Java学习

0 开始之前

- 本次学习开始于2021.4.30。2022年6月至7月，重新学习了Java相关语法，对本文大部分内容进行了重写。
- 2022年9月开始修读的 Java应用技术 课程的笔记在原文内容上进行增加和修改。
- 本文主要基于 布迪·克尼亚万的 Java经典入门指南。知乎电子书 链接

1 Java基础语法

1.0 关于Java

- **Java 语言是简单的**
    - Java 语言的语法与 C 和 C++ 接近。另一方面，Java 丢弃了 C++ 中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。特别地，Java 语言不使用指针，而是引用。并提供了自动分配和回收内存空间，使得程序员不必为内存管理而担忧。
- **Java 语言是可移植的**
    - 这种可移植性来源于体系结构中立性，另外，Java 还严格规定了各个基本数据类型的长度。Java 系统本身也具有很强的可移植性，Java 编译器是用 Java 实现的，Java 的运行环境是用 ANSIC 实现的。
- **Java 语言是体系结构中立的**
    - Java 程序（后缀为 java 的文件）在 Java 平台上被编译为体系结构中立的字节码格式（后缀为 class 的文件），然后可以在实现这个 Java 平台的任何系统中运行。这种途径适合于异构的网络环境和软件的分发。
- **Java 语言是解释型的**
    - Java 程序在 Java 平台上被编译为字节码格式，在运行时，Java 平台中的 Java 解释器对这些字节码进行解释执行，执行过程中需要的类在联接阶段被载入到运行环境中。
- **Java 是高性能的**
    - 与那些解释型的高级脚本语言相比，Java 的确是高性能的。事实上，Java 的运行速度随着 JIT(Just-In-Time，即时编译编译器）编译器技术的发展越来越接近于 C++。
- **Java 语言是多线程的**
    - 在 Java 语言中，线程是一种特殊的对象，它必须由 Thread 类或其子（孙）类来创建。通常有两种方法来创建线程：其一，使用型构为 Thread(Runnable) 的构造子类将一个实现了 Runnable 接口的对象包装成一个线程，其二，从 Thread 类派生出子类并重写 run 方法，使用该子类创建的对象即为线程。值得注意的是 Thread 类已经实现了 Runnable 接口，因此，任何一个线程均有它的 run 方法，而 run 方法中包含了线程所要运行的代码。线程的活动由一组方法来控制。Java 语言支持多个线程的同时执行，并提供多线程之间的同步机制（关键字为 synchronized）。

- **Java 语言是动态的**
    - Java 语言的设计目标之一是适应于动态变化的环境。Java 程序需要的类能够动态地被载入到运行环境，也可以通过网络来载入所需要的类。这也有利于软件的升级。另外，Java 中的类有一个运行时刻的表示，能进行运行时刻的类型检查。
    - 动态性与编程语言是编译型还是解释型有关。
- Java API：链接

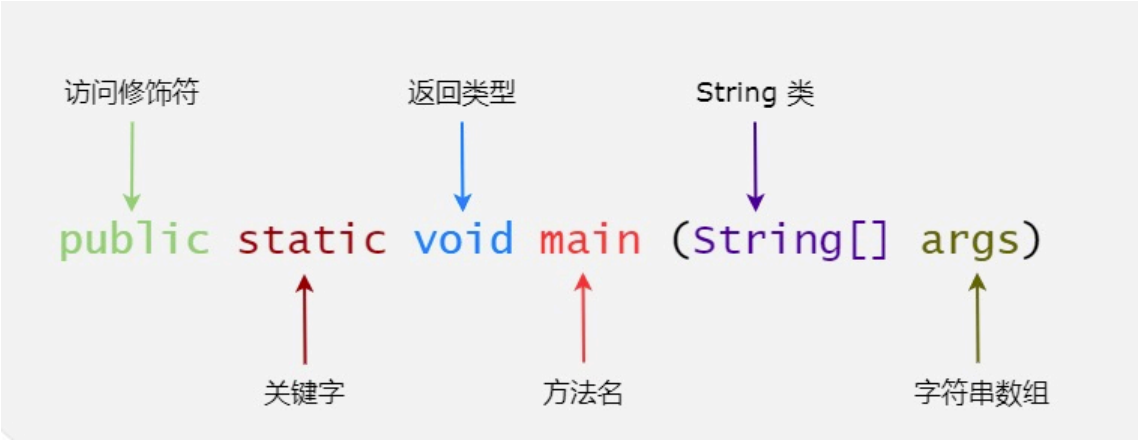
1.1 第一个Java程序

- Java 语言是面向对象的。Java 的代码的基本单位是一个个的 **类 (class)**，每个文件中只能有1个公共顶级类，但可以有多个非公共顶级类。
- Java 要求每个代码文件声明其所属的 **包 (package)** 。包的结构类似于文件夹的结构。在IDE 中新建类文件时，IDE 一般会帮助我们写好所属的包。
- 在 IDE 中编写 Java 程序的步骤是：新建 project–> 在源码（src）文件夹中新建类（class）–> 在创建出的类文件中编写代码。附属产生的 .iml 文件是配置文件。

以下是一个Helloworld程序：

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(”Hello World”); // 输出 Hello World
    }
}
```

可按下图进行简记和理解：



- 该代码中， `public class HelloWorld{ ... }` 定义了一个名为 HelloWorld 的 **类**。
    - Java 是 **大小写敏感** 的。
    - 按照习惯， **类名要用大写字母开头** 。
    - **public** 是这个类的 **修饰符** 说明这个类是公开的。
    - 花括号中的代码是这个类的定义。
- `public static void main(String[] args){ ... }` 定义了名为 **main** 的 **方法** 。
    - 按照习惯， **方法名用小写字母开头** 。
    - 方法是可执行的代码块，花括号中的内容就是这个方法的代码。

    - 括号中的内容是这个方法的 **参数** ，本方法中有一个类型为 **String[]** （即字符串数组），名称为 **args** 的参数。
    - **public static** 是这个方法的 **修饰符** ，说明这个方法是公开、静态的。
    - **void** 是方法的 **返回类型** 。
    - Java 规定，某个类定义的 `public static void main(String[] args)` 是Java程序的固定入口方法。因此，Java 程序总是从 `main` 方法开始执行。
    - 向main方法传递参数： `java className arg1 arg2 arg3...`

- 该代码被保存在 HelloWorld.java 文件中。 **Java 程序的源文件名与public类名必须相同** ，否则会导致编译错误。

命令行编译： `Javac HelloWorld.java` `Java HelloWorld`

1.2 标识符

类名、变量名、方法名都被称为 **标识符** 。

- 所有的标识符都应该以字母（A-Z 或者 a-z）,美元符（$）或者下划线（_）开始，之后可以是字母（A-Z 或者 a-z）,美元符（$）、下划线（_）或数字的任何字符组合
- 关键字不能用作标识符
- 标识符是大小写敏感的
- 合法标识符举例：age、$salary、_value、__1_value
- 非法标识符举例：123abc、-salary

1.3 数据类型

- Java语言中，所有的变量在使用前必须声明。声明变量的基本格式与C++语言相同。
- **整数类型**：byte，short，int，long
    - 一个整数的缺省类型为 int。int的长度被固定为32位，在不同的机器上长度一致，最大2147483647。
    - 如果表示一个整数为 long 类型，需要加后缀 L 或 l，如 31L。long 8 个字节。
    - 八进制数以 0 为前缀，如 027 等，十六进制数以 0x 或 0X 为前缀，如 0xAC, 0X1b 等。
    - byte类型占位1个字节，范围为-128~127。
- **浮点数类型**：float，double
    - 一个浮点数的缺省类型为 double，占8字节64位。表示一个浮点数为 float 型，需要加后缀 F 或 f，float 占32位4字节。
    - 浮点数可以用标准计数法（如 12.3）或科学技术法（如 2.5E4, 3.66e7 等）表示。科学计数法也称指数形式，E 或 e 前面的数称为尾数，后面的数称为阶码。
- **字符类型**：char， String
    - Java 中 char 类型是一个单一的 **16 位 Unicode 字符**。所以，**英文字符和中文字符都占用两个字节**。
    - 将 char 类型赋值给 int 类型即可显示一个字符的 Unicode 编码。如 `int n2 = '中';`
    - 用转义字符 **\u**+Unicode 编码（16 进制）来表示字符： `char c4 = '\u4e2d';`
    - 由多个字符组成的字符序列称为字符串，字符串用双引号引起来。
    - String为字符串常量。**注意其首字母为大写。**转义字符表：

表 2-5 转义字符

转义字符	含 义	转义字符	含 义
\0	空字符	\\	反斜杠字符
\b	退格符	\'	单引号
\n	换行符	\"	双引号
\r	回车符	\ddd	三位八进制数
\t	水平制表符	\xhh	二位十六进制数
\x	十六进制数		

- **布尔类型**：数据类型为 **boolean**，只有 `true` 和 `false` 两种取值。
- Java中变量的默认值可以简记为均初始为0。
- Java中，用 **final** 修饰符来修饰常量。如 `final double PI = 3.14;` 。常量在程序运行时是不能修改的。根据习惯，**常量名全部使用大写字母** 。
- 局部变量类型推断
    - `var` ，类似C++的auto关键字： `var numCars = 200; var code = 'a';`
    - 由Java10提供，尚有限制和不完善之处

1.4 运算

- **整数**
    - 整数运算会出现溢出，但不会报错。
- **浮点数**
    - 与整数运算不同， **浮点数运算时，除数为0不会报错。** 但是会返回几个特殊值：

```
double d1 = 0.0 / 0; // NaN
double d2 = 1.0 / 0; // Infinity
double d3 = -1.0 / 0; // -Infinity
```

- **类型转换**
    - 加宽转换一般可以是隐式的。
    - 缩窄转换必须是显式的。需要在括号中指定目标类型，例如从long类型到int类型的缩窄转换： `long a = 10; int b = (int)a;`
- 运算符
    - 一元： `+ - ++ -- ! ~` 递增递减（前缀后缀都支持） 逻辑取反 按位取反
    - 算术： `+ - * / %`
    - 关系： `== != < > <= >= instanceof`
    - 条件： `&& || ?!` 与C一致的三目运算符
    - 移位： `<< >> >>>` 无符号右移
    - 赋值： `=、+=、-=、*=、/=、%=、<<=、>>=、>>>=、&=、^= |=` 支持各种复合赋值运算符
    - 位运算： `& | ^`

1.5 输入输出

输入

JDK1.5增加了 Scanner 类，为数据输入带来了很大的方便。Scanner 类提供一系列方法，用于接收不同类型的数据。

表 2-8 Scanner 类的主要方法

方 法	功能描述
next()	读取一个字符串，读到空格，Tab 键或 Enter 键结束
nextLine()	读取一行（包括空格和 Tab 键），只有 Enter 键才结束
nextByte()	读取一字节
nextShort()	读取一个短整型
nextInt()	读取一个整型
nextLong()	读取一个长整型
nextFloat()	读取一个 float 型
nextDouble()	读取一个 double 型

```
import java.util.Scanner;
public class Code1_2 {
    public static void main(String[] args){
        Scanner scanner = new Scanner(System.in);
        int a = scanner.nextInt();
        System.out.println("answer:" + (a + 1));
    }
}
```

输入5，运行结果为：

```
answer:6
```

输出

- `System.out.println()` 输出括号中的内容并换行。
- `System.out.print()` 输出括号中的内容。输出多项内容时，用 `+` 分隔，如 `System.out.print("a=" + a + ", b=" + b);`
- `System.out.printf()` 格式化输出，与C/C++中的printf语法类似。占位符有：

表 2-6 格式控制符

格式控制符	说 明	格式控制符	说 明
%s	字符串类型	%f	浮点类型
%c	字符类型	%a	十六进制浮点类型
%b	布尔类型	%e	指数类型
%d	整数类型（十进制）	%g	通用浮点类型（f 和 e 类型中较短的）
%x	整数类型（十六进制）	%n	换行符
%o	整数类型（八进制）		

- 格式化输出：链接

1.6 语句

- `if else` 与C一致。
- `while` 与 `do while` 与C一致。 `break` 和 `continue` 关键字与C一致。
- `for` 与C一致。
    - 对于数组或集合支持范围 `for` 循环：

```
String[] names = {”John”, ”Mary”, ”Paul”};
for(String name: names) {
    System.out.println(name);
}
```

    - 范围 `for` 循环是传值的。不能直接修改遍历的容器的对象值，但是可以通过修改指针所指的对象来实现间接修改。

- `switch`

```
switch (expression) {
    case value_1:
        // ...
        break;
    case value_2:
        // ...
        break;
    default:
        // ...
}
```

1.7 数组

每次创建数组时，编译器都会在后台创建一个对象。**数组的所有元素都具有相同的类型（称为数组的元素类型）。数组不可调整大小**，没有元素的数组称为空数组。

数组的声明方法是： `type[] arrayName` 或 `type arrayName[]` 。**声明**数组并不创建数组或为其元素分配空间，编译器只是声明了一个对象引用。**创建**数组的一种方法是使用 `new` 关键字，创建数组时，还必须指定要创建的数组大小。例如： `int[] myArray = new int[105]` 。

数组创建完成后，它的元素值要么为 `null` （如果元素类型是引用类型），要么为元素类型的默认值（如果数组元素是基本类型），例如，int数组元素的默认值是 0。

- 不用 `new` 也可以创建和初始化数组： `String[] names = {”ABC”, ”XYZ”, ”zoo”};`
- 多维数组可以用类似的方法定义。如：
    - `int[][] a = new int[2][3];`
    - `int[][] b = { {1,2}, {2,3}, {3,4} };`

```
String[][] selections = new String[][] {
    {”One”},
    {”One”, ”Two”},
    {”One”, ”Two”, ”Three”}
}; // 数组每一行长度不同
```

- 数组所有元素初始化为默认值，整型都是0，浮点型是0.0，布尔型是false。

- `length` 字段用于获取数组中元素的数量。
    - 如果有数组 `int[][] a = new int[2][3]` ，那么 `a.length` 为 2， `a[1].length` 为 3。
- 使用一个负索引或大于等于数组大小的索引将抛出 `java.lang.ArrayIndexOutOfBounds Exception` 异常。

java.util.Arrays 类

`Arrays`类提供了操作数组的静态方法。

- `copyOf` 创建具有指定长度的新数组。新数组具有与原始数组相同的元素。如果新长度与原数组的长度不相同，则使用 null 或默认值填充新数组，或截断原数组。
    - `numbers = Arrays.copyOf(numbers, 4);`
- `sort` 对指定数组的元素排序
- `equals` 比较两个数组的内容是否相等
- `binarySearch` 查找

```
import java.util.Arrays;
public class BinarySearchDemo {
   public static void main(String[] args) {
      int[] primes = { 2,  3,  5,  7,  11,  13,  17,  19 };
      int index = Arrays.binarySearch(primes, 13);
      System.out.println(index); // 输出 5
      index = Arrays.binarySearch(primes, 4);
      System.out.println(index); // 输出 -3
   }
}
```

注意Java中的数组是对象，这意味着用`final`修饰的数组的值是可以被修改的。

```
final int[] m = new int[1];
m[0] = max(m[0], value); // 值可以发生变化
```

2 Java面向对象基础

Java是一种面向对象的编程语言。面向对象编程，英文是Object-Oriented Programming，简称OOP。

面向对象程序设计中的概念主要包括：对象、类、数据抽象、继承、动态绑定、数据封装、多态性、消息传递。通过这些概念面向对象的思想得到了具体的体现。

2.1 类

类（Class）是一个共享相同结构和行为的对象的集合。类定义了一件事物的抽象特点。通常来说，类定义了事物的属性和行为。类可以为程序提供模版和结构。一个类的方法和属性被称为“成员”。“人”是一种抽象概念，而具体的人则是人这个概念的 **实例（Instance）** 。类是一种模板，它定义了如何创建实例。**public 类的定义必须保存在与类名相同的文件中，一个 Java 源文件只能包含一个公共类，但是它可以包含多个非公共类。**

main方法

main方法是一个特殊方法，它提供了应用程序的入口点。一个应用程序通常有很多类，其中只有一个类需要main方法，此方法允许调用包含的类。 `public static void main(String[] args)`

向main方法传递参数：`java className arg1 arg2 arg3...`

构造方法

每个类必须至少有一个构造方法（constructor），如果类没有明确定义构造方法，编译器将添加一个构造方法，用于创建对象。构造方法没有返回值，甚至没有void。构造方法必须与类名相同。与C++不同的是，构造方法需要添加访问标识符。

对于一个类，先做初始化再做构造函数。

```
public class Employee {
    public int age;
    public double salary;
    public Employee() {
    }
    public Employee(int ageValue, double salaryValue) {
        age = ageValue;
        salary = salaryValue;
    }
}
```

可变参数方法

可变参数方法（Varargs）是 Java 的一个特性，它允许方法带有可变长度的参数列表。 省略号 `...` 表示指定类型的参数可有 0 个或多个。如果参数列表同时包含固定参数（参数必须存在）和可变参数，则可变参数必须放在最后。多个参数值传入后当作数组存储。

```
public double average(int... args)
double avg = average(10, 100, 1000);

public static void element(Integer... args) {
    for (int arg : args) {
        System.out.print(arg + ” ”);
    }
    System.out.println();
}
```

方法重载

类似C++的函数重载，Java类中允许多个方法具有相同的名称，只要每个方法接收不同的参数类型集即可。方法的返回值不需要考虑在内。

2.2 对象

对象也称为实例（instance）。「构造」一词通常用来代替「创建」，因此也称为「构造一个Employee对象」。另一个常用术语是实例化（instantiate），实例化Employee类与创建Employee实例的含义相同。

创建对象

Java中，定义了class，只是定义了对象模版，而要根据对象模版创建出真正的对象实例，**必须用new操作符**。`new` 操作符可以创建一个实例，然后，我们需要定义一个引用类型的变量来指向这个实例：

`Person girlfriend = new Person();`

此时，我们定义了一个 `Person` 类型的实例，并声明了一个与对象类型相同的对象引用，来操作它。我们可以通过 `girlfriend.age` 来访问其成员变量，通过 `girlfriend.sleep()` 来访问其成员方法。

null关键字

引用变量没有值时被称为拥有 `null` 值。 如果在一个方法中声明了一个局部引用变量，但是没有给它分配对象，那么为了满足编译器的要求，需要给它赋一个 `null` 值。 如： `Book book = null;` 由于类级引用变量在创建实例时初始化，因此不需要为它们赋 `null` 值。

不能访问 `null` 变量引用的字段或方法。

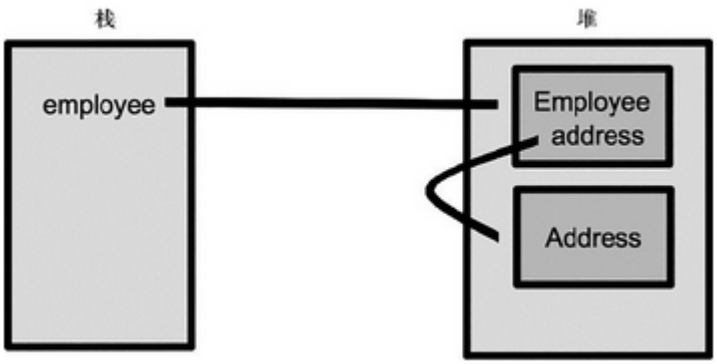
对象的内存分配

为程序分配的内存空间。在逻辑上分为两个部分：栈（stack）和堆（heap）。基本类型值在栈中分配，Java 对象则驻留在堆中。 当声明一个基本类型或引用变量时，如： `Book book;` ，**系统会在栈中留出一些字节，但其中包含的不是对象的数据，而是对象在堆中的地址**，初值为 `null` 。

当创建实例时， `Book book = new Book();` 这个实例存储在堆中，并且实例的地址被赋给引用变量book。一个对象可以被多个变量引用。

对于一个对象包含另一个对象的情况，创建一个Employee对象时，也同时创建了一个Address对象：

```
public class Employee {
    Address address = new Address();
}
Employee employee = new Employee();
```



- **Java的对象是由jvm自动回收的**，对开发者来说是不可控的。

this关键字

Java支持与C++类似的 `this` 关键字。

除此之外，Java的 `this` 还有一种特殊的方法，可以通过 `this` 作为方法名来调用其他的构造函数：

```
    Employee () {
    this(5, 6)
}
    Employee (int x, int y) {
    this.x = x;
    this.y = y;
}
```

创建对象的其他方法

> 我们已展示了用new关键字创建对象的方法。下面再介绍其他几种创建对象的方法。

> - 静态工厂方法。使用静态工厂方法实例化的类通常具有私有构造方法，这意味着不能使用new调用构造方法。相反，它提供了一个静态方法，该方法可以实现在它内部调用私有构造方法。由于静态方法是在同一个类中定义的，因此它可以访问私有构造方法。又因为它是一个静态方法，所以还可以在没有实例的情况下调用它。使用静态工厂方法的主要目标是提供一个特定实例，这个案例依赖于传递给方法的参数。一个典型的例子是java.time.LocalDate类，它表示日期，它的构造方法是私有的，但它提供了几个静态工厂方法来创建实例。LocalDate类的相关内容参见第 13 章。
> - 构建器（builder）类。使用构建器实例化的类的构造方法也是私有的。它提供了一个内部类来访问它的私有构造方法。使用构建器的原因是，有时需要一种简单的方法来定制一个带有多个参数的对象。在这里使用构造器要求构造方法接收一长串参数，这将使构造方法的可读性降低。构建器可以提供更整洁的方法来实例化类和定制结果对象，例如，HttpClient和HttpRequest类带有一个构建器，我们将在第 26 章中讨论它们。
> - 反射（reflection）。反射是 Java 中的一个强大的特性，它允许创建对象、调用方法和访问对象上的字段。之所以使用反射，主要是因为在编写代码时不知道要创建的对象类的名称。例如，在关系数据库中通常要求操作数据的 Java 代码与大多数数据库服务器一起工作，但是在编写代码时，不知道用户将使用哪种数据库。在这种情况下，反射就派上用场了。
> - 使用sun.misc.Unsafe类。这个类自 Sun Microsystem 推出 Java 以来一直存在。Unsafe类可用于底层编程，在不调用类的构造方法的情况下创建一个实例。由于Unsafe类现在还没有被支持，因此无法在 Java 文档中找到它。但是，任何 Java 编译器都可以编译使用它的代码。不要使用Unsafe类，除非读者确切知道要做什么。 上面这些都是先进的方法，本书没有对这些方法进行更深入的讨论，读者可以更进一步了解和探索。

2.3 包

Java 核心类定义在 `java.lang` 包中，完成输入和输出操作的类是 `java.io` 包的成员，等等。如果需要对包进行更详细的组织，可以创建与名称前面的部分相同的包，例如，Java 类库就有 `java.lang.annotation` 和 `java.lang.reflect` 包。但是，请注意，名称的共享并不意味着两个包相关， `java.lang` 包和 `java.lang.reflect` 是不同的。

以java开头的包名保留给核心类库，以javax开头的包是用来留给与核心类库配套的扩展类库用的，因此不能创建以java和javax开头的包。

除了对类进行组织，包还可以避免命名冲突，例如，假设存在两个同名类属于不同的包，应用程序不仅可以使用来自 A 公司的 `MathUtil` 类，还可以使用来自另一家公司的同名 `MathUtil` 类。为此，按照惯例，包名应该采用反转的域名。因此， `Sun` 的包名以 `com.sun` 开头。如果域名是 `brainysoftware.com` ，那么可以用 `com.brainysoftware` 作为包名开头。例如，可以把所有 `Applet` 程序放在 `com.brainysoftware.applet` 包中，把 `Servlet` 放在 `com.brainysoftware.servlet` 包中。

同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。

- 使用 `package` 关键字将类组织到某个包中。那么它的路径应该是 `net/java/util/Something.java` 这样保存的。

```
package net.java.util;
public class Something{
   ...
}
```

- 编译非默认包中的类，用 `/` 替换 `.`
    - 例如，要编译 `com.brainysoftware.common.MathUtil` 类，请将目录更改为工作目录（com的父目录）并输入 `javac com/brainysoftware/common/MathUtil.java` 。
    - 随后将在 `com/brainysoftware/common` 目录中创建 `MathUtil.class` 文件。 运行属于包的类 `java com/brainysoftware/common/MathUtil` 。

2.4 访问控制

适用于**类或类的成员**的访问控制符共4种： `public` 、 `protected` 、默认访问级别和 `private` 。默认访问级别有时也称为**包私有的**（package private）。当类声明前面没有访问控制修饰符时，该类具有默认的访问级别。具有默认访问级别的类只能被同一个包的其他类使用。类和接口不能声明为 **private**。

如果一个类的成员被 `protected` 访问控制符修饰，**那么这个成员既能被同一包下的其他类访问，也能被不同包下该类的子类访问。**

（update：对于**内部类**，可以被声明为private的，这样它仅能在它外部的类中被使用。）

```
class Chapter {
    ...
}
```

表 4-1　类成员的访问级别

访问级别	从其他包中的类	从同一个包中的类	从子类	从同一个类
public	是	是	是	是
protected	否	是	是	是
默认	否	是	否	是
private	否	否	否	是

- 如果一个Java源文件中定义的所有类都没有使用 `public` 修饰，那么这个Java源文件的文件名可以是一切合法的文件名;如果一个源文件中定义了一个 `public` 修饰的类，那么这个源文件的文件名必须与 `public` 修饰的类的类名相同。

> 构造方法也可以有 `public` 、 `protected` 、默认和 `private` 这 4 种访问级别。读者可能认为所有构造方法都必须是 `public` 的，因为拥有构造方法的目的是使类可实例化。然而，令人惊讶的是，事实并非如此。有些构造方法是 `private` 的，因此不能从其他类实例化它们的类。private构造方法通常在单例（ `singleton` ）类中使用。

2.5 导入

要使用其他包中的类，必须先导入包或导入要使用的类。Java 提供了 `import` 关键字，例如，要在代码中使用 `java.util.ArrayList` 类，必须使用下面的 `import` 语句： `import java.util.ArrayList;` `import` 语句必须放在 `package` 语句之后，类声明之前。

```
package app04;
import java.util.ArrayList;
```

```
public class Demo {
    ...
}
```

- 可以使用 `*` 来导入一个包中的所有成员。 `import java.util.*;`
- 在导入一个类以后，可以使用如上的 `ArrayList` 作为类名。
- 可以使用完全限定名来使用其他包的类，同时不需要导入它们。 `java.io.File file = new java.io.File(filename);`
    - 如果从不同的包导入了同名的类，在声明类时必须使用完全限定名。例如，Java 类库包含 `java.sql.Date` 类和 `java.util.Date` 类，同时导入这两个类，必须使用 `java.sql.Date` 和 `java.util.Date` 的完全限定名。
- `java.lang` 包中的成员被自动导入，因此，要使用 `java.lang.String` 类，并不需要显式导入。

2.6 静态成员

Java支持「静态成员」（static member），即不需要先实例化类即可调用的类成员。可以在没有实例的情况下调用它们。 `public static int a;`

```
package app04;
public class MathUtil {
    public static int add(int a, int b) {
        return a + b;
    }
}
MathUtil.add(a, b)
```

与C++类似的是，**静态方法只能访问其他静态方法或字段，而不能访问非静态字段**。由于 `main` 方法也是静态的，如下的代码无法通过编译：

```
package app04;
public class StaticDemo {
    public int b = 8;
    public static void main(String[] args) {
        System.out.println(b);
    }
}
```

这个问题有两个解决方案：将b定义为静态字段；或创建类的一个实例，然后使用实例访问b。

- **静态引用变量** `static Book book = new Book();`
    - 静态引用变量，这个变量将包含一个地址，但是引用的对象存储在内存堆中
    - 静态引用变量提供了一种很好的方法来公开需要在其他不同对象之间共享的对象。
- **静态常量** `static final int JANUARY = 1;`
    - 有时需要将所有静态 final 变量放在一个类中，这个类通常没有方法或其他字段，也从不实例化。

```
package app04;
public class Months {
    public static final int JANUARY = 1;
    public static final int FEBRUARY = 2;
    public static final int MARCH = 3;
    ...
}
int thisMonth = Months.JANUARY;
```

- **静态导入**
    - 可以使用 `import static` 关键字导入类的静态成员，例如，可以这样做： `import static java.util.Calendar.SATURDAY;`
- **静态成员初始化器**
    - 在 JVM 加载到内存后，它将开始执行DemoTest类的main方法。接下来，JVM 将按以下顺序做三件事情：加载、链接和初始化。

```
static{
        System.out.println(”Loading Table”);
}
```

2.7 静态工厂方法

前面学习了用 `new` 创建对象的方法，但是，Java 类库中的一些类不能以这种方式实例化，例如，不能使用 `new` 创建 `java.time.LocalDate` 类的实例，因为它的**构造方法是私有的**，相反，需要使用它的一个静态方法，例如用 `now` 创建该类的实例： `LocalDate today = LocalDate.now();`

```
package app04;
import java.time.LocalDate;
public class Discount {
    private int value;
    private Discount(int value) {
        this.value = value;
    }
    public int getValue() {
        return this.value;
    }
    public static Discount createSmallCustomerDiscount() {
        return new Discount(10);
    }
    public static Discount createBigCustomerDiscount() {
        return new Discount(12);
    }
}
```

- 当构造方法是公开的时，可以允许通过静态工厂方法和构造方法两种方式创建实例。

2.8 参数传递

> wjn：Java并非没有指针，Java实际上所有的对象都是指向对象的指针，而不是那个对象

用几个例子来理解：

- `int` 作为参数，因为 `int` 不是类，是基础类型，所以传入的相当于cpp里的 `int` ，值不变。

```
public class ReferencePassingTest {
    public static void increment(int x) {
        x++;
    }
    public static void reset(Point point) {
        point.x = 0;
        point.y = 0;
```

```
    }
    public static void main(String[] args) {
        int a = 9;
        increment(a);
        System.out.println(a);      // 输出 9
        Point p = new Point();
        p.x = 400;
        p.y = 600;
        reset(p);
        System.out.println(p.x);    // 输出 0
    }
}
```

- 将上下两个例子放在一起进行比较。当类作为参数时，可以理解为将指向一个类的实例的指针复制了一份作为传递的形式参数。
- `Integer` 是类，作为参数时传的相当于cpp的 `Integer*` 。 `swap` 函数里更改了形参的指针指向，对外面没有影响。
- 而 `Point` 虽然传的是指针，但改的不是指针本身存储的地址，而是指针指向的对象的内部的值。所以可以改掉。

```
public class HelloWorld {
    public static void swap(Integer a, Integer b) {
          Integer temp = a;
          a = b;
          b = temp;
    }
    public static void main(String[] args) {
        Integer x = 1;
        Integer y = -1;
        swap(x, y);
        System.out.println(x);      // 输出 1
    }
}
```

2.9 初始化

> 在初始化指定的类之前，我们必须先初始化其父类。如果父类还没有加载和链接，JVM 将首先加载和链接父类。同样，当父类即将初始化时，父类的父类也要先初始化。这个过程递归进行，直到初始化的类是层次结构中最顶层的类为止。

Java在类加载发生时就会进行初始化。不过，也可以编写代码，指定在每次创建类的实例时执行初始化。

静态初始化

在对静态变量进行初始化（用赋给变量的值或默认值）后，接下来执行 `static` 块中的代码。例如，清单 4.14 显示了带有静态代码的 `StaticCodeTest` 类，该类在加载时执行静态代码。与静态方法一样，在静态代码块中只能访问静态成员。

```
package app04;
public class StaticCodeTest {
    public static int a = 5;
    public static int b = a * 2;
    static {
        System.out.println(”static”);
        System.out.println(b);
    }
    public static void main(String[] args) {
        System.out.println(”main method”);
```

```
    }
}
```

实例初始化

实例初始化是包含在大括号中的代码：将对未被赋值的实例变量赋默认值。

```
{
    ...
}
```

实例初始化与静态初始化不同。后者发生在类加载时，它与实例化无关；相反，实例初始化发生在创建对象时。此外，与静态初始化器不同，实例初始化器可以访问实例变量。

Java变量的初始化顺序：静态变量或静态语句块------>实例变量或初始化语句块------>构造方法（注：这里实例变量的初始化是在构造方法中第一条指令执行前执行的，注意并不是构造方法之前）

可以参考：Java构造方法和变量初始化的执行顺序

2.10 继承

类继承的关键字是 `extend` 。如下定义了一个扩展Parent类的名为Child的子类。Java不允许多重继承，但多重继承的概念可以通过Java接口来实现。

```
public class Parent {
    ...
}
public class Child extends Parent {
    ...
}
```

> 所有没有显式扩展父类的 Java 类都将自动扩展java.lang.Object类。Object是 Java 中的最终超类。默认情况下，Parent类是Object类的子类。

与C++类似，Java允许将子类的实例赋给父类型的引用变量。如 `Animal animal = new Bird();` 。

在子类中，我们可以访问它的超类的 `public` 和 `protected` 方法和字段，但不能访问超类的 `private` 方法。如果子类和超类在同一个包中，还可以访问超类的默认方法和字段。

方法覆盖

要修改父类行为，可以通过覆盖方法实现。要覆盖一个方法，只需在子类中编写新方法，而不需要修改父类的任何内容。可以覆盖超类的 `public` 和 `protected` 方法。如果子类和超类位于同一个包中，还可以覆盖有默认访问级别的方法。

将超类中定义的方法的可见性从 `protected` 提升到 `public` ，这是允许的，但不允许降低可见性。

```
package app07;
public class Box {
    public int length;
    public int width;
    public int height;
    public Box(int length, int width, int height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }
```

```
    @Override
    public String toString() {
        return ”I am a Box.”;
    }
    @Override
    public Object clone() {
        return new Box(1, 1, 1);
    }
}
```

`Box` 类扩展了 `java.lang.Object` 类，这是一种隐式扩展，因为没有使用 `extends` 关键字。 `Box` 类覆盖了 `public toString` 方法和 `protected clone` 方法。 `Box` 类的 `clone` 方法是 `public` ，而 `Object` 类中的 `clone` 方法是 `protected` ，提升了可见性。

> 被覆盖的方法通常用@Override注解标注，这不是必需的，但这样做是很好的实践。

不能覆盖 `final` 方法。要使方法成为 `final` ，请在方法声明中使用 `final` 关键字。如 `public final java.lang.String toUpperCase(java.lang.String)`

构造方法

与C++类似，调用子类的构造方法前会递归调用父类的构造方法，直到到达 `java.lang.Object` 的构造方法。

```
package app07;
class Base {
    public Base() {
        System.out.println(”Base”);
    }
    public Base(String s) {
        System.out.println(”Base.” + s);
   }
}
public class Sub extends Base {
    public Sub(String s) {
        System.out.println(s);
    }
    public static void main(String[] args) {
         Sub sub = new Sub(”Start”);
   }
}
```

输出为：Base /n Start，表明父类的构造方法先调用。

Java编译器会悄悄地将Sub的构造方法更改为如下结构（这种更改不会修改源文件），其中，**关键字super表示当前对象的直接超类的实例。**由于 `super` 是从Sub的实例中调用的，因此 `super` 表示Base的实例，即它的直接超类。

```
public Sub(String s){
    super();
    System.out.println(s);
}
```

可以用 `super` 关键字显式地从子类的构造方法那里调用父类的构造方法，但 `super` 必须是构造方法中的第一个语句。如果希望调用超类中的另一个构造方法，使用 `super` 关键字可以非常方便地实现，例如，可以将Sub中的构造方法修改为以下内容：

```
public Sub(String s) {
    super(s); // 调用了父类的含参构造方法
    System.out.println(s);
}
```

> 子类从自己的构造方法调用父类的构造方法实际上是有意义的，因为子类的一个实例必须始终配有其每个父类的实例。这样，可以将对子类中未被覆盖的方法的调用传递给其父类，直至找到层次结构中的第一个方法。

由于**关键字super表示当前对象的直接超类的实例，**还可以通过 `super` 关键字调用在子类中可见的父类的指定成员。

类型转换

与C++类似，可以把一个子类的对象赋给父类类型的引用变量，即将子类的一个实例转换为它的父类。这称为**向上转换**（upcast）。向上转换Child对象，只需将该对象赋给Parent类型的引用变量。

```
Child child = new Child();
Parent parent = child;
```

对于上面的例子，parent引用了Child类型的对象，所以可以将parent转换回Child，称这种转换为向下转换（downcast）。**向下强制转换要求在括号中写入子类型**，例如： `Child child2 = (Child) parent;` **只有当父类引用已经指向子类的实例时，才允许向下转换成子类。**

final关键字

将类声明为 `final` 的可以使类成为最终类，不可被扩展。如 `public final class Pencil;`

instanceof运算符

`instanceof` 运算符用于检验一个对象是否是某种指定的类型，它通常用在 `if` 语句中, 对 `null` 引用变量应用 `instanceof` 将返回 `false` 。

由于子类是其超类的一种类型，因此下面的`if`语句将返回`true`，其中`Child`是`Parent`的子类：

```
String s = ”Hello”;
if (s instanceof java.lang.String) //true
String s = null;
if (s instanceof java.lang.String) //false
Child child = new Child();
if (child instanceof Parent)            //true
```

2.11 多态

如果有一个类型为A的引用变量a，那么给它赋一个类型为B的对象是合法的，这称为向上转换（upcasting）。 `A a = new B();`

> 上面的代码中，把B的一个实例赋给a时，a的类型是A。这意味着，不能调用不是在A中定义的B中的方法。但是，如果输出 `a.getClass().getName()` 的值，将得到 B 而不是 A。为什么呢？这是由于在编译时a的类型为A，因此编译器不允许调用不是在A中定义的 B 的方法。此外，在运行时，a的类型是B，正如 `a.getClass().getName()` 的返回值（B）所示。
>
> 这就是多态性的本质。**如果B覆盖了A中的一个方法（例如名为`play`的方法），那么调用`a.play()`将导致调用B中的play实现（而不是A中的）。**多态性使对象（本例中指是a引用的对象）能够确定在调用方法时选择哪个方法（A中的方法还是B中的方法）。多态性指明运行时调用的对象实现。但是实际上，多态性的作用还远不止于此。

> 如果调用a的另一个方法（例如名为 `stop` 的方法），而该方法没有在B中实现，那么怎么办呢？JVM 很聪明，知道如何做。它会在B的继承层次结构查找。B必须是A的一个子类，或者说，如果A是一个接口，那么B必须是实现A的另一个类的子类。否则，代码就不能编译。解决了这个问题之后，JVM 将沿着类层次结构向上，找到 `stop` 的实现并运行它。
>
> 现在，多态性的定义有了更丰富的含义：多态性是一个 OOP 特性，它使对象在接收到一个方法调用时能够确定调用哪个方法的实现。

- 除了 static 方法和 final 方法，Java 中的方法绑定都发生在运行时，而不是编译时。运行时绑定也称为后期绑定或动态绑定。因此，Java 中的后期绑定机制使多态性成为可能。
- 多态性对 static 方法不起作用，因为它们是前期绑定的。例如，如果 `Employee` 和 `Manager` 类中的work方法都是 static 的，那么调用 `employee.work()` 将输出 `I am an employee` 。此外，由于 final 方法不能扩展，因此多态性也不能用于 final 方法。

3 核心类

3.1 java.lang.Object

`java.lang.Object` 类表示一个 Java 对象。事实上，所有类都是这个类的直接或间接后代。以下给出了 `java.lang.Object` 类中的方法。

- `clone` 创建并返回此对象的副本。
- `equals` 比较参数和此对象。类必须实现此方法，以提供比较实例内容的方法。
- `getClass` 返回对象的 `java.lang.Class` 对象
- `toString` 返回对象的描述

3.2 java.lang.String

一个 `String` 是一段文本，也可以将 `String` 看作 Unicode 字符的序列。 `String` 对象可以由任意数量的字符组成。字符数为 0 的字符串称为空字符串。 `String` 对象是常量，一旦创建，它们的值就不能更改。因此， `String` 实例被称为不可变的（immutable）。

可以使用 `new` 关键字构造 `String` 对象，但更常用的是将字面值赋给 `String` 引用变量： `String s = ”Java haha!”;` 。两种的区别是使用 `new` 一定会创建一个新的实例。

- **比较**
  - `==` 号比较的是两个引用常量指向的地址，也就是引用的是不是同一个实例。

```
String s1 = new String(”Java”);
String s2 = new String(”Java”);
if (s1 == s2) {
    ... // 将不会被执行
}
```

  - 多数情况下要比较内容相同，应该使用 `String` 类的 `equals` 方法：

```
String s1 = ”Java”;
if(s1.equals(”Java”)) // true
if(”Java”.equals(s1))
if(s1 != null && s1.equals(”Java”))
```

由于 `s1` 可能为 `null` ，应当采取第三或第四行的比较方法，更安全。

- **字面值**
  - `+` 连接字符串字面值
  - `””` 双引号中不能换行
  - 可以连接 `String` 和一个基本类型或对象，如 `String s1 = ”Java” + 8;`
  - 如果一个对象与一个 `String` 连接，那么将调用前者的 `toString` 方法并且结果将与 `String` 相连接。
- **构造方法**
  - `public String()`
  - `public String(String original)`
  - `public String(char[] value)`
  - `public String(byte[] bytes)`
  - `public String(byte[] bytes, String encoding)`
- **方法**
  - `public char charAt(int index)` 返回指定索引处的字符
  - `public String concat(String s)` 将指定的字符串连接到当前字符串的末尾并返回结果
  - `public boolean equals(String anotherString)` 比较内容
  - `public boolean endsWith(String suffix)` 测试当前字符串是否以指定的后缀结束
  - `public boolean startsWith(String preffix)`
  - `public char toCharArray()` 转换为字符数组
  - `public boolean isEmpty()` 为空
  - `public String[] split(String regEx)` 将字符串分割为与指定正则表达式匹配的字符串，例如， `”He is cool”.split(” ”)` 将返回一个由 3 个字符串组成的数组： `「He」` `「is」` `「cool」` 。
  - `public String toLoWer/UpperCase` 大小写

```
String s = ”ABC”;
s.toLowerCase();
println(s);  // ABC, String的内容是不可修改的
```

  - `public int length()` 字符数
  - `public int indexOf(String subString)` 返回指定字符串第一次出现的索引
  - `public String subString(int beginIndex, int endIndex)` 子字符串
  - 此外，还有静态方法：
    - `String.valueof(23)` 将基本数据类型、char 数组或Object实例转换为字符串表示形式
    - `format` 略

```
String firstName = ”John”;
String lastName = ”Adams”;
System.out.format(”First name: %s. Last name: %s”, firstName, lastName);
```

3.2 java.lang.StringBuffer 和 java.lang.StringBuilder

`String` 对象是不可变的，如果需要在其中添加或插入字符，就不适合使用 `String` 对象，因为 `String` 上的字符串操作总是创建一个新的 `String` 对象。对于添加和插入字符，最好使用 `java.lang.StringBuffer` 或 `java.lang.StringBuilder` 类。一旦完成了对字符串的操作，我们就可以将 `StringBuffer` 或 `StringBuilder` 对象转换成一个 `String` 。

在 JDK 1.4 之前， `StringBuffer` 类专门用于可变字符串。 `StringBuffer` 中的方法是同步的（synchronized），这使得 `StringBuffer` 适合在多线程环境中使用。然而，同步的代价却是性能的损失。JDK 5 增加了 `StringBuilder` 类，它是 `StringBuffer` 的非同步版本。如果不需要同步，应该选择 `StringBuilder` 而不是 `StringBuffer` 。以下以 `StringBuilder` 为例，也适用于 `StringBuffer` 。

- **构造方法** 有4个：
    - `public StringBuilder()`
    - `public StringBuilder(CharSequence seq)` 传递 `java.lang.CharSequence` 参数
    - `public StringBuilder(int capacity)`
    - `public StringBuilder(String string)`
- 当没有指定容量时，默认容量为16。超过时会自动进行容量扩展。
- **类的方法**
    - `public int capacity()` 返回容量
    - `public int length()` 返回字符串长
    - `public StringBuilder append(String string)` 将指定字符串追加到末尾。 有多种重载方法，允许传递基本类型、 `char` 数组和 `java.lang.Object` 实例。
        - `append` 返回的是对象本身，因此支持连续调用： `s.append(”a”).append(”b”);`
    - `public StringBuilder insert(int offset, String string)` 在指定的位置插入指定的字符串。有多种重载方法，允许传递基本类型和 `java.lang.Object` 实例。

3.3 基本类型包装类

java.lang.Integer

- `Int` 和 `Integer` 的区别主要体现在以下几个方面：
    - 数据类型不同： `int` 是基础数据类型，而 `Integer` 是包装数据类型；
    - 默认值不同： `int` 的默认值是 0，而 `Integer` 的默认值是 `null` ；
    - 内存中存储的方式不同： `int` 在内存中直接存储的是数据值，而 `Integer` 实际存储的是对象引用，当 `new` 一个 `Integer` 时实际上是生成一个指针指向此对象；
    - 实例化方式不同： `Integer` 必须实例化才可以使用，而 `int` 不需要；
    - 变量的比较方式不同： `int` 可以使用 `==` 来对比两个变量是否相等，而 `Integer` 一定要使用 `equals` 来比较两个变量是否相等。

> - 当数值超过[−128, 127]之后，两个数值相同的 `Integer` 对象用 `==` 判断的结果是 `false` 。
> - 因此注意一定要使用 `equals` 来比较两个变量是否相等。

> Integer 有 byteValue 、 doubleValue 、 floatValue 、 intValue 、 longValue 和 shortValue 这些不带参数的方法，它们分别将包装类的值转换为 byte 、 double 、 float 、 int 、 long 和 short 。此外，使用 toString 方法可以将值转换为字符串，还可以使用静态方法将字符串解析为 int （通过 parseInt 方法）。 public static int parseInt(String string)

java.lang.Boolean

两种构造方式： Boolean b1 = new Boolean(false); Boolean b2 = new Boolean(”true”);

3.4 java.util.Scanner

使用 Scanner 接收键盘输入：在创建 Scanner 实例时为其传递一个 System.in ；然后，要接收用户输入，调用实例的 next 方法， next 方法用于缓冲用户从键盘或其他设备输入的字符，直到用户按回车键；最后，它返回一个包含用户输入的字符的 String ，但不包括回车符。

```
package app05;
import java.util.Scanner;
public class ScannerDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.print(”What’s your name? ”);
            String input = scanner.nextLine();
            if (input.isEmpty()) {
                break;
            }
            System.out.println(”Your name is ” + input + ”. ”);
        }
        scanner.close();
        System.out.println(”Good bye”);
    }
}
```

3.5 java.lang.Class

Class 类是 java.lang 包的一个成员。JVM 每次创建一个对象时，会同时创建一个 java.lang.Class 对象来描述对象的类型。同一个类的所有实例共享同一个 Class 对象。可以调用对象的 getClass 方法（该方法继承自 java.lang.Object ）来获取 Class 对象。

```
String country = ”a”;
Class myclass = country.getClass();
System.out.println(myClass.getName()); //java.lang.String
```

有了 Class 类，还可以在不使用 new 关键字的情况下创建对象，可以调用 Class 类的 forName 和 newInstance 两个方法实现对象的创建：

```
package app05;
public class ClassDemo {
    public static void main(String[] args) {
        String country = ”Fiji”;
        Class myClass = country.getClass();
        System.out.println(myClass.getName());
        Class klass = null;
        try {
```

```
            klass = Class.forName("app05.Test");
        } catch (ClassNotFoundException e) {
        }
        if (klass != null) {
            try {
                Test test = (Test) klass.newInstance();
                test.print();
            } catch (IllegalAccessException e) {
            } catch (InstantiationException e) {
            }
        }
    }
}
```

5 接口和抽象类

> 初学者会认为接口类似于用 `interface` 关键字声明的 Java 类，它的方法没有主体。 虽然这样描述也正确，但是把接口当作一个没有实现的类来处理就忽略了全局。一种更好的定义是「接口是一种契约」。它是服务提供者（服务器）和此类服务的用户（客户）之间的一种契约。有时服务器定义契约，有时客户定义契约。

接口可以定义字段和方法。在 JDK 1.8 之前，接口中的所有方法都是抽象的，但从 JDK 1.8 开始，还可在接口中定义默认方法和静态方法。除非另有说明，否则接口方法就是指抽象方法。

与类一样，接口具有 `public` 或默认访问级别。接口可以定义字段和方法。接口的所有成员都是隐含 `public` 的。

```
public interface Printable {
    void print(Object o);
}
```

这里即使 `print` 方法声明没有 `public` 关键字，它也是 `public` 的。就像 Java 类一样，接口也是创建对象的模板。但是，与普通类不同，接口不能实例化，它只是定义了一组 Java 类可以实现的方法。

要实现接口，需要在类声明之后使用 `implements` 关键字。一个类可以实现多个接口。例如，给出的 `CanonDriver` 类**实现**了 `Printable` 接口。方法实现应该使用 `@Override` 注解标注。

```
public class CanonDriver implements Printable {
    @Override
    public void print(Object obj) {
        // 实现打印功能的代码
    }
}
```

除非另有说明，否则接口的所有方法都是抽象的。实现类必须覆盖接口中的所有抽象方法。接口及其实现类之间的关系就像父类和子类的关系一样。**实现类的实例也是接口的实例**，例如，下面的 `if` 语句的计算结果为 `true`：

```
CanonDriver driver = new CanonDriver();
if (driver instanceof Printable)
```

> `instanceof` 是 Java 的一个二元操作符，类似于 `==`， `>` 等操作符。
>
> `instanceof` 是 Java 的保留关键字。它的作用是测试它左边的对象是否是它右边的类的实例，返回 `boolean` 的数据类型。

- 接口中的字段必须初始化，并且隐含为 public、static 和 final 的。按照惯例，接口的字段名应该使用全大写。所以可以说接口是一种特殊的类，由全局常量和公共的抽象方法组成，下面的代码有相同的效果：

```
public int STATUS = 1;
int STATUS = 1;
public static final STATUS = 1;
```

- 在接口中声明抽象方法就像在类中声明方法一样。但是，接口中的抽象方法没有方法体，它们直接以分号结束。所有抽象方法都是隐式 `public` 和 `abstract` 的。

5.1 接口的继承和扩展

接口支持继承。一个接口可以扩展另一个接口。如果A接口扩展了B接口，就说A是B的子接口，B是A的超接口。由于A直接继承B，因此B是A的直接超接口。继承A的任何接口都是B的间接子接口。

> 扩展接口的目的是什么呢？就是为了在不破坏现有代码的情况下安全地向接口添加功能。这是因为一旦接口发布，就不能向其添加新方法了。

```
// 文件名: Sports.java
public interface Sports
{
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

// 文件名: Football.java
public interface Football extends Sports
{
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

// 文件名: Hockey.java
public interface Hockey extends Sports
{
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

并且，不同于类，**接口支持多重继承**。

5.2 接口中的实现

在 Java 8 之前，不允许在接口中有任何实现。不过，Java 已经得到了发展，现在可以在接口中添加默认方法、静态方法甚至私有方法，虽然它们不能是 final 方法。

> 默认方法是指接口的默认方法，它是java8的新特性之一。顾名思义，默认方法就是接口提供一个默认实现，且不强制实现类去覆写的方法。默认方法用default关键字来修饰。

默认方法的语法格式如下：

```
public interface Vehicle {
   default void print(){
      System.out.println(”我是一辆车!”);
   }
}
```

一个应用的实例：

```
public class Java8Tester {
   public static void main(String args[]){
      Vehicle vehicle = new Car();
      vehicle.print();
   }
}

interface Vehicle {
   default void print(){
      System.out.println("我是一辆车!");
   }

   static void blowHorn(){
      System.out.println("按喇叭!!!");
   }
}

interface FourWheeler {
   default void print(){
      System.out.println("我是一辆四轮车!");
   }
}

class Car implements Vehicle, FourWheeler {
   public void print(){
      Vehicle.super.print();
      FourWheeler.super.print();
      Vehicle.blowHorn();
      System.out.println("我是一辆汽车!");
   }
}
```

有些接口有多个抽象方法，实现类必须实现所有这些方法。如果只需要某些方法，那么这可能是一项单调乏味的任务。鉴于此，可以创建一个通用实现类，该类使用默认代码实现接口中的抽象方法；然后，实现类可以扩展通用类，并仅覆盖它希望修改的方法。这种通用类称为基类（base class），使用起来十分方便。

5.3 抽象类

抽象类具有与接口类似的功能，即在服务提供者与其客户之间提供一种契约，同时，**抽象类还可以提供部分实现。需要显式覆盖的方法可声明为抽象方法。**因为抽象类不能实例化，所以仍然需要创建实现类，但不必覆盖不用的或不修改的方法。

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。一个类不能同时被 `abstract` 和 `final` 修饰。**如果一个类包含抽象方法，那么该类一定要声明为抽象类。抽象类可以包含抽象方法和非抽象方法。**

```
public abstract class SuperClass{
    abstract void m(); //抽象方法
}

class SubClass extends SuperClass{
     //实现抽象方法
      void m(){
          .........
```

```
    }
}
```

抽象方法

抽象方法是一种没有任何实现的方法，该方法的具体实现由子类提供。抽象方法不能被声明成 final 和 static。 任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。

如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。抽象方法的声明以分号结尾，例如：**`public abstract sample();`**。

> 在 Java 8 之前的时代，在抽象类和接口之间做出选择比现在容易。在那时，如果需要在扩展/实现类之间共享一些实现，则选择抽象类；否则，就选择接口。今天，也可以将实现放在接口中，因此创建抽象类的动机肯定会减少。然而，仍然有一些理由使用抽象类。
>
> （1）在抽象类中可以添加最终（final）方法，但在接口中不能。如果想阻止一个方法被覆盖，这一点非常重要。
>
> （2）在抽象类中，可以使用字段保存对象的状态。相反，接口中的字段是 public static 的，这意味着它们的值在实现类的所有实例之间共享。
>
> （3）在抽象类中可以定义构造方法，但在接口中不能。