

✧ 1 ✧ Java 应用与开发课程教学体系

很高兴同学们能够选修 Java 应用与开发课程。

希望我们一起通过这门课程的学习，建立 Java 语言编程的初步知识体系，掌握 Java 应用系统开发的方式、方法。更重要的，能够对编程这个事情、这项技能有更加深刻的认知，对未来的职业化发展有所促进。

Java 应用与开发课程的教学体系如图1.1所示，包括了 Java SE 和 Java EE 两个部分，每部分都涉及一些验证性实验，另外，会开展两次稍微大一点的集成开发项目。同时，在学习的过程中会穿插一些开发工具、设计模式、应用服务器和数据库的基本应用。

在课程学习的过程中，希望同学们要有足够的求知欲，养成良好的学习态度，具备不断探索的精神，多尝新、多实践、多总结。我想这是计算机专业人士应该具备的基本素养。

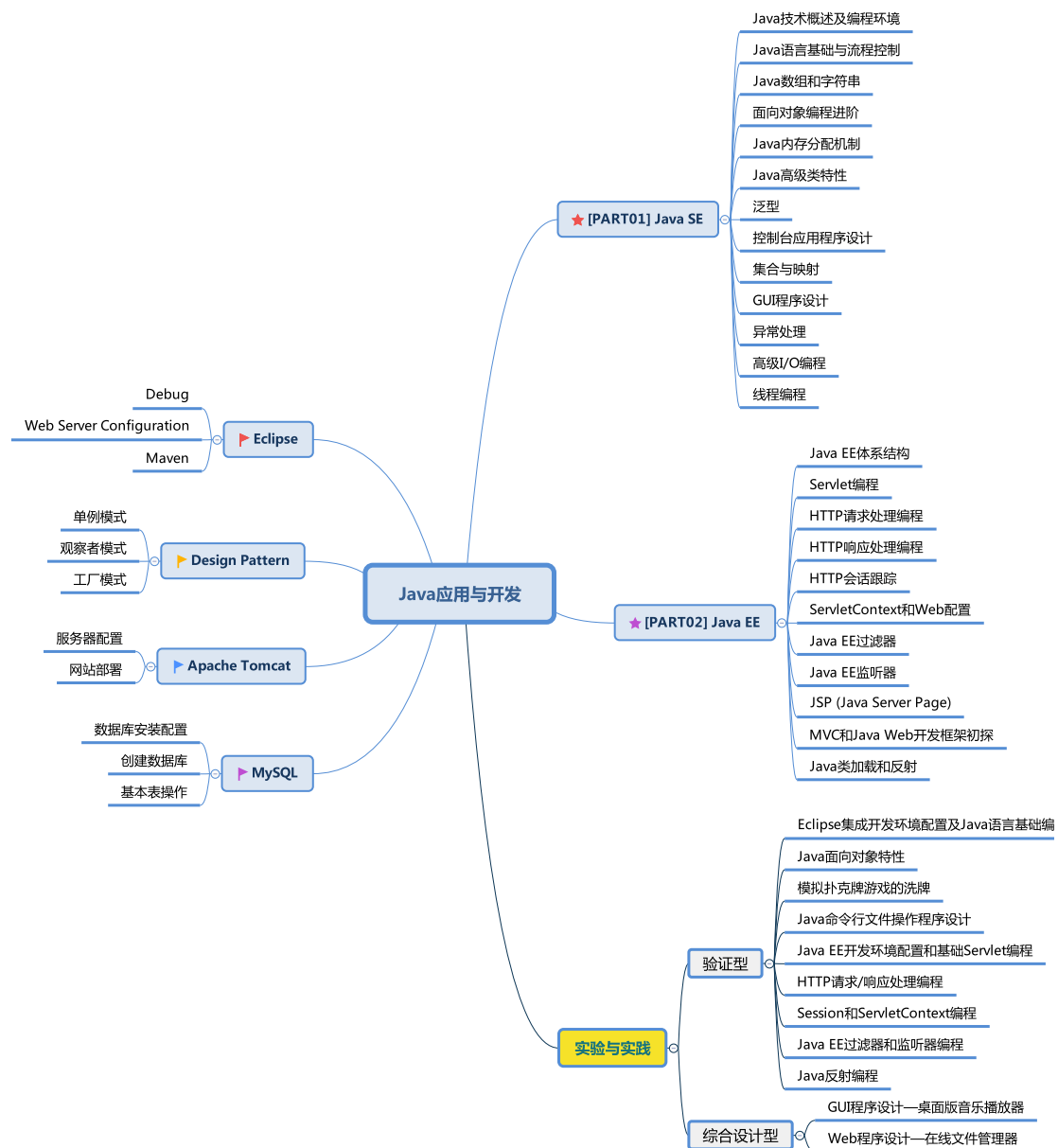


图 1.1 Java 应用与开发课程教学体系

✧ 2 ✧ Java 技术概述及开发环境

基本信息

课程名称： Java 应用与开发

授课教师： 王晓东

授课时间： 第一周

参考教材： 本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 讲解 Java 的发展历程，从 Java 的视角回顾 OOP；
2. 理解 Java 平台的相关概念和机制；
3. 掌握基本的 Java 开发环境配置方法。

授课方式

理论课： 多媒体教学、程序演示

实验课： 上机编程

教学内容

2.1 Java 技术概述

2.1.1 Java 发展简史

Java 的发展过程中伴随着多个伟大公司的起起落落。

1982 Sun 公司成立（安迪·贝托谢姆和麦克尼利）。

1986 Sun 公司上市。

1985 Sun 公司推出著名的 Java 语言。

2001 9.11 事件前，Sun 市值超过 1000 亿美元；此后，由于互联网泡沫的破碎，其市值在一个月内跌幅超过 90%。

2004 Sun 公司和微软在旷日持久的 Java 官司中和解，后者支付前者高达 10 亿美元的补偿费。

2006 共同创始人麦克尼利辞去 CEO 一职，舒瓦茨担任 CEO 后尝试将 Sun 从设备公司向软件服务型公司转型，但不成功。

2010 Sun 公司被甲骨文公司收购。

Java 语言的版本迭代历程如图4.1所示。

2.1.2 Java 技术的特点

Java 具备以下技术特点：

面向对象 Java 是一种以对象为中心，以消息为驱动的面向对象的编程语言。

平台无关性 分为源代码级（需重新编译源代码，如 C/C++）和目标代码级 (Java) 平台无关。

分布式 可支持分布式技术及平台开发。

可靠性 不支持直接操作指针，避免了对内存的非法访问；自动单元回收功能防止内存丢失等动态内存分配导致的问题；解释器运行时实施检查，可发现数组和字符串访问的越界；提供了异常处理机制。

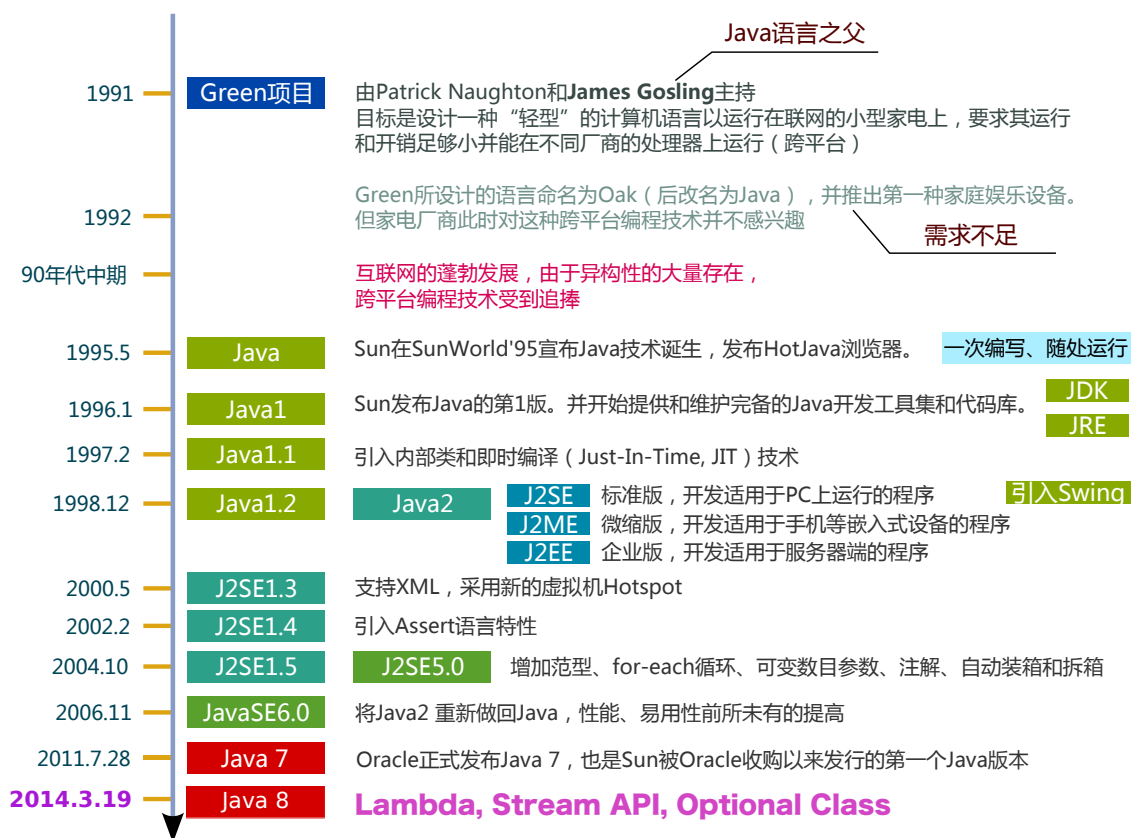


图 2.1 Java 版本迭代

多线程 C++ 没有内置的多线程机制，需调用操作系统的多线程功能来进行多线程程序设计；Java 提供了多线程支持。

网络编程 Java 具有丰富的网络编程库。

编译和解释并存 由编译器将 Java 源程序编译成字节码文件，再由运行系统解释执行字节码文件（解释器将字节码再翻译成二进制码运行）。

2.2 Java 平台核心机制

Java 技术栈如图2.2所示，程序的编译运行过程如图2.3所示。需要了解以下几个核心概念：

- Java 虚拟机
- 垃圾回收机制
- Java 运行时环境（Java Runtime Environment, JRE）

- JIT, Just-In-Time 传统解释器的解释执行是转换一条，运行完后就将其扔掉；JIT 会自动检测指令的运行情况，并将使用频率高（如循环运行）的指令解释后保存下来，下次调用时就无需再解释（相当于局部的编译执行），显著提高了 Java 的运行效率。

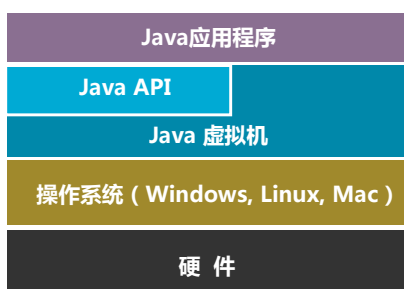


图 2.2 Java 技术栈

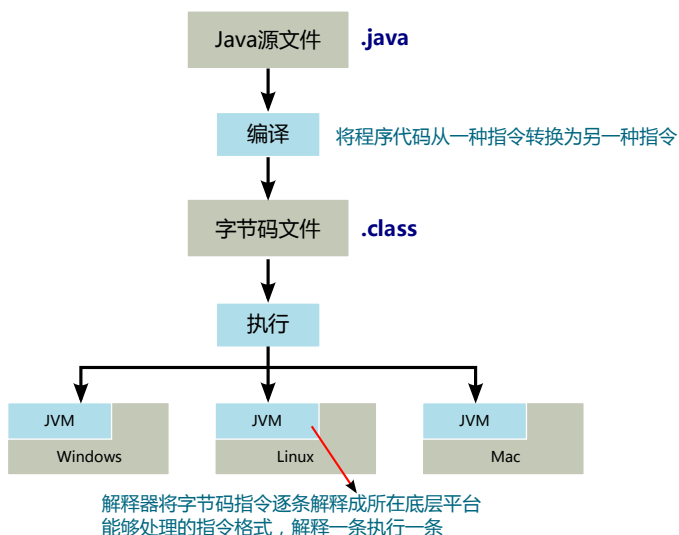


图 2.3 Java 程序编译运行过程

2.3 Java 开发环境

构建 Java 开发环境，需要首先获取和安装 Java 开发工具集，可以从 Oracle 官方网站链接 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 获取。下载完成后解压放入合适的磁盘目录下。

对于 Windows 操作系统，可以采用以下路径：

```
1 D:\Program Files\Java
```

对于 Linux 操作系统，可以采用以下路径：

```
1 /opt/jdk1.8.0_172
```

接下来进行环境变量配置，以 Windows 操作系统为例：

变量名 Path

变量值 D:\Program Files\Java\jdk1.8.0_172\bin

配置完成后，可以看到 JDK 目录中包含以下子目录和文件：

```
1 bin  COPYRIGHT db include javafx-src.zip
2 jre  lib  LICENSE man README.html release src.zip
3 THIRDPARTYLICENSEREADME-JAVAFX.txt THIRDPARTYLICENSEREADME.
   txt
```

对子目录的功能简要描述如下：

bin Java 开发工具，包括编译器、虚拟机、调试器、反编译器等；

jre Java 运行时，包括 Java 虚拟机、类库和其他资源文件；

lib 类库和所需支持性文件；

include 用于调试本地方法（底层平台）的 C++ 头文件；

src.zip 类库的源代码；

db Java DB 数据库，JDK6.0 新增项目，一种纯 Java 的关系型数据库。

2.4 Java 开发工具

业界普遍采用 Eclipse 或 IntelliJ IDEA 等集成开发环境进行 Java 大型工程开发，当然也可以采用文本编程工具 Vim 或 Emacs 等进行 Java 小型程序的开发。

本课程采用 Eclipse 作为首选集成开发环境。

2.5 Java 基本开发流程

本部分使用文本编程工具编写一个简单的 Java Hello World 程序，演示 Java 的基本开发和代码编译运行流程。首先，我们需要使用文本编程工具编写一个 Java 源文件 HelloWorld.java，文件命名必须与类名相同。

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

然后，使用 `javac` 工具将源文件编译为字节码文件，编译完成后，我们可以看到生成了 `HelloWord.class` 这一字节码文件。

```
1 > javac HelloWorld.java && ls
2 HelloWorld.class HelloWorld.java
```

接下来，我们使用 `java` 工具运行该程序，在终端正确打印了“Hi, Java”字符串。

```
1 > java HelloWorld
2 Hi, Java!
```

说明

编写 Java 应用程序需掌握的几条规则如下：

1. Java 语言拼写是大小写敏感的（Case-Sensitive）；
2. 一个源文件中可以定义多个 Java 类，但其中最多只能有一个类被定义为 Public 类；
3. 如果源文件中包含了 `public` 类，则源文件必须和该 `public` 类同名；
4. 一个源文件包含多个 Java 类时，编译后会生成多个字节码文件，即每个类都会生成一个单独的“.class”文件，且文件名与类名相同。

✧ 3 ✧ Java 语言基础与流程控制

基本信息

课程名称：Java 应用与开发

授课教师：王晓东

授课时间：第一周

参考教材：本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. Java 语言基础包括：数据类型、常量和变量、关键字与标识符、运算符与表达式、从键盘输入数据。
2. Java 流程控制包括：语句和复合语句、分支结构（选择结构）、循环结构、跳转语句。

授课方式

理论课：多媒体教学、程序演示

实验课：上机编程

教学内容

3.1 Java 语言基础

3.1.1 数据类型

Java 数据类型分为两大类：基本数据类型和引用数据类型。基本数据类型是由程序设计语言系统所定义、不可再划分的数据类型。所占内存大小固定，与软硬件环境无关，在内存中存放的是数据值本身。Java 的基本数据类型包括：整型（byte、short、int、long）、浮点型（float、double）、逻辑型（boolean）和字符型（char）。引用数据类型（复合数据类型）在内存中存放的是指向该数据的地址，不是数据值本身。引用数据类型包括类、数组、接口等。

数据类型的基本要素包括：

- 数据的性质（数据结构）
- 数据的取值范围（字节大小）
- 数据的存储方式
- 参与的运算

整型

Java 整型类型的数据位数及取值范围如表3.1所示。

表 3.1 整型数据类型

类型	数据位数	取值范围
byte（字节型）	8	$-128 \sim 127$ ，即 $-2^7 \sim 2^7 - 1$
short（短整型）	16	$-32768 \sim 32767$ ，即 $-2^{15} \sim 2^{15} - 1$
int（整型）（默认）	32	$-2147483648 \sim 2147483647$ ，即 $-2^{31} \sim 2^{31} - 1$
long（长整型）（l 或 L）	64	$-2^{63} \sim 2^{63} - 1$

浮点型

Java 浮点型类型的数据位数及取值范围如表3.2所示。

表 3.2 整型数据类型

类型	数据位数	取值范围
float（单精度）（f 或 F）	32	$1.4E - 45 \sim 3.4E + 38$
double（双精度）（默认）	64	$4.9E - 324 \sim 1.8E + 308$

逻辑型

逻辑型又称为布尔型（boolean），布尔型数据类型的特性如下：

- 布尔型数据只有 true（真）和 false（假）两个取值。
- 布尔型数据存储占 1 个字节，默认取值为 false。
- 布尔型数据 true 和 false 不能转换成数字表示形式。

字符型

- 字符型数据类型用来存储单个字符，采用的是 Unicode 字符集编码方案¹。
- 字符声明用单引号表示单个字符。
- 字符型数据可以转化为整型。

Code: 字符数据类型示例

```

1 public class CharDemo {
2     public static void main(String[] args) {
3         char a = 'J';
4         char b='Java'; //会报错
5     }
6 }
```

3.1.2 数据类型转换

数值型不同类型数据的转换

数值型不同类型数据之间的转换，自动类型转换需要符合以下条件：

1. 转换前的数据类型与转换后的类型兼容。

¹建议搜索理解什么是字符集和字符编码规则。

2. 转换后的数据类型的表示范围比转换前的类型大。
3. 条件 2 说明不同类型的数据进行运算时，需先转换为同一类型，然后进行运算。转换从“短”到“长”的优先关系为：
byte → short → char → int → long → float → double

如果要将较长的数据转换成较短的数据时（不安全）就要进行**强制类型转换**，格式如下：

```
1 (预转换的数据类型) 变量名;
```

字符串型数据与数值型数据相互转换

Code: 字符串数据转换为数值型数据示例

```
1 String myNumber = "1234.56";  
2 float myFloat = Float.parseFloat(MyNumber);
```

字符串可用加号“+”来实现连接操作。若其中某个操作数不是字符串，该操作在连接之前会自动将其转换成字符串。所以可用加号来实现自动的转换。

Code: 数值型数据转换成字符串数据示例

```
1 int myInt = 1234;           //定义整形变量MyInt  
2 String myString = "" + MyInt; //将整型数据转换成了字符串
```

3.1.3 常量和变量

常量

整型常量 八进制、十六进制、十进制长整型后需要加 l 或 L。

浮点型常量 单精度后加 f 或 F，双精度后加 d 或 D 可省略。

逻辑型常量 true 或者 false。

字符型常量 单引号。

字符串常量 双引号。

Code: 常量的声明

```

1 final int MAX = 10;
2 final float PI = 3.14f;

```

变量

变量的属性包括变量名、类型、值和地址。**Java** 语言程序中可以随时定义变量，不必集中在执行语句之前。

Code: 变量声明、初始化和赋值

```

1 int i, j = 0;
2 i = 8;
3 float k;
4 k = 3.6f;

```

3.1.4 关键字与标识符

Java 的关键字（Java 保留字）如表3.3所示。

表 3.3 Java 语言的关键字（保留字）

abstract	assert	boolean	break	byte	case
catch	char	class	continue	default	do
double	else	enum	extends	false	final
finally	float	for	if	implements	import
instanceof	int	interface	long	native	new
null	package	private	protected	public	return
short	static	super	switch	synchronized	this
volatile	throws	transient	true	try	void

标识符是用来表示变量名、类名、方法名、数组名和文件名的有效字符序列。**Java** 语言对标识符的规定如下：

- 可以由字母、数字、下划线 (_)、美元符号 (\$) 组合而成。
- 必须以字母、下划线或美元符号开头，不能以数字开头。
- 关键字不能当标识符使用。
- 区分大小写。

建议遵循驼峰命名，类名首字母大写，变量、方法及对象首字母小写的编码习惯。

3.1.5 运算符与表达式

按照运算符功能来分，Java 基本的运算符包括以下几类：

算术运算符	+, -, *, /, %, ++, --
关系运算符	>, <, >=, <=, ==, !=
逻辑运算符	!, &&, , &, , ^
位运算符	>>, <<, >>>, &, , ^, ~
赋值运算符	=, 扩展赋值运算符, 如 +=, /= 等
条件运算符	? :
其他运算符	包括分量运算符 .、下标运算符 [], 实例运算符 instanceof、内存分配运算符 new、强制类型转换运算符(类型)、方法调用运算符()等

图 3.1 Java 运算符

3.1.6 从键盘获得输入

由键盘输入的数据，不管是文字还是数字，Java 皆视为字符串，若是要由键盘输入获得数字则必须再经过类型转换。

Code: 获得键盘输入字符串并转换为数字

```
1 import java.io.*;
2 public class MyClass {
3     public static void main(String[] args) throws IOException {
4         int num1, num2;
5         String str1, str2;
6         InputStreamReader in;
7         in = new InputStreamReader(System.in);
8         BufferedReader buf;
9         buf = new BufferedReader(in);
10        System.out.print("请输入第一个数: ");
11        str1 = buf.readLine(); // 将输入的内容赋值给字符串变量 str1
12        num1 = Integer.parseInt(str1); // 将 str1 转成 int 类型后赋给 num1
13        System.out.print("请输入第二个数: ");
14        str2 = buf.readLine(); // 将输入的内容赋值给字符串变量 str2
```

```

15     num2 = Integer.parseInt(str2); //将 str2 转成 int 类型后赋给 num2
16     System.out.println(num1 + " * " + num2 + " = " + (num1 * num2));
17 }
18 }

```

为了简化输入操作，从 JavaSE 5 版本开始在 `java.util` 类库中新增了一个类专门用于输入操作的类 **Scanner**，可以使用该类输入一个对象。

Code: 使用 **Scanner** 获得键盘输入并转换为特定数据类型

```

1  import java.util.*;
2  public class MyClass {
3      public static void main(String[] args)
4      {
5          Scanner reader = new Scanner(System.in);
6          double num;
7          num = reader.nextDouble(); //按照 double 类型读取键盘输入
8          ...
9      }
10 }

```

Scanner 对象其他可用的数据读取方法包括：`nextByte()`、`nextDouble()`、`nextFloat()`、`nextInt()`、`nextLong()`、`nextShort()`、`next()`、`nextLine()`。

3.2 Java 流程控制

3.2.1 语句与复合语句

- Java 语言中语句可以是以分号 “;” 结尾的简单语句，也可以是用一对花括号 “{}” 括起来的复合语句。
- Java 中的注释形式：
 - 单行注释：//
 - 多行注释：/* */
 - 文件注释：/** */

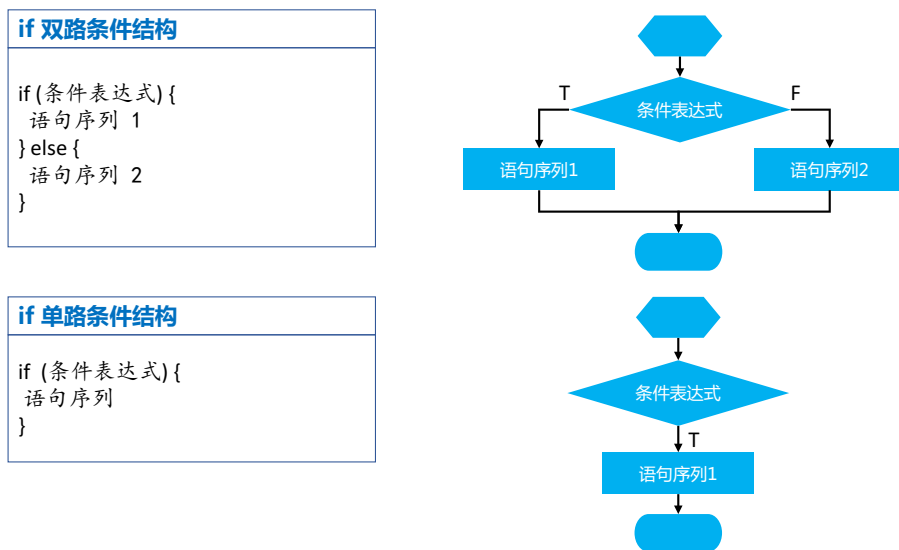


图 3.2 if 分支结构 1

3.2.2 分支结构

if 分支结构 1

if 分支结构 2

switch 分支结构

说明

在 Java 1.7 版本之后，switch 里表达式的类型可以为 String。

3.2.3 循环结构

while 循环

```

1 while(conditional expression) {
2     statements goes here ...
3 }
  
```

do-while 循环

```

1 do {
2     statements goes here ...
  
```

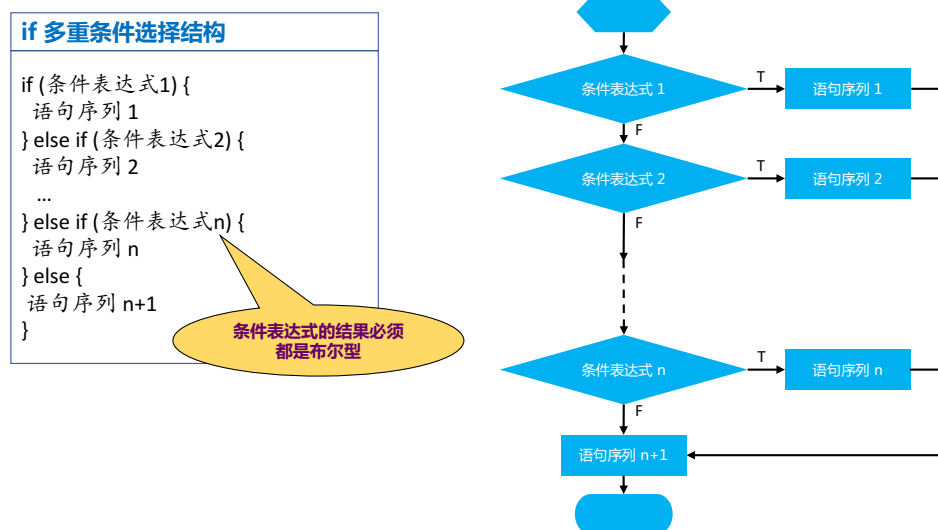



图 3.3 if 分支结构 2

```

3     }
4     while(conditional expression);

```

for 循环 1

```

1     int [] integers = {1, 2, 3, 4};

3     for (int j = 0; j < integers.length; j++) {
4         int i = integers[j];
5         System.out.println(i);
6     }

```

for 循环 2

```

1     int [] integers = {1, 2, 3, 4};

3     for (int i : integers) {
4         System.out.println(i);
5     }

```

循环中的跳转

break 语句 使程序的流程从一个语句块（switch 或循环结构）内跳出。

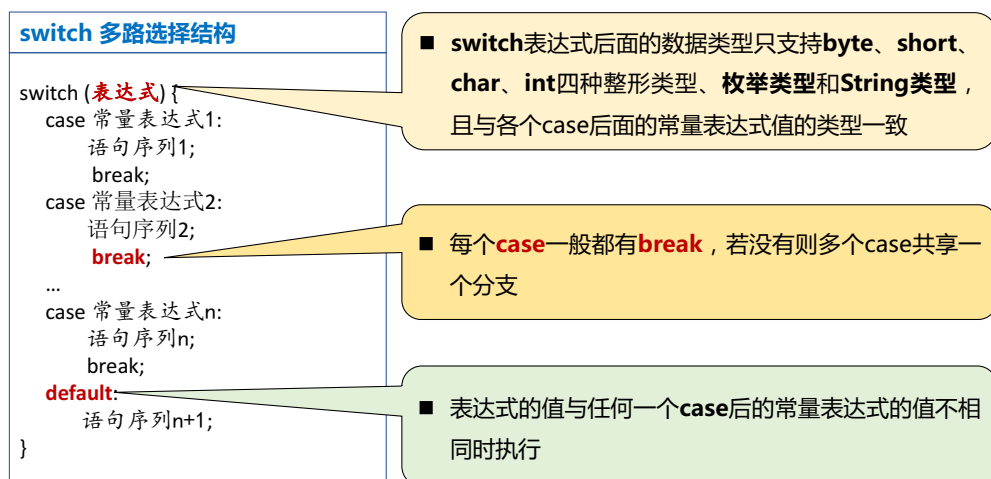


图 3.4 switch 分支结构

continue 语句 终止当前这一轮（次）的循环，进入下一轮（次）循环。

return 语句 用来使程序从方法（函数）中返回，可返回一个值。

3.3 课后习题

3.3.1 简答题

1. Java 语言定义类哪些基本数据类型？其存储结构分别是什么样的？
2. 自动类型转换的前提是什么？转换时的优先级顺序如何？
3. 数字字符串转换为数值类型数据时，可以使用的方法有哪些？

3.3.2 小编程

1. 编写程序，从键盘输入一个浮点数，然后将该浮点数的整数部分输出。
2. 编写程序，从键盘输入 2 个整数，然后计算它们相除后得到的结果并输出，注意排除 0 除问题。

实验设计

实验名称： Eclipse 集成开发环境配置及 Java 语言基础编程练习

上机时间： 第一周

实验手册： 无（参照实验内容完成）

实验内容： 本次实验需要完成以下内容：

1. 使用文本编辑器完成 Java Hello World 程序编写，使用 javac 和 java 编译运行该程序；
2. 熟悉 Eclipse 集成开发环境，学习创建 Java 工程，使用 Maven 创建 Java 工程；
3. 根据授课幻灯片和讲义，尝试实现其中所有的示例代码。

实验要求： 本次实验不需要提交实验报告。

✧ 4 ✧ Java 数组和字符串

基本信息

课程名称：Java 应用与开发

授课教师：王晓东

授课时间：第二周

参考教材：本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握 Java 数组的概念
2. 学会一维数组和二维数组的使用；认识 Arrays 类，掌握操作数组相关方法
3. 掌握 Java 字符串的概念，字符串与数组的关系；学会 String 类常用字符串操作方法

授课方式

理论课：多媒体教学、程序演示

实验课：上机编程

教学内容

4.1 数组的概念

数组是相同数据类型的元素按一定顺序排列的集合。在 Java 语言中，数组元素既可以为基本数据类型，也可以为对象。

Java 的内存分配 (基础)

栈内存 存放定义的基本类型的变量和对象的引用变量，超出作用域将自动释放。

堆内存 存放由 new 运算符创建的对象和数组，由 Java 虚拟机的自动垃圾回收器来管理。

Java 数组的主要特点包括以下方面：

- 数组是相同数据类型的元素的集合；
- 数组中的各元素有先后顺序，它们在内存中按照这个先后顺序连续存放；
- 数组的元素用整个数组的名字和它自己在数组中的顺序位置来表示。

例如，a[0] 表示名字为 a 的数组中的第一个元素，a[1] 表示数组 a 的第二个元素，依次类推。

4.2 一维数组

4.2.1 创建数组

创建 Java 数组一般需经过三个步骤：

1. 声明数组；
2. 创建内存空间；
3. 创建数组元素并赋值。

Code: 一维数组创建声明和内存分配

```
1 int [] x; //声明名称为x的int型数组, 未分配内存给数组
2 x = new int[10]; //x中包含有10个元素, 并分配空间
```

```
1 int [] x = new int[10]; //声明数组并动态分配内存
```

说明

用 `new` 分配内存的同时, 数组的每个元素都会自动赋默认值, 整型为 0, 实数为 0.0, 布尔型为 `false`, 引用型为 `null`。

4.2.2 一维数组的初始化

若在声明数组时进行赋值即初始化, 称为静态内存分配。

```
1 数据类型[] 数组名 = {初值0, 初值1, ..., 初值n};
```

Code: 一维数组静态初始化

```
1 int [] a = {1,2,3,4,5};
```

注意

在 Java 程序中声明数组时, 无论用何种方式定义数组, 都不能指定其长度。

4.3 二维数组

Java 中无真正的多维数组, 只是数组的数组。

4.3.1 二维数组的声明和内存分配

```
1 数据类型[][] 数组名;
2 数组名 = new 数据类型 [行数][列数];
3 数据类型[][] 数组名 = new 数据类型 [行数][列数];
```

4.3.2 二维数组定义的含义

- Java 中的二维数组看作是由多个一维数组构成

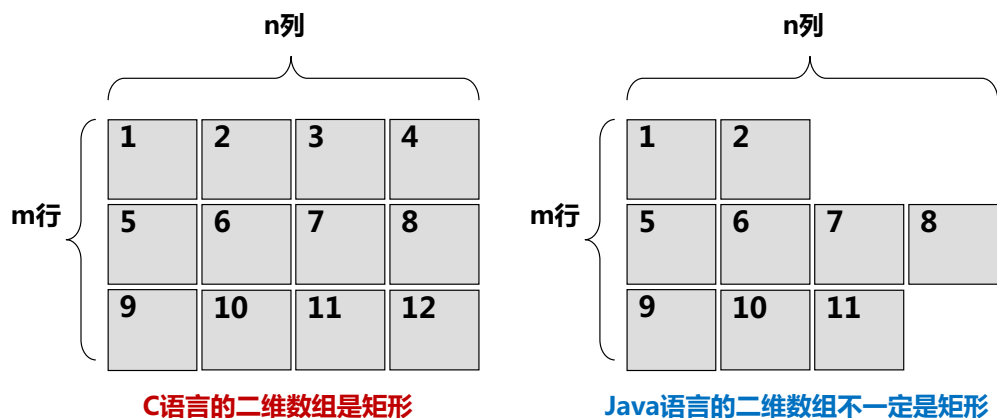


图 4.1 Java 版本迭代

- 二维数组申请内存必须指定**高层维数**

```
int[][] myArray1 = new int[10][];
int[][] myArray2 = new int[10][3];
```

- `int[][] x;`

表示定义了一个数组引用变量 `x`，第一个元素为 `x[0]`，最后一个为 `x[n-1]`，其长度不确定

- `x = new int[3][];`

表示数组 `x` 有三个元素，每个元素都是 `int[]` 类型的一维数组，分别为 `int x[0][]`、`int[] x[1]`、`int[] x[2]`

```
x[0] = new int[3]; x[1] = new int[2];
```

给 `x[0]`、`x[1]`、`x[2]` 赋值（长度可以不一样）

4.3.3 二维数组赋初值

```
1 int [][] a = {{11,22,33,44}, {66,77,88,99}};
```

注意

声明多维数组并初始化时不能指定其长度，否则出错。

4.4 Arrays 类

java.util.Arrays 工具类能方便地操作数组，它提供的所有方法都是静态的。该类具有以下功能：

给数组赋值 通过 fill 方法。

对数组排序 通过 sort 方法。

比较数组 通过 equals 方法比较数组中元素值是否相等。

查找数组元素 通过 binarySearch 方法能对排序好的数组进行二分查找法操作。

复制数组 把数组复制成一个长度为 length 的新数组。

Code: Array 操作示例

```
1  /*
2  * 数组比较 equals
3  */
4  String[] str1 = { "1", "2", "3" };
5  String[] str2 = { "1", "2", new String("3") };
6  System.out.println(Arrays.equals(str1, str2)); // 结果是true

8  /*
9  * 数组排序 sort
10 */
11 int[] score = { 79, 65, 93, 64, 88 };

13 // 将数组转换为字符串
14 String str = Arrays.toString(score);
15 System.out.println("原数组为: " + str);

17 Arrays.sort(score); // 作用是把一个数组按照有小到大进行排序，会改变原score
    而不是创建新对象

19 // 将数组转换为字符串
20 System.out.println("排序后数组为: " + Arrays.toString(score));

22 /*
23 * 把数组中的所有元素替换成一个值 fill
24 */
```



```

25 int [] num = { 1, 2, 3 };
26 Arrays.fill (num, 6); // 参数1: 数组对象; 参数2: 替换的值
27 System.out.println(Arrays.toString(num)); // 打印结果: [6, 6, 6]

29 /*
30 * 通过二分法查询元素值在数组中的下标 binarySearch
31 */
32 char [] a = { 'a', 'b', 'c', 'd', 'e' };
33 int i = Arrays.binarySearch(a, 'd');
34 System.out.println(i); // 结果是: 3

36 char [] b = { 'e', 'a', 'c', 'b', 'd' };
37 Arrays.sort(b);
38 int j = Arrays.binarySearch(b, 'e');
39 System.out.println(j); // 结果是: 4

41 /*
42 * 把数组内容复制到一个新数组中 copyOf
43 */
44 int [] c = { 1, 2, 3 };
45 int [] d = Arrays.copyOf(c, c.length + 2); // 参数1: 原数组 参数2: 新数组的长度
46 System.out.println("原数组为: " + Arrays.toString(c));
47 System.out.println("复制后的新数组为: " + Arrays.toString(d));

```

4.5 字符串

字符串是用一对双引号括起来的字符序列。Java 语言中，字符串常量或变量均用类实现。

4.5.1 字符串变量的创建

Code: 格式 1

```

1 String s; //声明字符串型引用变量s, 此时s的值为null
2 s = new String("Hello"); //在堆内存中分配空间, 并将s指向该字符串首地址

```

Code: 格式 2

```

1 String s = new String("Hello");

```

Code: 格式 3

```
1 String s = "Hello";
```

4.5.2 String 类的常用方法

Code: 求字符串长度

```
1 String str = new String("asdfzxc");  
2 int strlength = str.length(); //strlength = 7
```

Code: 获取字符串某一位置字符

```
1 char ch = str.charAt(4); //ch = z
```

Code: 提取子串

```
1 String str2 = str1.substring(2); //str2 = "dfzxc"  
2 String str3 = str1.substring(2,5); //str3 = "dfz"
```

Code: 字符串连接

```
1 String str = "aa".concat("bb").concat("cc");  
2 String str = "aa" + "bb" + "cc"; // 相当于上一行
```

Code: 字符串比较

```
1 String str1 = new String("abc");  
2 String str2 = new String("ABC");  
3 int a = str1.compareTo(str2); //a>0  
4 int b = str1.compareTo(str2); //b=0  
5 boolean c = str1.equals(str2); //c=false  
6 boolean d = str1.equalsIgnoreCase(str2); //d=true
```

Code: 字符串中字符的大小写转换

```
1 String str = new String("asDF");  
2 String str1 = str.toLowerCase(); //str1 = "asdf"
```

```
3 String str2 = str.toUpperCase(); //str2 = "ASDF"
```

Code: 字符串中字符的替换

```
1 String str = "asdzxcasd";  
2 String str1 = str.replace('a','g'); //str1 = "gsdzxcgsd"  
3 String str2 = str.replace("asd","fgh"); //str2 = "fghzxcfgh"  
4 String str3 = str.replaceFirst("asd","fgh"); //str3 = "fghzxcasd"  
5 String str4 = str.replaceAll("asd","fgh"); //str4 = "fghzxcfgh"
```

4.5.3 理解 Java 字符串

Code: String.java 部分代码

```
1 public final class String  
2 implements java.io. Serializable , Comparable<String>, CharSequence { //1  
  
4     /** The value is used for character storage. */  
5     private final char value []; //2  
  
7     /** The offset is the first index of the storage that is used. */  
8     private final int offset ;  
  
10    /** The count is the number of characters in the String. */  
11    private final int count;  
  
13    /** Cache the hash code for the string */  
14    private int hash; // Default to 0  
  
16    /** use serialVersionUID from JDK 1.0.2 for interoperability */  
17    private static final long serialVersionUID = -6849794470754667710L;  
18    .....  
  
20    public String substring(int beginIndex, int endIndex) { //3  
21        if (beginIndex < 0) {  
22            throw new StringIndexOutOfBoundsException(beginIndex);  
23        }  
24        if (endIndex > count) {  
25            throw new StringIndexOutOfBoundsException(endIndex);  
26        }  
27        if (beginIndex > endIndex) {
```

```
28         throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
29     }
30     return ((beginIndex == 0) && (endIndex == count)) ? this :
31         new String(offset + beginIndex, endIndex - beginIndex, value);
32     }
33 }
```

1. `String` 类是 `final` 类，即意味着 `String` 类不能被继承，并且它的成员方法都默认为 `final` 方法。
2. 从 `String` 类的成员属性可以看出 `String` 类其实是通过 `char` 数组来保存字符串的。
3. 无论是 `substring` 还是 `concat` 操作等都不是在原有的字符串上进行的，而是重新生成了一个新的字符串对象，最原始的字符串并没有被改变。

说明

`String` 对象一旦被创建就是固定不变的，对 `String` 对象的任何操作都不影响到原对象，而是会生成新的对象。

✧ 5 ✧ Java 面向对象编程进阶

基本信息

课程名称：Java 应用与开发

授课教师：王晓东

授课时间：第二周（根据校历，本周有两次课）

参考教材：本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握 Java 包、继承、访问控制、方法重写的概念、机制和使用方法
2. 理解 Java 关键字 super 和关键字 this，特别了解其指代的对象，编程中的用法

授课方式

理论课：多媒体教学、程序演示

实验课：上机编程

教学内容

5.1 包

为便于管理大型软件系统中数目众多的类，解决类的命名冲突问题以及进行访问控制，Java 引入包（package）机制，即将若干功能相关的类逻辑上分组打包到一起，提供类的多重类命名空间。

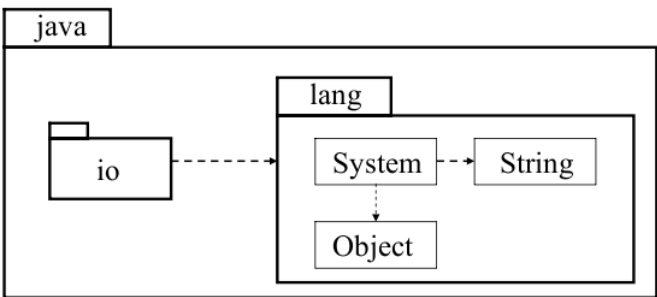


图 5.1 Java 包

5.1.1 JDK 常用包

JDK API 中的常用包如表所示。

表 5.1 JDK API 常用包

包名	功能说明	包的含义
java.lang	Java 语言程序设计的基础类	language 的简写
java.awt	创建图形用户界面和绘制图形图像的相关类	抽象窗口工具集
java.util	集合、日期、国际化、各种实用工具	utility 的简写
java.io	可提供数据输入/输出相关功能的类	input/output 的简写
java.net	Java 网络编程的相关功能类	网络
java.sql	提供数据库操作的相关功能类	结构化查询语言的简写

5.1.2 包的创建

package 语句作为 Java 源文件的第一条语句，指明该文件中定义的类所在的包（若缺省该语句，则指定为无名包）。语法格式如下：

```
1 package pkg1[.pkg2[.pkg3 ...]];
```

Code: 创建包

```
1 package p1;
2 public class Test {
3     public void m1() {
4         System.out.println("In class Test, method m1 is running!");
5     }
6 }
```

`package` 语句对所在源文件中定义的所有类型（包括接口、枚举、注解）均起作用。

Java 编译器把包对应于文件系统的目录管理，`package` 语句中，用“.”来指明包（目录）的层次。如果在程序 `Test.java` 中已定义了包 `p1`，编译时采用如下方式：

```
1 > javac Test.java
```

则编译器会在当前目录下生成 `Test.class` 文件。

若在命令行下使用如下命令：

```
1 > java -d /home/xiaodong/work01 Test.java
```

“-d /home/xiaodong/work01”是传给 Java 编译器的参数，用于指定此次编译生成的 `.class` 文件保存到该指定路径下，并且如果源文件中有 `package` 语句，则编译时会自动在目标路径下创建与包同名的目录 `p1`，再将生成的 `Test.class` 文件保存到该目录下。

5.1.3 导入包中的类

为使用定义在不同包中的 Java 类，需用 `import` 语句来引入所需要的类。语法格式：

```
1 import pkg1[.pkg2 ...](.classname|*);
```

Code: 导入和使用有名包中的类

```
1 import p1.Test; //or import p1.*;
2 public class TestPackage{
3     public static void main(String args[]){
4         Test t = new Test();
```

```
5     t.m1();
6 }
7 }
```

5.1.4 Java 包特性

一个类如果未声明为 **public** 的，则只能在其所在包中被使用，其他包中的类即使在源文件中使用 **import** 语句也无法引入它。可以不在源文件开头使用 **import** 语句导入要使用的有名包中的类，而是在程序代码中每次用到该类时都给出其完整的包层次，例如：

```
1 public class TestPackage{
2     public static void main(String args[]) {
3         p1.Test t = new p1.Test();
4         t.m1();
5     }
6 }
```

5.2 继承

5.2.1 继承的概念

继承（Inheritance）是面向对象编程的核心机制之一，其本质是在已有类型基础之上进行扩充或改造，得到新的数据类型，以满足新的需要。

根据需要定义 Java 类描述“人”和“学生”信息，示例代码如下：

Code: Class Person

```
1 public class Person {
2     public String name;
3     public int age;
4     public Date birthDate;
5     public String getInfo() {...}
6 }
```

Code: Class Student

```
1 public class Student {
2     public String name;
3     public int age;
```



```

4     public Date birthDate;
5     public String school;
6     public String getInfo() {...}
7 }

```

我们可以通过继承简化 `Student` 类的定义：

Code: Class `Student` extends `Person`

```

1     public class Student extends Person {
2         public String school;
3     }

```

Java 类声明的语法格式如下：

```

1     [< 修饰符 >] class < 类名 > [extends < 父类名 >] {
2         [< 属性声明 >]
3         [< 构造方法声明 >]
4         [< 方法声明 >]
5     }

```

`Object` 类是所有 `Java` 类的最高层父类，如果在类的声明中未使用 `extends` 关键字指明其父类，则默认父类为 `Object` 类。

`Java` 只支持单继承，不允许多重继承。即：

- 一个子类只能有一个父类；
- 一个父类可以派生出多个子类。

5.2.2 类之间的关系

依赖关系 一个类的方法中使用到另一个类的对象（`uses-a`）¹。

聚合关系 一个类的对象包含（通过属性引用）了另一个类的对象（`has-a`）²。

泛化关系 一般化关系（`is-a`），表示类之间的继承关系、类和接口之间的实现关系以及接口之间的继承关系。

¹车能够装载货物，车的装载功能（`load()` 方法）对货物（`goods`）有依赖。

²车有发动机、车轮等，`Car` 对象是由 `Engine` 等对象构成的。

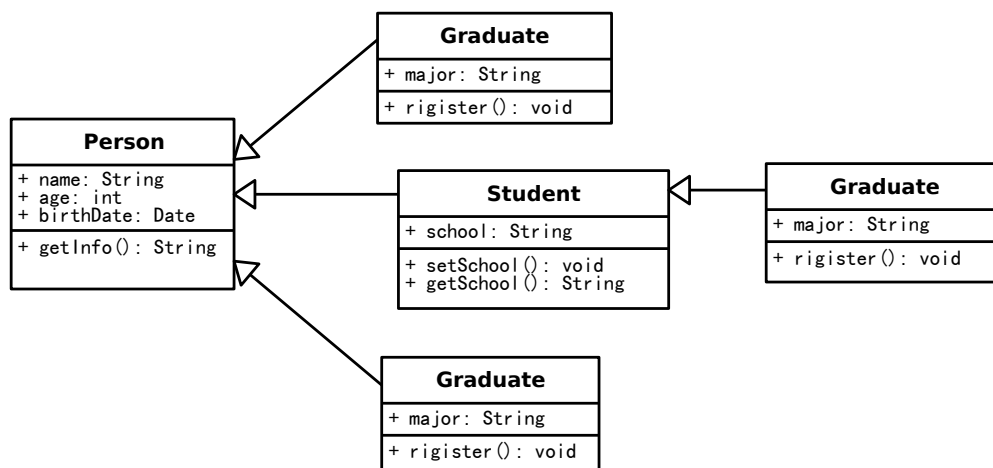


图 5.2 Java 包

5.3 访问控制

访问控制是指对 Java 类或类中成员的操作进行限制，即规定其在多大的范围内可以被直接访问。

5.3.1 类的访问控制

在声明 Java 类时可以在 class 关键字前使用 public 来修饰，也可以不使用该修饰符。public 的类可在任意场合被引入和使用，而非 public 的类只能在其所在包中被使用。

5.3.2 类中成员的访问控制

表 5.2 类成员的访问控制

修饰符/作用范围	同一个类	同一个包	子类	任何地方
public	yes	yes	yes	yes
protected	yes	yes	yes	no
无修饰符	yes	yes	no	no
private	yes	no	no	no

5.3.3 访问控制注意的一些问题

- 一般不提倡将属性声明为 `public` 的，而构造方法和需要外界直接调用的普通方法则适合声明为 `public` 的。
- 在位于不同的包内，必须是子类的对象才可以直接访问其父类的 `protected` 成员，而父类自身的对象反而不能访问其所在类中声明的 `protected` 成员。
- 所谓“访问控制”只是控制对 Java 类或类中成员的直接访问，而间接访问是不做控制的，也不该进行控制。

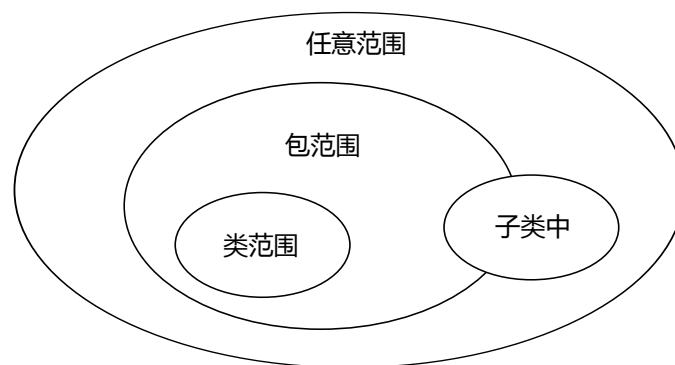


图 5.3 Java 访问控制

5.3.4 访问控制 `protected`

Code: A.java

```
1 package p1;
2 public class A {
3     public int m = 5;
4     protected int n = 6;
5 }
```

Code: B.java

```
1 package p2;
2 import p1.A;
3 public class B extends A {
4     public void mb() {
5         m = m + 1;
```

```

6      n = n * 2;
7  }

9  public static void main(String[] args) {
10     B b = new B();
11     b.m = 7; // 合法
12     b.n = 8; // 合法
13     A a = new A();
14     a.m = 9 // 合法
15     a.n = 10 // 非法
16 }
17 }

```

5.4 同名问题

5.4.1 方法重写

在子类中可以根据需要对从父类中继承来的方法进行重新定义, 此称方法重写 (Override) 或覆盖。

- 重写方法必须和被重写方法具有相同的方法名称、参数列表和返回值类型;
- 重写方法不能使用比被重写方法更严格的访问权限;
- 重写方法不允许声明抛出比被重写方法范围更大的异常类型。

Code: 方法重写示例: [Person.java](#)

```

1  public class Person {
2      String name;
3      int age;
4      public String getInfo() {
5          return "Name:" + name + "\t" + "age:" + age;
6      }
7  }

```

Code: 方法重写示例: [Student.java](#)

```

1  public class Student extends Person {
2      private String school;
3      public void setSchool(String scholl) {

```

```

4      this.school = school;
5  }
6  public String getSchool(){
7      return school;
8  }
9  public String getInfo() {
10     return "Name:" + name + "\tAge:" + age + "\tSchool:" + school;
11 }
12 }

```

Code: 方法重写示例: Parent.java

```

1  public class Parent {
2      public void method1() {...}
3  }

```

Code: 方法重写示例: Child.java

```

1  public class Child extends Parent {
2      private void method1() {} //非法, 权限更严格
3  }

```

5.4.2 同名属性

```

1  public class Person {
2      int age = 5;
3      public int getAge() {
4          return age;
5      }
6      public int getInfo() {
7          return age;
8      }
9  }

```

```

1  public class Student extends Person {
2      int age = 6;
3      public int getAge() {
4          return age;
5      }
6  }

```

```

1 public class Test {
2     public static void main(String args[]) {
3         Person p = new Person();
4         System.out.println(p.getAge());
5         Student s = new Student();
6         System.out.println(s.age);
7         System.out.println(s.getAge());
8         System.out.println(s.getInfo());
9     }
10 }

```

上述程序的输出结果为：

output

```

5
6
6
5

```

对上述 Student 对象同名属性的几点说明

1. 以“对象名. 属性名”方式直接访问时，使用的是子类中添加的属性 `age`；
2. 调用子类添加或者重写的方法时，方法中使用的是子类定义的属性 `age`；
3. 调用父类中定义的方法时，方法中使用的是父类中的属性 `age`，
4. 可以理解为“层次优先就近原则”，在哪个层次中的代码，就优先使用该层次类中定义的属性。不提倡使用同名属性。

5.4.3 关键字 `super`

在存在命名冲突（子类中存在方法重写或添加同名属性）的情况下，子类中的代码将自动使用子类中的同名属性或重写后的方法。当然也可以在子类中使用关键字 `super` 引用父类中的成分：

访问父类中定义的属性

```

1 super.<属性名>

```

调用父类中定义的成员方法

```
1 super.<方法名>(<实参列表>)
```

子类构造方法中调用父类的构造方法

```
1 super(<实参列表>)
```

`super` 的追溯不仅限于直接父类，而是先从直接父类开始查找，如果找不到则逐层上溯，一旦在某个层次父类中找到匹配成员即停止追溯并使用该成员。

Code: `super` 用法示例 A

```
1 class Animal {
2     protected int i = 1; //用于测试同名属性，无现实含义
3 }
4
5 class Person extends Animal {
6     protected int i = 2; //用于测试同名属性，无现实含义
7     private String name = "Tom";
8     private int age = 9;
9     public String getInfo() {
10         return "Name:" + name + "\tAge:" + age;
11     }
12     public void testI() {
13         System.out.println(super.i);
14         System.out.println(i);
15     }
16 }
```

Code: `super` 用法示例 B

```
1 class Student extends Person {
2     private int i = 3;
3     private String school = "THU";
4     public String getInfo() { //重写方法
5         return super.getInfo() + "\tSchool:" + school;
6     }
7     public void testI() { //重写方法
8         System.out.println(super.i);
9         System.out.println(i);
10    }
```

```

11 }
12 public class Test {
13     public static void main(String args[]) {
14         Person p = new Person();
15         System.out.println(p.getInfo());
16         p.testI();
17         Student s = new Student();
18         System.out.println(s.getInfo());
19         s.testI();
20     }
21 }

```

上述代码的输出结果为：

output

```

Name:Tom Age:9
1
2
Name:Tom Age:9 School:THU
2
3

```

5.4.4 关键字 this

在 Java 方法中，不但可以直接使用方法的局部变量，也可以使用调用该方法的对象。为解决可能出现的命名冲突，Java 语言引入 **this** 关键字来标明方法的当前对象。分为两种情况：

- 在普通方法中，关键字 **this** 代表方法的调用者，即本次调用了该方法的对象；
- 在构造方法中，关键字 **this** 代表该方法本次运行所创建的那个新对象。

this 作为一个特殊的引用类型变量，可以通过 “**this. 成员**” 的方式访问其引用的当前对象的属性和方法。

Code: **this** 用法示例

```

1 public class MyDate {
2     private int day = 17;
3     private int month = 2;

```



```
5     public MyDate(int day, int month) {  
6         this.day = day; // A  
7         this.month = month;  
8     }  
9     ... // Some methods  
  
11    public void setAll(int day, int month) {  
12        this.setMonth(month); // B  
13        this.setDay(day);  
14    }  
15 }
```

关于 this 的归纳说明

1. 在 Java 方法中直接给出变量名而不是“对象名.变量名”的方式访问一个变量，系统首先尝试作为局部变量来处理；如果方法中不存在该名字的局部变量，才会到方法当前对象的成员变量中查找。
2. 在 Java 方法中直接调用一个方法而不指定其调用者时，则默认调用者为当前对象 this。

实验设计

✎ 6 ✎ Java 面向对象编程进阶

基本信息

课程名称： Java 应用与开发

授课教师： 王晓东

授课时间： 第二周（根据校历，本周有两次课）

参考教材： 本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 理解多态和虚方法调用的概念，掌握其用法
2. 掌握方法重载的方法
3. 掌握 static 属性、方法和初始化块的用法
4. 了解设计模式，掌握单例设计模式
5. 掌握 final 关键字的概念和使用方法

授课方式

理论课： 多媒体教学、程序演示

实验课： 上机编程

教学内容

6.1 多态性

6.1.1 多态的概念

在 Java 中，子类的对象可以替代父类的对象使用称为**多态**。Java 引用变量与所引用对象间的类型匹配关系如下：

- 一个对象只能属于一种确定的数据类型，该类型自对象创建直至销毁不能改变。
- 一个引用类型变量可能引用（指向）多种不同类型的对象——既可以引用其声明类型的对象，也可以引用其声明类型的子类的对象。

```
1 Person p = new Student(); //Student 是 Person 的子类
```

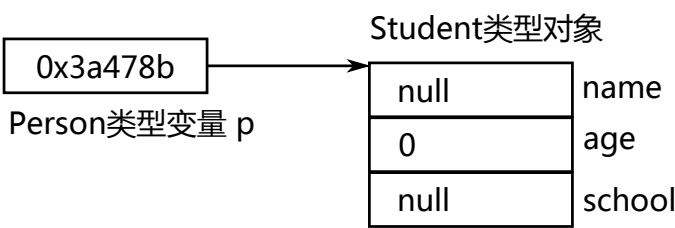


图 6.1 Java 多态

多态性同样适用与引用类型数组元素。

```
1 Person[] p = new Person[3];
2 p[0] = new Student(); // 假设 Student 类继承了 Person 类
3 p[1] = new Person();
4 p[2] = new Graduate(); //假设 Graduate 类继承了 Student 类
```

一个引用类型变量如果声明为父类的类型，但实际引用的是子类对象，该变量则不能再访问子类中添加的属性和方法，这体现了父类引用对子类对象的能力屏蔽性。

```
1 Student m = new Student();
2 m.setSchool("ouc"); // 合法
3 Person e = new Student();
```

```
4 e.setSchool("ouc"); // 非法
```

6.1.2 多态用法示例

Code: Person.java

```
1 public class Person {...}
```

Code: Student.java

```
1 public class Student extends Person {
2     private String school;

4     public void setSchool(String school) {
5         this.school = school;
6     }

8     public String getSchool() {
9         return school;
10    }

12    @Override
13    public String getInfo() {
14        return super.getInfo() + "\tSchool: " + school;
15    }
16 }
```

Code: PolymorphismSample.java

```
1 public class PolymorphismSample {
2     public void show(Person p) {
3         System.out.println(p.getInfo());
4     }

6     public static void main(String[] args) {
7         PolymorphismSample ps = new PolymorphismSample();
8         Person p = new Person();
9         ps.show(p);
10        Student s = new Student();
11        ps.show(s);
12    }
```

```
13 | }
```

多态提升方法通用性

以上代码中，`show()` 方法既可以处理 `Person` 类型的数据，又可以处理 `Student` 类型的数据，乃至未来定义的任何 `Person` 子类类型的数据，即不必为相关的每种类型单独声明一个处理方法，提高了代码的通用性。

6.1.3 虚方法调用

思考：一个引用类型的变量如果声明为父类的类型，但实际引用的是子类对象，则该变量就不能再访问子类中添加的属性和方法。但如果此时调用的是父类中声明过、且在子类中重写过的方法，情况如何？

补充代码。

6.1.4 对象造型

引用类型数据值之间的强制类型转换称为**造型**（Casting）。造型以下几种情况需要注意：

1. 从子类到父类的类型转换可以自动进行。

```
1 Person p = new Student();
```

2. 在多态的情况下，有时我们可能需要恢复一个对象的本来面目，以发挥其全部潜力。从父类到子类的类型转换必须通过造型实现。

```
1 Person p1 = new Student();  
2 Student s1 = (Student)p1; // 合法  
3 Person p2 = new Person();  
4 Student s2 = (Student)p2; // 非法
```

3. 无继承关系的引用类型间的转换是非法的。

```
1 String s = "Hello World!";  
2 Person p = (Person)s; // 非法
```

6.1.5 instanceof 运算符

如果运算符 `instanceof` 左侧的变量当前时刻所引用的对象的**真正类型**是其右侧给出的类型**或者是其子类**，则整个表达式的结果为 `true`。

```

1 class Person { --- }
2 class Student extends Person { --- }

4 public class Tool {
5     public void distribute(Person p) {
6         if (p instanceof Student) {
7             System.out.println("处理 Student 类型及其子类类型对象");
8         } else {
9             System.out.println("处理 Person 类型及其子类类型对象");
10        }
11    }
12 }


```

```

1 public class Test() {
2     public static void main(String[] args) {
3         Tool t = new Tool();
4         Student s = new Student();
5         t.distribute(t);
6     }
7 }

```

6.1.6 虚方法调用和造型

课程配套代码  package sample.oop.poly

- VirtualMethodSample.java
- Person.java
- Student.java

强调以下与虚方法调用和造型相关的重点：

- 系统根据运行时对象的真正类型来确定具体调用哪一个方法，这一机制被称为**虚方法调用**。
- 造型是引用类型数据值之间的强制类型转换。
- `instanceof` 运算符判断的是当前所引用对象的真正类型是什么，而不是声明的引用类型。

6.2 方法重载

6.2.1 方法重载的概念

在一个类中存在多个同名方法的情况称为**方法重载**（Overload）。Java 对方法重载有以下要求：

- 重载方法参数列表必须不同。
- 重载既可以用于普通方法，也可以用于构造方法。

课程配套代码  sample.oop.MethodOverloadSample.java

6.2.2 调用重载的构造方法

使用 `this` 调用当前类中重载构造方法

可以在构造方法的第一行使用关键字 `this` 调用其它（重载的）构造方法。

```
1 public class Person {  
2     ...  
3     public Person(String name,int age) {  
4         this.name = name;  
5         this.age = age;  
6     }  
7     public Person(String name) {  
8         this(name, 18);  
9     }  
10    ...  
11 }
```

注意

关键字 `this` 的此种用法只能用在构造方法中，且 `this()` 语句如果出现必须位于方法体中代码的第一行。

使用 `super` 调用父类构造方法

Code: [Person.java](#)


```

1 public class Person {
2     ... (此处没有无参构造方法)
3     public Person(String name, int age) {
4         this.name = name;
5         this.age = age;
6     }
7     ...
8 }

```

Code: Student.java

```

1 public class Student extends Person {
2     private String school;
3     public Student(String name, int age, String school) {
4         super(name, age); // 显式调用父类有参构造方法
5         this.school = school;
6     }
7     public Student(String school) { //编译出错
8         // super(); // 隐式调用父类有参构造方法，则自动调用父类无参构造方法
9         this.school = school;
10    }
11 }

```

上述代码为什么会编译出错？

在 Java 类的构造方法中一定直接或间接地调用了其父类的构造方法（Object 类除外）。

1. 在子类的构造方法中可使用 `super` 语句调用父类的构造方法，其格式为 `super(< 实参列表 >)`。
2. 如果子类的构造方法中既没有显式地调用父类构造方法，也没有使用 `this` 关键字调用同一个类的其他重载构造方法，则系统会默认调用父类无参数的构造方法，其格式为 `super()`。
3. 如果子类构造方法中既未显式调用父类构造方法，而父类中又没有无参的构造方法，则编译出错。

课程配套代码  sample.oop.ConstructorOverloadSample.java

6.2.3 对象构造/初始化细节

第一阶段 为新建对象的实例变量分配存储空间并进行默认初始化。

第二阶段 按下述步骤继续初始化实例变量：

1. 绑定构造方法参数；
2. 如有 `this()` 调用，则调用相应的重载构造方法然后跳转到步骤 5；
3. 显式或隐式追溯调用父类的构造方法（`Object` 类除外）；
4. 进行实例变量的显式初始化操作；
5. 执行当前构造方法的方法体中其余的语句。

6.3 关键字 `static`

在 Java 类中声明**属性、方法和内部类**时，可使用关键字 `static` 作为修饰符。

- `static` 标记的属性或方法由整个类（所有实例）共享，如访问控制权限允许，可不创建该类对象而直接用类名加“.”调用。
- `static` 成员也称“**类成员**”或“**静态成员**”，如“类属性”、“类变量”、“类方法”和“静态方法”等。

6.3.1 `static` 属性和方法

`static` 属性

- `static` 属性由其所在类（包括该类所有的实例）共享。
- 非 `static` 属性则必须依赖具体/特定的对象（实例）而存在。

`static` 方法

要在 `static` 方法中调用其所在类的非 `static` 成员，应首先创建一个该类的对象，通过该对象来访问其非 `static` 成员。

课程配套代码 ▶ `sample.oop.StaticMemberAndMethodSample.java`


6.3.2 初始化块

`static` 初始化块

在类的定义体中，方法的外部可包含 `static` 语句块，**`static` 块仅在其所属的类被载入时执行一次**，通常用于初始化 `static`（类）属性。

非 static 初始化块

非 static 的初始化块在创建对象时被自动调用。

课程配套代码  sample.oop.StaticInitBlockSample.java

6.3.3 静态导入

静态导入用于在一个类中导入其他类或接口中的 static 成员，语法格式如下：

```
1 import static <包路径>.<类名>.*
3 import static <包路径>.<类名>.<静态成员名>
```

Code: 静态导入应用示例

```
1 import static java.lang.Math.*;
2 public class Test {
3     public static void main(String[] args) {
4         double d = sin(PI * 0.45);
5         System.out.println(d);
6     }
7 }
```

6.3.4 Singleton 设计模式

所谓“模式”就是被验证为有效的常规问题的典型解决方案。设计模式（Design Pattern）在面向对象分析设计和软件开发中占有重要地位。好的设计模式可以使我们的更加方便的重用已有的成功设计和体系结构，极大的提高代码的重用性和可维护性。

经典设计模式分类主要分为以下三大类：

创建型模式 涉及对象的实例化，特点是不让用户代码依赖于对象的创建或排列方式，避免用户直接使用 new 创建对象。

工厂方法模式、抽象工厂方法模式、生成器模式、原型模式和单例模式

行为型模式 涉及怎样合理的设计对象之间的交互通信，以及合理为对象分配职责，让设计富有弹性、易维护、易复用。

责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式和访问者模式

结构型模式 涉及如何组合类和对象以形成更大的结构，和类有关的结构型模式涉及如何合理使用继承机制，和对象有关的结构型模式涉及如何合理的使用对象组合机制。

适配器模式、组合模式、代理模式、享元模式、外观模式、桥接模式和装饰模式

Singleton 设计模式也称“单子模式”或“单例模式”。

采用调试方式讲解示例代码

Singleton 代码的特点包括以下几个方面：

1. 使用静态属性 `onlyone` 来引用一个“全局性”的 Single 实例。
2. 将构造方法设置为 `private` 的，这样在外界将不能再使用 `new` 关键字来创建该类的新实例。
3. 提供 `public static` 的方法 `getSingle()` 以使外界能够获取该类的实例，达到全局可见的效果。

在任何使用到 Single 类的 Java 程序中（这里指的是一次运行中），需要确保只有一个 Single 类的实例存在（如 Web 应用 `ServletContext` 全局上下文对象），则使用该模式。

6.4 关键字 final

在声明 Java 类、变量和方法时可以使用关键字 `final` 来修饰，以使其具有“终态”的特性：

1. `final` 标记的类不能被继承；
2. `final` 标记的方法不能被子类重写；
3. `final` 标记的变量（成员变量或局部变量）即成为常量，只能赋值一次；
4. `final` 标记的成员变量必须在声明的同时或在每个构造方法中显式赋值，然后才能使用；
5. `final` 不允许用于修饰构造方法、抽象类以及抽象方法。

关键字 `final` 应用举例如下：

```
1 public final class Test {  
2     public static int totalNumber = 5;  
3     public final int id;  
4     public Test() {  
5         id = ++totalNumber; // 赋值一次  
6     }  
7     public static void main(String[] args) {  
8         Test t = new Test();  
9         System.out.println(t.id);  
10        final int i = 10;  
11        final int j;  
12        j = 20;  
13        j = 30; // 非法  
14    }  
15 }
```

实验设计

✚ 7 ✚ Java 内存模型与分配机制

基本信息

课程名称：Java 应用与开发

授课教师：王晓东

授课时间：第二周（根据校历，本周有两次课）

参考教材：本课程参考教材及资料如下：

- 陈国君主编，Java 程序设计基础（第 5 版），清华大学出版社，2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 理解 JVM 内存模型，掌握 JVM 内存构成
2. 理解 Java 程序的运行过程，学会通过调试模式观察内存的变化
3. 了解 Java 内存管理，认识垃圾回收
4. 建立编程时高效利用内存、避免内存溢出的理念

授课方式

理论课：多媒体教学、程序演示

实验课：上机编程

教学内容

7.1 Java 内存模型

7.1.1 Java 虚拟机（Java Virtual Machine, JVM）

Java 虚拟机的架构如图7.1所示。

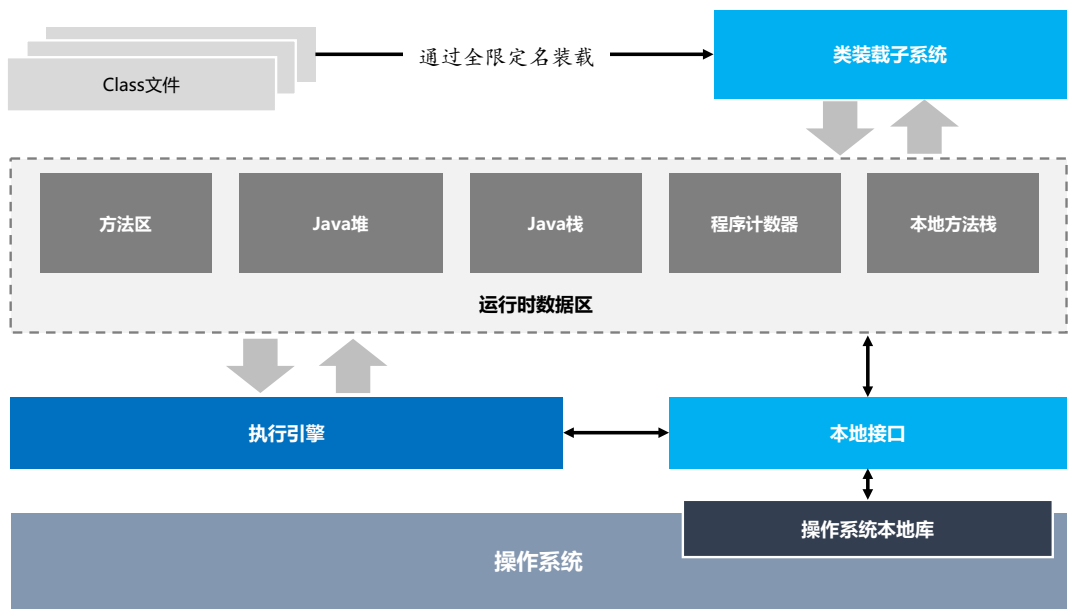


图 7.1 Java 虚拟机架构

- Java 程序运行在 JVM 上，JVM 是程序与操作系统之间的桥梁。
- JVM 实现了 Java 的平台无关性。
- JVM 是内存分配的前提。

7.1.2 JVM 内存模型

Java 程序运行过程会涉及的内存区域包括：

程序计数器 当前线程执行的字节码的行号指示器。

栈 保存局部变量的值，包括：用来保存基本数据类型的值；保存类的实例，即堆区对象的引用（指针），也可以用来保存加载方法时的帧。（Stack）

JVM内存模型

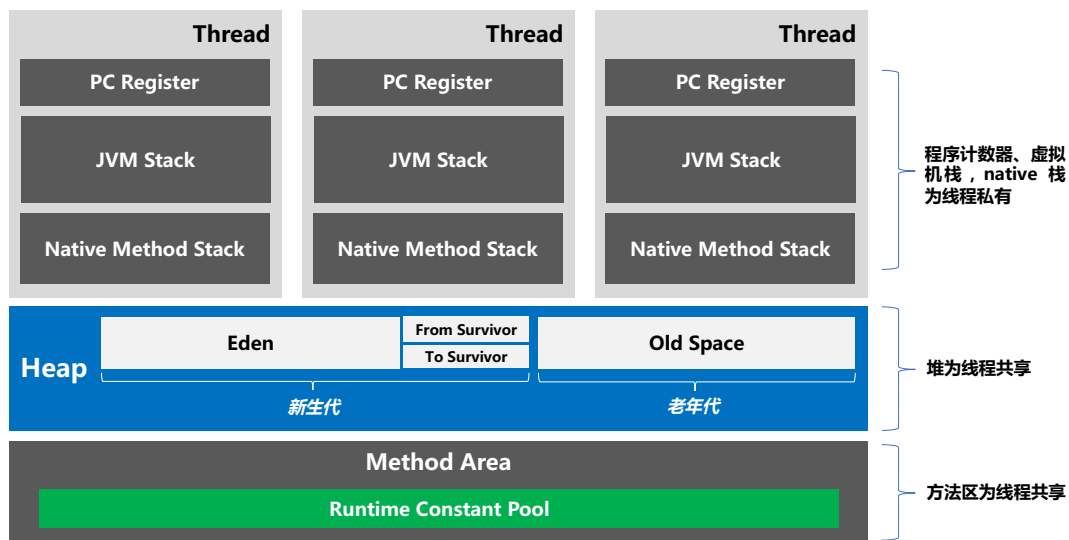


图 7.2 JVM 内存模型

堆 用来存放动态产生的数据，如 new 出来的对象和数组¹。（Heap）

常量池 JVM 为每个已加载的类型维护一个常量池，常量池就是这个类型用到的常量的一个有序集合。包括直接常量（基本类型、String）和对其他类型、方法、字段的符号引用。池中的数据和数组一样通过索引访问，常量池在 Java 程序的动态链接中起了核心作用。（Perm）

代码段 存放从硬盘上读取的源程序代码。（Perm）

数据段 存放 static 定义的静态成员。（Perm）

7.2 Java 程序内存运行分析

7.2.1 预备知识和所用讲解程序

1. 一个 Java 文件，只要有 main 入口方法，即可认为这是一个 Java 程序，可以单独编译运行。

¹注意创建出来的对象只包含属于各自的成员变量，并不包括成员方法。因为同一个类的对象拥有各自的成员变量，存储在各自的堆内存中，但是他们共享该类的方法，并不是每创建一个对象就把成员方法复制一次。

2. 无论是普通类型的变量还是引用类型的变量（俗称实例），都可以作为局部变量，他们都可以出现在栈中。
3. 普通类型的变量在栈中直接保存它所对应的值，而引用类型的变量保存的是一个指向堆区的指针。通过这个指针，就可以找到这个实例在堆区对应的对象。因此，普通类型变量只在栈区占用一块内存，而引用类型变量要在栈区和堆区各占一块内存。

Code: Test.java

```
1 public class Test {  
2     public static void main(String[] args) {  
3         Test test = new Test(); //1  
4         int data = 9; //2  
5         BirthDate d1 = new BirthDate(22, 12, 1982); //3  
6         BirthDate d2 = new BirthDate(10, 10, 1958); //4  
7         test.m1(data); //5  
8         test.m2(d1); //7  
9         test.m3(d2);  
10    }  
  
12    public void m1(int i) {  
13        i = 1234; //6  
14    }  
15    public void m2(BirthDate b) {  
16        b = new BirthDate(15, 6, 2010); //8  
17    }  
18    public void m3(BirthDate b) {  
19        b.setDay(18);  
20    }  
21 }
```

7.2.2 程序调用过程

程序调用过程（一）

- JVM 自动寻找 main 方法，执行第一句代码，创建一个 Test 类的实例，在栈中分配一块内存，存放一个指向堆区对象的指针 110925。
- 创建一个 int 型的变量 data，由于是基本类型，直接在栈中存放 data 对应的值 9。

- 创建两个 BirthDate 类的实例 d1、d2，在栈中分别存放了对应的指针指向各自的对象。它们在实例化时调用了有参数的构造方法，因此对象中有自定义初始值。

程序调用过程（二）

- 调用 test 对象的 m1 方法，以 data 为参数。JVM 读取这段代码时，检测到 i 是局部变量，则会把 i 放在栈中，并且把 data 的值赋给 i。

程序调用过程（三）

- 把 1234 赋值给 i。

程序调用过程（四）

- m1 方法执行完毕，立即释放局部变量 i 所占用的栈空间。

程序调用过程（五）

- 调用 test 对象的 m2 方法，以实例 d1 为参数。JVM 检测到 m2 方法中的 b 参数为局部变量，立即加入到栈中，由于是引用类型的变量，所以 b 中保存的是 d1 中的指针，此时 b 和 d1 指向同一个堆中的对象。在 b 和 d1 之间传递是指针。

程序调用过程（六）

- m2 方法中又实例化了一个 BirthDate 对象，并且赋给 b。在内部执行过程是：在堆区 new 了一个对象，并且把该对象的指针保存在栈中 b 对应空间，此时实例 b 不再指向实例 d1 所指向的对象，但是实例 d1 所指向的对象并无变化，未对 d1 造成任何影响。

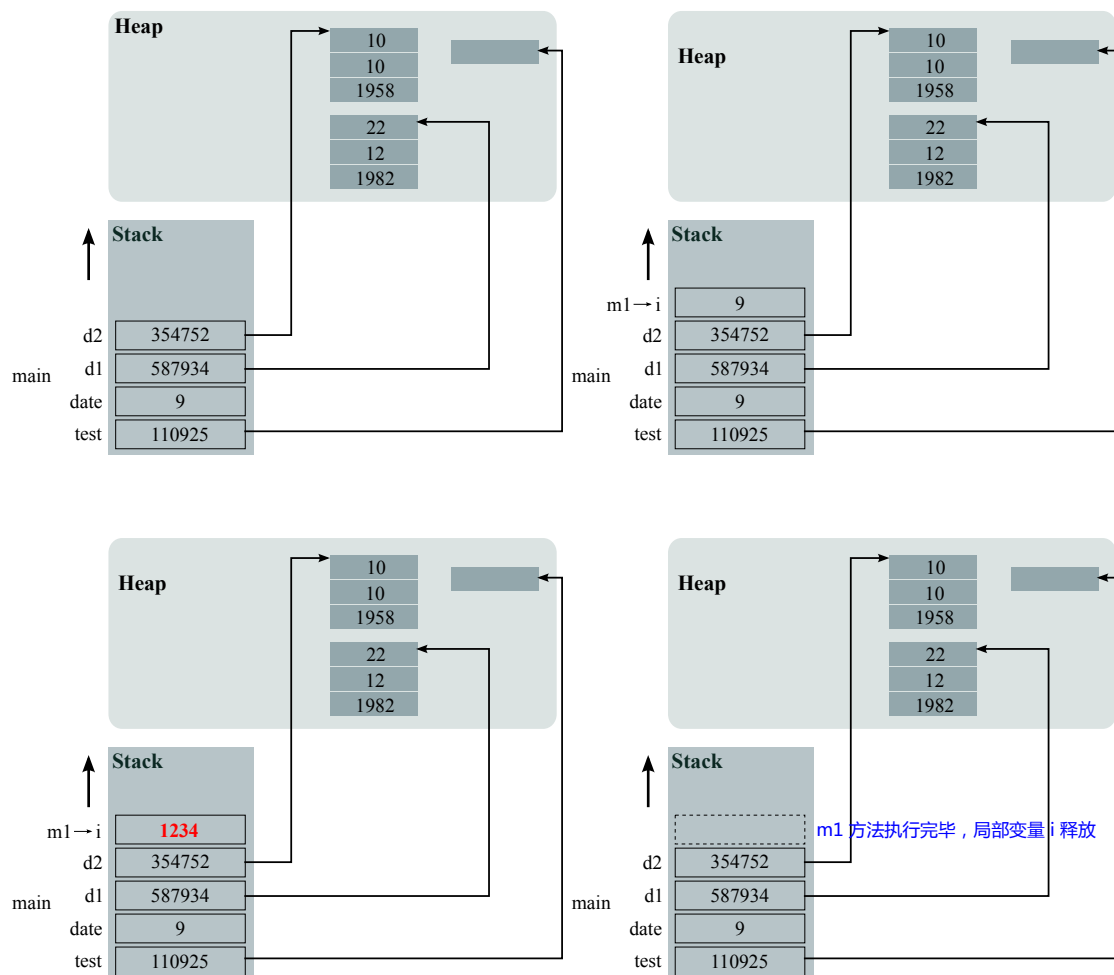
程序调用过程（七）

- m2 方法执行完毕，立即释放局部引用变量 b 所占的栈空间，注意只是释放了栈空间，堆空间要等待自动回收。

程序调用过程（八）

- 调用 test 实例的 m3 方法，以实例 d2 为参数。JVM 会在栈中为局部引用变量 b 分配空间，并且把 d2 中的指针存放在 b 中，此时 d2 和 b 指向同一个对象。再调用实例 b 的 setDay 方法，其实就是调用 d2 指向的对象的 setDay 方法。

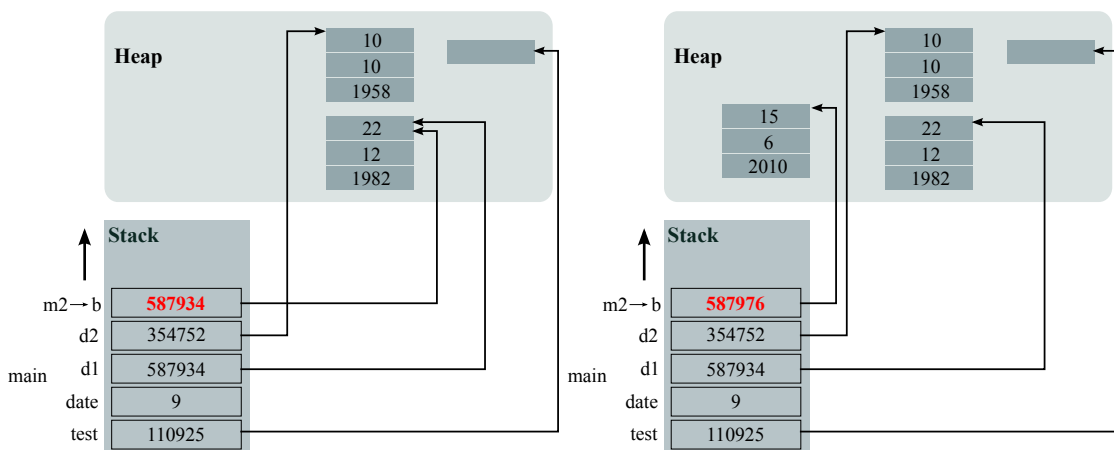
- 调用实例 b 的 setDay 方法会影响 d2，因为二者指向的是同一个对象。
- m3 方法执行完毕，立即释放局部引用变量 b。



7.2.3 Java 程序运行内存分析小结

- 基本类型和引用类型，二者作为局部变量时都存放在栈中。
- 基本类型直接在栈中保存值，引用类型在栈中保存一个指向堆区的指针，真正的对象存放在堆中。
- 作为参数时基本类型就直接传值，引用类型传指针。

注意什么是对象



```
1 MyClass a = new MyClass();
```

此时 `a` 是指向对象的指针，而不能说 `a` 是对象。指针存储在栈中，对象存储在堆中，操作实例实际上是通过指针间接操作对象。多个指针可以指向同一个对象。

- **栈中的数据和堆中的数据销毁并不是同步的。**方法一旦执行结束，栈中的局部变量立即销毁，但是堆中对象不一定销毁。因为可能有其他变量也指向了这个对象，直到栈中没有变量指向堆中的对象时，它才销毁；而且还不是马上销毁，要等垃圾回收扫描时才可以被销毁。
- **栈、堆、代码段、数据段等都是相对于应用程序而言的。**每一个应用程序都对应唯一的一个 JVM 实例，每一个 JVM 实例都有自己的内存区域，互不影响，并且这些内存区域是该 JVM 实例所有线程共享的。

7.3 Java 内存管理建议

7.3.1 Java 垃圾回收机制

JVM 的垃圾回收机制（GC）决定对象是否是垃圾对象，并进行回收。垃圾回收机制的特点包括：

- 垃圾内存并不是用完了马上就被释放，所以会产生内存释放不及时的现象，从而降低内存的使用效率。而当程序庞大的时候，这种现象更为明显。
- 垃圾回收工作本身需要消耗资源，同样会产生内存浪费。

JVM 中的对象生命周期一般分为 7 个阶段：①创建阶段、②应用阶段、③不可视阶段、④不可到达阶段、⑤可收集阶段、⑥终结阶段、⑦释放阶段。

Java 需要内存管理，在 JVM 中运行的对象的整个生命周期中，进行人为的内存管理是必要的，主要原因体现在：

- 虽然 JVM 已经代替开发者完成了对内存的管理，但是硬件本身的资源是有限的。
- 如果 Java 的开发人员不注意内存的使用依然会造成较高的内存消耗，导致性能的降低。

7.3.2 JVM 内存溢出和参数调优

当遇到 `OutOfMemoryError` 时该如何做？

- 常见的 OOM（Out Of Memory）内存溢出异常，就是堆内存空间不足以存放新对象实例时导致。
- 永久区内存溢出相对少见，一般是由于需要加载海量的 Class 数据，超过了非堆内存的容量导致。通常出现在 Web 应用刚刚启动时。因此 Web 应用推荐使用预加载机制，方便在部署时就发现并解决该问题。
- 栈内存也会溢出，但是更加少见。

对内存溢出的处理方法不外乎这两种：❶ 调整 JVM 内存配置；❷ 优化代码。
创建阶段的 JVM 内存配置优化需要关注以下项：

堆内存优化 调整 JVM 启动参数 `-Xms -Xmx -XX:newSize -XX:MaxNewSize`，如调整初始堆内存和最大对内存 `-Xms256M -Xmx512M`。或者调整初始 New Generation 的初始内存和最大内存 `-XX:newSize=128M -XX:MaxNewSize=128M`。

永久区内存优化 调整 `PermSize` 参数，如 `-XX:PermSize=256M -XX:MaxPermSize=512M`。

栈内存优化 调整每个线程的栈内存容量，如 `-Xss2048K`。

7.3.3 内存优化的示例

减少无谓的对象引用创建

[Code: Test 1](#)

```

1  for(int i=0; i<10000; i++) {
2      Object obj = new Object();
3  }

```

Code: Test 2

```

1  Object obj = null;
2  for( int i=0; i<10000; i++) {
3      obj = new Object();
4  }

```

内存性能分析

Test 2 比 Test 1 的性能要好。两段程序每次执行 for 循环都要创建一个 Object 的临时对象，JVM 的垃圾回收不会马上销毁但这些临时对象。相对于 Test 1，Test 2 则只在栈内存中保存一份对象的引用，而不必创建大量新临时变量，从而降低了内存的使用。

不要对同一对象初始化多次

```

1  public class A {
2      private Hashtable table = new Hashtable();
3      public A() {
4          table = new Hashtable();
5      }
6  }

```

内存性能分析

上述代码 new 了两个 Hashtable，但是却只使用了一个，另外一个则没有被引用而被忽略掉，浪费了内存。并且由于进行了两次 new 操作，也影响了代码的执行速度。另外，不要提前创建对象，尽量在需要的时候创建对象。

7.3.4 对象其他生命周期阶段内存管理

应用 即该对象至少有一个引用在维护它。

不可视 即超出该变量的作用域。

因为 JVM GC 并不是马上进行回收，而是要判断对象是否被其他引用维护。所以，如果我们在使用完一个对象以后对其进行 `obj = null` 或者

obj.doSomething() 操作，将其标记为空，则帮助 JVM 及时发现这个垃圾对象。

不可到达 即在 JVM 中找不到对该对象的直接或者间接的引用。

可收集，终结，释放 垃圾回收器发现该对象不可到达，finalize 方法已经被执行，或者对象空间已被重用的时候。

Java 的 finalize() 方法

Java 所有类都继承自 Object 类，而 finalize() 是 Object 类的一个函数，该函数在 Java 中类似于 C++ 的析构函数（仅仅是类似）。一般来说可以通过重载 finalize() 的形式来释放类中对象。

```
1 public class A {  
2     public Object a;  
  
4     public A() {  
5         a = new Object();  
6     }  
  
8     protected void finalize () throws java.lang.Throwable {  
9         a = null; // 标记为空，释放对象  
10        super. finalize (); // 递归调用超类中的 finalize 方法  
11    }  
12 }
```

什么时候 finalize() 被调用由 JVM 来决定。尽量少用 finalize() 函数，finalize() 函数是 Java 提供给程序员一个释放对象或资源的机会。但它会加大 GC 的工作量，因此尽量少采用 finalize 方式回收资源。

- 一般的，纯 Java 编写的 Class 不需要重写 finalize() 方法，因为 Object 已经实现了一个默认的，除非我们要实现特殊的功能。
- 用 Java 以外的代码编写的 Class(比如 JNI、C++ 的 new 方法分配的内存)，垃圾回收器并不能对这些部分进行正确的回收，这就需要我们覆盖默认的方法来实现对这部分内存的正确释放和回收。

实验设计

▣ 8 ▣ 高级类特性

基本信息

课程名称： Java 应用与开发

授课教师： 王晓东

授课时间： 第四周

参考教材： 本课程参考教材及资料如下：

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握抽象类和接口的概念、特性及定义方法
2. 理解抽象类和接口的异同和作用
3. 了解嵌套类的分类, 掌握嵌套类中静态嵌套类和匿名嵌套类的概念
4. 掌握匿名内部类的特征、继承和接口实现的用法
5. 掌握枚举类型的使用方法

授课方式

理论课： 多媒体教学、程序演示

实验课： 上机编程

教学内容

8.1 抽象类

8.1.1 抽象类的概念

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的。如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类往往用来表征对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同但是本质上相同的具体概念的抽象。

8.1.2 定义抽象类

- 在定义 Java 方法时可以只给出方法头，而不必给出方法的实现细节，这样的方法被称为**抽象方法**。
- 抽象方法必须用关键字**abstract**修饰。
- 包含抽象方法的类必须声明为抽象类，用关键字**abstract**修饰。

Code: 抽象类示例

```
1 public abstract class Animal { //定义为抽象类
2     private int age;
3
4     public void setAge(int age) {
5         this.age = age;
6     }
7
8     public int getAge(){
9         return age;
10    }
11
12    public abstract void eat(); //抽象方法
13 }
```

Code: 抽象类继承

```

1 public class Person extends Animal {
2     private String name;
3     public void setName(String name) {
4         this.name = name;
5     }
6     public String getName() {
7         return name;
8     }
9     public void eat() { //重写方法
10        System.out.println("洗手→烹饪→摆餐具→吃喝→收摊儿");
11    }
12 }

```

```

1 public class Bird extends Animal {
2     public void fly(){
3         System.out.println("我心飞翔!");
4     }
5     public void eat(){ //重写方法
6         System.out.println("直接吞食!");
7     }
8 }

```

8.1.3 抽象类的特性与作用

抽象类的特性

- 子类必须实现其父类中的所有抽象方法，否则该子类也只能声明为抽象类。
- 抽象类不能被实例化。**问题** 抽象类能否有构造方法？

抽象类的作用

抽象类主要是通过继承由其子类发挥作用，包括两方面：

代码重用 子类可以重用抽象类中的属性和非抽象方法。

规划 子类中通过抽象方法的重写来实现父类规划的功能。

抽象类的其他特性

- 抽象类中可以不包含抽象方法。主要用于当一个类已经定义了多个更适用的子类时，为避免误用功能相对较弱的父类对象，干脆限制其实例化。

- 子类中可以不全实现抽象父类中的抽象方法，但此时子类也只能声明为抽象类。
- 父类不是抽象类，但在子类中可以添加抽象方法，此情况下子类必须声明为抽象类。
- 多态性对于抽象类仍然适用，可以将引用类型变量（或方法的形参）声明为抽象类的类型。
- 抽象类中可以声明 `static` 属性和方法，只要访问控制权限允许，这些属性和方法可以通过 `<类名>.<类成员>` 的方法进行访问。

8.2 接口

8.2.1 接口（interface）的概念

在科技辞典中，“接口”被解释为“两个不同系统（或子程序）交接并通过它彼此作用的部分。在 Java 语言中，通过接口可以了解对象的交互界面，即明确对象提供的功能及其调用格式，而不需要了解其实现细节。

接口是抽象方法和常量值的定义的集合。从本质上讲，接口是一种特殊的抽象类，这种抽象类中只包含常量定义和方法声明，而没有变量和方法的实现。

8.2.2 定义接口

接口中定义的属性必须是 `public static final` 的，而接口中定义的方法则必须是 `public abstract` 的，因此这些关键字可以部分或全部省略。

Code: 接口示例（未简化）

```

1 public interface Runner {
2     public static final int id = 1;
3     public abstract void start();
4     public abstract void run();
5     public abstract void stop();
6 }
```

Code: 与上述代码等价的标准定义

```

1 public interface Runner {
2     int id = 1;
```

```

3     void start();
4     void run();
5     void stop();
6 }

```

8.2.3 接口的实现

和继承关系类似，类可以**实现**接口，且接口和实现类之间也存在多态性。类继承和接口实现的语法格式如下：

```

1  [<modifier>] class <name> [extends <superclass>] [implements <interface> [,<
    interface>]* ] {
2      <declarations>*
3  }

```

Code: 接口实现示例

```

1  public class Person implements Runner {
2      public void start() {
3          System.out.println("弯腰、蹬腿、咬牙、瞪眼、开跑");
4      }
5      public void run(){
6          System.out.println("摆动手臂、维持直线方向");
7      }
8      public void stop(){
9          System.out.println("减速直至停止、喝水");
10     }
11 }

```

通过接口可以指明多个类需要实现的方法，而这些类还可以根据需求继承各自的父类。或者说，**通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。**

类允许实现多重接口

课程配套代码  package sample.advance.interfacesample

8.2.4 接口间的继承

与接口的多重实现情况类似，由于不担心方法追溯调用上的不确定性，接口之间的继承允许“多重继承”的情况。

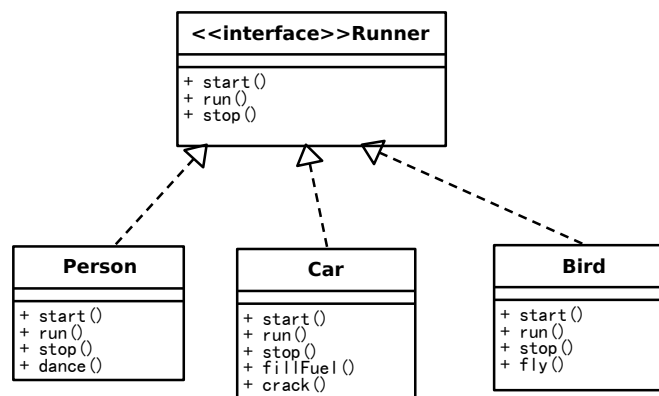


图 8.1 接口实现示例

```

1  interface A {
2      public void ma();
3  }
4  interface B {
5      public int mb(int i);
6  }
7  interface C extends A,B { //接口的多重继承
8      public String mc();
9  }
10 class D implements C {
11     public void ma() {
12         System.out.println("Implements method ma()!");
13     }
14     public int mb(int i) {
15         return 2000 + i;
16     }
17     public String mc() {
18         return "Hello!";
19     }
20 }
  
```

上述代码中的 D 类缺省继承了 Object 类，直接实现了接口 C，间接实现了接口 A 和 B，由于多态性的机制，将来 D 类的对象可以当作 Object、C、A 或 B 等类型使用。

8.2.5 接口特性总结

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。

- 接口可以被多重实现。
- 接口可以继承其它的接口，并添加新的属性和抽象方法，接口间支持多重继承。

8.3 抽象类和接口剖析

8.3.1 语法层面的区别

概念差异——语法差异——用法差异——设计哲学

- 抽象类可以提供成员方法的实现细节，而接口中只能存在 `public abstract` 方法
- 抽象类中的成员变量可以为各种类型，而接口中的成员变量只能是 `public static final` 类型
- 抽象类可以有静态代码块和静态方法，接口中不能含有静态代码块以及静态方法
- 一个类只能继承一个抽象类，而一个类却可以实现多个接口

8.3.2 设计层面的区别

- 抽象类是对类的抽象（可以抽象但不宜实例化），而接口是对行为的抽象。
- 抽象类是对类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。
- 抽象类作为很多子类的父类，它是一种模板式设计。**模板式设计：模板代表公共部分，公共部分需要改的则改动模板即可。**
- 而接口是一种行为规范，它是一种辐射式设计。**辐射式设计：接口进行了变更，则所有实现类都必须进行相应的改动。**

8.3.3 怎样才是合理的设计？（门和警报的示例）

以门和警报设计作为示例，一般来说，门都有 `open()` 和 `close()` 这两个动作。通过抽象类和接口来定义这个抽象概念：


```
1 abstract class Door {  
2     public abstract void open();  
3     public abstract void close();  
4 }
```

```
1 interface Door {  
2     public abstract void open();  
3     public abstract void close();  
4 }
```

问题

如果现在我们需要门具有报警 `alarm()` 的功能该如何实现？

思路一

将这三个功能都放在抽象类里面，这样一来所有继承这个抽象类的子类都具备了报警功能，但是有的门并不一定需要具备报警功能。**不合理抽象**

思路二

将这三个功能都放在接口里面，但需要用到报警功能的类就需要实现这个接口中的 `open()` 和 `close()`，也许这个类根本就不具备 `open()` 和 `close()` 这两个功能，比如火灾报警器。**不合理规划**

`Door` 的 `open()`、`close()` 和 `alarm()` 根本就属于两个不同范畴内的行为：

- `open()` 和 `close()` 属于门本身固有的行为特性。
- `alarm()` 属于延伸的附加行为。

更为合理的思路

❶ 单独将报警设计为一个接口，包含 `alarm()` 行为；❷ `Door` 设计为单独的抽象类，包含 `open()` 和 `close()` 两种行为；❸ 设计一个报警门继承 `Door` 类和实现 `Alarm` 接口。

课程配套代码 ▶ `package sample.advance.door`

8.4 嵌套类

8.4.1 什么是嵌套类

Java 语言支持类的嵌套定义，即允许将一个类定义在其他类的内部，其中内层的类被称为嵌套类。

嵌套类的分类

静态嵌套类 (Static Nested Class) 使用 static 修饰的嵌套类

内部类 (Inner Class) 非 static 的嵌套类

普通内部类 在类中的方法或语句块外部定义的非 static 类。

局部内部类 定义在方法或语句块中的类，也称局部类。

匿名内部类 定义在方法或语句块中，该类没有名字、只能在其所在之处使用一次。

8.4.2 静态嵌套类

静态嵌套类的特征

- 静态嵌套类不再依赖/引用外层类的特定对象，只是隐藏在另一个类中而已。
- 由于静态嵌套类的对象不依赖外层类的对象而独立存在，因而可以直接创建，进而也就无法在静态嵌套类中直接使用其外层类的非 static 成员。

课程配套代码  sample.advance.nestedclass.StaticNestedClassSample.java

8.4.3 匿名内部类

匿名内部类是局部类的一种简化。

当我们只在一处使用到某个类型时，可以将之定义为局部类，进而如果我们只是创建并使用该类的一个实例的话，那么连类的名字都可以省略。

8.4.4 使用匿名内部类

Code: [Person.java](#)

```

1  public abstract class Person {
2      private String name;
3      private int age;

5      public Person() {}

7      public Person(String name, int age) {
8          this.name = name;
9          this.age = age;
10     }

12     public String getInfo() {
13         return "Name: " + name + "\t Age: " + age;
14     }

16     public abstract void work();
17 }

```

Code: TestAnonymous.java

```

1  public class TestAnonymous {
2      public static void main(String[] args) {
3          Person sp = new Person() { // 匿名内部类
4              public void work() {
5                  System.out.println("个人信息: " + this.getInfo());
6                  System.out.println("I am sailing.");
7              }
8          };

10         sp.work();
11     }
12 }

```

对上述代码的解释如下：

定义一个新的 Java 内部类，该类本身没有名字，但继承了指定的父类 Person，并在此匿名子类中重写了父类的 work() 方法，然后立即创建了一个该匿名子类的对象，再将其地址保存到引用变量 sp 中待用。

由于匿名类没有类名，而构造方法必须与类同名，所以匿名类不能显式的定义构造方法，但系统允许在创建匿名类对象时将参数传给父类构造方法（使用父类的构造方法）。

```

1 Person sp = new Person("Kevin", 30) {
2     public void work() {
3         System.out.println("个人信息: " + this.getInfo());
4         System.out.println("I am sailing.");
5     }
6 };

```

匿名类除了可以继承现有父类之外，还可以实现接口，但不允许实现多个接口，且实现接口时就不能再继承父类了，反之亦然。

Code: [Swimmer.java](#)

```

1 public interface Swimmer {
2     public abstract void swim();
3 }

```

Code: [TestAnonymous2.java](#)

```

1 public class TestAnonymous2 {
2     public static void main(String[] args) {
3         TestAnonymous2 ta = new TestAnonymous2();
4         ta.test(new Swimmer() { // 匿名类实现接口
5             public void swim() {
6                 System.out.println("I am swimming.");
7             }
8         });
9
10        public void test(Swimmer swimmer) {
11            swimmer.swim();
12        }
13    }
14 }

```

8.5 枚举类型

8.5.1 枚举类型的概念

Java SE 5.0 开始，引入了一种新的引用数据结构**枚举类型**。枚举类型均自动继承 `java.lang.Enum` 类，使用一组常量值来表示特定的数据集合，该集合中数据的数目确定（通常较少），且这些数据只能取预先定义的值。

```

1 public enum Week {
2     MON, TUE, WED, THU, FRI, SAT, SUN
3 }

```

无枚举类型前如何解决上述需求？

一般采用声明多个整型常量的做法实现枚举类的功能。

```

1 public class Week {
2     public static final int MON = 1;
3     public static final int TUE = 2;
4     ...
5 }

```

8.5.2 遍历枚举类型常量值

可以使用静态方法 `values()` 遍历枚举类型常量值。


Code: [ListEnum.java](#)

```

1 public class ListEnum {
2     public static void main(String[] args) {
3         Week[] days = Week.values();
4         for (Week d: days) {
5             System.out.println(d);
6         }
7     }
8 }

```

8.5.3 组合使用枚举类型与 switch

课程配套代码  package sample.advance.enumclass

注意

1. `case` 字句必须省略其枚举类型的前缀，即只需要写成 `case SUN:`，而不允许写成 `case Week.SUN:`，否则编译出错。
2. 不必担心系统无法搞清这些常量名称的出处，因为 `switch` 后的小括号中的表达式已经指明本次要区分处理的是 `Week` 类型常量。

实验设计
