

01 Java快速入门

本章的主要内容是快速掌握Java程序的基础知识，了解并使用变量和各种数据类型，介绍基本的程序流程控制语句。

Java简介

Java最早是由SUN公司（已被Oracle收购）的詹姆斯·高斯林（高司令，人称Java之父）在上个世纪90年代初开发的一种编程语言，最初被命名为Oak，目标是针对小型家电设备的嵌入式应用，结果市场没啥反响。谁料到互联网的崛起，让Oak重新焕发了生机，于是SUN公司改造了Oak，在1995年以Java的名称正式发布，原因是Oak已经被人注册了，因此SUN注册了Java这个商标。随着互联网的高速发展，Java逐渐成为最重要的网络编程语言。

Java介于编译型语言和解释型语言之间。编译型语言如C、C++，代码是直接编译成机器码执行，但是不同的平台（x86、ARM等）CPU的指令集不同，因此，需要编译出每一种平台的对应机器码。解释型语言如Python、Ruby没有这个问题，可以由解释器直接加载源码然后运行，代价是运行效率太低。而Java是将代码编译成一种“字节码”，它类似于抽象的CPU指令，然后，针对不同平台编写虚拟机，不同平台的虚拟机负责加载字节码并执行，这样就实现了“一次编写，到处运行”的效果。当然，这是针对Java开发者而言。对于虚拟机，需要为每个平台分别开发。为了保证不同平台、不同公司开发的虚拟机都能正确执行Java字节码，SUN公司制定了一系列的Java虚拟机规范。从实践的角度看，JVM的兼容性做得非常好，低版本的Java字节码完全可以正常运行在高版本的JVM上。

随着Java的发展，SUN给Java又分出了三个不同版本：

- Java SE: Standard Edition
- Java EE: Enterprise Edition
- Java ME: Micro Edition

这三者之间有啥关系呢？



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”。

毫无疑问，Java SE是整个Java平台的核心，而Java EE是进一步学习Web应用所必须的。我们熟悉的Spring等框架都是Java EE开源生态系统的一部分。不幸的是，Java ME从来没有真正流行起来，反而是Android开发成为了移动平台的标准之一，因此，没有特殊需求，不建议学习Java ME。

因此我们推荐的Java学习路线图如下：

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；
4. 如果想要学习移动开发，那么就深入Android平台，掌握Android App开发。

无论怎么选择，Java SE的核心技术是基础，这个教程的目的就是让你完全精通Java SE！

Java版本

从1995年发布1.0版本开始，到目前为止，最新的Java版本是Java 13:

时间	版本
1995	1.0
1998	1.2
2000	1.3
2002	1.4
2004	1.5 / 5.0
2005	1.6 / 6.0
2011	1.7 / 7.0
2014	1.8 / 8.0
2017/9	1.9 / 9.0
2018/3	10
2018/9	11
2019/3	12
2019/9	13

本教程使用的Java版本是最新版的**Java 13**。

名词解释

初学者学Java，经常听到JDK、JRE这些名词，它们到底是啥？

- JDK: Java Development Kit
- JRE: Java Runtime Environment

简单地说，JRE就是运行Java字节码的虚拟机。但是，如果只有Java源码，要编译成Java字节码，就需要JDK，因为JDK除了包含JRE，还提供了编译器、调试器等开发工具。

二者关系如下：



要学习Java开发，当然需要安装JDK了。

那JSR、JCP.....又是啥？

- JSR规范：Java Specification Request
- JCP组织：Java Community Process

为了保证Java语言的规范性，SUN公司搞了一个JSR规范，凡是想给Java平台加一个功能，比如说访问数据库的功能，大家要先创建一个JSR规范，定义好接口，这样，各个数据库厂商都按照规范写出Java驱动程序，开发者就不用担心自己写的数据库代码在MySQL上能跑，却不能跑在PostgreSQL上。

所以JSR是一系列的规范，从JVM的内存模型到Web程序接口，全部都标准化了。而负责审核JSR的组织就是JCP。

一个JSR规范发布时，为了让大家有个参考，还要同时发布一个“参考实现”，以及一个“兼容性测试套件”：

- RI：Reference Implementation
- TCK：Technology Compatibility Kit

比如有人提议要搞一个基于Java开发的消息服务器，这个提议很好啊，但是光有提议还不行，得贴出真正能跑的代码，这就是RI。如果有其他人也想开发这样一个消息服务器，如何保证这些消息服务器对开发者来说接口、功能都是相同的？所以还得提供TCK。

通常来说，RI只是一个“能跑”的正确的代码，它不追求速度，所以，如果真要选择一个Java的消息服务器，一般是没人用RI的，大家都会选择一个有竞争力的商用或开源产品。

参考：Java消息服务JMS的JSR：<https://jcp.org/en/jsr/detail?id=914>

安装JDK

因为Java程序必须运行在JVM之上，所以，我们第一件事情就是安装JDK。

搜索JDK 13，确保从Oracle的官网下载最新的稳定版JDK：



找到Java SE 13.x的下载链接，下载安装即可。

设置环境变量

安装完JDK后，需要设置一个 `JAVA_HOME` 的环境变量，它指向JDK的安装目录。在Windows下，它是安装目录，类似：

```
C:\Program Files\Java\jdk-13
```

在Mac下，它在 `~/ .bash_profile` 里，它是：

```
export JAVA_HOME="/usr/libexec/java_home -v 13"
```

然后，把 `JAVA_HOME` 的 `bin` 目录附加到系统环境变量 `PATH` 上。在Windows下，它长这样：

```
Path=%JAVA_HOME%\bin;<现有的其他路径>
```

在Mac下，它在 `~/ .bash_profile` 里，长这样：

```
export PATH=$JAVA_HOME/bin:$PATH
```

把 `JAVA_HOME` 的 `bin` 目录添加到 `PATH` 中是为了在任意文件夹下都可以运行 `java`。打开命令提示符窗口，输入命令 `java -version`，如果一切正常，你会看到如下输出：

```
Command Prompt
```

```
Microsoft Windows [Version 10.0.0]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\> java -version
java version "13" ...
Java(TM) SE Runtime Environment
Java HotSpot(TM) 64-Bit Server VM

C:\>
```

如果你看到的版本号不是 **13**，而是 **12**、**1.8** 之类，说明系统存在多个JDK，且默认JDK不是JDK 13，需要把JDK 13提到 **PATH** 前面。

如果你得到一个错误输出：

```
Command Prompt
Microsoft Windows [Version 10.0.0]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\> java -version
'java' is not recognized as an internal or external command,
operable program or batch file.

C:\>
```

这是因为系统无法找到Java虚拟机的程序 **java.exe**，需要检查 **JAVA_HOME** 和 **PATH** 的配置。

可以参考[如何设置或更改PATH系统变量](#)。

JDK

细心的童鞋还可以在 **JAVA_HOME** 的 **bin** 目录下找到很多可执行文件：

- **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

我们来编写第一个Java程序。

打开文本编辑器，输入以下代码：

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

在一个Java程序中，你总能找到一个类似：

```
public class Hello {  
    ...  
}
```

的定义，这个定义被称为class（类），这里的类名是Hello，大小写敏感，class用来定义一个类，public表示这个类是公开的，public、class都是Java的关键字，必须小写，Hello是类的名字，按照习惯，首字母H要大写。而花括号{}中间则是类的定义。

注意到类的定义中，我们定义了一个名为main的方法：

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

方法是可执行的代码块，一个方法除了方法名main，还有用()括起来的方法参数，这里的main方法有一个参数，参数类型是String[]，参数名是args，public、static用来修饰方法，这里表示它是一个公开的静态方法，void是方法的返回类型，而花括号{}中间的就是方法的代码。

方法的代码每一行用;结束，这里只有一行代码，就是：

```
System.out.println("Hello, world!");
```

它用来打印一个字符串到屏幕上。

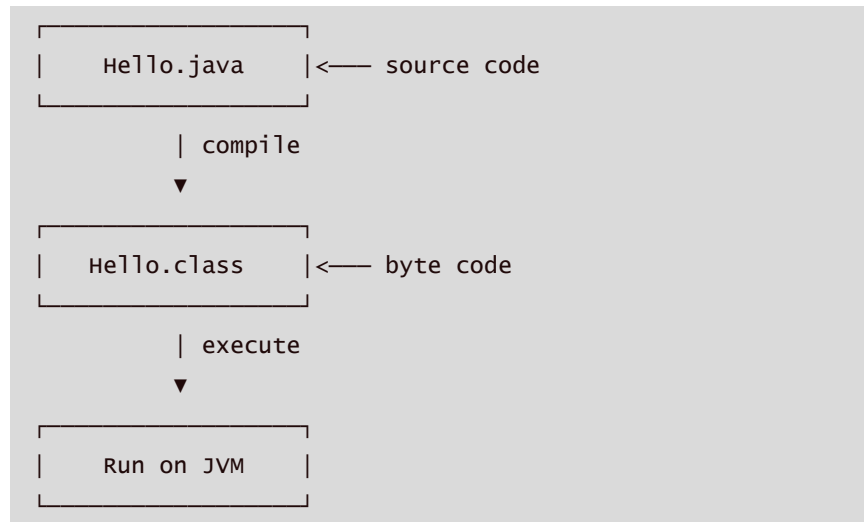
Java规定，某个类定义的public static void main(String[] args)是Java程序的固定入口方法，因此，Java程序总是从main方法开始执行。

注意到Java源码的缩进不是必须的，但是用缩进后，格式好看，很容易看出代码块的开始和结束，缩进一般是4个空格或者一个tab。

最后，当我们把代码保存为文件时，文件名必须是Hello.java，而且文件名也要注意大小写，因为要和我们定义的类名Hello完全保持一致。

如何运行Java程序

Java源码本质上是一个文本文件，我们需要先用 `javac` 把 `Hello.java` 编译成字节码文件 `Hello.class`，然后，用 `java` 命令执行这个字节码文件：



因此，可执行文件 `javac` 是编译器，而可执行文件 `java` 就是虚拟机。

第一步，在保存 `Hello.java` 的目录下执行命令 `javac Hello.java`：

```
$ javac Hello.java
```

如果源代码无误，上述命令不会有任何输出，而当前目录下会产生一个 `Hello.class` 文件：

```
$ ls
Hello.class Hello.java
```

第二步，执行 `Hello.class`，使用命令 `java Hello`：

```
$ java Hello
Hello, world!
```

注意：给虚拟机传递的参数 `Hello` 是我们定义类名，虚拟机自动查找对应的 `class` 文件并执行。

有一些童鞋可能知道，直接运行 `java Hello.java` 也是可以的：

```
$ java Hello.java
Hello, world!
```

这是Java 11新增的一个功能，它可以直接运行一个单文件源码！

需要注意的是，在实际项目中，单个不依赖第三方库的Java源码是非常罕见的，所以，绝大多数情况下，我们无法直接运行一个Java源码文件，原因是它需要依赖其他的库。

小结

- 一个Java源码只能定义一个 `public` 类型的class，并且class名称和文件名要完全一致；
- 使用 `javac` 可以将 `.java` 源码编译成 `.class` 字节码；
- 使用 `java` 可以运行一个已编译的Java程序，参数是类名。

Java代码助手

Java代码运行助手可以让你在线输入Java代码，然后通过本机运行的一个Java程序来执行代码。原理如下：

- 在网页输入代码；
- 点击Run按钮，代码被发送到本机正在运行的Java代码运行助手；
- Java代码运行助手将代码保存为临时文件，然后调用Java虚拟机执行代码；
- 网页显示代码执行结果：



下载

点击右键，目标另存为：[LearnJava.java](#)

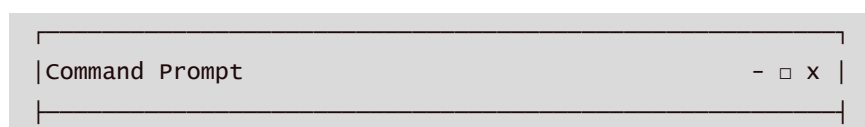
运行

在存放 `LearnJava.java` 的目录下运行命令：

```
C:\Users\michael\Downloads> java LearnJava.java
```

如果看到 `Ready for Java code on port 39193...` 表示运行成功。

不要关闭命令行窗口，最小化放到后台运行即可：




```
|Microsoft windows [Version 10.0.0] |
|(c) 2015 Microsoft Corporation. All rights reserved. |
| |
|C:\Users\michael\Downloads> java LearnJava.java |
|Ready for Java code on port 39193... |
|Press Ctrl + C to exit... |
| |
| |
| |
| |
| |
```

试试效果

需要支持HTML5的浏览器：

- IE >= 9
- Firefox
- Chrome
- Safari

```
// 测试代码
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
}
```

使用IDE

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

使用IDE的好处在于按，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，流行的用于Java开发的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免费版和付费版。

NetBeans

NetBeans是最早由SUN开发的开源IDE，由于使用人数较少，目前已不再流行。

使用Eclipse

你可以使用任何IDE进行Java学习和开发。我们不讨论任何关于IDE的优劣，本教程使用Eclipse作为开发演示环境，原因在于：

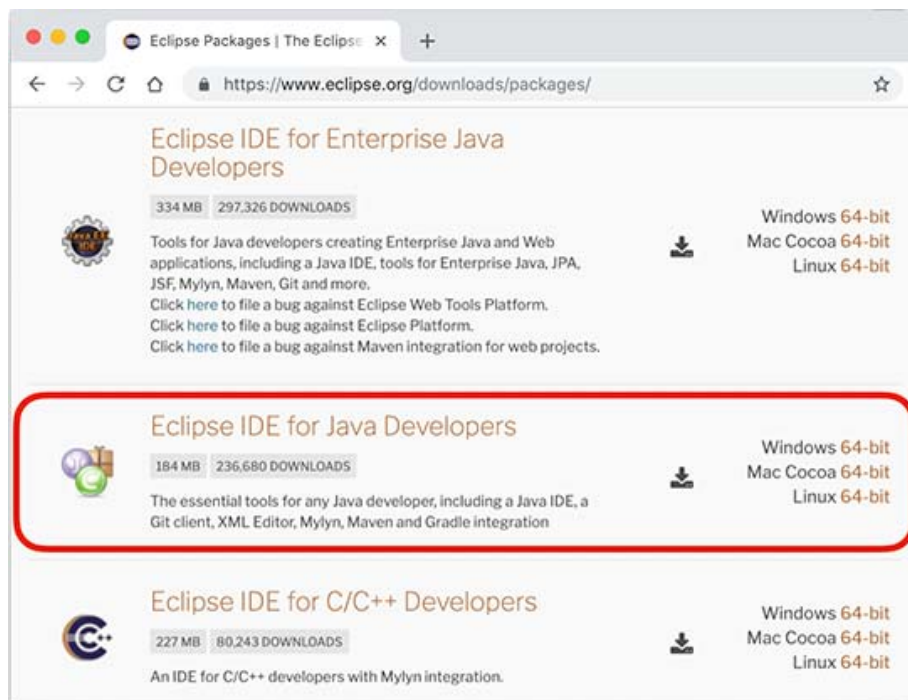
- 完全免费使用；
- 所有功能完全满足Java开发需求。

如果你使用Eclipse作为开发环境来学习本教程，还可以获得一个额外的好处：教程提供了一个基于Eclipse的**IDE练习插件**，可以直接在线导入Java工程！

安装Eclipse

Eclipse的发行版提供了预打包的开发环境，包括Java、JavaEE、C++、PHP、Rust等。从[这里](#)下载：

我们需要下载的版本是Eclipse IDE for Java Developers：



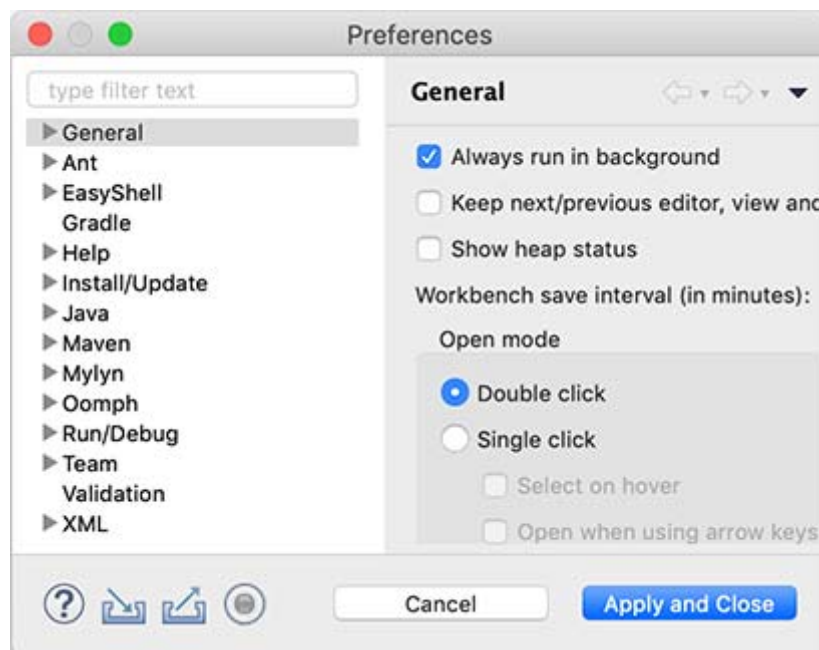
根据操作系统是Windows、Mac还是Linux，从右边选择对应的下载链接。

注意：教程从头到尾并不需要用到Enterprise Java的功能，所以不需要下载Eclipse IDE for Enterprise Java Developers

设置Eclipse

下载并安装完成后，我们启动Eclipse，对IDE环境做一个基本设置：

选择菜单“Eclipse/Window”-“Preferences”，打开配置对话框：



我们需要调整以下设置项：

General > Editors > Text Editors

钩上“Show line numbers”，这样编辑器会显示行号；

General > Workspace

钩上“Refresh using native hooks or polling”，这样Eclipse会自动刷新文件夹的改动；

对于“Text file encoding”，如果Default不是UTF-8，一定要改为“Other: UTF-8”，所有文本文件均使用UTF-8编码；

对于“New text file line delimiter”，建议使用Unix，即换行符使用\n而不是Windows的\r\n。

Java > Compiler

将“Compiler compliance level”设置为13，本教程的所有代码均使用Java 13的语法，并且编译到Java 13的版本。

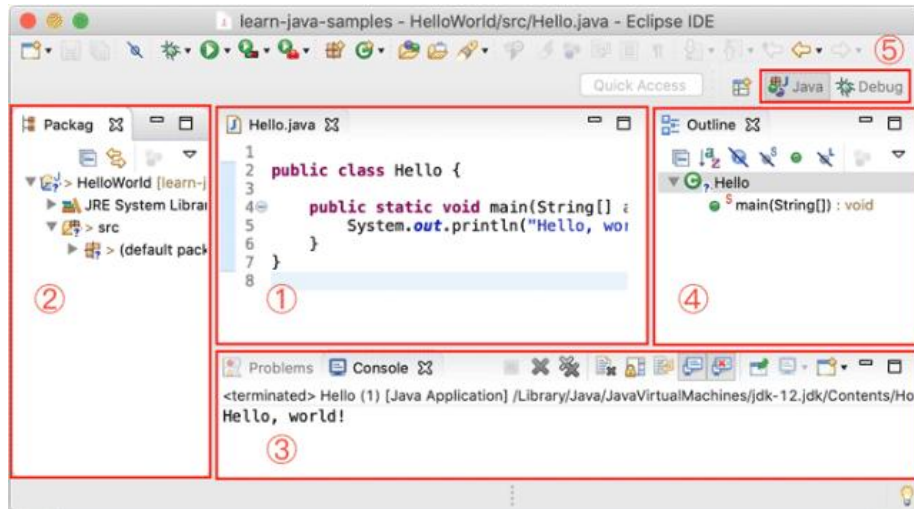
去掉“Use default compliance settings”并钩上“Enable preview features for Java 13”，这样我们就可以使用Java 13的预览功能。

Java > Installed JREs

在Installed JREs中应该看到Java SE 13，如果还有其他的JRE，可以删除，以确保Java SE 13是默认的JRE。

Eclipse IDE结构

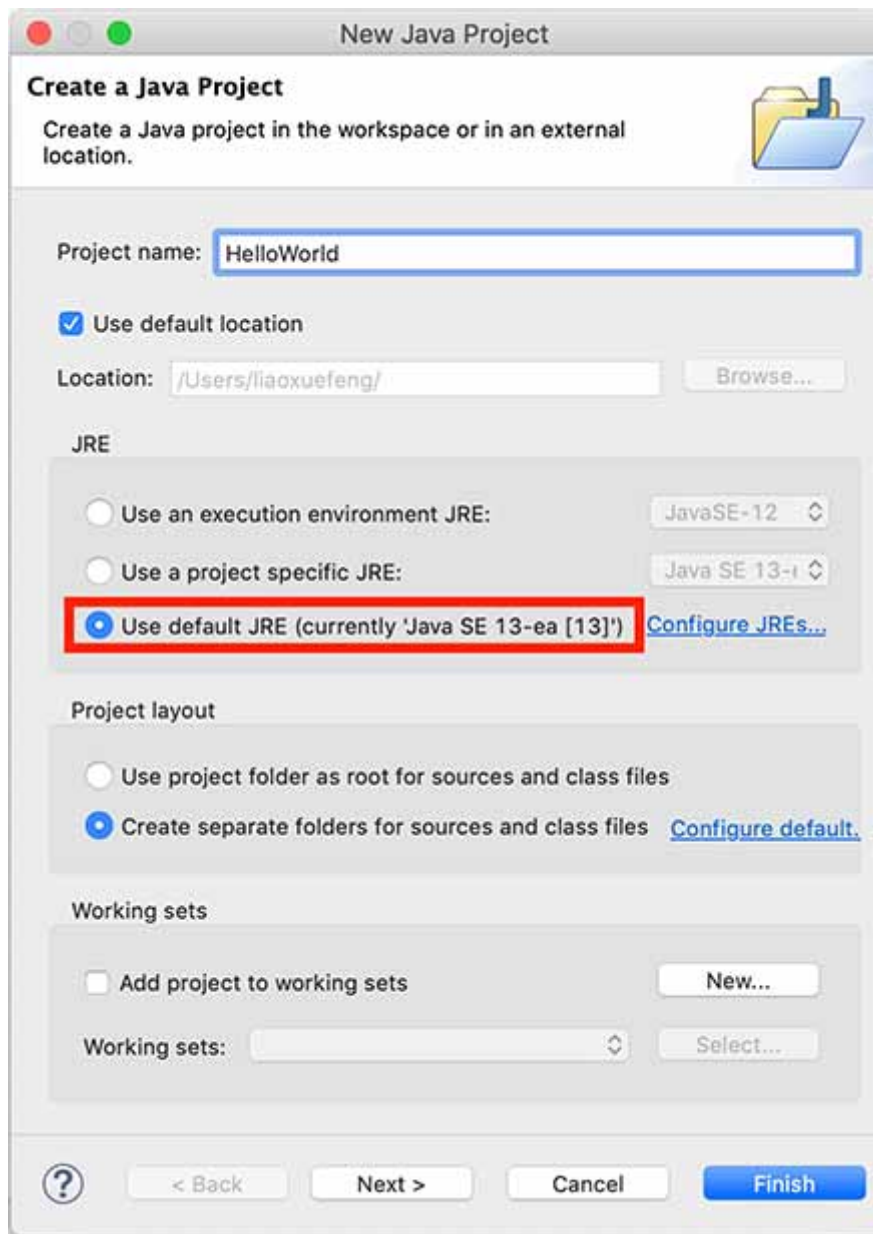
打开Eclipse后，整个IDE由若干个区域组成：



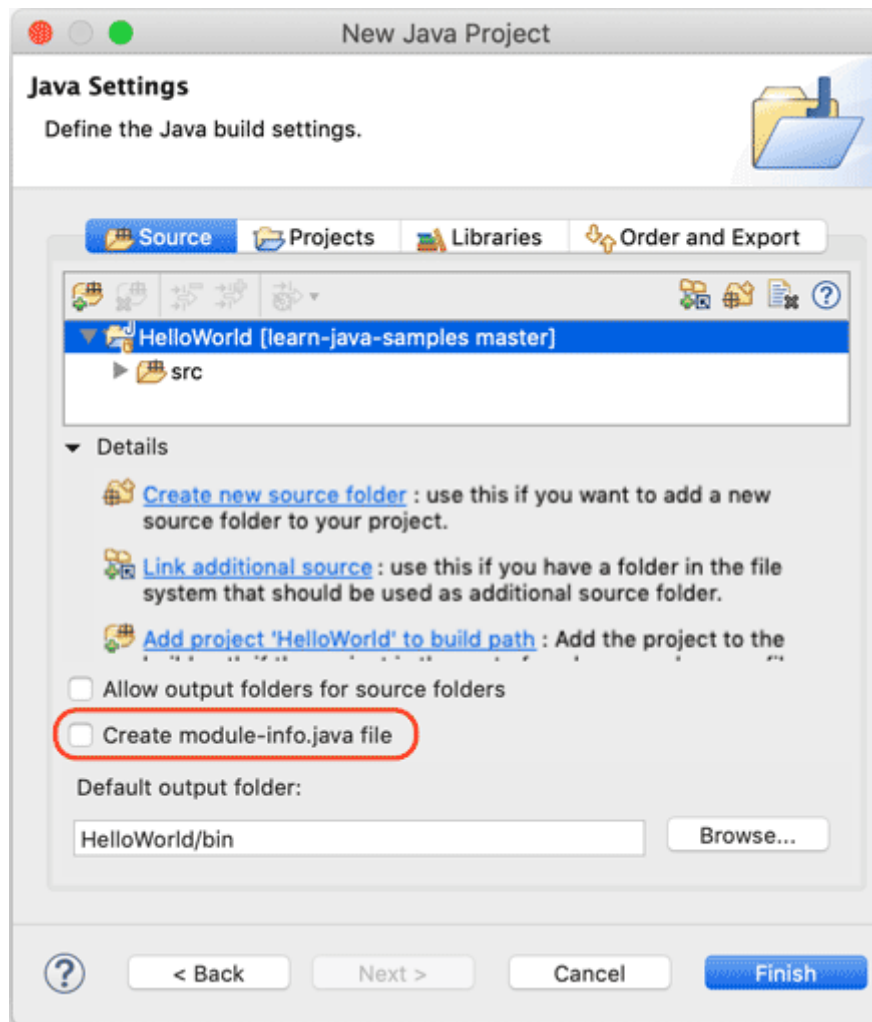
- 中间可编辑的文本区（见1）是编辑器，用于编辑源码；
- 分布在左右和下方的是视图：
 - Package Explorer（见2）是Java项目的视图
 - Console（见3）是命令行输出视图
 - Outline（见4）是当前正在编辑的Java源码的结构视图
- 视图可以任意组合，然后把一组视图定义成一个Perspective（见5），Eclipse预定义了Java、Debug等几个Perspective，用于快速切换。

新建Java项目

在Eclipse菜单选择“File”-“New”-“Java Project”，填入HelloWorld，JRE选择Java SE 13：



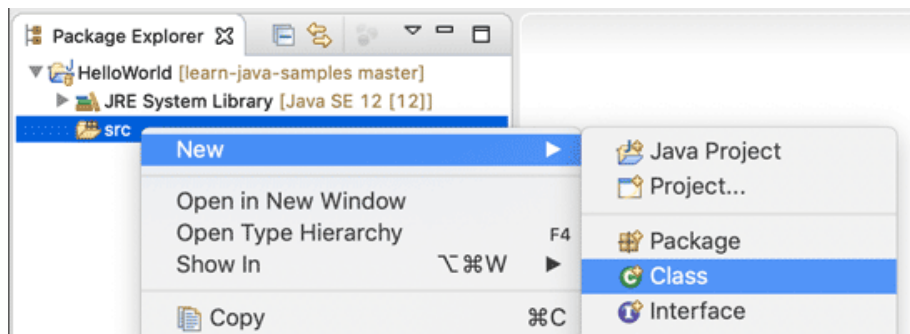
暂时不要勾选“Create module-info.java file”，因为模块化机制我们后面才会讲到：



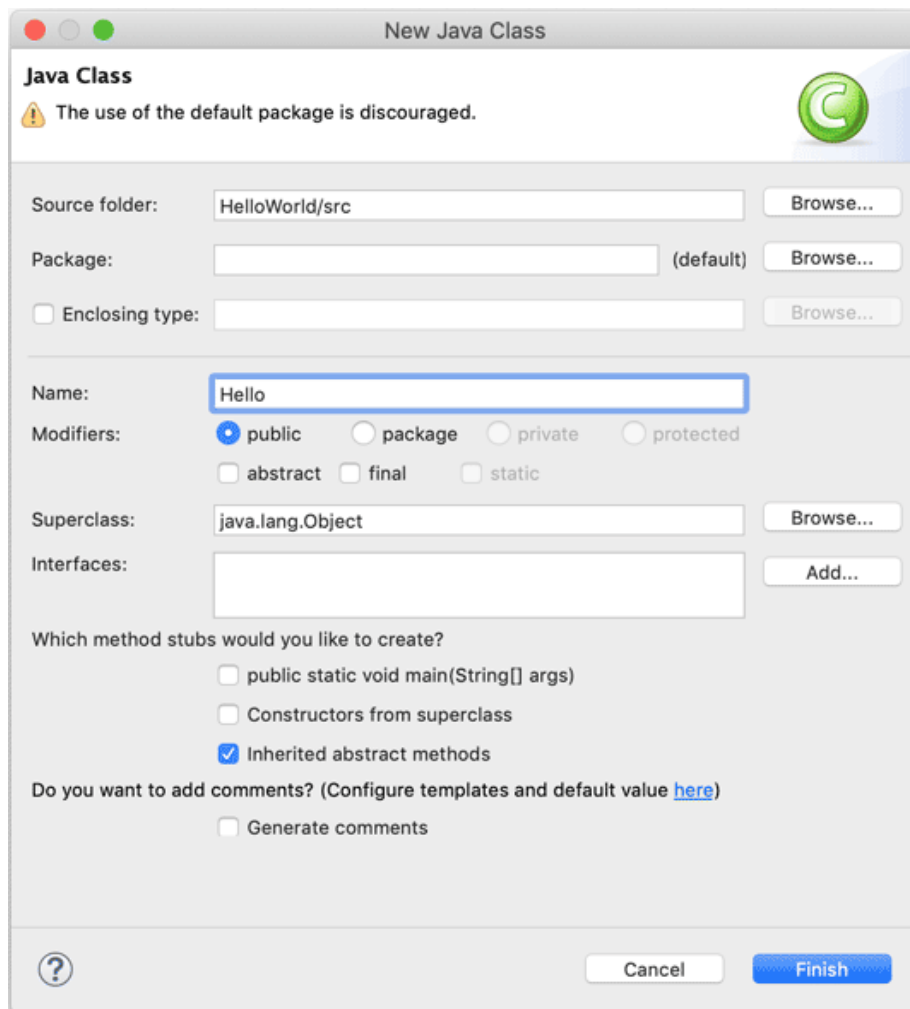
点击“Finish”就成功创建了一个名为HelloWorld的Java工程。

新建Java文件并运行

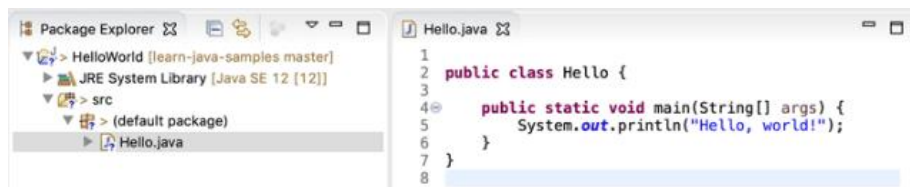
展开HelloWorld工程，选中源码目录src，点击右键，在弹出菜单中选择“New”-“Class”：



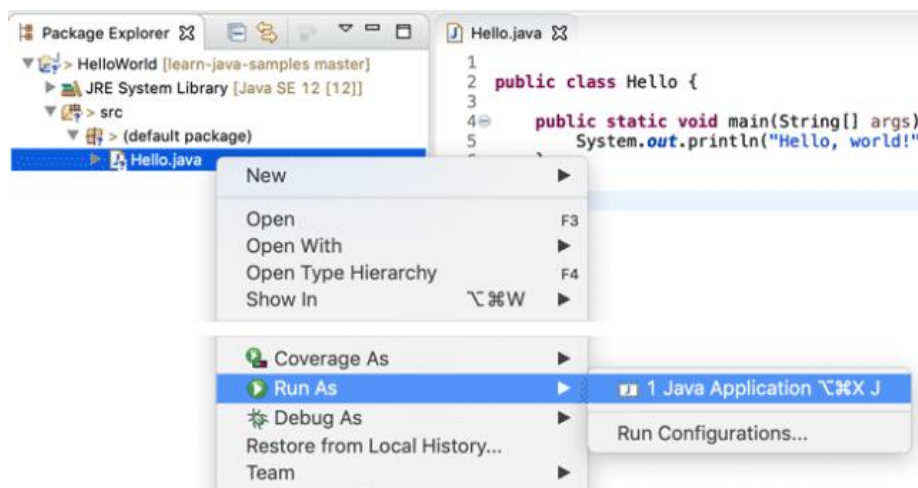
在弹出的对话框中，Name一栏填入Hello：



点击“Finish”，就自动在 `src` 目录下创建了一个名为 `Hello.java` 的源文件。我们双击打开这个源文件，填上代码：



保存，然后选中文件 `hello.java`，点击右键，在弹出的菜单中选中“Run As...”-“Java Application”：



在 `Console` 窗口中就可以看到运行结果：



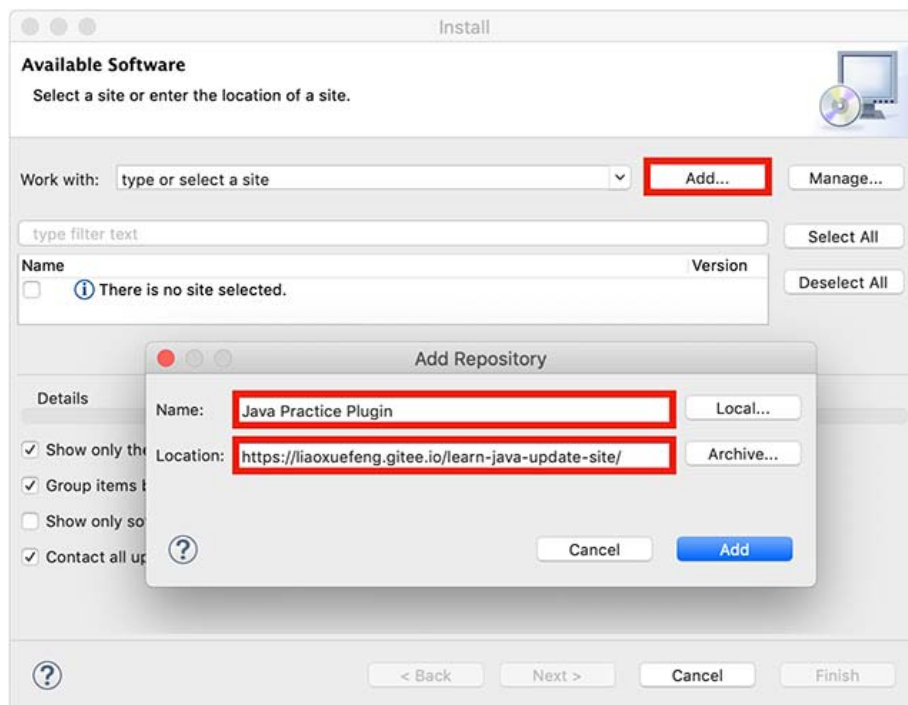
如果没有在主界面中看到 **Console** 窗口，请选中菜单“Window”-“Show View”-“Console”，即可显示。

使用IDE练习插件

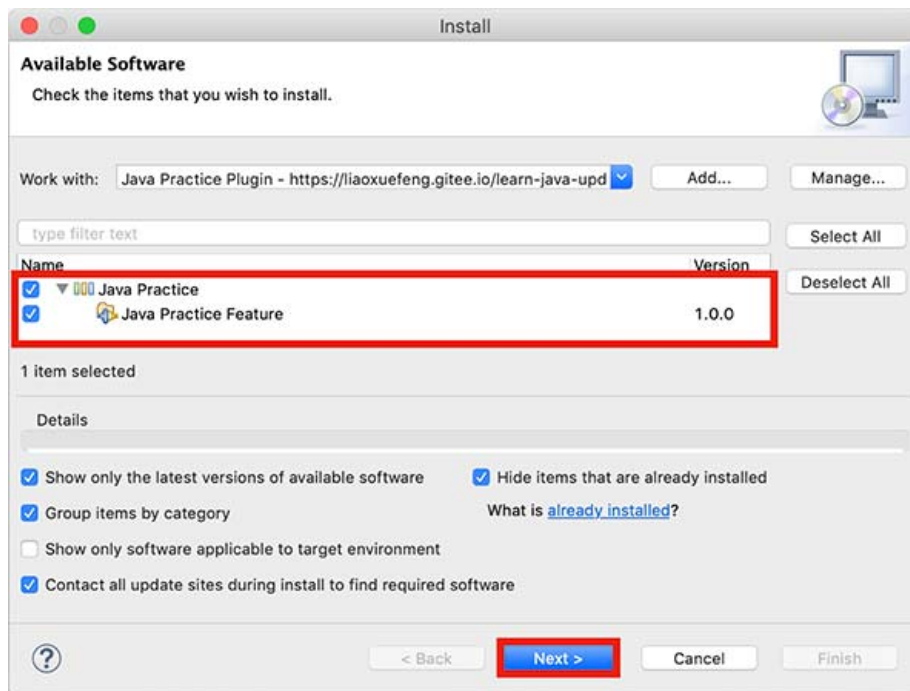
本教程提供一个Eclipse IDE的练习插件，可以非常方便地下载练习代码。

安装IDE练习插件

启动Eclipse，选择菜单“Help”-“Install New Software...”，在打开的对话框中：



点击“Add”，对Name填写一个任意的名称，例如“Java Practice Plugin”，对于Location，填入 <https://liao xuefeng.gitee.io/learn-java-update-site/>，然后点击“Add”添加：



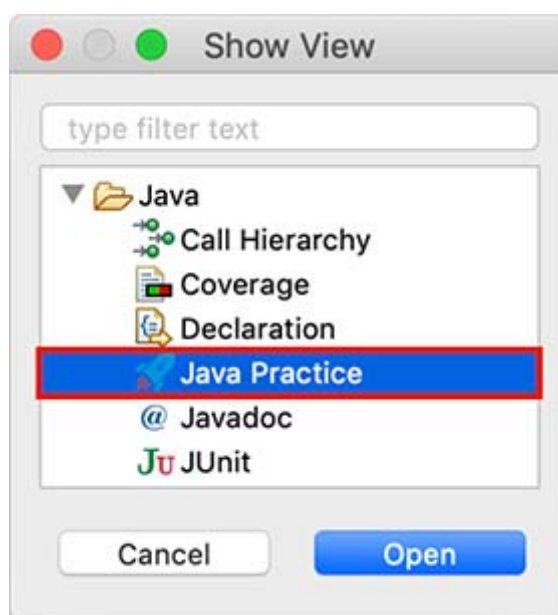
在列表中选“Java Practice Feature”，然后点击“Next”安装。

在安装过程中，由于插件代码没有数字签名，所以会弹出一个警告：

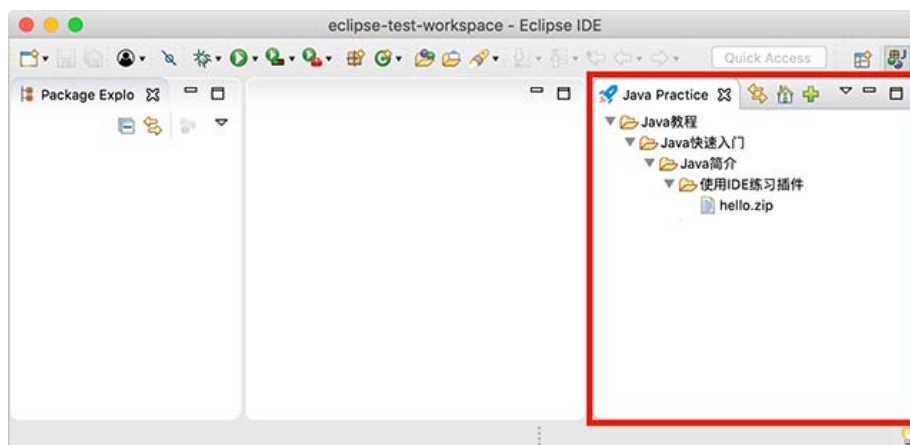


选择“Install anyway”继续安装，安装成功后，根据提示重启Eclipse即可。

重启Eclipse后，选择菜单“Window”-“Show View”-“Other...”：

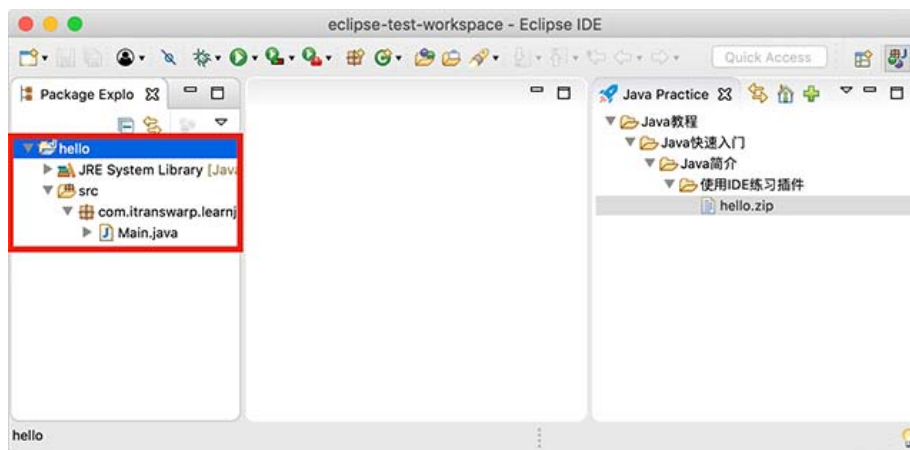


在弹出的对话框中选择“Java”-“Java Practice”，然后点击“Open”，即可在Eclipse中看到Java Practice插件：



导入练习

在“Java Practice”面板中，双击 **hello.zip**，按照提示导入工程，即可直接下载并导入到Eclipse中：



是不是非常方便？

Java程序基础

本节我们将介绍Java程序的基础知识，包括：

- Java程序基本结构
- 变量和数据类型
- 整数运算
- 浮点数运算
- 布尔运算
- 字符和字符串
- 数组类型

Java程序基本结构

我们先剖析一个完整的Java程序，它的基本结构是什么：

```

/**
 * 可以用来自动创建文档的注释
 */
public class Hello {
    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

```

因为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，这里定义的 `class` 名字就是 `Hello`：

```

public class Hello { // 类名是Hello
    // ...
} // class定义结束

```

类名要求：

- 类名必须以英文字母开头，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：

- Hello
- NoteBook
- VRPlayer

不好的类命名：

- hello
- Good123
- Note_Book
- _World

注意到 `public` 是访问修饰符，表示该 `class` 是公开的。

不写 `public`，也能正确编译，但是这个类将无法从命令行执行。

在 `class` 内部，可以定义若干方法（method）：

```

public class Hello {
    public static void main(String[] args) { // 方法名是
main
        // 方法代码...
    } // 方法定义结束
}

```

方法定义了一组执行语句，方法内部的代码将会被依次顺序执行。

这里的方法名是 `main`，返回值是 `void`，表示没有任何返回值。

我们注意到 `public` 除了可以修饰 `class` 外，也可以修饰方法。而关键字 `static` 是另一个修饰符，它表示静态方法，后面我们会讲解方法的类型，目前，我们只需要知道，Java 入口程序规定的方法必须是静态方法，方法名必须为 `main`，括号内的参数必须是 `String` 数组。

方法名也有命名规则，命名和 `class` 一样，但是首字母小写：

好的方法命名：

- `main`
- `goodMorning`
- `playVR`

不好的方法命名：

- `Main`
- `good123`
- `good_morning`
- `_playVR`

在方法内部，语句才是真正的执行代码。Java 的每一行语句必须以分号结束：

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

在Java程序中，注释是一种给人阅读的文本，不是程序的一部分，所以编译器会自动忽略注释。

Java有3种注释，第一种是单行注释，以双斜线开头，直到这一行的结尾结束：

```
// 这是注释...
```

而多行注释以 `/*` 星号开头，以 `*/` 结束，可以有多样：

```
/*  
这是注释  
blablabla...  
这也是注释  
*/
```

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：

```

/**
 * 可以用来自动创建文档的注释
 *
 * @author liaoxuefeng
 */
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}

```

这种特殊的多行注释需要写在类和方法的定义处，可以用于自动创建文档。

Java程序对格式没有明确的要求，多几个空格或者回车不影响程序的正确性，但是我们要养成良好的编程习惯，注意遵守Java社区约定的编码格式。

那约定的编码格式有哪些要求呢？其实我们在前面介绍的Eclipse IDE提供了快捷键 **Ctrl+Shift+F**（macOS是 **⌘+⇧+F**）帮助我们快速格式化代码的功能，Eclipse就是按照约定的编码格式对代码进行格式化的，所以只需要看看格式化后的代码长啥样就行了。具体的代码格式要求可以在Eclipse的设置中 **Java - Code Style** 查看。

变量和数据类型

变量

什么是变量？

变量就是初中数学的代数的概念，例如一个简单的方程， x ， y 都是变量：

$$y = x^2 + 1$$

在Java中，变量分为两种：基本类型的变量和引用类型的变量。

我们先讨论基本类型的变量。

在Java中，变量必须先定义后使用，在定义变量的时候，可以给它一个初始值。例如：

```
int x = 1;
```

上述语句定义了一个整型 `int` 类型的变量，名称为 `x`，初始值为 `1`。

不写初始值，就相当于给它指定了默认值。默认值总是 `0`。

来看一个完整的定义变量，然后打印变量值的例子：

```
public class Main {
    public static void main(String[] args) {
        int x = 100; // 定义int类型变量x，并赋予初始值100
        System.out.println(x); // 打印该变量的值
    }
}
```

变量的一个重要特点是可以重新赋值。例如，对变量`x`，先赋值`100`，再赋值`200`，观察两次打印的结果：

```
public class Main {
    public static void main(String[] args) {
        int x = 100; // 定义int类型变量x，并赋予初始值100
        System.out.println(x); // 打印该变量的值，观察是否为
100
        x = 200; // 重新赋值为200
        System.out.println(x); // 打印该变量的值，观察是否为
200
    }
}
```

注意到第一次定义变量`x`的时候，需要指定变量类型`int`，因此使用语句`int x = 100;`。而第二次重新赋值的时候，变量`x`已经存在了，不能再重复定义，因此不能指定变量类型`int`，必须使用语句`x = 200;`。

变量不但可以重新赋值，还可以赋值给其他变量。让我们来看一个例子：

```
public class Main {
    public static void main(String[] args) {
        int n = 100; // 定义变量n，同时赋值为100
        System.out.println("n = " + n); // 打印n的值

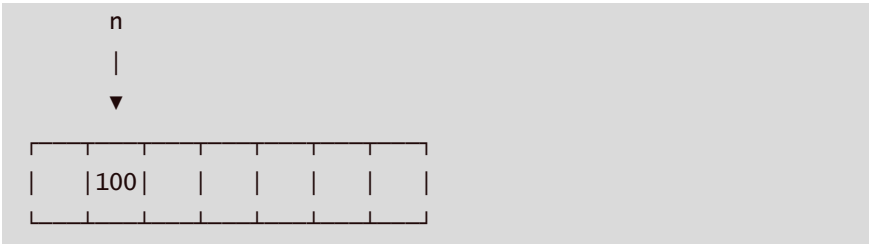
        n = 200; // 变量n赋值为200
        System.out.println("n = " + n); // 打印n的值

        int x = n; // 变量x赋值为n（n的值为200，因此赋值后x的值
也是200）
        System.out.println("x = " + x); // 打印x的值

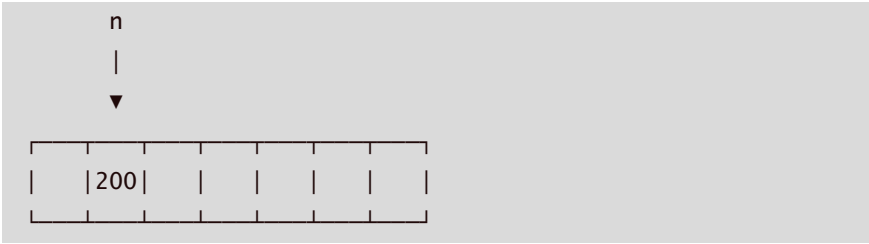
        x = x + 100; // 变量x赋值为x+100（x的值为200，因此赋值
后x的值是200+100=300）
        System.out.println("x = " + x); // 打印x的值
        System.out.println("n = " + n); // 再次打印n的值，n
应该是200还是300？
    }
}
```

我们一行一行地分析代码执行流程：

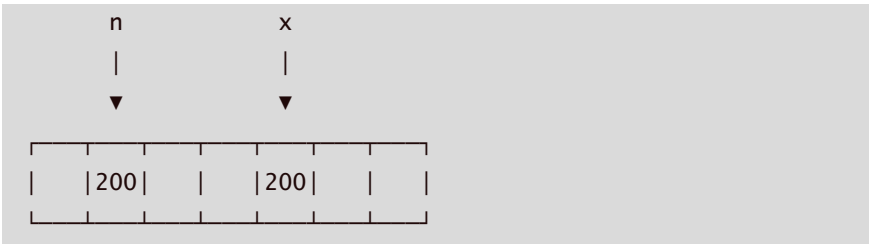
执行 `int n = 100;`，该语句定义了变量 `n`，同时赋值为 `100`，因此，JVM在内存中为变量 `n` 分配一个“存储单元”，填入值 `100`：



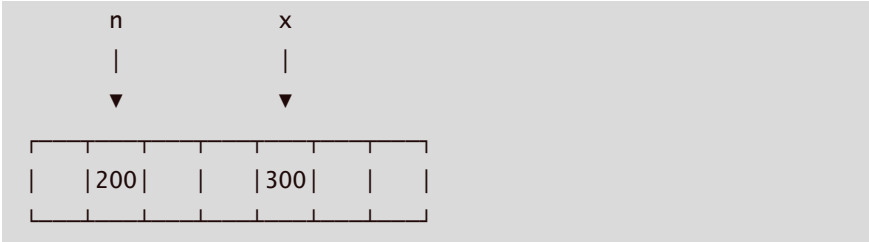
执行 `n = 200;` 时，JVM把 `200` 写入变量 `n` 的存储单元，因此，原有的值被覆盖，现在 `n` 的值为 `200`：



执行 `int x = n;` 时，定义了一个新的变量 `x`，同时对 `x` 赋值，因此，JVM需要新分配一个存储单元给变量 `x`，并写入和变量 `n` 一样的值，结果是变量 `x` 的值也变为 `200`：



执行 `x = x + 100;` 时，JVM首先计算等式右边的值 `x + 100`，结果为 `300`（因为此刻 `x` 的值为 `200`），然后，将结果 `300` 写入 `x` 的存储单元，因此，变量 `x` 最终的值变为 `300`：



可见，变量可以反复赋值。注意，等号 `=` 是赋值语句，不是数学意义上的相等，否则无法解释 `x = x + 100`。

基本数据类型

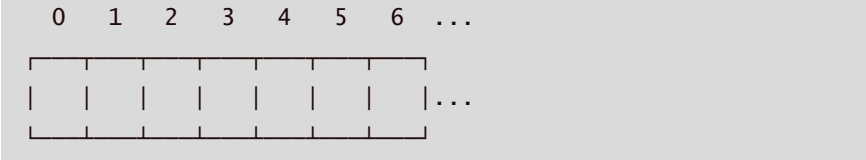
基本数据类型是CPU可以直接进行运算的类型。Java定义了以下几种基本数据类型：

- 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

Java定义的这些基本数据类型有什么区别呢？要了解这些区别，我们就必须简单了解一下计算机内存的基本结构。

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`。

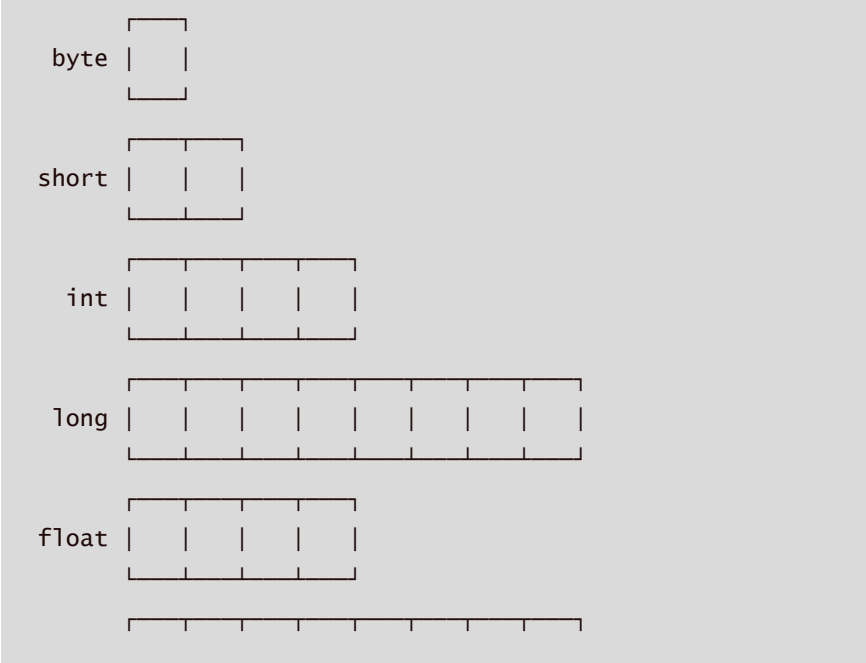
内存单元从0开始编号，称为内存地址。每个内存单元可以看作一间房间，内存地址就是门牌号。

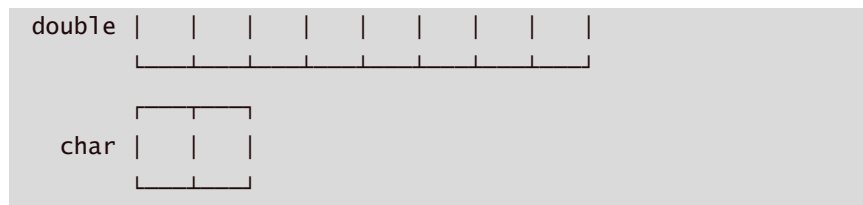


一个字节是1byte，1024字节是1K，1024K是1M，1024M是1G，1024G是1T。
一个拥有4T内存的计算机的字节数量就是：

$$\begin{aligned} 4T &= 4 \times 1024G \\ &= 4 \times 1024 \times 1024M \\ &= 4 \times 1024 \times 1024 \times 1024K \\ &= 4 \times 1024 \times 1024 \times 1024 \times 1024 \\ &= 4398046511104 \end{aligned}$$

不同的数据类型占用的字节数不一样。我们看一下Java基本数据类型占用的字节数：





`byte` 恰好就是一个字节，而 `long` 和 `double` 需要8个字节。

整型

对于整型类型，Java只定义了带符号的整型，因此，最高位的bit表示符号位（0表示正数，1表示负数）。各种整型能表示的最大范围如下：

- byte: -128 ~ 127
- short: -32768 ~ 32767
- int: -2147483648 ~ 2147483647
- long: -9223372036854775808 ~ 9223372036854775807

我们来看定义整型的例子：

```
public class Main {  
    public static void main(String[] args) {  
        int i = 2147483647;  
        int i2 = -2147483648;  
        int i3 = 2_000_000_000; // 加下划线更容易识别  
        int i4 = 0xff0000; // 十六进制表示的16711680  
        int i5 = 0b1000000000; // 二进制表示的512  
        long l = 9000000000000000000L; // long型的结尾需要加  
        L  
    }  
}
```

特别注意：同一个数的不同进制的表示是完全相同的，例如 `15=0xf=0b1111`。

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12.345x10²，也可以表示成1.2345x10³，所以称为浮点数。

下面是定义浮点数的例子：

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

对于 `float` 类型，需要加上 `f` 后缀。

浮点数可表示的范围非常大，`float` 类型可最大表示 3.4×10^38 ，而 `double` 类型可最大表示 1.79×10^308 。

布尔类型

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

Java语言对布尔类型的存储并没有做规定，因为理论上存储布尔类型只需要1 bit，但是通常JVM内部会把 `boolean` 表示为4字节整数。

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：

```
public class Main {
    public static void main(String[] args) {
        char a = 'A';
        char zh = '中';
        System.out.println(a);
        System.out.println(zh);
    }
}
```

注意 `char` 类型使用单引号 `'`，且仅有一个字符，要和双引号 `"` 的字符串类型区分开。

常量

定义变量的时候，如果加上 `final` 修饰符，这个变量就变成了常量：

```
final double PI = 3.14; // PI是一个常量
double r = 5.0;
double area = PI * r * r;
PI = 300; // compile error!
```

常量在定义时进行初始化后就不可再次赋值，再次赋值会导致编译错误。

常量的作用是用有意义的变量名来避免魔术数字（**Magic number**），例如，不要在代码中到处写 **3.14**，而是定义一个常量。如果将来需要提高计算精度，我们只需要在常量的定义处修改，例如，改成 **3.1416**，而不必在所有地方替换 **3.14**。

根据习惯，常量名通常全部大写。

var关键字

有些时候，类型的名字太长，写起来比较麻烦。例如：

```
StringBuilder sb = new StringBuilder();
```

这个时候，如果想省略变量类型，可以使用 **var** 关键字：

```
var sb = new StringBuilder();
```

编译器会根据赋值语句自动推断出变量 **sb** 的类型是 **StringBuilder**。对编译器来说，语句：

```
var sb = new StringBuilder();
```

实际上会自动变成：

```
StringBuilder sb = new StringBuilder();
```

因此，使用 **var** 定义变量，仅仅是少写了变量类型而已。

变量的作用范围

在Java中，多行语句用 **{}** 括起来。很多控制语句，例如条件判断和循环，都以 **{}** 作为它们自身的范围，例如：

```
if (...) { // if开始
    ...
    while (...) { while 开始
        ...
        if (...) { // if开始
            ...
        } // if结束
        ...
    } // while结束
    ...
} // if结束
```

只要正确地嵌套这些`{}`，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。超出了作用域引用这些变量，编译器会报错。举个例子：

```
{
    ...
    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 = "hi";
        ...
    } // 变量x和s作用域到此结束
    ...
} // 变量i作用域到此结束
```

定义变量时，要遵循作用域最小化原则，尽量将变量定义在尽可能小的作用域，并且，不要重复使用变量名。

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

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。四则运算规则和初等数学一致。例如：

```
public class Main {
    public static void main(String[] args) {
        int i = (100 + 200) * (99 - 88); // 3300
        int n = 7 * (5 + (i - 9)); // 23072
        System.out.println(i);
        System.out.println(n);
    }
}
```

整数的数值表示不但是精确的，而且整数运算永远是精确的，即使是除法也是精确的，因为两个整数相除只能得到结果的整数部分：

```
int x = 12345 / 67; // 184
```

求余运算使用%：

```
int y = 12345 % 67; // 12345÷67的余数是17
```

特别注意：整数的除法对于除数为0时运行时将报错，但编译不会报错。

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，而溢出不会出错，却会得到一个奇怪的结果：

```
public class Main {
    public static void main(String[] args) {
        int x = 2147483640;
        int y = 15;
        int sum = x + y;
        System.out.println(sum); // -2147483641
    }
}
```

要解释上述结果，我们把整数 2147483640 和 15 换成二进制做加法：

```
0111 1111 1111 1111 1111 1111 1111 1000
+ 0000 0000 0000 0000 0000 0000 0000 1111
-----
1000 0000 0000 0000 0000 0000 0000 0111
```

由于最高位计算结果为1，因此，加法结果变成了一个负数。

要解决上面的问题，可以把 int 换成 long 类型，由于 long 可表示的整型范围更大，所以结果就不会溢出：

```
long x = 2147483640;
long y = 15;
long sum = x + y;
System.out.println(sum); // 2147483655
```

还有一种简写的运算符，即 +=，-=，*=，/=，它们的使用方法如下：

```
n += 100; // 3409, 相当于 n = n + 100;
n -= 100; // 3309, 相当于 n = n - 100;
```

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：

```
public class Main {
    public static void main(String[] args) {
        int n = 3300;
        n++; // 3301, 相当于 n = n + 1;
        n--; // 3300, 相当于 n = n - 1;
        int y = 100 + (++n); // 不要这么写
        System.out.println(y);
    }
}
```

注意++写在前面和后面计算结果是不同的，++n表示先加1再引用n，n++表示先引用n再加1。不建议把++运算混入到常规运算中，容易自己把自己搞懵了。

移位运算

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：

```
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; // 11110000 00000000 00000000 00000000 = -536870912
```

左移29位时，由于最高位变成1，因此结果变成了负数。

类似的，对整数28进行右移，结果如下：

```

int n = 7;          // 00000000 00000000 00000000 00000111 =
7
int a = n >> 1;     // 00000000 00000000 00000000 00000011 =
3
int b = n >> 2;     // 00000000 00000000 00000000 00000001 =
1
int c = n >> 3;     // 00000000 00000000 00000000 00000000 =
0

```

如果对一个负数进行右移，最高位的`1`不动，结果仍然是一个负数：

```

int n = -536870912;
int a = n >> 1;     // 11110000 00000000 00000000 00000000 =
-268435456
int b = n >> 2;     // 10111000 00000000 00000000 00000000 =
-134217728
int c = n >> 28;    // 11111111 11111111 11111111 11111110 =
-2
int d = n >> 29;    // 11111111 11111111 11111111 11111111 =
-1

```

还有一种不带符号的右移运算，使用`>>>`，它的特点是符号位跟着动，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

```

对`byte`和`short`类型进行移位时，会首先转换为`int`再进行位移。

仔细观察可发现，左移实际上就是不断地 $\times 2$ ，右移实际上就是不断地 $\div 2$ 。

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
    }
}
```

上述按位与运算实际上可以看作两个整数表示的IP地址10.0.17.77和10.0.17.0，通过与运算，可以快速判断一个IP是否在给定的网段内。

运算优先级

在Java的计算表达式中，运算优先级从高到低依次是：

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

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 class Main {
    public static void main(String[] args) {
        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：

```
public class Main {
    public static void main(String[] args) {
        int i1 = 1234567;
        short s1 = (short) i1; // -10617
        System.out.println(s1);
        int i2 = 12345678;
        short s2 = (short) i2; // 24910
        System.out.println(s2);
    }
}
```

因此，强制转型的结果很可能是错的。

练习

计算前N个自然数的和可以根据公示：

$$\frac{(1+N) \times N}{2}$$

请根据公式计算前N个自然数的和：

```
// 计算前N个自然数的和
public class Main {
    public static void main(String[] args) {
        int n = 100;
        // TODO: sum = 1 + 2 + ... + n
        int sum = ???;
        System.out.println(sum);
        System.out.println(sum == 5050 ? "测试通过" : "测试失败");
    }
}
```

下载练习：计算前N个自然数的和（推荐使用IDE练习插件快速下载）

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。

举个栗子：

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，很显然，无论使用`float`还是`double`，都只能存储一个`0.1`的近似值。但是，`0.5`这个浮点数又可以精确地表示。

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：

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

浮点数在内存的表示方法和整数比更加复杂。Java的浮点数完全遵循IEEE-754标准，这也是绝大多数计算机平台都支持的浮点数标准表示方法。

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：

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

例如：

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

这三种特殊值在实际运算中很少碰到，我们只需要了解即可。

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

如果要进行四舍五入，可以对浮点数加上0.5再强制转型：

```
public class Main {
    public static void main(String[] args) {
        double d = 2.6;
        int n = (int) (d + 0.5);
        System.out.println(n);
    }
}
```

练习

根据一元二次方程 $ax^2 + bx + c = 0$ 的求根公式：

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

计算出一元二次方程的两个解：

```
// 一元二次方程
public class Main {
    public static void main(String[] args) {
        double a = 1.0;
        double b = 3.0;
        double c = -4.0;
        // 求平方根可用 Math.sqrt():
        // System.out.println(Math.sqrt(2)); ==> 1.414
        // TODO:
        double r1 = 0;
        double r2 = 0;
        System.out.println(r1);
        System.out.println(r2);
    }
}
```

```
        System.out.println(r1 == 1 && r2 == -4 ? "测试通过"
: "测试失败");
    }
}
```

下载练习：[计算一元二次方程的两个解](#)（推荐使用[IDE练习插件](#)快速下载）

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绝对值之差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。

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
```

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：

```
public class Main {
    public static void main(String[] args) {
        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：

```
public class Main {
    public static void main(String[] args) {
        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。

练习

判断指定年龄是否是小学生（6～12岁）：

```
// 布尔运算
public class Main {
    public static void main(String[] args) {
        int age = 7;
        // primary student的定义: 6~12岁
        boolean isPrimaryStudent = ???;
        System.out.println(isPrimaryStudent ? "Yes" :
            "No");
    }
}
```

下载练习：[判断指定年龄是否是小学生](#)（推荐使用[IDE练习插件](#)快速下载）

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

在Java中，字符和字符串是两个不同的类型。

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：

```
char c1 = 'A';
char c2 = '中';
```

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：

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

还可以直接用转义字符 `\u`+Unicode编码来表示一个字符：

```
// 注意是十六进制：
char c3 = '\u0041'; // 'A', 因为十六进制0041 = 十进制65
char c4 = '\u4e2d'; // '中', 因为十六进制4e2d = 十进制20013
```

字符串类型

和 `char` 类型不同，字符串类型 `String` 是引用类型，我们用双引号 `"..."` 表示字符串。一个字符串可以存储0个到任意个字符：

```
String s = ""; // 空字符串，包含0个字符
String s1 = "A"; // 包含一个字符
String s2 = "ABC"; // 包含3个字符
String s3 = "中文 ABC"; // 包含6个字符，其中有一个空格
```

因为字符串使用双引号 `"..."` 表示开始和结束，那如果字符串本身恰好包含一个 `"` 字符怎么表示？例如，`"abc"xyz"`，编译器就无法判断中间的引号究竟是字符串的一部分还是表示字符串结束。这个时候，我们需要借助转义字符 `\`：

```
String s = "abc\"xyz"; // 包含7个字符：a, b, c, ", x, y, z
```

因为 `\` 是转义字符，所以，两个 `\\` 表示一个 `\` 字符：

```
String s = "abc\\xyz"; // 包含7个字符：a, b, c, \, x, y, z
```

常见的转义字符包括：

- `\"` 表示字符 `"`
- `\'` 表示字符 `'`
- `\\` 表示字符 `\`
- `\n` 表示换行符
- `\r` 表示回车符
- `\t` 表示Tab
- `\u####` 表示一个Unicode编码的字符

例如：

```
String s = "ABC\n\u4e2d\u6587"; // 包含6个字符：A, B, C, 换行符, 中, 文
```

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：

```
public class Main {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "world";
        String s = s1 + " " + s2 + "!";
        System.out.println(s);
    }
}
```


如果用+连接字符串和其他数据类型，会将其他数据类型先自动转型为字符串，再连接：

```
public class Main {  
    public static void main(String[] args) {  
        int age = 25;  
        String s = "age is " + age;  
        System.out.println(s);  
    }  
}
```

多行字符串

如果我们要表示多行字符串，使用+号连接会非常不方便：

```
String s = "first line \n"  
    + "second line \n"  
    + "end";
```

从Java 13开始，字符串可以用""...""表示多行字符串（Text Blocks）了。举个例子：

```
public class Main {  
    public static void main(String[] args) {  
        String s = ""  
            SELECT * FROM  
            users  
            WHERE id > 100  
            ORDER BY name DESC  
            "";  
        System.out.println(s);  
    }  
}
```

上述多行字符串实际上是5行，在最后一个DESC后面还有一个\n。如果我们不想在字符串末尾加一个\n，就需要这么写：

```
String s = ""  
    SELECT * FROM  
    users  
    WHERE id > 100  
    ORDER BY name DESC"";
```

还需要注意到，多行字符串前面共同的空格会被去掉，即：

```
String s = ""
.....SELECT * FROM
.....  users
.....WHERE id > 100
.....ORDER BY name DESC
....."";
```

用`.`标注的空格都会被去掉。

如果多行字符串的排版不规则，那么，去掉的空格就会变成这样：

```
String s = ""
.....  SELECT * FROM
.....    users
.....WHERE id > 100
.....  ORDER BY name DESC
.....  "";
```

即总是以最短的行首空格为基准。

最后，由于多行字符串是作为Java 13的预览特性（Preview Language Features）实现的，编译的时候，我们还需要给编译器加上参数：

```
javac --source 13 --enable-preview Main.java
```

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：

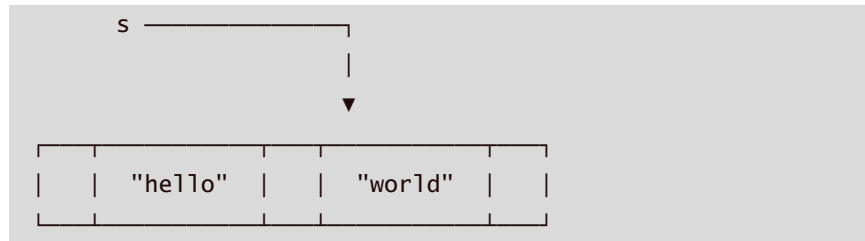
```
public class Main {
    public static void main(String[] args) {
        String s = "hello";
        System.out.println(s); // 显示 hello
        s = "world";
        System.out.println(s); // 显示 world
    }
}
```

观察执行结果，难道字符串`s`变了吗？其实变的不是字符串，而是变量`s`的“指向”。

执行`String s = "hello";`时，JVM虚拟机先创建字符串`"hello"`，然后，把字符串变量`s`指向它：



紧接着，执行 `s = "world"`；时，JVM虚拟机先创建字符串 `"world"`，然后，把字符串变量 `s` 指向它：



原来的字符串 `"hello"` 还在，只是我们无法通过变量 `s` 访问它而已。因此，字符串的不可变是指字符串内容不可变。

理解了引用类型的“指向”后，试解释下面的代码输出：

```
public class Main {
    public static void main(String[] args) {
        String s = "hello";
        String t = s;
        s = "world";
        System.out.println(t); // t是"hello"还是"world"?
    }
}
```

空值 `null`

引用类型的变量可以指向一个空值 `null`，它表示不存在，即该变量不指向任何对象。例如：

```
String s1 = null; // s1是null
String s2; // 没有赋初值，s2也是null
String s3 = s1; // s3也是null
String s4 = ""; // s4指向空字符串，不是null
```

注意要区分空值 `null` 和空字符串 `""`，空字符串是一个有效的字符串对象，它不等于 `null`。

练习

请将一组 `int` 值视为字符的 Unicode 编码，然后将它们拼成一个字符串：

```

public class Main {
    public static void main(String[] args) {
        // 请将下面一组int值视为字符的Unicode码，把它们拼成一个字
        符串：
        int a = 72;
        int b = 105;
        int c = 65281;
        // FIXME:
        String s = a + b + c;
        System.out.println(s);
    }
}

```

下载练习：[Unicode值拼接字符串](#)（推荐使用[IDE练习插件](#)快速下载）

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

如果我们有一组类型相同的变量，例如，5位同学的成绩，可以这么写：

```

public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩：
        int n1 = 68;
        int n2 = 79;
        int n3 = 91;
        int n4 = 85;
        int n5 = 62;
    }
}

```

但其实没有必要定义5个 `int` 变量。可以使用数组来表示“一组”`int` 类型。代码如下：

```
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns = new int[5];
        ns[0] = 68;
        ns[1] = 79;
        ns[2] = 91;
        ns[3] = 85;
        ns[4] = 62;
    }
}
```

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：

```
public class Main {
    public static void main(String[] args) {
        // 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：

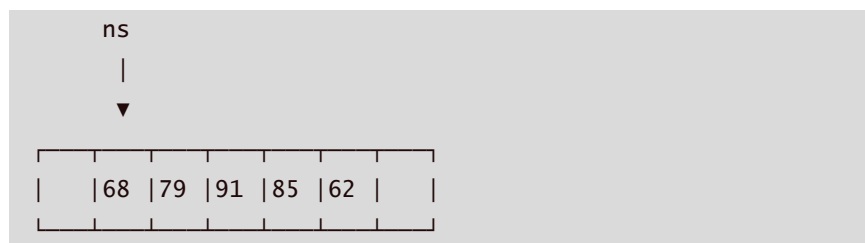
```
int[] ns = { 68, 79, 91, 85, 62 };
```

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：

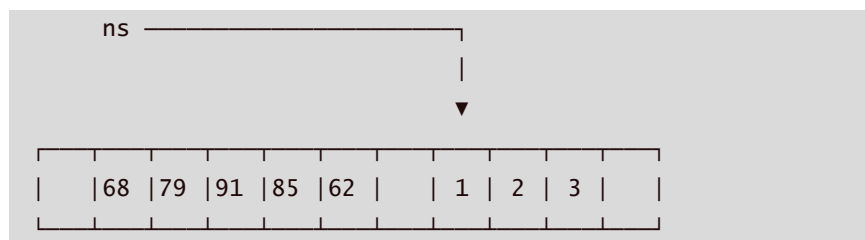
```
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩：
        int[] ns;
        ns = new int[] { 68, 79, 91, 85, 62 };
        System.out.println(ns.length); // 5
        ns = new int[] { 1, 2, 3 };
        System.out.println(ns.length); // 3
    }
}
```

数组大小变了吗？看上去好像是变了，但其实根本没变。

对于数组 `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。

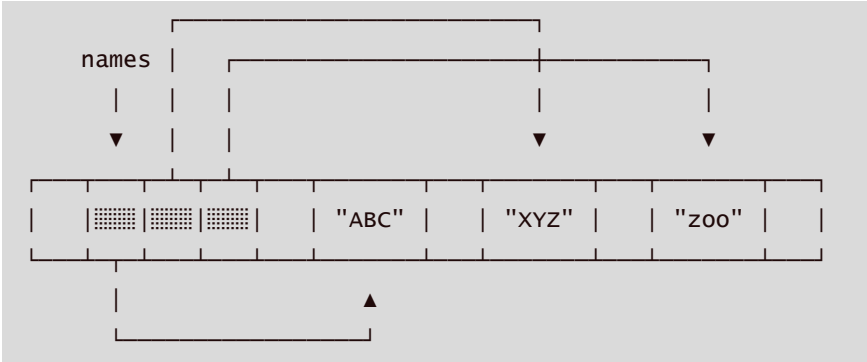
字符串数组

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？

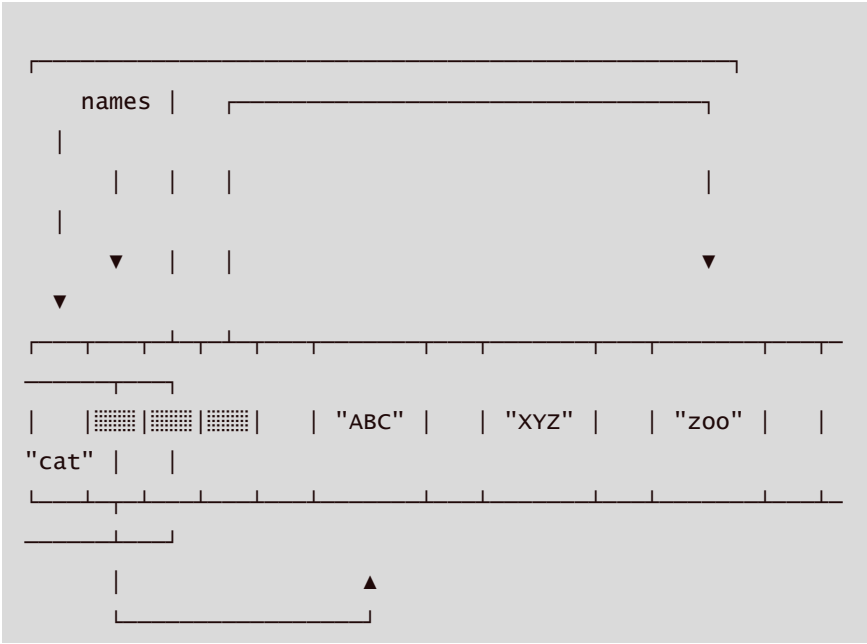
字符串是引用类型，因此我们先定义一个字符串数组：

```
String[] names = {
    "ABC", "XYZ", "zoo"
};
```

对于 `String[]` 类型的数组变量 `names`，它实际上包含3个元素，但每个元素都指向某个字符串对象：



对 `names[1]` 进行赋值，例如 `names[1] = "cat"`，效果如下：



这里注意到原来 `names[1]` 指向的字符串 `"XYZ"` 并没有改变，仅仅是将 `names[1]` 的引用从指向 `"XYZ"` 改成了指向 `"cat"`，其结果是字符串 `"XYZ"` 再也无法通过 `names[1]` 访问到了。

对“指向”有了更深入的理解后，试解释如下代码：

```
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；

流程控制

在Java程序中，JVM默认总是顺序执行以分号;结束的语句。但是，在实际的代码中，程序经常需要做条件判断、循环，因此，需要有多种流程控制语句，来实现程序的跳转和循环等功能。

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");  
    }  
}
```

注意观察上述代码的执行效果。

格式化输出

Java还提供了格式化输出的功能。为什么要格式化输出？因为计算机表示的数据不一定适合人来阅读：

```
public class Main {
    public static void main(String[] args) {
        double d = 12900000;
        System.out.println(d); // 1.29E7
    }
}
```

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数
    }
}
```

详细的格式化参数请参考JDK文档[java.util.Formatter](#)

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。

要测试输入，我们不能在线运行它，因为输入必须从命令行读取，因此，需要走编译、执行的流程：

```
$ javac Main.java
```

这个程序编译时如果有警告，可以暂时忽略它，在后面学习IO的时候再详细解释。编译成功后，执行：

```
$ java Main
Input your name: Bob
Input your age: 12
Hi, Bob, you are 12
```

根据提示分别输入一个字符串和整数后，我们得到了格式化的输出。

练习

请帮小明同学设计一个程序，输入上次考试成绩（int）和本次考试成绩（int），然后输出成绩提高的百分比，保留两位小数位（例如，21.75%）。

下载练习：[输入和输出练习](#)（推荐使用[IDE练习插件](#)快速下载）

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`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）。

让我们来看一个例子：

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

当条件`n >= 60`计算结果为`true`时，`if`语句块被执行，将打印“及格了”，否则，`if`语句块将被跳过。修改`n`的值可以看到执行效果。

注意到`if`语句包含的块可以包含多条语句：

```
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");
    }
}
```

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("END");
    }
}
```

但是，省略花括号并不总是一个好主意。假设某个时候，突然想给 `if` 语句块增加一条语句时：

```
public class Main {
    public static void main(String[] args) {
        int n = 50;
        if (n >= 60)
            System.out.println("及格了");
            System.out.println("恭喜你"); // 注意这条语句不是
if语句块的一部分
        System.out.println("END");
    }
}
```

由于使用缩进格式，很容易把两行语句都看成 `if` 语句的执行块，但实际上只有第一行语句是 `if` 的执行块。在使用 `git` 这些版本控制系统自动合并时更容易出问题，所以不推荐忽略花括号的写法。

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");
    }
}

```

修改上述代码`n`的值，观察`if`条件为`true`或`false`时，程序执行的语句块。

注意，`else`不是必须的。

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");
    }
}

```

串联的效果其实相当于：

```

if (n >= 90) {
    // n >= 90为true:
    System.out.println("优秀");
} else {
    // n >= 90为false:
    if (n >= 60) {
        // n >= 60为true:
        System.out.println("及格了");
    } else {
        // n >= 60为false:
        System.out.println("挂科了");
    }
}
}

```

在串联使用多个`if`时，要特别注意判断顺序。观察下面的代码：

```

public class Main {
    public static void main(String[] args) {
        int n = 100;
        if (n >= 60) {
            System.out.println("及格了");
        } else if (n >= 90) {
            System.out.println("优秀");
        } else {
            System.out.println("挂科了");
        }
    }
}

```

执行发现，`n = 100`时，满足条件`n >= 90`，但输出的不是"优秀"，而是"及格了"，原因是`if`语句从上到下执行时，先判断`n >= 60`成功后，后续`else`不再执行，因此，`if (n >= 90)`没有机会执行了。

正确的方式是按照判断范围从大到小依次判断：

```

// 从大到小依次判断：
if (n >= 90) {
    // ...
} else if (n >= 60) {
    // ...
} else {
    // ...
}

```

或者改写成从小到大依次判断：

```

// 从小到大依次判断：
if (n < 60) {
    // ...
} else if (n < 90) {
    // ...
} else {
    // ...
}

```

使用`if`时，还要特别注意边界条件。例如：

```
public class Main {
    public static void main(String[] args) {
        int n = 90;
        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");
        }
    }
}
```

假设我们期望90分或更高为“优秀”，上述代码输出的却是“及格”，原因是`>`和`>=`效果是不同的。

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`：

```

public class Main {
    public static void main(String[] args) {
        String s1 = null;
        if (s1.equals("hello")) {
            System.out.println("hello");
        }
    }
}

```

要避免 `NullPointerException` 错误，可以利用短路运算符 `&&`：


```
public class Main {
    public static void main(String[] args) {
        String s1 = null;
        if (s1 != null && s1.equals("hello")) {
            System.out.println("hello");
        }
    }
}
```

还可以把一定不是 `null` 的对象 `"hello"` 放到前面：例如：`if ("hello".equals(s)) { ... }`。

练习

请用 `if ... else` 编写一个程序，用于计算体质指数BMI，并打印结果。

BMI = 体重(kg)除以身高(m)的平方

BMI结果：

- 过轻：低于18.5
- 正常：18.5-25
- 过重：25-28
- 肥胖：28-32
- 非常肥胖：高于32

下载练习：[BMI练习](#)（推荐使用[IDE练习插件](#)快速下载）

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：

1. 单人模式
2. 多人模式
3. 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;
        }
    }
}
```

如果把 `switch` 语句翻译成 `if` 语句，那么上述的代码相当于：

```
if (option == 1) {
    System.out.println("Selected 1");
} else if (option == 2) {
    System.out.println("Selected 2");
} else if (option == 3) {
    System.out.println("Selected 3");
} else {
    System.out.println("Not selected");
}
```

对于多个 `==` 判断的情况，使用 `switch` 结构更加清晰。

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`。

如果有几个 `case` 语句执行的是同一组语句块，可以这么写：

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

```

        case 3:
            System.out.println("Selected 2, 3");
            break;
        default:
            System.out.println("Not selected");
            break;
    }
}
}

```

使用 `switch` 语句时，只要保证有 `break`，`case` 的顺序不影响程序逻辑：

```

switch (option) {
    case 3:
        ...
        break;
    case 2:
        ...
        break;
    case 1:
        ...
        break;
}

```

但是仍然建议按照自然顺序排列，便于阅读。

`switch` 语句还可以匹配字符串。字符串匹配时，是比较“内容相等”。例如：

```

public class Main {
    public static void main(String[] args) {
        String fruit = "apple";
        switch (fruit) {
            case "apple":
                System.out.println("Selected apple");
                break;
            case "pear":
                System.out.println("Selected pear");
                break;
            case "mango":
                System.out.println("Selected mango");
                break;
            default:
                System.out.println("No fruit selected");
                break;
        }
    }
}

```

`switch` 语句还可以使用枚举类型，枚举类型我们在后面讲解。

编译检查

使用IDE时，可以自动检查是否漏写了 `break` 语句和 `default` 语句，方法是打开IDE的编译检查。

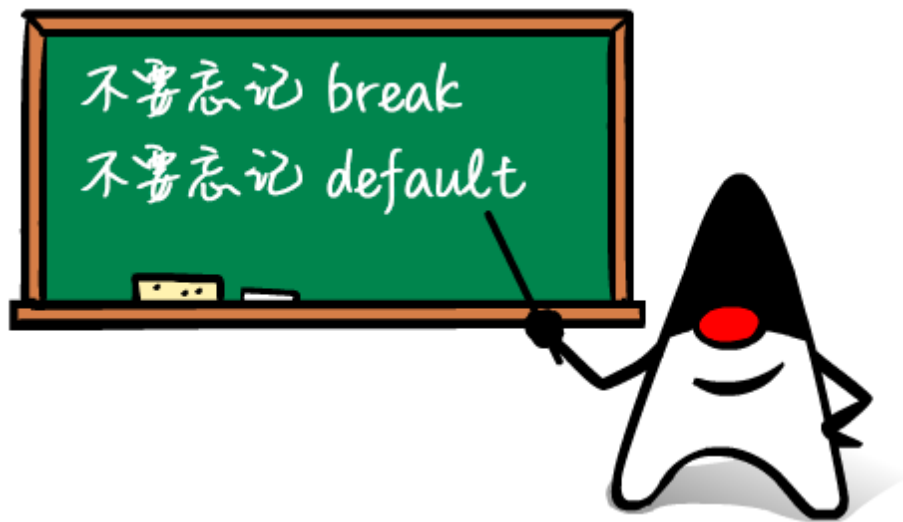
在Eclipse中，选择 `Preferences - Java - Compiler - Errors/warnings - Potential programming problems`，将以下检查标记为Warning：

- 'switch' is missing 'default' case
- 'switch' case fall-through

在Idea中，选择 `Preferences - Editor - Inspections - Java - Control flow issues`，将以下检查标记为Warning：

- Fallthrough in 'switch' statement
- 'switch' statement without 'default' branch

当 `switch` 语句存在问题时，即可在IDE中获得警告提示。



switch表达式

使用 `switch` 时，如果遗漏了 `break`，就会造成严重的逻辑错误，而且不易在源代码中发现错误。从Java 12开始，`switch` 语句升级为更简洁的表达式语法，使用类似模式匹配（Pattern Matching）的方法，保证只有一种路径会被执行，并且不需要 `break` 语句：

```
public class Main {  
    public static void main(String[] args) {  
        String fruit = "apple";  
        switch (fruit) {  
            case "apple" -> System.out.println("Selected  
apple");  
            case "pear" -> System.out.println("Selected  
pear");  
            case "mango" -> {  
                System.out.println("Selected mango");  
                System.out.println("Good choice!");  
            }  
        }  
    }  
}
```

```

    }
    default -> System.out.println("No fruit
selected");
    }
}
}

```

注意新语法使用 `->`，如果有多条语句，需要用 `{}` 括起来。不要写 `break` 语句，因为新语法只会执行匹配的语句，没有穿透效应。

很多时候，我们还可能用 `switch` 语句给某个变量赋值。例如：

```

int opt;
switch (fruit) {
case "apple":
    opt = 1;
    break;
case "pear":
case "mango":
    opt = 2;
    break;
default:
    opt = 0;
    break;
}

```

使用新的 `switch` 语法，不但不需要 `break`，还可以直接返回值。把上面的代码改写如下：

```

public class Main {
    public static void main(String[] args) {
        String fruit = "apple";
        int opt = switch (fruit) {
            case "apple" -> 1;
            case "pear", "mango" -> 2;
            default -> 0;
        }; // 注意赋值语句要以;结束
        System.out.println("opt = " + opt);
    }
}

```

这样可以获得更简洁的代码。

yield

大多数时候，在 `switch` 表达式内部，我们会返回简单的值。

但是，如果需要复杂的语句，我们也可以写很多语句，放到 `{...}` 里，然后，用 `yield` 返回一个值作为 `switch` 语句的返回值：

```

public class Main {
    public static void main(String[] args) {
        String fruit = "orange";
        int opt = switch (fruit) {
            case "apple" -> 1;
            case "pear", "mango" -> 2;
            default -> {
                int code = fruit.hashCode();
                yield code; // switch语句返回值
            }
        };
        System.out.println("opt = " + opt);
    }
}

```

由于 `switch` 表达式是作为Java 13的预览特性（Preview Language Features）实现的，编译的时候，我们还需要给编译器加上参数：

```
javac --source 13 --enable-preview Main.java
```

这样才能正常编译。

练习

使用 `switch` 实现一个简单的石头、剪子、布游戏。

下载练习：[switch练习](#)（推荐使用[IDE练习插件](#)快速下载）

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；
- 从Java 13开始，`switch` 语句升级为表达式，不再需要 `break`，并且允许使用 `yield` 返回值。

while循环

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。

例如，计算从1到100的和：

```
1 + 2 + 3 + 4 + ... + 100 = ?
```

除了用数列公式外，完全可以让计算机做100次循环累加。因为计算机的特点是计算速度非常快，我们让计算机循环一亿次也用不到1秒，所以很多计算的任务，人去算是算不了的，但是计算机算，使用循环这种简单粗暴的方法就可以快速得到结果。

我们先看Java提供的while条件循环。它的基本用法是：

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

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。

我们用while循环来累加1到100，可以这么写：

```
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  
    }  
}
```

注意到while循环是先判断循环条件，再循环，因此，有可能一次循环都不做。

对于循环条件判断，以及自增变量的处理，要特别注意边界条件。思考一下下面的代码为何没有获得正确结果：

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

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。

练习

使用`while`计算从`m`到`n`的和：

```
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);
    }
}
```

下载练习：[while练习](#)（推荐使用[IDE练习插件](#)快速下载）

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

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);  
    }  
}
```

使用`do while`循环时，同样要注意循环条件的判断。

练习

使用`do while`循环计算从`m`到`n`的和。

```
public class Main {  
    public static void main(String[] args) {  
        int sum = 0;  
        int m = 20;  
        int n = 100;  
        // 使用do while计算M+...+N:  
        do {  
        } while (false);  
        System.out.println(sum);  
    }  
}
```

下载练习：[do while练习](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- `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
}
```

如果我们要对一个整型数组的所有元素求和，可以用 `for` 循环实现：

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

上面代码的循环条件是 `i`。因为 `ns` 数组的长度是 `5`，因此，当循环 `5` 次后，`i` 的值被更新为 `5`，就不满足循环条件，因此 `for` 循环结束。

思考：如果把循环条件改为 `i<=ns.length`，会出现什么问题？

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。

使用 `for` 循环时，千万不要在循环体内修改计数器！在循环体中修改计数器常常导致莫名其妙的逻辑错误。对于下面的代码：

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

虽然不会报错，但是，数组元素只打印了一半，原因是循环内部的 `i = i + 1` 导致了计数器变量每次循环实际上加了2（因为 `for` 循环还会自动执行 `i++`）。因此，在 `for` 循环中，不要修改计数器的值。计数器的初始化、判断条件、每次循环后的更新条件统一放到 `for()` 语句中可以一目了然。

如果希望只访问索引为奇数的数组元素，应该把 `for` 循环改写为：

```
int[] ns = { 1, 4, 9, 16, 25 };
for (int i=0; i<ns.length; i=i+2) {
    System.out.println(ns[i]);
}
```

通过更新计数器的语句 `i=i+2` 就达到了这个效果，从而避免了在循环体内去修改变量 `i`。

使用 `for` 循环时，计数器变量 `i` 要尽量定义在 `for` 循环中：

```
int[] ns = { 1, 4, 9, 16, 25 };
for (int i=0; i<ns.length; i++) {
    System.out.println(ns[i]);
}
// 无法访问i
int n = i; // compile error!
```

如果变量 `i` 定义在 `for` 循环外：

```
int[] ns = { 1, 4, 9, 16, 25 };
int i;
for (i=0; i<ns.length; i++) {
    System.out.println(ns[i]);
}
// 仍然可以使用i
int n = i;
```

那么，退出**for**循环后，变量*i*仍然可以被访问，这就破坏了变量应该把访问范围缩到最小的原则。

灵活使用**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 (;;) {
    ...
}
```

通常不推荐这样写，但是，某些情况下，是可以省略**for**循环的某些语句的。

for each循环

for循环经常用来遍历数组，因为通过计数器可以根据索引来访问数组的每个元素：

```
int[] ns = { 1, 4, 9, 16, 25 };
for (int i=0; i<ns.length; i++) {
    System.out.println(ns[i]);
}
```

但是，很多时候，我们实际上真正想要访问的是数组每个元素的值。Java还提供了另一种**for each**循环，它可以更简单地遍历数组：

```
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，包括后面会介绍的 `List`、`Map` 等。

练习1

给定一个数组，请用 `for` 循环倒序输出每一个元素：

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

练习2

利用 `for each` 循环对数组每个元素求和：

```
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        int sum = 0;
        for (???) {
            // TODO
        }
        System.out.println(sum); // 55
    }
}
```

练习3

圆周率 π 可以使用公式计算：

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

请利用 `for` 循环计算 π :

```
public class Main {
    public static void main(String[] args) {
        double pi = 0;
        for (???) {
            // TODO
        }
        System.out.println(pi);
    }
}
```

下载练习: [for循环计算 \$\pi\$ 练习](#) (推荐使用[IDE练习插件](#)快速下载)

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");
    }
}
}

```

上面的代码是两个 `for` 循环嵌套。因为 `break` 语句位于内层的 `for` 循环，因此，它会跳出内层 `for` 循环，但不会跳出外层 `for` 循环。

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。

数组操作

本节我们将讲解对数组的操作，包括：

- 遍历；
- 排序。

以及多维数组的概念。

遍历数组

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`，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。

打印数组内容

直接打印数组变量，得到的是数组在JVM中的引用地址：

```
int[] ns = { 1, 1, 2, 3, 5, 8 };
System.out.println(ns); // 类似 [I@7852e922
```

这并没有什么意义，因为我们希望打印的数组的元素内容。因此，使用 `for each` 循环来打印它：

```
int[] ns = { 1, 1, 2, 3, 5, 8 };
for (int n : ns) {
    System.out.print(n + ", ");
}
```

使用 `for each` 循环打印也很麻烦。幸好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));
    }
}
```

练习

请按倒序遍历数组并打印每个元素：

```
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        // 倒序打印数组元素：
        for (???) {
            System.out.println(???);
        }
    }
}
```

下载练习：[倒序遍历数组练习](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- 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。像这么写是错误的：

```
int x = 1;
int y = 2;

x = y; // x现在是2
y = x; // y现在还是2
```

正确的写法是：

```
int x = 1;
int y = 2;

int t = x; // 把x的值保存在临时变量t中，t现在是1
x = y; // x现在是2
y = t; // y现在是t的值1
```

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：

```
import java.util.Arrays;

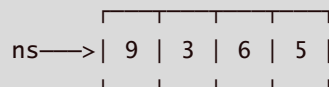
public class Main {
    public static void main(String[] args) {
        int[] ns = { 28, 12, 89, 73, 65, 18, 96, 50, 8, 36 };

        Arrays.sort(ns);
        System.out.println(Arrays.toString(ns));
    }
}
```

必须注意，对数组排序实际上修改了数组本身。例如，排序前的数组是：

```
int[] ns = { 9, 3, 6, 5 };
```

在内存中，这个整型数组表示如下：



```
ns—>| 9 | 3 | 6 | 5 |
```

当我们调用 `Arrays.sort(ns);` 后，这个整型数组在内存中变为：



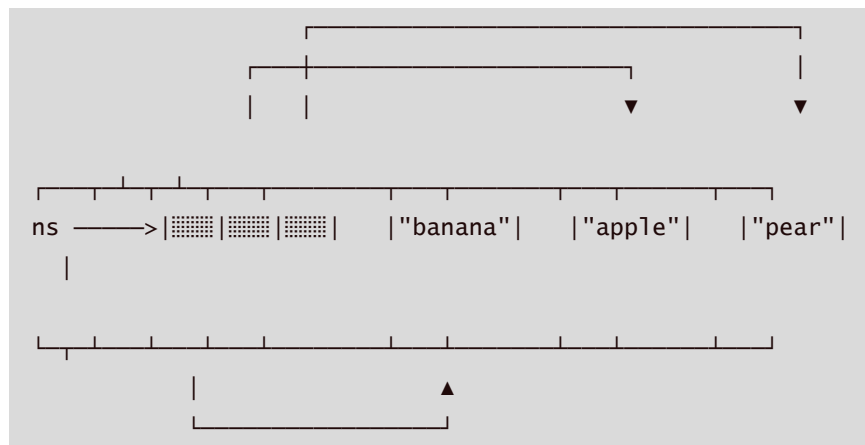
```
ns—>| 3 | 5 | 6 | 9 |
```

即变量 `ns` 指向的数组内容已经被改变了。

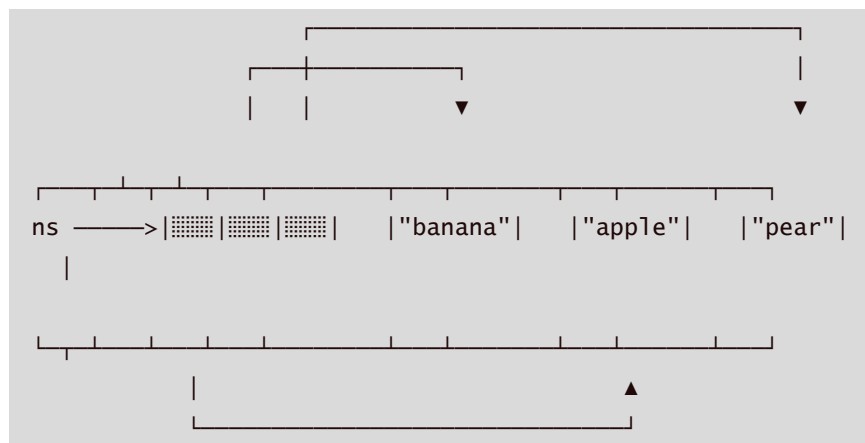
如果对一个字符串数组进行排序，例如：

```
String[] ns = { "banana", "apple", "pear" };
```

排序前，这个数组在内存中表示如下：



调用 `Arrays.sort(ns)`; 排序后，这个数组在内存中表示如下：



原来的3个字符串在内存中均没有任何变化，但是 `ns` 数组的每个元素指向变化了。

练习

请思考如何实现对数组进行降序排序：

```
// 降序排序
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));
        // TODO:
        // 排序后：
        System.out.println(Arrays.toString(ns));
        if (Arrays.toString(ns).equals("[96, 89, 73, 65, 50, 36, 28, 18, 12, 8]")) {
            System.out.println("测试成功");
        } else {
```

```

        System.out.println("测试失败");
    }
}
}

```

下载练习：[降序排序练习](#)（推荐使用[IDE练习插件](#)快速下载）

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

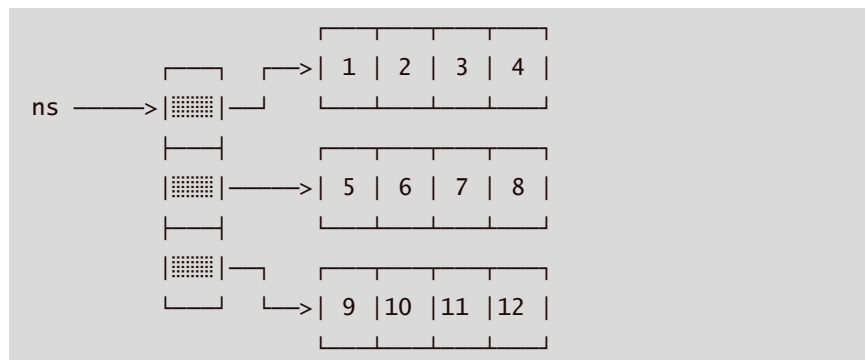
二维数组就是数组的数组。定义一个二维数组如下：

```

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
    }
}

```

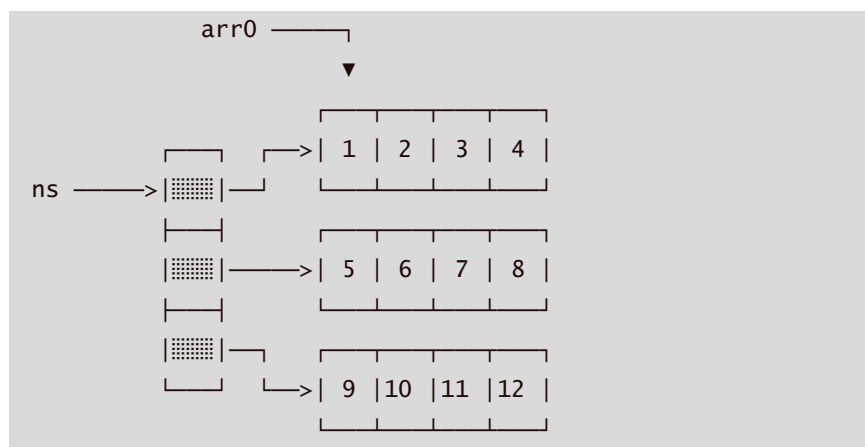
因为 **ns** 包含3个数组，因此，**ns.length** 为 **3**。实际上 **ns** 在内存中的结构如下：



如果我们定义一个普通数组 **arr0**，然后把 **ns[0]** 赋值给它：

```
public class Main {
    public static void main(String[] args) {
        int[][] ns = {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 }
        };
        int[] arr0 = ns[0];
        System.out.println(arr0.length); // 4
    }
}
```

实际上 `arr0` 就获取了 `ns` 数组的第0个元素。因为 `ns` 数组的每个元素也是一个数组，因此，`arr0` 指向的数组就是 `{ 1, 2, 3, 4 }`。在内存中，结构如下：



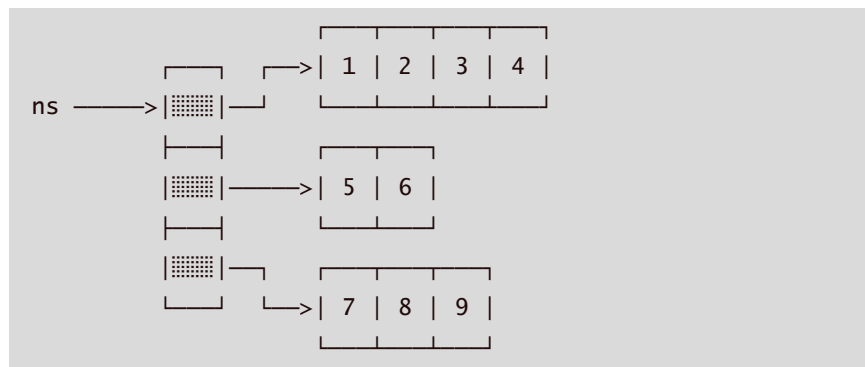
访问二维数组的某个元素需要使用 `array[row][col]`，例如：

```
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 }
};
```

这个二维数组在内存中的结构如下：



要打印一个二维数组，可以使用两层嵌套的for循环：

```
for (int[] arr : ns) {  
    for (int n : arr) {  
        System.out.print(n);  
        System.out.print(' ', ' ');  
    }  
    System.out.println();  
}
```

或者使用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));  
    }  
}
```

三维数组

三维数组就是二维数组的数组。可以这么定义一个三维数组：

```
int[][][] ns = {  
    {  
        {1, 2, 3},  
        {4, 5, 6},  
        {7, 8, 9}  
    },  
    {  
        {10, 11},  
        {12, 13}  
    },  
}
```

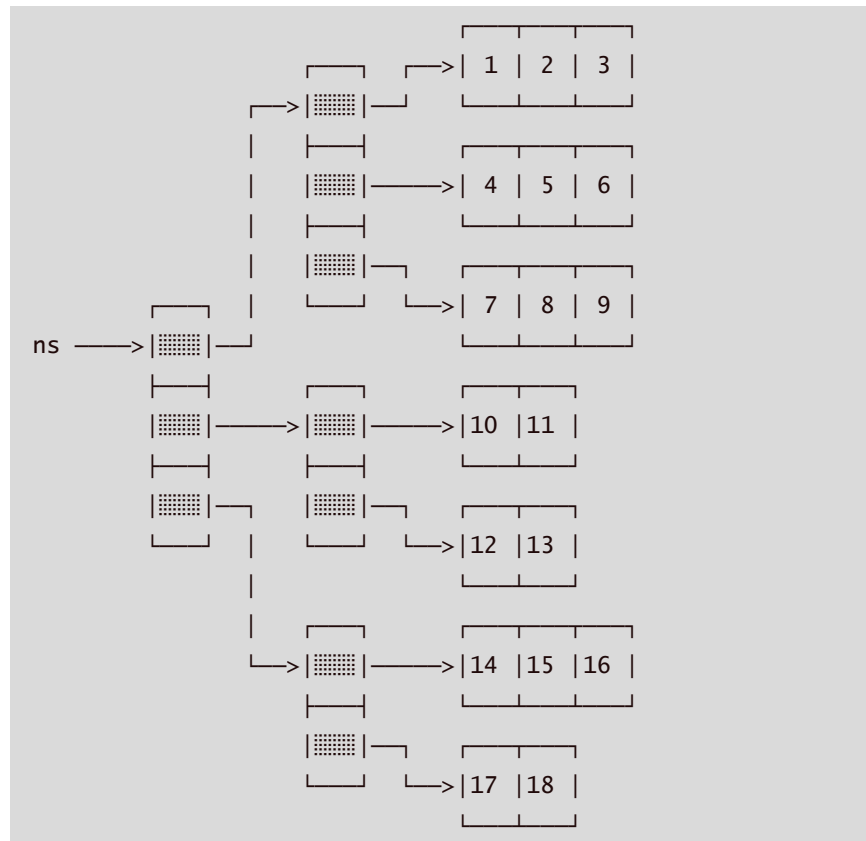


```

{
    {14, 15, 16},
    {17, 18}
}
};

```

它在内存中的结构如下：



如果我们要访问三维数组的某个元素，例如，`ns[2][0][1]`，只需要顺着定位找到对应的最终元素 **15** 即可。

理论上，我们可以定义任意的N维数组。但在实际应用中，除了二维数组在某些时候还能用得上，更高维度的数组很少使用。

练习

使用二维数组可以表示一组学生的各科成绩，请计算所有学生的平均分：

```

public class Main {
    public static void main(String[] args) {
        // 用二维数组表示的学生成绩：
        int[][] scores = {
            { 82, 90, 91 },
            { 68, 72, 64 },
            { 95, 91, 89 },
            { 67, 52, 60 },
            { 79, 81, 85 },
        };
    }
}

```

```

// TODO:
double average = 0;
System.out.println(average);
if (Math.abs(average - 77.733333) < 0.000001) {
    System.out.println("测试成功");
} else {
    System.out.println("测试失败");
}
}
}

```

下载练习：[计算平均分](#)（推荐使用[IDE练习插件](#)快速下载）

小结

- 二维数组就是数组的数组，三维数组就是二维数组的数组；
- 多维数组的每个数组元素长度都不要相同；
- 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);
        }
    }
}

```

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  
v 1.0
```

这样，程序就可以根据传入的命令行参数，作出不同的响应。

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。