西北工業大學

程序设计基础 (Java)

Fundamentals of Computer Programming

软件学院 **钱锋** 编

2024 年 4 月 16 日

程序设计基础 (Java)

钱锋[1,2]

2024 年 4 月 16 日

[1]Email: strik0r.qf@gmail.com
[2]西北工业大学软件学院, School of Software, Northwestern Polytechnical University, 西安 710072

目录

第 1 章　Java 编程基础

本章介绍 Java 程序设计的基础知识, 我们假定读者具有一定的程序设计知识, 并具有一定的 C 语言程序设计能力基础, 但这并不是必须的, 我们仍然会讲到程序设计中的大部分基本概念, 因此如果你并没有学过编程, 或者没有学过 C 语言, 都没有关系, 我们会尽量照顾到零基础的同学.

1.1　一维数组

数组 (array) 是具有固定大小的、具有相同类型变量的有序集合. 是程序设计中的一种重要的数据结构, 是对批量数据的一种高效、良好的组织方法, 在对数据的批量化操作中非常有用, 本节我们介绍 Java 中的一维数组.

1.1.1　数组的声明、初始化与访问

1. 数组变量的声明

声明一个数组需要声明一个引用[1]数组的变量 `arrayRefVar` 并指定数组的元素类型 `elementType`, 在 Java 中, 有两种声明数组的方法, 它们都是有效的:

```
elementType[] arrayRefVar; // 方法一
elementType arrayRefVar[]; // 方法二, 但并不推荐
```

其中, `elementType` 可以是任意的数据类型, 但数组中的所有元素必须具有相同的数据类型. `arrayRefVar` 是引用该数组的一个变量, 在后续的代码中我们可以利用 `arrayRefVar` 来访问该数组.

注意到声明数组的方法二其实是与 C/C++ 中声明数组的方法是相同的, Java 支持这样的声明方式, 但是我们更推荐采用方法一的方式来完成数组的声明.

2. 数组的创建

数组是引用数据类型, 引用数据类型与基本数据类型变量不同, *声明引用数据类型变量时并不会给该变量分配任何内存空间*, 它只是创建了一个对该变量的引用的存储位置. 这就是说, 数组变量和数组实际上是不同的, 更合适的表述是, *一个数组变量是对数组的一个引用*. 因此仅仅是声明一个数组, 那么此时 `arrayRefVar` 的值应该为 `null`, 因此在声明数组变量之后, 应该使用 `new` 操作符创建数组并将其引用赋给变量:

```
arrayRefVar = new elementType[arraySize];
```

其中 `arraySize` 为数组的大小, 即数组中可以存放的元素的个数. 这条语句新建了一个大小为 `arraySize` 的数组, 并将其引用赋给了变量 `arrayRefVar`.

[1]在 Java 中, 数组是一个引用类型.

对于数组变量的声明和创建, 其实可以将声明数组、创建数组、将数组的引用赋值给引用变量这三个步骤, 即上述的两条语句合并到一起:

```
elementType[] arrayRefVar = new elementType[arraySize];
```

我们要说明的是, 尽管数组变量与数组是不同的概念, 即数组变量 `arrayRefVar` 只是对数组的一个引用, 但我们在下文中通常不强调这两个概念之间的区别, 所以也可以说 `arrayRefVar` 是一个数组.

3. 数组的大小和默认值

在给数组分配内存空间时需要知道数组的大小, 因此在创建数组时必须指定数组的大小 `arraySize`, *数组的大小决定了数组中能够存放的元素个数, 在数组创建完成后, 数组的大小是不能再被修改的*. 数组的大小被存储在 `length` 属性中, 我们可以使用 `arrayRefVar.length` 来得到数组的大小.

数组被创建完毕后, 其元素被赋予默认值, 这与 C 是不同的. 在 C 语言中, 数组在创建完毕后, 这段内存空间中可能还存在一些垃圾数据, 因此直接使用未初始化的数组可能会引发程序的一些问题, 但在 Java 中你可以不用担心这个问题. Java 中各种类型数组的元素默认值如表 1.1 所示.

表 1.1: Java 中数组的元素默认值

数组类型	**默认值**
数值型	对应元素类型的 `0`
字符型	`'\u0000'`
`boolean`	`false`

4. 访问数组元素

可以通过数组元素的索引 (index) *来访问数组的元素*, 数组元素的索引从 `0` 开始, 因此, 数组元素索引的有效范围是 `0` 到 `arrayRefVar.length - 1`, 同时, 在有效范围内的表达式也可以作为索引使用.

利用索引访问到数组元素后, 就可以像使用通常的变量一样使用数组的元素了, 你可以用它来参与运算并给其他变量赋值, 也可以用一个有效的表达式来给它赋值.

例 1.1.1. 创建一个大小为 `6` 的, 元素类型为 `int` 的整型数组 `myList`, 并为其进行初始化赋值.

```
// 声明并创建数组
int[] myList = new int[6];

// 初始化赋值
myList[0] = 1;
myList[1] = 1;
myList[2] = 4;
myList[3] = 5;
myList[2+2] = 1;
myList[5] = 4;
```

此时, `myList` 在内存空间中的存在形式如图 1.1 所示. 注意到在上述代码中, 我们实际上还是用了表达式 `2 + 2` 来代替 `4` 来给数组的第五个 (要注意数组元素的索引是从 `0` 开始, 到 `myList.length - 1` 结束的) 元素赋值.

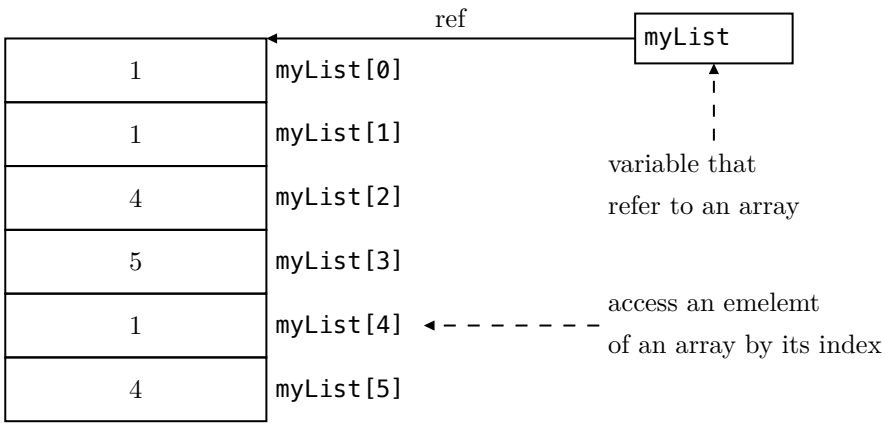

图 1.1: 数组的内存形式

5. 使用初值列表来初始化数组

Part I

面向对象分析与设计

第 2 章　类与对象

面向对象程序设计 (Object Oriented Programming, OOP) 就是使用对象进行程序设计. 本章与后续几章将以一个 GTD 个人事务管理系统为例讨论 Java 的面向对象特性, 如果读者想要了解有关面向对象的相关内容, 可以参阅笔者编写的《面向对象编程与设计》.

2.1　面向对象的基本概念

2.1.1　类与对象

对象 (object) 是类的一个实例, 有状态和行为. 是属性 (数据、状态) 和处理属性的方法 (行为、操作、服务) 的封装, 其中属性和方法作为一个不可分离的整体封装在一起, 这也意味着方法是依赖于对象存在的, 没有游离于对象之外的方法. 一个对象的状态是由**数据域** (field) 及其当前的值来表示的, 对象的行为是由**方法** (method) 来定义的.

类 (class) 是一个创建对象的模板, 它规定了该类中所有对象的属性和方法, 对象是类的实例, 可以从一个类中创建多个实例. 如果我们想要通过类为一类事物建模, 最关键的步骤是从这一类事物的许多具体的实例中抽象出主要特征.

我们可以通过类为我们平时生活中需要处理的工作/事务/任务 (task) 进行建模. 一个任务可以由以下的信息所确定:

- 对任务的简单描述 (description), 它指示了这一个任务大概将要做什么, 例如, “喂猫”、“完成本周的离散数学课程作业”、“观看 Strik0r 的新视频”或者“阅读《深入理解计算机系统》”.
- 截止时间 (deadline), 它指示了一个任务在哪一个具体的时间点后将不再有效. 具体而言, 如果你的离散数学课在本周日的 23:30 截止提交, 那么你必须在这个时间点前完成并提交, 否则将会影响你的平时成绩.
- 安排时间 (arrangement), 它指示了你决定大概在何时去做这一项任务, 要注意的是, 为任务所安排的时间和任务的截止时间是不同的, 这就是说, 虽然你的离散数学作业在本周日的 23:30 才截止, 但是你依然可以安排在今天、明天或者有效期内的任意一天来完成这项工作.
- 对任务的具体描述 (info), 它详细地记录了与任务有关的一些信息.

除此之外, 一个任务可能还涉及到“委托人”(即你把这项工作分配给谁来完成)、“优先级”(即这项任务应当被优先完成, 还是可以适当延迟, 或者干脆没有优先级)、“标签”(这是你给任务设置的标记, 可以用于在信息管理系统中进行筛选)、“前置任务”(即为了完成这项任务, 你必须先完成什么任务) 等. 我们现在只考虑简单描述 `description`、截止时间 `deadline`、安排时间 `arrangement` 和具体描述 `description`, 考虑到数据管理的需求, 我们再为它增加一个唯一的标识数据域 `id`. 那么一个任务实际上可以被一个包含有

简单描述、截止时间和安排时间的元组所唯一的确定, 我们抓住这一主要特征, 把这些任务抽象为一个类 `Task`, 我们将会在下一节中讲到如何在 Java 中将其创建出来.

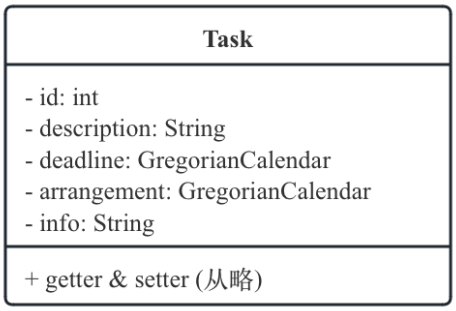

图 2.1: `Task` 类的 UML 类图

2.1.2 类的定义

Java 是纯面向对象的语言, 因此 Java 程序中的所有函数 (我们在 Java 中习惯将其称为**方法**, method) 都必须放在**类** (class, 本书中常称为 Java class) 中来定义, 更彻底地说, Java 程序中的所有内容都必须在类中来组织. 类是 Java 程序的基本组成部分, 不管多大的 Java 工程, 都是由若干个 Java class 组织起来的.

在 Java 中使用 `class` 关键字定义一个类, 具体的语法为

```
(accessModifier) class className {

}
```

关于 Java class 的定义, 我们做一些说明:

首先, 在 `class` 关键字前可以添加 `public` **访问修饰符** (access modifier), 我们会在后面详细介绍各种访问修饰符的作用及其使用方法, 现在你只需要知道, 访问修饰符用于控制程序的其他部分对这段代码的访问权限级别, Java 对访问权限的控制是封装性在 Java 中的一个具体实现.

因此, `public class Hello {}` 表示我们定义了一个访问权限为 `public` 的类 `Hello`, 而类中的所有元素的定义全部放在类名后面的大括号里.

其次, 在一个 Java 源文件中你可以定义多个类, 但是访问修饰符声明为 `public` 的类 (这样的类称为**主类**) 最多只能有一个 (可以没有), 并且如果有的话, 源文件的文件名必须与主类的类名相同, 这就是我在前文中叫你把源文件命名为 `Hello.java` 的原因. 如果你没有注意到我的提示, 把类 `Hello` 声明在了其他源文件中, 那么你将会收到编译器的警告: *错误: 类 `Hello` 是公共的, 应在名为 `Hello.java` 的文件中声明*.

一般来说, 我们约定一个源文件中仅编写一个类 (除非有测试需求).

第三, Java 的类命名是有一定的规则的, 不合适的类命名会让人感到困惑, 进而大大降低代码的可读性和可维护性, Java class 的命名规则为:

- 类名必须以字母开头, 后面可以是字母和数字的任意组合, 一般用名词来给类命名;
- 采用 `Pascal` 命名规则, 即每一个英文字母的首字母大写;

- 类名不能使用 Java 保留字;

程序清单 2.1.1. Task.java

```java
package cn.edu.nwpu.software.strik0r.TaskManagement.Task;

import java.util.GregorianCalendar;

/**
 * @author Strik0r
 * {@code Task} 类是个人事务管理系统中的 “任务” 的所在类
 * */
public class Task {
    private int id; // 任务的序列号
    private String description; // 简单描述
    private GregorianCalendar deadline; // 截止时间
    private GregorianCalendar arrangement; // 安排时间（计划去完成它的时间）
    private String info;
    // TODO: 完成有关优先级、完成状态、标签、评论、子任务、任务动态、位置提醒等属性的概要设计

    // getter & setter
    public int getId() {
        return id;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public GregorianCalendar getDeadline() {
        return deadline;
    }
    public void setDeadline(GregorianCalendar deadline) {
        this.deadline = deadline;
    }
    public GregorianCalendar getArrangement() {
        return arrangement;
    }
    public void setArrangement(GregorianCalendar arrangement) {
        this.arrangement = arrangement;
    }
    public String getInfo() {
        return info;
    }
```

```
42      public void setInfo(String info) {
43          this.info = info;
44      }
45  }
```

第 3 章　Java 语言概述

关于 Java 语言的历史、计算机基础等内容我们在本章不做赘述, 我们直接从 Java 语言的一些基本使用开始. 本章介绍一个简单的 Java 程序 `Hello.java`, 并介绍 Java 的一些基础语法和 Java 语言的特点.

3.1　Hello Java World!

例 3.1.1. (Hello, Java!). 这是一个经典的 hello 程序, 我们基于它来介绍 Java 程序的基本结构. 请编写一个 `Hello.java` 的 Java 类文件, 在 IDE 中输入以下代码, 然后尝试运行它: 这段程序的执行结果是向控制台输出信息 `Hello Java World!`. 接下来, 我们将从类的定义、方法的定义和 `main` 方法、Java 语句和 `.` 运算符三个角度来解释这段代码.

3.1.1　方法的定义和 `main` 方法

1. *方法的定义*

在 Java 中定义一个方法的语法为

```
(accessModifier) (static) returnType methodName(arg1Type arg1, ...) {

}
```

其中 `accessModifier` 和 `static` 是可选的, `returnType` 指的是该方法的返回类型, `methodName` 是方法名, `argType` 是该方法接受的参数的类型, `arg` 是该方法接收的参数名 (我们往往将其称为**形式参数**, 你只要把它理解为函数 $f(x)$ 里的 x 就行了, 其中, x 是一个 `argType` 类型的变量). 方法中需要执行的指令序列称为方法体, 放在方法声明后的大括号中.

2. *`main` 方法的定义*

在 `main` 方法的定义中, 我们使用了关键字 `public` 和 `static`, 并且指定 `main` 方法没有返回值 (`void`). 现在你不需要去管形式参数 `String[] args` 是什么 (老实说, 直到我在写这一章的时候我依然不知道它是什么), 我们会在后面详细介绍. 你需要知道的是:

1. `main` 方法是 Java 应用程序的执行入口, 它的固定声明格式为

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

现阶段, 请你向八股文一样把它死背住, 随着我们学习的深入, 你会逐渐理解这里面每一个关键词的含义.

2. 每一个类都可以有 `main` 方法, 即 `main` 方法也可以写在非 `public` 类中, 然后指定运行非 `public` 类, 那么此时程序的入口就将是该非 `public` 类的 `main` 方法.

3. 需要注意的是, `main` 方法的拼写是 main, 许多人会在不经意间错拼成 mian, 因此当找了很久找不到错误时, 不妨看看你有没有犯这一个低级的拼写错误.

3.1.2 Java 语句和 . 运算符

`System.out.println("Hello Java World!");` 是 `main` 方法中唯一的指令, 它的作用是向控制台输出字符串 `Hello Java World`, 在这里你不需要掌握太多, 需要你明确的是:

1. 一个 Java 语句需要用 `;` 结尾, 如果不加分号的话是会报错的, 语句末尾缺少分号或者输入全角分号 `；` 是初学者常犯的错误.

2. Java 中利用大括号 `{}` 来划分代码的结构, 先前提到的类体、方法体都是写在类声明、方法声明后的大括号中的, Java 中的大括号总是成对出现的, 请你养成成对输入大括号的习惯. 关于大括号的次行风格和行尾风格我们不再单独介绍, 我本人采用的是行尾风格 (听不懂吗? 没有关系, 如果你不知道我在说什么的话, 你就按照我的示例代码来输入大括号就可以了).

3. Java 是面向对象的语言, 因此我们通常用 `object.method()` 来调用对象的方法, 用 `object.attribute` 来访问对象的属性.

3.2 Java 的基础语法

3.2.1 Java 转义字符

表 3.1: Java 中的转义字符

转义字符	用途	备注
`\t`	制表位	用于实现对齐
`\n`	换行符	将光标移动到下一行的开头
`\"`	双引号	
`\'`	单引号	
`\r`	回车	将光标移动到当前行的开头, 新输出的内容会顶掉原先输出的内容

3.2.2 Java 注释

用于注解、说明、解释程序的文字就是注释 (comment), 被注释的文字不会被 **Java 虚拟机** (Java Virtual Machine, JVM) 解释执行, 因此你可以放心地往代码中添加注释, 注释能有效地提高代码的可读性并降低团队成员间的沟通成本, 养成良好的注释习惯是很重要的. 不写注释的代码就像从头到尾只有逻辑符号和运算过程, 而没有任何文字说明的数学文献一样粗暴.

良好的编程习惯是**自顶向下的实现**, 你首先需要做一个顶层的设计和规划, 然后再编码阶段先通过注释写出你的实现策略和实现思路, 然后再逐步用代码去实现你的想法. 你的实现思路可能是由若干个子模块组成的, 对于这些子模块, 采用同样的自顶向下实现方法, 直到它们被分解地足够基本以至于每一个模块只实现一个很简单的功能为止.

在 Java 中实现注释的方式有三种: 单行注释、多行注释和 javadoc 文档注释, 它们的使用方法如下:

```
// Single line comment

/*
  Multi line comments
  System.out.println("This line won't be executed.");
  */

/**
 * javadoc comments
 * @javadocTag this comment will be compliered by javadoc tool.
 */
```

关于注释的使用, 我们说明以下几点:

1. 每一个 `*/` 都会被认为是多行注释的结束, 因此多行注释内不可嵌套多行注释, 所以不要简单地把一段代码用 `/* */` 括起来就认为它们已经被全部注释掉了.

2. 文档注释的注释内容可以被 JDK 提供的 javadoc 工具所解析, 生成一套以网页文件的形式体现的程序说明文档, 其中 `@javadocTag` 是 javadoc 标签, 常见的 javadoc 标签见下表所示:

表 3.2: 常用 Javadoc 标签

标签	描述	示例
`@param`	用于描述方法的参数, 提供参数的名称和描述.	`/** * @param a 第一个加数 * @param b 第二个加数 */`
`@return`	用于描述方法的返回值类型和意义.	`/** * @return 学生的姓名 */`
`@throws`	用于描述方法可能抛出的异常类型和原因.	`/** * @throws Exception */`
`@deprecated`	标记方法或类已经过时, 提醒开发者不再使用.	`/** * @deprecated 使用新方法代替 */`
`@see`	引用其他类、方法、字段或文档.	`/** * 查看其他类的文档 * @see OtherClass */`

3. 类、方法的注释, 要用 javadoc 文档注释来写, 非 javadoc 文档注释内容, 往往是为了便于代码的维护, 即告诉读者这段代码的思路以及相关的注意事项.

3.3 Java 语言的特点

Java 的设计者编写了一个白皮书[1]来解释设计 Java 的初衷以及完成这些目标的情况. 接下来我们介绍其中的四个关键术语, 分别是面向对象、健壮性、可移植性和解释性, 它们是现阶段你需要掌握的 Java 的四个核心特点.

3.3.1 面向对象

Java 是一种纯粹的面向对象编程语言, 这意味着一切皆为对象. 对象是程序的基本构建块, 具有状态和行为. 相比于传统的面向过程编程, 面向对象编程使得代码更模块化, 易于维护和扩展. 在 Java 中, 类和对象的概念是核心, 而且通过接口的灵活运用, 实现了对多继承的替代方案.

3.3.2 健壮性

Java 的健壮性体现在多个方面. 首先, Java 强调编译期和运行时的错误检测, 通过严格的语法检查和类型检查, 能够在程序执行前发现潜在问题. 其次, Java 的垃圾回收机制有助于防止内存泄漏, 大大提高了程序的稳定性. 此外, 异常处理机制也使得开发者能够更好地处理错误情况, 确保程序在面对异常时依然能够稳健运行.

3.3.3 可移植性

Java 的可移植性是通过字节码和 Java 虚拟机 (JVM) 实现的. Java 程序在编译时生成字节码, 而不是直接生成机器码. 这种字节码可以在任何装有相应 Java 虚拟机的平台上运行. 这使得 Java 程序无需重新编写即可在不同的操作系统和硬件上执行, 为跨平台开发提供了强大的支持.

3.3.4 解释性

Java 是一种解释性语言, 它的程序在运行时通过 Java 虚拟机进行解释执行. 这种特性为 Java 带来了很多优势. 首先, Java 程序可以逐行执行, 无需预先编译成本地机器码, 使得开发和调试更为灵活. 其次, 解释性也有助于实现平台无关性, 因为相同的字节码可以在不同平台上被解释执行, 无需修改源代码.

除此之外, Java 在设计上注重安全性. 通过强大的安全管理器, Java 可以在运行时防止恶意代码的执行. Applet 的沙箱机制是其中的一例, 它限制了从网络上下载的代码对本地系统的访问权限, 防止了潜在的安全威胁. Java 还拥有庞大而丰富的生态系统, 有着丰富的标准库和框架. 这使得开发者可以轻松地利用现有的工具和资源来加速开发过程. 从企业级应用到移动应用, Java 生态系统涵盖了各种领域, 为开发者提供了广泛的选择和支持.

[1]可以在 `www.oracle.com/technetwork/java/langenv-140151/html` 上找到.

Part II

Java 程序设计

第 4 章　变量与运算符

变量是赋给值的标签, 指向了内存中数据的存储单元. Java 也用变量来存储值.

4.1　变量

4.2　数值数据类型及其操作

4.3 数值数据类型变量的类型转换

两个不同数值数据类型的操作数进行二元运算的时候, 或者把一个类型的变量赋值给另一个类型的变量的时候, 就需要进行类型转换. 在 Java 中, 数值数据类型类型转换分为自动类型转换和强制类型转换. **类型转换**是将一种数据类型的值转换成另一种数据类型的值的操作:

(1) 从占用字节数较小的类型转换成占用字节数较大的类型的转换称为**扩大类型** (widening a type), 在 Java 中, 扩大类型一般是自动完成的;

(2) 从占用字节数较大的类型转换成占用字节数较小的类型的转换称为**缩小类型** (narrowing a type), 在 Java 中如果想要缩小类型, 就必须显式地进行类型转换.

4.3.1 自动类型转换

在 Java 中, 数值数据类型之间的自动转换遵循**自动提升原则**, 即在一个表达式中有多个数参与运算时, 先把这些参与运算的数转换为容量最大的类型, 再计算表达式的结果. 怎么理解呢? 大烧杯能装的东西肯定比小烧杯能装的东西要多, 因此占用字节数更大的数据类型拥有更多的二进制位, 也就有更大的数据表示范围, 所以我们在计算时先把所有的数据都“倒进最大容量规格的那一个烧杯中”.

4.3.2 强制类型转换

第 5 章　继承和多态

面向对象范式将数据和方法结合在对象中, 使用面向对象范式的软件设计聚焦于对象以及对象上的操作. 面向对象方法结合了面向过程范式的强大之处, 并且进一步将数据和操作集成在对象中.

继承对于软件重用是一个重要且功能强大的特征.

第 6 章　抽象类和接口

6.1　抽象类

抽象方法不能包含在非抽象类中, 如果抽象父类的子类不能实现所有的抽象方法, 那么子类也必须定义为抽象的. 在继承自抽象类的非抽象子类中, 必须实现所有的抽象方法. 此外, 抽象方法是非静态的.

抽象类不能使用 new 操作符来初始化, 但仍然可以定义它的构造方法, 这个构造方法在其子类的构造方法中调用.

包含抽象方法的类必须是抽象的, 但可以定义一个不包含抽象方法的抽象类, 这个抽象类用作定义新子类的基类.

子类可以重写父类的方法并将其定义为抽象的, 这是很少见的一种做法, 一般用于父类的方法实现在子类中变得无效的情况. (尝试举例?)

即使父类是具体类, 子类也可以是抽象的.

不能使用 new 操作符来从一个抽象类创建一个实例, 但是抽象类可以用作一种数据类型.

6.2　接口

接口的目的是指明相关或者不相关的类的对象的共同行为. 定义接口的语法为

```
accessModifier interface InterfaceName {

}
```

在 Java 中, 接口可以看作是一种特殊的类, 你可以使用接口作为引用变量的数据类型或者作为类型转换的结果类型, 但是不能使用 new 操作符创建接口的实例.

一个类可以使用 implements 关键字来实现一个接口.

接口中修饰数据域的 public static final 和修饰方法的 public abstract 可以被忽略, 但是在实现该接口的类中, 必须将接口中的方法声明为 public 的.

如果在声明接口方法时使用了关键字 default, 那么你实际上给出了一个默认的接口方法, 默认的接口方法需要在接口的定义中实现. 实现该接口的类可以简单地使用默认的方法来实现, 或者重写该接口方法. 利用该特征, 在一个具有默认实现的已有接口中添加新的方法时, 只要声明默认方法并给出实现, 就可以不必为已经实现了该接口的已有类重新编写接口方法的实现.

接口中的公共静态方法与类中的公共静态方法相同.

在实现接口的默认方法和公共静态方法时, 可以定义并调用接口的私有方法.

表 6.1: 接口中的方法

方法声明部分形如……	使用方法
`public abstract returnType p()`	等价于 `returnType p()`
`public default returnType p()`	默认接口方法, 实现该接口的子类可以选择直接使用它或者重写它
`public static returnType p()`	公共静态方法, 与类中的公共静态方法的使用方式相同
`private returnType p()`	私有方法, 可以被默认接口方法和公共静态接口方法调用

6.3 标记接口

内部为空的接口称为**标记接口** (marker interface), 标记接口既不包括敞亮也不包括方法, 它用来表示一个类拥有某些希望具有的特征.

6.4 抽象类和接口的区别与联系

表 6.2: 抽象类与接口

	变量	构造方法	方法
抽象类	无限制	子类通过构造方法链调用构造方法, 抽象类不能用 `new` 操作符实例化	无限制
接口	所有的变量必须是公开、静态、最终, 即 `public static final` 的	没有构造方法, 不能用 `new` 操作符实例化	可以包含 `public abstract` 的抽象实例方法、`public default` 的默认接口方法以及 `public static` 的静态方法. 在实现默认接口方法与静态方法时, 作为辅助, 可以定义并调用 `private` 的接口内部私有方法.

Part III

数据结构与算法

第 7 章　线性表

7.1　链表

链表是一种数据结构, 其中的元素 (节点) 通过指针相互连接. 本节我们介绍如何在 Java 中实现一个简单的链表. 它主要有三个类构成——实现节点的 `Node` 类、实现链表本身的 `LinkedList` 类, 以及一个用于测试的编写了 `main` 方法的测试类.

7.1.1　`Node` 类: 链表的节点

在类 `Node` 中, 我们定义了两个属性: `data` 属性表示我们存放在这个节点中的数据的值, 而 `next` 属性指的是当前节点的下一节点. 为了保证程序的封装性不被破坏, 你可以把这些参数设置为私有 (`private`), 但现在我们简单起见, 把这些属性的访问修饰符都设置为缺省, 这意味着我们可以通过简单的赋值运算实现属性的访问和变更, 而不需要使用公共的 `getter` 和 `setter`.

我们为 `Node` 类提供了一个构造器, 它的作用是把节点的 `data` 属性设置为我们给定的形式参数 `data`, 然后默认它是链表的尾节点, 即 `this.next = null`.

7.1.2　`LinkedList` 类: 链表的实现

我们先给出 `LinkedList` 类的完整的实现:

Part IV

代码设计

第 8 章　代码设计概述

学习代码设计的相关知识可以帮助我们写出可扩展、可读和可维护的高质量代码.

表 8.1: 代码质量的评判标准

特征	含义	评判标准
可维护性	在不破坏原有代码设计、不引入新 bug 的情况下, 能够快速修改或添加代码	与多种因素有关: 代码简洁、可读性好、可扩展性好、分层清晰、模块化程度高、高内聚低耦合、遵守基于接口而非实现编程的设计原则
可读性	代码是否易读、易理解	代码符合代码规范、命名达意、注释详尽、函数长度合理、模块划分清晰、高内聚低耦合、Code Review
可扩展性	在不修改或少量修改原有代码的情况下, 能够通过扩展方式添加新功能	
灵活性	含义比较宽泛	
简洁性	代码简单、逻辑清晰	KISS 原则
可复用性	尽量减少重复代码的编写, 复用已有代码	DRY 原则
可测试性	易于编写单元测试	

8.1　关于面向对象编程范式的进一步讨论

面向对象编程范式因其丰富的特性 (封装、抽象、继承和多态) 可以实现很多复杂的设计思路, 因此它是很多设计原则、设计模式编码实现的基础.

8.1.1　面向对象编程范式的特性

1. 封装 (encapsulation)

封装也称为信息隐藏或数据访问保护. 类通过暴露有限的访问接口, 授权外部仅能通过类提供的方式来访问内部信息或数据. 这看似影响了程序代码的灵活性, 但其实过度灵活意味着不可控和不安全, 属性可以通过各种奇怪的方式被随意修改, 而且修改逻辑可能散落在代码的各个角落, 进而影响代码的可读性和可维护性.

类通过提供有限的方法暴露必要的操作, 能提高类的易用性. 将属性封装, 暴露少许的必要的方法给调用者, 调用者就不需要了解太多的业务细节, 用错的概率也会降低很多.

2. 继承 (inheritance)

继承最大的作用是代码复用.

3. 多态 (polymorphism)

多态是指, 在代码运行过程中我们可以用子类替换父类, 并调用子类的方法.

第 9 章　设计模式

9.1　创建型设计模式

9.1.1　单例模式

Part V

编程速查

第 10 章　时间相关的需求……

10.1　Calendar 类及其子类：表示精确到毫秒的特定时刻

先前我们曾经介绍过, 学习各种 Java 类的核心在于明确该类在对哪一种客观物质世界的对象进行建模. 一个具体的 GregorianCalendar 对象代表了一个具体的公历 (gregorian calendar) 时间节点.

表 10.1: Calendar 类的域常量

常量	说明
YEAR	年份
MONTH	月份, 其中 0 表示 1 月
DATE	日前
OUR	小时 (12 小时制)
HOUR_OF_THE_DAY	小时 (24 小时制)
MINUTE	分钟
SECOND	秒
DAY_OF_WEEK	一周中的哪一天, 1 表示星期天
DAY_OF_MONTH	同 DATE
DAY_OF_YEAR	一年中的哪一天, 其中 1 表示该年的第一天
WEEK_OF_MONTH	一月中的哪一周, 其中 1 表示该月的第一周
WEEK_OF_YEAR	一年中的哪一周, 其中 1 表示该年的第一周
AM_PM	0 表示上午, 1 表示下午

10.1.1　复习参考题

习题 10.1. 可以使用 Calendar 类来创建一个 Calendar 对象吗? 这显然是不可以的, 因为 Calendar 类是一个抽象类, 不能够通过 new 操作符构造一个 Calendar 对象.

参考文献

[1] [美] Cay S. Horstmann. Java 核心技术: 原书第 12 版. 卷 I: 开发基础 [M]. 林琪, 苏钰涵译. 北京: 机械工业出版社, 2022

[2] [美] Cay S. Horstmann. Java 核心技术: 原书第 12 版. 卷 II: 高级特性 [M]. 陈昊鹏译. 北京: 机械工业出版社, 2023

[3] 王争. 设计模式之美 [M]. 北京: 人民邮电出版社, 2022.

公诚勇毅　永矢毋忘
中华灿烂　工大无疆

2023 年　西北工业大学　中国西安