

目录

介绍	1
Java 快速入门	2
准备工作	2.1
安装 JDK	2.1.1
第一个 Java 程序	2.1.2
使用 IDE	2.1.3
Java 程序基础	2.2
Java 程序基本结构	2.2.1
变量和数据类型	2.2.2
整数运算	2.2.3
浮点数运算	2.2.4
布尔运算	2.2.5
字符和字符串	2.2.6
数组类型	2.2.7
控制流程	2.3
输入和输出	2.3.1
if 判断	2.3.2
switch 多重选择	2.3.3
while 和 do while 循环	2.3.4
for 循环	2.3.5
break 和 continue	2.3.6
数组操作	2.4
数组遍历	2.4.1
数组排序	2.4.2
多维数组	2.4.3
命令行参数	2.4.4
面向对象编程	3
方法	3.1
构造方法	3.2
方法重载	3.3
继承	3.4
多态	3.5
抽象类	3.6
接口	3.7
静态字段和静态方法	3.8

包	3.9
内部类	3.10
classpath 和 jar	3.11

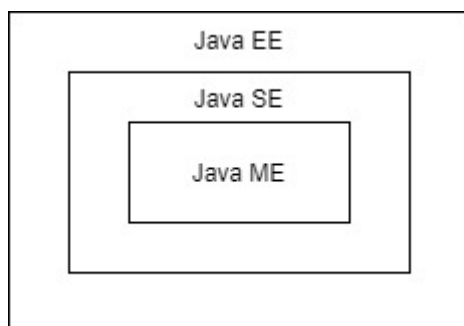
Java 简介

Java 是由 Sun Microsystems 公司于 1995 年 5 月推出的 Java 面向对象程序设计语言和 Java 平台的总称。由 James Gosling（高司令，人称 Java 之父）和同事们共同研发，并在 1995 年正式推出。后来 Sun 公司被 Oracle（甲骨文）公司收购，Java 也随之成为 Oracle 公司的产品。

Java 分为三个体系：

- Java SE(J2SE) (Java2 Platform Standard Edition, java平台标准版)
- Java EE(J2EE) (Java 2 Platform,Enterprise Edition, java平台企业版)
- Java ME(J2ME) (Java 2 Platform Micro Edition, java平台微型版)

Java 体系关系如下图：



简单来说，Java SE 就是标准版，包含标准的 JVM 和标准库，而 Java EE 是企业版，它只是在 Java SE 的基础上加上了大量的 API 和库，以便方便开发 Web 应用、数据库、消息服务等，Java EE 的应用使用的虚拟机和 Java SE 完全相同。











Java ME 就和 Java SE 不同，它是一个针对嵌入式设备的“瘦身版”，Java SE 的标准库无法在 Java ME 上使用，Java ME 的虚拟机也是“瘦身版”。

1. 首先要学习 Java SE，掌握 Java 语言本身、Java 核心开发技术以及 Java 标准库的使用；
2. 如果继续学习 Java EE，那么 Spring 框架、数据库开发、分布式架构就是需要学习的；
3. 如果要学习大数据开发，那么 Hadoop、Spark、Flink 这些大数据平台就是需要学习的，他们都基于 Java 或 Scala 开发；

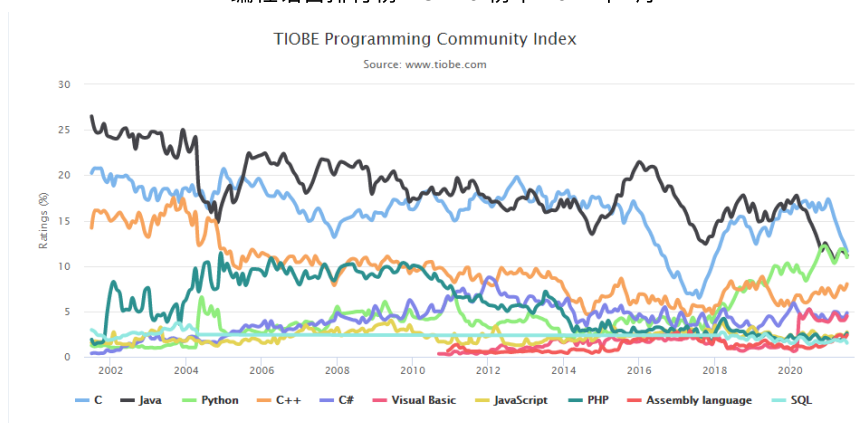
TIOBE 排行榜

TIOBE 排行榜是根据互联网上有经验的程序员、课程和第三方厂商的数量，并使用搜索引擎（如 Google、Bing、Yahoo!）以及 Wikipedia、Amazon、YouTube 和 Baidu（百度）统计出排名数据，可以反映某个编程语言的热门程度。

最新 TIOBE 编程语言排行榜显示，Java 在 2021 年 07 月榜单中排名第二，此前长年霸榜，和 C 语言一同占据编程语言的半壁江山。

Jul 2021	Jul 2020	Change	Programming Language	Ratings	Change
1	1		 C	11.62%	-4.83%
2	2		 Java	11.17%	-3.93%
3	3		 Python	10.95%	+1.86%
4	4		 C++	8.01%	+1.80%
5	5		 C#	4.83%	-0.42%
6	6		 Visual Basic	4.50%	-0.73%
7	7		 JavaScript	2.71%	+0.23%
8	9	▲	 PHP	2.58%	+0.68%
9	13	▲▲	 Assembly language	2.40%	+1.46%
10	11	▲	 SQL	1.53%	+0.13%

编程语言排行榜 TOP10 榜单 2021年7月



TOP10 编程语言走势图

JetBrains 开发者生态系统调查

JetBrains是一家捷克的软件开发公司，该公司位于捷克的布拉格，并在俄罗斯的圣彼得堡及美国麻州波士顿都设有办公室，该公司最为人所熟知的产品是Java编程语言开发撰写时所用的集成开发环境：IntelliJ IDEA。

根据《JetBrains 2020 开发者生态系统调查》显示，Java 任是最受欢迎的主要编程语言，并在多个领域都保持着强劲的趋势和热度。

重要发现

Java

是最受欢迎的主要编程语言。

JavaScript

是最常用的整体编程语言。

网站

是开发者开发的最常见应用程序类型。

Web (后端)

是最受欢迎的平台。

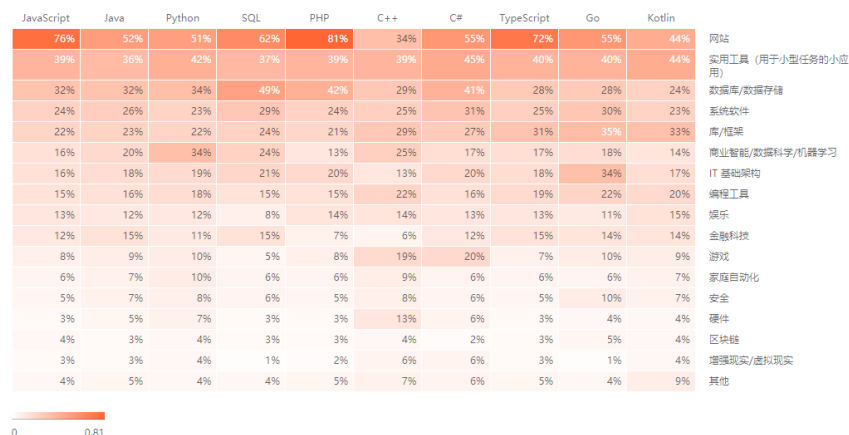
Go, Kotlin, Python

是开发者计划采用或迁移到的前 3 大语言。

Python

在过去 12 个月中使用的语言榜单中超过 Java。它是研究最多的语言。在过去 12 个月中，30% 的受访者开始或继续学习 Python，高于去年的百分比。

报告发现



开发软件类型报告

为什么 Java 应用最广泛？

从互联网到企业平台，Java 是应用最广泛的编程语言，原因在于：

- Java 是基于 JVM 虚拟机的跨平台语言，一次编写，到处运行；
- Java 程序易于编写，而且有内置垃圾收集，不必考虑内存管理；
- Java 虚拟机拥有工业级的稳定性和高度优化的性能，且经过了长时期的考验；
- Java 拥有最广泛的开源社区支持，各种高质量组件随时可用。

Java 语言常年霸占着三大市场：

- 互联网和企业应用，这是Java EE 的长期优势和市场地位，例如 Spring boot 构建企业级 Web 应用；
- 大数据平台，主要有 Hadoop、Spark、Flink 等，他们都是 Java 或 Scala（一种运行于 JVM 的编程语言）开发的；
- Android 移动平台，开发安卓应用软件。

安装 JDK

Java 程序必须运行在 JVM 之上，运行 Java 程序的第一件事情就是安装 JDK。

版本选择

关于版本的选择，现在最新版是 JDK 16，Oracle 公司于 2014 年发布的 JDK 8 引入了很多新特性，其中最主要的就是新增了 Lambda 表达式，允许把函数作为方法的参数（函数作为参数传递到方法中）。

同样运行于 JVM 之上的 Scala 语言在设计之初就集成了面向对象编程和函数式编程的各种特性，其中函数和类、对象一样，都是一等公民，函数可以独立存在，不需要依赖于类和对象。而在 Java 中，方法绝不可能脱离类和对象独立存在，即不具备面向过程编程的特性。

所以尽管距 JDK 8 发布已过多年，目前其任然具有庞大的用户基数，加上升级 JDK 版本导致的不兼容问题，很容易影响系统的稳定性，Java 开发的企业级应用一般比较复杂和庞大，重构和调试成本大，所以对于学习 Java 这门编程语言，推荐大家选择 JDK 8，官方对于这个版本也一直在维护升级。

下载安装

对于 Windows 操作系统，最方便的是直接 [Oracle 官网](#) 下载对应版本的安装包（.exe 文件），运行安装即可。也可下载编译后的压缩包（.tar.gz），手动配置环境变量，如下：

- JAVA_HOME = JDK 安装的根目录
- CLASSPATH = .;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;
- Path = %JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

安装或者配置完成后，在 cmd 窗口键入 `java -version`，若成功返回版本信息，则安装配置成功。

文件解读

我们在 JDK 安装目录下，可以找到以下执行文件：

- java：这个可执行程序其实就是 JVM，运行 Java 程序，就是启动 JVM，然后让 JVM 执行指定的编译后的代码；
- javac：这是 Java 的编译器，它用于把 Java 源码文件（以 .java 后缀结尾）编译为 Java 字节码文件（以 .class 后缀结尾）；
- jar：用于把一组 .class 文件打包成一个 .jar 文件，便于发布；
- javadoc：用于从 Java 源码中自动提取注释并生成文档；
- jdb：Java 调试器，用于开发阶段的运行调试。

第一个 Java 程序

Hello World

总所周知，学习一门编程语言，编写的第一个程序都是 **Hello World**，接下来我们打开一个文本编辑器，输入以下代码：

```
public class HelloWorld{  
    public static void main(String[] args){  
        System.out.println("Hello, world!")  
    }  
}
```

观察上述代码，我们使用一个 `main()` 方法。实际上，这个方法就是 Java 程序的入口方法，整个程序就从这个方法中的第一行代码开始运行。其写法固定，接收一个字符串数组的参数，返回为空。

我们在 `main()` 方法中写了一行代码 `System.out.println("Hello, world!")`，它用来打印一个字符串到屏幕上。

编译执行

保存文件为 `HelloWorld.java`，文件名必须与类名一致，文件后缀为 `.java`。

然后用 `javac` 命令把 `HelloWorld.java` 编译成字节码 `HelloWorld.class`：

```
javac HelloWorld.java
```

再用 `java` 命令执行该字节码文件：

```
java HelloWorld
```

graph TD
a(HelloWorld.java) -->|a1>源码| b(HelloWorld.class) -->|b1>字节码| c(JVM 之上运行)
a -- javac 编译 --> b
b -- java 执行 --> c
a1 --> b1

使用 IDE

IDE是集成开发环境：Integrated Development Environment的缩写。

使用IDE的好处在于，可以把编写代码、组织项目、编译、运行、调试等放到一个环境中运行，能极大地提高开发效率。

IDE提升开发效率主要靠以下几点：

- 编辑器的自动提示，可以大大提高敲代码的速度；
- 代码修改后可以自动重新编译，并直接运行；
- 可以方便地进行断点调试。

目前，Java 开发的主流 IDE 有：

Eclipse

[Eclipse](#) 是由 IBM 开发并捐赠给开源社区的一个 IDE，也是目前应用最广泛的 IDE。Eclipse 的特点是它本身是 Java 开发的，并且基于插件结构，即使是对 Java 开发的支持也是通过插件 JDT 实现的。

除了用于 Java 开发，Eclipse 配合插件也可以作为 C/C++ 开发环境、PHP 开发环境、Rust 开发环境等。

IntelliJ Idea

[IntelliJ Idea](#) 是由 JetBrains 公司开发的一个功能强大的 IDE，分为免费版和商用付费版。JetBrains 公司的 IDE 平台也是基于 IDE 平台+语言插件的模式，支持 Python 开发环境、Ruby 开发环境、PHP 开发环境等，这些开发环境也分为社区版（免费版）和付费版。

Visual Studio Code

[Visual Studio Code](#)（简称“VS Code”）是 Microsoft 在 2015年4月30 日 Build 开发者大会上正式宣布一个运行于 Mac OS X、Windows 和 Linux 之上的，针对于编写现代 Web 和云应用的跨平台源代码编辑器，在桌面上运行，并且可用于 Windows，macOS 和 Linux。它具有对 JavaScript，TypeScript 和 Node.js 的内置支持，并具有丰富的其他语言（例如 C++，C#，Java，Python，PHP，Go）和运行时（例如 .NET 和 Unity）扩展的生态系统。

最重要的是它是开源的，免费使用，插件丰富，有地表最强 IDE 美称！，其源码托管在 [Github](#) 上。

Java 程序基本结构

基本结构

我们来分析一个完整的 Java 程序，看看它的基本结构是什么：

```
/**
 * 可以用来自动创建 Java 文档的注释
 */
public class HelloWorld {
    public static void main(String[] args) {
        // 向屏幕输出文本：
        System.out.println("Hello, world!");
        /* 多行注释开始
        注释内容
        注释结束 */
    }
} // class定义结束
```

类名要求

Java 是面向对象的语言，一个程序的基本单位就是 class，class 是关键字，用于定义类，此处的 `HelloWorld` 是类名。

类名要求：

- 类名必须以英文字母开头，不能以数字和下划线开头，后接字母，数字和下划线的组合
- 习惯以大写字母开头

要注意遵守命名习惯，好的类命名：

- HelloWorld
- CityPOI

不好的类命名：

- helloworld
- Hello_World
- cityPOI

方法名要求

在 class 内部，可以定义若干方法，比如上述的 `main`，`main` 是方法名，`void` 是返回值类型，表示没有任何返回，一个方法必须要有一种返回类型，括号内的 `String[] args` 则是该方法接收的参数。`String[]` 是数据类型，为字符串数组。在方法内部，语句是真正的执行代码，Java 的每一行语句必须以分号结束。

方法名也有命名规则，命名和 class 一样，但是首字母小写：

- helloWorld

- cityPOI

不好的类命名：

- HelloWorld
- Hello_World
- CityPOI

注释

在 Java 程序中，注释是描述和解释程序的文本，供开发者阅读，不是程序的一部分，编译器会自动忽略注释。

Java 中有三种注释方式：

单行注释：

```
// 这是单行注释
```

多行注释：

```
/*  
这  
是  
多  
行  
注  
释  
*/
```

还有一种特殊的多行注释，以 `/**` 开头，以 `*/` 结束，如果有多行，每行通常以星号开头，用来自动创建文档：

```
/**  
 * 用来自动创建文档的注释  
 *  
 * @author liwei  
 */
```

变量和数据类型

变量

在 Java 中，变量分为两种：基本类型的变量和引用类型的变量，变量必须先定义后使用，在定义变量的时候，可以给它赋予初始值，变量可以重新赋值，但是不可以重复声明。

```
// 基本类型变量，更具体一点是整型变量，只声明，不赋值。  
int a;  
// 引用变量，更具体一点是字符串变量，声明同时初始化赋值。  
String s = "Hello, world!";
```

基本数据类型

基本数据类型是 CPU 可以直接进行运算的类型。Java 定义了以下几种基本数据类型：

- 整数类型：byte, short, int, long
- 浮点数类型：float, double
- 字符类型：char
- 布尔类型：boolean

计算机内存的最小存储单元是字节（byte），一个字节就是一个8位二进制数，即 8 个 bit。它的二进制表示范围从 00000000~11111111，换算成十进制是 0~255，换算成十六进制是 00~ff。

一个字节是 1byte，1024 字节是 1K，1024K 是 1M，1024M 是 1G，1024G 是 1T。进率为 1024，即 2 的 10 次方。

不同的数据类型占用的字节数不一样。以下是 Java 基本数据类型占用的字节数：

- byte: 1 个字节
- short: 2 个字节
- int: 4 个字节
- long: 8 个字节
- float: 4 个字节
- double: 8 个字节
- char: 2 个字节

浮点型

浮点类型的数就是小数，因为小数用科学计数法表示的时候，小数点是可以“浮动”的，如1234.5可以表示成 12.345×10^2 ，也可以表示成 1.2345×10^3 ，所以称为浮点数。Java 默认的浮点数类型是 double，如要声明为 float 则需要在浮点数后跟字母 f。

浮点数示例：

```
float f1 = 3.14f;
// 科学计数法表示的 3.14x10^38
float f2 = 3.14e38f;
double d = 1.79e308;
double d2 = -1.79e308;
// 科学计数法表示的 4.9x10^-324
double d3 = 4.9e-324;
```

布尔类型

Java 中 boolean 类型的[官方文档](#)的描述：

布尔数据类型只有两个可能的值：真和假。使用此数据类型为跟踪真/假条件的简单标记。这种数据类型就表示这一点信息，但是它的“大小”并不是精确定义

对虚拟机来说不存在 boolean 这个类型，boolean 类型数组的访问与修改共用 byte 类型数组的 baload 和 bastore 指令，单个 boolean 类型在编译后会使用 int 来表示。

总结：boolean 在数组情况下为1 个字节，单个 boolean 为4 个字节

布尔类型 boolean 只有 true 和 false 两个值，布尔类型总是关系运算的计算结果：

```
boolean b1 = true;
boolean b2 = false;
// 计算结果为 true
boolean isGreater = 5 > 3;
int age = 12;
// 计算结果为 false
boolean isAdult = age >= 18;
```

字符类型

字符类型 char 表示一个字符。Java 的 char 类型除了可表示标准的 ASCII 外，还可以表示一个 Unicode 字符：

```
char a = 'A';
char zh = '中';
```

引用类型

除了上述基本类型的变量，剩下的都是引用类型。例如，引用类型最常用的就是 String 字符串，引用类型的变量类似于 C 语言的指针，它内部存储一个“地址”，指向某个对象在内存的位置，不直接存储变量本身。

```
String s = "Hello, world!";
```

常量

定义变量的时候，如果加上 `final` 修饰符，表示这个变量在进行初始化赋值后就不再修改，这个变量就变成了常量：

```
final String name = "liwei";
```

注意：当变量被 `final` 修饰时，必须在声明时同时进行初始化赋值，且赋值后不可再修改。

变量的作用范围

在 Java 中，多行语句用 `{ }` 括起来。正确地嵌套使用 `{ }`，编译器就能识别出语句块的开始和结束。而在语句块中定义的变量，它有一个作用域，就是从定义处开始，到语句块结束。作用域之外无法引用这些变量：

```
{
    ...
    int i = 0; // 变量i从这里开始定义
    ...
    {
        ...
        int x = 1; // 变量x从这里开始定义
        ...
        {
            ...
            String s = "hello"; // 变量s从这里开始定义
            ...
        } // 变量s作用域到此结束
        ...
        // 注意，这是一个新的变量s，它和上面的变量同名，
        // 但是因为作用域不同，它们是两个不同的变量：
        String s = "world";
        ...
    } // 变量x和s作用域到此结束
    ...
} // 变量i作用域到此结束
```

小结

- Java提供了两种变量类型：基本类型和引用类型
- 基本类型包括整型，浮点型，布尔型，字符型。
- 变量可重新赋值，等号是赋值语句，不是数学意义的等号。
- 常量在初始化后不可重新赋值，使用常量便于理解程序意图。

整数运算

四则运算

Java 的整数运算遵循四则运算规则，可以使用任意嵌套的小括号。

```
public class Main {  
    public static void main(String[] args) {  
        // 3  
        int i = (1 + 2) * (99 - 98);  
        System.out.println(i);  
    }  
}
```

整数运算只能返回整数，所以两数相除，除不尽时返回的是结果的整数部分：

```
// 1  
int x = 15 / 10;
```

求余运算：

```
// 5  
int y = 15 % 10;
```

溢出

要特别注意，整数由于存在范围限制，如果计算结果超出了范围，就会产生溢出，且溢出不会出错。

```
int x = 2147483640;  
int y = 15;  
int sum = x + y;  
// -2147483641  
System.out.println(sum);
```

自增 / 自减

Java 还提供了 ++ 运算和 -- 运算，它们可以对一个整数进行加 1 和减 1 的操作：

```
int n = 0;  
// 1, 相当于 n = n + 1  
n++;  
// 0, 相当于 n = n - 1  
n--;
```

位移运算

在计算机中，整数总是以二进制的形式表示。例如，int类型的整数7使用4字节表示的二进制如下：

```
00000000 00000000 00000000 00000111
```

可以对整数进行移位运算。对整数7左移1位将得到整数14，左移两位将得到整数28，左移29位时，由于最高位变成1，因此结果变成了负数：

```
int n = 7;           // 00000000 00000000 00000000 00000111 = 7
int a = n << 1;      // 00000000 00000000 00000000 00001110 = 14
int b = n << 2;      // 00000000 00000000 00000000 00011100 = 28
int c = n << 28;     // 01110000 00000000 00000000 00000000 = 1879048192
int d = n << 29;     // 11100000 00000000 00000000 00000000 = -536870912\
```

还有一种无符号的右移运算，使用>>>，它的特点是不管符号位，右移后高位总是补0，因此，对一个负数进行>>>右移，它会变成正数，原因是最高位的1变成了0：

```
int n = -536870912;
int a = n >>> 1;    // 01110000 00000000 00000000 00000000 = 1879048192
int b = n >>> 2;    // 00111000 00000000 00000000 00000000 = 939524096
int c = n >>> 29;   // 00000000 00000000 00000000 00000111 = 7
int d = n >>> 31;   // 00000000 00000000 00000000 00000001 = 1
```

位运算

位运算是按位进行与、或、非和异或的运算。

与运算的规则是，必须两个数同时为1，结果才为1：

```
n = 0 & 0; // 0
n = 0 & 1; // 0
n = 1 & 0; // 0
n = 1 & 1; // 1
```

或运算的规则是，只要任意一个为1，结果就为1：

```
n = 0 | 0; // 0
n = 0 | 1; // 1
n = 1 | 0; // 1
n = 1 | 1; // 1
```

非运算的规则是，0和1互换：

```
n = ~0; // 1
n = ~1; // 0
```

异或运算的规则是，如果两个数不同，结果为1，否则为0：

```
n = 0 ^ 0; // 0
n = 0 ^ 1; // 1
n = 1 ^ 0; // 1
n = 1 ^ 1; // 0
```

对两个整数进行位运算，实际上就是按位对齐，然后依次对每一位进行运算。例如：

```
public class Main {
    public static void main(String[] args) {
        int i = 167776589; // 00001010 00000000 00010001 01001101
        int n = 167776512; // 00001010 00000000 00010001 00000000
        System.out.println(i & n); // 167776512
    }
}
```

运算优先级

- `()`
- `! ~ ++ --`
- `* / %`
- `+ -`
- `<< >> >>>`
- `&`
- `|`
- `+= -= *= /=`

记不住也没关系，只需要加括号就可以保证运算的优先级正确。

类型自动提升与强制转型

在运算过程中，如果参与运算的两个数类型不一致，那么计算结果为较大类型的整型。例如，short和int计算，结果总是int，原因是short首先自动被转型为int：

```
public static void typeConversion() {
    short s = 1234;
    int i = 123456;
    int x = s + i; // s自动转型为int
    short y = s + i; // 编译错误！
}
```

也可以将结果强制转型，即将大范围的整数转型为小范围的整数。强制转型使用(类型)，例如，将int强制转型为short：

```
int i = 12345;
short s = (short) i; // 12345
```

要注意，超出范围的强制转型会得到错误的结果，原因是转型时，int的两个高位字节直接被扔掉，仅保留了低位的两个字节。

小结

- 整数运算的结果永远是精确的；
- 运算结果会自动提升；
- 可以强制转型，但超出范围的强制转型会得到错误的结果；
- 应该选择合适范围的整型（int或long），没有必要为了节省内存而使用byte和short进行整数运算。

浮点数运算

浮点数运算和整数运算相比，只能进行加减乘除这些数值计算，不能做位运算和移位运算。

在计算机中，浮点数虽然表示的范围大，但是，浮点数有个非常重要的特点，就是浮点数常常无法精确表示。

浮点数0.1在计算机中就无法精确表示，因为十进制的0.1换算成二进制是一个无限循环小数，但是，0.5这个浮点数又可以精确地表示。

浮点数运算误差

因为浮点数常常无法精确表示，因此，浮点数运算会产生误差：

```
/**
 * 浮点数运算误差
 */
public static void floatDeviation() {
    double x = 1.0 / 10;
    double y = 1 - 9.0 / 10;
    // 观察x和y是否相等：
    System.out.println(x);
    System.out.println(y);
}
```

由于浮点数存在运算误差，所以比较两个浮点数是否相等常常会出现错误的结果。正确的比较方法是判断两个浮点数之差的绝对值是否小于一个很小的数：

```
// 比较x和y是否相等，先计算其差的绝对值：
double r = Math.abs(x - y);
// 再判断绝对值是否足够小：
if (r < 0.00001) {
    // 可以认为相等
} else {
    // 不相等
}
```

类型提升

如果参与运算的两个数其中一个为整型，那么整型可以自动提升到浮点型：

```
/**
 * 类型提升
 */
public static void typeUp() {
    int n = 5;
    double d = 1.2 + 24.0 / n; // 6.0
    System.out.println(d);
}
```

需要特别注意，在一个复杂的四则运算中，两个整数的运算不会出现自动提升的情况。例如：

```
double d = 1.2 + 24 / 5; // 5.2
```

计算结果为5.2，原因是编译器计算 $24 / 5$ 这个子表达式时，按两个整数进行运算，结果仍为整数4。

溢出

整数运算在除数为0时会报错，而浮点数运算在除数为0时，不会报错，但会返回几个特殊值：

- NaN表示Not a Number
- Infinity表示无穷大
- -Infinity表示负无穷大

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

强制转型

可以将浮点数强制转型为整数。在转型时，浮点数的小数部分会被丢掉。如果转型后超过了整型能表示的最大范围，将返回整型的最大值。例如：

```
int n1 = (int) 12.3; // 12
int n2 = (int) 12.7; // 12
int n3 = (int) -12.7; // -12
int n4 = (int) (12.7 + 0.5); // 13
int n5 = (int) 1.2e20; // 2147483647
```

小结

- 浮点数常常无法精确表示，并且浮点数的运算结果可能有误差；
- 比较两个浮点数通常比较它们的差的绝对值是否小于一个特定值；
- 整型和浮点型运算时，整型会自动提升为浮点型；
- 可以将浮点型强制转为整型，但超出范围后将始终返回整型的最大值。

布尔运算

对于布尔类型boolean，永远只有 `true` 和 `false` 两个值。

基本运算

布尔运算是一种关系运算，包括以下几类：

- 比较运算符： `>`，`>=`，`<`，`<=`，`==`，`!=`
- 与运算 `&&`
- 或运算 `||`
- 非运算 `!`

下面是一些示例：

```
boolean isGreater = 5 > 3; // true
int age = 12;
boolean isZero = age == 0; // false
boolean isNonZero = !isZero; // true
boolean isAdult = age >= 18; // false
boolean isTeenager = age > 6 && age < 18; // true
```

优先级

关系运算符的优先级从高到低依次是：

- `!`
- `>`，`>=`，`<`，`<=`
- `==`，`!=`
- `&&`
- `||`

短路运算

布尔运算的一个重要特点是短路运算。如果一个布尔运算的表达式能提前确定结果，则后续的计算不再执行，直接返回结果。

因为`false && x`的结果总是`false`，无论`x`是`true`还是`false`，因此，与运算在确定第一个值为`false`后，不再继续计算，而是直接返回`false`。

我们考察以下代码：

```

/**
 * 短路运算
 */
public static void shortCircuit() {
    boolean b = 5 < 3;
    boolean result = b && (5 / 0 > 0);
    System.out.println(result);
}

```

如果没有短路运算，&&后面的表达式会由于除数为0而报错，但实际上该语句并未报错，原因在于与运算是短路运算符，提前计算出了结果false。

如果变量b的值为true，则表达式变为true && (5 / 0 > 0)。因为无法进行短路运算，该表达式必定会由于除数为0而报错，可以自行测试。

类似的，对于||运算，只要能确定第一个值为true，后续计算也不再进行，而是直接返回true：

```
boolean result = true || (5 / 0 > 0); // true
```

三元运算符

Java还提供一个三元运算符b ? x : y，它根据第一个布尔表达式的结果，分别返回后续两个表达式之一的计算结果。示例：

```

/**
 * 三元运算符
 */
public static void ternaryOperator(){
    int n = -100;
    int x = n >= 0 ? n : -n;
    System.out.println(x);
}

```

上述语句的意思是，判断n >= 0是否成立，如果为true，则返回n，否则返回-n。这实际上是一个求绝对值的表达式。

注意到三元运算b ? x : y会首先计算b，如果b为true，则只计算x，否则，只计算y。此外，x和y的类型必须相同，因为返回值不是boolean，而是x和y之一。

小结

- 与运算和或运算是短路运算；
- 三元运算b ? x : y后面的类型必须相同，三元运算也是“短路运算”，只计算x或y。

字符和字符串

字符类型

字符类型char是基本数据类型，它是character的缩写。一个char保存一个Unicode字符。

因为Java在内存中总是使用Unicode表示字符，所以，一个英文字符和一个中文字符都用一个char类型表示，它们都占用两个字节。要显示一个字符的Unicode编码，只需将char类型直接赋值给int类型即可：

```
char c1 = 'A';
char c2 = '中';
int n1 = 'A'; // 字母“A”的Unicode编码是65
int n2 = '中'; // 汉字“中”的Unicode编码是20013
```

转义字符

所有的ASCII码都可以用“\”加数字（一般是8进制数字）来表示。而C中定义了一些字母前加“\”来表示常见的那些不能显示的ASCII字符，如\0,\t,\n等，就称为转义字符，因为后面的字符，都不是它本来的ASCII字符意思了。

ASCII 字符代码表 一																																			
高四位		ASCII 非打印控制字符																ASCII 打印字符																	
		0000				0001				0010				0011				0100				0101				0110				0111					
		0		1		2		3		4		5		6		7		0		1		2		3		4		5		6		7			
低四位		+进制	字符	ctrl	代码	字符	字解释	+进制	字符	ctrl	代码	字符	字解释	+进制	字符	ctrl	代码	字符	字解释	+进制	字符	ctrl	代码	字符	字解释	+进制	字符	ctrl	代码	字符	字解释	+进制	字符	ctrl	代码
0000	0	0		BLANK		0x00	NUL	空	16	▶	◀	0x01	DLE	数据链路转意	32	48	0	64	@	81	Q	97	`	112	p										
0001	1	1	☺	A	SOH	标题开始	17	◀	◀	0x02	DC1	设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q											
0010	2	2	☹	B	STX	正文开始	18	↕	↕	0x03	DC2	设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r											
0011	3	3	♥	C	ETX	正文结束	19	!!	!!	0x04	DC3	设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s											
0100	4	4	◆	D	EOT	传输结束	20	⏏	⏏	0x05	DC4	设备控制 4	36	%	52	4	68	D	84	T	100	d	116	t											
0101	5	5	♣	E	ENQ	查询	21	🔍	🔍	0x06	NAK	反确认	37	%	53	5	69	E	85	U	101	e	117	u											
0110	6	6	♠	F	ACK	确认	22	■	■	0x07	SYN	同步空闲	38	&	54	6	70	F	86	V	102	f	118	v											
0111	7	7	●	G	BEL	震铃	23	⬆	⬆	0x08	ETB	传输块结束	39	'	55	7	71	G	87	w	103	g	119	w											
1000	8	8	◼	H	BS	退格	24	↑	↑	0x09	CAH	取消	40	(56	8	72	H	88	X	104	h	120	x											
1001	9	9	○	I	TAB	水平制表符	25	↓	↓	0x0A	EM	媒体结束	41)	57	9	73	I	89	Y	105	i	121	y											
1010	A	10	◌	J	LF	换行/执行	26	→	→	0x0B	SUB	替換	42	*	58	10	74	J	90	Z	106	j	122	z											
1011	B	11	☎	K	VT	垂直制表符	27	←	←	0x0C	ESC	转义	43	+	59	11	75	K	91	[107	k	123	{											
1100	C	12	♀	L	FF	换页/翻页	28	↲	↲	0x0D	FS	文件分隔符	44	,	60	12	76	L	92	\	108	l	124												
1101	D	13	🎵	M	CR	回车	29	↔	↔	0x0E	GS	组分隔符	45	-	61	13	77	M	93]	109	m	125	}											
1110	E	14	🎵	N	SO	移出	30	▲	▲	0x0F	RS	记录分隔符	46	.	62	14	78	N	94	^	110	n	126	~											
1111	F	15	☼	O	SI	移入	31	▼	▼	0x10	US	单元分隔符	47	/	63	15	79	O	95	_	111	o	127	A											Back space

注：表中的ASCII字符可以用命令:ALT + “小键盘上的数字键” 输入

字符串连接

Java的编译器对字符串做了特殊照顾，可以使用+连接任意字符串和其他数据类型，这样极大地方便了字符串的处理。例如：

```
/**
 * 字符串拼接
 */
public static void concatenate() {
    String s1 = "Hello";
    String s2 = "world";
    String s = s1 + " " + s2 + "!";
    System.out.println(s);
}
```

不可变特性

Java的字符串除了是一个引用类型外，还有个重要特点，就是字符串不可变。考察以下代码：

```
public static void immutability() {
    String s = "hello";
    String t = s;
    s = "world";
    System.out.println(t); // t是"hello"还是"world"?
}
```

小结

- Java的字符类型char是基本类型，字符串类型String是引用类型；
- 基本类型的变量是“持有”某个数值，引用类型的变量是“指向”某个对象；
- 引用类型的变量可以是空值null；
- 要区分空值null和空字符串""。

数组类型

定义一个数组类型的变量，使用数组类型“类型[]”，例如，`int[]`。和单个基本类型变量不同，数组变量初始化必须使用`new int[5]`表示创建一个可容纳5个`int`元素的数组。

Java的数组有几个特点：

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`；数组一旦创建后，大小就不可改变。要访问数组中的某一个元素，需要使用索引。数组索引从0开始，例如，5个元素的数组，索引范围是0~4。

可以修改数组中的某一个元素，使用赋值语句，例如，`ns[1] = 79`；。

可以用数组变量`.length`获取数组大小：

```
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩：
        int[] ns = new int[5];
        System.out.println(ns.length); // 5
    }
}
```

数组是引用类型，在使用索引访问数组元素时，如果索引超出范围，运行时将报错：

```
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩：
        int[] ns = new int[5];
        int n = 5;
        System.out.println(ns[n]); // 索引n不能超出范围
    }
}
```

也可以在定义数组时直接指定初始化的元素，这样就不必写出数组大小，而是由编译器自动推算数组大小。例如：

```
// 5位同学的成绩：
int[] ns = new int[] { 68, 79, 91, 85, 62 };
System.out.println(ns.length); // 编译器自动推算数组大小为5
```

还可以进一步简写为：

```
int[] ns = { 68, 79, 91, 85, 62 };
```

注意数组是引用类型，并且数组大小不可变。我们观察下面的代码：


```
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns;
        ns = new int[] { 68, 79, 91, 85, 62 };
        System.out.println(ns.length); // 5
        ns = new int[] { 1, 2, 3 };
        System.out.println(ns.length); // 3
    }
}
```

数组大小变了吗？看上去好像是变了，但其实根本没变。

于数组`ns`来说，执行`ns = new int[] { 68, 79, 91, 85, 62 };`时，它指向一个5个元素的数组。

执行`ns = new int[] { 1, 2, 3 };`时，它指向一个新的3个元素的数组。

但是，原有的5个元素的数组并没有改变，只是无法通过变量`ns`引用到它们而已。

字符串数组

如果数组元素不是基本类型，而是一个引用类型，那么，修改数组元素会有哪些不同？

字符串是引用类型，因此我们先定义一个字符串数组：

```
public class Main {
    public static void main(String[] args) {
        String[] names = {"ABC", "XYZ", "zoo"};
        String s = names[1];
        names[1] = "cat";
        System.out.println(s); // s是"XYZ"还是"cat"?
    }
}
```

小结

- 数组是同一数据类型的集合，数组一旦创建后，大小就不可变；
- 可以通过索引访问数组元素，但索引超出范围将报错；
- 数组元素可以是值类型（如`int`）或引用类型（如`String`），但数组本身是引用类型；

输入和输出

输出

在前面的代码中，我们总是使用`System.out.println()`来向屏幕输出一些内容。

`println`是`print line`的缩写，表示输出并换行。因此，如果输出后不想换行，可以用`print()`：

```
public class Main {
    public static void main(String[] args) {
        System.out.print("A,");
        System.out.print("B,");
        System.out.print("C.");
        System.out.println();
        System.out.println("END");
    }
}
```

如果要把数据显示成我们期望的格式，就需要使用格式化输出的功能。格式化输出使用`System.out.printf()`，通过使用占位符`%?`，`printf()`可以把后面的参数格式化成指定格式：

```
public class Main {
    public static void main(String[] args) {
        double d = 3.1415926;
        System.out.printf("%.2f\n", d); // 显示两位小数3.14
        System.out.printf("%.4f\n", d); // 显示4位小数3.1416
    }
}
```

Java的格式化功能提供了多种占位符，可以把各种数据类型“格式化”成指定的字符串：

占位符	说明
%d	格式化输出整数
%x	格式化输出十六进制整数
%f	格式化输出浮点数
%e	格式化输出科学计数法表示的浮点数
%s	格式化字符串

注意，由于`%`表示占位符，因此，连续两个`%%`表示一个`%`字符本身。

占位符本身还可以有更详细的格式化参数。下面的例子把一个整数格式化成十六进制，并用0补足8位：

```
public class Main {
    public static void main(String[] args) {
        int n = 12345000;
        System.out.printf("n=%d, hex=%08x", n, n); // 注意，两个%占位符必须传入两个
    }
}
```

输入

和输出相比，Java的输入就要复杂得多。

我们先看一个从控制台读取一个字符串和一个整数的例子：

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // 创建Scanner对象
        System.out.print("Input your name: "); // 打印提示
        String name = scanner.nextLine(); // 读取一行输入并获取字符串
        System.out.print("Input your age: "); // 打印提示
        int age = scanner.nextInt(); // 读取一行输入并获取整数
        System.out.printf("Hi, %s, you are %d\n", name, age); // 格式化输出
    }
}
```

首先，我们通过import语句导入java.util.Scanner，import是导入某个类的语句，必须放到Java源代码的开头，后面我们在Java的package中会详细讲解如何使用import。

然后，创建Scanner对象并传入System.in。System.out代表标准输出流，而System.in代表标准输入流。直接使用System.in读取用户输入虽然是可以的，但需要更复杂的代码，而通过Scanner就可以简化后续的代码。

有了Scanner对象后，要读取用户输入的字符串，使用scanner.nextLine()，要读取用户输入的整数，使用scanner.nextInt()。Scanner会自动转换数据类型，因此不必手动转换。

小结

- Java提供的输出包括：System.out.println() / print() / printf()，其中printf()可以格式化输出；
- Java提供Scanner对象来方便输入，读取对应的类型可以使用：scanner.nextLine() / nextInt() / nextDouble() / ...

if 条件判断

在Java程序中，如果要根据条件来决定是否执行某一段代码，就需要if语句。

if

if语句的基本语法是：

```
if (条件) {  
    // 条件满足时执行  
}
```

根据 if 的计算结果（true还是false），JVM决定是否执行if语句块（即花括号{}包含的所有语句）。

当if语句块只有一行语句时，可以省略花括号{}：

```
public class Main {  
    public static void main(String[] args) {  
        int n = 70;  
        if (n >= 60) {  
            System.out.println("及格了");  
            System.out.println("恭喜你");  
        }  
        System.out.println("END");  
        //省略花括号{  
        int n = 70;  
        if (n >= 60)  
            System.out.println("及格了");  
        System.out.println("END");  
    }  
}
```

if else

if语句还可以编写一个else { ... }，当条件判断为false时，将执行else的语句块：

```
public class Main {  
    public static void main(String[] args) {  
        int n = 70;  
        if (n >= 60) {  
            System.out.println("及格了");  
        } else {  
            System.out.println("挂科了");  
        }  
        System.out.println("END");  
    }  
}
```

还可以用多个if ... else if ...串联。例如：

```

public class Main {
    public static void main(String[] args) {
        int n = 70;
        if (n >= 90) {
            System.out.println("优秀");
        } else if (n >= 60) {
            System.out.println("及格了");
        } else {
            System.out.println("挂科了");
        }
        System.out.println("END");
    }
}

```

浮点数判断

前面讲过了浮点数在计算机中常常无法精确表示，并且计算可能出现误差，因此，判断浮点数相等用`==`判断不靠谱：

```

public class Main {
    public static void main(String[] args) {
        double x = 1 - 9.0 / 10;
        if (x == 0.1) {
            System.out.println("x is 0.1");
        } else {
            System.out.println("x is NOT 0.1");
        }
    }
}

```

正确的方法是利用差值小于某个临界值来判断：

```

public class Main {
    public static void main(String[] args) {
        double x = 1 - 9.0 / 10;
        if (Math.abs(x - 0.1) < 0.00001) {
            System.out.println("x is 0.1");
        } else {
            System.out.println("x is NOT 0.1");
        }
    }
}

```

判断引用类型相等

在Java中，判断值类型的变量是否相等，可以使用`==`运算符。但是，判断引用类型的变量是否相等，`==`表示“引用是否相等”，或者说，是否指向同一个对象。例如，下面的两个String类型，它们的内容是相同的，但是，分别指向不同的对象，用`==`判断，结果为false：

```

public class Main {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "HELLO".toLowerCase();
        System.out.println(s1);
        System.out.println(s2);
        if (s1 == s2) {
            System.out.println("s1 == s2");
        } else {
            System.out.println("s1 != s2");
        }
    }
}

```

要判断引用类型的变量内容是否相等，必须使用equals()方法：

```

public class Main {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "HELLO".toLowerCase();
        System.out.println(s1);
        System.out.println(s2);
        if (s1.equals(s2)) {
            System.out.println("s1 equals s2");
        } else {
            System.out.println("s1 not equals s2");
        }
    }
}

```

注意：执行语句s1.equals(s2)时，如果变量s1为null，会报NullPointerException：

小结

- if ... else 可以做条件判断，else 是可选的；
- 不推荐省略花括号 {} ；
- 多个 if ... else 串联要特别注意判断顺序；
- 要注意 if 的边界条件；
- 要注意浮点数判断相等不能直接用 == 运算符；
- 引用类型判断内容相等要使用 equals() ， 注意避免 NullPointerException 。

switch 多重选择

除了if语句外，还有一种条件判断，是根据某个表达式的结果，分别去执行不同的分支。

例如，在游戏中，让用户选择选项：

- 单人模式
- 多人模式
- 退出游戏

这时，switch语句就派上用场了。switch语句根据switch (表达式)计算的结果，跳转到匹配的case结果，然后继续执行后续语句，直到遇到break结束执行。

我们看一个例子：

```
public class Main {  
    public static void main(String[] args) {  
        int option = 1;  
        switch (option) {  
            case 1:  
                System.out.println("Selected 1");  
                break;  
            case 2:  
                System.out.println("Selected 2");  
                break;  
            case 3:  
                System.out.println("Selected 3");  
                break;  
        }  
    }  
}
```

修改option的值分别为1、2、3，观察执行结果。

如果option的值没有匹配到任何case，例如option = 99，那么，switch语句不会执行任何语句。这时，可以给switch语句加一个default，当没有匹配到任何case时，执行default：

```
public class Main {
    public static void main(String[] args) {
        int option = 99;
        switch (option) {
            case 1:
                System.out.println("Selected 1");
                break;
            case 2:
                System.out.println("Selected 2");
                break;
            case 3:
                System.out.println("Selected 3");
                break;
            default:
                System.out.println("Not selected");
                break;
        }
    }
}
```

同时注意，上述“翻译”只有在switch语句中对每个case正确编写了break语句才能对应得上。

使用switch时，注意case语句并没有花括号{}，而且，case语句具有“穿透性”，漏写break将导致意想不到的结果：

```
public class Main {
    public static void main(String[] args) {
        int option = 2;
        switch (option) {
            case 1:
                System.out.println("Selected 1");
            case 2:
                System.out.println("Selected 2");
            case 3:
                System.out.println("Selected 3");
            default:
                System.out.println("Not selected");
        }
    }
}
```

当option = 2时，将依次输出"Selected 2"、"Selected 3"、"Not selected"，原因是从匹配到case 2开始，后续语句将全部执行，直到遇到break语句。因此，任何时候都不要忘记写break。

小结

- switch语句可以做多重选择，然后执行匹配的case语句后续代码；
- switch的计算结果必须是整型、字符串或枚举类型；
- 注意千万不要漏写break，建议打开fall-through警告；
- 总是写上default，建议打开missing default警告；

while 和 do while 循环

while

循环语句就是让计算机根据条件做循环计算，在条件满足时继续循环，条件不满足时退出循环。

while 循环在每次循环开始前，首先判断条件是否成立。如果计算结果为true，就把循环体内的语句执行一遍，如果计算结果为false，那就直接跳到while循环的末尾，继续往下执行。

基本用法如下：

```
while (条件表达式) {  
    循环语句  
}  
// 继续执行后续代码
```

例如，计算从1到100的和：

```
public class Main {  
    public static void main(String[] args) {  
        int sum = 0; // 累加的和，初始化为0  
        int n = 1;  
        while (n <= 100) { // 循环条件是n <= 100  
            sum = sum + n; // 把n累加到sum中  
            n ++; // n自身加1  
        }  
        System.out.println(sum); // 5050  
    }  
}
```

如果循环条件永远满足，那这个循环就变成了死循环。死循环将导致100%的CPU占用，用户会感觉电脑运行缓慢，所以要避免编写死循环代码。

如果循环条件的逻辑写得有问题，也会造成意料之外的结果：

```
public class Main {  
    public static void main(String[] args) {  
        int sum = 0;  
        int n = 1;  
        while (n > 0) {  
            sum = sum + n;  
            n ++;  
        }  
        System.out.println(n); // -2147483648  
        System.out.println(sum);  
    }  
}
```

表面上看，上面的while循环是一个死循环，但是，Java的int类型有最大值，达到最大值后，再加1会变成负数，结果，意外退出了while循环。

do while

在Java中，while循环是先判断循环条件，再执行循环。而另一种do while循环则是先执行循环，再判断条件，条件满足时继续循环，条件不满足时退出。它的用法是：

```
do {  
    执行循环语句  
} while (条件表达式);
```

可见，do while循环会至少循环一次。

我们把对1到100的求和用do while循环改写一下：

```
public class Main {  
    public static void main(String[] args) {  
        int sum = 0;  
        int n = 1;  
        do {  
            sum = sum + n;  
            n ++;  
        } while (n <= 100);  
        System.out.println(sum);  
    }  
}
```

小结

- while 循环先判断循环条件是否满足，再执行循环语句；
- while 循环可能一次都不执行；
- 编写循环时要注意循环条件，并避免死循环。
- do while 循环先执行循环，再判断条件；
- do while 循环会至少执行一次。

for 循环

除了while和do while循环，Java使用最广泛的是for循环。

for循环的功能非常强大，它使用计数器实现循环。for循环会先初始化计数器，然后，在每次循环前检测循环条件，在每次循环后更新计数器。计数器变量通常命名为i。

我们把1到100求和用for循环改写一下：

```
public class Main {
    public static void main(String[] args) {
        int sum = 0;
        for (int i=1; i<=100; i++) {
            sum = sum + i;
        }
        System.out.println(sum);
    }
}
```

在for循环执行前，会先执行初始化语句int i=1，它定义了计数器变量i并赋初始值为1，然后，循环前先检查循环条件i<=100，循环后自动执行i++，因此，和while循环相比，for循环把更新计数器的代码统一放到了一起。在for循环的循环体内部，不需要去更新变量i。

因此，for循环的用法是：

```
for (初始条件; 循环检测条件; 循环后更新计数器) {
    // 执行语句
}
```

注意for循环的初始化计数器总是会被执行，并且for循环也可能循环0次。

巧用for循环

for循环还可以缺少初始化语句、循环条件和每次循环更新语句，例如：

```
// 不设置结束条件：
for (int i=0; ; i++) {
    ...
}
// 不设置结束条件和更新语句：
for (int i=0; ;) {
    ...
}
// 什么都不设置：
for (;;) {
    ...
}
```

for each循环

Java还提供了另一种for each循环，它可以更简单地遍历数组或则集合中的元素：

```
public class Main {  
    public static void main(String[] args) {  
        int[] ns = { 1, 4, 9, 16, 25 };  
        for (int n : ns) {  
            System.out.println(n);  
        }  
    }  
}
```

和for循环相比，for each循环的变量n不再是计数器，而是直接对应到数组的每个元素。for each循环的写法也更简洁。但是，for each循环无法指定遍历顺序，也无法获取数组的索引。

除了数组外，for each循环能够遍历所有“可迭代”的数据类型，包括List、Map等。

小结

- for循环通过计数器可以实现复杂循环；
- for each循环可以直接遍历数组的每个元素；
- 最佳实践：计数器变量定义在for循环内部，循环体内部不修改计数器；

break 和 continue

无论是while循环还是for循环，有两个特别的语句可以使用，就是break语句和continue语句。

break

在循环过程中，可以使用break语句跳出当前循环。我们来看一个例子：

```
public class Main {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int i=1; ; i++) {  
            sum = sum + i;  
            if (i == 100) {  
                break;  
            }  
        }  
        System.out.println(sum);  
    }  
}
```

使用for循环计算从1到100时，我们并没有在for()中设置循环退出的检测条件。但是，在循环内部，我们用if判断，如果i==100，就通过break退出循环。

因此，break语句通常都是配合if语句使用。要特别注意，break语句总是跳出自己所在的那一层循环。例如：

```
public class Main {  
    public static void main(String[] args) {  
        for (int i=1; i<=10; i++) {  
            System.out.println("i = " + i);  
            for (int j=1; j<=10; j++) {  
                System.out.println("j = " + j);  
                if (j >= i) {  
                    break;  
                }  
            }  
            // break跳到这里  
            System.out.println("breaKed");  
        }  
    }  
}
```

实际上break可以跳出任何一层循环，使用 `loop_name:` 加在循环判断语句前，使用 `break loop_name` 跳出指定循环层，例如：

```

public class Main {
    public static void main(String[] args) {
        outer: for (int i=1; i<=10; i++) {
            System.out.println("i = " + i);
            inner: for (int j=1; j<=10; j++) {
                System.out.println("j = " + j);
                if (j >= i) {
                    break outer;
                }
            }
        }
        // break跳到这里
        System.out.println("breaKed");
    }
}

```

continue

break会跳出当前循环，也就是整个循环都不会执行了。而continue则是提前结束本次循环，直接继续执行下次循环。我们看一个例子：

```

public class Main {
    public static void main(String[] args) {
        int sum = 0;
        for (int i=1; i<=10; i++) {
            System.out.println("begin i = " + i);
            if (i % 2 == 0) {
                continue; // continue语句会结束本次循环
            }
            sum = sum + i;
            System.out.println("end i = " + i);
        }
        System.out.println(sum); // 25
    }
}

```

注意观察continue语句的效果。当i为奇数时，完整地执行了整个循环，因此，会打印begin i=1和end i=1。在i为偶数时，continue语句会提前结束本次循环，因此，会打印begin i=2但不会打印end i = 2。

在多层嵌套的循环中，continue语句同样是结束本次自己所在的循环。

小结

- break 语句可以跳出当前循环；
- break 语句通常配合 if，在满足条件时提前结束整个循环；
- break 语句总是跳出最近的一层循环；
- continue 语句可以提前结束本次循环；
- continue 语句通常配合 if，在满足条件时提前结束本次循环。

数组遍历

我们在Java程序基础里介绍了数组这种数据类型。有了数组，我们还需要来操作它。而数组最常见的一个操作就是遍历。

通过for循环就可以遍历数组。因为数组的每个元素都可以通过索引来访问，因此，使用标准的for循环可以完成一个数组的遍历：

```
public class Main {  
    public static void main(String[] args) {  
        int[] ns = { 1, 4, 9, 16, 25 };  
        for (int i=0; i<ns.length; i++) {  
            int n = ns[i];  
            System.out.println(n);  
        }  
    }  
}
```

为了实现for循环遍历，初始条件为i=0，因为索引总是从0开始，继续循环的条件为i<ns.length，因为当i=ns.length时，i已经超出了索引范围（索引范围是0 ~ ns.length-1），每次循环后，i++。

第二种方式是使用for each循环，直接迭代数组的每个元素：

```
public class Main {  
    public static void main(String[] args) {  
        int[] ns = { 1, 4, 9, 16, 25 };  
        for (int n : ns) {  
            System.out.println(n);  
        }  
    }  
}
```

注意：在for (int n : ns)循环中，变量n直接拿到ns数组的元素，而不是索引。

显然for each循环更加简洁。但是，for each循环无法拿到数组的索引，因此，到底用哪一种for循环，取决于我们的需要。

如果你想打印数组内容，Java标准库提供了Arrays.toString()，可以快速打印数组内容：

```
import java.util.Arrays;  
  
public class Main {  
    public static void main(String[] args) {  
        int[] ns = { 1, 1, 2, 3, 5, 8 };  
        System.out.println(Arrays.toString(ns));  
    }  
}
```

小结

方法

- 遍历数组可以使用 `for` 循环，`for` 循环可以访问数组索引，`for each` 循环直接迭代每个数组元素，但无法获取索引；
- 使用 `Arrays.toString()` 可以快速获取数组内容。

数组排序

对数组进行排序是程序中非常基本的需求。常用的排序算法有冒泡排序、插入排序和快速排序等。

我们来看一下如何使用冒泡排序算法对一个整型数组从小到大进行排序：

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] ns = { 28, 12, 89, 73, 65, 18, 96, 50, 8, 36 };
        // 排序前:
        System.out.println(Arrays.toString(ns));
        for (int i = 0; i < ns.length - 1; i++) {
            for (int j = 0; j < ns.length - i - 1; j++) {
                if (ns[j] > ns[j+1]) {
                    // 交换ns[j]和ns[j+1]:
                    int tmp = ns[j];
                    ns[j] = ns[j+1];
                    ns[j+1] = tmp;
                }
            }
        }
        // 排序后:
        System.out.println(Arrays.toString(ns));
    }
}
```

冒泡排序的特点是，每一轮循环后，最大的一个数被交换到末尾，因此，下一轮循环就可以“刨除”最后的数，每一轮循环都比上一轮循环的结束位置靠前一位。

另外，注意到交换两个变量的值必须借助一个临时变量。

实际上，Java的标准库已经内置了排序功能，我们只需要调用JDK提供的 `Arrays.sort()` 就可以排序（排序的类实现了 `Comparable` 接口）：

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] ns = { 28, 12, 89, 73, 65, 18, 96, 50, 8, 36 };
        Arrays.sort(ns);
        System.out.println(Arrays.toString(ns));
    }
}
```

小结

- 常用的排序算法有冒泡排序、插入排序和快速排序等；
- 冒泡排序使用两层 `for` 循环实现排序；
- 交换两个变量的值需要借助一个临时变量。
- 可以直接使用Java标准库提供的 `Arrays.sort()` 进行排序；
- 对数组排序会直接修改数组本身。

多维数组

二维数组

二维数组就是数组的数组。定义一个二维数组如下：

```
public class Main {
    public static void main(String[] args) {
        int[][] ns = {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 }
        };
        System.out.println(ns.length); // 3
    }
}
```

访问二维数组的某个元素需要使用 `array[row][col]`，例如：

```
System.out.println(ns[1][2]); // 7
```

二维数组的每个数组元素的长度并不要求相同，例如，可以这么定义ns数组：

```
int[][] ns = {
    { 1, 2, 3, 4 },
    { 5, 6 },
    { 7, 8, 9 }
};
```

要打印一个二维数组，可以使用Java标准库的`Arrays.deepToString()`：

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[][] ns = {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 }
        };
        System.out.println(Arrays.deepToString(ns));
    }
}
```

小结

- 二维数组就是数组的数组。
- 多维数组的每个数组元素长度都不要求相同；
 - 打印多维数组可以使用 `Arrays.deepToString()`；
- 最常见的多维数组是二维数组，访问二维数组的一个元素使用 `array[row][col]`。

命令行参数

Java程序的入口是 `main` 方法，而 `main` 方法可以接受一个命令行参数，它是一个 `String[]` 数组。

这个命令行参数由 JVM 接收用户输入并传给 `main` 方法：

```
public class Main {  
    public static void main(String[] args) {  
        for (String arg : args) {  
            System.out.println(arg);  
        }  
    }  
}
```

java 执行并传递命令行参数

我们可以利用接收到的命令行参数，根据不同的参数执行不同的代码。例如，实现一个 `-version` 参数，打印程序版本号：

```
public class Main {  
    public static void main(String[] args) {  
        for (String arg : args) {  
            if ("-version".equals(arg)) {  
                System.out.println("v 1.0");  
                break;  
            }  
        }  
    }  
}
```

上面这个程序必须在命令行执行，我们先编译它：

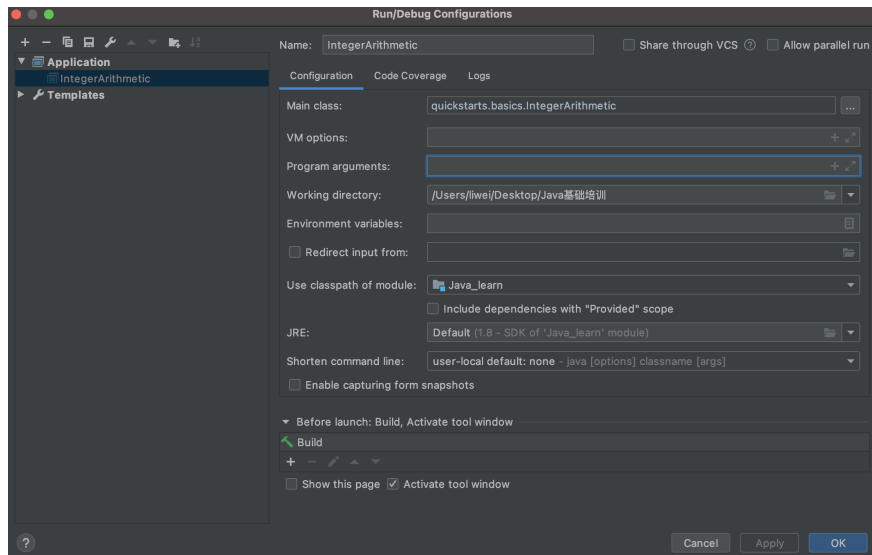
```
javac Main.java
```

然后，执行的时候，给它传递一个 `-version` 参数：

```
java Main -version
```

IDEA 配置命令行参数

如果一个类有 `main`（即有入口方法，可以执行），可以通过为其配置 Run/Debug Configuration，然后在右侧箭头的 Program arguments 里输入参数，不同参数用空格隔开。



小结

- 命令行参数类型是 `String[]` 数组；
- 命令行参数由 JVM 接收用户输入并传给 `main` 方法；
- 如何解析命令行参数需要由程序自己实现。

方法

定义方法

定义方法的语法是：

```
修饰符 方法返回类型 方法名(方法参数列表) {  
    若干方法语句;  
    return 方法返回值;  
}
```

方法返回值通过 `return` 语句实现，如果没有返回值，返回类型设置为 `void`，可以省略 `return`。

例如：

```
class Person {  
    private String name;  
    private int age;  
  
    public String getName() {  
        return this.name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return this.age;  
    }  
  
    public void setAge(int age) {  
        if (age < 0 || age > 100) {  
            throw new IllegalArgumentException("invalid age value");  
        }  
        this.age = age;  
    }  
}
```

修饰符

Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Java支持 4 种不同的访问权限：

- `default` (即默认，什么也不写)：在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。
- `private`：在同一类内可见。使用对象：变量、方法。注意：不能修饰类（外部类）
- `public`：对所有类可见。使用对象：类、接口、变量、方法
- `protected`：对同一包内的类和所有子类可见。使用对象：变量、方法。注意：不能修饰类（外部类）。

修饰符	当前类	同一包内	子孙类(同一包)	子孙类(不同包)	其他包
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	Y/N	N
default	Y	Y	Y	N	N
private	Y	N	N	N	N

受保护的访问修饰符-protected protected 需要从以下两个点来分析说明：

- 子类与基类在同一包中：被声明为 protected 的变量、方法和构造器能被同一个包中的任何其他类访问；
- 子类与基类不在同一包中：那么在子类中，子类实例可以访问其从基类继承而来的 protected 方法，而不能访问基类实例的protected方法。

protected 可以修饰数据成员，构造方法，方法成员，不能修饰类（内部类除外）。

接口及接口的成员变量和成员方法不能声明为 protected。

this变量

在方法内部，可以使用一个隐含的变量this，它始终指向当前实例。因此，通过this.field就可以访问当前实例的字段。

如果没有命名冲突，可以省略this。例如：

```
class Person {
    private String name;

    public String getName() {
        return name; // 相当于this.name
    }
}
```

但是，如果有局部变量和字段重名，那么局部变量优先级更高，就必须加上this：

```
class Person {
    private String name;

    public void setName(String name) {
        this.name = name; // 前面的this不可少，少了就变成局部变量name了
    }
}
```

方法参数

方法可以包含0个或任意个参数。方法参数用于接收传递给方法的变量值。调用方法时，必须严格按照参数的定义一一传递。例如：

```
class Person {  
    ...  
    public void setNameAndAge(String name, int age) {  
        ...  
    }  
}
```

调用这个setNameAndAge()方法时，必须有两个参数，且第一个参数必须为String，第二个参数必须为int：

```
Person ming = new Person();  
ming.setNameAndAge("Xiao Ming"); // 编译错误：参数个数不对  
ming.setNameAndAge(12, "Xiao Ming"); // 编译错误：参数类型不对
```

可变参数

可变参数用 类型... 定义，可变参数相当于数组类型：

```
class Group {  
    private String[] names;  
  
    public void setNames(String... names) {  
        this.names = names;  
    }  
}
```

上面的setNames()就定义了一个可变参数。调用时，可以这么写：

```
Group g = new Group();  
g.setNames("Xiao Ming", "Xiao Hong", "Xiao Jun"); // 传入3个String  
g.setNames("Xiao Ming", "Xiao Hong"); // 传入2个String  
g.setNames("Xiao Ming"); // 传入1个String  
g.setNames(); // 传入0个String
```

参数绑定

基本类型参数的传递，是调用方值的复制。双方各自的后续修改，互不影响。


```

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        int n = 15; // n的值为15
        p.setAge(n); // 传入n的值
        System.out.println(p.getAge()); // 15
        n = 20; // n的值改为20
        System.out.println(p.getAge()); // 15还是20?
    }
}

class Person {
    private int age;

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

引用类型参数的传递，调用方的变量，和接收方的参数变量，指向的是同一个对象。双方任意一方对这个对象的修改，都会影响对方（因为指向同一个对象嘛）。

```

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        String[] fullname = new String[] { "Homer", "Simpson" };
        p.setName(fullname); // 传入fullname数组
        System.out.println(p.getName()); // "Homer Simpson"
        fullname[0] = "Bart"; // fullname数组的第一个元素修改为"Bart"
        System.out.println(p.getName()); // "Homer Simpson"还是"Bart Simpson"?
    }
}

class Person {
    private String[] name;

    public String getName() {
        return this.name[0] + " " + this.name[1];
    }

    public void setName(String[] name) {
        this.name = name;
    }
}

```

小结

- 方法可以让外部代码安全地访问实例字段；
- 方法是一组执行语句，并且可以执行任意逻辑；
- 方法内部遇到 `return` 时返回，`void` 表示不返回任何值（注意和返回 `null` 不同）；
- 外部代码通过 `public` 方法操作实例，内部代码可以调用 `private` 方法；
- 理解方法的参数绑定。

构造方法

创建实例的时候，我们经常需要同时初始化这个实例的字段，例如：

```
Person ming = new Person();
ming.setName("小明");
ming.setAge(12);
```

初始化对象实例需要3行代码，而且，如果忘了调用setName()或者setAge()，这个实例内部的状态就是不正确的。

能否在创建对象实例时就把内部字段全部初始化为合适的值？

完全可以。

这时，我们就需要构造方法。

创建实例的时候，实际上是通过构造方法来初始化实例的。我们先来定义一个构造方法，能在创建Person实例的时候，一次性传入name和age，完成初始化：

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person("Xiao Ming", 15);
        System.out.println(p.getName());
        System.out.println(p.getAge());
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

由于构造方法是如此特殊，所以构造方法的名称就是类名。构造方法的参数没有限制，在方法内部，也可以编写任意语句。但是，和普通方法相比，构造方法没有返回值（也没有void），调用构造方法，必须用new操作符。

默认构造方法

是不是任何class都有构造方法？是的。

那前面我们并没有为Person类编写构造方法，为什么可以调用new Person()？

原因是如果一个类没有定义构造方法，编译器会自动为我们生成一个默认构造方法，它没有参数，也没有执行语句，类似这样：

```
class Person {  
    public Person() {  
    }  
}
```

要特别注意的是，如果我们自定义了一个构造方法，那么，编译器就不再自动创建默认构造方法。

如果既要能使用带参数的构造方法，又想保留不带参数的构造方法，那么只能把两个构造方法都定义出来：

```
public class Main {  
    public static void main(String[] args) {  
        Person p1 = new Person("Xiao Ming", 15); // 既可以调用带参数的构造方法  
        Person p2 = new Person(); // 也可以调用无参数构造方法  
    }  
}  
  
class Person {  
    private String name;  
    private int age;  
  
    public Person() {  
    }  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public int getAge() {  
        return this.age;  
    }  
}
```

没有在构造方法中初始化字段时，引用类型的字段默认是null，数值类型的字段用默认值，int类型默认值是0，布尔类型默认值是false。

在Java中，创建对象实例的时候，按照如下顺序进行初始化：

1. 先初始化字段，例如，int age = 10;表示字段初始化为10，double salary;表示字段默认初始化为0，String name;表示引用类型字段默认初始化为null；
2. 执行构造方法的代码进行初始化。

因此，构造方法的代码由于后运行，所以，既对字段进行初始化，又在构造方法中对字段进行初始化时，字段值最终由构造方法的代码确定。

多构造方法

可以定义多个构造方法，在通过new操作符调用的时候，编译器通过构造方法的参数数量、位置和类型自动区分：

```
class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public Person(String name) {  
        this.name = name;  
        this.age = 12;  
    }  
  
    public Person() {  
    }  
}
```

如果调用new Person("Xiao Ming", 20);, 会自动匹配到构造方法public Person(String, int)。

如果调用new Person("Xiao Ming");, 会自动匹配到构造方法public Person(String)。

如果调用new Person();, 会自动匹配到构造方法public Person()。

一个构造方法可以调用其他构造方法，这样做的目的是便于代码复用。调用其他构造方法的语法是this(...)：

```
class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public Person(String name) {  
        this(name, 18); // 调用另一个构造方法Person(String, int)  
    }  
  
    public Person() {  
        this("Unnamed"); // 调用另一个构造方法Person(String)  
    }  
}
```

小结

- 实例在创建时通过new操作符会调用其对应的构造方法，构造方法用于初始化实例；
- 没有定义构造方法时，编译器会自动创建一个默认的无参数构造方法；
- 可以定义多个构造方法，编译器根据参数自动判断；
- 可以在一个构造方法内部调用另一个构造方法，便于代码复用。

方法重载

在一个类中，我们可以定义多个方法。如果有一系列方法，它们的功能都是类似的，只有参数有所不同，那么，可以把这一组方法名做成同名方法。例如，在Hello类中，定义多个hello()方法：

```
class Hello {  
    public void hello() {  
        System.out.println("Hello, world!");  
    }  
  
    public void hello(String name) {  
        System.out.println("Hello, " + name + "!");  
    }  
  
    public void hello(String name, int age) {  
        if (age < 18) {  
            System.out.println("Hi, " + name + "!");  
        } else {  
            System.out.println("Hello, " + name + "!");  
        }  
    }  
}
```

这种方法名相同，但各自的参数不同，称为方法重载（Overload）。

注意：方法重载的返回值类型通常都是相同的。

方法重载的目的是，功能类似的方法使用同一名字，更容易记住，因此，调用起来更简单。

举个例子，String类提供了多个重载方法indexOf()，可以查找子串：

- int indexOf(int ch)：根据字符的Unicode码查找；
- int indexOf(String str)：根据字符串查找；
- int indexOf(int ch, int fromIndex)：根据字符查找，但指定起始位置；
- int indexOf(String str, int fromIndex)根据字符串查找，但指定起始位置。

试一试：

```
public class Main {  
    public static void main(String[] args) {  
        String s = "Test string";  
        int n1 = s.indexOf('t');  
        int n2 = s.indexOf("st");  
        int n3 = s.indexOf("st", 4);  
        System.out.println(n1);  
        System.out.println(n2);  
        System.out.println(n3);  
    }  
}
```

小结

- 方法重载是指多个方法的方法名相同，但各自的参数不同；

方法

- 重载方法应该完成类似的功能，参考 `String` 的 `indexOf()` ；
- 重载方法返回值类型应该相同。

继承

在前面的章节中，我们已经定义了Person类：

```
class Person {  
    private String name;  
    private int age;  
  
    public String getName() {...}  
    public void setName(String name) {...}  
    public int getAge() {...}  
    public void setAge(int age) {...}  
}
```

现在，假设需要定义一个Student类，字段如下：

```
class Student {  
    private String name;  
    private int age;  
    private int score;  
  
    public String getName() {...}  
    public void setName(String name) {...}  
    public int getAge() {...}  
    public void setAge(int age) {...}  
    public int getScore() { ... }  
    public void setScore(int score) { ... }  
}
```

仔细观察，发现Student类包含了Person类已有的字段和方法，只是多出了一个score字段和相应的getScore()、setScore()方法。

能不能在Student中不要写重复的代码？

这个时候，继承就派上用场了。

继承是面向对象编程中非常强大的一种机制，它首先可以复用代码。当我们让Student从Person继承时，Student就获得了Person的所有功能，我们只需要为Student编写新增的功能。

Java使用extends关键字来实现继承：

```

class Person {
    private String name;
    private int age;

    public String getName() {...}
    public void setName(String name) {...}
    public int getAge() {...}
    public void setAge(int age) {...}
}

class Student extends Person {
    // 不要重复name和age字段/方法,
    // 只需要定义新增score字段/方法:
    private int score;

    public int getScore() { ... }
    public void setScore(int score) { ... }
}

```

可见，通过继承，Student只需要编写额外的功能，不再需要重复代码。**注意：**子类自动获得了父类的所有字段，**严禁定义与父类重名的字段！**在OOP的术语中，我们把Person称为超类（super class），父类（parent class），基类（base class），把Student称为子类（subclass），扩展类（extended class）。

继承树

Object 是除本身外所有类的超类，任何类，除了Object，都会继承自某个类。Java只允许一个class继承自一个类，因此，一个类有且仅有一个父类。只有Object特殊，它没有父类。

在 IDEA 中，在类文件中右键，选择 Diagrams 就能查看类继承关系。

protected

继承有个特点，就是子类无法访问父类的private字段或者private方法。例如，Student类就无法访问Person类的name和age字段：

```

class Person {
    private String name;
    private int age;
}

class Student extends Person {
    public String hello() {
        return "Hello, " + name; // 编译错误：无法访问name字段
    }
}

```

这使得继承的作用被削弱了。为了让子类可以访问父类的字段，我们需要把private改为protected。用protected修饰的字段可以被子类访问：


```
class Person {
    protected String name;
    protected int age;
}

class Student extends Person {
    public String hello() {
        return "Hello, " + name; // OK!
    }
}
```

因此，protected关键字可以把字段和方法的访问权限控制在继承树内部，一个protected字段和方法可以被其子类，以及子类的子类所访问。

super

super关键字表示父类（超类）。子类引用父类的字段时，可以用super.fieldName。例如：

```
class Student extends Person {
    public String hello() {
        return "Hello, " + super.name;
    }
}
```

在Java中，任何class的构造方法，第一行语句必须是调用父类的构造方法。如果没有明确地调用父类的构造方法，编译器会帮我们自动加一句super();

如果父类没有默认的构造方法，子类就必须显式调用super()并给出参数以便让编译器定位到父类的一个合适的构造方法。

这里还顺带引出了另一个问题：即子类不会继承任何父类的构造方法。子类默认的构造方法是编译器自动生成的，不是继承的。

阻止继承

正常情况下，只要某个class没有final修饰符，那么任何类都可以从该class继承。例如：

```
public final class Student extends Person{...}
```

向上转型

如果一个引用变量的类型是Student，那么它可以指向一个Student类型的实例：

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

如果一个引用类型的变量是Person，那么它可以指向一个Person类型的实例：

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

现在问题来了：如果Student是从Person继承下来的，那么，一个引用类型为Person的变量，能否指向Student类型的实例？

```
Person p = new Student(); // ???
```

测试一下就可以发现，这种指向是允许的！

这是因为Student继承自Person，因此，它拥有Person的全部功能。Person类型的变量，如果指向Student类型的实例，对它进行操作，是没有问题的！

这种把一个子类类型安全地变为父类类型的赋值，被称为向上转型（upcasting）。

向上转型实际上是把一个子类型安全地变为更加抽象的父类型：

```
Student s = new Student();
Person p = s; // upcasting, ok
Object o1 = p; // upcasting, ok
Object o2 = s; // upcasting, ok
```

注意到继承树是Student > Person > Object，所以，可以把Student类型转型为Person，或者更高层次的Object。

向下转型

和向上转型相反，如果把一个父类类型强制转型为子类类型，就是向下转型（downcasting）。例如：

```
Person p1 = new Student(); // upcasting, ok
Person p2 = new Person();
Student s1 = (Student) p1; // ok
Student s2 = (Student) p2; // runtime error! ClassCastException!
```

如果测试上面的代码，可以发现：

Person类型p1实际指向Student实例，Person类型变量p2实际指向Person实例。在向下转型的时候，把p1转型为Student会成功，因为p1确实指向Student实例，把p2转型为Student会失败，因为p2的实际类型是Person，不能把父类变为子类，因为子类功能比父类多，多的功能无法凭空变出来。

因此，向下转型很可能会失败。失败的时候，Java虚拟机会报ClassCastException。

为了避免向下转型出错，Java提供了instanceof操作符，可以先判断一个实例究竟是不是某种类型：

```
Person p = new Person();
System.out.println(p instanceof Person); // true
System.out.println(p instanceof Student); // false

Student s = new Student();
System.out.println(s instanceof Person); // true
System.out.println(s instanceof Student); // true

Student n = null;
System.out.println(n instanceof Student); // false
```

instanceof实际上判断一个变量所指向的实例是否是指定类型，或者这个类型的子类。如果一个引用变量为null，那么对任何instanceof的判断都为false。

利用instanceof，在向下转型前可以先判断：

```
Person p = new Student();
if (p instanceof Student) {
    // 只有判断成功才会向下转型：
    Student s = (Student) p; // 一定会成功
}
```

小结

- 继承是面向对象编程的一种强大的代码复用方式；
- Java只允许单继承，所有类最终的根类是Object；
- protected允许子类访问父类的字段和方法；
- 子类的构造方法可以通过super()调用父类的构造方法；
- 可以安全地向上转型为更抽象的类型；
- 可以强制向下转型，最好借助instanceof判断；
- 子类和父类的关系是is，has关系不能用继承。

多态

在继承关系中，子类如果定义了一个与父类方法签名完全相同的方法，被称为覆写（Override）。

例如，在Person类中，我们定义了run()方法：

```
class Person {  
    public void run() {  
        System.out.println("Person.run");  
    }  
}
```

在子类Student中，覆写这个run()方法：

```
class Student extends Person {  
    @Override  
    public void run() {  
        System.out.println("Student.run");  
    }  
}
```

Override和Overload不同的是，如果方法签名不同，就是Overload，Overload方法是一个新方法；如果方法签名相同，并且返回值也相同，就是Override。

加上 `@Override` 可以让编译器帮助检查是否进行了正确的覆写，但是 `@Override` 不是必需的。希望进行覆写，但是不小心写错了方法签名，编译器会报错。

多态

那么，一个实际类型为Student，引用类型为Person的变量，调用其run()方法，调用的是Person还是Student的run()方法？

运行一下上面的代码就可以知道，实际调用的方法是Student的run()方法。因此可得出结论：

Java的实例方法调用是基于运行时的实际类型的动态调用，而非变量的声明类型。

这个非常重要的特性在面向对象编程中称之为多态。

即多态是指，针对某个类型的方法调用，其真正执行的方法取决于运行时期实际类型的方法。例如：

```

public class Main {
    public static void main(String[] args) {
        // 给一个有普通收入、工资收入和享受国务院特殊津贴的小伙伴算税：
        Income[] incomes = new Income[] {
            new Income(3000),
            new Salary(7500),
            new StateCouncilSpecialAllowance(15000)
        };
        System.out.println(totalTax(incomes));
    }

    public static double totalTax(Income... incomes) {
        double total = 0;
        for (Income income: incomes) {
            total = total + income.getTax();
        }
        return total;
    }
}

class Income {
    protected double income;

    public Income(double income) {
        this.income = income;
    }

    public double getTax() {
        return income * 0.1; // 税率10%
    }
}

class Salary extends Income {
    public Salary(double income) {
        super(income);
    }

    @Override
    public double getTax() {
        if (income <= 5000) {
            return 0;
        }
        return (income - 5000) * 0.2;
    }
}

class StateCouncilSpecialAllowance extends Income {
    public StateCouncilSpecialAllowance(double income) {
        super(income);
    }

    @Override
    public double getTax() {
        return 0;
    }
}

```

观察totalTax()方法：利用多态，totalTax()方法只需要和Income打交道，它完全不需要知道Salary和StateCouncilSpecialAllowance的存在，就可以正确计算出总的税。如果我们要新增一种稿费收入，只需要从Income派生，然后正确覆写getTax()方法就可以。把新的类型传入totalTax()，不需要修改任何代码。

所以，多态的特性就是，运行期才能动态决定调用的子类方法。对某个类型调用某个方法，执行的实际方法可能是某个子类的覆写方法。

多态具有一个非常强大的功能，就是允许添加更多类型的子类实现功能扩展，却不需要修改基于父类的代码。

小结

- 子类可以覆写父类的方法（Override），覆写在子类中改变了父类方法的行为；
- Java的方法调用总是作用于运行期对象的实际类型，这种行为称为多态；
- final修饰符有多种作用：
 - final修饰的方法可以阻止被覆写；
 - final修饰的class可以阻止被继承；
 - final修饰的field必须在创建对象时初始化，随后不可修改。

抽象方法

由于多态的存在，每个子类都可以覆写父类的方法，例如：

```
class Person {  
    public void run() { ... }  
}  
  
class Student extends Person {  
    @Override  
    public void run() { ... }  
}  
  
class Teacher extends Person {  
    @Override  
    public void run() { ... }  
}
```

从Person类派生的Student和Teacher都可以覆写run()方法。

如果父类Person的run()方法没有实际意义，能否去掉方法的执行语句？

```
class Person {  
    public void run(); // Compile Error!  
}
```

答案是不行，会导致编译错误，因为定义方法的时候，必须实现方法的语句。

如果父类的方法本身不需要实现任何功能，仅仅是为了定义方法签名，目的是让子类去覆写它，那么，可以把父类的方法声明为抽象方法：

```
class Person {  
    public abstract void run();  
}
```

把一个方法声明为abstract，表示它是一个抽象方法，本身没有实现任何方法语句。因为这个抽象方法本身是无法执行的，所以，Person类也无法被实例化。编译器会告诉我们，无法编译Person类，因为它包含抽象方法。

必须把Person类本身也声明为abstract，才能正确编译它：

```
abstract class Person {  
    public abstract void run();  
}
```

抽象类

如果一个class定义了方法，但没有具体执行代码，这个方法就是抽象方法，抽象方法用abstract修饰。

因为无法执行抽象方法，因此这个类也必须申明为抽象类（abstract class）。

使用abstract修饰的类就是抽象类。我们无法实例化一个抽象类：

```
Person p = new Person(); // 编译错误
```

无法实例化的抽象类有什么用？

因为抽象类本身被设计成只能用于被继承，因此，抽象类可以强迫子类实现其定义的抽象方法，否则编译会报错。因此，抽象方法实际上相当于定义了“规范”。

例如，Person类定义了抽象方法run()，那么，在实现子类Student的时候，就必须覆写run()方法：

```
public class Main {
    public static void main(String[] args) {
        Person p = new Student();
        p.run();
    }
}

abstract class Person {
    public abstract void run();
}

class Student extends Person {
    @Override
    public void run() {
        System.out.println("Student.run");
    }
}
```

面向抽象编程

当我们定义了抽象类Person，以及具体的Student、Teacher子类的时候，我们可以通过抽象类Person类型去引用具体的子类的实例：

```
Person s = new Student();
Person t = new Teacher();
```

这种引用抽象类的好处在于，我们对其进行方法调用，并不关心Person类型变量的具体子类型：

```
// 不关心Person变量的具体子类型：
s.run();
t.run();
```

同样的代码，如果引用的是一个新的子类，我们仍然不关心具体类型：

```
// 同样不关心新的子类是如何实现run()方法的：
Person e = new Employee();
e.run();
```

这种尽量引用高层类型，避免引用实际子类型的方式，称之为面向抽象编程。

面向抽象编程的本质就是：

- 上层代码只定义规范（例如：`abstract class Person`）；
- 不需要子类就可以实现业务逻辑（正常编译）；
- 具体的业务逻辑由不同的子类实现，调用者并不关心。

小结

- 通过`abstract`定义的方法是抽象方法，它只有定义，没有实现。抽象方法定义了子类必须实现的接口规范；
- 定义了抽象方法的`class`必须被定义为抽象类，从抽象类继承的子类必须实现抽象方法；
- 如果不实现抽象方法，则该子类仍是一个抽象类；
- 面向抽象编程使得调用者只关心抽象方法的定义，不关心子类的具体实现。

接口

在抽象类中，抽象方法本质上是定义接口规范：即规定高层类的接口，从而保证所有子类都有相同的接口实现，这样，多态就能发挥出威力。

如果一个抽象类没有字段，所有方法全部都是抽象方法：

```
abstract class Person {
    public abstract void run();
    public abstract String getName();
}
```

就可以把该抽象类改写为接口：interface。

在Java中，使用interface可以声明一个接口：

```
interface Person {
    void run();
    String getName();
}
```

所谓interface，就是比抽象类还要抽象的纯抽象接口，因为它连字段都不能有。因为接口定义的所有方法默认都是public abstract的，所以这两个修饰符不需要写出来（写不写效果都一样）。

当一个具体的class去实现一个interface时，需要使用implements关键字。举个例子：

```
class Student implements Person {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println(this.name + " run");
    }

    @Override
    public String getName() {
        return this.name;
    }
}
```

我们知道，在Java中，一个类只能继承自另一个类，不能从多个类继承。但是，一个类可以实现多个interface，例如：

```
class Student implements Person, Hello { // 实现了两个interface
    ...
}
```

抽象类和接口对比

抽象类和接口的对比如下：

abstract	class	interface
继承	只能extends一个class	可以implements多个interface
字段	可以定义实例字段	不能定义实例字段
抽象方法	可以定义抽象方法	可以定义抽象方法
非抽象方法	可以定义非抽象方法	可以定义default方法

接口继承

一个interface可以继承自另一个interface。interface继承自interface使用extends，它相当于扩展了接口的方法。例如：

```
interface Hello {  
    void hello();  
}  
  
interface Person extends Hello {  
    void run();  
    String getName();  
}
```

此时，Person接口继承自Hello接口，因此，Person接口现在实际上有3个抽象方法签名，其中一个来自继承的Hello接口。

default方法

在接口中，可以定义default方法。例如，把Person接口的run()方法改为default方法：

```
public class Main {  
    public static void main(String[] args) {  
        Person p = new Student("Xiao Ming");  
        p.run();  
    }  
}  
  
interface Person {  
    String getName();  
    default void run() {  
        System.out.println(getName() + " run");  
    }  
}  
  
class Student implements Person {  
    private String name;  
  
    public Student(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
}
```

实现类可以不必覆写default方法。default方法的目的是，当我们需要给接口新增一个方法时，会涉及到修改全部子类。如果新增的是default方法，那么子类就不必全部修改，只需要在需要覆写的地方去覆写新增方法。

default方法和抽象类的普通方法是有所不同的。因为interface没有字段，default方法无法访问字段，而抽象类的普通方法可以访问实例字段。

小结

- Java的接口（interface）定义了纯抽象规范，一个类可以实现多个接口；
- 接口也是数据类型，适用于向上转型和向下转型；
- 接口的所有方法都是抽象方法，接口不能定义实例字段；
- 接口可以定义default方法（JDK>=1.8）。

静态字段和方法

静态字段

在一个class中定义的字段，我们称之为实例字段。实例字段的特点是，每个实例都有独立的字段，各个实例的同名字段互不影响。

还有一种字段，是用static修饰的字段，称为静态字段：static field。

实例字段在每个实例中都有自己的一个独立“空间”，但是静态字段只有一个共享“空间”，所有实例都会共享该字段。举个例子：

```
class Person {  
    public String name;  
    public int age;  
    // 定义静态字段number:  
    public static int number;  
}
```

看看下面例子：

```
public class Main {  
    public static void main(String[] args) {  
        Person ming = new Person("Xiao Ming", 12);  
        Person hong = new Person("Xiao Hong", 15);  
        ming.number = 88;  
        System.out.println(hong.number);  
        hong.number = 99;  
        System.out.println(ming.number);  
    }  
}  
  
class Person {  
    public String name;  
    public int age;  
  
    public static int number;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

对于静态字段，无论修改哪个实例的静态字段，效果都是一样的：所有实例的静态字段都被修改了，原因是静态字段并不属于实例。

虽然实例可以访问静态字段，但是它们指向的其实都是Person class的静态字段。所以，所有实例共享一个静态字段。

静态方法

有静态字段，就有静态方法。用static修饰的方法称为静态方法。

调用实例方法必须通过一个实例变量，而调用静态方法则不需要实例变量，通过类名就可以调用。静态方法类似其它编程语言的函数。例如：

```
public class Main {
    public static void main(String[] args) {
        Person.setNumber(99);
        System.out.println(Person.number);
    }
}

class Person {
    public static int number;

    public static void setNumber(int value) {
        number = value;
    }
}
```

因为静态方法属于class而不属于实例，因此，静态方法内部，无法访问this变量，也无法访问实例字段，它只能访问静态字段。

通过实例变量也可以调用静态方法，但这只是编译器自动帮我们把实例改写成类名而已。

通常情况下，通过实例变量访问静态字段和静态方法，会得到一个编译警告。

静态方法经常用于工具类。例如：

- `Arrays.sort()`
- `Math.random()`

静态方法也经常用于辅助方法。注意到Java程序的入口main()也是静态方法。

接口的静态字段

因为interface是一个纯抽象类，所以它不能定义实例字段。但是，interface是可以有静态字段的，并且静态字段必须为final类型：

```
public interface Person {
    public static final int MALE = 1;
    public static final int FEMALE = 2;
}
```

实际上，因为interface的字段只能是public static final类型，所以我们可以把这些修饰符都去掉，上述代码可以简写为：

```
public interface Person {
    // 编译器会自动加上public static final:
    int MALE = 1;
    int FEMALE = 2;
}
```

编译器会自动把该字段变为public static final类型。

小结

- 静态字段属于所有实例“共享”的字段，实际上是属于class的字段；
- 调用静态方法不需要实例，无法访问this，但可以访问静态字段和其他静态方法；
- 静态方法常用于工具类和辅助方法。

包

在前面的代码中，我们把类和接口命名为Person、Student、Hello等简单名字。

在现实中，如果小明写了一个Person类，小红也写了一个Person类，现在，小白既想用小明Person，也想用小红的Person，怎么办？

如果小军写了一个Arrays类，恰好JDK也自带了一个Arrays类，如何解决类名冲突？

在Java中，我们使用package来解决名字冲突。

Java定义了一种名字空间，称之为包：package。一个类总是属于某个包，类名（比如Person）只是一个简写，真正的完整类名是包名.类名。

例如：

小明的Person类存放在包ming下面，因此，完整类名是ming.Person；

小红的Person类存放在包hong下面，因此，完整类名是hong.Person；

小军的Arrays类存放在包mr.jun下面，因此，完整类名是mr.jun.Arrays；

JDK的Arrays类存放在包java.util下面，因此，完整类名是java.util.Arrays。

在定义class的时候，我们需要在第一行声明这个class属于哪个包。

小明的Person.java文件：

```
package ming; // 申明包名ming

public class Person {
}
```

在Java虚拟机执行的时候，JVM只看完整类名，因此，只要包名不同，类就不同。

包可以是多层结构，用.隔开。例如：java.util。

包作用域

位于同一个包的类，可以访问包作用域的字段和方法。不用public、protected、private修饰的字段和方法就是包作用域。例如，Person类定义在hello包下面：

```
package hello;

public class Person {
    // 包作用域：
    void hello() {
        System.out.println("Hello!");
    }
}
```

Main类也定义在hello包下面：


```
package hello;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.hello(); // 可以调用, 因为Main和Person在同一个包
    }
}
```

import

在一个class中, 我们总会引用其他的class。例如, 小明的ming.Person类, 如果要引用小军的mr.jun.Arrays类, 他有三种写法:

第一种, 直接写出完整类名, 例如:

```
// Person.java
package ming;

public class Person {
    public void run() {
        mr.jun.Arrays arrays = new mr.jun.Arrays();
    }
}
```

很显然, 每次写完整类名比较痛苦。

因此, 第二种写法是用import语句, 导入小军的Arrays, 然后写简单类名:

```
// Person.java
package ming;

// 导入完整类名:
import mr.jun.Arrays;

public class Person {
    public void run() {
        Arrays arrays = new Arrays();
    }
}
```

在写import的时候, 可以使用*, 表示把这个包下面的所有class都导入进来 (但不包括子包的class):

```
// Person.java
package ming;

// 导入mr.jun包的所有class:
import mr.jun.*;

public class Person {
    public void run() {
        Arrays arrays = new Arrays();
    }
}
```

我们一般不推荐这种写法，因为在导入了多个包后，很难看出Arrays类属于哪个包。

还有一种import static的语法，它可以导入可以导入一个类的静态字段和静态方法：

```
package main;

// 导入System类的所有静态字段和静态方法：
import static java.lang.System.*;

public class Main {
    public static void main(String[] args) {
        // 相当于调用System.out.println(...)
        out.println("Hello, world!");
    }
}
```

小结

- Java 内建的 package 机制是为了避免 class 命名冲突；
- JDK的核心类使用 java.lang 包，编译器会自动导入；
- JDK的其它常用类定义在
java.util.* , java.math.* , java.text.* , ；
- 包名推荐使用倒置的域名，例如 org.apache 。

内部类

在Java程序中，通常情况下，我们把不同的类组织在不同的包下面，对于一个包下面的类来说，它们是在同一层次，没有父子关系。

还有一种类，它被定义在另一个类的内部，所以称为内部类（Nested Class）。Java的内部类分为好几种，通常情况用得不多，但也需要了解它们是如何使用的。

Inner Class

如果一个类定义在另一个类的内部，这个类就是Inner Class：

```
class Outer {
    class Inner {
        // 定义了一个Inner Class
    }
}
```

上述定义的Outer是一个普通类，而Inner是一个Inner Class，它与普通类有个最大的不同，就是Inner Class的实例不能单独存在，必须依附于一个Outer Class的实例。示例代码如下：

```
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer("Nested"); // 实例化一个Outer
        Outer.Inner inner = outer.new Inner(); // 实例化一个Inner
        inner.hello();
    }
}

class Outer {
    private String name;

    Outer(String name) {
        this.name = name;
    }

    class Inner {
        void hello() {
            System.out.println("Hello, " + Outer.this.name);
        }
    }
}
```

观察上述代码，要实例化一个Inner，我们必须首先创建一个Outer的实例，然后，调用Outer实例的new来创建Inner实例：

```
Outer.Inner inner = outer.new Inner();
```

这是因为Inner Class除了有一个this指向它自己，还隐含地持有一个Outer Class实例，可以用Outer.this访问这个实例。所以，实例化一个Inner Class不能脱离Outer实例。

Anonymous Class

还有一种定义Inner Class的方法，它不需要在Outer Class中明确地定义这个Class，而是在方法内部，通过匿名类（Anonymous Class）来定义。示例代码如下：

```
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer("Nested");
        outer.asyncHello();
    }
}

class Outer {
    private String name;

    Outer(String name) {
        this.name = name;
    }

    void asyncHello() {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello, " + Outer.this.name);
            }
        };
        new Thread(r).start();
    }
}
```

观察asyncHello()方法，我们在方法内部实例化了一个Runnable。Runnable本身是接口，接口是不能实例化的，所以这里实际上是定义了一个实现了Runnable接口的匿名类，并且通过new实例化该匿名类，然后转型为Runnable。在定义匿名类的时候就必须实例化它，定义匿名类的写法如下：

```
Runnable r = new Runnable() {
    // 实现必要的抽象方法...
};
```

匿名类和Inner Class一样，可以访问Outer Class的private字段和方法。之所以我们要定义匿名类，是因为在这里我们通常不关心类名，比直接定义Inner Class可以少写很多代码。

Static Nested Class

最后一种内部类和Inner Class类似，但是使用static修饰，称为静态内部类（Static Nested Class）：

```
public class Main {
    public static void main(String[] args) {
        Outer.StaticNested sn = new Outer.StaticNested();
        sn.hello();
    }
}

class Outer {
    private static String NAME = "OUTER";

    private String name;

    Outer(String name) {
        this.name = name;
    }

    static class StaticNested {
        void hello() {
            System.out.println("Hello, " + Outer.NAME);
        }
    }
}
```

用static修饰的内部类和Inner Class有很大的不同，它不再依附于Outer的实例，而是一个完全独立的类，因此无法引用Outer.this，但它可以访问Outer的private静态字段和静态方法。如果把StaticNested移到Outer之外，就失去了访问private的权限。

小结

Java的内部类可分为Inner Class、Anonymous Class和Static Nested Class三种：

- Inner Class和Anonymous Class本质上是相同的，都必须依附于Outer Class的实例，即隐含地持有Outer.this实例，并拥有Outer Class的private访问权限；
- Static Nested Class是独立类，但拥有Outer Class的private访问权限。

classpath 和 jar

classpath

classpath是JVM用到的一个环境变量，它用来指示JVM如何搜索class。

因为Java是编译型语言，源码文件是.java，而编译后的.class文件才是真正可以被JVM执行的字节码。因此，JVM需要知道，如果要加载一个abc.xyz.Hello的类，应该去哪搜索对应的Hello.class文件。

所以，classpath就是一组目录的集合，它设置的搜索路径与操作系统相关。例如，在Windows系统上，用;分隔，带空格的目录用""括起来，在Linux系统上，用:分隔。

现在我们假设classpath是.;C:\work\project1\bin;C:\shared，当JVM在加载abc.xyz.Hello这个类时，会依次查找：

- <当前目录>\abc\xyz\Hello.class
- C:\work\project1\bin\abc\xyz\Hello.class
- C:\shared\abc\xyz\Hello.class

注意到 . 代表当前目录。如果JVM在某个路径下找到了对应的class文件，就不再往后继续搜索。如果所有路径下都没有找到，就报错。

classpath的设定方法有两种：

在系统环境变量中设置classpath环境变量，不推荐；

在启动JVM时设置classpath变量，推荐。

我们强烈不推荐在系统环境变量中设置classpath，那样会污染整个系统环境。在启动JVM时设置classpath才是推荐的做法。实际上就是给java命令传入-classpath或-cp参数：

```
java -classpath .;C:\work\project1\bin;C:\shared abc.xyz.Hello
```

或者使用-cp的简写：

```
java -cp .;C:\work\project1\bin;C:\shared abc.xyz.Hello
```

jar包

如果有很多.class文件，散落在各层目录中，肯定不便于管理。如果能把目录打一个包，变成一个文件，就方便多了。

jar包就是用来干这个事的，它可以把package组织的目录层级，以及各个目录下的所有文件（包括.class文件和其他文件）都打成一个jar文件，这样一来，无论是备份，还是发给客户，就简单多了。

jar包实际上就是一个zip格式的压缩文件，而jar包相当于目录。如果我们要执行一个jar包的class，就可以把jar包放到classpath中：

```
java -cp ./hello.jar abc.xyz.Hello
```

这样JVM会自动在hello.jar文件里去搜索某个类。

jar包还可以包含一个特殊的/META-INF/MANIFEST.MF文件，MANIFEST.MF是纯文本，可以指定Main-Class和其它信息。JVM会自动读取这个MANIFEST.MF文件，如果存在Main-Class，我们就不必在命令行指定启动的类名，而是用更方便的命令：

```
java -jar hello.jar
```

jar包还可以包含其它jar包，这个时候，就需要在MANIFEST.MF文件里配置classpath了。

在大型项目中，不可能手动编写MANIFEST.MF文件，再手动创建zip包。Java社区提供了大量的开源构建工具，例如 [Maven](#)，可以非常方便地创建jar包。

小结

- JVM通过环境变量classpath决定搜索class的路径和顺序；
- 不推荐设置系统环境变量classpath，始终建议通过-cp命令传入；
- jar包相当于目录，可以包含很多.class文件，方便下载和使用；
- MANIFEST.MF文件可以提供jar包的信息，如Main-Class，这样可以直接运行jar包。