致性.

下面列举出 Common Annotations 1.0 里面的 10 个 Annotations Common Annotations

Annotation Retention Target Description

Generated Source ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE 用于标注生成的源代码

Resource Runtime TYPE, METHOD, FIELD 用于标注所依赖的资源，容器据此注入外部资源依赖，有基于字段的注入和基于 `setter` 方法的注入两种方式

Resources Runtime TYPE 同时标注多个外部依赖，容器会把所有这些外部依赖注入

PostConstruct Runtime METHOD 标注当容器注入所有依赖之后运行的方法，用来进行依赖注入后的初始化工作，只有一个方法可以标注为 `PostConstruct`

PreDestroy Runtime METHOD 当对象实例将要被从容器当中删掉之前，要执行的回调方法要标注为 `PreDestroy` RunAs Runtime TYPE 用于标注用什么安全角色来执行被标注类的方法，这个安全角色必须和 Container 的 Security 角色一致的。 RolesAllowed Runtime TYPE, METHOD 用于标注允许执行被标注类或方法的安全角色，这个安全角色必须和 Container 的 Security 角色一致的

`PermitAll` Runtime TYPE, METHOD 允许所有角色执行被标注的类或方法

`DenyAll` Runtime TYPE, METHOD 不允许任何角色执行被标注的类或方法，表明该类或方法不能在 Java EE 容器里面运行

`DeclareRoles` Runtime TYPE 用来定义可以被应用程序检验的安全角色，通常用 `isUserInRole` 来检验安全角色

注意：

1. `RolesAllowed, PermitAll, DenyAll` 不能同时应用到一个类或方法上

2. 标注在方法上的 `RolesAllowed, PermitAll, DenyAll` 会覆盖标注在类上的 `RolesAllowed, PermitAll, DenyAll`

3. `RunAs, RolesAllowed, PermitAll, DenyAll` 和 `DeclareRoles` 还没有加到 Java SE 6.0 上来

4. 处理以上 Annotations 的工作是由 Java EE 容器来做，Java SE 6.0 只是包含了上面表格的前五种 Annotations 的定义类，并没有包含处理这些 Annotations 的引擎，这个工作可以由 Pluggable Annotation Processing API(JSR 269) 来做

改动的地方最大的就是 java GUI 界面的显示了，JDK6.0（也就是 JDK1.6）支持最新的 windows vista 系统的 Windows Aero 视窗效果，而 JDK1.5 不支持!!!

你要在 vista 环境下编程的话最好装 jdk6.0，否则它总是换到 windows basic 视窗效果。

JDK 1.7 新特性

1, switch 中可以使用字符串

```
String s = "test";
```

```
switch (s) {
```

```
case "test" :
```

```

System.out.println("test");
case "test1" :
    System.out.println("test1");
    break ;
default :
    System.out.println("break");
    break ;
}

```

2, "<>" 这个玩意儿的运用 ListtempList = new ArrayList<>; 即泛型实例化类型自动推断。; // JDK 7 supports limited type inference for generic instance creation Listlst2 = new ArrayList<>; lst1.add("Mon"); lst1.add("Tue"); lst2.add("Wed")); lst2.add("Thu"); for (String item: lst1) { System.out.println(item); } for (String item: lst2) { System.out.println(item); } } }

3. 自定义自动关闭类

以下是 jdk7 api 中的接口, (不过注释太长, 删掉了 close 方法的一部分注释)

```

/** * A resource that must be closed when it is no longer needed. *
 * @author Josh Bloch * @since 1.7 */publicinterface AutoCloseable { /**
 * Closes this resource, relinquishing any underlying resources. * This method
 * is invoked automatically on objects managed by the * {@code try}-with-
 * resources statement. */void close throws Exception; }

```

只要实现该接口, 在该类对象销毁时自动调用 close 方法, 你可以在 close 方法关闭你想关闭的资源, 例子如下

```

class TryClose implements AutoCloseable { @Override publicvoid close
throw Exception { System.out.println(" Custom close method ...close re-
sources "); } }

```

//请看 jdk 自带类 BufferedReader 如何实现 close 方法 (当然还有很多类似类型的类)

```

public void close throws IOException {
    synchronized (lock) {
        if (in == null)
            return;
        in.close;
        in = null;
        cb = null;
    }
}

```

4. 新增一些取环境信息的工具方法

File System.getJavaIoTempDir // IO 临时文件夹
 File System.getJavaHomeDir // JRE 的安装目录
 File System.getUserHomeDir // 当前用户目录
 File System.getUserDir // 启动 java 进程时所在的目录

 5. Boolean 类型反转, 空指针安全, 参与位运算

Boolean Booleans.negate(Boolean booleanObj)
 True => False , False => True, Null => Null
 boolean Booleans.and(boolean array)
 boolean Booleans.or(boolean array)
 boolean Booleans.xor(boolean array)
 boolean Booleans.and(Boolean array)
 boolean Booleans.or(Boolean array)
 boolean Booleans.xor(Boolean array)

6. 两个 char 间的 equals

boolean Character.equalsIgnoreCase(char ch1, char ch2)

7, 安全的加减乘除

int Math.safeToInt(long value)
 int Math.safeNegate(int value)
 long Math.safeSubtract(long value1, int value2)
 long Math.safeSubtract(long value1, long value2)
 int Math.safeMultiply(int value1, int value2)
 long Math.safeMultiply(long value1, int value2)
 long Math.safeMultiply(long value1, long value2)
 long Math.safeNegate(long value)
 int Math.safeAdd(int value1, int value2)
 long Math.safeAdd(long value1, int value2)
 long Math.safeAdd(long value1, long value2)
 int Math.safeSubtract(int value1, int value2)

1. 对 Java 集合 (Collections) 的增强支持

在 JDK1.7 之前的版本中, Java 集合容器中存取元素的形式如下:

以 List、Set、Map 集合容器为例:

```
//创建 List 接口对象
list=new ArrayList;
list.add("item"); //用 add 方法获取对象
String item=list.get(0); //用 get 方法获取对象
//创建 Set 接口对象
```

```
set=new HashSet;
set.add("item"); //用 add 方法添加对象
//创建 Map 接口对象
map=new HashMap;
map.put("key",1); //用 put 方法添加对象
int value=map.get("key");
```

在 JDK1.7 中，摒弃了 Java 集合接口的实现类，如：ArrayList、HashSet 和 HashMap。而是直接采用、{} 的形式存入对象，采用的形式按照索引、键值来获取集合中的对象，如下：

```
list={"item"}; //向 List 集合中添加元素
String item=list[0]; //从 List 集合中获取元素
set={"item"}; //向 Set 集合对象中添加元素
Map
int value=map["key"]; //从 Map 集合中获取对象
```

2. 在 Switch 中可用 String

在之前的版本中是不支持在 Switch 语句块中用 String 类型的数据的，这个功能在 C# 语言中早已被支持，好在 JDK1.7 中加入了。

```
String s = "test";
switch (s) {
    case "test" :
        System.out.println("test");
    case "test1" :
        System.out.println("test1");
        break ;
    default :
        System.out.println("break");
        break ;
}
```

3. 数值可加下划线

例如： int one_million = 1_000_000;

4. 支持二进制文字

例如： int binary = 0b1001_1001;

5. 简化了可变参数方法的调用

当程序员试图使用一个不可具体化的可变参数并调用一个 *varargs*（可变）方法时，编辑器会生成一个“非安全操作”的警告。

6、在 try catch 异常捕捉中，一个 catch 可以写多个异常类型，用“|”隔开，jdk7 之前：

```

try { ..... } catch(ClassNotFoundException ex) { ex.printStackTrace; }
catch(SQLException ex) { ex.printStackTrace; }

jdk7 例子如下
try { ..... } catch(ClassNotFoundException|SQLException ex) { ex.printStackTrace;
}

```

7、jdk7 之前，你必须用 try{}finally{} 在 try 内使用资源，在 finally 中关闭资源，不管 try 中的代码是否正常退出或者异常退出。jdk7 之后，你可以不必要写 finally 语句来关闭资源，只要你在 try 的括号内部定义要使用的资源。请看例子：

jdk7 之前

```

import java.io.*; // Copy from one file to another file character by character. // Pre-JDK 7 requires you to close the resources using a finally block.
public class FileCopyPreJDK7 {
    public static void main(String args) {
        BufferedReader in = null;
        BufferedWriter out = null;
        try {
            in = new BufferedReader(new FileReader("in.txt"));
            out = new BufferedWriter(new FileWriter("out.txt"));
            int charRead;
            while ((charRead = in.read()) != -1) {
                System.out.printf("%c ", (char)charRead);
                out.write(charRead);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        } finally {
            // always close the stream
            if (in != null)
                in.close();
            if (out != null)
                out.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        try {
            in.read(); // Trigger IOException: Stream closed
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

```

jdk7 之后

```

import java.io.*; // Copy from one file to another file character by character. // JDK 7 has a try-with-resources statement, which ensures that // each resource opened in try is closed at the end of the statement.
public class FileCopyJDK7 {
    public static void main(String args) {
        try (BufferedReader in = new BufferedReader(new FileReader("in.txt"));
             BufferedWriter out = new BufferedWriter(new FileWriter("out.txt"))) {
            int charRead;
            while ((charRead = in.read()) != -1) {
                System.out.printf("%c ", (char)charRead);
                out.write(charRead);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

```

本教程将 Java8 的新特性逐一列出，并将使用简单的代码示例来指导你如何使用默认接口方法，lambda 表达式，方法引用以及多重 Annotation，之后你将会学到最新的 API 上的改进，比如流，函数式接口，Map 以及全新的日期 API

“Java is still not dead—and people are starting to figure that out.”

本教程将用带注释的简单代码来描述新特性，你将看不到大片吓人的文字。

一、接口的默认方法

Java 8 允许我们给接口添加一个非抽象的方法实现，只需要使用 default 关键字即可，这个特征又叫做扩展方法，示例如下：

```
interface Formula {
    double calculate(int a);
    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

Formula 接口在拥有 calculate 方法之外同时还定义了 sqrt 方法，实现了 Formula 接口的子类只需要实现一个 calculate 方法，默认方法 sqrt 将在子类上可以直接使用。

```
Formula formula = new Formula {
    @Override
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};
formula.calculate(100); // 100.0
formula.sqrt(16); // 4.0
```

文中的 formula 被实现为一个匿名类的实例，该代码非常容易理解，6 行代码实现了计算 $\text{sqrt}(a * 100)$ 。在下一节中，我们将会看到实现单方法接口的更简单的做法。

译者注：在 Java 中只有单继承，如果要让一个类赋予新的特性，通常是使用接口来实现，在 C++ 中支持多继承，允许一个子类同时具有多个父类的接口与功能，在其他语言中，让一个类同时具有其他的可复用代码的方法叫做 mixin。新的 Java 8 的这个特新在编译器实现的角度上来说更加接近 Scala 的 trait。在 C# 中也有名为扩展方法的概念，允许给已存在的类型扩展方法，和 Java 8 的这个在语义上有差别。

二、Lambda 表达式

首先看看在老版本的 Java 中是如何排列字符串的：

```
Listnames = Arrays.asList("peter", "anna", "mike", "xenia"); Collections.sort(names
new Comparator{
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

只需要给静态方法 Collections.sort 传入一个 List 对象以及一个比较器来按指定顺序排列。通常做法都是创建一个匿名的比较器对象然后将其传递给 sort 方法。

在 Java 8 中你就没必要使用这种传统的匿名对象的方式了，Java 8 提供了更简洁的语法，lambda 表达式：

复制代码代码如下:

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

看到了吧，代码变得更段且更具有可读性，但是实际上还可以写得更短：

复制代码代码如下:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

对于函数体只有一行代码的，你可以去掉大括号 {} 以及 return 关键字，但是你还可以写得更短点：

复制代码代码如下:

```
Collections.sort(names, (a, b) -> b.compareTo(a));
```

Java 编译器可以自动推导出参数类型，所以你可以不用再写一次类型。接下来我们看看 lambda 表达式还能作出什么更方便的东西来：

三、函数式接口

Lambda 表达式是如何在 java 的类型系统中表示的呢？每一个 lambda 表达式都对应一个类型，通常是接口类型。而“函数式接口”是指仅仅只包含一个抽象方法的接口，每一个该类型的 lambda 表达式都会被匹配到这个抽象方法。因为默认方法不算抽象方法，所以你也可以给你的函数式接口添加默认方法。

我们可以将 lambda 表达式当作任意只包含一个抽象方法的接口类型，确保你的接口一定达到这个要求，你只需要给你的接口添加 @FunctionalInterface 注解，编译器如果发现你标注了这个注解的接口有多于一个抽象方法的时候会报错的。

```
@FunctionalInterface
{Converter converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted); // 123}
```

需要注意如果 @FunctionalInterface 如果没有指定，上面的代码也是对的。

译者注将 lambda 表达式映射到一个单方法的接口上，这种做法在 Java 8 之前就有别的语言实现，比如 Rhino JavaScript 解释器，如果一个函数参数接收一个单方法的接口而你传递的是一个 function，Rhino 解释器会自动做一个单接口的实例到 function 的适配器，典型的应用场景有 org.w3c.dom.events.EventTarget 的 addEventListener 第二个参数 EventListener。

```
Integer converted = converter.convert("123");
System.out.println(converted); // 123
```

Java 8 允许你使用::关键字来传递方法或者构造函数引用，上面的代码展示了如何引用一个静态方法，我们也可以引用一个对象的方法：

复制代码代码如下:

```
converter = something::startsWith;
```

```
String converted = converter.convert("Java");
System.out.println(converted); // "J"
```

接下来看看构造函数是如何使用`::`关键字来引用的，首先我们定义一个包含多个构造函数的简单类：

```
class Person {
    String firstName;
    String lastName;
    Person {}  

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

接下来我们指定一个用来创建 `Person` 对象的对象工厂接口：

```
interface PersonFactory {
    P create(String firstName, String lastName);
}
```

这里我们使用构造函数引用来将他们关联起来，而不是实现一个完整的工厂：

```
PersonFactory
Person person = personFactory.create("Peter", "Parker");
```

我们只需要使用 `Person::new` 来获取 `Person` 类构造函数的引用，Java 编译器会自动根据 `PersonFactory.create` 方法的签名来选择合适的构造函数。

五、Lambda 作用域

在 `lambda` 表达式中访问外层作用域和老版本的匿名对象中的方式很相似。你可以直接访问标记了 `final` 的外层局部变量，或者实例的字段以及静态变量。

六、访问局部变量

```
final int num = 1;
stringConverter =
(from) -> String.valueOf(from + num);
stringConverter.convert(2); // 3
```

但是和匿名对象不同的是，这里的变量 `num` 可以不用声明为 `final`，该代码同样正确：

```
int num = 1;
```

不过这里的 `num` 必须不可被后面的代码修改（即隐性的具有 `final` 的语义），例如下面的就无法编译：

```
(from) -> String.valueOf(from + num);
```

```
num = 3;
```

在 lambda 表达式中试图修改 num 同样是不允许的。

七、访问对象字段与静态变量

和本地变量不同的是，lambda 内部对于实例的字段以及静态变量是即可读又可写。该行为和匿名对象是一致的：

```
class Lambda4 {
    static int outerStaticNum;
    int outerNum; ConverterStringConverter1 = (from) -> {
        outerNum = 23;
        return String.valueOf(from);
    }; StringConverter2 = (from) -> {
        outerStaticNum = 72;
        return String.valueOf(from);
    };
}
```

八、访问接口的默认方法

还记得第一节中的 formula 例子么，接口 Formula 定义了一个默认方法 sqrt 可以直接被 formula 的实例包括匿名对象访问到，但是在 lambda 表达式中这个是不行的。

Lambda 表达式中是无法访问到默认方法的，以下代码将无法编译：

复制代码代码如下：

```
Formula formula = (a) -> sqrt(a * 100);
```

Built-in Functional Interfaces

JDK 1.8 API 包含了很多内建的函数式接口，在老 Java 中常用到的比如 Comparator 或者 Runnable 接口，这些接口都增加了 @FunctionalInterface 注解以便能用在 lambda 上。

Java 8 API 同样还提供了很多全新的函数式接口来让工作更加方便，有一些接口是来自 Google Guava 库里的，即便你对这些很熟悉了，还是有必要看看这些是如何扩展到 lambda 上使用的。

Predicate 接口

Predicate 接口只有一个参数，返回 boolean 类型。该接口包含多种默认方法来将 Predicate 组合成其他复杂的逻辑（比如：与，或，非）：

```
Predicate predicate = (s) -> s.length > 0;
predicate.test("foo"); // true
predicate.negate().test("foo"); // false
isNotEmpty = isEmpty.negate();
```

Function 接口

Function 接口有一个参数并且返回一个结果，并附带了一些可以和其他函数组合的默认方法 (compose, andThen)：

```
Function<String, String> backToString = toInteger.andThen(String::valueOf);
backToString.apply("123"); // "123"
```

Supplier 接口

Supplier 接口返回一个任意范型的值，和 Function 接口不同的是该接口没有任何参数

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get(); // new Person
```

Consumer 接口

Consumer 接口表示执行在单个参数上的操作。

```
Consumer<Person> consumerGreeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

Comparator 接口

Comparator 是老 Java 中的经典接口，Java 8 在此之上添加了多种默认方法：

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);
Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");
comparator.compare(p1, p2); // > 0
comparator.reversed.compare(p1, p2); // < 0
```

Optional 接口

Optional 不是函数是接口，这是个用来防止 NullPointerException 异常的辅助类型，这是下一届中将要用到的重要概念，现在先简单的看看这个接口能干什么：

Optional 被定义为一个简单的容器，其值可能是 null 或者不是 null。在 Java 8 之前一般某个函数应该返回非空对象但是偶尔却可能返回了 null，而在 Java 8 中，不推荐你返回 null 而是返回 Optional。

```
Optional<String> optional = Optional.of("bam");
optional.isPresent(); // true
optional.get(); // "bam"
optional.orElse("fallback"); // "bam"
optional.ifPresent(s -> System.out.println(s.charAt(0))); // "b"
```

Stream 接口

java.util.Stream 表示能应用在一组元素上一次执行的操作序列。Stream 操作分为中间操作或者最终操作两种，最终操作返回一特定类型的计算结果，而中间操作返回 Stream 本身，这样你就可以将多个操作依次串起来。Stream 的创建需要指定一个数

据源，比如 `java.util.Collection` 的子类，`List` 或者 `Set`, `Map` 不支持。Stream 的操作可以串行执行或者并行执行。

```
List<String> stringCollection = new ArrayList<>;
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

Java 8 扩展了集合类，可以通过 `Collection.stream` 或者 `Collection.parallelStream` 来创建一个 Stream。下面几节将详细解释常用的 Stream 操作：

Filter 过滤

过滤通过一个 `predicate` 接口来过滤并只保留符合条件的元素，该操作属于中间操作，所以我们可以在过滤后的结果来应用其他 Stream 操作（比如 `forEach`）。`forEach` 需要一个函数来对过滤后的元素依次执行。`forEach` 是一个最终操作，所以我们不能在 `forEach` 之后来执行其他 Stream 操作。

```
StringCollection
    .stream
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);
// "aaa2", "aaa1"
```

Sort 排序

排序是一个中间操作，返回的是排序好后的 Stream。如果你不指定一个自定义的 `Comparator` 则会使用默认排序。

```
StringCollection
    .stream
    .sorted
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);
// "aaa1", "aaa2"
```

需要注意的是，排序只创建了一个排列好后的 Stream，而不会影响原有的数据源，排序之后原数据 `StringCollection` 是不会被修改的：

复制代码代码如下：

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Map 映射

中间操作 map 会将元素根据指定的 Function 接口来依次将元素转成另外的对象，下面的示例展示了将字符串转换为大写字符串。你也可以通过 map 来讲对象转换成其他类型，map 返回的 Stream 类型是根据你 map 传递进去的函数的返回值决定的。

```
stringCollection
.stream
.map(String::toUpperCase)
.sorted((a, b) -> b.compareTo(a))
.forEach(System.out::println);
// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Match 匹配

Stream 提供了多种匹配操作，允许检测指定的 Predicate 是否匹配整个 Stream。所有的匹配操作都是最终操作，并返回一个 boolean 类型的值。

```
boolean anyStartsWithA =
stringCollection
.stream
.anyMatch((s) -> s.startsWith("a"));
System.out.println(anyStartsWithA); // true
boolean allStartsWithA =
stringCollection
.stream
.allMatch((s) -> s.startsWith("a"));
System.out.println(allStartsWithA); // false
boolean noneStartsWithZ =
stringCollection
.stream
.noneMatch((s) -> s.startsWith("z"));
System.out.println(noneStartsWithZ); // true
```

Count 计数

计数是一个最终操作，返回 Stream 中元素的个数，返回值类型是 long。

```
long startsWithB =
stringCollection
.stream
.filter((s) -> s.startsWith("b"))
.count;
System.out.println(startsWithB); // 3
```

Reduce 规约

这是一个最终操作，允许通过指定的函数来讲 stream 中的多个元素规约为一个元素，规约后的结果是通过 Optional 接口表示的：

```
Optional<String> reduced =  
    stringCollection  
.stream  
.sorted  
.reduce((s1, s2) -> s1 + "#" + s2);  
reduced.ifPresent(System.out::println);  
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"  
并行 Streams
```

前面提到过 Stream 有串行和并行两种，串行 Stream 上的操作是在一个线程中依次完成，而并行 Stream 则是在多个线程上同时执行。

下面的例子展示了是如何通过并行 Stream 来提升性能：

```
int max = 1000000;  
List<String> values = new ArrayList<>(max);  
for (int i = 0; i < max; i++) {  
    UUID uuid = UUID.randomUUID();  
    values.add(uuid.toString());  
}
```

然后我们计算一下排序这个 Stream 要耗时多久，

串行排序：

```
long t0 = System.nanoTime();  
long count = values.stream().sorted().count();  
System.out.println(count);  
long t1 = System.nanoTime();  
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);  
System.out.println(String.format("sequential sort took: %d ms", millis));  
long count = values.parallelStream().sorted().count();  
System.out.println(count);
```

```
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);  
System.out.println(String.format("parallel sort took: %d ms", millis));  
// 并行排序耗时: 472 ms
```

上面两个代码几乎是一样的，但是并行版的快了 50% 之多，唯一需要做的改动就是将 stream 改为 parallelStream。

Map

前面提到过，Map 类型不支持 stream，不过 Map 提供了一些新的有用的方法来处理一些日常任务。

```
Mapmap = new HashMap<>;
for (int i = 0; i < 10; i++) {
    map.putIfAbsent(i, "val" + i);
}
map.forEach((id, val) -> System.out.println(val));
```

以上代码很容易理解，putIfAbsent 不需要我们做额外的存在性检查，而 forEach 则接收一个 Consumer 接口来对 map 里的每一个键值对进行操作。

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3); // val33
map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9); // false
map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23); // true
map.computeIfAbsent(3, num -> "bam");
map.get(3); // val33
```

接下来展示如何在 Map 里删除一个键值全都匹配的项：

```
map.remove(3, "val3");
map.get(3); // val33
map.remove(3, "val33");
map.get(3); // null
```

另外一个有用的方法：

复制代码代码如下：

```
map.getOrDefault(42, "not found"); // not found
```

对 Map 的元素做合并也变得很容易了：

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9); // val9
map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9); // val9concat
```

Merge 做的事情是如果键名不存在则插入，否则则对原键对应的值做合并操作并重新插入到 map 中。

九、Date API

Java 8 在包 java.time 下包含了一组全新的时间日期 API。新的日期 API 和开源的 Joda-Time 库差不多，但又不完全一样，下面的例子展示了这组新 API 里最重要的部分：

Clock 时钟

`Clock` 类提供了访问当前日期和时间的方法，`Clock` 是时区敏感的，可以用来取代 `System.currentTimeMillis` 来获取当前的微秒数。某一个特定的时间点也可以使用 `Instant` 类来表示，`Instant` 类也可以用来创建老的 `java.util.Date` 对象。

```
Clock clock = Clock.systemDefaultZone;
long millis = clock.millis;
Instant instant = clock.instant;
Date legacyDate = Date.from(instant); // legacy java.util.Date
```

Timezones 时区

在新 API 中时区使用 `ZoneId` 来表示。时区可以很方便的使用静态方法 `of` 来获取到。时区定义了到 UTS 时间的时间差，在 `Instant` 时间点对象到本地日期对象之间转换的时候是极其重要的。

```
System.out.println(ZoneId.getAvailableZoneIds);
// prints all available timezone ids
ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules);
System.out.println(zone2.getRules);
// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

LocalTime 本地时间

`LocalTime` 定义了一个没有时区信息的时间，例如晚上 10 点，或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。之后比较时间并以小时和分钟为单位计算两个时间的时间差：

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);
System.out.println(now1.isBefore(now2)); // false
long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);
System.out.println(hoursBetween); // -3
System.out.println(minutesBetween); // -239
```

`LocalTime` 提供了多种工厂方法来简化对象的创建，包括解析时间字符串。

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late); // 23:59:59
DateTimeFormatter germanFormatter =
    DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)
        .withLocale(Locale.GERMAN);
```

```
LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime); // 13:37
```

LocalDate 本地日期

LocalDate 表示了一个确切的日期，比如 2014-03-11。该对象值是不可变的，用起来和 LocalTime 基本一致。下面的例子展示了如何给 Date 对象加减天/月/年。另外要注意的是这些对象是不可变的，操作返回的总是一个新实例。

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);
LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek); // FRIDAY
```

从字符串解析一个 LocalDate 类型和解析 LocalTime 一样简单：

```
DateTimeFormatter germanFormatter =
DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)
.withLocale(Locale.GERMAN);
LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
System.out.println(xmas); // 2014-12-24
```

LocalDateTime 本地日期时间

LocalDateTime 同时表示了时间和日期，相当于前两节内容合并到一个对象上了。

LocalDateTime 和 LocalDate 一样，都是不可变的。LocalDateTime 提供了一些能访问具体字段的方法。

```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER,
31, 23, 59, 59);
```

```
DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek); // WEDNESDAY
Month month = sylvester.getMonth();
System.out.println(month); // DECEMBER
long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay); // 1439
```

只要附加上时区信息，就可以将其转换为一个时间点 Instant 对象，Instant 时间点对象可以很容易的转换为老式的 java.util.Date。

```
Instant instant = sylvester
.atZone(ZoneId.systemDefault())
.toInstant();
Date legacyDate = Date.from(instant);
```

```
System.out.println(legacyDate); // Wed Dec 31 23:59:59 CET 2014
```

格式化 `LocalDateTime` 和格式化时间和日期一样的，除了使用预定义好的格式外，我们也可以自己定义格式：

```
DateTimeFormatter formatter =
DateTimeFormatter
.ofPattern("MMM dd, yyyy - HH:mm");
LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13",
formatter);
String string = formatter.format(parsed);
System.out.println(string); // Nov 03, 2014 - 07:13
```

和 `java.text.NumberFormat` 不一样的是新版的 `DateTimeFormatter` 是不可变的，所以它是线程安全的。

关于时间日期格式的详细信息：<http://download.java.net/jdk8/docs/api/java/time/format/>

十、Annotation 注解

在 Java 8 中支持多重注解了，先看个例子来理解一下是什么意思。

首先定义一个包装类 `Hints` 注解用来放置一组具体的 `Hint` 注解：

```
@interface Hints {
    Hint value;
}
@Repeatable(Hints.class)
@interface Hint {
    String value;
}
```

Java 8 允许我们把同一个类型的注解使用多次，只需要给该注解标注一下 `@Repeatable` 即可。

例 1：使用包装类当容器来存多个注解（老方法）

复制代码代码如下：

```
@Hints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

例 2：使用多重注解（新方法）

复制代码代码如下：

```
@Hint("hint1")
@Hint("hint2")
class Person {}
```

第二个例子里 java 编译器会隐性的帮你定义好 `@Hints` 注解，了解这一点有助于你用反射来获取这些信息：

```
Hint hint = Person.class.getAnnotation(Hint.class);
```

```
System.out.println(hint); // null
Hints hints1 = Person.class.getAnnotation(Hints.class);
System.out.println(hints1.value.length); // 2
Hint hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length); // 2
```

即便我们没有在 Person 类上定义 @Hints 注解，我们还是可以通过 getAnnotation(Hints.class) 来获取 @Hints 注解，更加方便的方法是使用 getAnnotationsByType 可以直接获取到所有的 @Hint 注解。

另外 Java 8 的注解还增加到两种新的 target 上了：

复制代码代码如下：

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

关于 Java 8 的新特性就写到这了，肯定还有更多的特性等待发掘。JDK 1.8 里还有很多很有用的东西，比如 Arrays.parallelSort, StampedLock 和 CompletableFuture 等等。

附录 E 部分练习和思考题参考答案

E.1 初识 Java

解答 1. 练习1.4[在24页] 的一个实现如下所示：

```
1 public class KDW {  
2  
3     public static void main(String[] args) {  
4         System.out.println("** *** ***** * * * * *");  
5         System.out.println("** *** ** * * *** * * ");  
6         System.out.println("** *** ** * * * * * * *");  
7         System.out.println("** *** ** * * * * * * *");  
8         System.out.println("***** * * * * * * * * * *");  
9         System.out.println("** *** ** * * * * * * * *");  
10        System.out.println("** *** ** * * * * * * *");  
11        System.out.println("** *** ** * * *** * * *");  
12        System.out.println("** *** ***** * * * *");  
13    }  
14  
15 }
```

E.2 面向对象编程基础

解答 2. default 修饰的类，在其他包中不可见。

E.3 ??

解答 3. 计算两个整数的和、差、积的 lambda 表达式示例如下：

```
1 public class Calculator {  
2  
3     interface IntegerMath {  
4         int operation(int a, int b);  
5     }  
6 }
```

```

7   public int operateBinary(int a, int b, IntegerMath op) {
8       return op.operation(a, b);
9   }
10
11  public static void main(String... args) {
12
13      Calculator myApp = new Calculator();
14      IntegerMath addition = (a, b) -> a + b;
15      IntegerMath subtraction = (a, b) -> a - b;
16      IntegerMath multiply = (a, b) -> a * b;
17      System.out.println("40 + 2 = " +
18          myApp.operateBinary(40, 2, addition));
19      System.out.println("20 - 10 = " +
20          myApp.operateBinary(20, 10, subtraction));
21      System.out.println("20 * 2 = " +
22          myApp.operateBinary(20, 2, multiply));
23  }
24 }
```

E.4 Enum

解答 4. Planet 的定义如下¹:

```

1  public enum Planet {
2      MERCURY (3.303e+23, 2.4397e6),
3      VENUS (4.869e+24, 6.0518e6),
4      EARTH (5.976e+24, 6.37814e6),
5      MARS (6.421e+23, 3.3972e6),
6      JUPITER (1.9e+27, 7.1492e7),
7      SATURN (5.688e+26, 6.0268e7),
8      URANUS (8.686e+25, 2.5559e7),
9      NEPTUNE (1.024e+26, 2.4746e7);
10
11     private final double mass; // in kilograms
12     private final double radius; // in meters
13     Planet(double mass, double radius) {
14         this.mass = mass;
15         this.radius = radius;
16     }
17     private double mass() { return mass; }
18     private double radius() { return radius; }
19
20     // universal gravitational constant (m3 kg-1 s-2)
21     public static final double G = 6.67300E-11;
22
23     double surfaceGravity() {
24         return G * mass / (radius * radius);
```

¹ 参考: <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

```
25     }
26     double surfaceWeight(double otherMass) {
27         return otherMass * surfaceGravity();
28     }
29     public static void main(String[] args) {
30         if (args.length != 1) {
31             System.err.println("Usage: java Planet <earth_weight>");
32             System.exit(-1);
33         }
34         double earthWeight = Double.parseDouble(args[0]);
35         double mass = earthWeight/EARTH.surfaceGravity();
36         for (Planet p : Planet.values())
37             System.out.printf("Your weight on %s is %f%n",
38                               p, p.surfaceWeight(mass));
39     }
40 }
```

附录 F Jar 包和 CLASSPATH

F.1 jar 包概念和用法

F.2 CLASSPATH

CLASSPATH 环境变量是 Java 查询 class 文件的路径，也就是说，Java 虚拟机在运行时依靠 CLASSPATH 环境变量来决定从哪里加载所需要的 class 文件。

F.2.1 CLASSPATH 的默认值

CLASSPATH 的默认值是“.”，即当前目录。也就是说，Java 虚拟机从只从当前目录加载所需要的的 class 文件。

下面我们分别研究两种情况：

例 F.1. 执行默认包的 Java 应用程序

此种情形是初学者最容易理解的情形，即我们的 Java 应用程序没有定义包。比如我们有如下的目录结构：

```
lesson
└── Test.java
```

即在目录 lesson 下只有一个 Java 源文件 Test.java。Test.java 文件定义如下：

```
1 public class Test{
2     public static void main(String[] args) {
3         System.out.println("hello world!");
4     }
5 }
```

于是我们可以在 lesson 目录下执行如下的操作编译和执行 Test 应用程序：

```
~/lesson$ javac Test.java  
~/lesson$ java Test  
hello world!
```

java Test 的意思即 Java 虚拟机查找一个叫做 Test 的类文件并尝试运行 Test.class 的 main 方法。如果我们没有修改过 CLASSPATH 环境变量，则默认的 CLASSPATH 环境变量值是“.”，即 Java 虚拟机从当前目录查找 Test.class，自然在这种情形下是成功的，因此打印出了“hello world!”。

例 F.2. 执行使用包的 Java 应用程序

假设我们在 lesson 目录下创建如图 F.1 的目录结构：

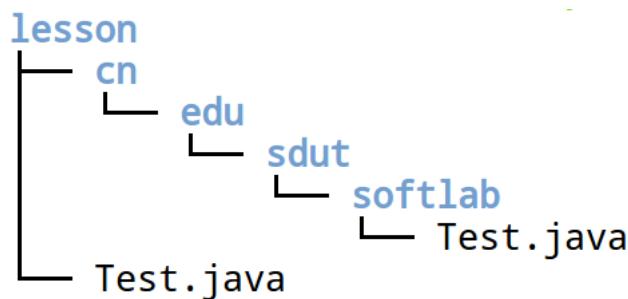


图 F.1：包的目录结构

其中 cn/edu/sdut/softlab/Test.java 内容如下：

```
1 package cn.edu.sdut.softlab;  
2 public class Test{  
3     public static void main(String[] args) {  
4         System.out.println("hello world!");  
5     }  
6 }
```

那么我们该如何编译和执行使用了包的 Test 应用程序呢？首先我们看如何编译 Test.java。在任何目录下都可以正常编译 Test.java，比如在 lesson 目录下编译，也可以到 lesson/cn/edu/sdut/softlab 目录下编译，也可以在 lesson/cn/edu/sdut 目录下编译：

```
cd ~/lesson  
javac cn/edu/sdut/softlab/Test.java  
cd ~/lesson/cn/edu/sdut/softlab  
javac Test.java  
cd ~/lesson/cn/edu/sdut  
javac softlab/Test.java
```

编译的结果都是在 lesson/cn/edu/sdut/softlab 目录下生成了 Test.class 文件，那么如何执行这个 Test.class 文件呢？我们尝试一下：

```
cd lesson  
~/lesson$ javac cn/edu/sdut/softlab/Test.java  
~/lesson$ java cn.edu.sdut.softlab.Test  
~/lesson$ cd cn/edu/sdut/softlab/  
~/lesson/cn/edu/sdut/softlab$ java Test  
错误: 找不到或无法加载主类 Test  
~/lesson/cn/edu/sdut/softlab$ java cn.edu.sdut.softlab.Test  
错误: 找不到或无法加载主类 cn.edu.sdut.softlab.Test  
~/lesson/cn/edu/sdut/softlab$ cd ~/lesson/cn/edu/sdut/  
~/lesson/cn/edu/sdut$ java Test  
错误: 找不到或无法加载主类 Test  
~/lesson/cn/edu/sdut$ java cn.edu.sdut.softlab.Test  
错误: 找不到或无法加载主类 cn.edu.sdut.softlab.Test
```

可以看出，只能在 lesson 目录下执行 cn.edu.sdut.soft.Test 应用程序，这是因为默认的 CLASSPATH 是当前目录，则 Java 虚拟机从当前目录开始寻找 cn.edu.sdut.softlab.Test.class 文件，即寻找 ./cn/edu/sdut/softlab/Test.class 文件，自然是可以找到的。请自行分析为什么上面其他情况下无法正确执行 Test 应用程序。

那么我们如何保证在任何目录下都能够正确执行 Test 应用程序呢？实际上，只要告知 Java 虚拟机包 cn.edu.sdut.softlab 的起始目录，Java 虚拟机就可以根据这个起始目录查找到这个包下面的 class 文件。这个起始目录可以写到 CLASSPATH，也可以在命令行使用 -cp 明确指定当前应该使用什么样的 CLASSPATH：

```
cd /opt  
/opt$ java -cp ~/lesson cn.edu.sdut.softlab.Test  
hello world!
```

设置 CLASSPATH 的情形如下：

```
/opt$ export CLASSPATH=$CLASSPATH:~/lesson  
/opt$ java cn.edu.sdut.softlab.Test  
hello world!
```

具体是设置 CLASSPATH 还是使用命令行的 -cp 参数要视具体情况而定。一般情况下，如果只是临时执行一个 Java 应用程序，则使用命令行参数 -cp 即可。如果一个 Java 应用程序要多次调用或者作为工具类提供给其他应用程序使用，则设置 CLASSPATH 比较合适。命令行参数 -cp 类似于局部变量，环境变量 CLASSPATH 类似于全局变量。

更多 CLASSPATH 知识请参考：<https://docs.oracle.com/javase/8/docs/technotes/tools/windows/classpath.html>, <http://docs.oracle.com/javase/8/docs/technotes/tools/findingclasses.html>

F.3 war/ear 包

附录 G Idea 的常用快捷键

完整的 Idea 的默认快捷键可从这里下载: https://resources.jetbrains.com/storage/products/idea/docs/IntelliJIDEA_ReferenceCard.pdf, 建议大家打印出来置于案头以便参考。其中最常用的快捷键列表如下:

Ctrl+Space: 智能提示

Alt+Enter: 快速修复

Shift+F6: 改名字

Alt+Insert: 代码插入, 自动生成代码

Ctrl+W/Ctrl+Shift+W: 自动按照语法选中代码

Shift+Shift: 弹出万能搜索框

Ctrl+N/Ctrl+Shift+N: 搜索并打开指定文件/类

Ctrl+Alt+L: 格式化代码

Ctrl+Alt+O: 重新组织化 import

Alt+Shift+F10: 运行程序

F7/F8/F9: 调试时对应 Step into/Step over/Continue

附录 H Google 编码规范

H.1 介绍

本文档为 Google Java 编程规范的完整定义。依照此规范编写的 Java 源码文件可以被称为 **Google Style**。

和其他编程规范指南一样，规范不仅包括了代码的结构美学，也包括了其他一些业界约定俗成的公约和普遍采用的标准。本文档中的规范基本都是业界已经达成共识的标准，我们尽量避免去定义那些还存在争议的地方。

H.1.1 术语说明

本文档除非特殊说明，否则：

1. **class**（类）统指普通的 `class` 类型、`enum` 枚举类型、`interface` 类型和 `annotation` 类型。
2. **comment**（注释）总是指 `implementation comments`（实现注释，`/* */`）。我们不使用“文档注释”这样的说法，而会直接说 `javadoc`。

其他术语说明，将在文档中需要说明的地方单独说明。

H.1.2 1.2 文档说明

本文档中的代码并不一定符合所有规范。即使这些代码遵循 **Google Style**，但这不是唯一的代码规范。例子中可选的格式风格也不应该作为强制执行的规范。

H.2 源码文件基础

H.2.1 文件名

源码文件名由它所包含的顶级 `class` 的类名（大小写敏感），加上 `.java` 后缀组成。（除了 `package-info.java` 文件）。

H.2.2 文件编码: UTF-8

源码文件使用 UTF-8 编码。

H.2.3 特殊字符

H.2.3.1 2.3.1 空格字符

除了换行符外, ASCII 水平空白字符 (0x20) 是源码文件中唯一支持的空格字符。这意味着:

- 其他空白字符将被转义。
- Tab 字符不被用作缩进控制。

H.2.3.2 2 特殊转义字符串

任何需要转义字符串表示的字符 (例如 \b, \t, \n, \f, \r, \', \\等), 采用这种转义字符串的方式表示, 而不采用对应字符的八进制数 (例如 \012) 或 Unicode 码 (例如 \u000a) 表示。

H.2.3.3 非 ASCII 字符

对于其余非 ASCII 字符, 直接使用 Unicode 字符 (例如 ∞), 或者使用对应的 Unicode 码 (例如 \u221e) 转义, 都是允许的。唯一需要考虑的是, 何种方式更能使代码容易阅读和理解。

注意: 在使用 `unicode` 码转义, 或者甚至是直接使用 `unicode` 字符的时候, 添加一点说明注释将对别人读懂代码很有帮助。

例子:

```

1 String unitAbbrev = "\u03bc"; Best: perfectly clear even without a comment.
2 String unitAbbrev = "\u03bcs"; // "μs" Allowed, but there's no reason to do this.
3 String unitAbbrev = "\u03bcs"; // Greek letter mu, "s" Allowed, but awkward and prone
   to mistakes.
4 String unitAbbrev = "\u03bcs"; Poor: the reader has no idea what this is.
5 return '\ufeff' + content; // byte order mark Good: use escapes for non-printable
   characters, and comment if necessary.

```

注意: 不要因为担心一些程序无法正常处理非 ASCII 字符而不使用它, 从而导致代码易读性变差。如果出现这样的问题, 应该由出现问题的程序去解决。

H.3 源码文件结构

源码文件按照先后顺序，由以下几部分组成：

- License 或者 copyright 声明信息。（如果需要声明）
- 包声明语句。
- import 语句。
- class 类声明（每个源码文件只能有唯一一个顶级 class）。

每个部分之间应该只有一行空行作为间隔。

H.3.1 lincense 或者 copyright 的声明信息。

如果需要声明 lincense 或 copyright 信息，应该在文件开始时声明。

H.3.2 包声明

包声明的行，没有行长度的限制。单行长度限制（4.4 部分有详细说明，80 或 100）不适用于包声明。

H.3.3 import 语句

H.3.3.1 不使用通配符 import

不应该使用通配符 import，不管是否是静态导入。

H.3.3.2 没有行长度限制

import 语句的行，没有行长度的限制。单行长度限制（H.4.4 有详细说明，80 或 100）不适用于 import 语句所在行。

H.3.3.3 顺序和空行

import 语句应该被分为几个组，每个组之间由单行的空行隔开。分组的顺序如下：

1. 所有的 static import 为归为一组。
2. com.google 包的 import 归为一组。
3. 使用的第三方包的引用。每个顶级第三方包归为一组。第三方包之间按 ASCII 码排序。例如： android, com, junit, org, sun

4. `java` 包归为一组。

5. `javax` 包归为一组。

同一组内的 `import` 语句之间不应用空行隔开。同一组中的 `import` 语句按 ASCII 码排序。

H.3.4 类声明

H.3.4.1 只声明唯一一个顶级 `class`

每个源码文件中只能有一个顶级 `class`。`package-info.java` 文件除外。

H.3.4.2 类成员顺序

类成员的顺序对代码的易读性有很大影响，但是没有一个统一正确的标准。不同的类可能有不同的排序方式。

重要的是，每个 `class` 都要按照一定的逻辑规律排序。当被问及时，能够解释清楚为什么这样排序。例如，新增加的成员方法，不是简单地放在 `class` 代码最后面，按日期排序也不是按逻辑排序。

H.3.4.3 重载方法：不应该分开

当一个类有多个构造函数，或者多个同名成员方法时，这些函数应该写在一起，不应该被其他成员分开。

H.4 格式

术语说明：块状结构（block-like construct）指类、成员函数和构造函数的实现部分（花括号中间部分）。注意，在后面的 4.8.3.1 节中讲到数组初始化，所有的数组初始化都可以被认为是一个块状结构（非强制）。

H.4.1 花括号

H.4.1.1 花括号在需要的地方使用

花括号一般用在 `if`, `else`, `for`, `do`, 和 `while` 等语句。甚至当它的实现为空或者只有一句话时，也需要使用。

H.4.1.2 非空语句块采用 K&R 风格

对于非空语句块，花括号遵循 K&R 风格：

- 左括号前不换行。
- 左括号后换行。
- 右括号前换行。
- 如果右括号结束一个语句块或者函数体、构造函数体或者有命名的类体，则需要换行。例如，当右括号后面接 `else` 或者逗号时，不应该换行。

例子：

```
1 return new MyClass() {
2     @Override public void method() {
3         if(condition()) {
4             try {
5                 something();
6             } catch (ProblemException e) {
7                 recover();
8             }
9         }
10    }
11 };
```

一些例外的情况，将在 4.8.1 节讲枚举类型的时候讲到。

H.4.1.3 空语句块：使代码更简洁

一个空的语句块，可以在左花括号之后直接接右花括号，中间不需要空格或换行。但是当一个由几个语句块联合组成的语句块时，则需要换行。（例如：`if/else-if/else try/catch/finally`）。

例子：

```
1 void doNothing() {}
```

H.4.2 语句块的缩进：2 空格

每当一个新的语句块产生，缩进就增加两个空格。当这个语句块结束时，缩进恢复到上一层级的缩进格数。缩进要求对整个语句块中的代码和注释都适用。（例子可参考之前 4.1.2 节中的例子）。

H.4.3 一行最多只有一句代码

每句代码的结束都需要换行。

H.4.4 行长度限制: 80 或 100

不同的项目可以选择采用 80 个字符或者 100 个字符作为限制。除了以下几个特殊情况外，其他代码内容都需要遵守这个长度限制。这在H.4.5会有详细解释。

例外：

- 按照行长度限制，无法实现地方（例如：javadoc 中超长的 URL 地址，或者一个超长的 JSNI 方法的引用）；
- package 和 import 语句不受长度限制。（见H.3.2、H.3.3）；
- 注释中的命令行指令行，将被直接复制到 shell 中执行的。

H.4.5 长行断行

术语说明：当一行代码按照其他规范都合法，只是为了避免超出行长度限制而换行时，称为长行断行。

长行断行，没有一个适合所有场景的全面、确定的规范。但很多相同的情况，我们经常使用一些行之有效的断行方法。

注意：将长行封装为函数，或者使用局部变量的方法，也可以解决一些超出行长度限制的情况。并非一定要断行。

H.4.5.1 在何处断行

断行的主要原则是：选择在更高一级的语法逻辑的地方断行。其他一些原则如下：

- 当一个非赋值运算的语句断行时，在运算符号之前断行。（这与 Google 的 C++ 规范和 JavaScript 规范等其他规范不同）。
- 当一个赋值运算语句断行时，一般在赋值符号之后断行。但是也可以在之前断行。
- 在调用函数或者构造函数需要断行时，与函数名相连的左括号要在一行。也就是在左括号之后断行。
- 逗号断行时，要和逗号隔开的前面的语句断行。也就是在逗号之后断行。

H.4.5.2 断行的缩进：至少 4 个字符

当断行之后，在第一行之后的行，我们叫做延续行。每一个延续行在第一行的基础上至少缩进四个字符。

当原行之后有多个延续行的情况，缩进可以大于 4 个字符。如果多个延续行之间由同样的语法元素断行，它们可以采用相同的缩进。

4.6.3 节介绍水平对齐中，解决了使用多个空格与之前行缩进对齐的问题。

H.4.6 空白空间

H.4.6.1 4.6.1 垂直空白

单行空行在以下情况使用：

- 类成员间需要空行隔开：例如成员变量、构造函数、成员函数、内部类、静态初始化语句块（static initializers）、实例初始化语句块（instance initializers）。
- 例外：成员变量之间的空白行不是必需的。一般多个成员变量中间的空行，是为了对成员变量做逻辑上的分组。
- 在函数内部，根据代码逻辑分组的需要，设置空白行作为间隔。
- 类的第一个成员之前，或者最后一个成员结束之后，用空行间隔。（可选）
- 本文档中其他部分介绍的需要空行的情况。（例如 H.3.3 中的 import 语句）

单空行时使用多行空行是允许的，但是不要求也不鼓励。

H.4.6.2 水平空白

除了语法、其他规则、词语分隔、注释和 javadoc 外，水平的 ASCII 空格只在以下情况出现：

- 所有保留的关键字与紧接它之后的位于同一行的左括号之间需要用空格隔开。（例如 if、for、catch）
- 所有保留的关键字与在它之前的右花括号之间需要空格隔开。（例如 else、catch）
- 在左花括号之前都需要空格隔开。只有两种例外：

```
1 @SomeAnnotation({a, b});  
2 String[][] x = {{"foo"}};
```

- 所有的二元运算符和三元运算符的两边，都需要空格隔开。

- 逗号、冒号、分号和右括号之后，需要空格隔开。
- `//` 双斜线开始一行注释时。双斜线两边都应该用空格隔开。并且可使用多个空格，但是不做强制要求。
- 变量声明时，变量类型和变量名之间需要用空格隔开。
- 初始化一个数组时，花括号之间可以用空格隔开，也可以不使用。（例如：`new int[] {5, 6}` 和 `new int[] { 5, 6 }` 都可以）

注意：这一原则不影响一行开始或者结束时的空格。只针对行内部字符之间的隔开。

H.4.6.3 水平对齐：不做强制要求

术语说明：水平对齐，是指通过添加多个空格，使本行的某一符号与上一行的某一符号上下对齐。

这种对齐是被允许的，但是不会做强制要求。

以下是没有水平对齐和水平对齐的例子；

```

1 private int x; // this is fine
2 private Color color; // this too
3 private int x; // permitted, but future edits
4 private Color color; // may leave it unaligned

```

注意：水平对齐能够增加代码的可读性，但是增加了未来维护代码的难度。考虑到维护时只需要改变一行代码，之前的对齐可以不需要改动。为了对齐，你更有可能改了一行代码，同时需要更改附近的好几行代码，而这几行代码的改动，可能又会引起一些为了保持对齐的代码改动。那原本这行改动，我们称之为“爆炸半径”。这种改动，在最坏的情况下可能会导致大量的无意义的工作，即使在最好的情况下，也会影响版本历史信息，减慢代码 review 的速度，引起更多 merge 代码冲突的情况。

H.4.7 分组括号：建议使用

非必须的分组括号只有在编写代码者和代码审核者都认为大家不会因为没有它而导致代码理解错误的时候，或者它不会使代码更易理解的时候才能省略。没有理由认为所有阅读代码的人都能记住所有 java 运算符的优先级。

H.4.8 特殊结构

H.4.8.1 枚举类型

每个逗号后接一个枚举变量，不要求换行。

枚举类型，如果没有函数和 javadoc，处理格式是可以按照数组初始化来处理。

例子：

```
1 private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

枚举类型也是一种类 (Class)，因此 Class 类的其他格式要求，也适用于枚举类型。

H.4.8.2 4.8.2 变量声明

4.8.2.1 每次声明一个变量 不要采用一个声明，声明多个变量。例如 int a, b;

4.8.2.2 当需要时才声明，尽快完成初始化 局部变量不应该习惯性地放在语句块的开始处声明，而应该尽量离它第一次使用的地方最近的地方声明，以减小它们的使用范围。

局部变量应该在声明的时候就进行初始化。如果不能在声明时初始化，也应该尽快完成初始化。

H.4.8.3 4.8.3 数组

4.8.3.1 数组初始化：可以类似块代码处理 所有数组的初始化，都可以采用和块代码相同的格式处理。例如以下格式都是允许的：

```
1 new int[] {
2   0,1,2,3
3 }
4
5 new int[] {
6   0,1,
7   2,3
8 }
9
10 new int[] {
11   0,
12   1,
13   2,
14   3
15 }
16
17 new int[] {0,1,2,3}
```

4.8.3.2 不能像 C 风格一样声明数组 方括号应该是变量类型的一部分，因此不应该和变量名放在一起。例如：应该是 String[] args，而不是 String args[]。

H.4.8.4 switch 语句

术语说明：switch 语句是指在 switch 花括号中，包含了一组或多组语句块。每组语句块都由一个或多个 switch 标签（例如 case FOO: 或者 default:）打头。

4.8.4.1 缩进 和其他语句块一样，switch 花括号之后缩进两个字符。

每个 switch 标签之后，后面紧接的非标签的新行，按照花括号相同的处理方式缩进两个字符。在标签结束后，恢复到之前的缩进，类似花括号结束。

4.8.4.2 继续向下执行的注释 在 switch 语句中，每个标签对应的代码执行完后，都应该通过语句结束（例如：break、continue、return 或抛出异常），否则应该通过注释说明，代码需要继续向下执行下一个标签的代码。注释说明文字只要能说明代码需要继续往下执行都可以（通常是 //fall through）。这个注释在最后一个标签之后不需要注释。例如：

```

1 switch (input) {
2     case 1:
3     case 2:
4         prepareOneOrTwo();
5         //fall through
6     case 3:
7         handleOneTwoOrThree();
8         break;
9     default:
10        handleLargeNumber(input);
11 }
```

4.8.4.3 default 标签需要显式声明 每个 switch 语句中，都需要显式声明 default 标签。即使没有任何代码也需要显示声明。

H.4.8.5 Annotations

Annotations 应用到类、函数或者构造函数时，应紧接 javadoc 之后。每一行只有一个 Annotations。

Annotations 所在行不受行长度限制，也不需要增加缩进。例如：

```

1 @Override
2 @Nullable
3 public String getNameIfPresent() { ... }
```

例外情况：

如果 Annotations 只有一个，并且不带参数。则它可以和类或方法名放在同一行。例如：

```
1 @Override public int hashCode() { ... }
```

Annotations 应用到成员变量时，也是紧接 javadoc 之后。不同的是，多个 annotations 可以放在同一行。例如：

```
1 @Partial @Mock DataLoader loader;
```

对于参数或者局部变量使用 Annotations 的情况，没有特定的规范。

H.4.8.6 注释

4.8.6.1 语句块的注释风格 注释的缩进与它所注释的代码缩进相同。可以采用 `/* */` 进行注释，也可以用 `//` 进行注释。当使用 `/**/` 进行多行注释时，每一行都应该以 `*` 开始，并且 `*` 应该上下对齐。

例如：

```
1 /*
2  * this is okay.
3  */
4
5 // Also is this
6 // these
7
8 /* Or you can
9  * even do this. */
```

多行注释时，如果你希望集成开发环境能自动对齐注释，你应该使用 `/**/`，`//` 一般不会自动对齐。

H.4.8.7 4.8.7 修饰符

多个类和成员变量的修饰符，按 Java Language Specification 中介绍的先后顺序排序。具体是：

`public protected private abstract static final transient volatile synchronized native strictfp`

H.5 命名

H.5.1 适用于所有命名标识符的通用规范

标示符只应该使用 ASCII 字母、数字和下划线，字母大小写敏感。因此所有的标示符，都应该能匹配正则表达式 `\w+`。

Google Style 中，标示符不需要使用特殊的前缀或后缀，例如：name_，mName，s_name 和 kName。

H.5.2 不同类型的标示符规范

H.5.2.1 包名

包名全部用小写字母，通过. 将各级连在一起。不应该使用下划线。

H.5.2.2 类名

类型的命名，采用以大写字母开头的大小写字符间隔的方式 (UpperCamelCase)。

class 命名一般使用名词或名词短语。interface 的命名有时也可以使用形容词或形容词短语。annotation 没有明确固定的规范。

测试类的命名，应该以它所测试的类的名字为开头，并在最后加上 Test 结尾。例如：HashTest、HashIntegrationTest。

H.5.2.3 方法名

方法命名，采用以小写字母开头的大小写字符间隔的方式 (lowerCamelCase)。

方法命名一般使用动词或者动词短语。

在 JUnit 的测试方法中，可以使用下划线，用来区分测试逻辑的名字，经常使用如下的结构：test<MethodUnderTest>_<state>。例如：testPop_emptyStack。

测试方法也可以用其他方式进行命名。

H.5.2.4 常量名

常量命名，全部使用大写字符，词与词之间用下划线隔开。(CONSTANCE_CASE)。

常量是一个静态成员变量，但不是所有的静态成员变量都是常量。在选择使用常量命名规则给变量命名时，你需要明确这个变量是否是常量。例如，如果这个变量的状态可以发生改变，那么这个变量几乎可以肯定不是常量。只是计划不会发生改变的变量不足以成为一个常量。下面是常量和非常量的例子：

```

1 // Constants
2 static final int NUMBER = 5;
3 static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
4 static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is immutable
5 static final SomeMutableType[] EMPTY_ARRAY = {};
6 enum SomeEnum { ENUM_CONSTANT }
7
8 // Not constants
9 static String nonFinal = "non-final";

```

```
10 final String nonStatic = "non-static";
11 static final Set<String> mutableCollection = new HashSet<String>();
12 static final ImmutableSet<SomeMutableType> mutableElements = ImmutableSet.of(mutable);
13 static final Logger logger = Logger.getLogger(MyClass.getName());
14 static final String[] nonEmptyArray = {"these", "can", "change"};
```

常量一般使用名词或者名词短语命名。

H.5.2.5 非常量的成员变量名

非常量的成员变量命名（包括静态变量和非静态变量），采用 lowerCamelCase 命名。

一般使用名词或名词短语。

H.5.2.6 参数名

参数命名采用 lowerCamelCase 命名。

应该避免使用一个字符作为参数的命名方式。

H.5.2.7 局部变量名

局部变量采用 lowerCamelCase 命名。它相对于其他类型的命名，可以采用更简短宽松的方式。

但即使如此，也应该尽量避免采用单个字母进行命名的情况，除了在循环体内使用的临时变量。

即使局部变量是 final、不可改变的，它也不能被认为是常量，也不应该采用常量的命名方式去命名。

H.5.2.8 类型名

类型名有两种命名方式：

- 单独一个大写字母，有时后面再跟一个数字。（例如，E、T、X、T2）。
- 像一般的 class 命名一样（见 5.2.2 节），再在最后接一个大写字母。（例如，RequestT、FooBarT）。

H.5.3 Camel case 的定义

有时一些短语被写成 Camel case 的时候可以有多种写法。例如一些缩写词汇，或者一些组合词：IPv6 或者 iOS 等。

为了统一写法，Google style 给出了一种几乎可以确定为一种的写法。

- 将字符全部转换为 ASCII 字符，并且去掉' 等符号。例如，“Müller’s algorithm” 被转换为 “Muellers algorithm”。
- 将上一步转换的结果拆分成一个一个的词语。从空格处和从其他剩下的标点符号处划分。
注意：一些已经是 Camel case 的词语，也应该在这个时候被拆分。（例如 Ad-Words 被拆分为 ad words）。但是例如 iOS 之类的词语，它其实不是一个 Camel case 的词语，而是人们惯例使用的一个词语，因此不用做拆分。
- 经过上面两部后，先将所有的字母转换为小写，再把每个词语的第一个字母转换为大写。
- 最后，将所有词语连在一起，形成一个标示符。

注意：词语原来的大小写规则，应该被完全忽略。以下是一些例子：

| Prose form | Correct | Incorrect |
|----------------------|-----------------|------------------|
| “XML HTTP request” | XmlHttpRequest | XMLHTTPRequest |
| “new customer ID” | newCustomerId | newCustomerID |
| “inner stopwatch” | innerStopwatch | innerStopWatch |
| “supports IPv6 iOS?” | supportsIPv6Ios | supportIPv6OnIOS |
| “YouTube importer” | YouTubeImporter | YoutubeImporter* |

* 号表示可以接受，但是不建议使用。

注意，有些词语在英文中，可以用 - 连接使用，也可以不使用 - 直接使用。例如 “nonempty” 和 “non-empty” 都是可以的。因此，方法名字为 checkNonempty 或者 checkNonEmpty 都是可以的。

H.6 编程实践

H.6.1 @override 都应该使用

@override annotations 只要是符合语法的，都应该使用。

H.6.2 异常捕获不应该被忽略

一般情况下，catch 住的异常不应该被忽略，而是都需要做适当的处理。例如将错误日志打印出来，或者如果认为这种异常不会发生，则应该作为断言异常重新抛出。

如果这个 catch 住的异常确实不需要任何处理，也应该通过注释做出说明。例如：

```
1 try {
2     int i = Integer.parseInt(response);
3     return handleNumericReponse(i);
4 } catch (NumberFormatException ok) {
5     // it's not numeric; that's fine, just continue
6 }
7
8 return handleTextResponse(response);
```

例外：在测试类里，有时会针对方法是否会抛出指定的异常，这样的异常是可以被忽略的。但是这个异常通常需要命名为：expected。例如：

```
1 try {
2     emptyStack.pop();
3     fail();
4 } catch (NoSuchElementException expected) {
5 }
```

H.6.3 静态成员的访问：应该通过类，而不是对象

当一个静态成员被访问时，应该通过 class 名去访问，而不应该使用这个 class 的具体实例对象。例如：

```
1 Foo aFoo = ...;
2 Foo.aStaticMethod(); // good
3 aFoo.aStaticMethod(); // bad
4 somethingThatYieldAFoo().astaticMethod(); very bad
```

H.6.4 不使用 Finalizers 方法

重载 Object 的 finalize 方法是非常非常罕见的。

注意：不应该使用这以方法。如果你认为你必须使用，请先仔细阅读并理解 Effective Java 第七条 “Avoid Finalizers”。然后不要使用它。

H.7 Javadoc

H.7.1 格式规范

H.7.1.1 通用格式

最基本的 javadoc 的通用格式如下例：

```

1 /**
2  * Muptiple lines of javadoc text are written here,
3  * wrapped normally...
4 */
5 public int method(String p1) {...}

```

或者为单行格式：

```
1 /** An especially short bit of Javadoc. */
```

通用格式在任何时候使用都是可以的。当 javadoc 块只有一行时，可以使用单行格式来替代通用格式。

H.7.1.2 段落

空白行：是指 javadoc 中，上下两个段落之间只有上下对齐的 * 字符的行。每个段落的第一行在第一个字符之前，有一个 <p> 标签，并且之后不要有任何空格。

H.7.1.3 @ 从句

所有标准的 @ 从句，应该按照如下的顺序添加：@param、@return、@throws、@deprecated。并且这四种 @ 从句，不应该出现在一个没有描述的 Javadoc 块中。

当 @ 从句无法在一行写完时，应该断行。延续行在第一行的 @ 字符的位置，缩进至少 4 个字符单位。

H.7.2 摘要片段

每个类或者成员的 javadoc，都是由一个摘要片段开始的。这个片段非常重要。因为它是类或者方法在使用时唯一能看到的文本说明。

主要摘要只是一个片段，应该是一个名词短语或者动词短语，而不应该是一个完整的句子。但是它应该像一个完整的句子一样使用标点符号。

注意：一种常见的错误是以这种形式使用 javadoc：/** @return the customer ID */. 这是不对的。应该改为：/** Returns the customer ID. */.

H.7.3 何处应该使用 Javadoc

至少，Javadoc 应该应用于所有的 public 类、public 和 protected 的成员变量和方法。和少量例外的情况。例外情况如下。

H.7.3.1 例外：方法本身已经足够说明的情况

当方法本身很显而易见时，可以不需要 javadoc。例如：getFoo。没有必要加上 javadoc 说明 “Returns the foo”。

单元测试中的方法基本都能通过方法名，显而易见地知道方法的作用。因此不需要增加 javadoc。

注意：有时候不应该引用此例外，来省略一些用户需要知道的信息。例如：getCanonicalName。当大部分代码阅读者不知道 canonical name 是什么意思时，不应该省略 Javadoc，认为只能写/** Returns the canonical name. */。

H.7.3.2 例外：重载方法

重载方法有时不需要再写 Javadoc。

H.7.3.3 例外：可选的 javadoc

一些在包外不可见的 class 和成员变量或方法，根据需要，也可以使用 javadoc。当一个注释用以说明这个类、变量或者方法的总体目标或行为时，可以使用 Javadoc。

参考文献

- [1] 关于编程语言的数量统计. [Online]. Available: <https://zh.wikipedia.org/wiki/%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80>
- [2] 松本行弘, 代码的未来, 1st ed. 人民邮电出版社, 2013.
- [3] wikimedia. Java 发展历史. wikipedia. [Online]. Available: <https://zh.wikipedia.org/wiki/Java>
- [4] “Java brief history.” [Online]. Available: <http://wiki.dzsc.com/info/7239.html>
- [5] E. G. H. J. Vlissides, 设计模式-可复用面向对象软件的基础. 机械工业出版社, 2007.
- [6] J. Bloch, *Effective Java* 中文版. 机械工业出版社, 2009.