

这是专门针对小白的零基础Java教程。

# Java教程

为什么要学Java？

因为Java是全球排名第一的编程语言，Java工程师也是市场需求最大的软件工程师，选择Java，就是选择了高薪。



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；
- 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。

这意味着Java拥有最广泛的就业市场。

## 教程特色

虽然是零基础Java教程，但是覆盖了从基础到高级的Java核心编程，从小白成长到架构师，实现硬实力高薪就业！

还可以边学边练，而且可以在线练习！

并且，时刻更新至最新版Java！目前教程版本是：

## Java 13!

最重要的是：

# 免费！

不要犹豫了！现在开始学习Java，从入门到架构师！



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快速入门



通过本章的学习，可以编写基本的Java程序。

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。

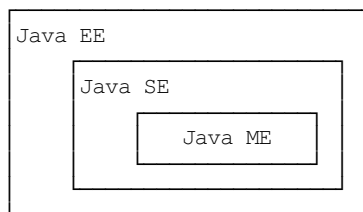
## Java简介

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**。

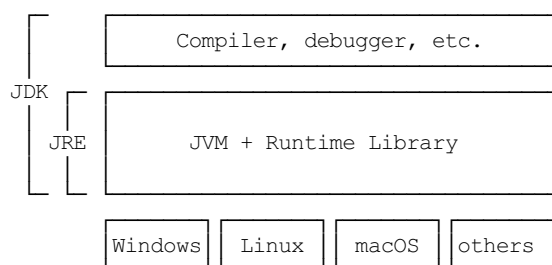
## 名词解释

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，如果真正要选择一个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>

请问Java之父是：

```
-----  
    James Bond  
[x] James Gosling  
    James Simons
```

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。

## 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 - □ x
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 - □ x
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。



我们来编写第一个Java程序。

## 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。

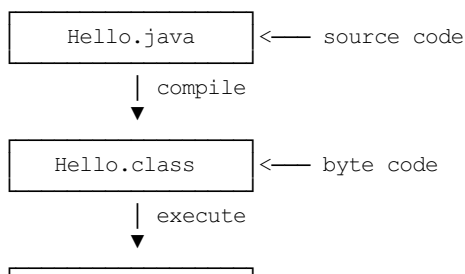
Java规定，某个类定义的public static void main(String[] args)是Java程序的固定入口方法，因此，Java程序总是从main方法开始执行。

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：

## Java代码助手

- 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](#)

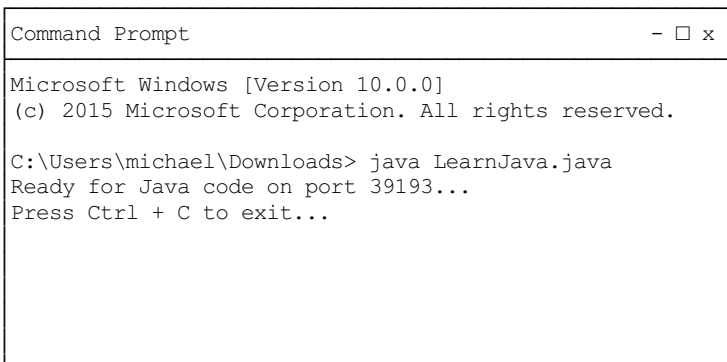
### 运行

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：



### 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");
    }
}
```

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

## 使用IDE

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：

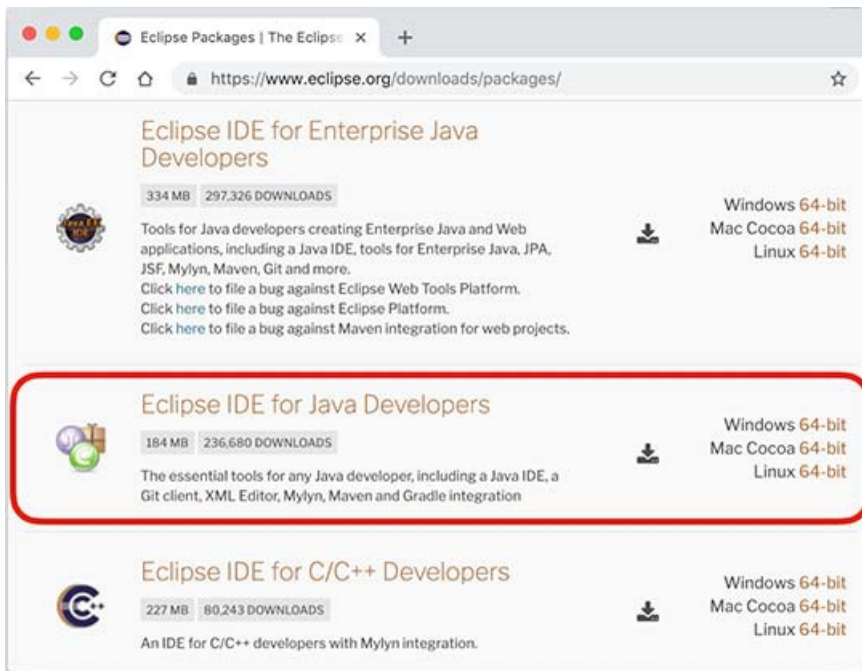
- 完全免费使用；
- 所有功能完全满足Java开发需求。

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：



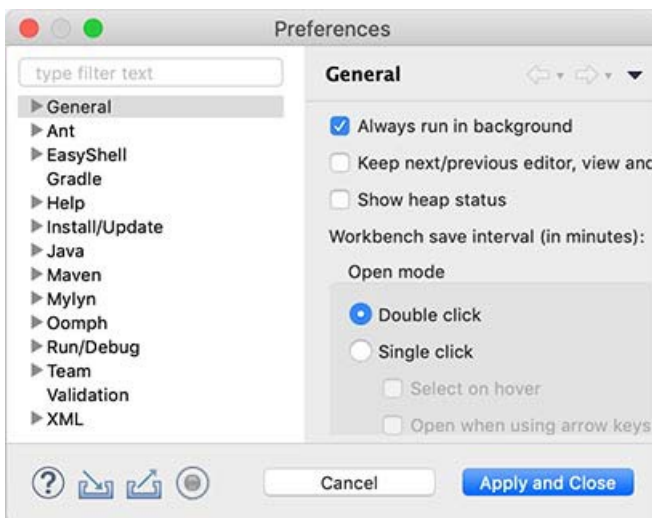
根据操作系统是Windows、Mac还是Linux，从右边选择对应的下载链接。

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

勾选“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。

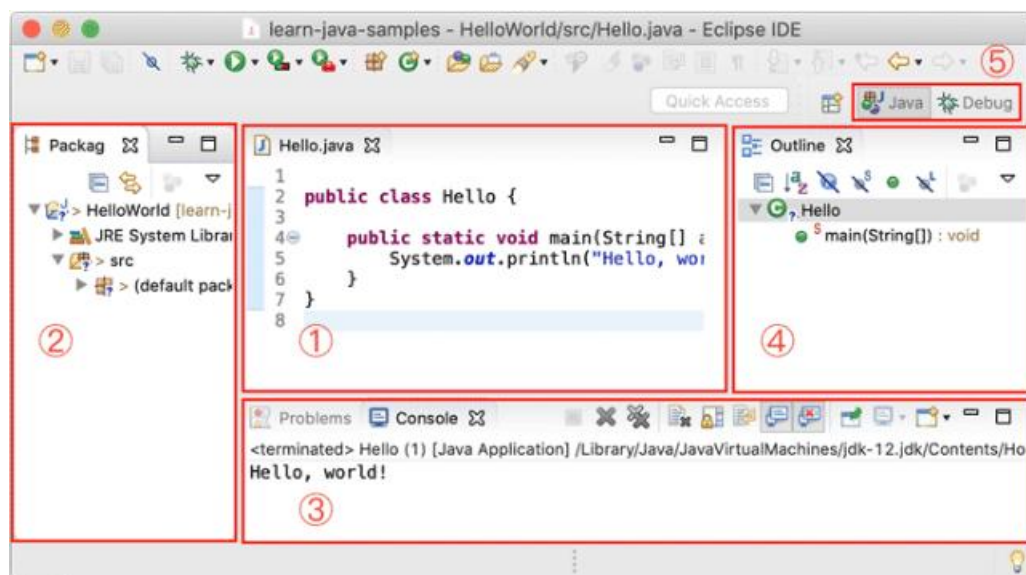
去掉“Use default compliance settings”并钩上“Enable preview features for Java 13”，这样我们就可以使用Java 13的预览功能。

## 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

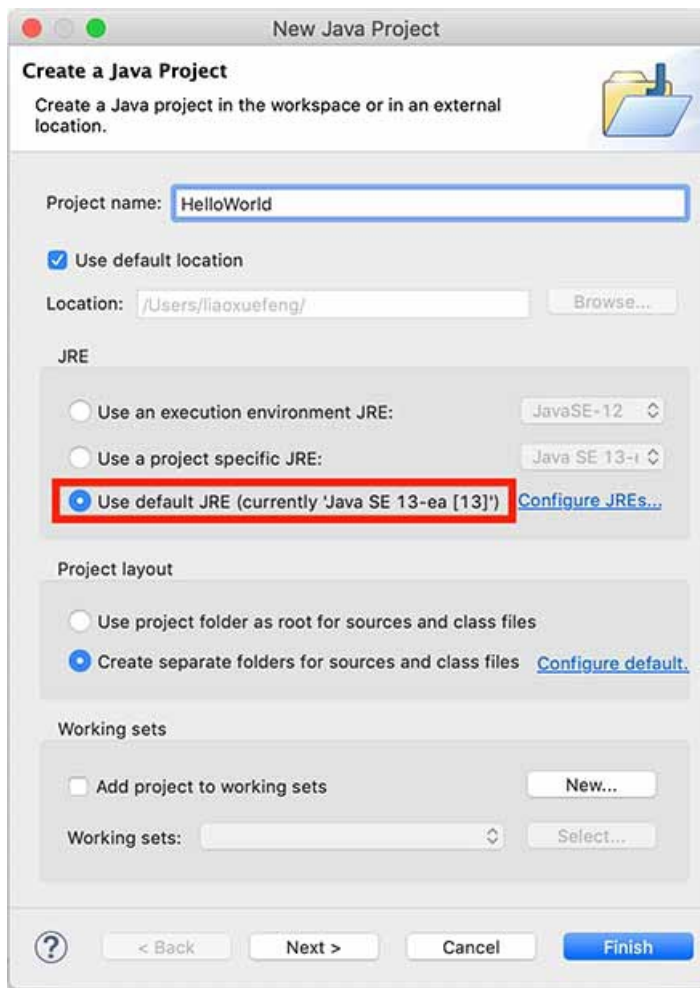
打开Eclipse后，整个IDE由若干个区域组成：



- 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。

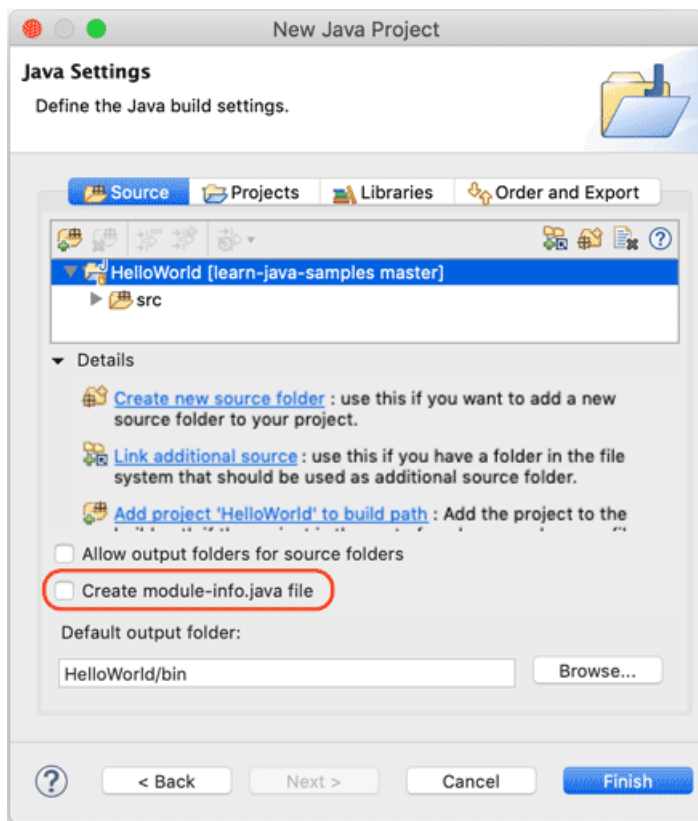
## 新建Java项目

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：

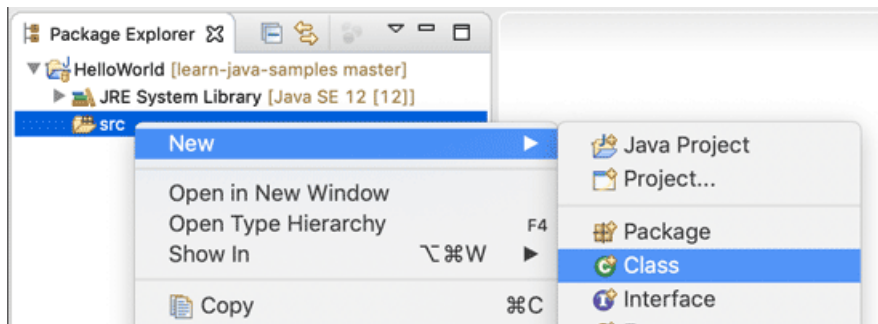




点击“Finish”就成功创建了一个名为HelloWorld的Java工程。


## 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：

**New Java Class**

 The use of the default package is discouraged.

Source folder: HelloWorld/src Browse...

Package:  (default) Browse...

☐ Enclosing type:  Browse...

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected  
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces:  Add...

Which method stubs would you like to create?

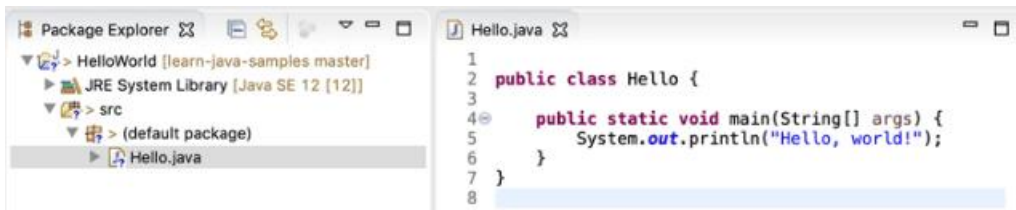
☐ public static void main(String[] args)  
☐ Constructors from superclass  
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

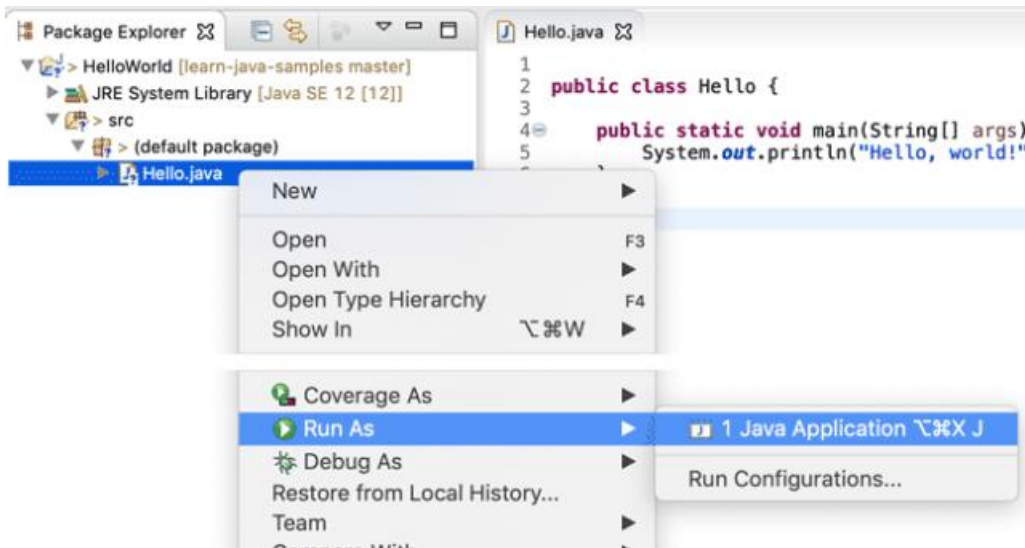
? Cancel Finish

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：



```
1 public class Hello {  
2  
3  
4     public static void main(String[] args) {  
5         System.out.println("Hello, world!");  
6     }  
7 }  
8
```

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：



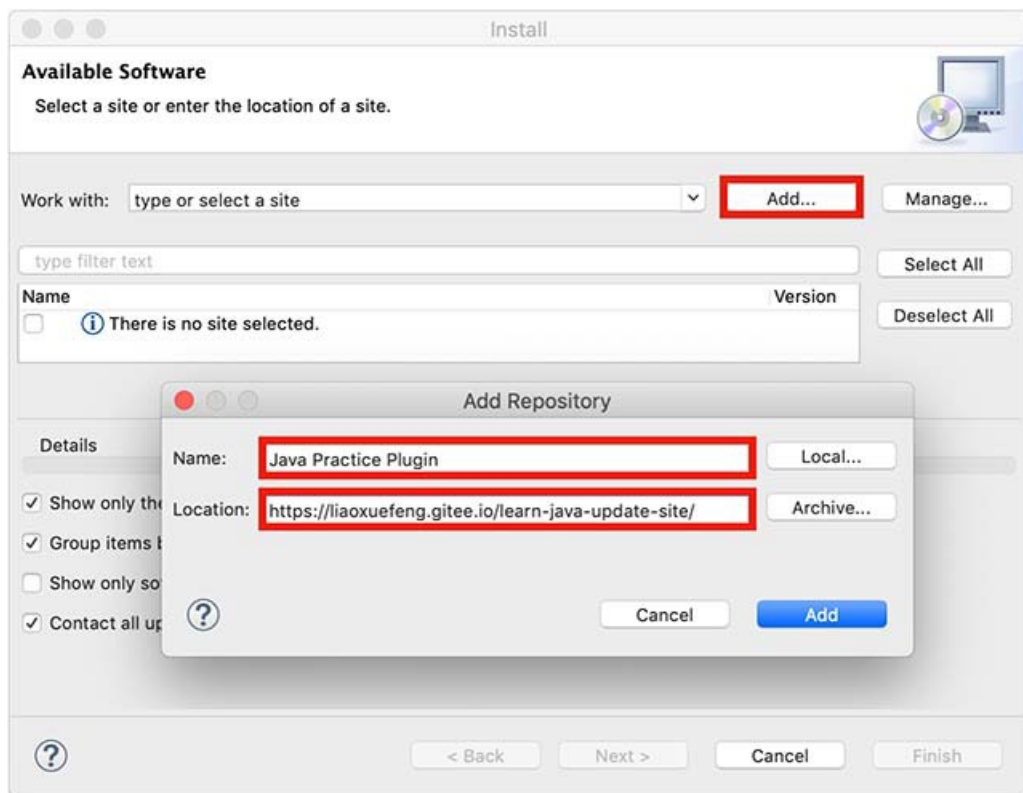
如果没有在主界面中看到Console窗口，请选中菜单“Window”-“Show View”-“Console”，即可显示。

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。

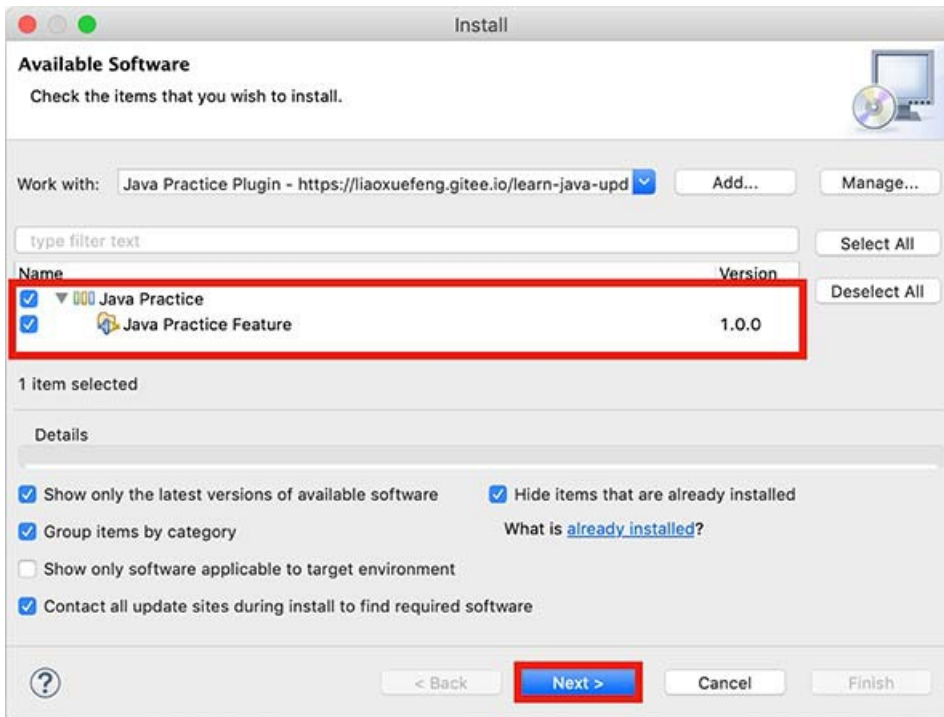
## 使用IDE练习插件

### 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://liaoxuefeng.gitee.io/learn-java-update-site/>，然后点击“Add”添加：



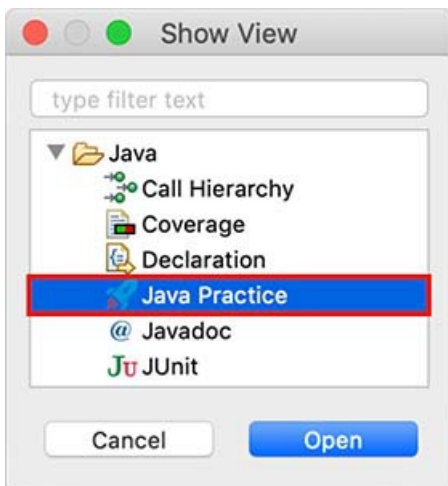
在列表中选中“Java Practice Feature”，然后点击“Next”安装。

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：

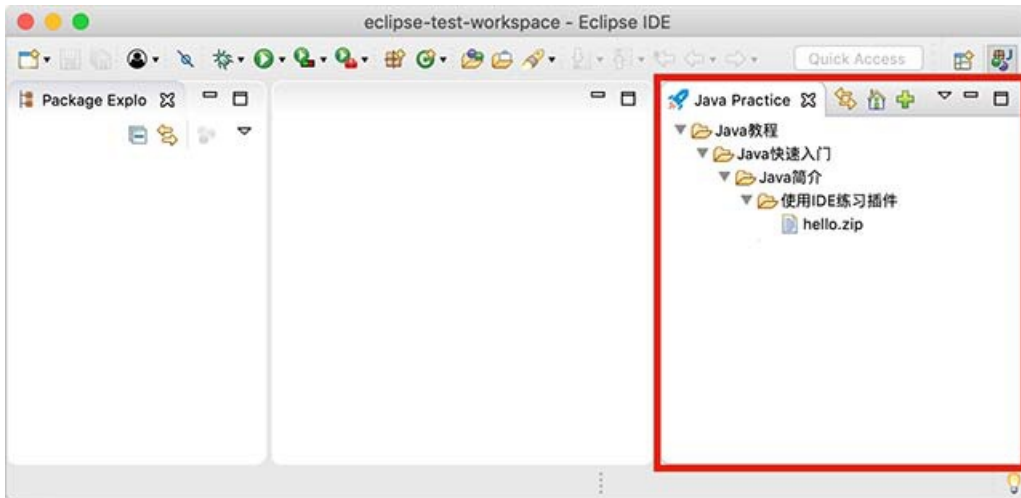


选择“Install anyway”继续安装，安装成功后，根据提示重启Eclipse即可。

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

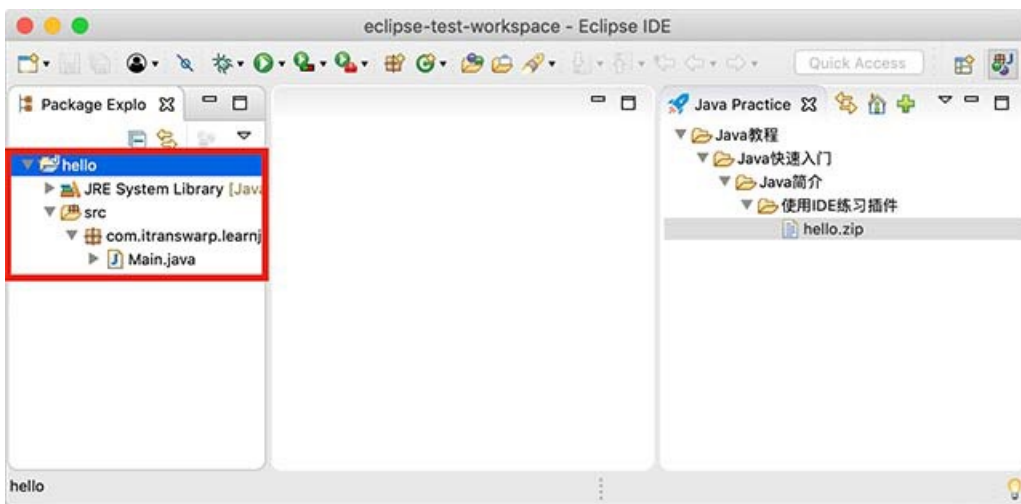


在弹出的对话框中选择“Java”-“Java Practice”，然后点击“Open”，即可在Eclipse中看到Java Practice插件：



## 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？

本节我们将介绍Java程序的基础知识，包括：

## Java程序基础

- 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



我们先剖析一个完整的Java程序，它的基本结构是什么：

## Java程序基本结构

```
/**
 * 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，可以有多行：

```
/*
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

# 变量和数据类型

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：

```
// 定义并打印变量
----
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：

```
// 重新赋值变量
----
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：

```
// 变量之间的赋值
----
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）
    }
}
```

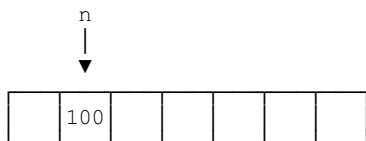
```

        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?
    }
}

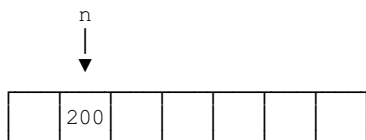
```

我们一行一行地分析代码执行流程：

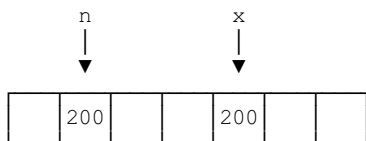
执行 `int n = 100;`，该语句定义了变量 `n`，同时赋值为100，因此，JVM在内存中为变量 `n` 分配一个“存储单元”，填入值100：



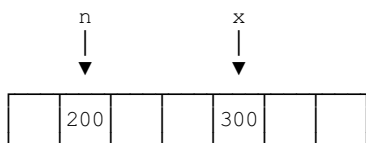
执行 `n = 200;` 时，JVM把200写入变量 `n` 的存储单元，因此，原有的值被覆盖，现在 `n` 的值为200：



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

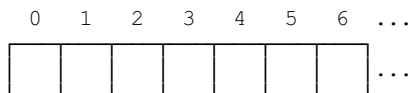
基本数据类型是CPU可以直接进行运算的类型。Java定义了以下几种基本数据类型：

- 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。

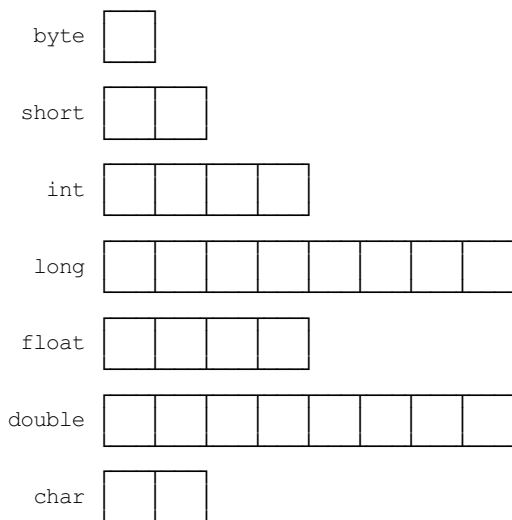
内存单元从0开始编号，称为内存地址。每个内存单元可以看作一间房间，内存地址就是门牌号。



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：

```
// 定义整型
----
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 $12.345 \times 10^2$ ，也可以表示成 $1.2345 \times 10^3$ ，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 $3.14 \times 10^{38}$ 
double d = 1.79e308;
double d2 = -1.79e308;
double d3 = 4.9e-324; // 科学计数法表示的 $4.9 \times 10^{-324}$ 
```

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 $3.4 \times 10^{38}$ ，而double类型可最大表示 $1.79 \times 10^{308}$ 。

## 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：

```
// 字符类型
----
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。

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：

## 整数运算

```
// 四则运算
-----
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：

```
// 运算溢出
-----
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 1000
+ 0000 0000 0000 0000 0000 0000 1111
-----
  1000 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：

```
// 自增/自减运算
----
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
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
```

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
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;     // 10000000 00000000 00000000 00000001 <= -2
int d = n >> 29;     // 10000000 00000000 00000000 00000000 <= -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×2，右移实际上就是不断地÷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：

```
// 位运算
----
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：

```
// 类型自动提升与强制转型
----
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：

```

// 强制转型
----
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");
    }
}

```

## [计算前N个自然数的和](#)

## 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。

## 浮点数运算

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：

```
// 浮点数运算误差
----
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是整型，那么整型可以自动提升到浮点型：

```
// 类型提升
----
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：

```
// 四舍五入
----
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");
    }
}
```

[计算一元二次方程的两个解](#)

## 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。

对于布尔类型boolean，永远只有true和false两个值。

## 布尔运算

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
```

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 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
----
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");
        ----
    }
}
```

[判断指定年龄是否是小学生](#)

## 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`。



在Java中，字符和字符串是两个不同的类型。

# 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：

```
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：

```
// 字符串连接
```

```

----
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：

```

// 字符串连接
----
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：

```

// 多行字符串
----
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
```

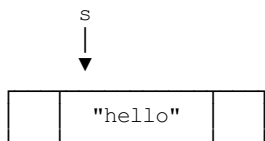
## 不可变特性

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：

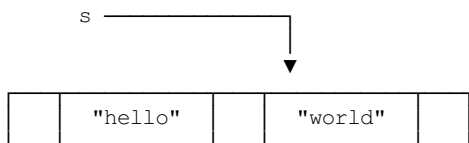
```
// 字符串不可变
----
public class Main {
    public static void main(String[] args) {
        String s = "hello";
        System.out.println(s); // 显示 hello
        s = "world";
        System.out.println(s); // 显示 world
    }
}
```

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：

```
// 字符串不可变
----
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);
        ----
    }
}
```

[Unicode值拼接字符串](#)

## 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""。

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：

## 数组类型

```
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：

```
// 数组
----
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;。

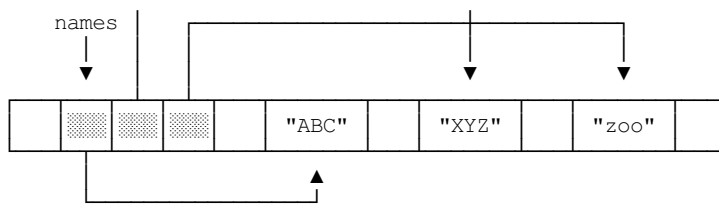
可以用数组变量.length获取数组大小：

```
// 数组
----
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
    }
}
```

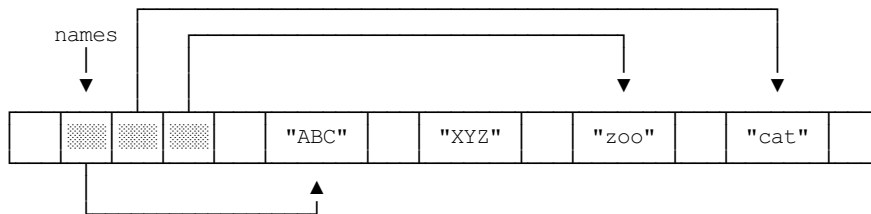
数组是引用类型，在使用索引访问数组元素时，如果索引超出范围，运行时将报错：

```
// 数组
----
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
    }
}
```





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：

```
// 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

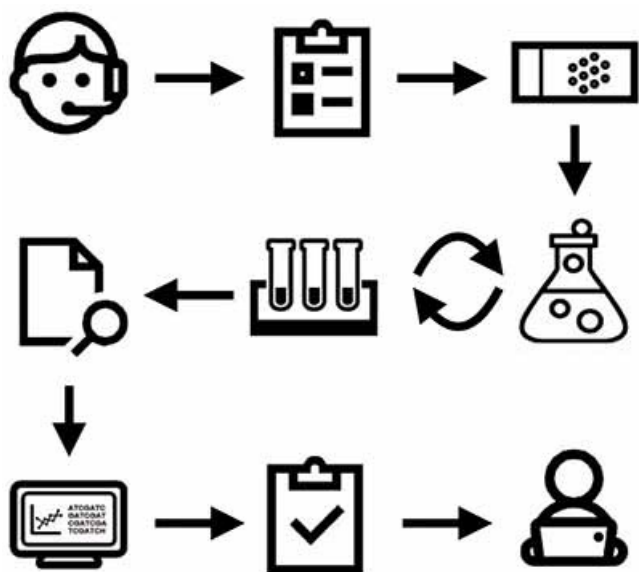
数组是同一数据类型的集合，数组一旦创建后，大小就不可变；

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；

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。

## 流程控制



本节我们将介绍if条件判断、switch多重选择和各种循环语句。



## 输出

# 输入和输出

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()`：

```
// 输出
----
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：

```
// 格式化输出
----
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：

```
// 格式化输出
----
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
<code>%d</code>	格式化输出整数
<code>%x</code>	格式化输出十六进制整数
<code>%f</code>	格式化输出浮点数
<code>%e</code>	格式化输出科学计数法表示的浮点数
<code>%s</code>	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：

```
// 格式化输出
```

```

----
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%）。

### [输入和输出练习](#)

## 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() / ...

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判断

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：

```
// 条件判断  
----  
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：

```
// 条件判断  
----  
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{}：

```
// 条件判断  
----  
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：

```
// 条件判断  
----  
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：

```

// 条件判断
----
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：

```

// 条件判断
----
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：

```

// 条件判断
----
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：

```

// 条件判断
----
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：

```

// 条件判断
----
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：

```
// 条件判断
----
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：

```
// 条件判断
----
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：

```
// 条件判断
----
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：

```
// 条件判断
----
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&&：

```
// 条件判断
----
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

## [BMI练习](#)

## 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。

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。

## switch多重选择

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：

```
// switch
----
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：

```
// switch
----
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：

```

// switch
----
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：

```

// switch
----
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：

```
// switch
----
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。

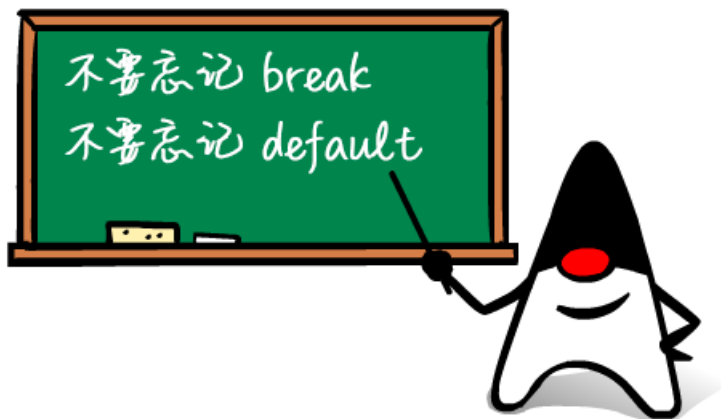
在Eclipse中，选择Preferences - Java - Compiler - Errors/Warnings - Potential programming problems，将以下检查标记为Warning:

- '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：

```
// switch
----
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：

```
// switch
----
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：

```
// yield
----
```

```

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。

[switch练习](#)

## 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。

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

例如，计算从1到100的和：

$1 + 2 + 3 + 4 + \dots + 100 = ?$

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：

```
// while  
----  
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：

```
// while  
----  
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：

```
// while  
----  
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：

```

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

```

## [while练习](#)

## 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。

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 while循环

```
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：

```
// do-while  
----  
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。

```
// do while  
----  
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);  
    }  
}
```

[do while练习](#)

### 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。

除了while和do while循环，Java使用最广泛的是for循环。

## 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：

```
// for
----
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：

```
// for
----
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<ns.length。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：

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



```

    }
}

```

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：

```
// for each
----
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：

```
// for
----
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：

```
// for each
----
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 $\pi$ 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 $\pi$ ：

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

```
    }  
    System.out.println(pi);  
  }  
}
```

### [for循环计算 \$\pi\$ 练习](#)

## 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；

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和continue

### 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：

```
// break
----
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：

```
// break
----
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("brokeed");
        }
    }
}
```

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：

```
// continue
----
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。

本节我们将讲解对数组的操作，包括：

## 数组操作

- 遍历；
- 排序。

以及多维数组的概念。



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。

## 遍历数组

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：

```
// 遍历数组
----
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：

```
// 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);
        }
    }
}
```

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：

```
// 遍历数组
----
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(??);  
        }  
    }  
}
```

[倒序遍历数组练习](#)

## 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。



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。

## 数组排序

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：

```
// 冒泡排序
----
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：

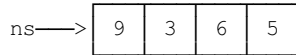
```
// 排序
----
import java.util.Arrays;

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));
    }
}
```

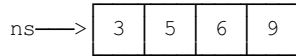
必须注意，对数组排序实际上修改了数组本身。例如，排序前的数组是：

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

在内存中，这个整型数组表示如下：



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

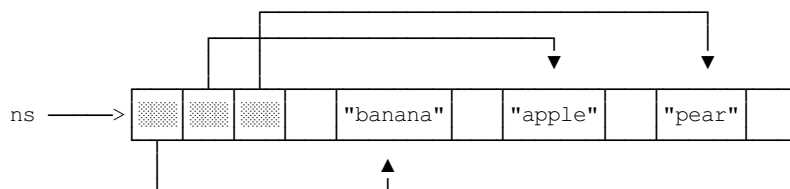


即变量`ns`指向的数组内容已经被改变了。

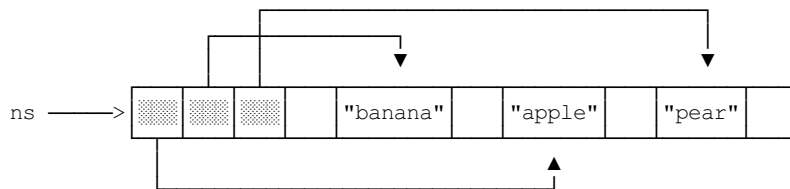
如果对一个字符串数组进行排序，例如：

```
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");
        }
    }
}
```

[降序排序练习](#)

## 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。

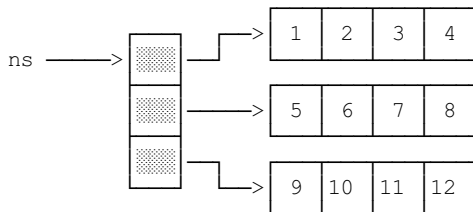
## 二维数组

# 多维数组

二维数组就是数组的数组。定义一个二维数组如下：

```
// 二维数组
-----
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
    }
}
```

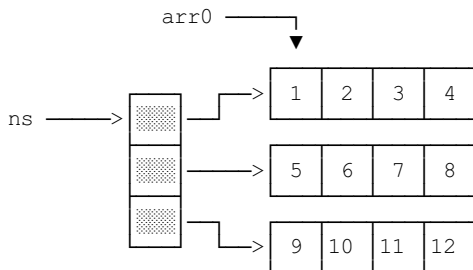
因为ns包含3个数组，因此，ns.length为3。实际上ns在内存中的结构如下：



如果我们定义一个普通数组arr0，然后把ns[0]赋值给它：

```
// 二维数组
-----
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：



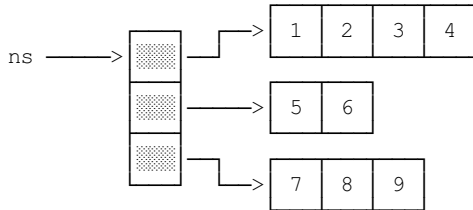
访问二维数组的某个元素需要使用array[row][col]，例如：

```
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()`：

```
// 二维数组
----
import java.util.Arrays;

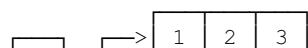
public class Main {
    public static void main(String[] args) {
        int[][] ns = {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 }
        };
        System.out.println(Arrays.deepToString(ns));
    }
}
```

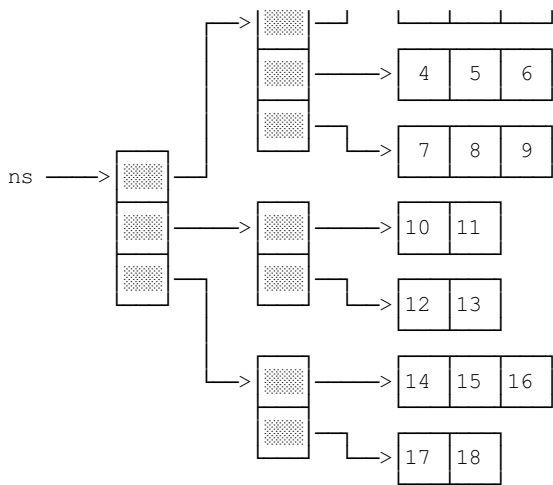
## 三维数组

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");
        }
    }
}
```

## 计算平均分

## 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]`。

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。

## 命令行参数

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。

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

## 面向对象编程

那什么是面向对象编程？

和面向对象编程不同的，是面向过程编程。面向过程编程，是把模型分解成一步一步的过程。比如，老板告诉你，要编写一个TODO任务，必须按照以下步骤一步一步来：

1. 读取文件；
2. 编写TODO；
3. 保存文件。



而面向对象编程，顾名思义，你得首先有个对象：



有了对象后，就可以和对象进行互动：

```
GirlFriend gf = new GirlFriend();  
gf.name = "Alice";  
gf.send("flowers");
```



因此，面向对象编程，是一种通过对象的方式，把现实世界映射到计算机模型的一种编程方法。

在本章中，我们将讨论：

面向对象的基本概念，包括：

- 类
- 实例
- 方法

面向对象的实现方式，包括：

- 继承
- 多态

Java语言本身提供的机制，包括：

- package
- classpath
- jar

以及Java标准库提供的核心类，包括：

- 字符串
- 包装类型
- JavaBean
- 枚举
- 常用工具类

通过本章的学习，完全可以理解并掌握面向对象编程的基本思想。



面向对象编程，是一种通过对象的方式，把现实世界映射到计算机模型的一种编程方法。

## 面向对象基础

现实世界中，我们定义了“人”这种抽象概念，而具体的人则是“小明”、“小红”、“小军”等一个个具体的人。所以，“人”可以定义为一个类（**class**），而具体的人则是实例（**instance**）：

现实世界 计算机模型 **Java**代码

人	类 / class	<code>class Person { }</code>
小明	实例 / <code>ming</code>	<code>Person ming = new Person()</code>
小红	实例 / <code>hong</code>	<code>Person hong = new Person()</code>
小军	实例 / <code>jun</code>	<code>Person jun = new Person()</code>

同样的，“书”也是一种抽象的概念，所以它是类，而《Java核心技术》、《Java编程思想》、《Java学习笔记》则是实例：

现实世界 计算机模型 **Java**代码

书	类 / class	<code>class Book { }</code>
Java核心技术	实例 / <code>book1</code>	<code>Book book1 = new Book()</code>
Java编程思想	实例 / <code>book2</code>	<code>Book book2 = new Book()</code>
Java学习笔记	实例 / <code>book3</code>	<code>Book book3 = new Book()</code>

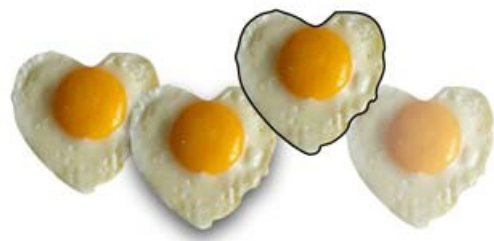
### **class**和**instance**

所以，只要理解了**class**和**instance**的概念，基本上就明白了什么是面向对象编程。

**class**是一种对象模版，它定义了如何创建实例，因此，**class**本身就是一种数据类型：



而**instance**是对象实例，**instance**是根据**class**创建的实例，可以创建多个**instance**，每个**instance**类型相同，但各自属性可能不相同：



### 定义**class**

在**Java**中，创建一个类，例如，给这个类命名为**Person**，就是定义一个**class**：

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

一个class可以包含多个字段（field），字段用来描述一个类的特征。上面的Person类，我们定义了两个字段，一个是String类型的字段，命名为name，一个是int类型的字段，命名为age。因此，通过class，把一组数据汇集到一个对象上，实现了数据封装。

public是用来修饰字段的，它表示这个字段可以被外部访问。

我们再看另一个Book类的定义：

```
class Book {  
    public String name;  
    public String author;  
    public String isbn;  
    public double price;  
}
```

请指出Book类的各个字段。

## 创建实例

定义了class，只是定义了对对象模版，而要根据对象模版创建出真正的对象实例，必须用new操作符。

new操作符可以创建一个实例，然后，我们需要定义一个引用类型的变量来指向这个实例：

```
Person ming = new Person();
```

上述代码创建了一个Person类型的实例，并通过变量ming指向它。

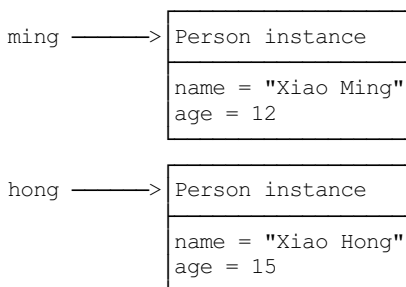
注意区分Person ming是定义Person类型的变量ming，而new Person()是创建Person实例。

有了指向这个实例的变量，我们就可以通过这个变量来操作实例。访问实例变量可以用变量.字段，例如：

```
ming.name = "Xiao Ming"; // 对字段name赋值  
ming.age = 12; // 对字段age赋值  
System.out.println(ming.name); // 访问字段name
```

```
Person hong = new Person();  
hong.name = "Xiao Hong";  
hong.age = 15;
```

上述两个变量分别指向两个不同的实例，它们在内存中的结构如下：



两个instance拥有class定义的名字和age字段，且各自都有一份独立的数据，互不干扰。

## 练习

请定义一个City类，该class具有如下字段：

- name: 名称，String类型
- latitude: 纬度，double类型
- longitude: 经度，double类型

实例化几个City并赋值，然后打印。

```
// City.java
----
public class Main {
    public static void main(String[] args) {
        City bj = new City();
        bj.name = "Beijing";
        bj.latitude = 39.903;
        bj.longitude = 116.401;
        System.out.println(bj.name);
        System.out.println("location: " + bj.latitude + ", " + bj.longitude);
    }
}

class City {
    ???
}
```

## 小结

在OOP中，class和instance是“模版”和“实例”的关系；

定义class就是定义了一种数据类型，对应的instance是这种数据类型的实例；

class定义的field，在每个instance都会拥有各自的field，且互不干扰；

通过new操作符创建新的instance，然后用变量指向它，即可通过变量来引用这个instance；

访问实例字段的方法是变量名.字段名；

指向instance的变量都是引用变量。

一个class可以包含多个field，例如，我们给Person类就定义了两个field：

## 方法

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

但是，直接把field用public暴露给外部可能会破坏封装性。比如，代码可以这样写：

```
Person ming = new Person();  
ming.name = "Xiao Ming";  
ming.age = -99; // age设置为负数
```

显然，直接操作field，容易造成逻辑混乱。为了避免外部代码直接去访问field，我们可以用private修饰field，拒绝外部访问：

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

试试private修饰的field有什么效果：

```
// private field  
----  
public class Main {  
    public static void main(String[] args) {  
        Person ming = new Person();  
        ming.name = "Xiao Ming"; // 对字段name赋值  
        ming.age = 12; // 对字段age赋值  
    }  
}  
  
class Person {  
    private String name;  
    private int age;  
}
```

是不是编译报错？把访问field的赋值语句去了就可以正常编译了。



把field从public改成private，外部代码不能访问这些field，那我们定义这些field有什么用？怎么才能给它赋值？怎么才能读取它的值？

所以我们需要使用方法（method）来让外部代码可以间接修改field：

```
// private field  
----  
public class Main {
```

```

    public static void main(String[] args) {
        Person ming = new Person();
        ming.setName("Xiao Ming"); // 设置name
        ming.setAge(12); // 设置age
        System.out.println(ming.getName() + ", " + ming.getAge());
    }
}

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;
    }
}

```

虽然外部代码不能直接修改`private`字段，但是，外部代码可以调用方法`setName()`和`setAge()`来间接修改`private`字段。在方法内部，我们就有机会检查参数对不对。比如，`setAge()`就会检查传入的参数，参数超出了范围，直接报错。这样，外部代码就没有任何机会把`age`设置成不合理的值。

对`setName()`方法同样可以做检查，例如，不允许传入`null`和空字符串：

```

public void setName(String name) {
    if (name == null || name.isBlank()) {
        throw new IllegalArgumentException("invalid name");
    }
    this.name = name.strip(); // 去掉首尾空格
}

```

同样，外部代码不能直接读取`private`字段，但可以通过`getName()`和`getAge()`间接获取`private`字段的值。

所以，一个类通过定义方法，就可以给外部代码暴露一些操作的接口，同时，内部自己保证逻辑一致性。

调用方法的语法是实例变量.方法名(参数);。一个方法调用就是一个语句，所以不要忘了在末尾加;。例如：`ming.setName("Xiao Ming");`。

## 定义方法

从上面的代码可以看出，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`。

## private方法

有`public`方法，自然就有`private`方法。和`private`字段一样，`private`方法不允许外部调用，那我们定义`private`方法

有什么用？

定义private方法的理由是内部方法是可以调用private方法的。例如：

```
// private method
----
public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        ming.setBirth(2008);
        System.out.println(ming.getAge());
    }
}

class Person {
    private String name;
    private int birth;

    public void setBirth(int birth) {
        this.birth = birth;
    }

    public int getAge() {
        return calcAge(2019); // 调用private方法
    }

    // private方法:
    private int calcAge(int currentYear) {
        return currentYear - this.birth;
    }
}
```

观察上述代码，calcAge()是一个private方法，外部代码无法调用，但是，内部方法getAge()可以调用它。

此外，我们还注意到，这个Person类只定义了birth字段，没有定义age字段，获取age时，通过方法getAge()返回的是一个实时计算的值，并非存储在某个字段的值。这说明方法可以封装一个类的对外接口，调用方不需要知道也不关心Person实例在内部到底有没有age字段。

## 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
```

完全可以把可变参数改写为String[]类型：

```
class Group {
    private String[] names;

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

但是，调用方需要自己先构造String[]，比较麻烦。例如：

```
Group g = new Group();
g.setNames(new String[] { "Xiao Ming", "Xiao Hong", "Xiao Jun" }); // 传入1个String[]
```

另一个问题是，调用方可以传入null：

```
Group g = new Group();
g.setNames(null);
```

而可变参数可以保证无法传入null，因为传入0个参数时，接收到的实际值是一个空数组而不是null。

## 参数绑定

调用方把参数传递给实例方法时，调用时传递的值会按参数位置一一绑定。

那什么是参数绑定？

我们先观察一个基本类型参数的传递：

```
// 基本类型参数绑定
----
```



```

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;
    }
}

```

运行代码，从结果可知，修改外部的局部变量n，不影响实例p的age字段，原因是setAge()方法获得的参数，复制了n的值，因此，p.age和局部变量n互不影响。

结论：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。

我们再看一个传递引用参数的例子：

```

// 引用类型参数绑定
----
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;
    }
}

```

注意到setName()的参数现在是一个数组。一开始，把fullname数组传进去，然后，修改fullname数组的内容，结果发现，实例p的字段p.name也被修改了！

结论：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）。

有了上面的结论，我们再看一个例子：

```

// 引用类型参数绑定
----
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        String bob = "Bob";
    }
}

```

```

        p.setName(bob); // 传入bob变量
        System.out.println(p.getName()); // "Bob"
        bob = "Alice"; // bob改名为Alice
        System.out.println(p.getName()); // "Bob"还是"Alice"?
    }
}

class Person {
    private String name;

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

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

```

不要怀疑引用参数绑定的机制，试解释为什么上面的代码两次输出都是"Bob"。

## 练习

```

public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        ming.setName("小明");
        ming.setAge(12);
        System.out.println(ming.getAge());
    }
}
----
class Person {
    private String name;

    public String getName() {
        return name;
    }

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

```

[给Person类增加getAge/setAge方法](#)

## 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。

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：

## 构造方法

```
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：

```
// 构造方法
----
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：

```
// 构造方法
----
public class Main {
    public static void main(String[] args) {
        Person p = new Person(); // 编译错误:找不到这个构造方法
    }
}

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;
    }
}
```

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：

```
// 构造方法
----
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：

```
class Person {
    private String name; // 默认初始化为null
    private int age; // 默认初始化为0

    public Person() {
    }
}
```

也可以对字段直接进行初始化：

```
class Person {
    private String name = "Unnamed";
    private int age = 10;
}
```

那么问题来了：既对字段进行初始化，又在构造方法中对字段进行初始化：

```
class Person {
    private String name = "Unnamed";
    private int age = 10;

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

当我们创建对象的时候，`new Person("Xiao Ming", 12)`得到的对象实例，字段的初始值是啥？

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，`new Person("Xiao Ming", 12)`的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)
    }
}

```

## 练习

请给Person类增加(String, int)的构造方法:

```

public class Main {
    public static void main(String[] args) {
        // TODO: 给Person增加构造方法:
        Person ming = new Person("小明", 12);
        System.out.println(ming.getName());
        System.out.println(ming.getAge());
    }
}
-----
class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

```

[给Person类增加\(String, int\)的构造方法](#)

## 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空参数构造方法；

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。

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：

## 方法重载

```
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：

```
// String.indexOf()
----
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);
    }
}
```

## 练习

```
public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        Person hong = new Person();
        ming.setName("Xiao Ming");
        // TODO: 给Person增加重载方法setName(String, String):
        hong.setName("Xiao", "Hong");
        System.out.println(ming.getName());
    }
}
```

```
        System.out.println(hong.getName());
    }
}
----
class Person {
    private String name;

    public String getName() {
        return name;
    }

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

[给Person增加重载方法](#)

## 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；

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。



在前面的章节中，我们已经定义了Person类：

## 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) {...}
}
```

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;

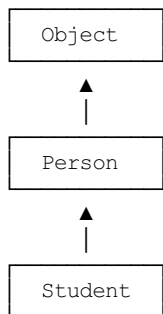
    public int getScore() { ... }
    public void setScore(int score) { ... }
}
```

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。

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）。

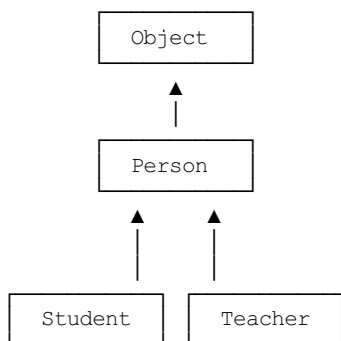
## 继承树

注意到我们在定义Person的时候，没有写extends。在Java中，没有明确写extends的类，编译器会自动加上extends Object。所以，任何类，除了Object，都会继承自某个类。下图是Person、Student的继承树：



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。

类似的，如果我们定义一个继承自Person的Teacher，它们的继承树关系如下：



## 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，后面我们还会详细讲解。

## 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;
    }
}
```

实际上，这里使用`super.name`，或者`this.name`，或者`name`，效果都是一样的。编译器会自动定位到父类的`name`字段。

但是，在某些时候，就必须使用`super`。我们来看一个例子：

```
// super
----
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Xiao Ming", 12, 89);
    }
}

class Person {
    protected String name;
    protected int age;

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

class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        this.score = score;
    }
}
```

运行上面的代码，会得到一个编译错误，大意是在`Student`的构造方法中，无法调用`Person`的构造方法。

这是因为在`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();`，所以，`Student`类的构造方法实际上是这样：

```
class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(); // 自动调用父类的构造方法
        this.score = score;
    }
}
```

但是，`Person`类并没有无参数的构造方法，因此，编译失败。

解决方法是调用`Person`类存在的某个构造方法。例如：

```
class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(name, age); // 调用父类的构造方法Person(String, int)
        this.score = score;
    }
}
```

```
}
```

这样就可以正常编译了！

因此我们得出结论：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。

## 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
}

```

## 区分继承和组合

在使用继承时，我们要注意逻辑一致性。

考察下面的Book类：

```

class Book {
    protected String name;
    public String getName() {...}
    public void setName(String name) {...}
}

```

这个Book类也有name字段，那么，我们能不能让Student继承自Book呢？

```

class Student extends Book {
    protected int score;
}

```

显然，从逻辑上讲，这是不合理的，Student不应该从Book继承，而应该从Person继承。

究其原因，是因为Student是Person的一种，它们是is关系，而Student并不是Book。实际上Student和Book的关系是has关系。

具有has关系不应该使用继承，而是使用组合，即Student可以持有一个Book实例：

```

class Student extends Person {
    protected Book book;
    protected int score;
}

```

因此，继承是is关系，组合是has关系。

## 练习

定义PrimaryStudent，从Student继承，并新增一个grade字段：

```

public class Main {
    public static void main(String[] args) {
        Person p = new Person("小明", 12);
        Student s = new Student("小红", 20, 99);
        // TODO: 定义PrimaryStudent，从Student继承，新增grade字段：
        Student ps = new PrimaryStudent("小军", 9, 100, 5);
        System.out.println(ps.getScore());
    }
}

class Person {
    protected String name;
}

```

```

    protected int age;

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

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

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

class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(name, age);
        this.score = score;
    }

    public int getScore() { return score; }
}
----
class PrimaryStudent {
    // TODO
}

```

## [继承练习](#)

## 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。

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）。

## 多态

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，如果方法签名如果不同，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**。

注意：方法名相同，方法参数相同，但方法返回值不同，也是不同的方法。在Java程序中，出现这种情况，编译器会报错。

```
class Person {
    public void run() { ... }
}

class Student extends Person {
    // 不是Override, 因为参数不同:
    public void run(String s) { ... }
    // 不是Override, 因为返回值不同:
    public int run() { ... }
}
```

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

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

class Person {
    public void run() {}
}

public class Student extends Person {
    @Override // Compile error!
    public void run(String s) {}
}
```

但是@Override不是必需的。

在上一节中，我们已经知道，引用变量的声明类型可能与其实际类型不符，例如：

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

现在，我们考虑一种情况，如果子类覆写了父类的方法：

```
// override
----
public class Main {
```

```

    public static void main(String[] args) {
        Person p = new Student();
        p.run(); // 应该打印Person.run还是Student.run?
    }
}

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

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

```

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，实际上调用的方法是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。它的英文拼写非常复杂：**Polymorphic**。

## 多态

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：

```

Person p = new Student();
p.run(); // 无法确定运行时究竟调用哪个run()方法

```

有童鞋会问，从上面的代码一看就明白，肯定调用的是Student的run()方法啊。

但是，假设我们编写这样一个方法：

```

public void runTwice(Person p) {
    p.run();
    p.run();
}

```

它传入的参数类型是Person，我们是无法知道传入的参数实际类型究竟是Person，还是Student，还是Person的其他子类，因此，也无法确定调用的是不是Person类定义的run()方法。

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。这种不确定性的方法调用，究竟有什么作用？

我们还是来举栗子。

假设我们定义一种收入，需要给它报税，那么先定义一个Income类：

```

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

```

对于工资收入，可以减去一个基数，那么我们可以从Income派生出SalaryIncome，并覆写getTax()：

```

class Salary extends Income {
    @Override
    public double getTax() {

```



```

        if (income <= 5000) {
            return 0;
        }
        return (income - 5000) * 0.2;
    }
}

```

如果你享受国务院特殊津贴，那么按照规定，可以全部免税：

```

class StateCouncilSpecialAllowance extends Income {
    @Override
    public double getTax() {
        return 0;
    }
}

```

现在，我们要编写一个报税的财务软件，对于一个人的所有收入进行报税，可以这么写：

```

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

```

来试一下：

```

// Polymorphic
----
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。

可见，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。

## 覆写Object方法

因为所有的class最终都继承自Object，而Object定义了几个重要的方法：

- toString()：把instance输出为String；
- equals()：判断两个instance是否逻辑相等；
- hashCode()：计算一个instance的哈希值。

在必要的情况下，我们可以覆写Object的这几个方法。例如：

```

class Person {
    ...
    // 显示更有意义的字符串：
    @Override
    public String toString() {
        return "Person:name=" + name;
    }

    // 比较是否相等：
    @Override
    public boolean equals(Object o) {
        // 当且仅当o为Person类型：
        if (o instanceof Person) {
            Person p = (Person) o;
            // 并且name字段相同时，返回true：
            return this.name.equals(p.name);
        }
        return false;
    }

    // 计算hash：
    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
}

```

## 调用super

在子类的覆写方法中，如果要调用父类的被覆写的方法，可以通过super来调用。例如：

```

class Person {
    protected String name;
    public String hello() {

```

```

        return "Hello, " + name;
    }
}

Student extends Person {
    @Override
    public String hello() {
        // 调用父类的hello()方法:
        return super.hello() + "!";
    }
}

```

## final

继承可以允许子类覆写父类的方法。如果一个父类不允许子类对它的某个方法进行覆写，可以把该方法标记为`final`。用`final`修饰的方法不能被`Override`：

```

class Person {
    protected String name;
    public final String hello() {
        return "Hello, " + name;
    }
}

Student extends Person {
    // compile error: 不允许覆写
    @Override
    public String hello() {
    }
}

```

如果一个类不希望任何其他类继承自它，那么可以把这个类本身标记为`final`。用`final`修饰的类不能被继承：

```

final class Person {
    protected String name;
}

// compile error: 不允许继承自Person
Student extends Person {
}

```

对于一个类的实例字段，同样可以用`final`修饰。用`final`修饰的字段在初始化后不能被修改。例如：

```

class Person {
    public final String name = "Unnamed";
}

```

对`final`字段重新赋值会报错：

```

Person p = new Person();
p.name = "New Name"; // compile error!

```

可以在构造方法中初始化`final`字段：

```

class Person {
    public final String name;
    public Person(String name) {
        this.name = name;
    }
}

```

这种方法更为常用，因为可以保证实例一旦创建，其`final`字段就不可修改。

## 练习

给一个有工资收入和稿费收入的小伙伴算税。

## [计算所得税](#)

### 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。

由于多态的存在，每个子类都可以覆写父类的方法，例如：

## 抽象类

```
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。

能不能去掉父类的run()方法？

答案还是不行，因为去掉父类的run()方法，就失去了多态的特性。例如，runTwice()就无法编译：

```
public void runTwice(Person p) {
    p.run(); // Person没有run()方法，会导致编译错误
    p.run();
}
```

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：

```
// abstract class
----
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。

## 练习

用抽象类给一个有工资收入和稿费收入的小伙伴算税。

[用抽象类算税](#)

## 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。

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。

## 接口

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
    ...
}
```

## 术语

注意区分术语：

Java的接口特指interface的定义，表示一个接口类型和一组方法签名，而编程接口泛指接口规范，如方法签名，数据格式，网络协议等。

抽象类和接口的对比如下：

	<b>abstract class</b>	<b>interface</b>
继承	只能extends一个class	可以implements多个interface



	abstract class	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：

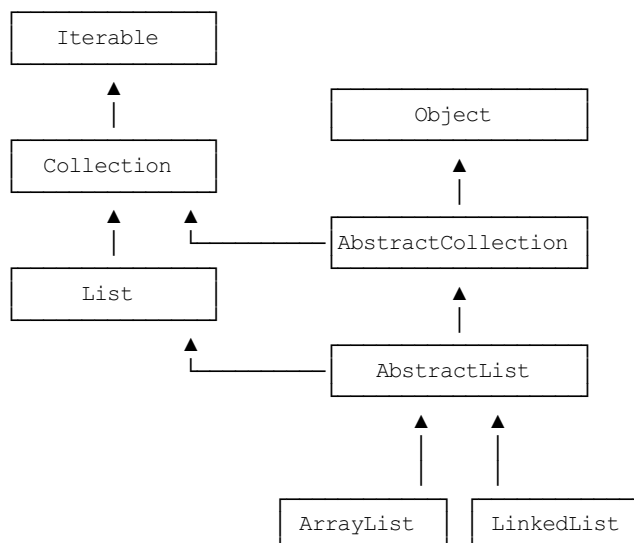
```
interface Hello {
    void hello();
}

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。

## 继承关系

合理设计interface和abstract class的继承关系，可以充分复用代码。一般来说，公共逻辑适合放在abstract class中，具体逻辑放到各个子类，而接口层次代表抽象程度。可以参考Java的集合类定义的一组接口、抽象类以及具体子类的继承关系：



在使用的时候，实例化的对象永远只能是某个具体的子类，但总是通过接口去引用它，因为接口比抽象类更抽象：

```
List list = new ArrayList(); // 用List接口引用具体子类的实例
Collection coll = list; // 向上转型为Collection接口
Iterable it = coll; // 向上转型为Iterable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：

```
// interface
----
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。

## 练习

用接口给一个有工资收入和稿费收入的小伙伴算税。

[用接口算税](#)

## 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 $\geq$ 1.8）。

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。

## 静态字段和静态方法

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

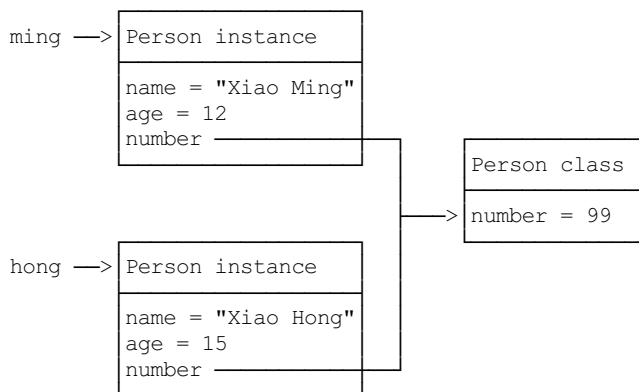
实例字段在每个实例中都有自己的一个独立“空间”，但是静态字段只有一个共享“空间”，所有实例都会共享该字段。举个例子：

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

我们来看看下面的代码：

```
// static field  
----  
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。

因此，不推荐用实例变量.静态字段去访问静态字段，因为在Java程序中，实例对象并没有静态字段。在代码中，实例对象能访问静态字段只是因为编译器可以根据实例类型自动转换为类名.静态字段来访问静态对象。

推荐用类名来访问静态字段。可以把静态字段理解为描述class本身的字段（非实例字段）。对于上面的代码，更好的写法是：

```
Person.number = 99;
System.out.println(Person.number);
```

## 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：

```
// static method
----
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。

## 练习

给Person类增加一个静态字段`count`和静态方法`getCount`，统计实例创建的个数。

[静态字段和静态方法](#)

## 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。

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。

## 包

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，小白既想用小明的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 {
}
```

小军的Arrays.java文件：

```
package mr.jun; // 申明包名mr.jun

public class Arrays {
}
```

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。

要特别注意：包没有父子关系。java.util和java.util.zip是不同的包，两者没有任何继承关系。

没有定义包名的class，它使用的是默认包，非常容易引起名字冲突，因此，不推荐不写包名的做法。

我们还需要按照包结构把上面的Java文件组织起来。假设以package\_sample作为根目录，src作为源码目录，那么所有文件结构就是：

```
package_sample
├── src
│   ├── hong
│   │   └── Person.java
│   ├── ming
│   │   └── Person.java
│   └── mr
│       └── jun
│           └── Arrays.java
```

即所有Java文件对应的目录层次要和包的层次一致。

编译后的.class文件也需要按照包结构存放。如果使用IDE，把编译后的.class文件放到bin目录下，那么，编译的文件结构就是：

```
package_sample
└─ bin
   └─ hong
      └─ Person.class
   └─ ming
      └─ Person.class
   └─ mr
      └─ jun
         └─ Arrays.class
```

编译的命令相对比较复杂，我们需要在src目录下执行javac命令：

```
javac -d ../bin ming/Person.java hong/Person.java mr/jun/Arrays.java
```

在IDE中，会自动根据包结构编译所有Java源码，所以不必担心使用命令行编译的复杂命令。

## 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!");
    }
}

```

import static很少使用。

Java编译器最终编译出的.class文件只使用完整类名，因此，在代码中，当编译器遇到一个class名称时：

- 如果是完整类名，就直接根据完整类名查找这个class；
- 如果是简单类名，按下面的顺序依次查找：
  - 查找当前package是否存在这个class；
  - 查找import的包是否包含这个class；
  - 查找java.lang包是否包含这个class。

如果按照上面的规则还无法确定类名，则编译报错。

我们来看一个例子：

```

// Main.java
package test;

import java.text.Format;

public class Main {
    public static void main(String[] args) {
        java.util.List list; // ok, 使用完整类名 -> java.util.List
        Format format = null; // ok, 使用import的类 -> java.text.Format
        String s = "hi"; // ok, 使用java.lang包的String -> java.lang.String
        System.out.println(s); // ok, 使用java.lang包的System -> java.lang.System
        MessageFormat mf = null; // 编译错误: 无法找到MessageFormat: MessageFormat cannot be resolved to a type
    }
}

```

因此，编写class的时候，编译器会自动帮我们做两个import动作：

- 默认自动import当前package的其他class；



- 默认自动import java.lang.\*。

注意：自动导入的是java.lang包，但类似java.lang.reflect这些包仍需要手动导入。

如果有两个class名称相同，例如，mr.jun.Arrays和java.util.Arrays，那么只能import其中一个，另一个必须写完整类名。

## 最佳实践

为了避免名字冲突，我们需要确定唯一的包名。推荐的做法是使用倒置的域名来确保唯一性。例如：

- org.apache
- org.apache.commons.log
- com.liaoxuefeng.sample

子包就可以根据功能自行命名。

要注意不要和java.lang包的类重名，即自己的类不要使用这些名字：

- String
- System
- Runtime
- ...

要注意也不要和JDK常用类重名：

- java.util.List
- java.text.Format
- java.math.BigInteger
- ...

## 练习

请按如下包结构创建工程项目：

```
oop-package
├── src
│   └── com
│       └── itranswarp
│           ├── sample
│           │   └── Main.java
│           └── world
│               └── Person.java
```

[Package结构](#)

## 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。

在Java中，我们经常看到public、protected、private这些修饰符。在Java中，这些修饰符可以用来限定访问作用域。

## 作用域

### public

定义为public的class、interface可以被其他任何类访问：

```
package abc;

public class Hello {
    public void hi() {
    }
}
```

上面的Hello是public，因此，可以被其他包的类访问：

```
package xyz;

class Main {
    void foo() {
        // Main可以访问Hello
        Hello h = new Hello();
    }
}
```

定义为public的field、method可以被其他类访问，前提是首先有访问class的权限：

```
package abc;

public class Hello {
    public void hi() {
    }
}
```

上面的hi()方法是public，可以被其他类调用，前提是首先要能访问Hello类：

```
package xyz;

class Main {
    void foo() {
        Hello h = new Hello();
        h.hi();
    }
}
```

### private

定义为private的field、method无法被其他类访问：

```
package abc;

public class Hello {
    // 不能被其他类调用：
    private void hi() {
    }

    public void hello() {
        this.hi();
    }
}
```

实际上，确切地说，private访问权限被限定在class的内部，而且与方法声明顺序无关。推荐把private方法放到后面，因为public方法定义了类对外提供的功能，阅读代码的时候，应该先关注public方法：

```
package abc;
```

```
public class Hello {
    public void hello() {
        this.hi();
    }

    private void hi() {
    }
}
```

由于Java支持嵌套类，如果一个类内部还定义了嵌套类，那么，嵌套类拥有访问private的权限：

```
// private
----
public class Main {
    public static void main(String[] args) {
        Inner i = new Inner();
        i.hi();
    }

    // private方法:
    private static void hello() {
        System.out.println("private hello!");
    }

    // 静态内部类:
    static class Inner {
        public void hi() {
            Main.hello();
        }
    }
}
```

定义在一个class内部的class称为嵌套类（nested class），Java支持好几种嵌套类。

## protected

protected作用于继承关系。定义为protected的字段和方法可以被子类访问，以及子类的子类：

```
package abc;

public class Hello {
    // protected方法:
    protected void hi() {
    }
}
```

上面的protected方法可以被继承的类访问：

```
package xyz;

class Main extends Hello {
    void foo() {
        Hello h = new Hello();
        // 可以访问protected方法:
        h.hi();
    }
}
```

## package

最后，包作用域是指一个类允许访问同一个package的没有public、private修饰的class，以及没有public、protected、private修饰的字段和方法。

```
package abc;
// package权限的类:
class Hello {
    // package权限的方法:
```

```

    void hi() {
    }
}

```

只要在同一包，就可以访问package权限的class、field和method:

```

package abc;

class Main {
    void foo() {
        // 可以访问package权限的类:
        Hello h = new Hello();
        // 可以调用package权限的方法:
        h.hi();
    }
}

```

注意，包名必须完全一致，包没有父子关系，com.apache和com.apache.abc是不同的包。

## 局部变量

在方法内部定义的变量称为局部变量，局部变量作用域从变量声明处开始到对应的块结束。方法参数也是局部变量。

```

package abc;

public class Hello {
    void hi(String name) { // ①
        String s = name.toLowerCase(); // ②
        int len = s.length(); // ③
        if (len < 10) { // ④
            int p = 10 - len; // ⑤
            for (int i=0; i<10; i++) { // ⑥
                System.out.println(); // ⑦
            } // ⑧
        } // ⑨
    } // ⑩
}

```

我们观察上面的hi()方法代码:

- 方法参数name是局部变量，它的作用域是整个方法，即①~⑩；
- 变量s的作用域是定义处到方法结束，即②~⑩；
- 变量len的作用域是定义处到方法结束，即③~⑩；
- 变量p的作用域是定义处到if块结束，即⑤~⑨；
- 变量i的作用域是for循环，即⑥~⑧。

使用局部变量时，应该尽可能把局部变量的作用域缩小，尽可能延后声明局部变量。

## final

Java还提供了一个final修饰符。final与访问权限不冲突，它有很多作用。

用final修饰class可以阻止被继承:

```

package abc;

// 无法被继承:
public final class Hello {
    private int n = 0;
    protected void hi(int t) {
        long i = t;
    }
}

```

```
}
```

用final修饰method可以阻止被子类覆写:

```
package abc;

public class Hello {
    // 无法被覆写:
    protected final void hi() {
    }
}
```

用final修饰field可以阻止被重新赋值:

```
package abc;

public class Hello {
    private final int n = 0;
    protected void hi() {
        this.n = 1; // error!
    }
}
```

用final修饰局部变量可以阻止被重新赋值:

```
package abc;

public class Hello {
    protected void hi(final int t) {
        t = 1; // error!
    }
}
```

## 最佳实践

如果不确定是否需要public, 就不声明为public, 即尽可能少地暴露对外的字段和方法。

把方法定义为package权限有助于测试, 因为测试类和被测试类只要位于同一个package, 测试代码就可以访问被测试类的package权限方法。

一个.java文件只能包含一个public类, 但可以包含多个非public类。如果有public类, 文件名必须和public类的名字相同。

## 小结

Java内建的访问权限包括public、protected、private和package权限;

Java在方法内部定义的变量是局部变量, 局部变量的作用域从变量声明开始, 到一个块结束;

final修饰符不是访问权限, 它可以修饰class、field和method;

一个.java文件只能包含一个public类, 但可以包含多个非public类。

在Java中，我们经常听到classpath这个东西。网上有很多关于“如何设置classpath”的文章，但大部分设置都不靠谱。

## classpath和jar

到底什么是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，可能长这样：

```
C:\work\project1\bin;C:\shared;"D:\My Documents\project1\bin"
```

在Linux系统上，用:分隔，可能长这样：

```
/usr/shared:/usr/local/bin:/home/liaoxuefeng/bin
```

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-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
```

没有设置系统环境变量，也没有传入-cp参数，那么JVM默认的classpath为.，即当前目录：

```
java abc.xyz.Hello
```

上述命令告诉JVM只在当前目录搜索Hello.class。

在IDE中运行Java程序，IDE自动传入的-cp参数是当前工程的bin目录和引入的jar包。

通常，我们在自己编写的class中，会引用Java核心库的class，例如，String、ArrayList等。这些class应该上哪去找？

有很多“如何设置classpath”的文章会告诉你把JVM自带的rt.jar放入classpath，但事实上，根本不需要告诉JVM如何去Java核心库查找class，JVM怎么可能笨到连自己的核心库在哪都不知道？

不要把任何Java核心库添加到classpath中！JVM根本不依赖classpath加载核心库！

更好的做法是，不要设置classpath! 默认的当前目录.对于绝大多数情况都够用了。

## 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。

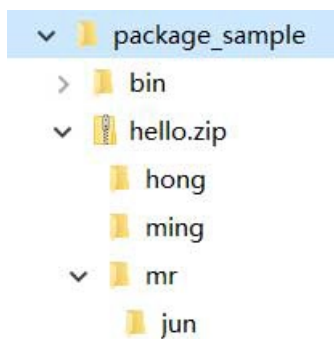
那么问题来了：如何创建jar包？

因为jar包就是zip包，所以，直接在资源管理器中，找到正确的目录，点击右键，在弹出的快捷菜单中选择“发送到”，“压缩(zipped)文件夹”，就制作了一个zip文件。然后，把后缀从.zip改为.jar，一个jar包就创建成功。

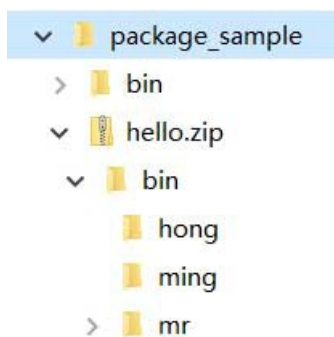
假设编译输出的目录结构是这样：

```
package_sample
├── bin
│   ├── hong
│   │   └── Person.class
│   ├── ming
│   │   └── Person.class
│   └── mr
│       └── jun
│           └── Arrays.class
```

这里需要特别注意的是，jar包里的第一层目录，不能是bin，而应该是hong、ming、mr。如果在Windows的资源管理器中看，应该长这样：



如果长这样：



说明打包打得有问题，JVM仍然无法从jar包中查找正确的class，原因是hong.Person必须按hong/Person.class存放，而不是bin/hong/Person.class。

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。



从Java 9开始，JDK又引入了模块（Module）。

# 模块

什么是模块？这要从Java 9之前的版本说起。

我们知道，.class文件是JVM看到的最小可执行文件，而一个大型程序需要编写很多Class，并生成一堆.class文件，很不便于管理，所以，jar文件就是class文件的容器。

在Java 9之前，一个大型Java程序会生成自己的jar文件，同时引用依赖的第三方jar文件，而JVM自带的Java标准库，实际上也是以jar文件形式存放的，这个文件叫rt.jar，一共有60多M。

如果是自己开发的程序，除了一个自己的app.jar以外，还需要一堆第三方的jar包，运行一个Java程序，一般来说，命令行写这样：

```
java -cp app.jar:a.jar:b.jar:c.jar com.liaoxuefeng.sample.Main
```

注意：JVM自带的标准库rt.jar不要写到classpath中，写了反而会干扰JVM的正常运行。

如果漏写了某个运行时需要用到的jar，那么在运行期极有可能抛出ClassNotFoundException。

所以，jar只是用于存放class的容器，它并不关心class之间的依赖。

从Java 9开始引入的模块，主要是为了解决“依赖”这个问题。如果a.jar必须依赖另一个b.jar才能运行，那我们应该给a.jar加点说明啥的，让程序在编译和运行的时候能自动定位到b.jar，这种自带“依赖关系”的class容器就是模块。

为了表明Java模块化的决心，从Java 9开始，原有的Java标准库已经由一个单一巨大的rt.jar分拆成了几十个模块，这些模块以.jmod扩展名标识，可以在\$JAVA\_HOME/jmods目录下找到它们：

- java.base.jmod
- java.compiler.jmod
- java.datatransfer.jmod
- java.desktop.jmod
- ...

这些.jmod文件每一个都是一个模块，模块名就是文件名。例如：模块java.base对应的文件就是java.base.jmod。模块之间的依赖关系已经被写入到模块内的module-info.class文件了。所有的模块都直接或间接地依赖java.base模块，只有java.base模块不依赖任何模块，它可以被看作是“根模块”，好比所有的类都是从Object直接或间接继承而来。

把一堆class封装为jar仅仅是一个打包的过程，而把一堆class封装为模块则不但需要打包，还需要写入依赖关系，并且还可以包含二进制代码（通常是JNI扩展）。此外，模块支持多版本，即在同一个模块中可以为不同的JVM提供不同的版本。

## 编写模块

那么，我们应该如何编写模块呢？还是以具体的例子来说。首先，创建模块和原有的创建Java项目是完全一样的，以oop-module工程为例，它的目录结构如下：

```
oop-module
├── bin
├── build.sh
├── src
│   ├── com
│   │   └── itranswarp
│   │       └── sample
│   │           ├── Greeting.java
│   │           └── Main.java
│   └── module-info.java
```

其中，bin目录存放编译后的class文件，src目录存放源码，按包名的目录结构存放，仅仅在src目录下多了一

一个module-info.java这个文件，这就是模块的描述文件。在这个模块中，它长这样：

```
module hello.world {
    requires java.base; // 可不写，任何模块都会自动引入java.base
    requires java.xml;
}
```

其中，module是关键字，后面的hello.world是模块的名称，它的命名规范与包一致。花括号的requires xxx;表示这个模块需要引用的其他模块名。除了java.base可以被自动引入外，这里我们引入了一个java.xml的模块。

当我们使用模块声明了依赖关系后，才能使用引入的模块。例如，Main.java代码如下：

```
package com.itranswarp.sample;

// 必须引入java.xml模块后才能使用其中的类：
import javax.xml.XMLConstants;

public class Main {
    public static void main(String[] args) {
        Greeting g = new Greeting();
        System.out.println(g.hello(XMLConstants.XML_NS_PREFIX));
    }
}
```

如果把requires java.xml;从module-info.java中去掉，编译将报错。可见，模块的重要作用就是声明依赖关系。

下面，我们用JDK提供的命令行工具来编译并创建模块。

首先，我们把工作目录切换到oop-module，在当前目录下编译所有的.java文件，并存放到bin目录下，命令如下：

```
$ javac -d bin src/module-info.java src/com/itranswarp/sample/*.java
```

如果编译成功，现在项目结构如下：

```
oop-module
├── bin
│   ├── com
│   │   └── itranswarp
│   │       └── sample
│   │           ├── Greeting.class
│   │           └── Main.class
│   └── module-info.class
└── src
    ├── com
    │   └── itranswarp
    │       └── sample
    │           ├── Greeting.java
    │           └── Main.java
    └── module-info.java
```

注意到src目录下的module-info.java被编译到bin目录下的module-info.class。

下一步，我们需要把bin目录下的所有class文件先打包成jar，在打包的时候，注意传入--main-class参数，让这个jar包能自己定位main方法所在的类：

```
$ jar --create --file hello.jar --main-class com.itranswarp.sample.Main -C bin .
```

现在我们就在当前目录下得到了hello.jar这个jar包，它和普通jar包并无区别，可以直接使用命令java -jar hello.jar来运行它。但是我们的目标是创建模块，所以，继续使用JDK自带的jmod命令把一个jar包转换成模块：

```
$ jmod create --class-path hello.jar hello.jmod
```

于是，在当前目录下我们又得到了hello.jmod这个模块文件，这就是最后打包出来的传说中的模块！

## 运行模块

要运行一个jar，我们使用`java -jar xxx.jar`命令。要运行一个模块，我们只需要指定模块名。试试：

```
$ java --module-path hello.jmod --module hello.world
```

结果是一个错误：

```
Error occurred during initialization of boot layer
java.lang.module.FindException: JMOD format not supported at execution time: hello.jmod
```

原因是.jmod不能被放入--module-path中。换成.jar就没问题了：

```
$ java --module-path hello.jar --module hello.world
Hello, xml!
```

那我们辛辛苦苦创建的hello.jmod有什么用？答案是我们可以用它来打包JRE。

## 打包JRE

前面讲了，为了支持模块化，Java 9首先带头把自己的一个巨大无比的rt.jar拆成了几十个.jmod模块，原因就是，运行Java程序的时候，实际上我们用到的JDK模块，并没有那么多。不需要的模块，完全可以删除。

过去发布一个Java应用程序，要运行它，必须下载一个完整的JRE，再运行jar包。而完整的JRE块头很大，有100多M。怎么给JRE瘦身呢？

现在，JRE自身的标准库已经分拆成了模块，只需要带上程序用到的模块，其他的模块就可以被裁剪掉。怎么裁剪JRE呢？并不是说把系统安装的JRE给删掉部分模块，而是“复制”一份JRE，但只带上用到的模块。为此，JDK提供了jlink命令来干这件事。命令如下：

```
$ jlink --module-path hello.jmod --add-modules java.base,java.xml,hello.world --output jre/
```

我们在--module-path参数指定了我们自己的模块hello.jmod，然后，在--add-modules参数中指定了我们用到的3个模块java.base、java.xml和hello.world，用,分隔。最后，在--output参数指定输出目录。

现在，在当前目录下，我们可以找到jre目录，这是一个完整的并且带有我们自己hello.jmod模块的JRE。试试直接运行这个JRE：

```
$ jre/bin/java --module hello.world
Hello, xml!
```

要分发我们自己的Java应用程序，只需要把这个jre目录打个包给对方发过去，对方直接运行上述命令即可，既不用下载安装JDK，也不用知道如何配置我们自己的模块，极大地方便了分发和部署。

## 访问权限

前面我们讲过，Java的class访问权限分为public、protected、private和默认的包访问权限。引入模块后，这些访问权限的规则就要稍微做些调整。

确切地说，class的这些访问权限只在一个模块内有效，模块和模块之间，例如，a模块要访问b模块的某个class，必要条件是b模块明确地导出了可以访问的包。

举个例子：我们编写的模块hello.world用到了模块java.xml的一个类javax.xml.XMLConstants，我们之所以能直接使用这个类，是因为模块java.xml的module-info.java中声明了若干导出：

```
module java.xml {
    exports java.xml;
    exports javax.xml.catalog;
    exports javax.xml.datatype;
    ...
}
```

只有它声明的导出的包，外部代码才被允许访问。换句话说，如果外部代码想要访问我们的hello.world模块中的com.itranswarp.sample.Greeting类，我们必须将其导出：

```
module hello.world {  
    exports com.itranswarp.sample;  
  
    requires java.base;  
    requires java.xml;  
}
```

因此，模块进一步隔离了代码的访问权限。

## 练习

请下载并练习如何打包模块和JRE。

[打包模块和JRE](#)

## 小结

Java 9引入的模块目的是为了管理依赖；

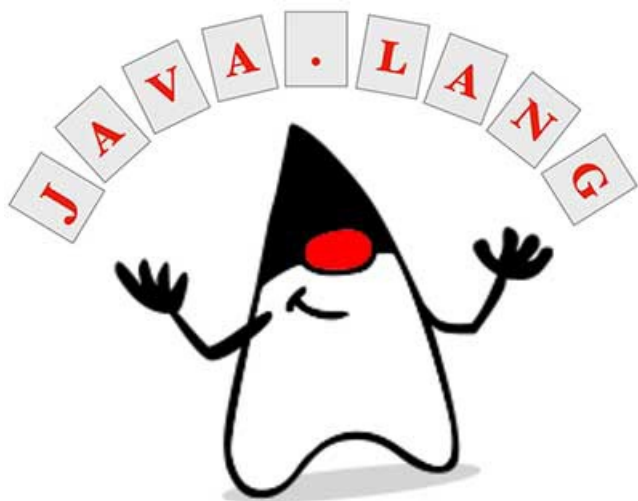
使用模块可以按需打包JRE；

使用模块对类的访问权限有了进一步限制。

本节我们将介绍Java的核心类，包括：

## Java核心类

- 字符串
- `StringBuilder`
- `StringJoiner`
- 包装类型
- `JavaBean`
- 枚举
- 常用工具类



## String

# 字符串和编码

在Java中，String是一个引用类型，它本身也是一个class。但是，Java编译器对String有特殊处理，即可以直接用"..."来表示一个字符串：

```
String s1 = "Hello!";
```

实际上字符串在String内部是通过一个char[]数组表示的，因此，按下面的写法也是可以的：

```
String s2 = new String(new char[] { 'H', 'e', 'l', 'l', 'o', '!' });
```

因为String太常用了，所以Java提供了"..."这种字符串字面量表示方法。

Java字符串的一个重要特点就是字符串不可变。这种不可变性是通过内部的private final char[]字段，以及没有任何修改char[]的方法实现的。

我们来看一个例子：

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

根据上面代码的输出，试解释字符串内容是否改变。

## 字符串比较

当我们想要比较两个字符串是否相同时，要特别注意，我们实际上是想比较字符串的内容是否相同。必须使用equals()方法而不能用==。

我们看下面的例子：

```
// String
----
public class Main {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
    }
}
```

从表面上看，两个字符串用==和equals()比较都为true，但实际上那只是Java编译器在编译期，会自动把所有相同的字符串当作一个对象放入常量池，自然s1和s2的引用就是相同的。

所以，这种==比较返回true纯属巧合。换一种写法，==比较就会失败：

```
// String
----
public class Main {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "HELLO".toLowerCase();
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
    }
}
```

```
    }  
}
```

结论：两个字符串比较，必须总是使用`equals()`方法。

要忽略大小写比较，使用`equalsIgnoreCase()`方法。

`String`类还提供了多种方法来搜索子串、提取子串。常用的方法有：

```
// 是否包含子串：  
"Hello".contains("ll"); // true
```

注意到`contains()`方法的参数是`CharSequence`而不是`String`，因为`CharSequence`是`String`的父类。

搜索子串的更多的例子：

```
"Hello".indexOf("l"); // 2  
"Hello".lastIndexOf("l"); // 3  
"Hello".startsWith("He"); // true  
"Hello".endsWith("lo"); // true
```

提取子串的例子：

```
"Hello".substring(2); // "llo"  
"Hello".substring(2, 4); "ll"
```

注意索引号是从0开始的。

## 去除首尾空白字符

使用`trim()`方法可以移除字符串首尾空白字符。空白字符包括空格，`\t`，`\r`，`\n`：

```
" \tHello\r\n ".trim(); // "Hello"
```

注意：`trim()`并没有改变字符串的内容，而是返回了一个新字符串。

另一个`strip()`方法也可以移除字符串首尾空白字符。它和`trim()`不同的是，类似中文的空格字符`\u3000`也会被移除：

```
"\u3000Hello\u3000".strip(); // "Hello"  
" Hello ".stripLeading(); // "Hello "  
" Hello ".stripTrailing(); // " Hello"
```

`String`还提供了`isEmpty()`和`isBlank()`来判断字符串是否为空和空白字符串：

```
"".isEmpty(); // true, 因为字符串长度为0  
" ".isEmpty(); // false, 因为字符串长度不为0  
"\n".isBlank(); // true, 因为只包含空白字符  
" Hello ".isBlank(); // false, 因为包含非空白字符
```

## 替换子串

要在字符串中替换子串，有两种方法。一种是根据字符或字符串替换：

```
String s = "hello";  
s.replace('l', 'w'); // "hewwo", 所有字符'l'被替换为'w'  
s.replace("ll", "~~"); // "he~~o", 所有子串"ll"被替换为"~~"
```

另一种是通过正则表达式替换：

```
String s = "A,B;C ,D";  
s.replaceAll("[\\,\\;\\s]+", ","); // "A,B,C,D"
```

上面的代码通过正则表达式，把匹配的子串统一替换为`,`。关于正则表达式的用法我们会在后面详细讲解。

## 分割字符串

要分割字符串，使用`split()`方法，并且传入的也是正则表达式：

```
String s = "A,B,C,D";
String[] ss = s.split("\\,"); // {"A", "B", "C", "D"}
```

## 拼接字符串

拼接字符串使用静态方法`join()`，它用指定的字符串连接字符串数组：

```
String[] arr = {"A", "B", "C"};
String s = String.join("****", arr); // "A****B****C"
```

## 类型转换

要把任意基本类型或引用类型转换为字符串，可以使用静态方法`valueOf()`。这是一个重载方法，编译器会根据参数自动选择合适的方法：

```
String.valueOf(123); // "123"
String.valueOf(45.67); // "45.67"
String.valueOf(true); // "true"
String.valueOf(new Object()); // 类似java.lang.Object@636be97c
```

要把字符串转换为其他类型，就需要根据情况。例如，把字符串转换为`int`类型：

```
int n1 = Integer.parseInt("123"); // 123
int n2 = Integer.parseInt("ff", 16); // 按十六进制转换，255
```

把字符串转换为`boolean`类型：

```
boolean b1 = Boolean.parseBoolean("true"); // true
boolean b2 = Boolean.parseBoolean("FALSE"); // false
```

要特别注意，`Integer`有个`getInteger(String)`方法，它不是将字符串转换为`int`，而是把该字符串对应的系统变量转换为`Integer`：

```
Integer.getInteger("java.version"); // 版本号，11
```

## 转换为char[]

`String`和`char[]`类型可以互相转换，方法是：

```
char[] cs = "Hello".toCharArray(); // String -> char[]
String s = new String(cs); // char[] -> String
```

如果修改了`char[]`数组，`String`并不会改变：

```
// String <-> char[]
----
public class Main {
    public static void main(String[] args) {
        char[] cs = "Hello".toCharArray();
        String s = new String(cs);
        System.out.println(s);
        cs[0] = 'X';
        System.out.println(s);
    }
}
```

这是因为通过`new String(char[])`创建新的`String`实例时，它并不会直接引用传入的`char[]`数组，而是会复制一份，所以，修改外部的`char[]`数组不会影响`String`实例内部的`char[]`数组，因为这是两个不同的数组。

从`String`的不变性设计可以看出，如果传入的对象有可能改变，我们需要复制而不是直接引用。

例如，下面的代码设计了一个`Score`类保存一组学生的成绩：



```
// int[]
import java.util.Arrays;
----
public class Main {
    public static void main(String[] args) {
        int[] scores = new int[] { 88, 77, 51, 66 };
        Score s = new Score(scores);
        s.printScores();
        scores[2] = 99;
        s.printScores();
    }
}

class Score {
    private int[] scores;
    public Score(int[] scores) {
        this.scores = scores;
    }

    public void printScores() {
        System.out.println(Arrays.toString(scores));
    }
}
```

观察两次输出，由于Score内部直接引用了外部传入的int[]数组，这会造成外部代码对int[]数组的修改，影响到Score类的字段。如果外部代码不可信，这就会造成安全隐患。

请修复Score的构造方法，使得外部代码对数组的修改不影响Score实例的int[]字段。

## 字符编码

在早期的计算机系统中，为了给字符编码，美国国家标准学会（**American National Standard Institute: ANSI**）制定了一套英文字母、数字和常用符号的编码，它占用一个字节，编码范围从0到127，最高位始终为0，称为ASCII编码。例如，字符'A'的编码是0x41，字符'l'的编码是0x31。

如果要把汉字也纳入计算机编码，很显然一个字节是不够的。GB2312标准使用两个字节表示一个汉字，其中第一个字节的最高位始终为1，以便和ASCII编码区分开。例如，汉字'中'的GB2312编码是0xd6d0。

类似的，日文有Shift\_JIS编码，韩文有EUC-KR编码，这些编码因为标准不统一，同时使用，就会产生冲突。

为了统一全球所有语言的编码，全球统一码联盟发布了Unicode编码，它把世界上主要语言都纳入同一个编码，这样，中文、日文、韩文和其他语言就不会冲突。

Unicode编码需要两个或者更多字节表示，我们可以比较中英文字符在ASCII、GB2312和Unicode的编码：

英文字符'A'的ASCII编码和Unicode编码：

ASCII:	41	
Unicode:	00	41

英文字符的Unicode编码就是简单地在前面添加一个00字节。

中文字符'中'的GB2312编码和Unicode编码：

GB2312:	d6	d0
Unicode:	4e	2d

那我们经常使用的UTF-8又是什么编码呢？因为英文字符的Unicode编码高字节总是00，包含大量英文的文本会浪费空

间，所以，出现了UTF-8编码，它是一种变长编码，用来把固定长度的Unicode编码变成1~4字节的变长编码。通过UTF-8编码，英文字符'A'的UTF-8编码变为0x41，正好和ASCII码一致，而中文'中'的UTF-8编码为3字节0xe4b8ad。

UTF-8编码的另一个好处是容错能力强。如果传输过程中某些字符出错，不会影响后续字符，因为UTF-8编码依靠高字节位来确定一个字符究竟是几个字节，它经常用来作为传输编码。

在Java中，char类型实际上就是两个字节的Unicode编码。如果我们要手动把字符串转换成其他编码，可以这样做：

```
byte[] b1 = "Hello".getBytes(); // 按ISO8859-1编码转换，不推荐
byte[] b2 = "Hello".getBytes("UTF-8"); // 按UTF-8编码转换
byte[] b2 = "Hello".getBytes("GBK"); // 按GBK编码转换
byte[] b3 = "Hello".getBytes(StandardCharsets.UTF_8); // 按UTF-8编码转换
```

注意：转换编码后，就不再是char类型，而是byte类型表示的数组。

如果要把已知编码的byte[]转换为String，可以这样做：

```
byte[] b = ...
String s1 = new String(b, "GBK"); // 按GBK转换
String s2 = new String(b, StandardCharsets.UTF_8); // 按UTF-8转换
```

始终牢记：Java的String和char在内存中总是以Unicode编码表示。

## 延伸阅读

对于不同版本的JDK，String类在内存中有不同的优化方式。具体来说，早期JDK版本的String总是以char[]存储，它的定义如下：

```
public final class String {
    private final char[] value;
    private final int offset;
    private final int count;
}
```

而较新的JDK版本的String则以byte[]存储：如果String仅包含ASCII字符，则每个byte存储一个字符，否则，每两个byte存储一个字符，这样做的目的是为了节省内存，因为大量的长度较短的String通常仅包含ASCII字符：

```
public final class String {
    private final byte[] value;
    private final byte coder; // 0 = LATIN1, 1 = UTF16
}
```

对于使用者来说，String内部的优化不影响任何已有代码，因为它的public方法签名是不变的。

## 小结

- Java字符串String是不可变对象；
- 字符串操作不改变原字符串内容，而是返回新字符串；
- 常用的字符串操作：提取子串、查找、替换、大小写转换等；
- Java使用Unicode编码表示String和char；
- 转换编码就是将String和byte[]转换，需要指定编码；
- 转换为byte[]时，始终优先考虑UTF-8编码。

Java编译器对String做了特殊处理，使得我们可以直接用+拼接字符串。

## StringBuilder

考察下面的循环代码：

```
String s = "";
for (int i = 0; i < 1000; i++) {
    s = s + "," + i;
}
```

虽然可以直接拼接字符串，但是，在循环中，每次循环都会创建新的字符串对象，然后扔掉旧的字符串。这样，绝大部分字符串都是临时对象，不但浪费内存，还会影响GC效率。

为了能高效拼接字符串，Java标准库提供了StringBuilder，它是一个可变对象，可以预分配缓冲区，这样，往StringBuilder中新增字符时，不会创建新的临时对象：

```
StringBuilder sb = new StringBuilder(1024);
for (int i = 0; i < 1000; i++) {
    sb.append(',');
    sb.append(i);
}
String s = sb.toString();
```

StringBuilder还可以进行链式操作：

```
// 链式操作
----
public class Main {
    public static void main(String[] args) {
        var sb = new StringBuilder(1024);
        sb.append("Mr ")
          .append("Bob")
          .append("!!")
          .insert(0, "Hello, ");
        System.out.println(sb.toString());
    }
}
```

如果我们查看StringBuilder的源码，可以发现，进行链式操作的关键是，定义的append()方法会返回this，这样，就可以不断调用自身的其他方法。

仿照StringBuilder，我们也可以设计支持链式操作的类。例如，一个可以不断增加的计数器：

```
// 链式操作
----
public class Main {
    public static void main(String[] args) {
        Adder adder = new Adder();
        adder.add(3)
              .add(5)
              .inc()
              .add(10);
        System.out.println(adder.value());
    }
}

class Adder {
    private int sum = 0;

    public Adder add(int n) {
        sum += n;
        return this;
    }

    public Adder inc() {
```

```

        sum ++;
        return this;
    }

    public int value() {
        return sum;
    }
}

```

注意：对于普通的字符串+操作，并不需要我们将其改写为StringBuilder，因为Java编译器在编译时就自动把多个连续的+操作编码为StringConcatFactory的操作。在运行期，StringConcatFactory会自动把字符串连接操作优化为数组复制或者StringBuilder操作。

你可能还听说过StringBuffer，这是Java早期的一个StringBuilder的线程安全版本，它通过同步来保证多个线程操作StringBuffer也是安全的，但是同步会带来执行速度的下降。

StringBuilder和StringBuffer接口完全相同，现在完全没有必要使用StringBuffer。

## 练习

请使用StringBuilder构造一个INSERT语句：

```

public class Main {
    public static void main(String[] args) {
        String[] fields = { "name", "position", "salary" };
        String table = "employee";
        String insert = buildInsertSql(table, fields);
        System.out.println(insert);
        String s = "INSERT INTO employee (name, position, salary) VALUES (?, ?, ?)";
        System.out.println(s.equals(insert) ? "测试成功" : "测试失败");
    }
}
----
static String buildInsertSql(String table, String[] fields) {
    // TODO:
    return "";
}
----
}

```

## [StringBuilder练习](#)

## 小结

StringBuilder是可变对象，用来高效拼接字符串；

StringBuilder可以支持链式操作，实现链式操作的关键是返回实例本身；

StringBuffer是StringBuilder的线程安全版本，现在很少使用。

要高效拼接字符串，应该使用StringBuilder。

## StringJoiner

很多时候，我们拼接的字符串像这样：

```
// Hello Bob, Alice, Grace!
----
public class Main {
    public static void main(String[] args) {
        String[] names = {"Bob", "Alice", "Grace"};
        var sb = new StringBuilder();
        sb.append("Hello ");
        for (String name : names) {
            sb.append(name).append(", ");
        }
        // 注意去掉最后的", ":
        sb.delete(sb.length() - 2, sb.length());
        sb.append("!");
        System.out.println(sb.toString());
    }
}
```

类似用分隔符拼接数组的需求很常见，所以Java标准库还提供了一个StringJoiner来干这个事：

```
import java.util.StringJoiner;
----
public class Main {
    public static void main(String[] args) {
        String[] names = {"Bob", "Alice", "Grace"};
        var sj = new StringJoiner(", ");
        for (String name : names) {
            sj.add(name);
        }
        System.out.println(sj.toString());
    }
}
```

慢着！用StringJoiner的结果少了前面的"Hello "和结尾的"!!"！遇到这种情况，需要给StringJoiner指定“开头”和“结尾”：

```
import java.util.StringJoiner;
----
public class Main {
    public static void main(String[] args) {
        String[] names = {"Bob", "Alice", "Grace"};
        var sj = new StringJoiner(", ", "Hello ", "!");
        for (String name : names) {
            sj.add(name);
        }
        System.out.println(sj.toString());
    }
}
```

那么StringJoiner内部是如何拼接字符串的呢？如果查看源码，可以发现，StringJoiner内部实际上就是使用了StringBuilder，所以拼接效率和StringBuilder几乎是一模一样的。

### String.join()

String还提供了一个静态方法join()，这个方法在内部使用了StringJoiner来拼接字符串，在不需要指定“开头”和“结尾”的时候，用String.join()更方便：

```
String[] names = {"Bob", "Alice", "Grace"};
var s = String.join(", ", names);
```

### 练习

请使用StringJoiner构造一个SELECT语句：

```
import java.util.StringJoiner;

public class Main {
```

```

public static void main(String[] args) {
    String[] fields = { "name", "position", "salary" };
    String table = "employee";
    String select = buildSelectSql(table, fields);
    System.out.println(select);
    System.out.println("SELECT name, position, salary FROM employee".equals(select) ? "测试成功" : "测试失败");
}
----
static String buildSelectSql(String table, String[] fields) {
    // TODO:
    return "";
}
----
}

```

[StringJoiner练习](#)

## 小结

用指定分隔符拼接字符串数组时，使用StringJoiner或者String.join()更方便；

用StringJoiner拼接字符串时，还可以额外附加一个“开头”和“结尾”。

我们已经知道，Java的数据类型分两种：

## 包装类型

- 基本类型：byte, short, int, long, boolean, float, double, char
- 引用类型：所有class和interface类型

引用类型可以赋值为null，表示空，但基本类型不能赋值为null：

```
String s = null;
int n = null; // compile error!
```

那么，如何把一个基本类型视为对象（引用类型）？

比如，想要把int基本类型变成一个引用类型，我们可以定义一个Integer类，它只包含一个实例字段int，这样，Integer类就可以视为int的包装类（Wrapper Class）：

```
public class Integer {
    private int value;

    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return this.value;
    }
}
```

定义好了Integer类，我们就可以把int和Integer互相转换：

```
Integer n = null;
Integer n2 = new Integer(99);
int n3 = n2.intValue();
```

实际上，因为包装类型非常有用，Java核心库为每种基本类型都提供了对应的包装类型：

### 基本类型 对应的引用类型

boolean	java.lang.Boolean
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character

我们可以直接使用，并不需要自己去定义：

```
// Integer:
----
public class Main {
    public static void main(String[] args) {
        int i = 100;
        // 通过new操作符创建Integer实例 (不推荐使用, 会有编译警告):
        Integer n1 = new Integer(i);
        // 通过静态方法valueOf(int)创建Integer实例:
        Integer n2 = Integer.valueOf(i);
        // 通过静态方法valueOf(String)创建Integer实例:
        Integer n3 = Integer.valueOf("100");
        System.out.println(n3.intValue());
    }
}
```

```
    }  
}
```

## Auto Boxing

因为int和Integer可以互相转换：

```
int i = 100;  
Integer n = Integer.valueOf(i);  
int x = n.intValue();
```

所以，Java编译器可以帮助我们自动在int和Integer之间转型：

```
Integer n = 100; // 编译器自动使用Integer.valueOf(int)  
int x = n; // 编译器自动使用Integer.intValue()
```

这种直接把int变为Integer的赋值写法，称为自动装箱（Auto Boxing），反过来，把Integer变为int的赋值写法，称为自动拆箱（Auto Unboxing）。

注意：自动装箱和自动拆箱只发生在编译阶段，目的是为了少写代码。

装箱和拆箱会影响代码的执行效率，因为编译后的class代码是严格区分基本类型和引用类型的。并且，自动拆箱执行时可能会报NullPointerException：

```
// NullPointerException  
----  
public class Main {  
    public static void main(String[] args) {  
        Integer n = null;  
        int i = n;  
    }  
}
```

## 不变类

所有的包装类型都是不变类。我们查看Integer的源码可知，它的核心代码如下：

```
public final class Integer {  
    private final int value;  
}
```

因此，一旦创建了Integer对象，该对象就是不变的。

对两个Integer实例进行比较要特别注意：绝对不能用==比较，因为Integer是引用类型，必须使用equals()比较：

```
// == or equals?  
----  
public class Main {  
    public static void main(String[] args) {  
        Integer x = 127;  
        Integer y = 127;  
        Integer m = 99999;  
        Integer n = 99999;  
        System.out.println("x == y: " + (x==y)); // true  
        System.out.println("m == n: " + (m==n)); // false  
        System.out.println("x.equals(y): " + x.equals(y)); // true  
        System.out.println("m.equals(n): " + m.equals(n)); // true  
    }  
}
```

仔细观察结果的童鞋可以发现，==比较，较小的两个相同的Integer返回true，较大的两个相同的Integer返回false，这是因为Integer是不变类，编译器把Integer x = 127;自动变为Integer x = Integer.valueOf(127);，为了节省内存，Integer.valueOf()对于较小的数，始终返回相同的实例，因此，==比较“恰好”为true，但我们绝不能因为Java标准库的Integer内部有缓存优化就用==比较，必须用equals()方法比较两个Integer。

按照语义编程，而不是针对特定的底层实现去“优化”。



因为Integer.valueOf()可能始终返回同一个Integer实例，因此，在我们自己创建Integer的时候，以下两种方法：

- 方法1: Integer n = new Integer(100);
- 方法2: Integer n = Integer.valueOf(100);

方法2更好，因为方法1总是创建新的Integer实例，方法2把内部优化留给Integer的实现者去做，即使当前版本没有优化，也有可能在下一个版本进行优化。

我们把能创建“新”对象的静态方法称为静态工厂方法。Integer.valueOf()就是静态工厂方法，它尽可能地返回缓存的实例以节省内存。

创建新对象时，优先选用静态工厂方法而不是new操作符。

如果我们考察Byte.valueOf()方法的源码，可以看到，标准库返回的Byte实例全部是缓存实例，但调用者并不关心静态工厂方法以何种方式创建新实例还是直接返回缓存的实例。

## 进制转换

Integer类本身还提供了大量方法，例如，最常用的静态方法parseInt()可以把字符串解析成一个整数：

```
int x1 = Integer.parseInt("100"); // 100
int x2 = Integer.parseInt("100", 16); // 256, 因为按16进制解析
```

Integer还可以把整数格式化为指定进制的字符串：

```
// Integer:
----
public class Main {
    public static void main(String[] args) {
        System.out.println(Integer.toString(100)); // "100", 表示为10进制
        System.out.println(Integer.toString(100, 36)); // "2s", 表示为36进制
        System.out.println(Integer.toHexString(100)); // "64", 表示为16进制
        System.out.println(Integer.toOctalString(100)); // "144", 表示为8进制
        System.out.println(Integer.toBinaryString(100)); // "1100100", 表示为2进制
    }
}
```

注意：上述方法的输出都是String，在计算机内存中，只用二进制表示，不存在十进制或十六进制的表示方法。int n = 100在内存中总是以4字节的二进制表示：

00000000	00000000	00000000	01100100
----------	----------	----------	----------

我们经常使用的System.out.println(n)；是依靠核心库自动把整数格式化为10进制输出并显示在屏幕上，使用Integer.toHexString(n)则通过核心库自动把整数格式化为16进制。

这里我们注意到程序设计的一个重要原则：数据的存储和显示要分离。

Java的包装类型还定义了一些有用的静态变量

```
// boolean只有两个值true/false，其包装类型只需要引用Boolean提供的静态字段：
Boolean t = Boolean.TRUE;
Boolean f = Boolean.FALSE;
// int可表示的最大/最小值：
int max = Integer.MAX_VALUE; // 2147483647
int min = Integer.MIN_VALUE; // -2147483648
// long类型占用的bit和byte数量：
int sizeOfLong = Long.SIZE; // 64 (bits)
int bytesOfLong = Long.BYTES; // 8 (bytes)
```

最后，所有的整数和浮点数的包装类型都继承自Number，因此，可以非常方便地直接通过包装类型获取各种基本类型：

```
// 向上转型为Number：
```

```
Number num = new Integer(999);
// 获取byte, int, long, float, double:
byte b = num.byteValue();
int n = num.intValue();
long ln = num.longValue();
float f = num.floatValue();
double d = num.doubleValue();
```

## 处理无符号整型

在Java中，并没有无符号整型（Unsigned）的基本数据类型。byte、short、int和long都是带符号整型，最高位是符号位。而C语言则提供了CPU支持的全部数据类型，包括无符号整型。无符号整型和有符号整型的转换在Java中就需要借助包装类型的静态方法完成。

例如，byte是有符号整型，范围是-128~+127，但如果把byte看作无符号整型，它的范围就是0~255。我们把一个负的byte按无符号整型转换为int：

```
// Byte
----
public class Main {
    public static void main(String[] args) {
        byte x = -1;
        byte y = 127;
        System.out.println(Byte.toUnsignedInt(x)); // 255
        System.out.println(Byte.toUnsignedInt(y)); // 127
    }
}
```

因为byte的-1的二进制表示是11111111，以无符号整型转换后的int就是255。

类似的，可以把一个short按unsigned转换为int，把一个int按unsigned转换为long。

## 小结

Java核心库提供的包装类型可以把基本类型包装为class；

自动装箱和自动拆箱都是在编译期完成的（JDK>=1.5）；

装箱和拆箱会影响执行效率，且拆箱时可能发生NullPointerException；

包装类型的比较必须使用equals()；

整数和浮点数的包装类型都继承自Number；

包装类型提供了大量实用方法。

在Java中，有很多class的定义都符合这样的规范：

## JavaBean

- 若干private实例字段；
- 通过public方法来读写实例字段。

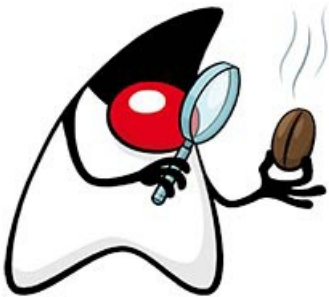
例如：

```
public 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) { this.age = age; }  
}
```

如果读写方法符合以下这种命名规范：

```
// 读方法：  
public Type getXyz()  
// 写方法：  
public void setXyz(Type value)
```

那么这种class被称为JavaBean：



上面的字段是xyz，那么读写方法名分别以get和set开头，并且后接大写字母开头的字段名Xyz，因此两个读写方法名分别是getXyz()和setXyz()。

boolean字段比较特殊，它的读方法一般命名为isXyz()：

```
// 读方法：  
public boolean isChild()  
// 写方法：  
public void setChild(boolean value)
```

我们通常把一组对应的读方法（getter）和写方法（setter）称为属性（property）。例如，name属性：

- 对应的读方法是String getName()
- 对应的写方法是setName(String)

只有getter的属性称为只读属性（read-only），例如，定义一个age只读属性：

- 对应的读方法是int getAge()
- 无对应的写方法setAge(int)

类似的，只有setter的属性称为只写属性（write-only）。

很明显，只读属性很常见，只写属性不常见。

属性只需要定义getter和setter方法，不一定需要对应的字段。例如，child只读属性定义如下：

```
public 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) { this.age = age; }

    public boolean isChild() {
        return age <= 6;
    }
}
```

可以看出，getter和setter也是一种数据封装的方法。

## JavaBean的作用

JavaBean主要用来传递数据，即把一组数据组合成一个JavaBean便于传输。此外，JavaBean可以方便地被IDE工具分析，生成读写属性的代码，主要用在图形界面的可视化设计中。

通过IDE，可以快速生成getter和setter。例如，在Eclipse中，先输入以下代码：

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

然后，点击右键，在弹出的菜单中选择“Source”，“Generate Getters and Setters”，在弹出的对话框中选需要生成getter和setter方法的字段，点击确定即可由IDE自动完成所有方法代码。

## 枚举JavaBean属性

要枚举一个JavaBean的所有属性，可以直接使用Java核心库提供的Introspector：

```
import java.beans.*;
----
public class Main {
    public static void main(String[] args) throws Exception {
        BeanInfo info = Introspector.getBeanInfo(Person.class);
        for (PropertyDescriptor pd : info.getPropertyDescriptors()) {
            System.out.println(pd.getName());
            System.out.println("  " + pd.getReadMethod());
            System.out.println("  " + pd.getWriteMethod());
        }
    }
}

class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

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

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

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

运行上述代码，可以列出所有的属性，以及对应的读写方法。注意class属性是从Object继承的getClass()方法带来的。

## 小结

**JavaBean**是一种符合命名规范的class，它通过getter和setter来定义属性；

属性是一种通用的叫法，并非Java语法规定；

可以利用IDE快速生成getter和setter；

使用Introspector.getBeanInfo() 可以获取属性列表。

在Java中，我们可以通过static final来定义常量。例如，我们希望定义周一到周日这7个常量，可以用7个不同的int表示：

## 枚举类

```
public class Weekday {
    public static final int SUN = 0;
    public static final int MON = 1;
    public static final int TUE = 2;
    public static final int WED = 3;
    public static final int THU = 4;
    public static final int FRI = 5;
    public static final int SAT = 6;
}
```

使用常量的时候，可以这么引用：

```
if (day == Weekday.SAT || day == Weekday.SUN) {
    // TODO: work at home
}
```

也可以把常量定义为字符串类型，例如，定义3种颜色的常量：

```
public class Color {
    public static final String RED = "r";
    public static final String GREEN = "g";
    public static final String BLUE = "b";
}
```

使用常量的时候，可以这么引用：

```
String color = ...
if (Color.RED.equals(color)) {
    // TODO:
}
```

无论是int常量还是String常量，使用这些常量来表示一组枚举值的时候，有一个严重的问题就是，编译器无法检查每个值的合理性。例如：

```
if (weekday == 6 || weekday == 7) {
    if (tasks == Weekday.MON) {
        // TODO:
    }
}
```

上述代码编译和运行均不会报错，但存在两个问题：

- 注意到Weekday定义的常量范围是0~6，并不包含7，编译器无法检查不在枚举中的int值；
- 定义的常量仍可与其他变量比较，但其用途并非枚举星期值。

### enum

为了让编译器能自动检查某个值在枚举的集合内，并且，不同用途的枚举需要不同的类型来标记，不能混用，我们可以使用enum来定义枚举类：

```
// enum
----
public class Main {
    public static void main(String[] args) {
        Weekday day = Weekday.SUN;
        if (day == Weekday.SAT || day == Weekday.SUN) {
            System.out.println("Work at home!");
        } else {
            System.out.println("Work at office!");
        }
    }
}

enum Weekday {
    SUN, MON, TUE, WED, THU, FRI, SAT;
}
```

注意到定义枚举类是通过关键字enum实现的，我们只需依次列出枚举的常量名。

和int定义的常量相比，使用enum定义枚举有如下好处：

首先，enum常量本身带有类型信息，即Weekday.SUN类型是Weekday，编译器会自动检查出类型错误。例如，下面的语句不可能编译通过：

```
int day = 1;
if (day == Weekday.SUN) { // Compile error: bad operand types for binary operator '=='
}
```

其次，不可能引用到非枚举的值，因为无法通过编译。

最后，不同类型的枚举不能互相比较或者赋值，因为类型不符。例如，不能给一个Weekday枚举类型的变量赋值为Color枚举类型的值：

```
Weekday x = Weekday.SUN; // ok!
Weekday y = Color.RED; // Compile error: incompatible types
```

这就使得编译器可以在编译期自动检查出所有可能的潜在错误。

## enum的比较

使用enum定义的枚举类是一种引用类型。前面我们讲到，引用类型比较，要使用equals()方法，如果使用==比较，它比较的是两个引用类型的变量是否是同一个对象。因此，引用类型比较，要始终使用equals()方法，但enum类型可以例外。

这是因为enum类型的每个常量在JVM中只有一个唯一实例，所以可以直接用==比较：

```
if (day == Weekday.FRI) { // ok!
}
if (day.equals(Weekday.SUN)) { // ok, but more code!
}
```

## enum类型

通过enum定义的枚举类，和其他的class有什么区别？

答案是没有任何区别。enum定义的类型就是class，只不过它有以下几个特点：

- 定义的enum类型总是继承自java.lang.Enum，且无法被继承；
- 只能定义出enum的实例，而无法通过new操作符创建enum的实例；
- 定义的每个实例都是引用类型的唯一实例；
- 可以将enum类型用于switch语句。

例如，我们定义的Color枚举类：

```
public enum Color {
    RED, GREEN, BLUE;
}
```

编译器编译出的class大概就像这样：

```
public final class Color extends Enum { // 继承自Enum，标记为final class
    // 每个实例均为全局唯一：
    public static final Color RED = new Color();
    public static final Color GREEN = new Color();
    public static final Color BLUE = new Color();
    // private构造方法，确保外部无法调用new操作符：
    private Color() {}
}
```

所以，编译后的enum类和普通class并没有任何区别。但是我们自己无法按定义普通class那样来定义enum，必须使用enum关键字，这是Java语法规定的。

因为enum是一个class，每个枚举的值都是class实例，因此，这些实例有一些方法：

### name()

返回常量名，例如：

```
String s = Weekday.SUN.name(); // "SUN"
```

### ordinal()

返回定义的常量的顺序，从0开始计数，例如：

```
int n = Weekday.MON.ordinal(); // 1
```

改变枚举常量定义的顺序就会导致ordinal()返回值发生变化。例如：

```
public enum Weekday {
    SUN, MON, TUE, WED, THU, FRI, SAT;
}
```

```
}
```

和

```
public enum Weekday {  
    MON, TUE, WED, THU, FRI, SAT, SUN;  
}
```

的ordinal就是不同的。如果在代码中编写了类似if(x.ordinal()==1)这样的语句，就要保证enum的枚举顺序不能变。新增的常量必须放在最后。

有些童鞋会想，Weekday的枚举常量如果要和int转换，使用ordinal()不是非常方便？比如这样写：

```
String task = Weekday.MON.ordinal() + "/ppt";  
saveToFile(task);
```

但是，如果不小心修改了枚举的顺序，编译器是无法检查出这种逻辑错误的。要编写健壮的代码，就不要依靠ordinal()的返回值。因为enum本身是class，所以我们可以定义private的构造方法，并且，给每个枚举常量添加字段：

```
// enum  
----  
public class Main {  
    public static void main(String[] args) {  
        Weekday day = Weekday.SUN;  
        if (day.dayValue == 6 || day.dayValue == 0) {  
            System.out.println("Work at home!");  
        } else {  
            System.out.println("Work at office!");  
        }  
    }  
}  
  
enum Weekday {  
    MON(1), TUE(2), WED(3), THU(4), FRI(5), SAT(6), SUN(0);  
  
    public final int dayValue;  
  
    private Weekday(int dayValue) {  
        this.dayValue = dayValue;  
    }  
}
```

这样就无需担心顺序的变化，新增枚举常量时，也需要指定一个int值。

注意：枚举类的字段也可以是非final类型，即可以在运行期修改，但是不推荐这样做！

默认情况下，对枚举常量调用toString()会返回和name()一样的字符串。但是，toString()可以被覆写，而name()则不行。我们可以给Weekday添加toString()方法：

```
// enum  
----  
public class Main {  
    public static void main(String[] args) {  
        Weekday day = Weekday.SUN;  
        if (day.dayValue == 6 || day.dayValue == 0) {  
            System.out.println("Today is " + day + ". Work at home!");  
        } else {  
            System.out.println("Today is " + day + ". Work at office!");  
        }  
    }  
}  
  
enum Weekday {  
    MON(1, "星期一"), TUE(2, "星期二"), WED(3, "星期三"), THU(4, "星期四"), FRI(5, "星期五"), SAT(6, "星期六"), SUN(0, "星期日");  
  
    public final int dayValue;  
    private final String chinese;  
  
    private Weekday(int dayValue, String chinese) {  
        this.dayValue = dayValue;  
        this.chinese = chinese;  
    }  
  
    @Override  
    public String toString() {  
        return this.chinese;  
    }  
}
```

覆写toString()的目的是在输出时更有可读性。



注意：判断枚举常量的名字，要始终使用`name()`方法，绝不能调用`toString()`！

## switch

最后，枚举类可以应用在`switch`语句中。因为枚举类天生具有类型信息和有限个枚举常量，所以比`int`、`String`类型更适合用在`switch`语句中：

```
// switch
----
public class Main {
    public static void main(String[] args) {
        Weekday day = Weekday.SUN;
        switch(day) {
            case MON:
            case TUE:
            case WED:
            case THU:
            case FRI:
                System.out.println("Today is " + day + ". Work at office!");
                break;
            case SAT:
            case SUN:
                System.out.println("Today is " + day + ". Work at home!");
                break;
            default:
                throw new RuntimeException("cannot process " + day);
        }
    }
}

enum Weekday {
    MON, TUE, WED, THU, FRI, SAT, SUN;
}
```

加上`default`语句，可以在漏写某个枚举常量时自动报错，从而及时发现错误。

## 小结

Java使用`enum`定义枚举类型，它被编译器编译为`final class Xxx extends Enum { ... }`；

通过`name()`获取常量定义的字符串，注意不要使用`toString()`；

通过`ordinal()`返回常量定义的顺序（无实质意义）；

可以为`enum`编写构造方法、字段和方法

`enum`的构造方法要声明为`private`，字段强烈建议声明为`final`；

`enum`适合用在`switch`语句中。

## BigInteger

# BigInteger

在Java中，由CPU原生提供的整型最大范围是64位long型整数。使用long型整数可以直接通过CPU指令进行计算，速度非常快。

如果我们使用的整数范围超过了long型怎么办？这个时候，就只能用软件来模拟一个大整数。java.math.BigInteger就是用来表示任意大小的整数。BigInteger内部用一个int[]数组来模拟一个非常大的整数：

```
BigInteger bi = new BigInteger("1234567890");
System.out.println(bi.pow(5)); // 2867971860299718107233761438093672048294900000
```

对BigInteger做运算的时候，只能使用实例方法，例如，加法运算：

```
BigInteger i1 = new BigInteger("1234567890");
BigInteger i2 = new BigInteger("12345678901234567890");
BigInteger sum = i1.add(i2); // 12345678902469135780
```

和long型整数运算比，BigInteger不会有范围限制，但缺点是速度比较慢。

也可以把BigInteger转换成long型：

```
BigInteger i = new BigInteger("123456789000");
System.out.println(i.longValue()); // 123456789000
System.out.println(i.multiply(i).longValueExact()); // java.lang.ArithmeticException: BigInteger out of long range
```

使用longValueExact()方法时，如果超出了long型的范围，会抛出ArithmeticException。

BigInteger和Integer、Long一样，也是不可变类，并且也继承自Number类。因为Number定义了转换为基本类型的几个方法：

- 转换为byte: byteValue()
- 转换为short: shortValue()
- 转换为int: intValue()
- 转换为long: longValue()
- 转换为float: floatValue()
- 转换为double: doubleValue()

因此，通过上述方法，可以把BigInteger转换成基本类型。如果BigInteger表示的范围超过了基本类型的范围，转换时将丢失高位信息，即结果不一定是准确的。如果需要准确地转换成基本类型，可以使用intValueExact()、longValueExact()等方法，在转换时如果超出范围，将直接抛出ArithmeticException异常。

如果BigInteger的值甚至超过了float的最大范围（ $3.4 \times 10^{38}$ ），那么返回的float是什么呢？

```
// BigInteger to float
import java.math.BigInteger;
----
public class Main {
    public static void main(String[] args) {
        BigInteger n = new BigInteger("999999").pow(99);
        float f = n.floatValue();
        System.out.println(f);
    }
}
```

## 小结

BigInteger用于表示任意大小的整数；

BigInteger是不变类，并且继承自Number；

将BigInteger转换成基本类型时可使用longValueExact()等方法保证结果准确。

和BigInteger类似，BigDecimal可以表示一个任意大小且精度完全准确的浮点数。

## BigDecimal

```
BigDecimal bd = new BigDecimal("123.4567");
System.out.println(bd.multiply(bd)); // 15241.55677489
```

BigDecimal用scale()表示小数位数，例如：

```
BigDecimal d1 = new BigDecimal("123.45");
BigDecimal d2 = new BigDecimal("123.4500");
BigDecimal d3 = new BigDecimal("1234500");
System.out.println(d1.scale()); // 2, 两位小数
System.out.println(d2.scale()); // 4
System.out.println(d3.scale()); // 0
```

通过BigDecimal的stripTrailingZeros()方法，可以将一个BigDecimal格式化为一个相等的，但去掉了末尾0的BigDecimal：

```
BigDecimal d1 = new BigDecimal("123.4500");
BigDecimal d2 = d1.stripTrailingZeros();
System.out.println(d1.scale()); // 4
System.out.println(d2.scale()); // 2, 因为去掉了00

BigDecimal d3 = new BigDecimal("1234500");
BigDecimal d4 = d3.stripTrailingZeros();
System.out.println(d3.scale()); // 0
System.out.println(d4.scale()); // -2
```

如果一个BigDecimal的scale()返回负数，例如，-2，表示这个数是个整数，并且末尾有2个0。

可以对一个BigDecimal设置它的scale，如果精度比原始值低，那么按照指定的方法进行四舍五入或者直接截断：

```
import java.math.BigDecimal;
import java.math.RoundingMode;
-----
public class Main {
    public static void main(String[] args) {
        BigDecimal d1 = new BigDecimal("123.456789");
        BigDecimal d2 = d1.setScale(4, RoundingMode.HALF_UP); // 四舍五入, 123.4568
        BigDecimal d3 = d1.setScale(4, RoundingMode.DOWN); // 直接截断, 123.4567
        System.out.println(d2);
        System.out.println(d3);
    }
}
```

对BigDecimal做加、减、乘时，精度不会丢失，但是做除法时，存在无法除尽的情况，这时，就必须指定精度以及如何截断：

```
BigDecimal d1 = new BigDecimal("123.456");
BigDecimal d2 = new BigDecimal("23.456789");
BigDecimal d3 = d1.divide(d2, 10, RoundingMode.HALF_UP); // 保留10位小数并四舍五入
```

在比较两个BigDecimal的值是否相等时，要特别注意，使用equals()方法不但要求两个BigDecimal的值相等，还要求它们的scale()相等：

```
BigDecimal d1 = new BigDecimal("123.456");
BigDecimal d2 = new BigDecimal("123.45600");
System.out.println(d1.equals(d2)); // false, 因为scale不同
System.out.println(d1.equals(d2.stripTrailingZeros())); // true, 因为d2去除尾部0后scale变为2
System.out.println(d1.compareTo(d2)); // 0
```

必须使用compareTo()方法来比较，它根据两个值的大小分别返回负数、正数和0，分别表示小于、大于和等于。

总是使用compareTo()比较两个BigDecimal的值，不要使用equals()！

如果查看BigDecimal的源码，可以发现，实际上一个BigDecimal是通过一个BigInteger和一个scale来表示的，即BigInteger表示一个完整的整数，而scale表示小数位数：

```
public class BigDecimal extends Number implements Comparable<BigDecimal> {  
    private final BigInteger intVal;  
    private final int scale;  
}
```

BigDecimal也是从Number继承的，也是不可变对象。

## 小结

BigDecimal用于表示精确的小数，常用于财务计算；

比较BigDecimal的值是否相等，必须使用compareTo()而不能使用equals()。

Java的核心库提供了大量的现成的类供我们使用。本节我们介绍几个常用的工具类。

## 常用工具类

### Math

顾名思义，Math类就是用来进行数学计算的，它提供了大量的静态方法来便于我们实现数学计算：

求绝对值：

```
Math.abs(-100); // 100
Math.abs(-7.8); // 7.8
```

取最大或最小值：

```
Math.max(100, 99); // 100
Math.min(1.2, 2.3); // 1.2
```

计算 $x^y$ 次方：

```
Math.pow(2, 10); // 2的10次方=1024
```

计算 $\sqrt{x}$ ：

```
Math.sqrt(2); // 1.414...
```

计算 $e^x$ 次方：

```
Math.exp(2); // 7.389...
```

计算以 $e$ 为底的对数：

```
Math.log(4); // 1.386...
```

计算以10为底的对数：

```
Math.log10(100); // 2
```

三角函数：

```
Math.sin(3.14); // 0.00159...
Math.cos(3.14); // -0.9999...
Math.tan(3.14); // -0.0015...
Math.asin(1.0); // 1.57079...
Math.acos(1.0); // 0.0
```

Math还提供了几个数学常量：

```
double pi = Math.PI; // 3.14159...
double e = Math.E; // 2.7182818...
Math.sin(Math.PI / 6); // sin( $\pi/6$ ) = 0.5
```

生成一个随机数 $x$ ， $x$ 的范围是 $0 \leq x < 1$ ：

```
Math.random(); // 0.53907... 每次都不同
```

如果我们要生成一个区间在 $[MIN, MAX)$ 的随机数，可以借助Math.random()实现，计算如下：

```
// 区间在[MIN, MAX)的随机数
public class Main {
    public static void main(String[] args) {
        double x = Math.random(); // x的范围是[0,1)
        double min = 10;
        double max = 50;
        double y = x * (max - min) + min; // y的范围是[10,50)
    }
}
```

```

        long n = (long) y; // n的范围是[10,50)的整数
        System.out.println(y);
        System.out.println(n);
    }
}

```

有些童鞋可能注意到Java标准库还提供了一个StrictMath，它提供了和Math几乎一模一样的方法。这两个类的区别在于，由于浮点数计算存在误差，不同的平台（例如x86和ARM）计算的结果可能不一致（指误差不同），因此，StrictMath保证所有平台计算结果都是完全相同的，而Math会尽量针对平台优化计算速度，所以，绝大多数情况下，使用Math就足够了。

## Random

Random用来创建伪随机数。所谓伪随机数，是指只要给定一个初始的种子，产生的随机数序列是完全一样的。

要生成一个随机数，可以使用nextInt()、nextLong()、nextFloat()、nextDouble()：

```

Random r = new Random();
r.nextInt(); // 2071575453,每次都不一样
r.nextInt(10); // 5,生成一个[0,10)之间的int
r.nextLong(); // 8811649292570369305,每次都不一样
r.nextFloat(); // 0.54335...生成一个[0,1)之间的float
r.nextDouble(); // 0.3716...生成一个[0,1)之间的double

```

有童鞋问，每次运行程序，生成的随机数都是不同的，没看出伪随机数的特性来。

这是因为我们创建Random实例时，如果不给定种子，就使用系统当前时间戳作为种子，因此每次运行时，种子不同，得到的伪随机数序列就不同。

如果我们在创建Random实例时指定一个种子，就会得到完全确定的随机数序列：

```

import java.util.Random;
----
public class Main {
    public static void main(String[] args) {
        Random r = new Random(12345);
        for (int i = 0; i < 10; i++) {
            System.out.println(r.nextInt(100));
        }
        // 51, 80, 41, 28, 55...
    }
}

```

前面我们使用的Math.random()实际上内部调用了Random类，所以它也是伪随机数，只是我们无法指定种子。

## SecureRandom

有伪随机数，就有真随机数。实际上真正的真随机数只能通过量子力学原理来获取，而我们想要的是一个不可预测的安全的随机数，SecureRandom就是用来创建安全的随机数的：

```

SecureRandom sr = new SecureRandom();
System.out.println(sr.nextInt(100));

```

SecureRandom无法指定种子，它使用RNG（random number generator）算法。JDK的SecureRandom实际上有多种不同的底层实现，有的使用安全随机种子加上伪随机数算法来产生安全的随机数，有的使用真正的随机数生成器。实际使用的时候，可以优先获取高强度的安全随机数生成器，如果没有提供，再使用普通等级的安全随机数生成器：

```

import java.util.Arrays;
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
----
public class Main {
    public static void main(String[] args) {
        SecureRandom sr = null;
        try {

```

```
        sr = SecureRandom.getInstanceStrong(); // 获取高强度安全随机数生成器
    } catch (NoSuchAlgorithmException e) {
        sr = new SecureRandom(); // 获取普通的安全随机数生成器
    }
    byte[] buffer = new byte[16];
    sr.nextBytes(buffer); // 用安全随机数填充buffer
    System.out.println(Arrays.toString(buffer));
}
}
```

SecureRandom的安全性是通过操作系统提供的安全的随机种子来生成随机数。这个种子是通过CPU的热噪声、读写磁盘的字节、网络流量等各种随机事件产生的“熵”。

在密码学中，安全的随机数非常重要。如果使用不安全的伪随机数，所有加密体系都将被攻破。因此，时刻牢记必须使用SecureRandom来产生安全的随机数。

需要使用安全随机数的时候，必须使用SecureRandom，绝不能使用Random！

## 小结

Java提供的常用工具类有：

- **Math:** 数学计算
- **Random:** 生成伪随机数
- **SecureRandom:** 生成安全的随机数

程序运行的时候，经常会发生各种错误。

## 异常处理

比如，使用Excel的时候，它有时候会报错：



本章我们讨论如何在Java程序中处理各种异常情况。





在计算机程序运行的过程中，总是会出现各种各样的错误。

## Java的异常

有一些错误是用户造成的，比如，希望用户输入一个int类型的年龄，但是用户的输入是abc：

```
// 假设用户输入了abc:
String s = "abc";
int n = Integer.parseInt(s); // NumberFormatException!
```

程序想要读写某个文件的内容，但是用户已经把它删除了：

```
// 用户删除了该文件:
String t = readFile("C:\\abc.txt"); // FileNotFoundException!
```

还有一些错误是随机出现，并且永远不可能避免的。比如：

- 网络突然断了，连接不到远程服务器；
- 内存耗尽，程序崩溃了；
- 用户点“打印”，但根本没有打印机；
- .....

所以，一个健壮的程序必须处理各种各样的错误。

所谓错误，就是程序调用某个函数的时候，如果失败了，就表示出错。

调用方如何获知调用失败的信息？有两种方法：

方法一：约定返回错误码。

例如，处理一个文件，如果返回0，表示成功，返回其他整数，表示约定的错误码：

```
int code = processFile("C:\\test.txt");
if (code == 0) {
    // ok:
} else {
    // error:
    switch (code) {
        case 1:
            // file not found:
        case 2:
            // no read permission:
        default:
            // unknown error:
    }
}
```

因为使用int类型的错误码，想要处理就非常麻烦。这种方式常见于底层C函数。

方法二：在语言层面上提供一个异常处理机制。

Java内置了一套异常处理机制，总是使用异常来表示错误。

异常是一种class，因此它本身带有类型信息。异常可以在任何地方抛出，但只需要在上层捕获，这样就和调用分离了：

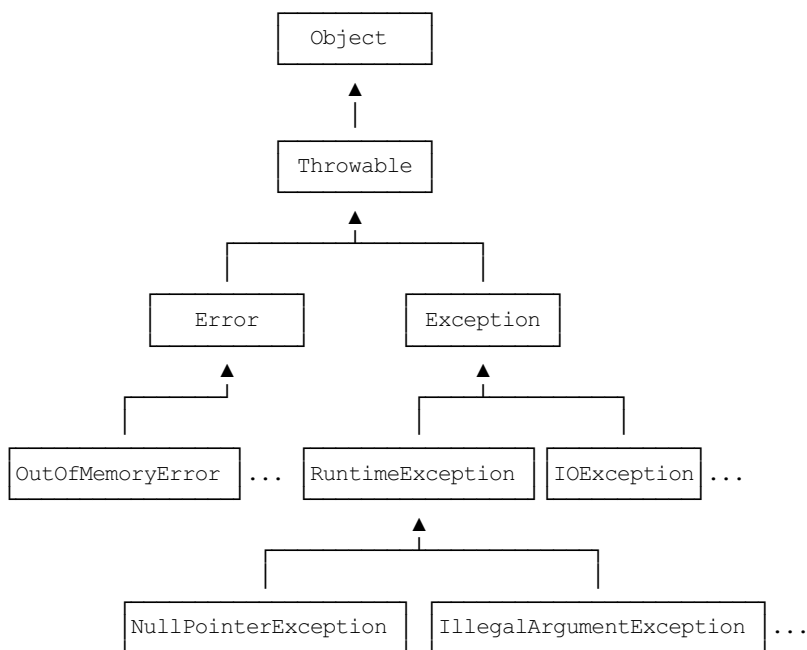
```
try {
    String s = processFile("C:\\test.txt");
    // ok:
} catch (FileNotFoundException e) {
    // file not found:
} catch (SecurityException e) {
    // no read permission:
} catch (IOException e) {
```

```

    // io error:
} catch (Exception e) {
    // other error:
}

```

因为Java的异常是class，它的继承关系如下：



从继承关系可知：Throwable是异常体系的根，它继承自Object。Throwable有两个体系：Error和Exception，Error表示严重的错误，程序对此一般无能为力，例如：

- OutOfMemoryError: 内存耗尽
- NoClassDefFoundError: 无法加载某个Class
- StackOverflowError: 栈溢出

而Exception则是运行时的错误，它可以被捕获并处理。

某些异常是应用程序逻辑处理的一部分，应该捕获并处理。例如：

- NumberFormatException: 数值类型的格式错误
- FileNotFoundException: 未找到文件
- SocketException: 读取网络失败

还有一些异常是程序逻辑编写不对造成的，应该修复程序本身。例如：

- NullPointerException: 对某个null的对象调用方法或字段
- IndexOutOfBoundsException: 数组索引越界

Exception又分为两大类：

1. RuntimeException以及它的子类；
2. 非RuntimeException（包括IOException、ReflectiveOperationException等等）

Java规定：

- 必须捕获的异常，包括Exception及其子类，但不包括RuntimeException及其子类，这种类型的异常称为Checked Exception。
- 不需要捕获的异常，包括Error及其子类，RuntimeException及其子类。

## 捕获异常

捕获异常使用try...catch语句，把可能发生异常的代码放到try {...}中，然后使用catch捕获对应的Exception及其子类：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
----
public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) {
        try {
            // 用指定编码转换String为byte[]:
            return s.getBytes("GBK");
        } catch (UnsupportedEncodingException e) {
            // 如果系统不支持GBK编码，会捕获到UnsupportedEncodingException:
            System.out.println(e); // 打印异常信息
            return s.getBytes(); // 尝试使用默认编码
        }
    }
}
```

如果我们不捕获UnsupportedEncodingException，会出现编译失败的问题：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
----
public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) {
        return s.getBytes("GBK");
    }
}
```

编译器会报错，错误信息类似：unreported exception UnsupportedEncodingException; must be caught or declared to be thrown. 并且准确地指出需要捕获的语句是return s.getBytes("GBK");。意思是说，像UnsupportedEncodingException这样的Checked Exception，必须被捕获。

这是因为String.getBytes(String)方法定义是：

```
public byte[] getBytes(String charsetName) throws UnsupportedEncodingException {
    ...
}
```

在方法定义的时候，使用throws xxx表示该方法可能抛出的异常类型。调用方在调用的时候，必须强制捕获这些异常，否则编译器会报错。

在toGBK()方法中，因为调用了String.getBytes(String)方法，就必须捕获UnsupportedEncodingException。我们也可以不捕获它，而是在方法定义处用throws表示toGBK()方法可能会抛出UnsupportedEncodingException，就可以让toGBK()方法通过编译器检查：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
----
public class Main {
    public static void main(String[] args) {
```

```

        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) throws UnsupportedOperationException {
        return s.getBytes("GBK");
    }
}

```

上述代码仍然会得到编译错误，但这一次，编译器提示的不是调用`return s.getBytes("GBK");`的问题，而是`byte[] bs = toGBK("中文");`。因为在`main()`方法中，调用`toGBK()`，没有捕获它声明的可能抛出的`UnsupportedEncodingException`。

修复方法是在`main()`方法中捕获异常并处理：

```

// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
----
public class Main {
    public static void main(String[] args) {
        try {
            byte[] bs = toGBK("中文");
            System.out.println(Arrays.toString(bs));
        } catch (UnsupportedEncodingException e) {
            System.out.println(e);
        }
    }

    static byte[] toGBK(String s) throws UnsupportedOperationException {
        // 用指定编码转换String为byte[]:
        return s.getBytes("GBK");
    }
}

```

可见，只要是方法声明的**Checked Exception**，不在调用层捕获，也必须在更高的调用层捕获。所有未捕获的异常，最终也必须在`main()`方法中捕获，不会出现漏写`try`的情况。这是由编译器保证的。`main()`方法也是最后捕获`Exception`的机会。

如果是测试代码，上面的写法就略显麻烦。如果不想写任何`try`代码，可以直接把`main()`方法定义为`throws Exception`：

```

// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;
----
public class Main {
    public static void main(String[] args) throws Exception {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) throws UnsupportedOperationException {
        // 用指定编码转换String为byte[]:
        return s.getBytes("GBK");
    }
}

```

因为`main()`方法声明了可能抛出`Exception`，也就声明了可能抛出所有的`Exception`，因此在内部就无需捕获了。代价就是一旦发生异常，程序会立刻退出。

还有一些童鞋喜欢在`toGBK()`内部“消化”异常：

```

static byte[] toGBK(String s) {
    try {
        return s.getBytes("GBK");
    } catch (UnsupportedEncodingException e) {
        // 什么也不干
    }
}

```

```
    }  
    return null;
```

这种捕获后不处理的方式是非常不好的，即使真的什么也做不了，也要先把异常记录下来：

```
static byte[] toGBK(String s) {  
    try {  
        return s.getBytes("GBK");  
    } catch (UnsupportedEncodingException e) {  
        // 先记下来再说：  
        e.printStackTrace();  
    }  
    return null;
```

所有异常都可以调用`printStackTrace()`方法打印异常栈，这是一个简单有用的快速打印异常的方法。

## 小结

Java使用异常来表示错误，并通过`try ... catch`捕获异常；

Java的异常是class，并且从`Throwable`继承；

`Error`是无需捕获的严重错误，`Exception`是应该捕获的可处理的错误；

`RuntimeException`无需强制捕获，非`RuntimeException`（**Checked Exception**）需强制捕获，或者用`throws`声明；

不推荐捕获了异常但不进行任何处理。

在Java中，凡是可能抛出异常的语句，都可以用try ... catch捕获。把可能发生异常的语句放在try { ... }中，然后使用catch捕获对应的Exception及其子类。

## 捕获异常

### 多catch语句

可以使用多个catch语句，每个catch分别捕获对应的Exception及其子类。JVM在捕获到异常后，会从上到下匹配catch语句，匹配到某个catch后，执行catch代码块，然后不再继续匹配。

简单地说就是：多个catch语句只有一个能被执行。例如：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println(e);
    } catch (NumberFormatException e) {
        System.out.println(e);
    }
}
```

存在多个catch的时候，catch的顺序非常重要：子类必须写在前面。例如：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println("IO error");
    } catch (UnsupportedEncodingException e) { // 永远捕获不到
        System.out.println("Bad encoding");
    }
}
```

对于上面的代码，UnsupportedEncodingException异常是永远捕获不到的，因为它是IOException的子类。当抛出UnsupportedEncodingException异常时，会被catch (IOException e) { ... }捕获并执行。

因此，正确的写法是把子类放到前面：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
    } catch (IOException e) {
        System.out.println("IO error");
    }
}
```

### finally语句

无论是否有异常发生，如果我们都希望执行一些语句，例如清理工作，怎么写？

可以把执行语句写若干遍：正常执行的放到try中，每个catch再写一遍。例如：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
    } catch (Exception e) {
        // ...
    } finally {
        // ...
    }
}
```

```

        process3();
        System.out.println("END");
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
        System.out.println("END");
    } catch (IOException e) {
        System.out.println("IO error");
        System.out.println("END");
    }
}

```

上述代码无论是否发生异常，都会执行`System.out.println("END");`这条语句。

那么如何消除这些重复的代码？Java的`try ... catch`机制还提供了`finally`语句，`finally`语句块保证有无错误都会执行。上述代码可以改写如下：

```

public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
    } catch (IOException e) {
        System.out.println("IO error");
    } finally {
        System.out.println("END");
    }
}

```

注意`finally`有几个特点：

1. `finally`语句不是必须的，可写可不写；
2. `finally`总是最后执行。

如果没有发生异常，就正常执行`try { ... }`语句块，然后执行`finally`。如果发生了异常，就中断执行`try { ... }`语句块，然后跳转执行匹配的`catch`语句块，最后执行`finally`。

可见，`finally`是用来保证一些代码必须执行的。

某些情况下，可以没有`catch`，只使用`try ... finally`结构。例如：

```

void process(String file) throws IOException {
    try {
        ...
    } finally {
        System.out.println("END");
    }
}

```

因为方法声明了可能抛出的异常，所以可以不写`catch`。

## 捕获多种异常

如果某些异常的处理逻辑相同，但是异常本身不存在继承关系，那么就得编写多条`catch`子句：

```

public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println("Bad input");
    } catch (NumberFormatException e) {
        System.out.println("Bad input");
    } catch (Exception e) {
        System.out.println("Unknown error");
    }
}

```

```
    }  
}
```

因为处理IOException和NumberFormatException的代码是相同的，所以我们可以把它两用|合并到一起：

```
public static void main(String[] args) {  
    try {  
        process1();  
        process2();  
        process3();  
    } catch (IOException | NumberFormatException e) { // IOException或NumberFormatException  
        System.out.println("Bad input");  
    } catch (Exception e) {  
        System.out.println("Unknown error");  
    }  
}
```

## 练习

用try ... catch捕获异常并处理。

[捕获异常练习](#)

## 小结

使用try ... catch ... finally时：

- 多个catch语句的匹配顺序非常重要，子类必须放在前面；
- finally语句保证了有无异常都会执行，它是可选的；
- 一个catch语句也可以匹配多个非继承关系的异常。



## 异常的传播

# 抛出异常

当某个方法抛出了异常时，如果当前方法没有捕获异常，异常就会被抛到上层调用方法，直到遇到某个`try ... catch`被捕获为止：

```
// exception
----
public class Main {
    public static void main(String[] args) {
        try {
            process1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        process2();
    }

    static void process2() {
        Integer.parseInt(null); // 会抛出NumberFormatException
    }
}
```

通过`printStackTrace()`可以打印出方法的调用栈，类似：

```
java.lang.NumberFormatException: null
    at java.base/java.lang.Integer.parseInt(Integer.java:614)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at Main.process2(Main.java:16)
    at Main.process1(Main.java:12)
    at Main.main(Main.java:5)
```

`printStackTrace()`对于调试错误非常有用，上述信息表示：`NumberFormatException`是在`java.lang.Integer.parseInt`方法中被抛出的，调用层次从上到下依次是：

1. `main()` 调用 `process1()`；
2. `process1()` 调用 `process2()`；
3. `process2()` 调用 `Integer.parseInt(String)`；
4. `Integer.parseInt(String)` 调用 `Integer.parseInt(String, int)`。

查看`Integer.java`源码可知，抛出异常的方法代码如下：

```
public static int parseInt(String s, int radix) throws NumberFormatException {
    if (s == null) {
        throw new NumberFormatException("null");
    }
    ...
}
```

并且，每层调用均给出了源代码的行号，可直接定位。

## 抛出异常

当发生错误时，例如，用户输入了非法的字符，我们就可以抛出异常。

如何抛出异常？参考`Integer.parseInt()`方法，抛出异常分两步：

1. 创建某个`Exception`的实例；
2. 用`throw`语句抛出。

下面是一个例子：

```
void process2(String s) {
    if (s==null) {
```

```

        NullPointerException e = new NullPointerException();
        throw e;
    }
}

```

实际上，绝大部分抛出异常的代码都会合并写成一行：

```

void process2(String s) {
    if (s==null) {
        throw new NullPointerException();
    }
}

```

如果一个方法捕获了某个异常后，又在catch子句中抛出新的异常，就相当于把抛出的异常类型“转换”了：

```

void process1(String s) {
    try {
        process2();
    } catch (NullPointerException e) {
        throw new IllegalArgumentException();
    }
}

void process2(String s) {
    if (s==null) {
        throw new NullPointerException();
    }
}

```

当process2()抛出NullPointerException后，被process1()捕获，然后抛出IllegalArgumentException()。

如果在main()中捕获IllegalArgumentException，我们看看打印的异常栈：

```

// exception
----
public class Main {
    public static void main(String[] args) {
        try {
            process1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        try {
            process2();
        } catch (NullPointerException e) {
            throw new IllegalArgumentException();
        }
    }

    static void process2() {
        throw new NullPointerException();
    }
}

```

打印出的异常栈类似：

```

java.lang.IllegalArgumentException
    at Main.process1(Main.java:15)
    at Main.main(Main.java:5)

```

这说明新的异常丢失了原始异常信息，我们已经看不到原始异常NullPointerException的信息了。

为了能追踪到完整的异常栈，在构造异常的时候，把原始的Exception实例传进去，新的Exception就可以持有原始Exception信息。对上述代码改进如下：

```

// exception
----
public class Main {
    public static void main(String[] args) {
        try {
            process1();

```

```

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        try {
            process2();
        } catch (NullPointerException e) {
            throw new IllegalArgumentException(e);
        }
    }

    static void process2() {
        throw new NullPointerException();
    }
}

```

运行上述代码，打印出的异常栈类似：

```

java.lang.IllegalArgumentException: java.lang.NullPointerException
    at Main.process1(Main.java:15)
    at Main.main(Main.java:5)
Caused by: java.lang.NullPointerException
    at Main.process2(Main.java:20)
    at Main.process1(Main.java:13)

```

注意到Caused by: Xxx，说明捕获的IllegalArgumentException并不是造成问题的根源，根源在于NullPointerException，是在Main.process2()方法抛出的。

在代码中获取原始异常可以使用Throwable.getCause()方法。如果返回null，说明已经是“根异常”了。

有了完整的异常栈的信息，我们才能快速定位并修复代码的问题。

如果我们在try或者catch语句块中抛出异常，finally语句是否会执行？例如：

```

// exception
----
public class Main {
    public static void main(String[] args) {
        try {
            Integer.parseInt("abc");
        } catch (Exception e) {
            System.out.println("caught");
            throw new RuntimeException(e);
        } finally {
            System.out.println("finally");
        }
    }
}

```

上述代码执行结果如下：

```

caught
finally
Exception in thread "main" java.lang.RuntimeException: java.lang.NumberFormatException: For input string: "abc"
    at Main.main(Main.java:8)
Caused by: java.lang.NumberFormatException: For input string: "abc"
    at ...

```

第一行打印了caught，说明进入了catch语句块。第二行打印了finally，说明执行了finally语句块。

因此，在catch中抛出异常，不会影响finally的执行。JVM会先执行finally，然后抛出异常。

## 异常屏蔽

如果在执行finally语句时抛出异常，那么，catch语句的异常还能否继续抛出？例如：

```

// exception
----
public class Main {
    public static void main(String[] args) {

```

```

        try {
            Integer.parseInt("abc");
        } catch (Exception e) {
            System.out.println("caught");
            throw new RuntimeException(e);
        } finally {
            System.out.println("finally");
            throw new IllegalArgumentException();
        }
    }
}

```

执行上述代码，发现异常信息如下：

```

caught
finally
Exception in thread "main" java.lang.IllegalArgumentException
    at Main.main(Main.java:11)

```

这说明finally抛出异常后，原来在catch中准备抛出的异常就“消失”了，因为只能抛出一个异常。没有被抛出的异常称为“被屏蔽”的异常（**Suppressed Exception**）。

在极少数的情况下，我们需要获知所有的异常。如何保存所有的异常信息？方法是先用origin变量保存原始异常，然后调用Throwable.addSuppressed()，把原始异常添加进来，最后在finally抛出：

```

// exception
----
public class Main {
    public static void main(String[] args) throws Exception {
        Exception origin = null;
        try {
            System.out.println(Integer.parseInt("abc"));
        } catch (Exception e) {
            origin = e;
            throw e;
        } finally {
            Exception e = new IllegalArgumentException();
            if (origin != null) {
                e.addSuppressed(origin);
            }
            throw e;
        }
    }
}

```

当catch和finally都抛出了异常时，虽然catch的异常被屏蔽了，但是，finally抛出的异常仍然包含了它：

```

Exception in thread "main" java.lang.IllegalArgumentException
    at Main.main(Main.java:11)
Suppressed: java.lang.NumberFormatException: For input string: "abc"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at Main.main(Main.java:6)

```

通过Throwable.getSuppressed()可以获取所有的Suppressed Exception。

绝大多数情况下，在finally中不要抛出异常。因此，我们通常不需要关心Suppressed Exception。

## 练习

如果传入的参数为负，则抛出IllegalArgumentException。

[抛出异常练习](#)

## 小结

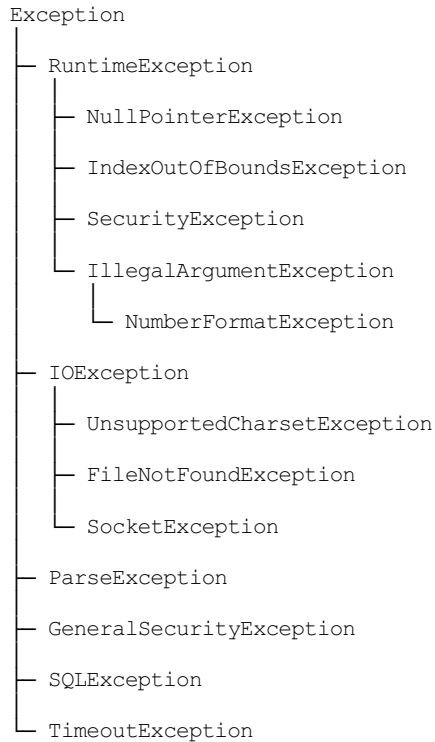
调用printStackTrace()可以打印异常的传播栈，对于调试非常有用；

捕获异常并再次抛出新的异常时，应该持有原始异常信息；

通常不要在finally中抛出异常。如果在finally中抛出异常，应该原始异常加入到原有异常中。调用方可通过Throwable.getSuppressed() 获取所有添加的Suppressed Exception。

Java标准库定义的常用异常包括：

## 自定义异常



当我们在代码中需要抛出异常时，尽量使用JDK已定义的异常类型。例如，参数检查不合法，应该抛出IllegalArgumentException：

```
static void process1(int age) {
    if (age <= 0) {
        throw new IllegalArgumentException();
    }
}
```

在一个大型项目中，可以自定义新的异常类型，但是，保持一个合理的异常继承体系是非常重要的。

一个常见的做法是自定义一个BaseException作为“根异常”，然后，派生出各种业务类型的异常。

BaseException需要从一个适合的Exception派生，通常建议从RuntimeException派生：

```
public class BaseException extends RuntimeException {
}
```

其他业务类型的异常就可以从BaseException派生：

```
public class UserNotFoundException extends BaseException {
}
```

```
public class LoginFailedException extends BaseException {
}
```

...

自定义的BaseException应该提供多个构造方法：

```
public class BaseException extends RuntimeException {
    public BaseException() {
        super();
    }
}
```

```
}

public BaseException(String message, Throwable cause) {
    super(message, cause);
}

public BaseException(String message) {
    super(message);
}

public BaseException(Throwable cause) {
    super(cause);
}
}
```

上述构造方法实际上都是原样照抄`RuntimeException`。这样，抛出异常的时候，就可以选择合适的构造方法。通过IDE可以根据父类快速生成子类的构造方法。

## 练习

[从BaseException派生自定义异常](#)

## 小结

抛出异常时，尽量复用JDK已定义的异常类型；

自定义异常体系时，推荐从`RuntimeException`派生“根异常”，再派生出业务异常；

自定义异常时，应该提供多种构造方法。

断言（Assertion）是一种调试程序的方式。在Java中，使用assert关键字来实现断言。

## 使用断言

我们先看一个例子：

```
public static void main(String[] args) {
    double x = Math.abs(-123.45);
    assert x >= 0;
    System.out.println(x);
}
```

语句`assert x >= 0;`即为断言，断言条件`x >= 0`预期为`true`。如果计算结果为`false`，则断言失败，抛出`AssertionError`。

使用`assert`语句时，还可以添加一个可选的断言消息：

```
assert x >= 0 : "x must >= 0";
```

这样，断言失败的时候，`AssertionError`会带上消息`x must >= 0`，更加便于调试。

**Java断言的特点是：**断言失败时会抛出`AssertionError`，导致程序结束退出。因此，断言不能用于可恢复的程序错误，只应该用于开发和测试阶段。

对于可恢复的程序错误，不应该使用断言。例如：

```
void sort(int[] arr) {
    assert arr != null;
}
```

应该抛出异常并在上层捕获：

```
void sort(int[] arr) {
    if (x == null) {
        throw new IllegalArgumentException("array cannot be null");
    }
}
```

当我们在程序中使用`assert`时，例如，一个简单的断言：

```
// assert
----
public class Main {
    public static void main(String[] args) {
        int x = -1;
        assert x > 0;
        System.out.println(x);
    }
}
```

断言`x`必须大于0，实际上`x`为-1，断言肯定失败。执行上述代码，发现程序并未抛出`AssertionError`，而是正常打印了`x`的值。

这是怎么肥四？为什么`assert`语句不起作用？

这是因为JVM默认关闭断言指令，即遇到`assert`语句就自动忽略了，不执行。

要执行`assert`语句，必须给Java虚拟机传递`-enableassertions`（可简写为`-ea`）参数启用断言。所以，上述程序必须在命令行下运行才有效果：

```
$ java -ea Main.java
Exception in thread "main" java.lang.AssertionError
    at Main.main(Main.java:5)
```



还可以有选择地对特定地类启用断言，命令行参数是：`-ea:com.itranswarp.sample.Main`，表示只对`com.itranswarp.sample.Main`这个类启用断言。

或者对特定地包启用断言，命令行参数是：`-ea:com.itranswarp.sample...`（注意结尾有3个.），表示对`com.itranswarp.sample`这个包启动断言。

实际开发中，很少使用断言。更好的方法是编写单元测试，后续我们会讲解JUnit的使用。

## 小结

断言是一种调试方式，断言失败会抛出`AssertionError`，只能在开发和测试阶段启用断言；

对可恢复的错误不能使用断言，而应该抛出异常；

断言很少被使用，更好的方法是编写单元测试。

在编写程序的过程中，发现程序运行结果与预期不符，怎么办？当然是用`System.out.println()`打印出执行过程中的某些变量，观察每一步的结果与代码逻辑是否符合，然后有针对性地修改代码。

## 使用JDK Logging

代码改好了怎么办？当然是删除没有用的`System.out.println()`语句了。

如果改代码又改出问题怎么办？再加上`System.out.println()`。

反复这么搞几次，很快大家就发现使用`System.out.println()`非常麻烦。

怎么办？

解决方法是使用日志。

那什么是日志？日志就是**Logging**，它的目的是为了取代`System.out.println()`。

输出日志，而不是用`System.out.println()`，有以下几个好处：

1. 可以设置输出样式，避免自己每次都写`"ERROR: " + var;`
2. 可以设置输出级别，禁止某些级别输出。例如，只输出错误日志；
3. 可以被重定向到文件，这样可以在程序运行结束后查看日志；
4. 可以按包名控制日志级别，只输出某些包打的日志；
5. 可以.....

总之就是好处很多啦。

那如何使用日志？

因为Java标准库内置了日志包`java.util.logging`，我们可以直接用。先看一个简单的例子：

```
// logging
import java.util.logging.Level;
import java.util.logging.Logger;
----
public class Hello {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger();
        logger.info("start process...");
        logger.warning("memory is running out...");
        logger.fine("ignored.");
        logger.severe("process will be terminated...");
    }
}
```

运行上述代码，得到类似如下的输出：

```
Mar 02, 2019 6:32:13 PM Hello main
INFO: start process...
Mar 02, 2019 6:32:13 PM Hello main
WARNING: memory is running out...
Mar 02, 2019 6:32:13 PM Hello main
SEVERE: process will be terminated...
```

对比可见，使用日志最大的好处是，它自动打印了时间、调用类、调用方法等很多有用的信息。

再仔细观察发现，4条日志，只打印了3条，`logger.fine()`没有打印。这是因为，日志的输出可以设定级别。JDK的Logging定义了7个日志级别，从严重到普通：

- SEVERE
- WARNING
- INFO
- CONFIG

- FINE
- FINER
- FINEST

因为默认级别是INFO，因此，INFO级别以下的日志，不会被打印出来。使用日志级别的好处在于，调整级别，就可以屏蔽掉很多调试相关的日志输出。

使用Java标准库内置的Logging有以下局限：

Logging系统在JVM启动时读取配置文件并完成初始化，一旦开始运行main()方法，就无法修改配置；

配置不太方便，需要在JVM启动时传递参数-Djava.util.logging.config.file=<config-file-name>。

因此，Java标准库内置的Logging使用并不是非常广泛。更方便的日志系统我们稍后介绍。

## 练习

使用logger.severe()打印异常：

```
import java.io.UnsupportedEncodingException;
import java.util.logging.Logger;

public class Main {
    public static void main(String[] args) {
        ----
        Logger logger = Logger.getLogger(Main.class.getName());
        logger.info("Start process...");
        try {
            "".getBytes("invalidCharsetName");
        } catch (UnsupportedEncodingException e) {
            // TODO: 使用logger.severe()打印异常
        }
        logger.info("Process end.");
        ----
    }
}
```

## [打印异常](#)

## 小结

日志是为了替代System.out.println()，可以定义格式，重定向到文件等；

日志可以存档，便于追踪问题；

日志记录可以按级别分类，便于打开或关闭某些级别；

可以根据配置文件调整日志，无需修改代码；

Java标准库提供了java.util.logging来实现日志功能。

和Java标准库提供的日志不同，Commons Logging是一个第三方日志库，它是由Apache创建的日志模块。

## 使用Commons Logging

Commons Logging的特色是，它可以挂接不同的日志系统，并通过配置文件指定挂接的日志系统。默认情况下，Commons Logging自动搜索并使用Log4j（Log4j是另一个流行的日志系统），如果没有找到Log4j，再使用JDK Logging。

使用Commons Logging只需要和两个类打交道，并且只有两步：

第一步，通过LogFactory获取Log类的实例；第二步，使用Log实例的方法打日志。

示例代码如下：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
----
public class Main {
    public static void main(String[] args) {
        Log log = LogFactory.getLog(Main.class);
        log.info("start...");
        log.warn("end.");
    }
}
```

运行上述代码，肯定会得到编译错误，类似error: package org.apache.commons.logging does not exist（找不到org.apache.commons.logging这个包）。因为Commons Logging是一个第三方提供的库，所以，必须先把它[下载](#)下来。下载后，解压，找到commons-logging-1.2.jar这个文件，再把Java源码Main.java放到一个目录下，例如work目录：

```
work
├── commons-logging-1.2.jar
└── Main.java
```

然后用javac编译Main.java，编译的时候要指定classpath，不然编译器找不到我们引用的org.apache.commons.logging包。编译命令如下：

```
javac -cp commons-logging-1.2.jar Main.java
```

如果编译成功，那么当前目录下就会多出一个Main.class文件：

```
work
├── commons-logging-1.2.jar
├── Main.java
└── Main.class
```

现在可以执行这个Main.class，使用java命令，也必须指定classpath，命令如下：

```
java -cp .;commons-logging-1.2.jar Main
```

注意到传入的classpath有两部分：一个是.，一个是commons-logging-1.2.jar，用;分割。.表示当前目录，如果没有这个.，JVM不会在当前目录搜索Main.class，就会报错。

如果在Linux或macOS下运行，注意classpath的分隔符不是;，而是::

```
java -cp .:commons-logging-1.2.jar Main
```

运行结果如下：

```
Mar 02, 2019 7:15:31 PM Main main
INFO: start...
Mar 02, 2019 7:15:31 PM Main main
WARNING: end.
```

**Commons Logging**定义了6个日志级别：

- FATAL
- ERROR
- WARNING
- INFO
- DEBUG
- TRACE

默认级别是INFO。

使用**Commons Logging**时，如果在静态方法中引用Log，通常直接定义一个静态类型变量：

```
// 在静态方法中引用Log：
public class Main {
    static final Log log = LoggerFactory.getLog(Main.class);

    static void foo() {
        log.info("foo");
    }
}
```

在实例方法中引用Log，通常定义一个实例变量：

```
// 在实例方法中引用Log：
public class Person {
    protected final Log log = LoggerFactory.getLog(getClass());

    void foo() {
        log.info("foo");
    }
}
```

注意到实例变量log的获取方式是LogFactory.getLog(getClass())，虽然也可以用LogFactory.getLog(Person.class)，但是前一种方式有个非常大的好处，就是子类可以直接使用该log实例。例如：

```
// 在子类中使用父类实例化的log：
public class Student extends Person {
    void bar() {
        log.info("bar");
    }
}
```

由于Java类的动态特性，子类获取的log字段实际上相当于LogFactory.getLog(Student.class)，但是从父类继承而来，并且无需改动代码。

此外，**Commons Logging**的日志方法，例如info()，除了标准的info(String)外，还提供了一个非常有用的重载方法：info(String, Throwable)，这使得记录异常更加简单：

```
try {
    ...
} catch (Exception e) {
    log.error("got exception!", e);
}
```

## 练习

使用log.error(String, Throwable)打印异常。

[Commons Logging](#)练习

## 小结

Commons Logging是使用最广泛的日志模块；

Commons Logging的API非常简单；

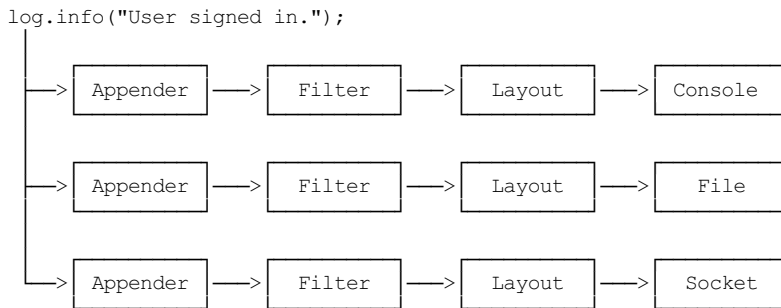
Commons Logging可以自动检测并使用其他日志模块。

前面介绍了Commons Logging，可以作为“日志接口”来使用。而真正的“日志实现”可以使用Log4j。

## 使用Log4j

Log4j是一种非常流行的日志框架，最新版本是2.x。

Log4j是一个组件化设计的日志系统，它的架构大致如下：



当我们使用Log4j输出一条日志时，Log4j自动通过不同的Appender把同一条日志输出到不同的目的地。例如：

- **console:** 输出到屏幕；
- **file:** 输出到文件；
- **socket:** 通过网络输出到远程计算机；
- **jdbc:** 输出到数据库

在输出日志的过程中，通过Filter来过滤哪些log需要被输出，哪些log不需要被输出。例如，仅输出ERROR级别的日志。

最后，通过Layout来格式化日志信息，例如，自动添加日期、时间、方法名称等信息。

上述结构虽然复杂，但我们在实际使用的时候，并不需要关心Log4j的API，而是通过配置文件来配置它。

以XML配置为例，使用Log4j的时候，我们把一个log4j2.xml的文件放到classpath下就可以让Log4j读取配置文件并按照我们的配置来输出日志。下面是一个配置文件的例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Properties>
    <!-- 定义日志格式 -->
    <Property name="log.pattern">%d{MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36}%nmsg%n%n</Property>
    <!-- 定义文件名变量 -->
    <Property name="file.err.filename">log/err.log</Property>
    <Property name="file.err.pattern">log/err.%i.log.gz</Property>
  </Properties>
  <!-- 定义Appender，即目的地 -->
  <Appenders>
    <!-- 定义输出到屏幕 -->
    <Console name="console" target="SYSTEM_OUT">
      <!-- 日志格式引用上面定义的log.pattern -->
      <PatternLayout pattern="${log.pattern}" />
    </Console>
    <!-- 定义输出到文件，文件名引用上面定义的file.err.filename -->
    <RollingFile name="err" bufferedIO="true" fileName="${file.err.filename}" filePattern="${file.err.pattern}">
      <PatternLayout pattern="${log.pattern}" />
      <Policies>
        <!-- 根据文件大小自动切割日志 -->
        <SizeBasedTriggeringPolicy size="1 MB" />
      </Policies>
      <!-- 保留最近10份 -->
      <DefaultRolloverStrategy max="10" />
    </RollingFile>
  </Appenders>
  <Loggers>
    <Root level="info">
      <!-- 对info级别的日志，输出到console -->
      <AppenderRef ref="console" level="info" />
      <!-- 对error级别的日志，输出到err，即上面定义的RollingFile -->
```

```
<AppenderRef ref="err" level="error" />
</Root>
</Loggers>
</Configuration>
```

虽然配置Log4j比较繁琐，但一旦配置完成，使用起来就非常方便。对上面的配置文件，凡是INFO级别的日志，会自动输出到屏幕，而ERROR级别的日志，不但会输出到屏幕，还会同时输出到文件。并且，一旦日志文件达到指定大小（1MB），Log4j就会自动切割新的日志文件，并最多保留10份。

有了配置文件还不够，因为Log4j也是一个第三方库，我们需要从[这里](#)下载Log4j，解压后，把以下3个jar包放到classpath中：

- log4j-api-2.x.jar
- log4j-core-2.x.jar
- log4j-jcl-2.x.jar

因为Commons Logging会自动发现并使用Log4j，所以，把上一节下载的commons-logging-1.2.jar也放到classpath中。

要打印日志，只需要按Commons Logging的写法写，不需要改动任何代码，就可以得到Log4j的日志输出，类似：

```
03-03 12:09:45.880 [main] INFO com.itranswarp.learnjava.Main
Start process...
```

## 最佳实践

在开发阶段，始终使用Commons Logging接口来写入日志，并且开发阶段无需引入Log4j。如果需要把日志写入文件，只需要把正确的配置文件和Log4j相关的jar包放入classpath，就可以自动把日志切换成使用Log4j写入，无需修改任何代码。

## 练习

根据配置文件，观察Log4j写入的日志文件。

[commons logging + log4j](#)

## 小结

通过Commons Logging实现日志，不需要修改代码即可使用Log4j 使用Log4j只需要把log4j2.xml和相关jar放入classpath 如果要更换Log4j，只需要移除log4j2.xml和相关jar 只有扩展Log4j时，才需要引用Log4j的接口（例如，将日志加密写入数据库的功能，需要自己开发）



前面介绍了Commons Logging和Log4j这一对好基友，它们一个负责充当日志API，一个负责实现日志底层，搭配使用非常便于开发。

## 使用SLF4J和Logback

有的童鞋可能还听说过SLF4J和Logback。这两个东东看上去也像日志，它们又是啥？

其实SLF4J类似于Commons Logging，也是一个日志接口，而Logback类似于Log4j，是一个日志的实现。

为什么有了Commons Logging和Log4j，又会蹦出来SLF4J和Logback？

这是因为Java有着非常悠久的开源历史，不但OpenJDK本身是开源的，而且我们用到的第三方库，几乎全部都是开源的。

开源生态丰富的一个特定就是，同一个功能，可以找到若干种互相竞争的开源库。

因为对Commons Logging的接口不满意，有人就搞了SLF4J。因为对Log4j的性能不满意，有人就搞了Logback。

我们先来看看SLF4J对Commons Logging的接口有何改进。

在Commons Logging中，我们要打印日志，有时候得这么写：

```
int score = 99;
p.setScore(score);
log.info("Set score " + score + " for Person " + p.getName() + " ok.");
```

拼字符串是一个非常麻烦的事情，所以SLF4J的日志接口改进成这样了：

```
int score = 99;
p.setScore(score);
logger.info("Set score {} for Person {} ok.", score, p.getName());
```

我们靠猜也能猜出来，SLF4J的日志接口传入的是一个带占位符的字符串，用后面的变量自动替换占位符，所以看起来更加自然。

如何使用SLF4J？它的接口实际上和Commons Logging几乎一模一样：

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class Main {
    final Logger logger = LoggerFactory.getLogger(getClass());
}
```

对比一下Commons Logging和SLF4J的接口：

Commons Logging	SLF4J
org.apache.commons.logging.Log	org.slf4j.Logger
org.apache.commons.logging.LogFactory	org.slf4j.LoggerFactory

不同之处就是Log变成了Logger，LogFactory变成了LoggerFactory。

使用SLF4J和Logback和前面讲到的使用Commons Logging加Log4j是类似的，先分别下载[SLF4J](#)和[Logback](#)，然后把以下jar包放到classpath下：

- slf4j-api-1.7.x.jar
- logback-classic-1.2.x.jar
- logback-core-1.2.x.jar

然后使用SLF4J的Logger和LoggerFactory即可。和Log4j类似，我们仍然需要一个Logback的配置文件，把logback.xml放到classpath下，配置如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration>

<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
    <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
  </encoder>
</appender>

<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <encoder>
    <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
    <charset>utf-8</charset>
  </encoder>
  <file>log/output.log</file>
  <rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
    <fileNamePattern>log/output.log.%i</fileNamePattern>
  </rollingPolicy>
  <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
    <MaxFileSize>1MB</MaxFileSize>
  </triggeringPolicy>
</appender>

<root level="INFO">
  <appender-ref ref="CONSOLE" />
  <appender-ref ref="FILE" />
</root>
</configuration>

```

运行即可获得类似如下的输出：

```
13:15:25.328 [main] INFO com.itranswarp.learnjava.Main - Start process...
```

从目前的趋势来看，越来越多的开源项目从Commons Logging加Log4j转向了SLF4J加Logback。

## 练习

根据配置文件，观察Logback写入的日志文件。

[slf4j+logback](#)

## 小结

SLF4J和Logback可以取代Commons Logging和Log4j；

始终使用SLF4J的接口写入日志，使用Logback只需要配置，不需要修改代码。

什么是反射？

# 反射

反射就是**Reflection**，Java的反射是指程序在运行期可以拿到一个对象的所有信息。

正常情况下，如果我们要调用一个对象的方法，或者访问一个对象的字段，通常会传入对象实例：

```
// Main.java
import com.itranswarp.learnjava.Person;

public class Main {
    String getFullName(Person p) {
        return p.getFirstName() + " " + p.getLastName();
    }
}
```

但是，如果不能获得Person类，只有一个Object实例，比如这样：

```
String getFullName(Object obj) {
    return ???
}
```

怎么办？有童鞋会说：强制转型啊！

```
String getFullName(Object obj) {
    Person p = (Person) obj;
    return p.getFirstName() + " " + p.getLastName();
}
```

强制转型的时候，你会发现一个问题：编译上面的代码，仍然需要引用Person类。不然，去掉import语句，你看能不能编译通过？

所以，反射是为了解决在运行期，对某个实例一无所知的情况下，如何调用其方法。



除了int等基本类型外，Java的其他类型全部都是class（包括interface）。例如：

## Class类

- String
- Object
- Runnable
- Exception
- ...

仔细思考，我们可以得出结论：class（包括interface）的本质是数据类型（Type）。无继承关系的数据类型无法赋值：

```
Number n = new Double(123.456); // OK
String s = new Double(123.456); // compile error!
```

而class是由JVM在执行过程中动态加载的。JVM在第一次读取到一种class类型时，将其加载进内存。

每加载一种class，JVM就为其创建一个Class类型的实例，并关联起来。注意：这里的Class类型是一个名叫Class的class。它长这样：

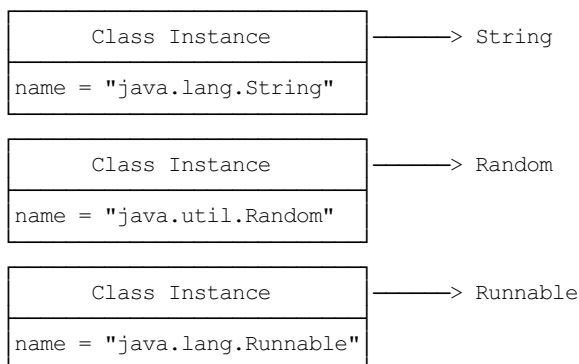
```
public final class Class {
    private Class() {}
}
```

以String类为例，当JVM加载String类时，它首先读取String.class文件到内存，然后，为String类创建一个Class实例并关联起来：

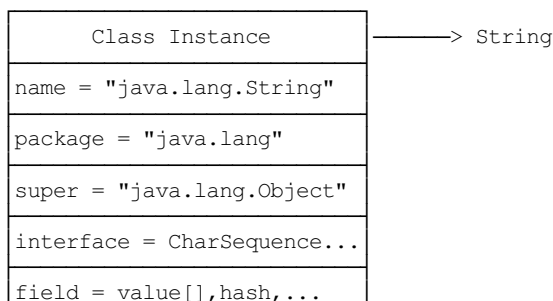
```
Class cls = new Class(String);
```

这个Class实例是JVM内部创建的，如果我们查看JDK源码，可以发现Class类的构造方法是private，只有JVM能创建Class实例，我们自己的Java程序是无法创建Class实例的。

所以，JVM持有的每个Class实例都指向一个数据类型（class或interface）：



一个Class实例包含了该class的所有完整信息：



```
method = indexOf()...
```

由于JVM为每个加载的class创建了对应的Class实例，并在实例中保存了该class的所有信息，包括类名、包名、父类、实现的接口、所有方法、字段等，因此，如果获取了某个Class实例，我们就可以通过这个Class实例获取到该实例对应的class的所有信息。

这种通过Class实例获取class信息的方法称为反射（**Reflection**）。

如何获取一个class的Class实例？有三个方法：

方法一：直接通过一个class的静态变量class获取：

```
Class cls = String.class;
```

方法二：如果我们有一个实例变量，可以通过该实例变量提供的getClass()方法获取：

```
String s = "Hello";
Class cls = s.getClass();
```

方法三：如果知道一个class的完整类名，可以通过静态方法Class.forName()获取：

```
Class cls = Class.forName("java.lang.String");
```

因为Class实例在JVM中是唯一的，所以，上述方法获取的Class实例是同一个实例。可以用==比较两个Class实例：

```
Class cls1 = String.class;

String s = "Hello";
Class cls2 = s.getClass();

boolean sameClass = cls1 == cls2; // true
```

注意一下Class实例比较和instanceof的差别：

```
Integer n = new Integer(123);

boolean b3 = n instanceof Integer; // true
boolean b4 = n instanceof Number; // true

boolean b1 = n.getClass() == Integer.class; // true
boolean b2 = n.getClass() == Number.class; // false
```

用instanceof不但匹配当前类型，还匹配当前类型的子类。而用==判断class实例可以精确地判断数据类型，但不能作子类型比较。

通常情况下，我们应该用instanceof判断数据类型，因为面向抽象编程的时候，我们不关心具体的子类型。只有在需要精确判断一个类型是不是某个class的时候，我们才使用==判断class实例。

因为反射的目的是为了获得某个实例的信息。因此，当我们拿到某个Object实例时，我们可以通过反射获取该Object的class信息：

```
void printObjectInfo(Object obj) {
    Class cls = obj.getClass();
}
```

要从Class实例获取获取的基本信息，参考下面的代码：

```
// reflection
----
public class Main {
    public static void main(String[] args) {
        printClassInfo("").getClass();
        printClassInfo(Runnable.class);
        printClassInfo(java.time.Month.class);
        printClassInfo(String[].class);
    }
}
```

```

        printClassInfo(int.class);
    }

    static void printClassInfo(Class cls) {
        System.out.println("Class name: " + cls.getName());
        System.out.println("Simple name: " + cls.getSimpleName());
        if (cls.getPackage() != null) {
            System.out.println("Package name: " + cls.getPackage().getName());
        }
        System.out.println("is interface: " + cls.isInterface());
        System.out.println("is enum: " + cls.isEnum());
        System.out.println("is array: " + cls.isArray());
        System.out.println("is primitive: " + cls.isPrimitive());
    }
}

```

注意到数组（例如String[]）也是一种Class，而且不同于String.class，它的类名是[Ljava.lang.String。此外，JVM为每一种基本类型如int也创建了Class，通过int.class访问。

如果获取到了一个Class实例，我们就可以通过该Class实例来创建对应类型的实例：

```

// 获取String的Class实例：
Class cls = String.class;
// 创建一个String实例：
String s = (String) cls.newInstance();

```

上述代码相当于new String()。通过Class.newInstance()可以创建类实例，它的局限是：只能调用public的无参数构造方法。带参数的构造方法，或者非public的构造方法都无法通过Class.newInstance()被调用。

## 动态加载

JVM在执行Java程序的时候，并不是一性把所有用到的class全部加载到内存，而是第一次需要用到class时才加载。例如：

```

// Main.java
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            create(args[0]);
        }
    }

    static void create(String name) {
        Person p = new Person(name);
    }
}

```

当执行Main.java时，由于用到了Main，因此，JVM首先会把Main.class加载到内存。然而，并不会加载Person.class，除非程序执行到create()方法，JVM发现需要加载Person类时，才会首次加载Person.class。如果没有执行create()方法，那么Person.class根本就不会被加载。

这就是JVM动态加载class的特性。

动态加载class的特性对于Java程序非常重要。利用JVM动态加载class的特性，我们才能在运行期根据条件加载不同的实现类。例如，Commons Logging总是优先使用Log4j，只有当Log4j不存在时，才使用JDK的logging。利用JVM动态加载特性，大致的实现代码如下：

```

// Commons Logging优先使用Log4j:
LogFactory factory = null;
if (isClassPresent("org.apache.logging.log4j.Logger")) {
    factory = createLog4j();
} else {
    factory = createJdkLog();
}

boolean isClassPresent(String name) {
    try {

```

```
        Class.forName(name);  
        return true;  
    } catch (Exception e) {  
        return false;  
    }  
}
```

这就是为什么我们只需要把Log4j的jar包放到classpath中，Commons Logging就会自动使用Log4j的原因。

## 小结

JVM为每个加载的class及interface创建了对应的Class实例来保存class及interface的所有信息；

获取一个class对应的Class实例后，就可以获取该class的所有信息；

通过Class实例获取class信息的方法称为反射（Reflection）；

JVM总是动态加载class，可以在运行期根据条件来控制加载class。

对任意的一个Object实例，只要我们获取了它的Class，就可以获取它的一切信息。

## 访问字段

我们先看看如何通过Class实例获取字段信息。Class类提供了以下几个方法来获取字段：

- **Field getField(name):** 根据字段名获取某个public的field（包括父类）
- **Field getDeclaredField(name):** 根据字段名获取当前类的某个field（不包括父类）
- **Field[] getFields():** 获取所有public的field（包括父类）
- **Field[] getDeclaredFields():** 获取当前类的所有field（不包括父类）

我们来看一下示例代码：

```
// reflection
----
public class Main {
    public static void main(String[] args) throws Exception {
        Class stdClass = Student.class;
        // 获取public字段"score":
        System.out.println(stdClass.getField("score"));
        // 获取继承的public字段"name":
        System.out.println(stdClass.getField("name"));
        // 获取private字段"grade":
        System.out.println(stdClass.getDeclaredField("grade"));
    }
}

class Student extends Person {
    public int score;
    private int grade;
}

class Person {
    public String name;
}
```

上述代码首先获取Student的Class实例，然后，分别获取public字段、继承的public字段以及private字段，打印出的Field类似：

```
public int Student.score
public java.lang.String Person.name
private int Student.grade
```

一个Field对象包含了一个字段的所有信息：

- **getName():** 返回字段名称，例如，"name"；
- **getType():** 返回字段类型，也是一个Class实例，例如，String.class；
- **getModifiers():** 返回字段的修饰符，它是一个int，不同的bit表示不同的含义。

以String类的value字段为例，它的定义是：

```
public final class String {
    private final byte[] value;
}
```

我们用反射获取该字段的信息，代码如下：

```
Field f = String.class.getDeclaredField("value");
f.getName(); // "value"
f.getType(); // class [B 表示byte[]类型
int m = f.getModifiers();
Modifier.isFinal(m); // true
Modifier.isPublic(m); // false
Modifier.isProtected(m); // false
Modifier.isPrivate(m); // true
```



```
Modifier.isStatic(m); // false
```

## 获取字段值

利用反射拿到字段的一个Field实例只是第一步，我们还可以拿到一个实例对应的该字段的值。

例如，对于一个Person实例，我们可以先拿到name字段对应的Field，再获取这个实例的name字段的值：

```
// reflection
import java.lang.reflect.Field;
----
public class Main {

    public static void main(String[] args) throws Exception {
        Object p = new Person("Xiao Ming");
        Class c = p.getClass();
        Field f = c.getDeclaredField("name");
        Object value = f.get(p);
        System.out.println(value); // "Xiao Ming"
    }
}

class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }
}
```

上述代码先获取Class实例，再获取Field实例，然后，用Field.get(Object)获取指定实例的指定字段的值。

运行代码，如果不出意外，会得到一个IllegalAccessException，这是因为name被定义为一个private字段，正常情况下，Main类无法访问Person类的private字段。要修复错误，可以将private改为public，或者，在调用Object value = f.get(p);前，先写一句：

```
f.setAccessible(true);
```

调用Field.setAccessible(true)的意思是，别管这个字段是不是public，一律允许访问。

可以试着加上上述语句，再运行代码，就可以打印出private字段的值。

有童鞋会问：如果使用反射可以获取private字段的值，那么类的封装还有什么意义？

答案是正常情况下，我们总是通过p.name来访问Person的name字段，编译器会根据public、protected和private决定是否允许访问字段，这样就达到了数据封装的目的。

而反射是一种非常规的用法，使用反射，首先代码非常繁琐，其次，它更多地是给工具或者底层框架来使用，目的是在不知道目标实例任何信息的情况下，获取特定字段的值。

此外，setAccessible(true)可能会失败。如果JVM运行期存在SecurityManager，那么它会根据规则进行检查，有可能阻止setAccessible(true)。例如，某个SecurityManager可能不允许对java和javax开头的package的类调用setAccessible(true)，这样可以保证JVM核心库的安全。

## 设置字段值

通过Field实例既然可以获取到指定实例的字段值，自然也可以设置字段的值。

设置字段值是通过Field.set(Object, Object)实现的，其中第一个Object参数是指定的实例，第二个Object参数是待修改的值。示例代码如下：

```
// reflection
import java.lang.reflect.Field;
----
public class Main {
```

```

    public static void main(String[] args) throws Exception {
        Person p = new Person("Xiao Ming");
        System.out.println(p.getName()); // "Xiao Ming"
        Class c = p.getClass();
        Field f = c.getDeclaredField("name");
        f.setAccessible(true);
        f.set(p, "Xiao Hong");
        System.out.println(p.getName()); // "Xiao Hong"
    }
}

class Person {
    private String name;

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

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

```

运行上述代码，打印的name字段从Xiao Ming变成了Xiao Hong，说明通过反射可以直接修改字段的值。

同样的，修改非public字段，需要首先调用setAccessible(true)。

## 练习

利用反射给字段赋值：[reflect-field](#)

## 小结

Java的反射API提供的Field类封装了字段的所有信息：

通过Class实例的方法可以获取Field实

例：getField(), getFields(), getDeclaredField(), getDeclaredFields();

通过Field实例可以获取字段信息：getName(), getType(), getModifiers();

通过Field实例可以读取或设置某个对象的字段，如果存在访问限制，要首先调用setAccessible(true)来访问非public字段。

通过反射读写字段是一种非常规方法，它会破坏对象的封装。

我们已经能通过Class实例获取所有Field对象，同样的，可以通过Class实例获取所有Method信息。Class类提供了以下几个方法来获取Method：

## 调用方法

- Method `getMethod(name, Class...)`：获取某个public的Method（包括父类）
- Method `getDeclaredMethod(name, Class...)`：获取当前类的某个Method（不包括父类）
- Method[] `getMethods()`：获取所有public的Method（包括父类）
- Method[] `getDeclaredMethods()`：获取当前类的所有Method（不包括父类）

我们来看一下示例代码：

```
// reflection
----
public class Main {
    public static void main(String[] args) throws Exception {
        Class stdClass = Student.class;
        // 获取public方法getScore，参数为String：
        System.out.println(stdClass.getMethod("getScore", String.class));
        // 获取继承的public方法getName，无参数：
        System.out.println(stdClass.getMethod("getName"));
        // 获取private方法getGrade，参数为int：
        System.out.println(stdClass.getDeclaredMethod("getGrade", int.class));
    }
}

class Student extends Person {
    public int getScore(String type) {
        return 99;
    }
    private int getGrade(int year) {
        return 1;
    }
}

class Person {
    public String getName() {
        return "Person";
    }
}
```

上述代码首先获取Student的Class实例，然后，分别获取public方法、继承的public方法以及private方法，打印出的Method类似：

```
public int Student.getScore(java.lang.String)
public java.lang.String Person.getName()
private int Student.getGrade(int)
```

一个Method对象包含一个方法的所有信息：

- `getName()`：返回方法名称，例如："getScore"；
- `getReturnType()`：返回方法返回值类型，也是一个Class实例，例如：String.class；
- `getParameterTypes()`：返回方法的参数类型，是一个Class数组，例如：{String.class, int.class}；
- `getModifiers()`：返回方法的修饰符，它是一个int，不同的bit表示不同的含义。

## 调用方法

当我们获取到一个Method对象时，就可以对它进行调用。我们以下面的代码为例：

```
String s = "Hello world";
String r = s.substring(6); // "world"
```

如果用反射来调用substring方法，需要以下代码：

```
// reflection
import java.lang.reflect.Method;
----
public class Main {
    public static void main(String[] args) throws Exception {
        // String对象:
        String s = "Hello world";
        // 获取String substring(int)方法, 参数为int:
        Method m = String.class.getMethod("substring", int.class);
        // 在s对象上调用该方法并获取结果:
        String r = (String) m.invoke(s, 6);
        // 打印调用结果:
        System.out.println(r);
    }
}
```

注意到substring()有两个重载方法, 我们获取的是String substring(int)这个方法。思考一下如何获取String substring(int, int)方法。

对Method实例调用invoke就相当于调用该方法, invoke的第一个参数是对象实例, 即在哪个实例上调用该方法, 后面的可变参数要与方法参数一致, 否则将报错。

## 调用静态方法

如果获取到的Method表示一个静态方法, 调用静态方法时, 由于无需指定实例对象, 所以invoke方法传入的第一个参数永远为null。我们以Integer.parseInt(String)为例:

```
// reflection
import java.lang.reflect.Method;
----
public class Main {
    public static void main(String[] args) throws Exception {
        // 获取Integer.parseInt(String)方法, 参数为String:
        Method m = Integer.class.getMethod("parseInt", String.class);
        // 调用该静态方法并获取结果:
        Integer n = (Integer) m.invoke(null, "12345");
        // 打印调用结果:
        System.out.println(n);
    }
}
```

## 调用非public方法

和Field类似, 对于非public方法, 我们虽然可以通过Class.getDeclaredMethod()获取该方法实例, 但直接对其调用将得到一个IllegalAccessException。为了调用非public方法, 我们通过Method.setAccessible(true)允许其调用:

```
// reflection
import java.lang.reflect.Method;
----
public class Main {
    public static void main(String[] args) throws Exception {
        Person p = new Person();
        Method m = p.getClass().getDeclaredMethod("setName", String.class);
        m.setAccessible(true);
        m.invoke(p, "Bob");
        System.out.println(p.name);
    }
}

class Person {
    String name;
    private void setName(String name) {
        this.name = name;
    }
}
```

此外, setAccessible(true)可能会失败。如果JVM运行期存在SecurityManager, 那么它会根据规则进行检查, 有可

能阻止`setAccessible(true)`。例如，某个`SecurityManager`可能不允许对`java`和`javax`开头的`package`的类调用`setAccessible(true)`，这样可以保证JVM核心库的安全。

## 多态

我们来考察这样一种情况：一个`Person`类定义了`hello()`方法，并且它的子类`Student`也覆写了`hello()`方法，那么，从`Person.class`获取的`Method`，作用于`Student`实例时，调用的方法到底是哪个？

```
// reflection
import java.lang.reflect.Method;
----
public class Main {
    public static void main(String[] args) throws Exception {
        // 获取Person的hello方法：
        Method h = Person.class.getMethod("hello");
        // 对Student实例调用hello方法：
        h.invoke(new Student());
    }
}

class Person {
    public void hello() {
        System.out.println("Person:hello");
    }
}

class Student extends Person {
    public void hello() {
        System.out.println("Student:hello");
    }
}
```

运行上述代码，发现打印出的是`Student:hello`，因此，使用反射调用方法时，仍然遵循多态原则：即总是调用实际类型的覆写方法（如果存在）。上述的反射代码：

```
Method m = Person.class.getMethod("hello");
m.invoke(new Student());
```

实际上相当于：

```
Person p = new Student();
p.hello();
```

## 练习

利用反射调用方法：[reflect-method](#)

## 小结

Java的反射API提供的`Method`对象封装了方法的所有信息：

通过`Class`实例的方法可以获取`Method`实

例：`getMethod()`，`getMethods()`，`getDeclaredMethod()`，`getDeclaredMethods()`；

通过`Method`实例可以获取方法信息：`getName()`，`getReturnType()`，`getParameterTypes()`，`getModifiers()`；

通过`Method`实例可以调用某个对象的方法：`Object invoke(Object instance, Object... parameters)`；

通过设置`setAccessible(true)`来访问非`public`方法；

通过反射调用方法时，仍然遵循多态原则。

我们通常使用new操作符创建新的实例：

## 调用构造方法

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

如果通过反射来创建新的实例，可以调用Class提供的newInstance()方法：

```
Person p = Person.class.newInstance();
```

调用Class.newInstance()的局限是，它只能调用该类的public无参数构造方法。如果构造方法带有参数，或者不是public，就无法直接通过Class.newInstance()来调用。

为了调用任意的构造方法，Java的反射API提供了Constructor对象，它包含一个构造方法的所有信息，可以创建一个实例。Constructor对象和方法非常类似，不同之处仅在于它是一个构造方法，并且，调用结果总是返回实例：

```
import java.lang.reflect.Constructor;
----
public class Main {
    public static void main(String[] args) throws Exception {
        // 获取构造方法Integer(int):
        Constructor cons1 = Integer.class.getConstructor(int.class);
        // 调用构造方法:
        Integer n1 = (Integer) cons1.newInstance(123);
        System.out.println(n1);

        // 获取构造方法Integer(String)
        Constructor cons2 = Integer.class.getConstructor(String.class);
        Integer n2 = (Integer) cons2.newInstance("456");
        System.out.println(n2);
    }
}
```

通过Class实例获取Constructor的方法如下：

- getConstructor(Class...)：获取某个public的Constructor；
- getDeclaredConstructor(Class...)：获取某个Constructor；
- getConstructors()：获取所有public的Constructor；
- getDeclaredConstructors()：获取所有Constructor。

注意Constructor总是当前类定义的构造方法，和父类无关，因此不存在多态的问题。

调用非public的Constructor时，必须首先通过setAccessible(true)设置允许访问。setAccessible(true)可能会失败。

## 小结

Constructor对象封装了构造方法的所有信息；

通过Class实例的方法可以获取Constructor实例：

例：getConstructor()，getConstructors()，getDeclaredConstructor()，getDeclaredConstructors()；

通过Constructor实例可以创建一个实例对象：newInstance(Object... parameters)；通过设置setAccessible(true)来访问非public构造方法。

当我们获取到某个Class对象时，实际上就获取到了一个类的类型：

## 获取继承关系

```
Class cls = String.class; // 获取到String的Class
```

还可以用实例的getClass()方法获取：

```
String s = "";
Class cls = s.getClass(); // s是String，因此获取到String的Class
```

最后一种获取Class的方法是通过Class.forName("")，传入Class的完整类名获取：

```
Class s = Class.forName("java.lang.String");
```

这三种方式获取的Class实例都是同一个实例，因为JVM对每个加载的Class只创建一个Class实例来表示它的类型。

### 获取父类的Class

有了Class实例，我们还可以获取它的父类的Class：

```
// reflection
----
public class Main {
    public static void main(String[] args) throws Exception {
        Class i = Integer.class;
        Class n = i.getSuperclass();
        System.out.println(n);
        Class o = n.getSuperclass();
        System.out.println(o);
        System.out.println(o.getSuperclass());
    }
}
```

运行上述代码，可以看到，Integer的父类类型是Number，Number的父类是Object，Object的父类是null。除Object外，其他任何非interface的Class都必定存在一个父类类型。

### 获取interface

由于一个类可能实现一个或多个接口，通过Class我们就可以查询到实现的接口类型。例如，查询Integer实现的接口：

```
// reflection
import java.lang.reflect.Method;
----
public class Main {
    public static void main(String[] args) throws Exception {
        Class s = Integer.class;
        Class[] is = s.getInterfaces();
        for (Class i : is) {
            System.out.println(i);
        }
    }
}
```

运行上述代码可知，Integer实现的接口有：

- java.lang.Comparable
- java.lang.constant.Constable
- java.lang.constant.ConstantDesc

要特别注意：getInterfaces()只返回当前类直接实现的接口类型，并不包括其父类实现的接口类型：

```
// reflection
import java.lang.reflect.Method;
----
public class Main {
    public static void main(String[] args) throws Exception {
        Class s = Integer.class.getSuperclass();
        Class[] is = s.getInterfaces();
        for (Class i : is) {
            System.out.println(i);
        }
    }
}
```

Integer的父类是Number，Number实现的接口是java.io.Serializable。

此外，对所有interface的Class调用getSuperclass()返回的是其父interface或者null：

```
System.out.println(java.io.DataInputStream.class.getSuperclass()); // java.io.FilterInputStream，因为DataInputStream继承自FilterInputStream
System.out.println(java.io.Closeable.class.getSuperclass()); // null，对接口调用getSuperclass()总是返回null，获取接口的父接口要用getInterfaces()
```

如果一个类没有实现任何interface，那么getInterfaces()返回空数组。

### 继承关系

当我们判断一个实例是否是某个类型时，正常情况下，使用instanceof操作符：

```
Object n = Integer.valueOf(123);
boolean isDouble = n instanceof Double; // false
boolean isInteger = n instanceof Integer; // true
boolean isNumber = n instanceof Number; // true
boolean isSerializable = n instanceof java.io.Serializable; // true
```

如果是两个Class实例，要判断一个向上转型是否成立，可以调用isAssignableFrom()：

```
// Integer i = ?
Integer.class.isAssignableFrom(Integer.class); // true, 因为Integer可以赋值给Integer
// Number n = ?
Number.class.isAssignableFrom(Integer.class); // true, 因为Integer可以赋值给Number
// Object o = ?
Object.class.isAssignableFrom(Integer.class); // true, 因为Integer可以赋值给Object
// Integer i = ?
Integer.class.isAssignableFrom(Number.class); // false, 因为Number不能赋值给Integer
```

## 小结

通过Class对象可以获取继承关系：

- Class getSuperclass()：获取父类类型；
- Class[] getInterfaces()：获取当前类实现的所有接口。

通过Class对象的isAssignableFrom()方法可以判断一个向上转型是否可以实现。



我们来比较Java的class和interface的区别：

## 动态代理

- 可以实例化class（非abstract）；
- 不能实例化interface。

所有interface类型的变量总是通过向上转型并指向某个实例的：

```
CharSequence cs = new StringBuilder();
```

有没有可能不编写实现类，直接在运行期创建某个interface的实例呢？

这是可能的，因为Java标准库提供了一种动态代理（Dynamic Proxy）的机制：可以在运行期动态创建某个interface的实例。

什么叫运行期动态创建？听起来好像很复杂。所谓动态代理，是和静态相对应的。我们来看静态代码怎么写：

定义接口：

```
public interface Hello {  
    void morning(String name);  
}
```

编写实现类：

```
public class HelloWorld implements Hello {  
    public void morning(String name) {  
        System.out.println("Good morning, " + name);  
    }  
}
```

创建实例，转型为接口并调用：

```
Hello hello = new HelloWorld();  
hello.morning("Bob");
```

这种方式就是我们通常编写代码的方式。

还有一种方式是动态代码，我们仍然先定义了接口Hello，但是我们并不去编写实现类，而是直接通过JDK提供的一个Proxy.newProxyInstance()创建了一个Hello接口对象。这种没有实现类但是在运行期动态创建了一个接口对象的方式，我们称为动态代码。JDK提供的动态创建接口对象的方式，就叫动态代理。

一个最简单的动态代理实现如下：

```
import java.lang.reflect.InvocationHandler;  
import java.lang.reflect.Method;  
import java.lang.reflect.Proxy;  
----  
public class Main {  
    public static void main(String[] args) {  
        InvocationHandler handler = new InvocationHandler() {  
            @Override  
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
                System.out.println(method);  
                if (method.getName().equals("morning")) {  
                    System.out.println("Good morning, " + args[0]);  
                }  
                return null;  
            }  
        };  
        Hello hello = (Hello) Proxy.newProxyInstance(  
            Hello.class.getClassLoader(), // 传入ClassLoader  
            new Class[] { Hello.class }, // 传入要实现的接口  
            handler); // 传入处理调用方法的InvocationHandler
```

```
        hello.morning("Bob");
    }
}

interface Hello {
    void morning(String name);
}
```

在运行期动态创建一个interface实例的方法如下：

1. 定义一个InvocationHandler实例，它负责实现接口的方法调用；
2. 通过Proxy.newProxyInstance()创建interface实例，它需要3个参数：
  1. 使用的ClassLoader，通常就是接口类的ClassLoader；
  2. 需要实现的接口数组，至少需要传入一个接口进去；
  3. 用来处理接口方法调用的InvocationHandler实例。
3. 将返回的Object强制转型为接口。

动态代理实际上是JDK在运行期动态创建class字节码并加载的过程，它并没有什么黑魔法，把上面的动态代理改写为静态实现类大概长这样：

```
public class HelloDynamicProxy implements Hello {
    InvocationHandler handler;
    public HelloDynamicProxy(InvocationHandler handler) {
        this.handler = handler;
    }
    public void morning(String name) {
        handler.invoke(
            this,
            Hello.class.getMethod("morning"),
            new Object[] { name });
    }
}
```

其实就是JDK帮我们自动编写了一个上述类（不需要源码，可以直接生成字节码），并不存在可以直接实例化接口的黑魔法。

## 小结

Java标准库提供了动态代理功能，允许在运行期动态创建一个接口的实例；

动态代理是通过Proxy创建代理对象，然后将接口方法“代理”给InvocationHandler完成的。

本节我们将介绍Java程序的一种特殊“注释”——注解（Annotation）。

## 注解



什么是注解（Annotation）？注解是放在Java源码的类、方法、字段、参数前的一种特殊“注释”：

## 使用注解

```
// this is a component:
@Resource("hello")
public class Hello {
    @Inject
    int n;

    @PostConstruct
    public void hello(@Param String name) {
        System.out.println(name);
    }

    @Override
    public String toString() {
        return "Hello";
    }
}
```

注解会被编译器直接忽略，注解则可以被编译器打包进入class文件，因此，注解是一种用作标注的“元数据”。

### 注解的作用

从JVM的角度看，注解本身对代码逻辑没有任何影响，如何使用注解完全由工具决定。

Java的注解可以分为三类：

第一类是由编译器使用的注解，例如：

- `@Override`：让编译器检查该方法是否正确地实现了覆写；
- `@SuppressWarnings`：告诉编译器忽略此处代码产生的警告。

这类注解不会被编译进入.class文件，它们在编译后就被编译器扔掉了。

第二类是由工具处理.class文件使用的注解，比如有些工具会在加载class的时候，对class做动态修改，实现一些特殊的功能。这类注解会被编译进入.class文件，但加载结束后并不会存在于内存中。这类注解只被一些底层库使用，一般我们不必自己处理。

第三类是在程序运行期能够读取的注解，它们在加载后一直存在于JVM中，这也是最常用的注解。例如，一个配置了`@PostConstruct`的方法会在调用构造方法后自动被调用（这是Java代码读取该注解实现的功能，JVM并不会识别该注解）。

定义一个注解时，还可以定义配置参数。配置参数可以包括：

- 所有基本类型；
- `String`；
- 枚举类型；
- 基本类型、`String`以及枚举的数组。

因为配置参数必须是常量，所以，上述限制保证了注解在定义时就已经确定了每个参数的值。

注解的配置参数可以有默认值，缺少某个配置参数时将使用默认值。

此外，大部分注解会有一个名为value的配置参数，对此参数赋值，可以只写常量，相当于省略了value参数。

如果只写注解，相当于全部使用默认值。

举个栗子，对以下代码：

```
public class Hello {
```

```
@Check(min=0, max=100, value=55)
public int n;

@Check(value=99)
public int p;

@Check(99) // @Check(value=99)
public int x;

@Check
public int y;
}
```

@Check就是一个注解。第一个@Check(min=0, max=100, value=55)明确定义三个参数，第二个@Check(value=99)只定义了一个value参数，它实际上和@Check(99)是完全一样的。最后一个@Check表示所有参数都使用默认值。

## 小结

注解（Annotation）是Java语言用于工具处理的标注：

注解可以配置参数，没有指定配置参数使用默认值；

如果参数名称是value，且只有一个参数，那么可以省略参数名称。

Java语言使用@interface语法来定义注解（Annotation），它的格式如下：

## 定义注解

```
public @interface Report {  
    int type() default 0;  
    String level() default "info";  
    String value() default "";  
}
```

注解的参数类似无参数方法，可以用default设定一个默认值（强烈推荐）。最常用的参数应当命名为value。

### 元注解

有一些注解可以修饰其他注解，这些注解就称为元注解（meta annotation）。Java标准库已经定义了一些元注解，我们只需要使用元注解，通常不需要自己去编写元注解。

#### @Target

最常用的元注解是@Target。使用@Target可以定义Annotation能够被应用于源码的哪些位置：

- 类或接口：ElementType.TYPE;
- 字段：ElementType.FIELD;
- 方法：ElementType.METHOD;
- 构造方法：ElementType.CONSTRUCTOR;
- 方法参数：ElementType.PARAMETER。

例如，定义注解@Report可用在方法上，我们必须添加一个@Target (ElementType.METHOD)：

```
@Target (ElementType.METHOD)  
public @interface Report {  
    int type() default 0;  
    String level() default "info";  
    String value() default "";  
}
```

定义注解@Report可用在方法或字段上，可以把@Target注解参数变为数组{ ElementType.METHOD, ElementType.FIELD }：

```
@Target({  
    ElementType.METHOD,  
    ElementType.FIELD  
})  
public @interface Report {  
    ...  
}
```

实际上@Target定义的value是ElementType[]数组，只有一个元素时，可以省略数组的写法。

#### @Retention

另一个重要的元注解@Retention定义了Annotation的生命周期：

- 仅编译期：RetentionPolicy.SOURCE;
- 仅class文件：RetentionPolicy.CLASS;
- 运行期：RetentionPolicy.RUNTIME。

如果@Retention不存在，则该Annotation默认为CLASS。因为通常我们自定义的Annotation都是RUNTIME，所以，务必要加上@Retention(RetentionPolicy.RUNTIME)这个元注解：

```
@Retention(RetentionPolicy.RUNTIME)
```

```
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

## **@Repeatable**

使用@Repeatable这个元注解可以定义Annotation是否可重复。这个注解应用不是特别广泛。

```
@Repeatable
@Target(ElementType.TYPE)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

经过@Repeatable修饰后，在某个类型声明处，就可以添加多个@Report注解：

```
@Report(type=1, level="debug")
@Report(type=2, level="warning")
public class Hello {
}
```

## **@Inherited**

使用@Inherited定义子类是否可继承父类定义的Annotation。@Inherited仅针对@Target(ElementType.TYPE)类型的annotation有效，并且仅针对class的继承，对interface的继承无效：

```
@Inherited
@Target(ElementType.TYPE)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

在使用的时候，如果一个类用到了@Report：

```
@Report(type=1)
public class Person {
}
```

则它的子类默认也定义了该注解：

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

## **如何定义Annotation**

我们总结一下定义Annotation的步骤：

第一步，用@interface定义注解：

```
public @interface Report {
}
```

第二步，添加参数、默认值：

```
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

把最常用的参数定义为value()，推荐所有参数都尽量设置默认值。

第三步，用元注解配置注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

其中，必须设置@Target和@Retention，@Retention一般设置为RUNTIME，因为我们自定义的注解通常要求在运行期读取。一般情况下，不必写@Inherited和@Repeatable。

## 小结

Java使用@interface定义注解：

可定义多个参数和默认值，核心参数使用value名称；

必须设置@Target来指定Annotation可以应用的范围；

应当设置@Retention(RetentionPolicy.RUNTIME)便于运行期读取该Annotation。



Java的注解本身对代码逻辑没有任何影响。根据@Retention的配置：

## 处理注解

- SOURCE类型的注解在编译期就被丢掉了；
- CLASS类型的注解仅保存在class文件中，它们不会被加载进JVM；
- RUNTIME类型的注解会被加载进JVM，并且在运行期可以被程序读取。

如何使用注解完全由工具决定。SOURCE类型的注解主要由编译器使用，因此我们一般只使用，不编写。CLASS类型的注解主要由底层工具库使用，涉及到class的加载，一般我们很少用到。只有RUNTIME类型的注解不但要使用，还经常需要编写。

因此，我们只讨论如何读取RUNTIME类型的注解。

因为注解定义后也是一种class，所有的注解都继承自java.lang.annotation.Annotation，因此，读取注解，需要使用反射API。

Java提供的使用反射API读取Annotation的方法包括：

判断某个注解是否存在于Class、Field、Method或Constructor：

- Class.isAnnotationPresent(Class)
- Field.isAnnotationPresent(Class)
- Method.isAnnotationPresent(Class)
- Constructor.isAnnotationPresent(Class)

例如：

```
// 判断@Report是否存在于Person类：
Person.class.isAnnotationPresent(Report.class);
```

使用反射API读取Annotation：

- Class.getAnnotation(Class)
- Field.getAnnotation(Class)
- Method.getAnnotation(Class)
- Constructor.getAnnotation(Class)

例如：

```
// 获取Person定义的@Report注解：
Report report = Person.class.getAnnotation(Report.class);
int type = report.type();
String level = report.level();
```

使用反射API读取Annotation有两种方法。方法一是先判断Annotation是否存在，如果存在，就直接读取：

```
Class cls = Person.class;
if (cls.isAnnotationPresent(Report.class)) {
    Report report = cls.getAnnotation(Report.class);
    ...
}
```

第二种方法是直接读取Annotation，如果Annotation不存在，将返回null：

```
Class cls = Person.class;
Report report = cls.getAnnotation(Report.class);
if (report != null) {
    ...
}
```

读取方法、字段和构造方法的Annotation和Class类似。但要读取方法参数的Annotation就比较麻烦一点，因为方法参

数本身可以看成是一个数组，而每个参数又可以定义多个注解，所以，一次获取方法参数的所有注解就必须用一个二维数组来表示。例如，对于以下方法定义的注解：

```
public void hello(@NotNull @Range(max=5) String name, @NotNull String prefix) {  
}
```

要读取方法参数的注解，我们先用反射获取Method实例，然后读取方法参数的所有注解：

```
// 获取Method实例：  
Method m = ...  
// 获取所有参数的Annotation：  
Annotation[][] annos = m.getParameterAnnotations();  
// 第一个参数（索引为0）的所有Annotation：  
Annotation[] annosOfName = annos[0];  
for (Annotation anno : annosOfName) {  
    if (anno instanceof Range) { // @Range注解  
        Range r = (Range) anno;  
    }  
    if (anno instanceof NotNull) { // @NotNull注解  
        NotNull n = (NotNull) anno;  
    }  
}
```

## 使用注解

注解如何使用，完全由程序自己决定。例如，JUnit是一个测试框架，它会自动运行所有标记为@Test的方法。

我们来看一个@Range注解，我们希望用它来定义一个String字段的规则：字段长度满足@Range的参数定义：

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.FIELD)  
public @interface Range {  
    int min() default 0;  
    int max() default 255;  
}
```

在某个JavaBean中，我们可以使用该注解：

```
public class Person {  
    @Range(min=1, max=20)  
    public String name;  
  
    @Range(max=10)  
    public String city;  
}
```

但是，定义了注解，本身对程序逻辑没有任何影响。我们必须自己编写代码来使用注解。这里，我们编写一个Person实例的检查方法，它可以检查Person实例的String字段长度是否满足@Range的定义：

```
void check(Person person) throws IllegalArgumentException, ReflectiveOperationException {  
    // 遍历所有Field：  
    for (Field field : person.getClass().getFields()) {  
        // 获取Field定义的@Range：  
        Range range = field.getAnnotation(Range.class);  
        // 如果@Range存在：  
        if (range != null) {  
            // 获取Field的值：  
            Object value = field.get(person);  
            // 如果值是String：  
            if (value instanceof String) {  
                String s = (String) value;  
                // 判断值是否满足@Range的min/max：  
                if (s.length() < range.min() || s.length() > range.max()) {  
                    throw new IllegalArgumentException("Invalid field: " + field.getName());  
                }  
            }  
        }  
    }  
}
```

```
}
```

这样一来，我们通过@Range注解，配合check()方法，就可以完成Person实例的检查。注意检查逻辑完全是我们自己编写的，JVM不会自动给注解添加任何额外的逻辑。

## 练习

使用@Range注解来检查Java Bean的字段。如果字段类型是String，就检查String的长度，如果字段是int，就检查int的范围。

[annotation-range-check](#)

## 小结

可以在运行期通过反射读取RUNTIME类型的注解，注意千万不要漏写@Retention(RetentionPolicy.RUNTIME)，否则运行期无法读取到该注解。

可以通过程序处理注解来实现相应的功能：

- 对JavaBean的属性值按规则进行检查；
- JUnit会自动运行@Test标记的测试方法。

泛型是一种“代码模板”，可以用一套代码套用各种类型。

## 泛型



**Cup<T>**

**Cup<Water>**

**Cup<Coffee>**

**Cup<Tea>**

本节我们详细讨论Java的泛型编程。

在讲解什么是泛型之前，我们先观察Java标准库提供的ArrayList，它可以看作“可变长度”的数组，因为用起来比数组更方便。

## 什么是泛型

实际上ArrayList内部就是一个Object[]数组，配合存储一个当前分配的长度，就可以充当“可变数组”：

```
public class ArrayList {
    private Object[] array;
    private int size;
    public void add(Object e) {...}
    public void remove(int index) {...}
    public Object get(int index) {...}
}
```

如果用上述ArrayList存储String类型，会有这么几个缺点：

- 需要强制转型；
- 不方便，易出错。

例如，代码必须这么写：

```
ArrayList list = new ArrayList();
list.add("Hello");
// 获取到Object，必须强制转型为String：
String first = (String) list.get(0);
```

很容易出现ClassCastException，因为容易“误转型”：

```
list.add(new Integer(123));
// ERROR: ClassCastException:
String second = (String) list.get(1);
```

要解决上述问题，我们可以为String单独编写一种ArrayList：

```
public class StringArrayList {
    private String[] array;
    private int size;
    public void add(String e) {...}
    public void remove(int index) {...}
    public String get(int index) {...}
}
```

这样一来，存入的必须是String，取出的也一定是String，不需要强制转型，因为编译器会强制检查放入的类型：

```
StringArrayList list = new StringArrayList();
list.add("Hello");
String first = list.get(0);
// 编译错误：不允许放入非String类型：
list.add(new Integer(123));
```

问题暂时解决。

然而，新的问题是，如果要存储Integer，还需要为Integer单独编写一种ArrayList：

```
public class IntegerArrayList {
    private Integer[] array;
    private int size;
    public void add(Integer e) {...}
    public void remove(int index) {...}
    public Integer get(int index) {...}
}
```

实际上，还需要为其他所有class单独编写一种ArrayList：

- LongArrayList
- DoubleArrayList
- PersonArrayList
- ...

这是不可能的，JDK的class就有上千个，而且它还不知道其他人编写的class。

为了解决新的问题，我们必须把ArrayList变成一种模板：ArrayList<T>，代码如下：

```
public class ArrayList<T> {
    private T[] array;
    private int size;
    public void add(T e) {...}
    public void remove(int index) {...}
    public T get(int index) {...}
}
```

T可以是任何class。这样一来，我们就实现了：编写一次模版，可以创建任意类型的ArrayList：

```
// 创建可以存储String的ArrayList:
ArrayList<String> strList = new ArrayList<String>();
// 创建可以存储Float的ArrayList:
ArrayList<Float> floatList = new ArrayList<Float>();
// 创建可以存储Person的ArrayList:
ArrayList<Person> personList = new ArrayList<Person>();
```

因此，泛型就是定义一种模板，例如ArrayList<T>，然后在代码中为用到的类创建对应的ArrayList<类型>：

```
ArrayList<String> strList = new ArrayList<String>();
```

由编译器针对类型作检查：

```
strList.add("hello"); // OK
String s = strList.get(0); // OK
strList.add(new Integer(123)); // compile error!
Integer n = strList.get(0); // compile error!
```

这样一来，既实现了编写一次，万能匹配，又通过编译器保证了类型安全：这就是泛型。

## 向上转型

在Java标准库中的ArrayList<T>实现了List<T>接口，它可以向上转型为List<T>：

```
public class ArrayList<T> implements List<T> {
    ...
}
```

```
List<String> list = new ArrayList<String>();
```

即类型ArrayList<T>可以向上转型为List<T>。

要特别注意：不能把ArrayList<Integer>向上转型为ArrayList<Number>或List<Number>。

这是为什么呢？假设ArrayList<Integer>可以向上转型为ArrayList<Number>，观察一下代码：

```
// 创建ArrayList<Integer>类型:
ArrayList<Integer> integerList = new ArrayList<Integer>();
// 添加一个Integer:
integerList.add(new Integer(123));
// “向上转型”为ArrayList<Number>:
ArrayList<Number> numberList = integerList;
// 添加一个Float，因为Float也是Number:
numberList.add(new Float(12.34));
// 从ArrayList<Integer>获取索引为1的元素（即添加的Float）:
Integer n = integerList.get(1); // ClassCastException!
```

我们把一个`ArrayList<Integer>`转型为`ArrayList<Number>`类型后，这个`ArrayList<Number>`就可以接受`Float`类型，因为`Float`是`Number`的子类。但是，`ArrayList<Number>`实际上和`ArrayList<Integer>`是同一个对象，也就是`ArrayList<Integer>`类型，它不可能接受`Float`类型，所以在获取`Integer`的时候将产生`ClassCastException`。

实际上，编译器为了避免这种错误，根本就不允许把`ArrayList<Integer>`转型为`ArrayList<Number>`。

`ArrayList<Integer>`和`ArrayList<Number>`两者完全没有继承关系。

## 小结

泛型就是编写模板代码来适应任意类型；

泛型的好处是使用时不必对类型进行强制转换，它通过编译器对类型进行检查；

注意泛型的继承关系：可以把`ArrayList<Integer>`向上转型为`List<Integer>`（`T`不能变！），但不能把`ArrayList<Integer>`向上转型为`ArrayList<Number>`（`T`不能变成父类）。

使用ArrayList时，如果不定义泛型类型时，泛型类型实际上就是Object：

## 使用泛型

```
// 编译器警告：
List list = new ArrayList();
list.add("Hello");
list.add("World");
String first = (String) list.get(0);
String second = (String) list.get(1);
```

此时，只能把<T>当作Object使用，没有发挥泛型的优势。

当我们定义泛型类型<String>后，List<T>的泛型接口变为强类型List<String>：

```
// 无编译器警告：
List<String> list = new ArrayList<String>();
list.add("Hello");
list.add("World");
// 无强制转型：
String first = list.get(0);
String second = list.get(1);
```

当我们定义泛型类型<Number>后，List<T>的泛型接口变为强类型List<Number>：

```
List<Number> list = new ArrayList<Number>();
list.add(new Integer(123));
list.add(new Double(12.34));
Number first = list.get(0);
Number second = list.get(1);
```

编译器如果能自动推断出泛型类型，就可以省略后面的泛型类型。例如，对于下面的代码：

```
List<Number> list = new ArrayList<Number>();
```

编译器看到泛型类型List<Number>就可以自动推断出后面的ArrayList<T>的泛型类型必须是ArrayList<Number>，因此，可以把代码简写为：

```
// 可以省略后面的Number，编译器可以自动推断泛型类型：
List<Number> list = new ArrayList<>();
```

### 泛型接口

除了ArrayList<T>使用了泛型，还可以在接口中使用泛型。例如，Arrays.sort(Object[])可以对任意数组进行排序，但待排序的元素必须实现Comparable<T>这个泛型接口：

```
public interface Comparable<T> {
    /**
     * 返回-1：当前实例比参数o小
     * 返回0：当前实例与参数o相等
     * 返回1：当前实例比参数o大
     */
    int compareTo(T o);
}
```

可以直接对String数组进行排序：

```
// sort
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        ----
        String[] ss = new String[] { "Orange", "Apple", "Pear" };
        Arrays.sort(ss);
        System.out.println(Arrays.toString(ss));
    }
}
```



```

----
    }
}

```

这是因为String本身已经实现了Comparable<String>接口。如果换成我们自定义的Person类型试试：

```

// sort
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        ----
        Person[] ps = new Person[] {
            new Person("Bob", 61),
            new Person("Alice", 88),
            new Person("Lily", 75),
        };
        Arrays.sort(ps);
        System.out.println(Arrays.toString(ps));
        ----
    }
}

class Person {
    String name;
    int score;
    Person(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String toString() {
        return this.name + "," + this.score;
    }
}

```

运行程序，我们会得到ClassCastException，即无法将Person转型为Comparable。我们修改代码，让Person实现Comparable<T>接口：

```

// sort
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Person[] ps = new Person[] {
            new Person("Bob", 61),
            new Person("Alice", 88),
            new Person("Lily", 75),
        };
        Arrays.sort(ps);
        System.out.println(Arrays.toString(ps));
    }
}

----
class Person implements Comparable<Person> {
    String name;
    int score;
    Person(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
    public String toString() {
        return this.name + "," + this.score;
    }
}

```

运行上述代码，可以正确实现按name进行排序。

也可以修改比较逻辑，例如，按score从高到低排序。请自行修改测试。

## 小结

使用泛型时，把泛型参数<T>替换为需要的class类型，例如：ArrayList<String>，ArrayList<Number>等；

可以省略编译器能自动推断出的类型，例如：`List<String> list = new ArrayList<>();`；

不指定泛型参数类型时，编译器会给出警告，且只能将<T>视为Object类型；

可以在接口中定义泛型类型，实现此接口的类必须实现正确的泛型类型。

编写泛型类比普通类要复杂。通常来说，泛型类一般用在集合类中，例如ArrayList<T>，我们很少需要编写泛型类。

## 编写泛型

如果我们确实需要编写一个泛型类，那么，应该如何编写它？

可以按照以下步骤来编写一个泛型类。

首先，按照某种类型，例如：String，来编写类：

```
public class Pair {
    private String first;
    private String last;
    public Pair(String first, String last) {
        this.first = first;
        this.last = last;
    }
    public String getFirst() {
        return first;
    }
    public String getLast() {
        return last;
    }
}
```

然后，标记所有的特定类型，这里是String：

```
public class Pair {
    private String first;
    private String last;
    public Pair(String first, String last) {
        this.first = first;
        this.last = last;
    }
    public String getFirst() {
        return first;
    }
    public String getLast() {
        return last;
    }
}
```

最后，把特定类型String替换为T，并申明<T>：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

熟练后即可直接从T开始编写。

### 静态方法

编写泛型类时，要特别注意，泛型类型<T>不能用于静态方法。例如：

```
public class Pair<T> {
```

```

private T first;
private T last;
public Pair(T first, T last) {
    this.first = first;
    this.last = last;
}
public T getFirst() { ... }
public T getLast() { ... }

// 对静态方法使用<T>:
public static Pair<T> create(T first, T last) {
    return new Pair<T>(first, last);
}
}

```

上述代码会导致编译错误，我们无法在静态方法`create()`的方法参数和返回类型上使用泛型类型`T`。

有些同学在网上搜索发现，可以在`static`修饰符后面加一个`<T>`，编译就能通过：

```

public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public T getLast() { ... }

    // 可以编译通过:
    public static <T> Pair<T> create(T first, T last) {
        return new Pair<T>(first, last);
    }
}

```

但实际上，这个`<T>`和`Pair<T>`类型的`<T>`已经没有任何关系了。

对于静态方法，我们可以单独改写为“泛型”方法，只需要使用另一个类型即可。对于上面的`create()`静态方法，我们应该把它改为另一种泛型类型，例如，`<K>`：

```

public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public T getLast() { ... }

    // 静态泛型方法应该使用其他类型区分:
    public static <K> Pair<K> create(K first, K last) {
        return new Pair<K>(first, last);
    }
}

```

这样才能清楚地将静态方法的泛型类型和实例类型的泛型类型区分开。

## 多个泛型类型

泛型还可以定义多种类型。例如，我们希望`Pair`不总是存储两个类型一样的对象，就可以使用类型`<T, K>`：

```

public class Pair<T, K> {
    private T first;
    private K last;
    public Pair(T first, K last) {
        this.first = first;
        this.last = last;
    }
}

```

```
    }  
    public T getFirst() { ... }  
    public K getLast() { ... }  
}
```

使用的时候，需要指出两种类型：

```
Pair<String, Integer> p = new Pair<>("test", 123);
```

Java标准库的Map<K, V>就是使用两种泛型类型的例子。它对**Key**使用一种类型，对**Value**使用另一种类型。

## 小结

编写泛型时，需要定义泛型类型<T>；

静态方法不能引用泛型类型<T>，必须定义其他类型（例如<K>）来实现静态泛型方法；

泛型可以同时定义多种类型，例如Map<K, V>。

泛型是一种类似“模板代码”的技术，不同语言的泛型实现方式不一定相同。

## 擦拭法

Java语言的泛型实现方式是擦拭法（Type Erasure）。

所谓擦拭法是指，虚拟机对泛型其实一无所知，所有的工作都是编译器做的。

例如，我们编写了一个泛型类Pair<T>，这是编译器看到的代码：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

而虚拟机根本不知道泛型。这是虚拟机执行的代码：

```
public class Pair {
    private Object first;
    private Object last;
    public Pair(Object first, Object last) {
        this.first = first;
        this.last = last;
    }
    public Object getFirst() {
        return first;
    }
    public Object getLast() {
        return last;
    }
}
```

因此，Java使用擦拭法实现泛型，导致了：

- 编译器把类型<T>视为Object；
- 编译器根据<T>实现安全的强制转型。

使用泛型的时候，我们编写的代码也是编译器看到的代码：

```
Pair<String> p = new Pair<>("Hello", "world");
String first = p.getFirst();
String last = p.getLast();
```

而虚拟机执行的代码并没有泛型：

```
Pair p = new Pair("Hello", "world");
String first = (String) p.getFirst();
String last = (String) p.getLast();
```

所以，Java的泛型是由编译器在编译时实行的，编译器内部永远把所有类型T视为Object处理，但是，在需要转型的时候，编译器会根据T的类型自动为我们实行安全地强制转型。

了解了Java泛型的实现方式——擦拭法，我们就知道了Java泛型的局限：

局限一：<T>不能是基本类型，例如int，因为实际类型是Object，Object类型无法持有基本类型：

```
Pair<int> p = new Pair<>(1, 2); // compile error!
```

局限二：无法取得带泛型的Class。观察以下代码：

```
public class Main {
    public static void main(String[] args) {
        ----
        Pair<String> p1 = new Pair<>("Hello", "world");
        Pair<Integer> p2 = new Pair<>(123, 456);
        Class c1 = p1.getClass();
        Class c2 = p2.getClass();
        System.out.println(c1==c2); // true
        System.out.println(c1==Pair.class); // true
        ----
    }
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

因为T是Object，我们对Pair<String>和Pair<Integer>类型获取Class时，获取到的是同一个Class，也就是Pair类的Class。

换句话说，所有泛型实例，无论T的类型是什么，getClass()返回同一个Class实例，因为编译后它们全部都是Pair<Object>。

局限三：无法判断带泛型的Class：

```
Pair<Integer> p = new Pair<>(123, 456);
// Compile error:
if (p instanceof Pair<String>.class) {
}
```

原因和前面一样，并不存在Pair<String>.class，而是只有唯一的Pair.class。

局限四：不能实例化T类型：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair() {
        // Compile error:
        first = new T();
        last = new T();
    }
}
```

上述代码无法通过编译，因为构造方法的两行语句：

```
first = new T();
last = new T();
```

擦拭后实际上变成了：

```
first = new Object();
last = new Object();
```

这样一来，创建`new Pair<String>()`和创建`new Pair<Integer>()`就全部成了`Object`，显然编译器要阻止这种类型不对的代码。

要实例化`T`类型，我们必须借助额外的`Class<T>`参数：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(Class<T> clazz) {
        first = clazz.newInstance();
        last = clazz.newInstance();
    }
}
```

上述代码借助`Class<T>`参数并通过反射来实例化`T`类型，使用的时候，也必须传入`Class<T>`。例如：

```
Pair<String> pair = new Pair<>(String.class);
```

因为传入了`Class<String>`的实例，所以我们借助`String.class`就可以实例化`String`类型。

## 不恰当的覆写方法

有些时候，一个看似正确定义的方法会无法通过编译。例如：

```
public class Pair<T> {
    public boolean equals(T t) {
        return this == t;
    }
}
```

这是因为，定义的`equals(T t)`方法实际上会被擦拭成`equals(Object t)`，而这个方法是继承自`Object`的，编译器会阻止一个实际上会变成覆写的泛型方法定义。

换个方法名，避开与`Object.equals(Object)`的冲突就可以成功编译：

```
public class Pair<T> {
    public boolean same(T t) {
        return this == t;
    }
}
```

## 泛型继承

一个类可以继承自一个泛型类。例如：父类的类型是`Pair<Integer>`，子类的类型是`IntPair`，可以这么继承：

```
public class IntPair extends Pair<Integer> {
}
```

使用的时候，因为子类`IntPair`并没有泛型类型，所以，正常使用即可：

```
IntPair ip = new IntPair(1, 2);
```

前面讲了，我们无法获取`Pair<T>`的`T`类型，即给定一个变量`Pair<Integer> p`，无法从`p`中获取到`Integer`类型。

但是，在父类是泛型类型的情况下，编译器就必须把类型`T`（对`IntPair`来说，也就是`Integer`类型）保存到子类的`class`文件中，不然编译器就不知道`IntPair`只能存取`Integer`这种类型。

在继承了泛型类型的情况下，子类可以获取父类的泛型类型。例如：`IntPair`可以获取到父类的泛型类型`Integer`。获取父类的泛型类型代码比较复杂：

```
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;

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



```

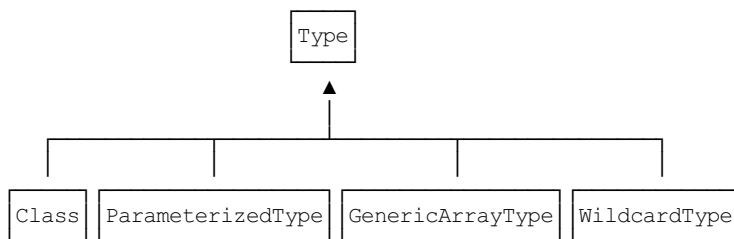
----
    Class<IntPair> clazz = IntPair.class;
    Type t = clazz.getGenericSuperclass();
    if (t instanceof ParameterizedType) {
        ParameterizedType pt = (ParameterizedType) t;
        Type[] types = pt.getActualTypeArguments(); // 可能有多个泛型类型
        Type firstType = types[0]; // 取第一个泛型类型
        Class<?> typeClass = (Class<?>) firstType;
        System.out.println(typeClass); // Integer
    }
----
}
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}

class IntPair extends Pair<Integer> {
    public IntPair(Integer first, Integer last) {
        super(first, last);
    }
}

```

因为Java引入了泛型，所以，只用Class来标识类型已经不够了。实际上，Java的类型系统结构如下：



## 小结

Java的泛型是采用擦拭法实现的；

擦拭法决定了泛型<T>：

- 不能是基本类型，例如：int；
- 不能获取带泛型类型的Class，例如：Pair<String>.class；
- 不能判断带泛型类型的类型，例如：x instanceof Pair<String>；
- 不能实例化T类型，例如：new T()。

泛型方法要防止重复定义方法，例如：public boolean equals(T obj)；

子类可以获取父类的泛型类型<T>。

我们前面已经讲到了泛型的继承关系：`Pair<Integer>`不是`Pair<Number>`的子类。

## extends通配符

假设我们定义了`Pair<T>`:

```
public class Pair<T> { ... }
```

然后，我们又针对`Pair<Number>`类型写了一个静态方法，它接收的参数类型是`Pair<Number>`:

```
public class PairHelper {
    static int add(Pair<Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
}
```

上述代码是可以正常编译的。使用的时候，我们传入：

```
int sum = PairHelper.add(new Pair<Number>(1, 2));
```

注意：传入的类型是`Pair<Number>`，实际参数类型是`(Integer, Integer)`。

既然实际参数是`Integer`类型，试试传入`Pair<Integer>`:

```
public class Main {
    ----
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }

    static int add(Pair<Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
    ----
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

直接运行，会得到一个编译错误：

```
incompatible types: Pair<Integer> cannot be converted to Pair<Number>
```

原因很明显，因为`Pair<Integer>`不是`Pair<Number>`的子类，因此，`add(Pair<Number>)`不接受参数类型`Pair<Integer>`。

但是从`add()`方法的代码可知，传入`Pair<Integer>`是完全符合内部代码的类型规范，因为语句：

```
Number first = p.getFirst();
Number last = p.getLast();
```

实际类型是Integer，引用类型是Number，没有问题。问题在于方法参数类型定死了只能传入Pair<Number>。

有没有办法使得方法参数接受Pair<Integer>？办法是有的，这就是使用Pair<? extends Number>使得方法接收所有泛型类型为Number或Number子类的Pair类型。我们把代码改写如下：

```
public class Main {
    ----
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }

    static int add(Pair<? extends Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
    ----
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

这样一来，给方法传入Pair<Integer>类型时，它符合参数Pair<? extends Number>类型。这种使用<? extends Number>的泛型定义称之为上界通配符（Upper Bounds Wildcards），即把泛型类型T的上界限定在Number了。

除了可以传入Pair<Integer>类型，我们还可以传入Pair<Double>类型，Pair<BigDecimal>类型等等，因为Double和BigDecimal都是Number的子类。

如果我们考察对Pair<? extends Number>类型调用getFirst()方法，实际的方法签名变成了：

```
<? extends Number> getFirst();
```

即返回值是Number或Number的子类，因此，可以安全赋值给Number类型的变量：

```
Number x = p.getFirst();
```

然后，我们不可预测实际类型就是Integer，例如，下面的代码是无法通过编译的：

```
Integer x = p.getFirst();
```

这是因为实际的返回类型可能是Integer，也可能是Double或者其他类型，编译器只能确定类型一定是Number的子类（包括Number类型本身），但具体类型无法确定。

我们再来考察一下Pair<T>的set方法：

```
public class Main {
    ----
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }
}
```

```

    }

    static int add(Pair<? extends Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        p.setFirst(new Integer(first.intValue() + 100));
        p.setLast(new Integer(last.intValue() + 100));
        return p.getFirst().intValue() + p.getFirst().intValue();
    }
}
----
}

class Pair<T> {
    private T first;
    private T last;

    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }

    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setLast(T last) {
        this.last = last;
    }
}
}

```

不出意外，我们会得到一个编译错误：

```

incompatible types: Integer cannot be converted to CAP#1
where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number

```

编译错误发生在`p.setFirst()`传入的参数是`Integer`类型。有些童鞋会问了，既然`p`的定义是`Pair<? extends Number>`，那么`setFirst(? extends Number)`为什么不能传入`Integer`？

原因还在于擦除法。如果我们传入的`p`是`Pair<Double>`，显然它满足参数定义`Pair<? extends Number>`，然而，`Pair<Double>`的`setFirst()`显然无法接受`Integer`类型。

这就是`<? extends Number>`通配符的一个重要限制：方法参数签名`setFirst(? extends Number)`无法传递任何`Number`类型给`setFirst(? extends Number)`。

这里唯一的例外是可以给方法参数传入`null`：

```

p.setFirst(null); // ok, 但是后面会抛出NullPointerException
p.getFirst().intValue(); // NullPointerException

```

## extends通配符的作用

如果我们考察Java标准库的`java.util.List<T>`接口，它实现的是一个类似“可变数组”的列表，主要功能包括：

```

public interface List<T> {
    int size(); // 获取个数
    T get(int index); // 根据索引获取指定元素
    void add(T t); // 添加一个新元素
    void remove(T t); // 删除一个已有元素
}

```

现在，让我们定义一个方法来处理列表的每个元素：

```
int sumOfList(List<? extends Integer> list) {
    int sum = 0;
    for (int i=0; i<list.size(); i++) {
        Integer n = list.get(i);
        sum = sum + n;
    }
    return sum;
}
```

为什么我们定义的方法参数类型是`List<? extends Integer>`而不是`List<Integer>`? 从方法内部代码看, 传入`List<? extends Integer>`或者`List<Integer>`是完全一样的, 但是, 注意到`List<? extends Integer>`的限制:

- 允许调用`get()`方法获取`Integer`的引用;
- 不允许调用`set(? extends Integer)`方法并传入任何`Integer`的引用 (`null`除外)。

因此, 方法参数类型`List<? extends Integer>`表明了该方法内部只会读取`List`的元素, 不会修改`List`的元素 (因为无法调用`add(? extends Integer)`、`remove(? extends Integer)`这些方法。换句话说, 这是一个对参数`List<? extends Integer>`进行只读的方法 (恶意调用`set(null)`除外)。

## 使用`extends`限定T类型

在定义泛型类型`Pair<T>`的时候, 也可以使用`extends`通配符来限定`T`的类型:

```
public class Pair<T extends Number> { ... }
```

现在, 我们只能定义:

```
Pair<Number> p1 = null;
Pair<Integer> p2 = new Pair<>(1, 2);
Pair<Double> p3 = null;
```

因为`Number`、`Integer`和`Double`都符合`<T extends Number>`。

非`Number`类型将无法通过编译:

```
Pair<String> p1 = null; // compile error!
Pair<Object> p2 = null; // compile error!
```

因为`String`、`Object`都不符合`<T extends Number>`, 因为它们不是`Number`类型或`Number`的子类。

## 小结

使用类似`<? extends Number>`通配符作为方法参数时表示:

- 方法内部可以调用获取`Number`引用的方法, 例如: `Number n = obj.getFirst();`
- 方法内部无法调用传入`Number`引用的方法 (`null`除外), 例如: `obj.setFirst(Number n);`

即一句话总结: 使用`extends`通配符表示可以读, 不能写。

使用类似`<T extends Number>`定义泛型类时表示:

- 泛型类型限定为`Number`以及`Number`的子类。

我们前面已经讲到了泛型的继承关系：`Pair<Integer>`不是`Pair<Number>`的子类。

## super通配符

考察下面的set方法：

```
void set(Pair<Integer> p, Integer first, Integer last) {
    p.setFirst(first);
    p.setLast(last);
}
```

传入`Pair<Integer>`是允许的，但是传入`Pair<Number>`是不允许的。

和`extends`通配符相反，这次，我们希望接受`Pair<Integer>`类型，以及`Pair<Number>`、`Pair<Object>`，因为`Number`和`Object`是`Integer`的父类，`setFirst(Number)`和`setFirst(Object)`实际上允许接受`Integer`类型。

我们使用`super`通配符来改写这个方法：

```
void set(Pair<? super Integer> p, Integer first, Integer last) {
    p.setFirst(first);
    p.setLast(last);
}
```

注意到`Pair<? super Integer>`表示，方法参数接受所有泛型类型为`Integer`或`Integer`父类的`Pair`类型。

下面的代码可以被正常编译：

```
public class Main {
    ----
    public static void main(String[] args) {
        Pair<Number> p1 = new Pair<>(12.3, 4.56);
        Pair<Integer> p2 = new Pair<>(123, 456);
        setSame(p1, 100);
        setSame(p2, 200);
        System.out.println(p1.getFirst() + ", " + p1.getLast());
        System.out.println(p2.getFirst() + ", " + p2.getLast());
    }

    static void setSame(Pair<? super Integer> p, Integer n) {
        p.setFirst(n);
        p.setLast(n);
    }
    ----
}

class Pair<T> {
    private T first;
    private T last;

    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }

    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setLast(T last) {
        this.last = last;
    }
}
```

考察Pair<? super Integer>的setFirst()方法，它的方法签名实际上是：

```
void setFirst(? super Integer);
```

因此，可以安全地传入Integer类型。

再考察Pair<? super Integer>的getFirst()方法，它的方法签名实际上是：

```
? super Integer getFirst();
```

这里注意到我们无法使用Integer类型来接收getFirst()的返回值，即下面的语句将无法通过编译：

```
Integer x = p.getFirst();
```

因为如果传入的实际类型是Pair<Number>，编译器无法将Number类型转型为Integer。

注意：虽然Number是一个抽象类，我们无法直接实例化它。但是，即便Number不是抽象类，这里仍然无法通过编译。此外，传入Pair<Object>类型时，编译器也无法将Object类型转型为Integer。

唯一可以接收getFirst()方法返回值的是Object类型：

```
Object obj = p.getFirst();
```

因此，使用<? super Integer>通配符表示：

- 允许调用set(? super Integer)方法传入Integer的引用；
- 不允许调用get()方法获得Integer的引用。

唯一例外是可以获取Object的引用：Object o = p.getFirst()。

换句话说，使用<? super Integer>通配符作为方法参数，表示方法内部代码对于参数只能写，不能读。

## 使用super限定T类型

在定义泛型类型Pair<T>的时候，也可以使用super通配符来限定T的类型：

```
public class Pair<T super Integer> { ... }
```

现在，我们只能定义：

```
Pair<Number> p1 = null;  
Pair<Integer> p2 = new Pair<>(1, 2);  
Pair<Object> p3 = null;
```

因为Number、Integer和Object都符合<T super Number>。

非Integer类型以及它的子类都将无法通过编译：

```
Pair<String> p1 = null; // compile error!
```

## 对比extends和super通配符

我们再回顾一下extends通配符。作为方法参数，<? extends T>类型和<? super T>类型的区别在于：

- <? extends T>允许调用读方法T get()获取T的引用，但不允许调用写方法set(T)传入T的引用（传入null除外）；
- <? super T>允许调用写方法set(T)传入T的引用，但不允许调用读方法T get()获取T的引用（获取Object除外）。

一个是允许读不允许写，另一个是允许写不允许读。

先记住上面的结论，我们来看Java标准库的Collections类定义的copy()方法：

```
public class Collections {
    // 把src的每个元素复制到dest中：
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        for (int i=0; i<src.size(); i++) {
            T t = src.get(i);
            dest.add(t);
        }
    }
}
```

它的作用是把一个List的每个元素依次添加到另一个List中。它的第一个参数是List<? super T>，表示目标List，第二个参数List<? extends T>，表示要复制的List。我们可以简单地用for循环实现复制。在for循环中，我们可以看到，对于类型<? extends T>的变量src，我们可以安全地获取类型T的引用，而对于类型<? super T>的变量dest，我们可以安全地传入T的引用。

这个copy()方法的定义就完美地展示了extends和super的意图：

- copy()方法内部不会读取dest，因为不能调用dest.get()来获取T的引用；
- copy()方法内部也不会修改src，因为不能调用src.add(T)。

这是由编译器检查来实现的。如果在方法代码中意外修改了src，或者意外读取了dest，就会导致一个编译错误：

```
public class Collections {
    // 把src的每个元素复制到dest中：
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        ...
        T t = dest.get(0); // compile error!
        src.add(t); // compile error!
    }
}
```

这个copy()方法的另一个好处是可以安全地把一个List<Integer>添加到List<Number>，但是无法反过来添加：

```
// copy List<Integer> to List<Number> ok:
List<Number> numList = ...;
List<Integer> intList = ...;
Collections.copy(numList, intList);

// ERROR: cannot copy List<Number> to List<Integer>:
Collections.copy(intList, numList);
```

而这些都是通过super和extends通配符，并由编译器强制检查来实现的。

## PECS原则

何时使用extends，何时使用super？为了便于记忆，我们可以用PECS原则：Producer Extends Consumer Super。

即：如果需要返回T，它是生产者（Producer），要使用extends通配符；如果需要写入T，它是消费者（Consumer），要使用super通配符。

还是以Collections的copy()方法为例：

```
public class Collections {
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        for (int i=0; i<src.size(); i++) {
            T t = src.get(i); // src是producer
            dest.add(t); // dest是consumer
        }
    }
}
```

需要返回T的src是生产者，因此声明为List<? extends T>，需要写入T的dest是消费者，因此声明为List<? super T>。



## 无限定通配符

我们已经讨论了`<? extends T>`和`<? super T>`作为方法参数的作用。实际上，Java的泛型还允许使用无限定通配符（Unbounded Wildcard Type），即只定义一个`?`：

```
void sample(Pair<?> p) {  
}
```

因为`<?>`通配符既没有`extends`，也没有`super`，因此：

- 不允许调用`set(T)`方法并传入引用（`null`除外）；
- 不允许调用`T get()`方法并获取`T`引用（只能获取`Object`引用）。

换句话说，既不能读，也不能写，那只能做一些`null`判断：

```
static boolean isNull(Pair<?> p) {  
    return p.getFirst() == null || p.getLast() == null;  
}
```

大多数情况下，可以引入泛型参数`<T>`消除`<?>`通配符：

```
static <T> boolean isNull(Pair<T> p) {  
    return p.getFirst() == null || p.getLast() == null;  
}
```

`<?>`通配符有一个独特的特点，就是：`Pair<?>`是所有`Pair<T>`的超类：

```
public class Main {  
    ----  
    public static void main(String[] args) {  
        Pair<Integer> p = new Pair<>(123, 456);  
        Pair<?> p2 = p; // 安全地向上转型  
        System.out.println(p2.getFirst() + ", " + p2.getLast());  
    }  
    ----  
}  
  
class Pair<T> {  
    private T first;  
    private T last;  
  
    public Pair(T first, T last) {  
        this.first = first;  
        this.last = last;  
    }  
  
    public T getFirst() {  
        return first;  
    }  
    public T getLast() {  
        return last;  
    }  
    public void setFirst(T first) {  
        this.first = first;  
    }  
    public void setLast(T last) {  
        this.last = last;  
    }  
}
```

上述代码是可以正常编译运行的，因为`Pair<Integer>`是`Pair<?>`的子类，可以安全地向上转型。

## 小结

使用类似`<? super Integer>`通配符作为方法参数时表示：

- 方法内部可以调用传入`Integer`引用的方法，例如：`obj.setFirst(Integer n);`；

- 方法内部无法调用获取Integer引用的方法（Object除外），例如：`Integer n = obj.getFirst();`。

即使用`super`通配符表示只能写不能读。

使用`extends`和`super`通配符要遵循PECS原则。

使用类似`<T super Integer>`定义泛型类时表示：

- 泛型类型限定为Integer或Integer的超类。

无限定通配符`<?>`很少使用，可以用`<T>`替换，同时它是所有`<T>`类型的超类。

Java的部分反射API也是泛型。例如：Class<T>就是泛型：

## 泛型和反射

```
// compile warning:
Class clazz = String.class;
String str = (String) clazz.newInstance();
```

```
// no warning:
Class<String> clazz = String.class;
String str = clazz.newInstance();
```

调用Class的getSuperclass()方法返回的Class类型是Class<? super T>：

```
Class<? super String> sup = String.class.getSuperclass();
```

构造方法Constructor<T>也是泛型：

```
Class<Integer> clazz = Integer.class;
Constructor<Integer> cons = clazz.getConstructor(int.class);
Integer i = cons.newInstance(123);
```

我们可以声明带泛型的数组，但不能用new操作符创建带泛型的数组：

```
Pair<String>[] ps = null; // ok
Pair<String>[] ps = new Pair<String>[2]; // compile error!
```

必须通过强制转型实现带泛型的数组：

```
@SuppressWarnings("unchecked")
Pair<String>[] ps = (Pair<String>[]) new Pair[2];
```

使用泛型数组要特别小心，因为数组实际上在运行期没有泛型，编译器可以强制检查变量ps，因为它的类型是泛型数组。但是，编译器不会检查变量arr，因为它不是泛型数组。因为这两个变量实际上指向同一个数组，所以，操作arr可能导致从ps获取元素时报错，例如，以下代码演示了不安全地使用带泛型的数组：

```
Pair[] arr = new Pair[2];
Pair<String>[] ps = (Pair<String>[]) arr;

ps[0] = new Pair<String>("a", "b");
arr[1] = new Pair<Integer>(1, 2);

// ClassCastException:
Pair<String> p = ps[1];
String s = p.getFirst();
```

要安全地使用泛型数组，必须扔掉arr的引用：

```
@SuppressWarnings("unchecked")
Pair<String>[] ps = (Pair<String>[]) new Pair[2];
```

上面的代码中，由于拿不到原始数组的引用，就只能对泛型数组ps进行操作，这种操作就是安全的。

带泛型的数组实际上是编译器的类型擦除：

```
Pair[] arr = new Pair[2];
Pair<String>[] ps = (Pair<String>[]) arr;

System.out.println(ps.getClass() == Pair[].class); // true

String s1 = (String) arr[0].getFirst();
String s2 = ps[0].getFirst();
```

所以我们不能直接创建泛型数组T[]，因为擦拭后代码变为Object[]：

```
// compile error:
```

```
public class Abc<T> {
    T[] createArray() {
        return new T[5];
    }
}
```

必须借助Class<T>来创建泛型数组：

```
T[] createArray(Class<T> cls) {
    return (T[]) Array.newInstance(cls, 5);
}
```

我们还可以利用可变参数创建泛型数组T[]：

```
public class ArrayHelper {
    @SafeVarargs
    static <T> T[] asArray(T... objs) {
        return objs;
    }
}
```

```
String[] ss = ArrayHelper.asArray("a", "b", "c");
Integer[] ns = ArrayHelper.asArray(1, 2, 3);
```

## 谨慎使用泛型可变参数

在上面的例子中，我们看到，通过：

```
static <T> T[] asArray(T... objs) {
    return objs;
}
```

似乎可以安全地创建一个泛型数组。但实际上，这种方法非常危险。以下代码来自《Effective Java》的示例：

```
import java.util.Arrays;

public class Main {
    ----
    public static void main(String[] args) {
        String[] arr = asArray("one", "two", "three");
        System.out.println(Arrays.toString(arr));
        // ClassCastException:
        String[] firstTwo = pickTwo("one", "two", "three");
        System.out.println(Arrays.toString(firstTwo));
    }

    static <K> K[] pickTwo(K k1, K k2, K k3) {
        return asArray(k1, k2);
    }

    static <T> T[] asArray(T... objs) {
        return objs;
    }
    ----
}
```

直接调用asArray(T...)似乎没有问题，但是在另一个方法中，我们返回一个泛型数组就会产生ClassCastException，原因还是因为擦除法，在pickTwo()方法内部，编译器无法检测K[]的正确类型，因此返回了Object[]。

如果仔细观察，可以发现编译器对所有可变泛型参数都会发出警告，除非确认完全没有问题，才可以用@SafeVarargs消除警告。

如果在方法内部创建了泛型数组，最好不要将它返回给外部使用。

更详细的解释请参考《[Effective Java](#)》“Item 32: Combine generics and varargs judiciously”。

## 小结

部分反射API是泛型，例如：`Class<T>`，`Constructor<T>`；

可以声明带泛型的数组，但不能直接创建带泛型的数组，必须强制转型；

可以通过`Array.newInstance(Class<T>, int)`创建`T[]`数组，需要强制转型；

同时使用泛型和可变参数时需要特别小心。

本节我们将介绍Java的集合类型。集合类型也是Java标准库中被使用最多的类型。

## 集合



什么是集合（Collection）？集合就是“由若干个确定的元素所构成的整体”。例如，5只小兔构成的集合：

## Java集合简介

```
┌-----┐
│  (\_(\_    (\_/_    (\_/_    (\_/_    (\_(\_    │
│  (  _.-)    (•.•)    (>.<)    (^.^)    (='.')    │
│  c("_)_(")  (")_(")  (")_(")  (")_(")  o("_)" )    │
└-----┘
```

在数学中，我们经常遇到集合的概念。例如：

- 有限集合：
  - 一个班所有的同学构成的集合；
  - 一个网站所有的商品构成的集合；
  - ...
- 无限集合：
  - 全体自然数集合：1，2，3，.....
  - 有理数集合；
  - 实数集合；
  - ...

为什么要在计算机中引入集合呢？这是为了便于处理一组类似的数据，例如：

- 计算所有同学的总成绩和平均成绩；
- 列举所有的商品名称和价格；
- .....

在Java中，如果一个Java对象可以在内部持有若干其他Java对象，并对外提供访问接口，我们把这种Java对象称为集合。很显然，Java的数组可以看作是一种集合：

```
String[] ss = new String[10]; // 可以持有10个String对象
ss[0] = "Hello"; // 可以放入String对象
String first = ss[0]; // 可以获取String对象
```

既然Java提供了数组这种数据类型，可以充当集合，那么，我们为什么还需要其他集合类？这是因为数组有如下限制：

- 数组初始化后大小不可变；
- 数组只能按索引顺序存取。

因此，我们需要各种不同类型的集合类来处理不同的数据，例如：

- 可变大小的顺序链表；
- 保证无重复元素的集合；
- ...

### Collection

Java标准库自带的java.util包提供了集合类：Collection，它是所有其他集合类的根接口。在Collection的基础上，Java的java.util包主要提供了以下三种类型的集合：

- List：一种有序列表的集合，例如，按索引排列的Student的List；
- Set：一种保证没有重复元素的集合，例如，所有无重复名称的Student的Set；
- Map：一种通过键值（key-value）查找的映射表集合，例如，根据Student的name查找对应Student的Map。

**Java**集合的设计有几个特点：一是实现了接口和实现类相分离，例如，有序表的接口是`List`，具体的实现类有`ArrayList`，`LinkedList`等，二是支持泛型，我们可以限制在一个集合中只能放入同一种数据类型的元素，例如：

```
List<String> list = new ArrayList<>(); // 只能放入String类型
```

最后，**Java**访问集合总是通过统一的方式——迭代器（`Iterator`）来实现，它最明显的好处在于无需知道集合内部元素是按什么方式存储的。

由于**Java**的集合设计非常久远，中间经历过大规模改进，我们要注意到有一小部分集合类是遗留类，不应该继续使用：

- `Hashtable`：一种线程安全的`Map`实现；
- `Vector`：一种线程安全的`List`实现；
- `Stack`：基于`Vector`实现的LIFO的栈。

还有一小部分接口是遗留接口，也不应该继续使用：

- `Enumeration<E>`：已被`Iterator<E>`取代。

## 小结

**Java**的集合类定义在`java.util`包中，支持泛型，主要提供了3种集合类，包括`List`，`Set`和`Map`。**Java**集合使用统一的`Iterator`遍历，尽量不要使用遗留接口。

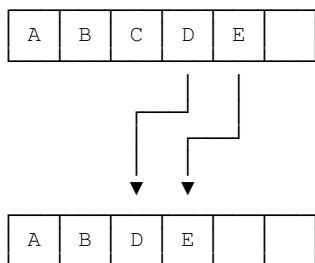


在集合类中，List是最基础的一种集合：它是一种有序链表。

## 使用List

List的行为和数组几乎完全相同：List内部按照放入元素的先后顺序存放，每个元素都可以通过索引确定自己的位置，List的索引和数组一样，从0开始。

数组和List类似，也是有序结构，如果我们使用数组，在添加和删除元素的时候，会非常不方便。例如，从一个已有的数组{'A', 'B', 'C', 'D', 'E'}中删除索引为2的元素：



这个“删除”操作实际上是把‘C’后面的元素依次往前挪一个位置，而“添加”操作实际上是把指定位置以后的元素都依次向后挪一个位置，腾出来的位置给新加的元素。这两种操作，用数组实现非常麻烦。

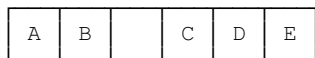
因此，在实际应用中，需要增删元素的有序列表，我们使用最多的是ArrayList。实际上，ArrayList在内部使用了数组来存储所有元素。例如，一个ArrayList拥有5个元素，实际数组大小为6（即有一个空位）：

size=5



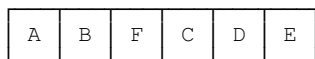
当添加一个元素并指定索引到ArrayList时，ArrayList自动移动需要移动的元素：

size=5



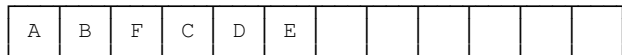
然后，往内部指定索引的数组位置添加一个元素，然后把size加1：

size=6



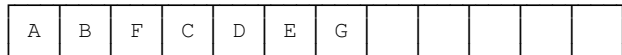
继续添加元素，但是数组已满，没有空闲位置的时候，ArrayList先创建一个更大的新数组，然后把旧数组的所有元素复制到新数组，紧接着用新数组取代旧数组：

size=6



现在，新数组就有了空位，可以继续添加一个元素到数组末尾，同时size加1：

size=7

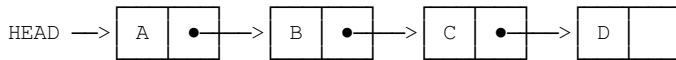


可见，ArrayList把添加和删除的操作封装起来，让我们操作List类似于操作数组，却不用关心内部元素如何移动。

我们考察List<E>接口，可以看到几个主要的接口方法：

- 在末尾添加一个元素：void add(E e)
- 在指定索引添加一个元素：void add(int index, E e)
- 删除指定索引的元素：int remove(int index)
- 删除某个元素：int remove(Object e)
- 获取指定索引的元素：E get(int index)
- 获取链表大小（包含元素的个数）：int size()

但是，实现List接口并非只能通过数组（即ArrayList的实现方式）来实现，另一种LinkedList通过“链表”也实现了List接口。在LinkedList中，它的内部每个元素都指向下一个元素：



我们来比较一下ArrayList和LinkedList：

	<b>ArrayList</b>	<b>LinkedList</b>
获取指定元素	速度很快	需要从头开始查找元素
添加元素到末尾	速度很快	速度很快
在指定位置添加/删除	需要移动元素	不需要移动元素
内存占用	少	较大

通常情况下，我们总是优先使用ArrayList。

## List的特点

使用List时，我们要关注List接口的规范。List接口允许我们添加重复的元素，即List内部的元素可以重复：

```
import java.util.ArrayList;
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple"); // size=1
        list.add("pear"); // size=2
        list.add("apple"); // 允许重复添加元素，size=3
        System.out.println(list.size());
    }
}
```

List还允许添加null：

```
import java.util.ArrayList;
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple"); // size=1
        list.add(null); // size=2
        list.add("pear"); // size=3
        String second = list.get(1); // null
        System.out.println(second);
    }
}
```

## 创建List

除了使用ArrayList和LinkedList，我们还可以通过List接口提供的of()方法，根据给定元素快速创建List：

```
List<Integer> list = List.of(1, 2, 5);
```

但是List.of()方法不接受null值，如果传入null，会抛出NullPointerException异常。

## 遍历List

和数组类型，我们要遍历一个List，完全可以用for循环根据索引配合get(int)方法遍历：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear", "banana");
        for (int i=0; i<list.size(); i++) {
            String s = list.get(i);
            System.out.println(s);
        }
    }
}
```

但这种方式并不推荐，一是代码复杂，二是因为get(int)方法只有ArrayList的实现是高效的，换成LinkedList后，索引越大，访问速度越慢。

所以我们要始终坚持使用迭代器Iterator来访问List。Iterator本身也是一个对象，但它是由List的实例调用iterator()方法的时候创建的。Iterator对象知道如何遍历一个List，并且不同的List类型，返回的Iterator对象实现也是不同的，但总是具有最高的访问效率。

Iterator对象有两个方法：boolean hasNext()判断是否有下一个元素，E next()返回下一个元素。因此，使用Iterator遍历List代码如下：

```
import java.util.Iterator;
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear", "banana");
        for (Iterator<String> it = list.iterator(); it.hasNext(); ) {
            String s = it.next();
            System.out.println(s);
        }
    }
}
```

有童鞋可能觉得使用Iterator访问List的代码比使用索引更复杂。但是，要记住，通过Iterator遍历List永远是最高效的方式。并且，由于Iterator遍历是如此常用，所以，Java的for each循环本身就可以帮我们使用Iterator遍历。把上面的代码再改写如下：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear", "banana");
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

上述代码就是我们编写遍历List的常见代码。

实际上，只要实现了Iterator接口的集合类都可以直接用for each循环来遍历，Java编译器本身并不知道如何遍历集合对象，但它会自动把for each循环变成Iterator的调用。

## List和Array转换

把List变为Array有三种方法，第一种是调用toArray()方法直接返回一个Object[]数组：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear", "banana");
        Object[] array = list.toArray();
        for (Object s : array) {
            System.out.println(s);
        }
    }
}
```

这种方法会丢失类型信息，所以实际应用很少。

第二种方式是给`toArray(T[])`传入一个类型相同的Array，List内部自动把元素复制到传入的Array中：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        Integer[] array = list.toArray(new Integer[3]);
        for (Integer n : array) {
            System.out.println(n);
        }
    }
}
```

注意到这个`toArray(T[])`方法的泛型参数`<T>`并不是List接口定义的泛型参数`<E>`，所以，我们实际上可以传入其他类型的数组，例如我们传入Number类型的数组，返回的仍然是Number类型：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        Number[] array = list.toArray(new Number[3]);
        for (Number n : array) {
            System.out.println(n);
        }
    }
}
```

但是，如果我们传入类型不匹配的数组，例如，`String[]`类型的数组，由于List的元素是Integer，所以无法放入String数组，这个方法会抛出`ArrayStoreException`。

如果我们传入的数组大小和List实际的元素个数不一致怎么办？根据[List接口](#)的文档，我们可以知道：

如果传入的数组不够大，那么List内部会创建一个新的刚好够大的数组，填充后返回；如果传入的数组比List元素还要多，那么填充完元素后，剩下的数组元素一律填充null。

实际上，最常用的是传入一个“恰好”大小的数组：

```
Integer[] array = list.toArray(new Integer[list.size()]);
```

最后一种更简洁的写法是通过List接口定义的`T[] toArray(IntFunction<T[]> generator)`方法：

```
Integer[] array = list.toArray(Integer[]::new);
```

这种函数式写法我们会在后续讲到。

反过来，把Array变为List就简单多了，通过`List.of(T...)`方法最简单：

```
Integer[] array = { 1, 2, 3 };
List<Integer> list = List.of(array);
```

对于JDK 11之前的版本，可以使用`Arrays.asList(T...)`方法把数组转换成List。

要注意的是，返回的List不一定是ArrayList或者LinkedList，因为List只是一个接口，如果我们调用List.of()，它返回的是一个只读List：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        list.add(999); // UnsupportedOperationException
    }
}
```

对只读List调用add()、remove()方法会抛出UnsupportedOperationException。

## 练习

给定一组连续的整数，例如：10, 11, 12, ....., 20，但其中缺失一个数字，试找出缺失的数字：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 构造从start到end的序列：
        final int start = 10;
        final int end = 20;
        List<Integer> list = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            list.add(i);
        }
        // 随机删除List中的一个元素：
        int removed = list.remove((int) (Math.random() * list.size()));
        int found = findMissingNumber(start, end, list);
        System.out.println(list.toString());
        System.out.println("missing number: " + found);
        System.out.println(removed == found ? "测试成功" : "测试失败");
    }
}
----
static int findMissingNumber(int start, int end, List<Integer> list) {
    return 0;
}
----
}
```

增强版：和上述题目一样，但整数不再有序，试找出缺失的数字：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 构造从start到end的序列：
        final int start = 10;
        final int end = 20;
        List<Integer> list = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            list.add(i);
        }
        // 洗牌算法shuffle可以随机交换List中的元素位置：
        Collections.shuffle(list);
        // 随机删除List中的一个元素：
        int removed = list.remove((int) (Math.random() * list.size()));
        int found = findMissingNumber(start, end, list);
        System.out.println(list.toString());
        System.out.println("missing number: " + found);
        System.out.println(removed == found ? "测试成功" : "测试失败");
    }
}
----
static int findMissingNumber(int start, int end, List<Integer> list) {
    return 0;
}
}
```

```
-----  
}
```

[找出缺失的数字](#)

## 小结

List是按索引顺序访问的长度可变的有序表，优先使用ArrayList而不是LinkedList；

可以直接使用for each遍历List；

List可以和Array相互转换。

我们知道List是一种有序链表：List内部按照放入元素的先后顺序存放，并且每个元素都可以通过索引确定自己的位置。

## 编写equals方法

List还提供了boolean contains(Object o)方法来判断List是否包含某个指定元素。此外，int indexOf(Object o)方法可以返回某个元素的索引，如果元素不存在，就返回-1。

我们来看一个例子：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("A", "B", "C");
        System.out.println(list.contains("C")); // true
        System.out.println(list.contains("X")); // false
        System.out.println(list.indexOf("C")); // 2
        System.out.println(list.indexOf("X")); // -1
    }
}
```

这里我们注意一个问题，我们往List中添加的"C"和调用contains("C")传入的"C"是不是同一个实例？

如果这两个"C"不是同一个实例，这段代码是否还能得到正确的结果？我们可以改写一下代码测试一下：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("A", "B", "C");
        System.out.println(list.contains(new String("C"))); // true or false?
        System.out.println(list.indexOf(new String("C"))); // 2 or -1?
    }
}
```

因为我们传入的是new String("C")，所以一定是不同的实例。结果仍然符合预期，这是为什么呢？

因为List内部并不是通过==判断两个元素是否相等，而是使用equals()方法判断两个元素是否相等，例如contains()方法可以实现如下：

```
public class ArrayList {
    Object[] elementData;
    public boolean contains(Object o) {
        for (int i = 0; i < size; i++) {
            if (o.equals(elementData[i])) {
                return true;
            }
        }
        return false;
    }
}
```

因此，要正确使用List的contains()、indexOf()这些方法，放入的实例必须正确覆写equals()方法，否则，放进去的实例，查找不到。我们之所以能正常放入String、Integer这些对象，是因为Java标准库定义的这些类已经正确实现了equals()方法。

我们以Person对象为例，测试一下：

```
import java.util.List;
----
public class Main {
    public static void main(String[] args) {
        List<Person> list = List.of(
            new Person("Xiao Ming"),

```

```

        new Person("Xiao Hong"),
        new Person("Bob")
    );
    System.out.println(list.contains(new Person("Bob"))); // false
}

class Person {
    String name;
    public Person(String name) {
        this.name = name;
    }
}

```

不出意外，虽然放入了`new Person("Bob")`，但是用另一个`new Person("Bob")`查询不到，原因就是`Person`类没有覆写`equals()`方法。

## 编写equals

如何正确编写`equals()`方法？`equals()`方法要求我们必须满足以下条件：

- 自反性（**Reflexive**）：对于非`null`的`x`来说，`x.equals(x)`必须返回`true`；
- 对称性（**Symmetric**）：对于非`null`的`x`和`y`来说，如果`x.equals(y)`为`true`，则`y.equals(x)`也必须为`true`；
- 传递性（**Transitive**）：对于非`null`的`x`、`y`和`z`来说，如果`x.equals(y)`为`true`，`y.equals(z)`也为`true`，那么`x.equals(z)`也必须为`true`；
- 一致性（**Consistent**）：对于非`null`的`x`和`y`来说，只要`x`和`y`状态不变，则`x.equals(y)`总是一致地返回`true`或者`false`；
- 对`null`的比较：即`x.equals(null)`永远返回`false`。

上述规则看上去似乎非常复杂，但其实代码实现`equals()`方法是很简单的，我们以`Person`类为例：

```

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

```

首先，我们要定义“相等”的逻辑含义。对于`Person`类，如果`name`相等，并且`age`相等，我们就认为两个`Person`实例相等。

因此，编写`equals()`方法如下：

```

public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        return this.name.equals(p.name) && this.age == p.age;
    }
    return false;
}

```

对于引用字段比较，我们使用`equals()`，对于基本类型字段的比较，我们使用`==`。

如果`this.name`为`null`，那么`equals()`方法会报错，因此，需要继续改写如下：

```

public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        boolean nameEquals = false;
        if (this.name == null && p.name == null) {
            nameEquals = true;
        }
        if (this.name != null) {
            nameEquals = this.name.equals(p.name);
        }
        return nameEquals && this.age == p.age;
    }
    return false;
}

```



```
}
```

如果Person有好几个引用类型的字段，上面的写法就太复杂了。要简化引用类型的比较，我们使用Objects.equals()静态方法：

```
public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        return Objects.equals(this.name, p.name) && this.age == p.age;
    }
    return false;
}
```

因此，我们总结一下equals()方法的正确编写方法：

1. 先确定实例“相等”的逻辑，即哪些字段相等，就认为实例相等；
2. 用instanceof判断传入的待比较的Object是不是当前类型，如果是，继续比较，否则，返回false；
3. 对引用类型用Objects.equals()比较，对基本类型直接用==比较。

使用Objects.equals()比较两个引用类型是否相等的目的是省去了判断null的麻烦。两个引用类型都是null时它们也是相等的。

如果不调用List的contains()、indexOf()这些方法，那么放入的元素就不需要实现equals()方法。

## 练习

给Person类增加equals方法，使得调用indexOf()方法返回正常：

```
import java.util.List;
-----
public class Main {
    public static void main(String[] args) {
        List<Person> list = List.of(
            new Person("Xiao", "Ming", 18),
            new Person("Xiao", "Hong", 25),
            new Person("Bob", "Smith", 20)
        );
        boolean exist = list.contains(new Person("Bob", "Smith", 20));
        System.out.println(exist ? "测试成功!" : "测试失败!");
    }
}

class Person {
    String firstName;
    String lastName;
    int age;
    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
}
```

## [覆写equals方法](#)

## 小结

在List中查找元素时，List的实现类通过元素的equals()方法比较两个元素是否相等，因此，放入的元素必须正确覆写equals()方法，Java标准库提供的String、Integer等已经覆写了equals()方法；

编写equals()方法可借助Objects.equals()判断。

如果不在List中查找元素，就不必覆写equals()方法。

我们知道，List是一种顺序列表，如果有一个存储学生Student实例的List，要在List中根据name查找某个指定的Student的分数，应该怎么办？

## 使用Map

最简单的方法是遍历List并判断name是否相等，然后返回指定元素：

```
List<Student> list = ...
Student target = null;
for (Student s : list) {
    if ("Xiao Ming".equals(s.name)) {
        target = s;
        break;
    }
}
System.out.println(target.score);
```

这种需求其实非常常见，即通过一个键去查询对应的值。使用List来实现存在效率非常低的问题，因为平均需要扫描一半的元素才能确定，而Map这种键值（key-value）映射表的数据结构，作用就是能高效通过key快速查找value（元素）。

用Map来实现根据name查询某个Student的代码如下：

```
import java.util.HashMap;
import java.util.Map;
----
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Xiao Ming", 99);
        Map<String, Student> map = new HashMap<>();
        map.put("Xiao Ming", s); // 将"Xiao Ming"和Student实例映射并关联
        Student target = map.get("Xiao Ming"); // 通过key查找并返回映射的Student实例
        System.out.println(target == s); // true, 同一个实例
        System.out.println(target.score); // 99
        Student another = map.get("Bob"); // 通过另一个key查找
        System.out.println(another); // 未找到返回null
    }
}

class Student {
    public String name;
    public int score;
    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

通过上述代码可知：Map<K, V>是一种键-值映射表，当我们调用put(K key, V value)方法时，就把key和value做了映射并放入Map。当我们调用V get(K key)时，就可以通过key获取到对应的value。如果key不存在，则返回null。和List类似，Map也是一个接口，最常用的实现类是HashMap。

如果只是想查询某个key是否存在，可以调用boolean containsKey(K key)方法。

如果我们在存储Map映射关系的时候，对同一个key调用两次put()方法，分别放入不同的value，会有什么问题呢？例如：

```
import java.util.HashMap;
import java.util.Map;
----
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        System.out.println(map.get("apple")); // 123
    }
}
```

```

        map.put("apple", 789); // 再次放入apple作为key, 但value变为789
        System.out.println(map.get("apple")); // 789
    }
}

```

重复放入key-value并不会有任何问题，但是一个key只能关联一个value。在上面的代码中，一开始我们把key对象"apple"映射到Integer对象123，然后再次调用put()方法把"apple"映射到789，这时，原来关联的value对象123就被“冲掉”了。实际上，put()方法的签名是V put(K key, V value)，如果放入的key已经存在，put()方法会返回被删除的旧的value，否则，返回null。

始终牢记：Map中不存在重复的key，因为放入相同的key，只会把原有的key-value对应的value给替换掉。

此外，在一个Map中，虽然key不能重复，但value是可以重复的：

```

Map<String, Integer> map = new HashMap<>();
map.put("apple", 123);
map.put("pear", 123); // ok

```

## 遍历Map

对Map来说，要遍历key可以使用for each循环遍历Map实例的keySet()方法返回的Set集合，它包含不重复的key的集合：

```

import java.util.HashMap;
import java.util.Map;
----
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        map.put("banana", 789);
        for (String key : map.keySet()) {
            Integer value = map.get(key);
            System.out.println(key + " = " + value);
        }
    }
}

```

同时遍历key和value可以使用for each循环遍历Map对象的entrySet()集合，它包含每一个key-value映射：

```

import java.util.HashMap;
import java.util.Map;
----
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        map.put("banana", 789);
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println(key + " = " + value);
        }
    }
}

```

Map和List不同的是，Map存储的是key-value的映射关系，并且，它不保证顺序。在遍历的时候，遍历的顺序既不一定是put()时放入的key的顺序，也不一定是key的排序顺序。使用Map时，任何依赖顺序的逻辑都是不可靠的。以HashMap为例，假设我们放入"A", "B", "C"这3个key，遍历的时候，每个key会保证被遍历一次且仅遍历一次，但顺序完全没有保证，甚至对于不同的JDK版本，相同的代码遍历的输出顺序都是不同的！

遍历Map时，不可假设输出的key是有序的！

## 练习

请编写一个根据name查找score的程序，并利用Map充当缓存，以提高查找效率：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        List<Student> list = List.of(
            new Student("Bob", 78),
            new Student("Alice", 85),
            new Student("Brush", 66),
            new Student("Newton", 99));
        var holder = new Students(list);
        System.out.println(holder.getScore("Bob") == 78 ? "测试成功!" : "测试失败!");
        System.out.println(holder.getScore("Alice") == 85 ? "测试成功!" : "测试失败!");
        System.out.println(holder.getScore("Tom") == -1 ? "测试成功!" : "测试失败!");
    }
}

class Students {
    List<Student> list;
    Map<String, Integer> cache;

    Students(List<Student> list) {
        this.list = list;
        cache = new HashMap<>();
    }

    /**
     * 根据name查找score，找到返回score，未找到返回-1
     */
    int getScore(String name) {
        // 先在Map中查找：
        Integer score = this.cache.get(name);
        if (score == null) {
            // TODO:
        }
        return score == null ? -1 : score.intValue();
    }

    Integer findInList(String name) {
        for (var ss : this.list) {
            if (ss.name.equals(name)) {
                return ss.score;
            }
        }
        return null;
    }
}

class Student {
    String name;
    int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

[find-student-score](#)

## 小结

Map是一种映射表，可以通过key快速查找value。

可以通过for each遍历keySet()，也可以通过for each遍历entrySet()，直接获取key-value。

最常用的一种Map实现是HashMap。

我们知道Map是一种键-值（key-value）映射表，可以通过key快速查找对应的value。

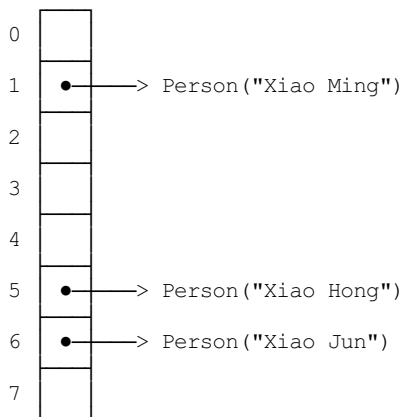
## 编写equals和hashCode

以HashMap为例，观察下面的代码：

```
Map<String, Person> map = new HashMap<>();
map.put("a", new Person("Xiao Ming"));
map.put("b", new Person("Xiao Hong"));
map.put("c", new Person("Xiao Jun"));

map.get("a"); // Person("Xiao Ming")
map.get("x"); // null
```

HashMap之所以能根据key直接拿到value，原因是它内部通过空间换时间的方法，用一个大数组存储所有value，并根据key直接计算出value应该存储在哪个索引：



如果key的值为"a"，计算得到的索引总是1，因此返回value为Person("Xiao Ming")，如果key的值为"b"，计算得到的索引总是5，因此返回value为Person("Xiao Hong")，这样，就不必遍历整个数组，即可直接读取key对应的value。

当我们使用key存取value的时候，就会引出一个问题：

我们放入Map的key是字符串"a"，但是，当我们获取Map的value时，传入的变量不一定就是放入的那个key对象。

换句话说讲，两个key应该是内容相同，但不一定是同一个对象。测试代码如下：

```
import java.util.HashMap;
import java.util.Map;
----
public class Main {
    public static void main(String[] args) {
        String key1 = "a";
        Map<String, Integer> map = new HashMap<>();
        map.put(key1, 123);

        String key2 = new String("a");
        map.get(key2); // 123

        System.out.println(key1 == key2); // false
        System.out.println(key1.equals(key2)); // true
    }
}
```

因为在Map的内部，对key做比较是通过equals()实现的，这一点和List查找元素需要正确覆写equals()是一样的，即正确使用Map必须保证：作为key的对象必须正确覆写equals()方法。

我们经常使用String作为key，因为String已经正确覆写了equals()方法。但如果我们放入的key是一个自己写的类，就必须保证正确覆写了equals()方法。

我们再思考一下HashMap为什么能通过key直接计算出value存储的索引。相同的key对象（使用equals()判断时返回true）必须要计算出相同的索引，否则，相同的key每次取出的value就不一定对。

通过key计算索引的方式就是调用key对象的hashCode()方法，它返回一个int整数。HashMap正是通过这个方法直接定位key对应的value的索引，继而直接返回value。

因此，正确使用Map必须保证：

1. 作为key的对象必须正确覆写equals()方法，相等的两个key实例调用equals()必须返回true；
2. 作为key的对象还必须正确覆写hashCode()方法，且hashCode()方法要严格遵循以下规范：
  - 如果两个对象相等，则两个对象的hashCode()必须相等；
  - 如果两个对象不相等，则两个对象的hashCode()尽量不要相等。

即对应两个实例a和b：

- 如果a和b相等，那么a.equals(b)一定为true，则a.hashCode()必须等于b.hashCode()；
- 如果a和b不相等，那么a.equals(b)一定为false，则a.hashCode()和b.hashCode()尽量不要相等。

上述第一条规范是正确性，必须保证实现，否则HashMap不能正常工作。

而第二条如果尽量满足，则可以保证查询效率，因为不同的对象，如果返回相同的hashCode()，会造成Map内部存储冲突，使存取的效率下降。

正确编写equals()的方法我们已经在[编写equals方法](#)一节中讲过了，以Person类为例：

```
public class Person {
    String firstName;
    String lastName;
    int age;
}
```

把需要比较的字段找出来：

- **firstName**
- **lastName**
- **age**

然后，引用类型使用Objects.equals()比较，基本类型使用==比较。

在正确实现equals()的基础上，我们还需要正确实现hashCode()，即上述3个字段分别相同的实例，hashCode()返回的int必须相同：

```
public class Person {
    String firstName;
    String lastName;
    int age;

    @Override
    int hashCode() {
        int h = 0;
        h = 31 * h + firstName.hashCode();
        h = 31 * h + lastName.hashCode();
        h = 31 * h + age;
        return h;
    }
}
```

注意到String类已经正确实现了hashCode()方法，我们在计算Person的hashCode()时，反复使用31\*h，这样做的目的是为了尽量把不同的Person实例的hashCode()均匀分布到整个int范围。

和实现equals()方法遇到的问题类似，如果firstName或lastName为null，上述代码工作起来就会抛NullPointerException。为了解决这个问题，我们在计算hashCode()的时候，经常借助Objects.hashCode()来计

算：

```
int hashCode() {  
    return Objects.hashCode(firstName, lastName, age);  
}
```

所以，编写`equals()`和`hashCode()`遵循的原则是：

`equals()`用到的用于比较的每一个字段，都必须在`hashCode()`中用于计算；`equals()`中没有使用到的字段，绝不可放在`hashCode()`中计算。

另外注意，对于放入`HashMap`的`value`对象，没有任何要求。

## 延伸阅读

既然`HashMap`内部使用了数组，通过计算`key`的`hashCode()`直接定位`value`所在的索引，那么第一个问题来了：**`hashCode()`返回的`int`范围高达 $\pm 21$ 亿**，先不考虑负数，`HashMap`内部使用的数组得有多大？

实际上`HashMap`初始化时默认的数组大小只有16，任何`key`，无论它的`hashCode()`有多大，都可以简单地通过：

```
int index = key.hashCode() & 0xf; // 0xf = 15
```

把索引确定在0~15，即永远不会超出数组范围，上述算法只是一种最简单的实现。

第二个问题：如果添加超过16个`key-value`到`HashMap`，数组不够用了怎么办？

添加超过一定数量的`key-value`时，`HashMap`会在内部自动扩容，每次扩容一倍，即长度为16的数组扩展为长度32，相应地，需要重新确定`hashCode()`计算的索引位置。例如，对长度为32的数组计算`hashCode()`对应的索引，计算方式要改为：

```
int index = key.hashCode() & 0x1f; // 0x1f = 31
```

由于扩容会导致重新分布已有的`key-value`，所以，频繁扩容对`HashMap`的性能影响很大。如果我们确定要使用一个容量为10000个`key-value`的`HashMap`，更好的方式是创建`HashMap`时就指定容量：

```
Map<String, Integer> map = new HashMap<>(10000);
```

虽然指定容量是10000，但`HashMap`内部的数组长度总是 $2^n$ ，因此，实际数组长度被初始化为比10000大的16384 ( $2^{14}$ )。

最后一个问题：如果不同的两个`key`，例如"a"和"b"，它们的`hashCode()`恰好是相同的（这种情况是完全可能的，因为不相等的两个实例，只要求`hashCode()`尽量不相等），那么，当我们放入：

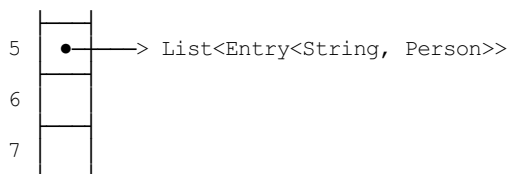
```
map.put("a", new Person("Xiao Ming"));  
map.put("b", new Person("Xiao Hong"));
```

时，由于计算出的数组索引相同，后面放入的"Xiao Hong"会不会把"Xiao Ming"覆盖了？

当然不会！使用`Map`的时候，只要`key`不相同，它们映射的`value`就互不干扰。但是，在`HashMap`内部，确实可能存在不同的`key`，映射到相同的`hashCode()`，即相同的数组索引上，肿么办？

我们就假设"a"和"b"这两个`key`最终计算出的索引都是5，那么，在`HashMap`的数组中，实际存储的不是一个`Person`实例，而是一个`List`，它包含两个`Entry`，一个是"a"的映射，一个是"b"的映射：

0	
1	
2	
3	
4	



在查找的时候，例如：

```
Person p = map.get("a");
```

**HashMap**内部通过"a"找到的实际上是`List<Entry<String, Person>>`，它还需要遍历这个List，并找到一个Entry，它的key字段是"a"，才能返回对应的Person实例。

我们把不同的key具有相同的`hashCode()`的情况称之为哈希冲突。在冲突的时候，一种最简单的解决办法是用List存储`hashCode()`相同的key-value。显然，如果冲突的概率越大，这个List就越长，Map的`get()`方法效率就越低，这就是为什么要尽量满足条件二：

如果两个对象不相等，则两个对象的`hashCode()`尽量不要相等。

`hashCode()`方法编写得越好，HashMap工作的效率就越高。

## 小结

要正确使用HashMap，作为key的类必须正确覆写`equals()`和`hashCode()`方法；

一个类如果覆写了`equals()`，就必须覆写`hashCode()`，并且覆写规则是：

- 如果`equals()`返回true，则`hashCode()`返回值必须相等；
- 如果`equals()`返回false，则`hashCode()`返回值尽量不要相等。

实现`hashCode()`方法可以通过`Objects.hashCode()`辅助方法实现。



因为HashMap是一种通过对key计算hashCode(), 通过空间换时间的方式, 直接定位到value所在的内部数组的索引, 因此, 查找效率非常高。

## 使用EnumMap

如果作为key的对象是enum类型, 那么, 还可以使用Java集合库提供的一种EnumMap, 它在内部以一个非常紧凑的数组存储value, 并且根据enum类型的key直接定位到内部数组的索引, 并不需要计算hashCode(), 不但效率最高, 而且没有额外的空间浪费。

我们以DayOfWeek这个枚举类型为例, 为它做一个“翻译”功能:

```
import java.time.DayOfWeek;
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        Map<DayOfWeek, String> map = new EnumMap<>(DayOfWeek.class);
        map.put(DayOfWeek.MONDAY, "星期一");
        map.put(DayOfWeek.TUESDAY, "星期二");
        map.put(DayOfWeek.WEDNESDAY, "星期三");
        map.put(DayOfWeek.THURSDAY, "星期四");
        map.put(DayOfWeek.FRIDAY, "星期五");
        map.put(DayOfWeek.SATURDAY, "星期六");
        map.put(DayOfWeek.SUNDAY, "星期日");
        System.out.println(map);
        System.out.println(map.get(DayOfWeek.MONDAY));
    }
}
```

使用EnumMap的时候, 我们总是用Map接口来引用它, 因此, 实际上把HashMap和EnumMap互换, 在客户端看来没有任何区别。

### 小结

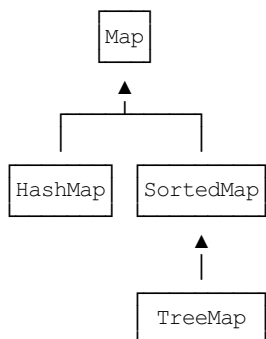
如果Map的key是enum类型, 推荐使用EnumMap, 既保证速度, 也不浪费空间。

使用EnumMap的时候, 根据面向抽象编程的原则, 应持有Map接口。

我们已经知道，HashMap是一种以空间换时间的映射表，它的实现原理决定了内部的Key是无序的，即遍历HashMap的Key时，其顺序是不可预测的（但每个Key都会遍历一次且仅遍历一次）。

## 使用TreeMap

还有一种Map，它在内部会对Key进行排序，这种Map就是SortedMap。注意到SortedMap是接口，它的实现类是TreeMap。



SortedMap保证遍历时以Key的顺序来进行排序。例如，放入的Key是"apple"、"pear"、"orange"，遍历的顺序一定是"apple"、"orange"、"pear"，因为String默认按字母排序：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new TreeMap<>();
        map.put("orange", 1);
        map.put("apple", 2);
        map.put("pear", 3);
        for (String key : map.keySet()) {
            System.out.println(key);
        }
        // apple, orange, pear
    }
}
```

使用TreeMap时，放入的Key必须实现Comparable接口。String、Integer这些类已经实现了Comparable接口，因此可以直接作为Key使用。作为Value的对象则没有任何要求。

如果作为Key的class没有实现Comparable接口，那么，必须在创建TreeMap时同时指定一个自定义排序算法：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        Map<Person, Integer> map = new TreeMap<>(new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.name.compareTo(p2.name);
            }
        });
        map.put(new Person("Tom"), 1);
        map.put(new Person("Bob"), 2);
        map.put(new Person("Lily"), 3);
        for (Person key : map.keySet()) {
            System.out.println(key);
        }
        // {Person: Bob}, {Person: Lily}, {Person: Tom}
        System.out.println(map.get(new Person("Bob"))); // 2
    }
}

class Person {
```

```

    public String name;
    Person(String name) {
        this.name = name;
    }
    public String toString() {
        return "{Person: " + name + "}";
    }
}

```

注意到Comparator接口要求实现一个比较方法，它负责比较传入的两个元素a和b，如果a<b，则返回负数，通常是-1，如果a==b，则返回0，如果a>b，则返回正数，通常是1。TreeMap内部根据比较结果对Key进行排序。

从上述代码执行结果可知，打印的Key确实是按照Comparator定义的顺序排序的。如果要根据Key查找Value，我们可以传入一个new Person("Bob")作为Key，它会返回对应的Integer值2。

另外，注意到Person类并未覆写equals()和hashCode()，因为TreeMap不使用equals()和hashCode()。

我们来看一个稍微复杂的例子：这次我们定义了Student类，并用分数score进行排序，高分在前：

```

import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        Map<Student, Integer> map = new TreeMap<>(new Comparator<Student>() {
            public int compare(Student p1, Student p2) {
                return p1.score > p2.score ? -1 : 1;
            }
        });
        map.put(new Student("Tom", 77), 1);
        map.put(new Student("Bob", 66), 2);
        map.put(new Student("Lily", 99), 3);
        for (Student key : map.keySet()) {
            System.out.println(key);
        }
        System.out.println(map.get(new Student("Bob", 66))); // null?
    }
}

class Student {
    public String name;
    public int score;
    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String toString() {
        return String.format("{%s: score=%d}", name, score);
    }
}

```

在for循环中，我们确实得到了正确的顺序。但是，且慢！根据相同的Key: new Student("Bob", 66)进行查找时，结果为null！

这是怎么肥四？难道TreeMap有问题？遇到TreeMap工作不正常时，我们首先回顾Java编程基本规则：出现问题，不要怀疑Java标准库，要从自身代码找原因。

在这个例子中，TreeMap出现问题，原因其实出在这个Comparator上：

```

public int compare(Student p1, Student p2) {
    return p1.score > p2.score ? -1 : 1;
}

```

在p1.score和p2.score不相等的时候，它的返回值是正确的，但是，在p1.score和p2.score相等的时候，它并没有返回0！这就是为什么TreeMap工作不正常的原因：TreeMap在比较两个Key是否相等时，依赖Key的compareTo()方法或者Comparator.compare()方法。在两个Key相等时，必须返回0。因此，修改代码如下：

```

public int compare(Student p1, Student p2) {

```

```
    if (p1.score == p2.score) {  
        return 0;  
    }  
    return p1.score > p2.score ? -1 : 1;  
}
```

或者直接借助`Integer.compare(int, int)`也可以返回正确的比较结果。

## 小结

`SortedMap`在遍历时严格按照**Key**的顺序遍历，最常用的实现类是`TreeMap`；

作为`SortedMap`的**Key**必须实现`Comparable`接口，或者传入`Comparator`；

要严格按照`compare()`规范实现比较逻辑，否则，`TreeMap`将不能正常工作。

在编写应用程序的时候，经常需要读写配置文件。例如，用户的设置：

## 使用Properties

```
# 上次最后打开的文件：
last_open_file=/data/hello.txt
# 自动保存文件的时间间隔：
auto_save_interval=60
```

配置文件的特点是，它的**Key-Value**一般都是String-String类型的，因此我们完全可以用Map<String, String>来表示它。

因为配置文件非常常用，所以Java集合库提供了一个Properties来表示一组“配置”。由于历史遗留原因，Properties内部本质上是一个Hashtable，但我们只需要用到Properties自身关于读写配置的接口。

### 读取配置文件

用Properties读取配置文件非常简单。Java默认配置文件以.properties为扩展名，每行以key=value表示，以#开头的是注释。以下是一个典型的配置文件：

```
# setting.properties

last_open_file=/data/hello.txt
auto_save_interval=60
```

可以从文件系统读取这个.properties文件：

```
String f = "setting.properties";
Properties props = new Properties();
props.load(new java.io.FileInputStream(f));

String filepath = props.getProperty("last_open_file");
String interval = props.getProperty("auto_save_interval", "120");
```

可见，用Properties读取配置文件，一共有三步：

1. 创建Properties实例；
2. 调用load()读取文件；
3. 调用getProperty()获取配置。

调用getProperty()获取配置时，如果key不存在，将返回null。我们还可以提供一个默认值，这样，当key不存在的时候，就返回默认值。

也可以从classpath读取.properties文件，因为load(InputStream)方法接收一个InputStream实例，表示一个字节流，它不一定是文件流，也可以是从jar包中读取的资源流：

```
Properties props = new Properties();
props.load(getClass().getResourceAsStream("/common/setting.properties"));
```

试试从内存读取一个字节流：

```
// properties
----
import java.io.*;
import java.util.Properties;

public class Main {
    public static void main(String[] args) throws IOException {
        String settings = "# test" + "\n" + "course=Java" + "\n" + "last_open_date=2019-08-07T12:35:01";
        ByteArrayInputStream input = new ByteArrayInputStream(settings.getBytes("UTF-8"));
        Properties props = new Properties();
        props.load(input);

        System.out.println("course: " + props.getProperty("course"));
    }
}
```

```

        System.out.println("last_open_date: " + props.getProperty("last_open_date"));
        System.out.println("last_open_file: " + props.getProperty("last_open_file"));
        System.out.println("auto_save: " + props.getProperty("auto_save", "60"));
    }
}

```

如果有多个.properties文件，可以反复调用load()读取，后读取的key-value会覆盖已读取的key-value:

```

Properties props = new Properties();
props.load(getClass().getResourceAsStream("/common/setting.properties"));
props.load(new FileInputStream("C:\\conf\\setting.properties"));

```

上面的代码演示了Properties的一个常用用法：可以把默认配置文件放到classpath中，然后，根据机器的环境编写另一个配置文件，覆盖某些默认的配置。

Properties设计的目的是存储String类型的key-value，但Properties实际上是从Hashtable派生的，它的设计实际上是有问题的，但是为了保持兼容性，现在已经没法修改了。除了getProperty()和setProperty()方法外，还有从Hashtable继承下来的get()和put()方法，这些方法的参数签名是Object，我们在使用Properties的时候，不要去调用这些从Hashtable继承下来的方法。

## 写入配置文件

如果通过setProperty()修改了Properties实例，可以把配置写入文件，以便下次启动时获得最新配置。写入配置文件使用store()方法:

```

Properties props = new Properties();
props.setProperty("url", "http://www.liaoxuefeng.com");
props.setProperty("language", "Java");
props.store(new FileOutputStream("C:\\conf\\setting.properties"), "这是写入的properties注释");

```

## 编码

早期版本的Java规定.properties文件编码是ASCII编码（ISO8859-1），如果涉及到中文就必须用name=\u4e2d\u6587来表示，非常别扭。从JDK9开始，Java的.properties文件可以使用UTF-8编码了。

不过，需要注意的是，由于load(InputStream)默认总是以ASCII编码读取字节流，所以会导致读到乱码。我们需要用另一个重载方法load(Reader)读取:

```

Properties props = new Properties();
props.load(new FileReader("settings.properties", StandardCharsets.UTF_8));

```

就可以正常读取中文。InputStream和Reader的区别是一个是字节流，一个是字符流。字符流在内存中已经以char类型表示了，不涉及编码问题。

## 小结

Java集合库提供的Properties用于读写配置文件.properties。 .properties文件可以使用UTF-8编码。

可以从文件系统、classpath或其他任何地方读取.properties文件。

读写Properties时，注意仅使用getProperty()和setProperty()方法，不要调用继承而来的get()和put()等方法。

我们知道，Map用于存储**key-value**的映射，对于充当**key**的对象，是不能重复的，并且，不但需要正确覆写equals()方法，还要正确覆写hashCode()方法。

## 使用Set

如果我们只需要存储不重复的**key**，并不需要存储映射的**value**，那么就可以使用Set。

Set用于存储不重复的元素集合，它主要提供以下几个方法：

- 将元素添加进Set<E>: boolean add(E e)
- 将元素从Set<E>删除: boolean remove(Object e)
- 判断是否包含元素: boolean contains(Object e)

我们来看几个简单的例子：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        System.out.println(set.add("abc")); // true
        System.out.println(set.add("xyz")); // true
        System.out.println(set.add("xyz")); // false, 添加失败, 因为元素已存在
        System.out.println(set.contains("xyz")); // true, 元素存在
        System.out.println(set.contains("XYZ")); // false, 元素不存在
        System.out.println(set.remove("hello")); // false, 删除失败, 因为元素不存在
        System.out.println(set.size()); // 2, 一共两个元素
    }
}
```

Set实际上相当于只存储**key**、不存储**value**的Map。我们经常用Set用于去除重复元素。

因为放入Set的元素和Map的**key**类似，都要正确实现equals()和hashCode()方法，否则该元素无法正确地放入Set。

最常用的Set实现类是HashSet，实际上，HashSet仅仅是对HashMap的一个简单封装，它的核心代码如下：

```
public class HashSet<E> implements Set<E> {
    // 持有一个HashMap:
    private HashMap<E, Object> map = new HashMap<>();

    // 放入HashMap的value:
    private static final Object PRESENT = new Object();

    public boolean add(E e) {
        return map.put(e, PRESENT) == null;
    }

    public boolean contains(Object o) {
        return map.containsKey(o);
    }

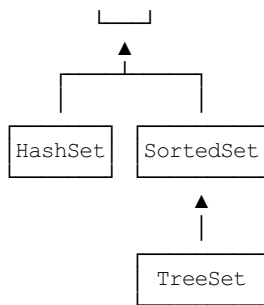
    public boolean remove(Object o) {
        return map.remove(o) == PRESENT;
    }
}
```

Set接口并不保证有序，而SortedSet接口则保证元素是有序的：

- HashSet是无序的，因为它实现了Set接口，并没有实现SortedSet接口；
- TreeSet是有序的，因为它实现了SortedSet接口。

用一张图表示：

Set



我们来看HashSet的输出：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("apple");
        set.add("banana");
        set.add("pear");
        set.add("orange");
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

注意输出的顺序既不是添加的顺序，也不是String排序的顺序，在不同版本的JDK中，这个顺序也可能是不同的。

把HashSet换成TreeSet，在遍历TreeSet时，输出就是有序的，这个顺序是元素的排序顺序：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        Set<String> set = new TreeSet<>();
        set.add("apple");
        set.add("banana");
        set.add("pear");
        set.add("orange");
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

使用TreeSet和使用TreeMap的要求一样，添加的元素必须正确实现Comparable接口，如果没有实现Comparable接口，那么创建TreeSet时必须传入一个Comparator对象。

## 练习

在聊天软件中，发送方发送消息时，遇到网络超时后就会自动重发，因此，接收方可能会收到重复的消息，在显示给用户看的时候，需要首先去重。请练习使用Set去除重复的消息：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        List<Message> received = List.of(
            new Message(1, "Hello!"),
            new Message(2, "发工资了吗? "),
            new Message(2, "发工资了吗? "),
            new Message(3, "去哪吃饭? "),
            new Message(3, "去哪吃饭? "),
            new Message(4, "Bye")
        );
    }
}
```



```

    );
    List<Message> displayMessages = process(received);
    for (Message message : displayMessages) {
        System.out.println(message.text);
    }
}

static List<Message> process(List<Message> received) {
    // TODO: 按sequence去除重复消息
    return received;
}

}

class Message {
    public final int sequence;
    public final String text;
    public Message(int sequence, String text) {
        this.sequence = sequence;
        this.text = text;
    }
}

```

## 小结

Set用于存储不重复的元素集合：

- 放入HashSet的元素与作为HashMap的key要求相同；
- 放入TreeSet的元素与作为TreeMap的Key要求相同；

利用Set可以去除重复元素；

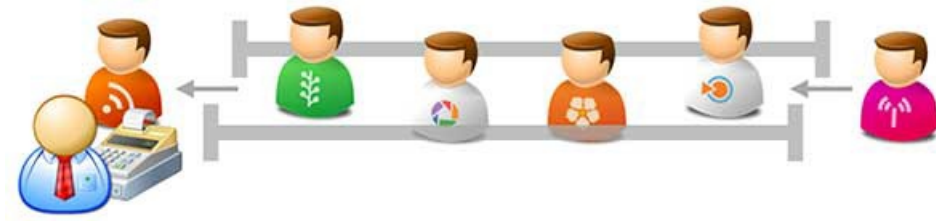
遍历SortedSet按照元素的排序顺序遍历，也可以自定义排序算法。

队列（Queue）是一种经常使用的集合。Queue实际上是实现了一个先进先出（FIFO: First In First Out）的有序表。它和List的区别在于，List可以在任意位置添加和删除元素，而Queue只有两个操作：

## 使用Queue

- 把元素添加到队列末尾；
- 从队列头部取出元素。

超市的收银台就是一个队列：



在Java的标准库中，队列接口Queue定义了以下几个方法：

- `int size()`：获取队列长度；
- `boolean add(E)/boolean offer(E)`：添加元素到队尾；
- `E remove()/E poll()`：获取队首元素并从队列中删除；
- `E element()/E peek()`：获取队首元素但并不从队列中删除。

对于具体的实现类，有的Queue有最大队列长度限制，有的Queue没有。注意到添加、删除和获取队列元素总是有两个方法，这是因为在添加或获取元素失败时，这两个方法的行为是不同的。我们用一个表格总结如下：

**throw Exception 返回false或null**

添加元素到队尾	<code>add(E e)</code>	<code>boolean offer(E e)</code>
取队首元素并删除	<code>E remove()</code>	<code>E poll()</code>
取队首元素但不删除	<code>E element()</code>	<code>E peek()</code>

举个栗子，假设我们有一个队列，对它做一个添加操作，如果调用`add()`方法，当添加失败时（可能超过了队列的容量），它会抛出异常：

```
Queue<String> q = ...
try {
    q.add("Apple");
    System.out.println("添加成功");
} catch (IllegalStateException e) {
    System.out.println("添加失败");
}
```

如果我们调用`offer()`方法来添加元素，当添加失败时，它不会抛异常，而是返回`false`：

```
Queue<String> q = ...
if (q.offer("Apple")) {
    System.out.println("添加成功");
} else {
    System.out.println("添加失败");
}
```

当我们需要从Queue中取出队首元素时，如果当前Queue是一个空队列，调用`remove()`方法，它会抛出异常：

```
Queue<String> q = ...
try {
    String s = q.remove();
    System.out.println("获取成功");
} catch (IllegalStateException e) {
    // 空队列异常
}
```

```

        System.out.println("获取失败");
    }
}

```

如果我们调用`poll()`方法来取出队首元素，当获取失败时，它不会抛异常，而是返回`null`：

```

Queue<String> q = ...
String s = q.poll();
if (s != null) {
    System.out.println("获取成功");
} else {
    System.out.println("获取失败");
}

```

因此，两套方法可以根据需要来选择使用。

注意：不要把`null`添加到队列中，否则`poll()`方法返回`null`时，很难确定是取到了`null`元素还是队列为空。

接下来我们以`poll()`和`peek()`为例来说“获取并删除”与“获取但不删除”的区别。对于`Queue`来说，每次调用`poll()`，都会获取队首元素，并且获取到的元素已经从队列中被删除了：

```

import java.util.LinkedList;
import java.util.Queue;
----
public class Main {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<>();
        // 添加3个元素到队列：
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        // 从队列取出元素：
        System.out.println(q.poll()); // apple
        System.out.println(q.poll()); // pear
        System.out.println(q.poll()); // banana
        System.out.println(q.poll()); // null, 因为队列是空的
    }
}

```

如果用`peek()`，因为获取队首元素时，并不会从队列中删除这个元素，所以可以反复获取：

```

import java.util.LinkedList;
import java.util.Queue;
----
public class Main {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<>();
        // 添加3个元素到队列：
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        // 队首永远都是apple，因为peek()不会删除它：
        System.out.println(q.peek()); // apple
        System.out.println(q.peek()); // apple
        System.out.println(q.peek()); // apple
    }
}

```

从上面的代码中，我们还可以发现，`LinkedList`即实现了`List`接口，又实现了`Queue`接口，但是，在使用的时候，如果我们把它当作`List`，就获取`List`的引用，如果我们把它当作`Queue`，就获取`Queue`的引用：

```

// 这是一个List：
List<String> list = new LinkedList<>();
// 这是一个Queue：
Queue<String> queue = new LinkedList<>();

```

始终按照面向对象编程的原则编写代码，可以大大提高代码的质量。

## 小结

队列Queue实现了一个先进先出（FIFO）的数据结构：

- 通过`add()`/`offer()`方法将元素添加到队尾；
- 通过`remove()`/`poll()`从队首获取元素并删除；
- 通过`element()`/`peek()`从队首获取元素但不删除。

要避免把`null`添加到队列。

我们知道，Queue是一个先进先出（FIFO）的队列。

## 使用PriorityQueue

在银行柜台办业务时，我们假设只有一个柜台在办理业务，但是办理业务的人很多，怎么办？

可以每个人先取一个号，例如：A1、A2、A3.....然后，按照号码顺序依次办理，实际上这就是一个Queue。

如果这时来了一个VIP客户，他的号码是V1，虽然当前排队的是A10、A11、A12.....但是柜台下一个呼叫的客户号码却是V1。

这个时候，我们发现，要实现“VIP插队”的业务，用Queue就不行了，因为Queue会严格按FIFO的原则取出队首元素。我们需要的是优先队列：PriorityQueue。

PriorityQueue和Queue的区别在于，它的出队顺序与元素的优先级有关，对PriorityQueue调用remove()或poll()方法，返回的总是优先级最高的元素。

要使用PriorityQueue，我们就必须给每个元素定义“优先级”。我们以实际代码为例，先看看PriorityQueue的行为：

```
import java.util.PriorityQueue;
import java.util.Queue;
----
public class Main {
    public static void main(String[] args) {
        Queue<String> q = new PriorityQueue<>();
        // 添加3个元素到队列：
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        System.out.println(q.poll()); // apple
        System.out.println(q.poll()); // banana
        System.out.println(q.poll()); // pear
        System.out.println(q.poll()); // null,因为队列为空
    }
}
```

我们放入的顺序是"apple"、"pear"、"banana"，但是取出的顺序却是"apple"、"banana"、"pear"，这是因为从字符串的排序看，"apple"排在最前面，"pear"排在最后面。

因此，放入PriorityQueue的元素，必须实现Comparable接口，PriorityQueue会根据元素的排序顺序决定出队的优先级。

如果我们要放入的元素并没有实现Comparable接口怎么办？PriorityQueue允许我们提供一个Comparator对象来判断两个元素的顺序。我们以银行排队业务为例，实现一个PriorityQueue：

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
----
public class Main {
    public static void main(String[] args) {
        Queue<User> q = new PriorityQueue<>(new UserComparator());
        // 添加3个元素到队列：
        q.offer(new User("Bob", "A1"));
        q.offer(new User("Alice", "A2"));
        q.offer(new User("Boss", "V1"));
        System.out.println(q.poll()); // Boss/V1
        System.out.println(q.poll()); // Bob/A1
        System.out.println(q.poll()); // Alice/A2
        System.out.println(q.poll()); // null,因为队列为空
    }
}

class UserComparator implements Comparator<User> {
    public int compare(User u1, User u2) {
```

```

        if (u1.number.charAt(0) == u2.number.charAt(0)) {
            // 如果两人的号都是A开头或者都是V开头,比较号的大小:
            return u1.number.compareTo(u2.number);
        }
        if (u1.number.charAt(0) == 'V') {
            // u1的号码是V开头,优先级高:
            return -1;
        } else {
            return 1;
        }
    }
}

class User {
    public final String name;
    public final String number;

    public User(String name, String number) {
        this.name = name;
        this.number = number;
    }

    public String toString() {
        return name + "/" + number;
    }
}

```

实现PriorityQueue的关键在于提供的UserComparator对象，它负责比较两个元素的大小（较小的在前）。UserComparator总是把V开头的号码优先返回，只有在开头相同的时候，才比较号码大小。

上面的UserComparator的比较逻辑其实还是有问题的，它会把A10排在A2的前面，请尝试修复该错误。

## 小结

PriorityQueue实现了一个优先队列：从队首获取元素时，总是获取优先级最高的元素。

PriorityQueue默认按元素比较的顺序排序（必须实现Comparable接口），也可以通过Comparator自定义排序算法（元素就不必实现Comparable接口）。

我们知道，Queue是队列，只能一头进，另一头出。

## 使用Deque

如果把条件放松一下，允许两头都进，两头都出，这种队列叫双端队列（Double Ended Queue），学名Deque。

Java集合提供了接口Deque来实现一个双端队列，它的功能是：

- 既可以添加到队尾，也可以添加到队首；
- 既可以从队首获取，又可以从队尾获取。

我们来比较一下Queue和Deque出队和入队的方法：

	Queue	Deque
添加元素到队尾	add(E e) / offer(E e)	addLast(E e) / offerLast(E e)
取队首元素并删除	E remove() / E poll()	E removeFirst() / E pollFirst()
取队首元素但不删除	E element() / E peek()	E getFirst() / E peekFirst()
添加元素到队首	无	addFirst(E e) / offerFirst(E e)
取队尾元素并删除	无	E removeLast() / E pollLast()
取队尾元素但不删除	无	E getLast() / E peekLast()

对于添加元素到队尾的操作，Queue提供了add()/offer()方法，而Deque提供了addLast()/offerLast()方法。添加元素到对首、取队尾元素的操作在Queue中不存在，在Deque中由addFirst()/removeLast()等方法提供。

注意到Deque接口实际上扩展自Queue：

```
public interface Deque<E> extends Queue<E> {  
    ...  
}
```

因此，Queue提供的add()/offer()方法在Deque中也可以使用，但是，使用Deque，最好不要调用offer()，而是调用offerLast()：

```
import java.util.Deque;  
import java.util.LinkedList;  
-----  
public class Main {  
    public static void main(String[] args) {  
        Deque<String> deque = new LinkedList<>();  
        deque.offerLast("A"); // A  
        deque.offerLast("B"); // B -> A  
        deque.offerFirst("C"); // B -> A -> C  
        System.out.println(deque.pollFirst()); // C, 剩下B -> A  
        System.out.println(deque.pollLast()); // B  
        System.out.println(deque.pollFirst()); // A  
        System.out.println(deque.pollFirst()); // null  
    }  
}
```

如果直接写deque.offer()，我们就需要思考，offer()实际上是offerLast()，我们明确地写上offerLast()，不需要思考就能一眼看出这是添加到队尾。

因此，使用Deque，推荐总是明确调用offerLast()/offerFirst()或者pollFirst()/pollLast()方法。

Deque是一个接口，它的实现类有ArrayDeque和LinkedList。

我们发现LinkedList真是一个全能选手，它即是List，又是Queue，还是Deque。但是我们在使用的时候，总是用特定的接口来引用它，这是因为持有接口说明代码的抽象层次更高，而且接口本身定义的方法代表了特定的用途。

// 不推荐的写法：

```
LinkedList<String> dl = new LinkedList<>();
```

```
dl.offerLast("z");  
// 推荐的写法:  
Deque<String> d2 = new LinkedList<>();  
d2.offerLast("z");
```

可见面向抽象编程的一个原则就是：尽量持有接口，而不是具体的实现类。

## 练习

## 小结

Deque实现了一个双端队列（**Double Ended Queue**），它可以：

- 将元素添加到队尾或队首：addLast()/offerLast()/addFirst()/offerFirst();
- 从队首 / 队尾获取元素并删除：removeFirst()/pollFirst()/removeLast()/pollLast();
- 从队首 / 队尾获取元素但不删除：getFirst()/peekFirst()/getLast()/peekLast();
- 总是调用xxxFirst()/xxxLast() 以便与Queue的方法区分开；
- 避免把null添加到队列。



栈（Stack）是一种后进先出（LIFO：Last In First Out）的数据结构。

## 使用Stack

什么是LIFO呢？我们先回顾一下Queue的特点FIFO：

$$\begin{array}{ccccccc} \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} & \rightarrow & \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} & \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} & \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} & \rightarrow & \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} \end{array}$$

所谓FIFO，是最先进队列的元素一定最早出队列，而LIFO是最后进Stack的元素一定最早出Stack。如何做到这一点呢？只需要把队列的一端封死：

$$\begin{array}{ccccc} \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} & \leftarrow & \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} & \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} & \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} & \begin{array}{c} (\backslash \backslash \\ (= '. ')) \\ o( \_ " ) \end{array} \end{array}$$

因此，Stack是这样一种数据结构：只能不断地往Stack中压入（push）元素，最后进去的必须最早弹出（pop）来：



Stack只有入栈和出栈的操作：

- 把元素压栈：push(E)；
- 把栈顶的元素“弹出”：pop(E)；
- 取栈顶元素但不弹出：peek(E)。

在Java中，我们用Deque可以实现Stack的功能：

- 把元素压栈：push(E)/addFirst(E)；
- 把栈顶的元素“弹出”：pop(E)/removeFirst()；
- 取栈顶元素但不弹出：peek(E)/peekFirst()。

为什么Java的集合类没有单独的Stack接口呢？因为有个遗留类名字就叫Stack，出于兼容性考虑，所以没办法创建Stack接口，只能用Deque接口来“模拟”一个Stack了。

当我们把Deque作为Stack使用时，注意只调用push()/pop()/peek()方法，不要调用addFirst()/removeFirst()/peekFirst()方法，这样代码更加清晰。

### Stack的作用

Stack在计算机中使用非常广泛，JVM在处理Java方法调用的时候就会通过栈这种数据结构维护方法调用的层次。例如：

```
static void main(String[] args) {
```

```

    foo(123);
}

static String foo(x) {
    return "F-" + bar(x + 1);
}

static int bar(int x) {
    return x << 2;
}

```

**JVM**会创建方法调用栈，每调用一个方法时，先将参数压栈，然后执行对应的方法；当方法返回时，返回值压栈，调用方法通过出栈操作获得方法返回值。

因为方法调用栈有容量限制，嵌套调用过多会造成栈溢出，即引发`StackOverflowError`：

```

// 测试无限递归调用
----
public class Main {
    public static void main(String[] args) {
        increase(1);
    }

    static int increase(int x) {
        return increase(x) + 1;
    }
}

```

我们再来看一个`Stack`的用途：对整数进行进制的转换就可以利用栈。

例如，我们要把一个`int`整数12500转换为十六进制表示的字符串，如何实现这个功能？

首先我们准备一个空栈：



然后计算 $12500 \div 16 = 781 \dots 4$ ，余数是4，把余数4压栈：



然后计算 $781 \div 16 = 48 \dots 13$ ，余数是13，13的十六进制用字母D表示，把余数D压栈：



然后计算 $48 \div 16 = 3 \dots 0$ ，余数是0，把余数0压栈：

0
D
4

最后计算 $3 \div 16 = 0 \dots 3$ ，余数是3，把余数3压栈：

3
0
D
4

当商是0的时候，计算结束，我们把栈的所有元素依次弹出，组成字符串30D4，这就是十进制整数12500的十六进制表示的字符串。

## 计算中缀表达式

在编写程序的时候，我们使用的带括号的数学表达式实际上是中缀表达式，即运算符在中间，例如： $1 + 2 * (9 - 5)$ 。

但是计算机执行表达式的时候，它并不能直接计算中缀表达式，而是通过编译器把中缀表达式转换为后缀表达式，例如： $1\ 2\ 9\ 5\ -\ *\ +$ 。

这个编译过程就会用到栈。我们先跳过编译这一步（涉及运算优先级，代码比较复杂），看看如何通过栈计算后缀表达式。

计算后缀表达式不考虑优先级，直接从左到右依次计算，因此计算起来简单。首先准备一个空的栈：

--

然后我们依次扫描后缀表达式 $1\ 2\ 9\ 5\ -\ *\ +$ ，遇到数字1，就直接扔到栈里：

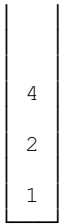
1
---

紧接着，遇到数字2，9，5，也扔到栈里：

5
9



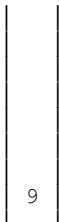
接下来遇到减号时，弹出栈顶的两个元素，并计算 $9-5=4$ ，把结果4压栈：



接下来遇到\*号时，弹出栈顶的两个元素，并计算 $2*4=8$ ，把结果8压栈：



接下来遇到+号时，弹出栈顶的两个元素，并计算 $1+8=9$ ，把结果9压栈：



扫描结束后，没有更多的计算了，弹出栈的唯一一个元素，得到计算结果9。

## 练习

请利用Stack把一个给定的整数转换为十六进制：

```
// 转十六进制
----
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String hex = toHex(12500);
        if (hex.equalsIgnoreCase("30D4")) {
            System.out.println("测试通过");
        } else {
            System.out.println("测试失败");
        }
    }

    static String toHex(int n) {
        return "";
    }
}
```

进阶练习：

请利用Stack把字符串中缀表达式编译为后缀表达式，然后再利用栈执行后缀表达式获得计算结果：

// 高难度练习，慎重选择！

```
-----
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String exp = "1 + 2 * (9 - 5)";
        SuffixExpression se = compile(exp);
        int result = se.execute();
        System.out.println(exp + " = " + result + " " + (result == 1 + 2 * (9 - 5) ? "✓" : "✗"));
    }

    static SuffixExpression compile(String exp) {
        // TODO:
        return new SuffixExpression();
    }
}

class SuffixExpression {
    int execute() {
        // TODO:
        return 0;
    }
}
```

进阶练习2:

请把带变量的中缀表达式编译为后缀表达式，执行后缀表达式时，传入变量的值并获得计算结果：

// 超高难度练习，慎重选择！

```
-----
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String exp = "x + 2 * (y - 5)";
        SuffixExpression se = compile(exp);
        Map<String, Integer> env = Map.of("x", 1, "y", 9);
        int result = se.execute(env);
        System.out.println(exp + " = " + result + " " + (result == 1 + 2 * (9 - 5) ? "✓" : "✗"));
    }

    static SuffixExpression compile(String exp) {
        // TODO:
        return new SuffixExpression();
    }
}

class SuffixExpression {
    int execute(Map<String, Integer> env) {
        // TODO:
        return 0;
    }
}
```

[Stack练习](#)

## 小结

栈（Stack）是一种后进先出（LIFO）的数据结构，操作栈的元素的方法有：

- 把元素压栈：push(E)；
- 把栈顶的元素“弹出”：pop(E)；
- 取栈顶元素但不弹出：peek(E)。

在Java中，我们用Deque可以实现Stack的功能，注意只调用push()/pop()/peek()方法，避免调用Deque的其他方法。

最后，不要使用遗留类Stack。

Java的集合类都可以使用for each循环，List、Set和Queue会迭代每个元素，Map会迭代每个key。以List为例：

## 使用Iterator

```
List<String> list = List.of("Apple", "Orange", "Pear");
for (String s : list) {
    System.out.println(s);
}
```

实际上，Java编译器并不知道如何遍历List。上述代码能够编译通过，只是因为编译器把for each循环通过Iterator改写为了普通的for循环：

```
for (Iterator<String> it = list.iterator(); it.hasNext(); ) {
    String s = it.next();
    System.out.println(s);
}
```

我们把这种通过Iterator对象遍历集合的模式称为迭代器。

使用迭代器的好处在于，调用方总是以统一的方式遍历各种集合类型，而不必关系它们内部的存储结构。

例如，我们虽然知道ArrayList在内部是以数组形式存储元素，并且，它还提供了get(int)方法。虽然我们可以用for循环遍历：

```
for (int i=0; i<list.size(); i++) {
    Object value = list.get(i);
}
```

但是这样一来，调用方就必须知道集合的内部存储结构。并且，如果把ArrayList换成LinkedList，get(int)方法耗时会随着index的增加而增加。如果把ArrayList换成Set，上述代码就无法编译，因为Set内部没有索引。

用Iterator遍历就没有上述问题，因为Iterator对象是集合对象自己在内部创建的，它自己知道如何高效遍历内部的数据集合，调用方则获得了统一的代码，编译器才能把标准的for each循环自动转换为Iterator遍历。

如果我们自己编写了一个集合类，想要使用for each循环，只需满足以下条件：

- 集合类实现Iterable接口，该接口要求返回一个Iterator对象；
- 用Iterator对象迭代集合内部数据。

这里的关键在于，集合类通过调用iterator()方法，返回一个Iterator对象，这个对象必须自己知道如何遍历该集合。

一个简单的Iterator示例如下，它总是以倒序遍历集合：

```
// Iterator
----
import java.util.*;

public class Main {
    public static void main(String[] args) {
        ReverseList<String> rlist = new ReverseList<>();
        rlist.add("Apple");
        rlist.add("Orange");
        rlist.add("Pear");
        for (String s : rlist) {
            System.out.println(s);
        }
    }
}

class ReverseList<T> implements Iterable<T> {

    List<T> list = new ArrayList<>();

    public void add(T t) {
```

```

        list.add(t);
    }

    @Override
    public Iterator<T> iterator() {
        return new ReverseIterator(list.size());
    }

    class ReverseIterator implements Iterator<T> {
        int index;

        ReverseIterator(int index) {
            this.index = index;
        }

        @Override
        public boolean hasNext() {
            return index > 0;
        }

        @Override
        public T next() {
            index--;
            return ReverseList.this.list.get(index);
        }
    }
}

```

虽然ReverseList和ReverseIterator的实现类稍微比较复杂，但是，注意到这是底层集合库，只需编写一次。而调用方则完全按for each循环编写代码，根本不需要知道集合内部的存储逻辑和遍历逻辑。

在编写Iterator的时候，我们通常可以用一个内部类来实现Iterator接口，这个内部类可以直接访问对应的外部类的所有字段和方法。例如，上述代码中，内部类ReverseIterator可以用ReverseList.this获得当前外部类的this引用，然后，通过这个this引用就可以访问ReverseList的所有字段和方法。

## 小结

Iterator是一种抽象的数据访问模型。使用Iterator模式进行迭代的好处有：

- 对任何集合都采用同一种访问模型；
- 调用者对集合内部结构一无所知；
- 集合类返回的Iterator对象知道如何迭代。

Java提供了标准的迭代器模型，即集合类实现java.util.Iterable接口，返回java.util.Iterator实例。



`Collections`是JDK提供的工具类，同样位于`java.util`包中。它提供了一系列静态方法，能更方便地操作各种集合。

## 使用Collections

注意`Collections`结尾多了一个s，不是`Collection`！

我们一般看方法名和参数就可以确认`Collections`提供的该方法的功能。例如，对于以下静态方法：

```
public static boolean addAll(Collection<? super T> c, T... elements) { ... }
```

`addAll()`方法可以给一个`Collection`类型的集合添加若干元素。因为方法签名是`Collection`，所以我们可以传入`List`，`Set`等各种集合类型。

### 创建空集合

`Collections`提供了一系列方法来创建空集合：

- 创建空**List**: `List<T> emptyList()`
- 创建空**Map**: `Map<K, V> emptyMap()`
- 创建空**Set**: `Set<T> emptySet()`

要注意到返回的空集合是不可变集合，无法向其中添加或删除元素。

此外，也可以用各个集合接口提供的`of(T...)`方法创建空集合。例如，以下创建空`List`的两个方法是等价的：

```
List<String> list1 = List.of();  
List<String> list2 = Collections.emptyList();
```

### 创建单元素集合

`Collections`提供了一系列方法来创建一个单元素集合：

- 创建一个元素的**List**: `List<T> singletonList(T o)`
- 创建一个元素的**Map**: `Map<K, V> singletonMap(K key, V value)`
- 创建一个元素的**Set**: `Set<T> singleton(T o)`

要注意到返回的单元素集合也是不可变集合，无法向其中添加或删除元素。

此外，也可以用各个集合接口提供的`of(T...)`方法创建单元素集合。例如，以下创建单元素`List`的两个方法是等价的：

```
List<String> list1 = List.of("apple");  
List<String> list2 = Collections.singleton("apple");
```

实际上，使用`List.of(T...)`更方便，因为它既可以创建空集合，也可以创建单元素集合，还可以创建任意个元素的集合：

```
List<String> list1 = List.of(); // empty list  
List<String> list2 = List.of("apple"); // 1 element  
List<String> list3 = List.of("apple", "pear"); // 2 elements  
List<String> list4 = List.of("apple", "pear", "orange"); // 3 elements
```

### 排序

`Collections`可以对`List`进行排序。因为排序会直接修改`List`元素的位置，因此必须传入可变`List`：

```
import java.util.*;  
----  
public class Main {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<>();  
        list.add("apple");  
    }  
}
```

```

        list.add("pear");
        list.add("orange");
        // 排序前:
        System.out.println(list);
        Collections.sort(list);
        // 排序后:
        System.out.println(list);
    }
}

```

## 洗牌

`Collections`提供了洗牌算法，即传入一个有序的`List`，可以随机打乱`List`内部元素的顺序，效果相当于让计算机洗牌：

```

import java.util.*;
-----
public class Main {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for (int i=0; i<10; i++) {
            list.add(i);
        }
        // 洗牌前:
        System.out.println(list);
        Collections.shuffle(list);
        // 洗牌后:
        System.out.println(list);
    }
}

```

## 不可变集合

`Collections`还提供了一组方法把可变集合封装成不可变集合：

- 封装成不可变**List**: `List<T> unmodifiableList(List<? extends T> list)`
- 封装成不可变**Set**: `Set<T> unmodifiableSet(Set<? extends T> set)`
- 封装成不可变**Map**: `Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)`

这种封装实际上是通过创建一个代理对象，拦截掉所有修改方法实现的。我们来看看效果：

```

import java.util.*;
-----
public class Main {
    public static void main(String[] args) {
        List<String> mutable = new ArrayList<>();
        mutable.add("apple");
        mutable.add("pear");
        // 变为不可变集合:
        List<String> immutable = Collections.unmodifiableList(mutable);
        immutable.add("orange"); // UnsupportedOperationException!
    }
}

```

然而，继续对原始的可变`List`进行增删是可以的，并且，会直接影响到封装后的“不可变”`List`：

```

import java.util.*;
-----
public class Main {
    public static void main(String[] args) {
        List<String> mutable = new ArrayList<>();
        mutable.add("apple");
        mutable.add("pear");
        // 变为不可变集合:
        List<String> immutable = Collections.unmodifiableList(mutable);
        mutable.add("orange");
        System.out.println(immutable);
    }
}

```

```
    }  
}
```

因此，如果我们希望把一个可变List封装成不可变List，那么，返回不可变List后，最好立刻扔掉可变List的引用，这样可以保证后续操作不会意外改变原始对象，从而造成“不可变”List变化了：

```
import java.util.*;  
-----  
public class Main {  
    public static void main(String[] args) {  
        List<String> mutable = new ArrayList<>();  
        mutable.add("apple");  
        mutable.add("pear");  
        // 变为不可变集合：  
        List<String> immutable = Collections.unmodifiableList(mutable);  
        // 立刻扔掉mutable的引用：  
        mutable = null;  
        System.out.println(immutable);  
    }  
}
```

## 线程安全集合

Collections还提供了一组方法，可以把线程不安全的集合变为线程安全的集合：

- 变为线程安全的List: `List<T> synchronizedList(List<T> list)`
- 变为线程安全的Set: `Set<T> synchronizedSet(Set<T> s)`
- 变为线程安全的Map: `Map<K, V> synchronizedMap(Map<K, V> m)`

多线程的概念我们会在后面讲。因为从Java 5开始，引入了更高效的并发集合类，所以上述这几个同步方法已经没有什么用了。

## 小结

Collections类提供了一组工具方法来方便使用集合类：

- 创建空集合；
- 创建单元素集合；
- 创建不可变集合；
- 排序 / 洗牌等操作。

IO是指Input/Output，即输入和输出。以内存为中心：

## IO

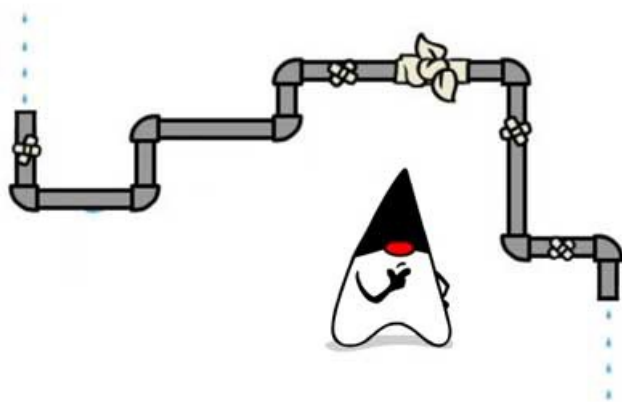
- **Input**指从外部读入数据到内存，例如，把文件从磁盘读取到内存，从网络读取数据到内存等等。
- **Output**指把数据从内存输出到外部，例如，把数据从内存写入到文件，把数据从内存输出到网络等等。

为什么要把数据读到内存才能处理这些数据？因为代码是在内存中运行的，数据也必须读到内存，最终的表示方式无非是byte数组，字符串等，都必须存放在内存里。

从Java代码来看，输入实际上就是从外部，例如，硬盘上的某个文件，把内容读到内存，并且以Java提供的某种数据类型表示，例如，byte[]，String，这样，后续代码才能处理这些数据。

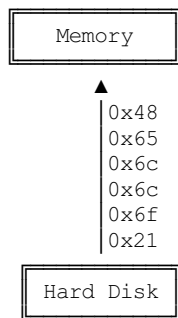
因为内存有“易失性”的特点，所以必须把处理后的数据以某种方式输出，例如，写入到文件。**Output**实际上就是把Java表示的数据格式，例如，byte[]，String等输出到某个地方。

IO流是一种顺序读写数据的模式，它的特点是单向流动。数据类似自来水一样在水管中流动，所以我们把它称为IO流。



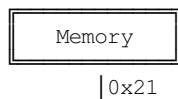
### InputStream / OutputStream

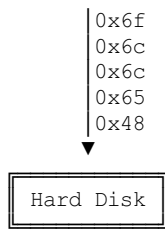
IO流以byte（字节）为最小单位，因此也称为字节流。例如，我们要从磁盘读入一个文件，包含6个字符，就相当于读入了6个字节的数据：



这6个字节是按顺序读入的，所以是输入字节流。

反过来，我们把6个字节从内存写入磁盘文件，就是输出字节流：





在Java中，InputStream代表输入字节流，OutputStream代表输出字节流，这是最基本的两种IO流。

## Reader / Writer

如果我们需要读写的是字符，并且字符不全是单字节表示的ASCII字符，那么，按照char来读写显然更方便，这种流称为字符流。

Java提供了Reader和Writer表示字符流，字符流传输的最小数据单位是char。

例如，我们把char[]数组Hi你好这4个字符用Writer字符流写入文件，并且使用UTF-8编码，得到的最终文件内容是8个字节，英文字符H和i各占一个字节，中文字符你好各占3个字节：

```
0x48
0x69
0xe4bda0
0xe5a5bd
```

反过来，我们用Reader读取以UTF-8编码的这8个字节，会从Reader中得到Hi你好这4个字符。

因此，Reader和Writer本质上是一个能自动编解码的InputStream和OutputStream。

使用Reader，数据源虽然是字节，但我们读入的数据都是char类型的字符，原因是Reader内部把读入的byte做了编码，转换成了char。使用InputStream，我们读入的数据和原始二进制数据一模一样，是byte[]数组，但是我们可以自己把二进制byte[]数组按照某种编码转换为字符串。究竟使用Reader还是InputStream，要取决于具体的使用场景。如果数据源不是文本，就只能使用InputStream，如果数据源是文本，使用Reader更方便一些。Writer和OutputStream是类似的。

## 同步和异步

同步IO是指，读写IO时代码必须等待数据返回后才继续执行后续代码，它的优点是代码编写简单，缺点是CPU执行效率低。

而异步IO是指，读写IO时仅发出请求，然后立刻执行后续代码，它的优点是CPU执行效率高，缺点是代码编写复杂。

Java标准库的包java.io提供了同步IO，而java.nio则是异步IO。上面我们讨论的InputStream、OutputStream、Reader和Writer都是同步IO的抽象类，对应的具体实现类，以文件为例，有FileInputStream、FileOutputStream、FileReader和FileWriter。

本节我们只讨论Java的同步IO，即输入/输出流的IO模型。

## 小结

IO流是一种流式的数据输入/输出模型：

- 二进制数据以byte为最小单位在InputStream/OutputStream中单向流动；
- 字符数据以char为最小单位在Reader/Writer中单向流动。

Java标准库的java.io包提供了同步IO功能：

- 字节流接口：InputStream/OutputStream；

- 字符流接口：Reader/Writer。

在计算机系统中，文件是非常重要的存储方式。**Java**的标准库`java.io`提供了`File`对象来操作文件和目录。

## File对象

要构造一个`File`对象，需要传入文件路径：

```
import java.io.*;
-----
public class Main {
    public static void main(String[] args) {
        File f = new File("C:\\Windows\\notepad.exe");
        System.out.println(f);
    }
}
```

构造`File`对象时，既可以传入绝对路径，也可以传入相对路径。绝对路径是以根目录开头的完整路径，例如：

```
File f = new File("C:\\Windows\\notepad.exe");
```

注意**Windows**平台使用`\`作为路径分隔符，在**Java**字符串中需要用`\\`表示一个`\`。**Linux**平台使用`/`作为路径分隔符：

```
File f = new File("/usr/bin/javac");
```

传入相对路径时，相对路径前面加上当前目录就是绝对路径：

```
// 假设当前目录是C:\Docs
File f1 = new File("sub\\javac"); // 绝对路径是C:\Docs\sub\javac
File f3 = new File(".\\sub\\javac"); // 绝对路径是C:\Docs\sub\javac
File f3 = new File("../sub\\javac"); // 绝对路径是C:\sub\javac
```

可以用`.`表示当前目录，`..`表示上级目录。

`File`对象有3种形式表示的路径，一种是`getPath()`，返回构造方法传入的路径，一种是`getAbsolutePath()`，返回绝对路径，一种是`getCanonicalPath()`，它和绝对路径类似，但是返回的是规范路径。

什么是规范路径？我们看以下代码：

```
import java.io.*;
-----
public class Main {
    public static void main(String[] args) throws IOException {
        File f = new File(".");
        System.out.println(f.getPath());
        System.out.println(f.getAbsolutePath());
        System.out.println(f.getCanonicalPath());
    }
}
```

绝对路径可以表示成`C:\Windows\System32\..\notepad.exe`，而规范路径就是把`.`和`..`转换成标准的绝对路径后的路径：`C:\Windows\notepad.exe`。

因为**Windows**和**Linux**的路径分隔符不同，`File`对象有一个静态变量用于表示当前平台的系统分隔符：

```
System.out.println(File.separator); // 根据当前平台打印"\或"/"
```

## 文件和目录

`File`对象既可以表示文件，也可以表示目录。特别要注意的是，构造一个`File`对象，即使传入的文件或目录不存在，代码也不会出错，因为构造一个`File`对象，并不会导致任何磁盘操作。只有当我们调用`File`对象的某些方法的时候，才真正进行磁盘操作。

例如，调用`isFile()`，判断该`File`对象是否是一个已存在的文件，调用`isDirectory()`，判断该`File`对象是否是一个已存在的目录：

```
import java.io.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        File f1 = new File("C:\\Windows");
        File f2 = new File("C:\\Windows\\notepad.exe");
        File f3 = new File("C:\\Windows\\nothing");
        System.out.println(f1.isFile());
        System.out.println(f1.isDirectory());
        System.out.println(f2.isFile());
        System.out.println(f2.isDirectory());
        System.out.println(f3.isFile());
        System.out.println(f3.isDirectory());
    }
}
```

用File对象获取到一个文件时，还可以进一步判断文件的权限和大小：

- boolean canRead(): 是否可读；
- boolean canWrite(): 是否可写；
- boolean canExecute(): 是否可执行；
- long length(): 文件字节大小。

对目录而言，是否可执行表示能否列出它包含的文件和子目录。

## 创建和删除文件

当File对象表示一个文件时，可以通过createNewFile()创建一个新文件，用delete()删除该文件：

```
File file = new File("/path/to/file");
if (file.createNewFile()) {
    // 文件创建成功:
    // TODO:
    if (file.delete()) {
        // 删除文件成功:
    }
}
```

有些时候，程序需要读写一些临时文件，File对象提供了createTempFile()来创建一个临时文件，以及deleteOnExit()在JVM退出时自动删除该文件。

```
import java.io.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        File f = File.createTempFile("tmp-", ".txt"); // 提供临时文件的前缀和后缀
        f.deleteOnExit(); // JVM退出时自动删除
        System.out.println(f.isFile());
        System.out.println(f.getAbsolutePath());
    }
}
```

## 遍历文件和目录

当File对象表示一个目录时，可以使用list()和listFiles()列出目录下的文件和子目录名。listFiles()提供了一系列重载方法，可以过滤不想要的文件和目录：

```
import java.io.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        File f = new File("C:\\Windows");
        File[] fs1 = f.listFiles(); // 列出所有文件和子目录
        printFiles(fs1);
        File[] fs2 = f.listFiles(new FilenameFilter() { // 仅列出.exe文件
            public boolean accept(File dir, String name) {
                return name.endsWith(".exe"); // 返回true表示接受该文件
            }
        });
    }
}
```



```

    }
    });
    printFiles(fs2);
}

static void printFiles(File[] files) {
    System.out.println("=====");
    if (files != null) {
        for (File f : files) {
            System.out.println(f);
        }
    }
    System.out.println("=====");
}
}

```

和文件操作类似，**File**对象如果表示一个目录，可以通过以下方法创建和删除目录：

- `boolean mkdir()`：创建当前**File**对象表示的目录；
- `boolean mkdirs()`：创建当前**File**对象表示的目录，并在必要时将不存在的父目录也创建出来；
- `boolean delete()`：删除当前**File**对象表示的目录，当前目录必须为空才能删除成功。

## Path

Java标准库还提供了一个Path对象，它位于`java.nio.file`包。Path对象和File对象类似，但操作更加简单：

```

import java.io.*;
import java.nio.file.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        Path p1 = Paths.get(".", "project", "study"); // 构造一个Path对象
        System.out.println(p1);
        Path p2 = p1.toAbsolutePath(); // 转换为绝对路径
        System.out.println(p2);
        Path p3 = p2.normalize(); // 转换为规范路径
        System.out.println(p3);
        File f = p3.toFile(); // 转换为File对象
        System.out.println(f);
        for (Path p : Paths.get("..").toAbsolutePath()) { // 可以直接遍历Path
            System.out.println("  " + p);
        }
    }
}

```

如果需要对目录进行复杂的拼接、遍历等操作，使用Path对象更方便。

## 练习

请利用File对象列出指定目录下的所有子目录和文件，并按层次打印。

例如，输出：

```

Documents/
  word/
    1.docx
    2.docx
  work/
    abc.doc
  ppt/
  other/

```

如果不指定参数，则使用当前目录，如果指定参数，则使用指定目录。

[io-file](#)

## 小结

Java标准库的`java.io.File`对象表示一个文件或者目录：

- 创建`File`对象本身不涉及IO操作；
- 可以获取路径 / 绝对路径 / 规范路径：`getPath()/getAbsolutePath()/getCanonicalPath()`；
- 可以获取目录的文件和子目录：`list()/listFiles()`；
- 可以创建或删除文件和目录。

InputStream就是Java标准库提供的最基本的输入流。它位于java.io这个包里。java.io包提供了所有同步IO的功能。

# InputStream

要特别注意的一点是，InputStream并不是一个接口，而是一个抽象类，它是所有输入流的超类。这个抽象类定义的一个最重要的方法就是int read()，签名如下：

```
public abstract int read() throws IOException;
```

这个方法会读取输入流的下一个字节，并返回字节表示的int值（0~255）。如果已读到末尾，返回-1表示不能继续读取了。

FileInputStream是InputStream的一个子类。顾名思义，FileInputStream就是从文件流中读取数据。下面的代码演示了如何完整地读取一个FileInputStream的所有字节：

```
public void readFile() throws IOException {
    // 创建一个FileInputStream对象：
    InputStream input = new FileInputStream("src/readme.txt");
    for (;;) {
        int n = input.read(); // 反复调用read()方法，直到返回-1
        if (n == -1) {
            break;
        }
        System.out.println(n); // 打印byte的值
    }
    input.close(); // 关闭流
}
```

在计算机中，类似文件、网络端口这些资源，都是由操作系统统一管理的。应用程序在运行的过程中，如果打开了一个文件进行读写，完成后要及时地关闭，以便让操作系统把资源释放掉，否则，应用程序占用的资源会越来越多，不但白白占用内存，还会影响其他应用程序的运行。

InputStream和OutputStream都是通过close()方法来关闭流。关闭流就会释放对应的底层资源。

我们还要注意在读取或写入IO流的过程中，可能会发生错误，例如，文件不存在导致无法读取，没有写权限导致写入失败，等等，这些底层错误由Java虚拟机自动封装成IOException异常并抛出。因此，所有与IO操作相关的代码都必须正确处理IOException。

仔细观察上面的代码，会发现一个潜在的问题：如果读取过程中发生了IO错误，InputStream就没法正确地关闭，资源也就没法及时释放。

因此，我们需要用try ... finally来保证InputStream在无论是否发生IO错误的时候都能够正确地关闭：

```
public void readFile() throws IOException {
    InputStream input = null;
    try {
        input = new FileInputStream("src/readme.txt");
        int n;
        while ((n = input.read()) != -1) { // 利用while同时读取并判断
            System.out.println(n);
        }
    } finally {
        if (input != null) { input.close(); }
    }
}
```

用try ... finally来编写上述代码会感觉比较复杂，更好的写法是利用Java 7引入的新的try(resource)的语法，只需要编写try语句，让编译器自动为我们关闭资源。推荐的写法如下：

```
public void readFile() throws IOException {
    try (InputStream input = new FileInputStream("src/readme.txt")) {
        int n;
        while ((n = input.read()) != -1) {
            System.out.println(n);
        }
    }
}
```

```

    }
} // 编译器在此自动为我们写入finally并调用close()
}

```

实际上，编译器并不会特别地为InputStream加上自动关闭。编译器只看try(resource = ...)中的对象是否实现了java.lang.AutoCloseable接口，如果实现了，就自动加上finally语句并调用close()方法。InputStream和OutputStream都实现了这个接口，因此，都可以用在try(resource)中。

## 缓冲

在读取流的时候，一次读取一个字节并不是最高效的方法。很多流支持一次性读取多个字节到缓冲区，对于文件和网络流来说，利用缓冲区一次性读取多个字节效率往往要高很多。InputStream提供了两个重载方法来支持读取多个字节：

- int read(byte[] b)：读取若干字节并填充到byte[]数组，返回读取的字节数
- int read(byte[] b, int off, int len)：指定byte[]数组的偏移量和最大填充数

利用上述方法一次读取多个字节时，需要先定义一个byte[]数组作为缓冲区，read()方法会尽可能多地读取字节到缓冲区，但不会超过缓冲区的大小。read()方法的返回值不再是字节的int值，而是返回实际读取了多少个字节。如果返回-1，表示没有更多的数据了。

利用缓冲区一次读取多个字节的代码如下：

```

public void readFile() throws IOException {
    try (InputStream input = new FileInputStream("src/readme.txt")) {
        // 定义1000个字节的缓冲区：
        byte[] buffer = new byte[1000];
        int n;
        while ((n = input.read(buffer)) != -1) { // 读取到缓冲区
            System.out.println("read " + n + " bytes.");
        }
    }
}

```

## 阻塞

在调用InputStream的read()方法读取数据时，我们说read()方法是阻塞（Blocking）的。它的意思是，对于下面的代码：

```

int n;
n = input.read(); // 必须等待read()方法返回才能执行下一行代码
int m = n;

```

执行到第二行代码时，必须等read()方法返回后才能继续。因为读取IO流相比执行普通代码，速度会慢很多，因此，无法确定read()方法调用到底要花费多长时间。

## InputStream实现类

用FileInputStream可以从文件获取输入流，这是InputStream常用的一个实现类。此外，ByteArrayInputStream可以在内存中模拟一个InputStream：

```

import java.io.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data = { 72, 101, 108, 108, 111, 33 };
        try (InputStream input = new ByteArrayInputStream(data)) {
            int n;
            while ((n = input.read()) != -1) {
                System.out.println((char)n);
            }
        }
    }
}

```

ByteArrayInputStream实际上是把一个byte[]数组在内存中变成一个InputStream，虽然实际应用不多，但测试的时候，可以用它来构造一个InputStream。

举个栗子：我们想从文件中读取所有字节，并转换成char然后拼成一个字符串，可以这么写：

```
public class Main {
    public static void main(String[] args) throws IOException {
        String s;
        try (InputStream input = new FileInputStream("C:\\test\\README.txt")) {
            int n;
            StringBuilder sb = new StringBuilder();
            while ((n = input.read()) != -1) {
                sb.append((char) n);
            }
            s = sb.toString();
        }
        System.out.println(s);
    }
}
```

要测试上面的程序，就真的需要在本地硬盘上放一个真实的文本文件。如果我们把代码稍微改造一下，提取一个readAsString()的方法：

```
public class Main {
    public static void main(String[] args) throws IOException {
        String s;
        try (InputStream input = new FileInputStream("C:\\test\\README.txt")) {
            s = readAsString(input);
        }
        System.out.println(s);
    }

    public static String readAsString(InputStream input) throws IOException {
        int n;
        StringBuilder sb = new StringBuilder();
        while ((n = input.read()) != -1) {
            sb.append((char) n);
        }
        return sb.toString();
    }
}
```

对这个String readAsString(InputStream input)方法进行测试就相当简单，因为不一定要传入一个真的FileInputStream：

```
import java.io.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data = { 72, 101, 108, 108, 111, 33 };
        try (InputStream input = new ByteArrayInputStream(data)) {
            String s = readAsString(input);
            System.out.println(s);
        }
    }

    public static String readAsString(InputStream input) throws IOException {
        int n;
        StringBuilder sb = new StringBuilder();
        while ((n = input.read()) != -1) {
            sb.append((char) n);
        }
        return sb.toString();
    }
}
```

这就是面向抽象编程原则的应用：接受InputStream抽象类型，而不是具体的FileInputStream类型，从而使得代码可以处理InputStream的任意实现类。

## 小结

Java标准库的`java.io.InputStream`定义了所有输入流的超类:

- `FileInputStream`实现了文件流输入;
- `ByteArrayInputStream`在内存中模拟一个字节流输入。

总是使用`try(resource)`来保证`InputStream`正确关闭。

和InputStream相反，OutputStream是Java标准库提供的最基本的输出流。

## OutputStream

和InputStream类似，OutputStream也是抽象类，它是所有输出流的超类。这个抽象类定义的一个最重要的方法就是void write(int b)，签名如下：

```
public abstract void write(int b) throws IOException;
```

这个方法会写入一个字节到输出流。要注意的是，虽然传入的是int参数，但只会写入一个字节，即只写入int最低8位表示字节的部分（相当于b & 0xff）。

和InputStream类似，OutputStream也提供了close()方法关闭输出流，以便释放系统资源。要特别注意：OutputStream还提供了一个flush()方法，它的目的是将缓冲区的内容真正输出到目的地。

为什么要有flush()？因为向磁盘、网络写入数据的时候，出于效率的考虑，操作系统并不是输出一个字节就立刻写入到文件或者发送到网络，而是把输出的字节先放到内存的一个缓冲区里（本质上就是一个byte[]数组），等到缓冲区写满了，再一次性写入文件或者网络。对于很多IO设备来说，一次写一个字节和一次写1000个字节，花费的时间几乎是完全一样的，所以OutputStream有个flush()方法，能强制把缓冲区内容输出。

通常情况下，我们不需要调用这个flush()方法，因为缓冲区写满了OutputStream会自动调用它，并且，在调用close()方法关闭OutputStream之前，也会自动调用flush()方法。

但是，在某些情况下，我们必须手动调用flush()方法。举个栗子：

小明正在开发一款在线聊天软件，当用户输入一句话后，就通过OutputStream的write()方法写入网络流。小明测试的时候发现，发送方输入后，接收方根本收不到任何信息，怎么肥四？

原因就在于写入网络流是先写入内存缓冲区，等缓冲区满了才会一次性发送到网络。如果缓冲区大小是4K，则发送方要敲几千个字符后，操作系统才会把缓冲区的内容发送出去，这个时候，接收方会一次性收到大量消息。

解决办法就是每输入一句话后，立刻调用flush()，不管当前缓冲区是否已满，强迫操作系统把缓冲区的内容立刻发送出去。

实际上，InputStream也有缓冲区。例如，从FileInputStream读取一个字节时，操作系统往往会一次性读取若干字节到缓冲区，并维护一个指针指向未读的缓冲区。然后，每次我们调用int read()读取下一个字节时，可以直接返回缓冲区的下一个字节，避免每次读一个字节都导致IO操作。当缓冲区全部读完后继续调用read()，则会触发操作系统的下一次读取并再次填满缓冲区。

## FileOutputStream

我们以FileOutputStream为例，演示如何将若干个字节写入文件流：

```
public void writeFile() throws IOException {
    OutputStream output = new FileOutputStream("out/readme.txt");
    output.write(72); // H
    output.write(101); // e
    output.write(108); // l
    output.write(108); // l
    output.write(111); // o
    output.close();
}
```

每次写入一个字节非常麻烦，更常见的方法是一次性写入若干字节。这时，可以用OutputStream提供的重载方法void write(byte[])来实现：

```
public void writeFile() throws IOException {
    OutputStream output = new FileOutputStream("out/readme.txt");
    output.write("Hello".getBytes("UTF-8")); // Hello
    output.close();
}
```

和InputStream一样，上述代码没有考虑到在发生异常的情况下如何正确地关闭资源。写入过程也会经常发生IO错误，例如，磁盘已满，无权限写入等等。我们需要用try(resource)来保证OutputStream在无论是否发生IO错误的时候都能够正确地关闭：

```
public void writeFile() throws IOException {
    try (OutputStream output = new FileOutputStream("out/readme.txt")) {
        output.write("Hello".getBytes("UTF-8")); // Hello
    } // 编译器在此自动为我们写入finally并调用close()
}
```

## 阻塞

和InputStream一样，OutputStream的write()方法也是阻塞的。

## OutputStream实现类

用FileOutputStream可以从文件获取输出流，这是OutputStream常用的一个实现类。此外，ByteArrayOutputStream可以在内存中模拟一个OutputStream：

```
import java.io.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data;
        try (ByteArrayOutputStream output = new ByteArrayOutputStream()) {
            output.write("Hello ".getBytes("UTF-8"));
            output.write("world!".getBytes("UTF-8"));
            data = output.toByteArray();
        }
        System.out.println(new String(data, "UTF-8"));
    }
}
```

ByteArrayOutputStream实际上是把一个byte[]数组在内存中变成一个OutputStream，虽然实际应用不多，但测试的时候，可以用它来构造一个OutputStream。

## 练习

请利用InputStream和OutputStream，编写一个复制文件的程序，它可以带参数运行：

```
java CopyFile.java source.txt copy.txt
```

### [CopyFile练习](#)

## 小结

Java标准库的java.io.OutputStream定义了所有输出流的超类：

- FileOutputStream实现了文件流输出；
- ByteArrayOutputStream在内存中模拟一个字节流输出。

某些情况下需要手动调用OutputStream的flush()方法来强制输出缓冲区。

总是使用try(resource)来保证OutputStream正确关闭。



Java的IO标准库提供的InputStream根据来源可以包括：

## Filter模式

- FileInputStream：从文件读取数据，是最终数据源；
- ServletInputStream：从HTTP请求读取数据，是最终数据源；
- Socket.getInputStream()：从TCP连接读取数据，是最终数据源；
- ...

如果我们要给FileInputStream添加缓冲功能，则可以从FileInputStream派生一个类：

```
BufferedFileInputStream extends FileInputStream
```

如果要给FileInputStream添加计算签名的功能，类似的，也可以从FileInputStream派生一个类：

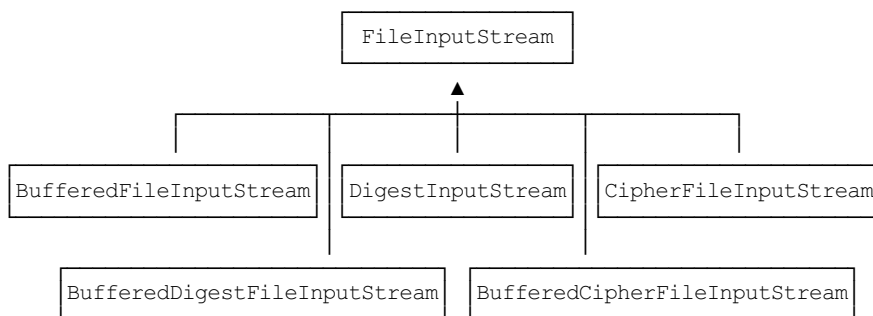
```
DigestFileInputStream extends FileInputStream
```

如果要给FileInputStream添加加密/解密功能，还是可以从FileInputStream派生一个类：

```
CipherFileInputStream extends FileInputStream
```

如果要给FileInputStream添加缓冲和签名的功能，那么我们还需要派生BufferedDigestFileInputStream。如果要给FileInputStream添加缓冲和加解密的功能，则需要派生BufferedCipherFileInputStream。

我们发现，给FileInputStream添加3种功能，至少需要3个子类。这3种功能的组合，又需要更多的子类：



这还只是针对FileInputStream设计，如果针对另一种InputStream设计，很快会出现子类爆炸的情况。

因此，直接使用继承，为各种InputStream附加更多的功能，根本无法控制代码的复杂度，很快就会失控。

为了解决依赖继承会导致子类数量失控的问题，JDK首先将InputStream分为两大类：

一类是直接提供数据的基础InputStream，例如：

- FileInputStream
- ByteArrayInputStream
- ServletInputStream
- ...

一类是提供额外附加功能的InputStream，例如：

- BufferedInputStream
- DigestInputStream
- CipherInputStream
- ...

当我们需要给一个“基础”InputStream附加各种功能时，我们先确定这个能提供数据源的InputStream，因为我们需要的数据总得来自某个地方，例如，FileInputStream，数据来自文件：

```
InputStream file = new FileInputStream("test.gz");
```

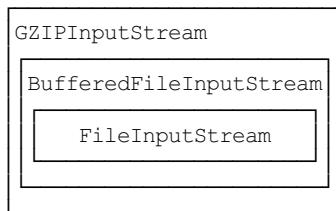
紧接着，我们希望`FileInputStream`能提供缓冲的功能来提高读取的效率，因此我们用`BufferedInputStream`包装这个`InputStream`，得到的包装类型是`BufferedInputStream`，但它仍然被视为一个`InputStream`：

```
InputStream buffered = new BufferedInputStream(file);
```

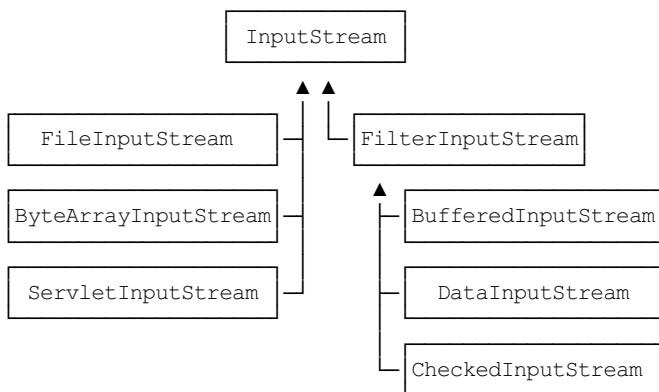
最后，假设该文件已经用`gzip`压缩了，我们希望直接读取解压缩的内容，就可以再包装一个`GZIPInputStream`：

```
InputStream gzip = new GZIPInputStream(buffered);
```

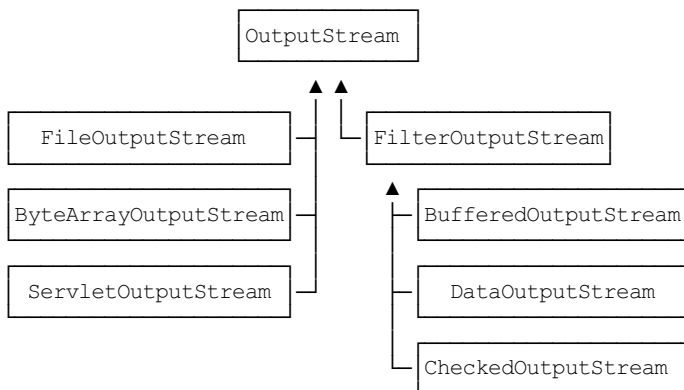
无论我们包装多少次，得到的对象始终是`InputStream`，我们直接用`InputStream`来引用它，就可以正常读取：



上述这种通过一个“基础”组件再叠加各种“附加”功能组件的模式，称之为**Filter模式**（或者装饰器模式：**Decorator**）。它可以让我们通过少量的类来实现各种功能的组合：



类似的，`OutputStream`也是以这种模式来提供各种功能：



## 编写FilterInputStream

我们也可以自己编写`FilterInputStream`，以便可以把自己的`FilterInputStream`“叠加”到任何一个`InputStream`中。

下面的例子演示了如何编写一个`CountInputStream`，它的作用是对输入的字节进行计数：

```

import java.io.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data = "hello, world!".getBytes("UTF-8");
        try (CountInputStream input = new CountInputStream(new ByteArrayInputStream(data))) {
            int n;
            while ((n = input.read()) != -1) {
                System.out.println((char)n);
            }
            System.out.println("Total read " + input.getBytesRead() + " bytes");
        }
    }
}

class CountInputStream extends FilterInputStream {
    private int count = 0;

    CountInputStream(InputStream in) {
        super(in);
    }

    public int getBytesRead() {
        return this.count;
    }

    public int read() throws IOException {
        int n = in.read();
        if (n != -1) {
            this.count++;
        }
        return n;
    }

    public int read(byte[] b, int off, int len) throws IOException {
        int n = in.read(b, off, len);
        this.count += n;
        return n;
    }
}

```

注意到在叠加多个FilterInputStream，我们只需要持有最外层的InputStream，并且，当最外层的InputStream关闭时（在try(resource)块的结束处自动关闭），内层的InputStream的close()方法也会被自动调用，并最终调用到最核心的“基础”InputStream，因此不存在资源泄露。

## 小结

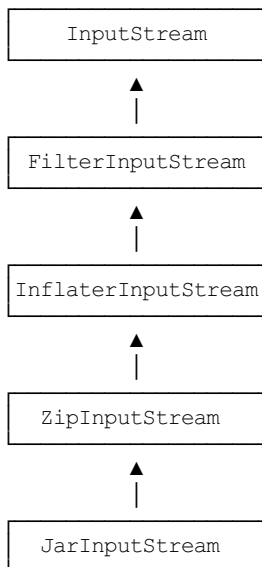
Java的IO标准库使用Filter模式为InputStream和OutputStream增加功能：

- 可以把一个InputStream和任意个FilterInputStream组合；
- 可以把一个OutputStream和任意个FilterOutputStream组合。

Filter模式可以在运行期动态增加功能（又称Decorator模式）。

ZipInputStream是一种FilterInputStream，它可以直接读取zip包的内容：

## 操作Zip



另一个JarInputStream是从ZipInputStream派生，它增加的主要功能是直接读取jar文件里面的MANIFEST.MF文件。因为本质上jar包就是zip包，只是额外附加了一些固定的描述文件。

### 读取zip包

我们来看看ZipInputStream的基本用法。

我们要创建一个ZipInputStream，通常是传入一个FileInputStream作为数据源，然后，循环调用getNextEntry()，直到返回null，表示zip流结束。

一个ZipEntry表示一个压缩文件或目录，如果是压缩文件，我们就用read()方法不断读取，直到返回-1：

```
try (ZipInputStream zip = new ZipInputStream(new FileInputStream(...))) {
    ZipEntry entry = null;
    while ((entry = zip.getNextEntry()) != null) {
        String name = entry.getName();
        if (!entry.isDirectory()) {
            int n;
            while ((n = zip.read()) != -1) {
                ...
            }
        }
    }
}
```

### 写入zip包

ZipOutputStream是一种FilterOutputStream，它可以直接写入内容到zip包。我们要先创建一个ZipOutputStream，通常是包装一个FileOutputStream，然后，每写入一个文件前，先调用putNextEntry()，然后用write()写入byte[]数据，写入完毕后调用closeEntry()结束这个文件的打包。

```
try (ZipOutputStream zip = new ZipOutputStream(new FileOutputStream(...))) {
    File[] files = ...
    for (File file : files) {
        zip.putNextEntry(new ZipEntry(file.getName()));
        zip.write(getFileDataAsBytes(file));
        zip.closeEntry();
    }
}
```

```
    }  
}
```

上面的代码没有考虑文件的目录结构。如果要实现目录层次结构，`new ZipEntry(name)`传入的`name`要用相对路径。

## 小结

`ZipInputStream`可以读取zip格式的流，`ZipOutputStream`可以把多份数据写入zip包；

配合`FileInputStream`和`FileOutputStream`就可以读写zip文件。

很多Java程序启动的时候，都需要读取配置文件。例如，从一个.properties文件中读取配置：

## 读取classpath资源

```
String conf = "C:\\conf\\default.properties";
try (InputStream input = new FileInputStream(conf)) {
    // TODO:
}
```

这段代码要正常执行，必须在C盘创建conf目录，然后在目录里创建default.properties文件。但是，在Linux系统上，路径和Windows的又不一样。

因此，从磁盘的固定目录读取配置文件，不是一个好的办法。

有没有路径无关的读取文件的方式呢？

我们知道，Java存放.class的目录或jar包也可以包含任意其他类型的文件，例如：

- 配置文件，例如.properties；
- 图片文件，例如.jpg；
- 文本文件，例如.txt，.csv；
- .....

从classpath读取文件就可以避免不同环境下文件路径不一致的问题：如果我们把default.properties文件放到classpath中，就不用关心它的实际存放路径。

在classpath中的资源文件，路径总是以/开头，我们先获取当前的Class对象，然后调用getResourceAsStream()就可以直接从classpath读取任意的资源文件：

```
try (InputStream input = getClass().getResourceAsStream("/default.properties")) {
    // TODO:
}
```

调用getResourceAsStream()需要特别注意的一点是，如果资源文件不存在，它将返回null。因此，我们需要检查返回的InputStream是否为null，如果为null，表示资源文件在classpath中没有找到：

```
try (InputStream input = getClass().getResourceAsStream("/default.properties")) {
    if (input != null) {
        // TODO:
    }
}
```

如果我们把默认的配置放到jar包中，再从外部文件系统读取一个可选的配置文件，就可以做到既有默认的配置，又可以让用户自己修改配置：

```
Properties props = new Properties();
props.load(inputStreamFromClassPath("/default.properties"));
props.load(inputStreamFromFile("./conf.properties"));
```

这样读取配置文件，应用程序启动就更加灵活。

### 小结

把资源存储在classpath中可以避免文件路径依赖；

Class对象的getResourceAsStream()可以从classpath中读取指定资源；

根据classpath读取资源时，需要检查返回的InputStream是否为null。

序列化是指把一个Java对象变成二进制内容，本质上就是一个byte[]数组。

## 序列化

为什么要把Java对象序列化呢？因为序列化后可以把byte[]保存到文件中，或者把byte[]通过网络传输到远程，这样，就相当于把Java对象存储到文件或者通过网络传输出去了。

有序列化，就有反序列化，即把一个二进制内容（也就是byte[]数组）变回Java对象。有了反序列化，保存到文件中的byte[]数组又可以“变回”Java对象，或者从网络上读取byte[]并把它“变回”Java对象。

我们来看看如何把一个Java对象序列化。

一个Java对象要能序列化，必须实现一个特殊的java.io.Serializable接口，它的定义如下：

```
public interface Serializable {  
}
```

Serializable接口没有定义任何方法，它是一个空接口。我们把这样的空接口称为“标记接口”（Marker Interface），实现了标记接口的类仅仅是给自身贴了个“标记”，并没有增加任何方法。

### 序列化

把一个Java对象变为byte[]数组，需要使用ObjectOutputStream。它负责把一个Java对象写入一个字节流：

```
import java.io.*;  
import java.util.Arrays;  
----  
public class Main {  
    public static void main(String[] args) throws IOException {  
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();  
        try (ObjectOutputStream output = new ObjectOutputStream(buffer)) {  
            // 写入int:  
            output.writeInt(12345);  
            // 写入String:  
            output.writeUTF("Hello");  
            // 写入Object:  
            output.writeObject(Double.valueOf(123.456));  
        }  
        System.out.println(Arrays.toString(buffer.toByteArray()));  
    }  
}
```

ObjectOutputStream既可以写入基本类型，如int，boolean，也可以写入String（以UTF-8编码），还可以写入实现了Serializable接口的Object。

因为写入Object时需要大量的类型信息，所以写入的内容很大。

### 反序列化

和ObjectOutputStream相反，ObjectInputStream负责从一个字节流读取Java对象：

```
try (ObjectInputStream input = new ObjectInputStream(...)) {  
    int n = input.readInt();  
    String s = input.readUTF();  
    Double d = (Double) input.readObject();  
}
```

除了能读取基本类型和String类型外，调用readObject()可以直接返回一个Object对象。要把它变成一个特定类型，必须强制转型。

readObject()可能抛出的异常有：

- ClassNotFoundException：没有找到对应的Class；

- `InvalidClassException`: `Class`不匹配。

对于`ClassNotFoundException`，这种情况常见于一台电脑上的Java程序把一个Java对象，例如，`Person`对象序列化以后，通过网络传给另一台电脑上的另一个Java程序，但是这台电脑的Java程序并没有定义`Person`类，所以无法反序列化。

对于`InvalidClassException`，这种情况常见于序列化的`Person`对象定义了一个`int`类型的`age`字段，但是反序列化时，`Person`类定义的`age`字段被改成了`long`类型，所以导致`class`不兼容。

为了避免这种`class`定义变动导致的不兼容，Java的序列化允许`class`定义一个特殊的`serialVersionUID`静态变量，用于标识Java类的序列化“版本”，通常可以由IDE自动生成。如果增加或修改了字段，可以改变`serialVersionUID`的值，这样就能自动阻止不匹配的`class`版本：

```
public class Person implements Serializable {  
    private static final long serialVersionUID = 2709425275741743919L;  
}
```

要特别注意反序列化的几个重要特点：

反序列化时，由JVM直接构造出Java对象，不调用构造方法，构造方法内部的代码，在反序列化时根本不可能执行。

## 安全性

因为Java的序列化机制可以导致一个实例能直接从`byte[]`数组创建，而不经构造方法，因此，它存在一定的安全隐患。一个精心构造的`byte[]`数组被反序列化后可以执行特定的Java代码，从而导致严重的安全漏洞。

实际上，Java本身提供的基于对象的序列化和反序列化机制既存在安全性问题，也存在兼容性问题。更好的序列化方法是通过JSON这样的通用数据结构来实现，只输出基本类型（包括`String`）的内容，而不存储任何与代码相关的信息。

## 小结

可序列化的Java对象必须实现`java.io.Serializable`接口，类似`Serializable`这样的空接口被称为“标记接口”（**Marker Interface**）；

反序列化时不调用构造方法，可设置`serialVersionUID`作为版本号（非必需）；

Java的序列化机制仅适用于Java，如果需要与其它语言交换数据，必须使用通用的序列化方法，例如JSON。



Reader是Java的IO库提供的另一个输入流接口。和InputStream的区别是，InputStream是一个字节流，即以byte为单位读取，而Reader是一个字符流，即以char为单位读取：

# Reader

## InputStream

字节流，以byte为单位

读取字节（-1，0~255）：int read()

读到字节数组：int read(byte[] b)

## Reader

字符流，以char为单位

读取字符（-1，0~65535）：int read()

读到字符数组：int read(char[] c)

java.io.Reader是所有字符输入流的超类，它最主要的方法是：

```
public int read() throws IOException;
```

这个方法读取字符流的下一个字符，并返回字符表示的int，范围是0~65535。如果已读到末尾，返回-1。

## FileReader

FileReader是Reader的一个子类，它可以打开文件并获取Reader。下面的代码演示了如何完整地读取一个FileReader的所有字符：

```
public void readFile() throws IOException {
    // 创建一个FileReader对象：
    Reader reader = new FileReader("src/readme.txt"); // 字符编码是???
    for (;;) {
        int n = reader.read(); // 反复调用read()方法，直到返回-1
        if (n == -1) {
            break;
        }
        System.out.println((char)n); // 打印char
    }
    reader.close(); // 关闭流
}
```

如果我们读取一个纯ASCII编码的文本文件，上述代码工作是没有问题的。但如果文件中包含中文，就会出现乱码，因为FileReader默认的编码与系统相关，例如，Windows系统的默认编码可能是GBK，打开一个UTF-8编码的文本文件就会出现乱码。

要避免乱码问题，我们需要在创建FileReader时指定编码：

```
Reader reader = new FileReader("src/readme.txt", StandardCharsets.UTF_8);
```

和InputStream类似，Reader也是一种资源，需要保证出错的时候也能正确关闭，所以我们需要用try (resource)来保证Reader在无论有没有IO错误的时候都能够正确地关闭：

```
try (Reader reader = new FileReader("src/readme.txt", StandardCharsets.UTF_8)) {
    // TODO
}
```

Reader还提供了一次性读取若干字符并填充到char[]数组的方法：

```
public int read(char[] c) throws IOException
```

它返回实际读入的字符个数，最大不超过char[]数组的长度。返回-1表示流结束。

利用这个方法，我们可以先设置一个缓冲区，然后，每次尽可能地填充缓冲区：

```
public void readFile() throws IOException {
    try (Reader reader = new FileReader("src/readme.txt", StandardCharsets.UTF_8)) {
        char[] buffer = new char[1000];
        int n;
        while ((n = reader.read(buffer)) != -1) {

```

```

        System.out.println("read " + n + " chars.");
    }
}

```

## CharArrayReader

CharArrayReader可以在内存中模拟一个Reader，它的作用实际上是把一个char[]数组变成一个Reader，这和ByteArrayInputStream非常类似：

```

try (Reader reader = new CharArrayReader("Hello".toCharArray())) {
}

```

## StringReader

StringReader可以直接把String作为数据源，它和CharArrayReader几乎一样：

```

try (Reader reader = new StringReader("Hello")) {
}

```

## InputStreamReader

Reader和InputStream有什么关系？

除了特殊的CharArrayReader和StringReader，普通的Reader实际上是基于InputStream构造的，因为Reader需要从InputStream中读入字节流（byte），然后，根据编码设置，再转换为char就可以实现字符流。如果我们查看FileReader的源码，它在内部实际上持有一个FileInputStream。

既然Reader本质上是一个基于InputStream的byte到char的转换器，那么，如果我们已经有一个InputStream，想把它转换为Reader，是完全可行的。InputStreamReader就是这样一个转换器，它可以把任何InputStream转换为Reader。示例代码如下：

```

// 持有InputStream:
InputStream input = new FileInputStream("src/readme.txt");
// 变换为Reader:
Reader reader = new InputStreamReader(input, "UTF-8");

```

构造InputStreamReader时，我们需要传入InputStream，还需要指定编码，就可以得到一个Reader对象。上述代码可以通过try (resource)更简洁地改写如下：

```

try (Reader reader = new InputStreamReader(new FileInputStream("src/readme.txt"), "UTF-8")) {
    // TODO:
}

```

上述代码实际上就是FileReader的一种实现方式。

使用try (resource)结构时，当我们关闭Reader时，它会在内部自动调用InputStream的close()方法，所以，只需要关闭最外层的Reader对象即可。

使用InputStreamReader，可以把一个InputStream转换成一个Reader。

## 小结

Reader定义了所有字符输入流的超类：

- FileReader实现了文件字符流输入，使用时需要指定编码；
- CharArrayReader和StringReader可以在内存中模拟一个字符流输入。

Reader是基于InputStream构造的：可以通过InputStreamReader在指定编码的同时将任何InputStream转换为Reader。

总是使用try (resource)保证Reader正确关闭。

Reader是带编码转换器的InputStream，它把byte转换为char，而Writer就是带编码转换器的OutputStream，它把char转换为byte并输出。

# Writer

Writer和OutputStream的区别如下：

OutputStream	Writer
字节流，以byte为单位	字符流，以char为单位
写入字节（0~255）：void write(int b)	写入字符（0~65535）：void write(int c)
写入字节数组：void write(byte[] b)	写入字符数组：void write(char[] c)
无对应方法	写入String：void write(String s)

Writer是所有字符输出流的超类，它提供的方法主要有：

- 写入一个字符（0~65535）：void write(int c)；
- 写入字符数组的所有字符：void write(char[] c)；
- 写入String表示的所有字符：void write(String s)。

## FileWriter

FileWriter就是向文件中写入字符流的Writer。它的使用方法和FileReader类似：

```
try (Writer writer = new FileWriter("readme.txt", StandardCharsets.UTF_8)) {
    writer.write('H'); // 写入单个字符
    writer.write("Hello".toCharArray()); // 写入char[]
    writer.write("Hello"); // 写入String
}
```

## CharArrayWriter

CharArrayWriter可以在内存中创建一个Writer，它的作用实际上是构造一个缓冲区，可以写入char，最后得到写入的char[]数组，这和ByteArrayOutputStream非常类似：

```
try (Writer writer = new CharArrayWriter()) {
    writer.write(65);
    writer.write(66);
    writer.write(67);
    char[] data = writer.toCharArray(); // { 'A', 'B', 'C' }
}
```

## StringWriter

StringWriter也是一个基于内存的Writer，它和CharArrayWriter类似。实际上，StringWriter在内部维护了一个StringBuffer，并对外提供了Writer接口。

## OutputStreamWriter

除了CharArrayWriter和StringWriter外，普通的Writer实际上是基于OutputStream构造的，它接收char，然后在内部自动转换成一个或多个byte，并写入OutputStream。因此，OutputStreamWriter就是一个将任意的OutputStream转换为Writer的转换器：

```
try (Writer writer = new OutputStreamWriter(new FileOutputStream("readme.txt"), "UTF-8")) {
    // TODO:
}
```

上述代码实际上就是FileWriter的一种实现方式。这和上一节的InputStreamReader是一样的。

## 小结

Writer定义了所有字符输出流的超类:

- FileWriter实现了文件字符流输出;
- CharArrayWriter和StringWriter在内存中模拟一个字符流输出。

使用try (resource)保证Writer正确关闭。

Writer是基于OutputStream构造的, 可以通过OutputStreamWriter将OutputStream转换为Writer, 转换时需要指定编码。

PrintStream是一种FilterOutputStream，它在OutputStream的接口上，额外提供了一些写入各种数据类型的方法：

## PrintStream和PrintWriter

- 写入int: `print(int)`
- 写入boolean: `print(boolean)`
- 写入String: `print(String)`
- 写入Object: `print(Object)`，实际上相当于`print(object.toString())`
- ...

以及对应的一组`println()`方法，它会自动加上换行符。

我们经常使用的`System.out.println()`实际上就是使用PrintStream打印各种数据。其中，`System.out`是系统默认提供的PrintStream，表示标准输出：

```
System.out.print(12345); // 输出12345
System.out.print(new Object()); // 输出类似java.lang.Object@3c7a835a
System.out.println("Hello"); // 输出Hello并换行
```

`System.err`是系统默认提供的标准错误输出。

PrintStream和OutputStream相比，除了添加了一组`print()/println()`方法，可以打印各种数据类型，比较方便外，它还有一个额外的优点，就是不会抛出IOException，这样我们在编写代码的时候，就不必捕获IOException。

### PrintWriter

PrintStream最终输出的总是byte数据，而PrintWriter则是扩展了Writer接口，它的`print()/println()`方法最终输出的是char数据。两者的使用方法几乎是一模一样的：

```
import java.io.*;
----
public class Main {
    public static void main(String[] args) {
        StringWriter buffer = new StringWriter();
        try (PrintWriter pw = new PrintWriter(buffer)) {
            pw.println("Hello");
            pw.println(12345);
            pw.println(true);
        }
        System.out.println(buffer.toString());
    }
}
```

### 小结

PrintStream是一种能接收各种数据类型的输出，打印数据时比较方便：

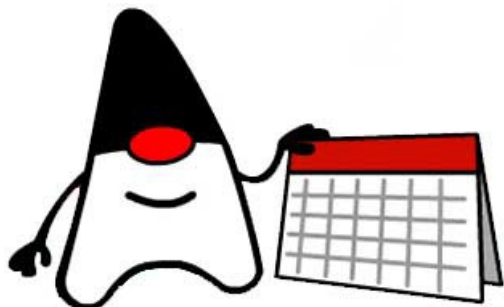
- `System.out`是标准输出；
- `System.err`是标准错误输出。

PrintWriter是基于Writer的输出。

日期与时间是计算机处理的重要数据。绝大部分程序的运行都要和时间打交道。

## 日期与时间

本节我们将详细讲解Java程序如何正确处理日期与时间。



在计算机中，我们经常需要处理日期和时间。

## 基本概念

这是日期：

- 2019-11-20
- 2020-1-1

这是时间：

- 12:30:59
- 2020-1-1 20:21:59

日期是指某一天，它不是连续变化的，而是应该被看成离散的。

而时间有两种概念，一种是不带日期的时间，例如，12:30:59。另一种是带日期的时间，例如，2020-1-1 20:21:59，只有这种带日期的时间能唯一确定某个时刻，不带日期的时间是无法确定一个唯一时刻的。

### 本地时间

当我们说当前时刻是2019年11月20日早上8:15的时候，我们说的实际上是本地时间。在国内就是北京时间。在这个时刻，如果地球上不同地方的人们同时看一眼手表，他们各自的本地时间是不同的：



所以，不同的时区，在同一时刻，本地时间是不同的。全球一共分为24个时区，伦敦所在的时区称为标准时区，其他时区按东 / 西偏移的小时区分，北京所在的时区是东八区。

### 时区

因为光靠本地时间还无法唯一确定一个准确的时刻，所以我们还需要给本地时间加上一个时区。时区有好几种表示方式。

一种是以GMT或者UTC加时区偏移表示，例如：GMT+08:00或者UTC+08:00表示东八区。

GMT和UTC可以认为基本是等价的，只是UTC使用更精确的原子钟计时，每隔几年会有一个闰秒，我们在开发程序的时候可以忽略两者的误差，因为计算机的时钟在联网的时候会自动与时间服务器同步时间。

另一种是缩写，例如，CST表示China Standard Time，也就是中国标准时间。但是CST也可以表示美国中部时间Central Standard Time USA，因此，缩写容易产生混淆，我们尽量不要使用缩写。

最后一种是以洲 / 城市表示，例如，Asia/Shanghai，表示上海所在地的时区。特别注意城市名称不是任意的城市，而是由国际标准组织规定的城市。

因为时区的存在，东八区的2019年11月20日早上8:15，和西五区的2019年11月19日晚上19:15，他们的时刻是相同的：



时刻相同的意思就是，分别在两个时区的两个人，如果在这一刻通电话，他们各自报出自己手表上的时间，虽然本地时间是不同的，但是这两个时间表示的时刻是相同的。

夏令时

时区还不是最复杂的，更复杂的是夏令时。所谓夏令时，就是夏天开始的时候，把时间往后拨1小时，夏天结束的时候，再把时间往前拨1小时。我们国家实行过一段时间夏令时，1992年就废除了，但是矫情的美国人到现在还在使用，所以时间换算更加复杂。



因为涉及到夏令时，相同的时区，如果表示的方式不同，转换出的时间是不一样的。我们举个栗子：

对于2019-11-20和2019-6-20两个日期来说，假设北京人在纽约：

- 如果以GMT或者UTC作为时区，无论日期是多少，时间都是19:00；
- 如果以国家 / 城市表示，例如America / NewYork，虽然纽约也在西五区，但是，因为夏令时的存在，在不同的日期，GMT时间和纽约时间可能是不一样的：

时区	2019-11-20	2019-6-20
GMT-05:00	19:00	19:00
UTC-05:00	19:00	19:00
America/New_York	19:00	20:00

实行夏令时的不同地区，进入和退出夏令时的时间很可能是不同的。同一个地区，根据历史上是否实行过夏令时，标准时间在不同年份换算成当地时间也是不同的。因此，计算夏令时，没有统一的公式，必须按照一组给定的规则来算，并且，该规则要定期更新。



计算夏令时请使用标准库提供的相关类，不要试图自己计算夏令时。

## 本地化

在计算机中，通常使用Locale表示一个国家或地区的日期、时间、数字、货币等格式。Locale由语言\_国家的字母缩写构成，例如，zh\_CN表示中文+中国，en\_US表示英文+美国。语言使用小写，国家使用大写。

对于日期来说，不同的Locale，例如，中国和美国的表示方式如下：

- zh\_CN: 2016-11-30
- en\_US: 11/30/2016

计算机用Locale在日期、时间、货币和字符串之间进行转换。一个电商网站会根据用户所在的Locale对用户显示如下：

	中国用户	美国用户
购买价格	12000.00	12,000.00
购买日期	2016-11-30	11/30/2016

## 小结

在编写日期和时间的程序前，我们要准确理解日期、时间和时刻的概念。

由于存在本地时间，我们需要理解时区的概念，并且必须牢记由于夏令时的存在，同一地区用GMT/UTC和城市表示的时区可能导致时间不同。

计算机通过Locale来针对当地用户习惯格式化日期、时间、数字、货币等。

在计算机中，应该如何表示日期和时间呢？

## Date和Calendar

我们经常看到的日期和时间表示方式如下：

- 2019-11-20 0:15:00 GMT+00:00
- 2019年11月20日8:15:00
- 11/19/2019 19:15:00 America/New\_York

如果直接以字符串的形式存储，那么不同的格式，不同的语言会让表示方式非常繁琐。

在理解日期和时间的表示方式之前，我们先要理解数据的存储和展示。

当我们定义一个整型变量并赋值时：

```
int n = 123400;
```

编译器会把上述字符串（程序源码就是一个字符串）编译成字节码。在程序的运行期，变量n指向的内存实际上是一个4字节区域：

00	01	e2	08
----	----	----	----

注意到计算机内存除了二进制的0/1外没有其他任何格式。上述十六机制是为了简化表示。

当我们用`System.out.println(n)`打印这个整数的时候，实际上`println()`这个方法在内部把`int`类型转换成`String`类型，然后打印出字符串123400。

类似的，我们也可以以十六进制的形式打印这个整数，或者，如果n表示一个价格，我们就以\$123,400.00的形式来打印它：

```
import java.text.*;
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        int n = 123400;
        // 123400
        System.out.println(n);
        // 1e208
        System.out.println(Integer.toHexString(n));
        // $123,400.00
        System.out.println(NumberFormat.getCurrencyInstance(Locale.US).format(n));
    }
}
```

可见，整数123400是数据的存储格式，它的存储格式非常简单。而我们打印的各种各样的字符串，则是数据的展示格式。展示格式有多种形式，但本质上它就是一个转换方法：

```
String toDisplay(int n) { ... }
```

理解了数据的存储和展示，我们回头看看以下几种日期和时间：

- 2019-11-20 0:15:01 GMT+00:00
- 2019年11月20日8:15:01
- 11/19/2019 19:15:01 America/New\_York

它们实际上是数据的展示格式，分别按英国时区、中国时区、纽约时区对同一个时刻进行展示。而这个“同一个时刻”在计算机中存储的本质只是是一个整数，我们称它为Epoch Time。

Epoch Time是计算从1970年1月1日零点（格林威治时区 / GMT+00:00）到现在所经历的秒数，例如：

1574208900表示从1970年1月1日零点GMT时区到该时刻一共经历了1574208900秒，换算成伦敦、北京和纽约时间分别是：

1574208900 = 北京时间2019-11-20 8:15:00  
= 伦敦时间2019-11-20 0:15:00  
= 纽约时间2019-11-19 19:15:00



因此，在计算机中，只需要存储一个整数1574208900表示某一时刻。当需要显示为某一地区的当地时间时，我们就把它格式化为一个字符串：

```
String displayDateTime(int n, String timezone) { ... }
```

Epoch Time又称为时间戳，在不同的编程语言中，会有几种存储方式：

- 以秒为单位的整数：1574208900，缺点是精度只能到秒；
- 以毫秒为单位的整数：1574208900123，最后3位表示毫秒数；
- 以秒为单位的浮点数：1574208900.123，小数点后面表示零点几秒。

它们之间转换非常简单。而在Java程序中，时间戳通常是用long表示的毫秒数，即：

```
long t = 1574208900123L;
```

转换成北京时间就是2019-11-20T8:15:00.123。要获取当前时间戳，可以使用System.currentTimeMillis()，这是Java程序获取时间戳最常用的方法。

## 标准库API

我们再来看一下Java标准库提供的API。Java标准库有两套处理日期和时间的API：

- 一套定义在java.util这个包里面，主要包括Date、Calendar和TimeZone这几个类；
- 一套新的API是在Java 8引入的，定义在java.time这个包里面，主要包括LocalDateTime、ZonedDateTime、ZoneId等。

为什么会有新旧两套API呢？因为历史遗留原因，旧的API存在很多问题，所以引入了新的API。

那么我们能不能跳过旧的API直接用新的API呢？如果涉及到遗留代码就不行，因为很多遗留代码仍然使用旧的API，所以目前仍然需要对旧的API有一定了解，很多时候还需要在新旧两种对象之间进行转换。

本节我们快速讲解旧API的常用类型和方法。

## Date

java.util.Date是用于表示一个日期和时间的对象，注意与java.sql.Date区分，后者用在数据库中。如果观察Date的源码，可以发现它实际上存储了一个long类型的以毫秒表示的时间戳：

```
public class Date implements Serializable, Cloneable, Comparable<Date> {
```

```

        private transient long fastTime;

        ...
    }

```

我们来看Date的基本用法:

```

import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        // 获取当前时间:
        Date date = new Date();
        System.out.println(date.getYear() + 1900); // 必须加上1900
        System.out.println(date.getMonth() + 1); // 0~11, 必须加上1
        System.out.println(date.getDate()); // 1~31, 不能加1
        // 转换为String:
        System.out.println(date.toString());
        // 转换为GMT时区:
        System.out.println(date.toGMTString());
        // 转换为本地时区:
        System.out.println(date.toLocaleString());
    }
}

```

注意getYear()返回的年份必须加上1900, getMonth()返回的月份是0~11分别表示1~12月, 所以要加1, 而getDate()返回的日期范围是1~31, 又不能加1。

打印本地时区表示的日期和时间时, 不同的计算机可能会有不同的结果。如果我们想要针对用户的偏好精确地控制日期和时间的格式, 就可以使用SimpleDateFormat对一个Date进行转换。它用预定义的字符串表示格式化:

- yyyy: 年
- MM: 月
- dd: 日
- HH: 小时
- mm: 分钟
- ss: 秒

我们来看如何以自定义的格式输出:

```

import java.text.*;
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        // 获取当前时间:
        Date date = new Date();
        var sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        System.out.println(sdf.format(date));
    }
}

```

Java的格式化预定义了许多不同的格式, 我们以MMM和E为例:

```

import java.text.*;
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        // 获取当前时间:
        Date date = new Date();
        var sdf = new SimpleDateFormat("E MMM dd, yyyy");
        System.out.println(sdf.format(date));
    }
}

```

上述代码在不同的语言环境会打印出类似Sun Sep 15, 2019这样的日期。可以从[JDK文档](#)查看详细的格式说明。一般来说，字母越长，输出越长。以M为例，假设当前月份是9月：

- M: 输出9
- MM: 输出09
- MMM: 输出Sep
- MMMM: 输出September

Date对象有几个严重的问题：它不能转换时区，除了toGMTString()可以按GMT+0:00输出外，Date总是以当前计算机系统的默认时区为基础进行输出。此外，我们也很难对日期和时间进行加减，计算两个日期相差多少天，计算某个月第一个星期一的日期等。

## Calendar

Calendar可以用于获取并设置年、月、日、时、分、秒，它和Date比，主要多了一个可以做简单的日期和时间运算的功能。

我们来看Calendar的基本用法：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        // 获取当前时间：
        Calendar c = Calendar.getInstance();
        int y = c.get(Calendar.YEAR);
        int m = 1 + c.get(Calendar.MONTH);
        int d = c.get(Calendar.DAY_OF_MONTH);
        int w = c.get(Calendar.DAY_OF_WEEK);
        int hh = c.get(Calendar.HOUR_OF_DAY);
        int mm = c.get(Calendar.MINUTE);
        int ss = c.get(Calendar.SECOND);
        int ms = c.get(Calendar.MILLISECOND);
        System.out.println(y + "-" + m + "-" + d + " " + w + " " + hh + ":" + mm + ":" + ss + "." + ms);
    }
}
```

注意到Calendar获取年月日这些信息变成了get(int field)，返回的年份不必转换，返回的月份仍然要加1，返回的星期要特别注意，1~7分别表示周日，周一，.....，周六。

Calendar只有一种方式获取，即Calendar.getInstance()，而且一获取到就是当前时间。如果我们想给它设置成特定的一个日期和时间，就必须先清除所有字段：

```
import java.text.*;
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        // 当前时间：
        Calendar c = Calendar.getInstance();
        // 清除所有：
        c.clear();
        // 设置2019年：
        c.set(Calendar.YEAR, 2019);
        // 设置9月：注意8表示9月：
        c.set(Calendar.MONTH, 8);
        // 设置2日：
        c.set(Calendar.DATE, 2);
        // 设置时间：
        c.set(Calendar.HOUR_OF_DAY, 21);
        c.set(Calendar.MINUTE, 22);
        c.set(Calendar.SECOND, 23);
        System.out.println(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(c.getTime()));
        // 2019-09-02 21:22:23
    }
}
```

利用`Calendar.getTime()`可以将一个`Calendar`对象转换成`Date`对象，然后就可以用`SimpleDateFormat`进行格式化了。

## TimeZone

`Calendar`和`Date`相比，它提供了时区转换的功能。时区用`TimeZone`对象表示：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        TimeZone tzDefault = TimeZone.getDefault(); // 当前时区
        TimeZone tzGMT9 = TimeZone.getTimeZone("GMT+09:00"); // GMT+9:00时区
        TimeZone tzNY = TimeZone.getTimeZone("America/New_York"); // 纽约时区
        System.out.println(tzDefault.getID()); // Asia/Shanghai
        System.out.println(tzGMT9.getID()); // GMT+09:00
        System.out.println(tzNY.getID()); // America/New_York
    }
}
```

时区的唯一标识是以字符串表示的ID，我们获取指定`TimeZone`对象也是以这个ID为参数获取，`GMT+09:00`、`Asia/Shanghai`都是有效的时区ID。要列出系统支持的所有ID，请使用`TimeZone.getAvailableIDs()`。

有了时区，我们就可以对指定时间进行转换。例如，下面的例子演示了如何将北京时间2019-11-20 8:15:00转换为纽约时间：

```
import java.text.*;
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        // 当前时间：
        Calendar c = Calendar.getInstance();
        // 清除所有：
        c.clear();
        // 设置为北京时区：
        c.setTimeZone(TimeZone.getTimeZone("Asia/Shanghai"));
        // 设置年月日时分秒：
        c.set(2019, 10 /* 11月 */, 20, 8, 15, 0);
        // 显示时间：
        var sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        sdf.setTimeZone(TimeZone.getTimeZone("America/New_York"));
        System.out.println(sdf.format(c.getTime()));
        // 2019-11-19 19:15:00
    }
}
```

可见，利用`Calendar`进行时区转换的步骤是：

1. 清除所有字段；
2. 设定指定时区；
3. 设定日期和时间；
4. 创建`SimpleDateFormat`并设定目标时区；
5. 格式化获取的`Date`对象（注意`Date`对象无时区信息，时区信息存储在`SimpleDateFormat`中）。

因此，本质上时区转换只能通过`SimpleDateFormat`在显示的时候完成。

`Calendar`也可以对日期和时间进行简单的加减：

```
import java.text.*;
import java.util.*;
----
public class Main {
    public static void main(String[] args) {
        // 当前时间：
        Calendar c = Calendar.getInstance();
```

```

        // 清除所有：
        c.clear();
        // 设置年月日时分秒：
        c.set(2019, 10 /* 11月 */, 20, 8, 15, 0);
        // 加5天并减去2小时：
        c.add(Calendar.DAY_OF_MONTH, 5);
        c.add(Calendar.HOUR_OF_DAY, -2);
        // 显示时间：
        var sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date d = c.getTime();
        System.out.println(sdf.format(d));
        // 2019-11-25 6:15:00
    }
}

```

## 小结

计算机表示的时间是以整数表示的时间戳存储的，即**Epoch Time**，Java使用long型来表示以毫秒为单位的时间戳，通过`System.currentTimeMillis()`获取当前时间戳。

Java有两套日期和时间的API:

- 旧的**Date**、**Calendar**和**TimeZone**;
- 新的**LocalDateTime**、**ZonedDateTime**、**ZoneId**等。

分别位于**java.util**和**java.time**包中。

从Java 8开始，`java.time`包提供了新的日期和时间API，主要涉及的类型有：

## LocalDateTime

- 本地日期和时间： `LocalDateTime`, `LocalDate`, `LocalTime`;
- 带时区的日期和时间： `ZonedDateTime`;
- 时刻： `Instant`;
- 时区： `ZoneId`, `ZoneOffset`;
- 时间间隔： `Duration`。

以及一套新的用于取代`SimpleDateFormat`的格式化类型`DateTimeFormatter`。

和旧的API相比，新API严格区分了时刻、本地日期、本地时间和带时区的日期时间，并且，对日期和时间进行运算更加方便。

此外，新API修正了旧API不合理的常量设计：

- `Month`的范围用1~12表示1月到12月；
- `Week`的范围用1~7表示周一到周日。

最后，新API的类型几乎全部是不变类型（和`String`类似），可以放心使用不必担心被修改。

### LocalDateTime

我们首先来看最常用的`LocalDateTime`，它表示一个本地日期和时间：

```
import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        LocalDate d = LocalDate.now(); // 当前日期
        LocalTime t = LocalTime.now(); // 当前时间
        LocalDateTime dt = LocalDateTime.now(); // 当前日期和时间
        System.out.println(d); // 严格按照ISO 8601格式打印
        System.out.println(t); // 严格按照ISO 8601格式打印
        System.out.println(dt); // 严格按照ISO 8601格式打印
    }
}
```

本地日期和时间通过`now()`获取到的总是以当前默认时区返回的，和旧API不同，`LocalDateTime`、`LocalDate`和`LocalTime`默认严格按照[ISO 8601](#)规定的日期和时间格式进行打印。

上述代码其实有一个小问题，在获取3个类型的时候，由于执行一行代码总会消耗一点时间，因此，3个类型的日期和时间很可能对不上（时间的毫秒数基本上不同）。为了保证获取到同一时刻的日期和时间，可以改写如下：

```
LocalDateTime dt = LocalDateTime.now(); // 当前日期和时间
LocalDate d = dt.toLocalDate(); // 转换到当前日期
LocalTime t = dt.toLocalTime(); // 转换到当前时间
```

反过来，通过指定的日期和时间创建`LocalDateTime`可以通过`of()`方法：

```
// 指定日期和时间：
LocalDate d2 = LocalDate.of(2019, 11, 30); // 2019-11-30，注意11=11月
LocalTime t2 = LocalTime.of(15, 16, 17); // 15:16:17
LocalDateTime dt2 = LocalDateTime.of(2019, 11, 30, 15, 16, 17);
LocalDateTime dt3 = LocalDateTime.of(d2, t2);
```

因为严格按照ISO 8601的格式，因此，将字符串转换为`LocalDateTime`就可以传入标准格式：

```
LocalDateTime dt = LocalDateTime.parse("2019-11-19T15:16:17");
LocalDate d = LocalDate.parse("2019-11-19");
LocalTime t = LocalTime.parse("15:16:17");
```



注意ISO 8601规定的日期和时间分隔符是T。标准格式如下：

- 日期：yyyy-MM-dd
- 时间：HH:mm:ss
- 带毫秒的时间：HH:mm:ss.SSS
- 日期和时间：yyyy-MM-dd'THH:mm:ss
- 带毫秒的日期和时间：yyyy-MM-dd'THH:mm:ss.SSS

## DateTimeFormatter

如果要自定义输出的格式，或者要把一个非ISO 8601格式的字符串解析成LocalDateTime，可以使用新的DateTimeFormatter：

```
import java.time.*;
import java.time.format.*;
----
public class Main {
    public static void main(String[] args) {
        // 自定义格式化：
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        System.out.println(dtf.format(LocalDateTime.now()));

        // 用自定义格式解析：
        LocalDateTime dt2 = LocalDateTime.parse("2019/11/30 15:16:17", dtf);
        System.out.println(dt2);
    }
}
```

LocalDateTime提供了对日期和时间进行加减的非常简单的链式调用：

```
import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2019, 10, 26, 20, 30, 59);
        System.out.println(dt);
        // 加5天减3小时：
        LocalDateTime dt2 = dt.plusDays(5).minusHours(3);
        System.out.println(dt2); // 2019-10-31T17:30:59
        // 减1月：
        LocalDateTime dt3 = dt2.minusMonths(1);
        System.out.println(dt3); // 2019-09-30T17:30:59
    }
}
```

注意到月份加减会自动调整日期，例如从2019-10-31减去1个月得到的结果是2019-09-30，因为9月没有31日。

对日期和时间进行调整则使用withXxx()方法，例如：withHour(15)会把10:11:12变为15:11:12：

- 调整年：withYear()
- 调整月：withMonth()
- 调整日：withDayOfMonth()
- 调整时：withMonth()
- 调整分：withMinute()
- 调整秒：withSecond()

示例代码如下：

```
import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2019, 10, 26, 20, 30, 59);
        System.out.println(dt);
        // 日期变为31日：
        LocalDateTime dt2 = dt.withDayOfMonth(31);
    }
}
```

```

        System.out.println(dt2); // 2019-10-31T20:30:59
        // 月份变为9:
        LocalDateTime dt3 = dt2.withMonth(9);
        System.out.println(dt3); // 2019-09-30T20:30:59
    }
}

```

同样注意到调整月份时，会相应地调整日期，即把2019-10-31的月份调整为9时，日期也自动变为30。

实际上，LocalDateTime还有一个通用的with()方法允许我们做更复杂的运算。例如：

```

import java.time.*;
import java.time.temporal.*;
----
public class Main {
    public static void main(String[] args) {
        // 本月第一天0:00时刻:
        LocalDateTime firstDay = LocalDate.now().withDayOfMonth(1).atStartOfDay();
        System.out.println(firstDay);

        // 本月最后一天:
        LocalDate lastDay = LocalDate.now().with(TemporalAdjusters.lastDayOfMonth());
        System.out.println(lastDay);

        // 下月第一天:
        LocalDate nextMonthFirstDay = LocalDate.now().with(TemporalAdjusters.firstDayOfNextMonth());
        System.out.println(nextMonthFirstDay);

        // 本月第1个工作日:
        LocalDate firstWeekday = LocalDate.now().with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY));
        System.out.println(firstWeekday);
    }
}

```

对于计算某个月第1个周日这样的问题，新的API可以轻松完成。

要判断两个LocalDateTime的先后，可以使用isBefore()、isAfter()方法，对于LocalDate和LocalTime类似：

```

import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        LocalDateTime target = LocalDateTime.of(2019, 11, 19, 8, 15, 0);
        System.out.println(now.isBefore(target));
        System.out.println(LocalDate.now().isBefore(LocalDate.of(2019, 11, 19)));
        System.out.println(LocalTime.now().isAfter(LocalTime.parse("08:15:00")));
    }
}

```

注意到LocalDateTime无法与时间戳进行转换，因为LocalDateTime没有时区，无法确定某一时刻。后面我们要介绍的ZonedDateTime相当于LocalDateTime加时区的组合，它具有时区，可以与long表示的时间戳进行转换。

## Duration和Period

Duration表示两个时刻之间的时间间隔。另一个类似的Period表示两个日期之间的天数：

```

import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        LocalDateTime start = LocalDateTime.of(2019, 11, 19, 8, 15, 0);
        LocalDateTime end = LocalDateTime.of(2020, 1, 9, 19, 25, 30);
        Duration d = Duration.between(start, end);
        System.out.println(d); // PT1235H10M30S

        Period p = LocalDate.of(2019, 11, 19).until(LocalDate.of(2020, 1, 9));
        System.out.println(p); // P1M21D
    }
}

```

```
}  
}
```

注意到两个`LocalDateTime`之间的差值使用`Duration`表示，类似`PT1235H10M30S`，表示1235小时10分钟30秒。而两个`LocalDate`之间的差值用`Period`表示，类似`P1M21D`，表示1个月21天。

`Duration`和`Period`的表示方法也符合ISO 8601的格式，它以`P...T...`的形式表示，`P...T`之间表示日期间隔，`T`后面表示时间间隔。如果是`PT...`的格式表示仅有时间间隔。利用`ofXxx()`或者`parse()`方法也可以直接创建`Duration`：

```
Duration d1 = Duration.ofHours(10); // 10 hours  
Duration d2 = Duration.parse("P1DT2H3M"); // 1 day, 2 hours, 3 minutes
```

有的童鞋可能发现Java 8引入的`java.time`API。怎么和一个开源的[Joda Time](#)很像？难道JDK也开始抄袭开源了？其实正是因为开源的Joda Time设计很好，应用广泛，所以JDK团队邀请Joda Time的作者Stephen Colebourne共同设计了`java.time`API。

## 小结

Java 8引入了新的日期和时间API，它们是不变类，默认按ISO 8601标准格式化和解析；

使用`LocalDateTime`可以非常方便地对日期和时间进行加减，或者调整日期和时间，它总是返回新对象；

使用`isBefore()`和`isAfter()`可以判断日期和时间的先后；

使用`Duration`和`Period`可以表示两个日期和时间的“区间间隔”。

LocalDateTime总是表示本地日期和时间，要表示一个带时区的日期和时间，我们就需要ZonedDateTime。

## ZonedDateTime

可以简单地把ZonedDateTime理解成LocalDateTime加ZoneId。ZoneId是java.time引入的新的时区类，注意和旧的java.util.TimeZone区别。

要创建一个ZonedDateTime对象，有以下几种方法，一种是通过now()方法返回当前时间：

```
import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        ZonedDateTime zbj = ZonedDateTime.now(); // 默认时区
        ZonedDateTime zny = ZonedDateTime.now(ZoneId.of("America/New_York")); // 用指定时区获取当前时间
        System.out.println(zbj);
        System.out.println(zny);
    }
}
```

观察打印的两个ZonedDateTime，发现它们时区不同，但表示的时间都是同一时刻（毫秒数不同是执行语句时的时间差）：

```
2019-09-15T20:58:18.786182+08:00[Asia/Shanghai]
2019-09-15T08:58:18.788860-04:00[America/New_York]
```

另一种方式是通过给一个LocalDateTime附加一个ZoneId，就可以变成ZonedDateTime：

```
import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        LocalDateTime ldt = LocalDateTime.of(2019, 9, 15, 15, 16, 17);
        ZonedDateTime zbj = ldt.atZone(ZoneId.systemDefault());
        ZonedDateTime zny = ldt.atZone(ZoneId.of("America/New_York"));
        System.out.println(zbj);
        System.out.println(zny);
    }
}
```

以这种方式创建的ZonedDateTime，它的日期和时间与LocalDateTime相同，但附加的时区不同，因此是两个不同的时刻：

```
2019-09-15T15:16:17+08:00[Asia/Shanghai]
2019-09-15T15:16:17-04:00[America/New_York]
```

## 时区转换

要转换时区，首先我们需要有一个ZonedDateTime对象，然后，通过withZoneSameInstant()将关联时区转换到另一个时区，转换后日期和时间都会相应调整。

下面的代码演示了如何将北京时间转换为纽约时间：

```
import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        // 以中国时区获取当前时间：
        ZonedDateTime zbj = ZonedDateTime.now(ZoneId.of("Asia/Shanghai"));
        // 转换为纽约时间：
        ZonedDateTime zny = zbj.withZoneSameInstant(ZoneId.of("America/New_York"));
        System.out.println(zbj);
        System.out.println(zny);
    }
}
```

要特别注意，时区转换的时候，由于夏令时的存在，不同的日期转换的结果很可能是不同的。这是北京时间9月15日的转换结果：

```
2019-09-15T21:05:50.187697+08:00[Asia/Shanghai]
2019-09-15T09:05:50.187697-04:00[America/New_York]
```

这是北京时间11月15日的转换结果：

```
2019-11-15T21:05:50.187697+08:00[Asia/Shanghai]
2019-11-15T08:05:50.187697-05:00[America/New_York]
```

两次转换后的纽约时间有1小时的夏令时时差。

涉及到时区时，千万不要自己计算时差，否则难以正确处理夏令时。

有了ZonedDateTime，将其转换为本地时间就非常简单：

```
ZonedDateTime zdt = ...
LocalDateTime ldt = zdt.toLocalDateTime();
```

转换为LocalDateTime时，直接丢弃了时区信息。

## 练习

某航线从北京飞到纽约需要13小时20分钟，请根据北京起飞日期和时间计算到达纽约的当地日期和时间。

```
import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        LocalDateTime departureAtBeijing = LocalDateTime.of(2019, 9, 15, 13, 0, 0);
        int hours = 13;
        int minutes = 20;
        LocalDateTime arrivalAtNewYork = calculateArrivalAtNY(departureAtBeijing, hours, minutes);
        System.out.println(departureAtBeijing + " -> " + arrivalAtNewYork);
        // test:
        if (!LocalDateTime.of(2019, 10, 15, 14, 20, 0)
            .equals(calculateArrivalAtNY(LocalDateTime.of(2019, 10, 15, 13, 0, 0), 13, 20))) {
            System.err.println("测试失败!");
        } else if (!LocalDateTime.of(2019, 11, 15, 13, 20, 0)
            .equals(calculateArrivalAtNY(LocalDateTime.of(2019, 11, 15, 13, 0, 0), 13, 20))) {
            System.err.println("测试失败!");
        }
    }

    static LocalDateTime calculateArrivalAtNY(LocalDateTime bj, int h, int m) {
        return bj;
    }
}
```

提示：ZonedDateTime仍然提供了plusDays()等加减操作。

[flight-time练习](#)

## 小结

ZonedDateTime是带时区的日期和时间，可用于时区转换；

ZonedDateTime和LocalDateTime可以相互转换。

使用旧的Date对象时，我们用SimpleDateFormat进行格式化显示。使用新的LocalDateTime或ZonedDateTime时，我们要进行格式化显示，就要使用DateTimeFormatter。

## DateTimeFormatter

和SimpleDateFormat不同的是，DateTimeFormatter不但是不变对象，它还是线程安全的。线程的概念我们会在后面涉及到。现在我们只需要记住：因为SimpleDateFormat不是线程安全的，使用的时候，只能在方法内部创建新的局部变量。而DateTimeFormatter可以只创建一个实例，到处引用。

创建DateTimeFormatter时，我们仍然通过传入格式化字符串实现：

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
```

格式化字符串的使用方式与SimpleDateFormat完全一致。

另一种创建DateTimeFormatter的方法是，传入格式化字符串时，同时指定Locale：

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("E, yyyy-MMMM-dd HH:mm", Locale.US);
```

这种方式可以按照Locale默认习惯格式化。我们来看实际效果：

```
import java.time.*;
import java.time.format.*;
import java.util.Locale;
----
public class Main {
    public static void main(String[] args) {
        ZonedDateTime zdt = ZonedDateTime.now();
        var formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm ZZZZ");
        System.out.println(formatter.format(zdt));

        var zhFormatter = DateTimeFormatter.ofPattern("yyyy MMM dd EE HH:mm", Locale.CHINA);
        System.out.println(zhFormatter.format(zdt));

        var usFormatter = DateTimeFormatter.ofPattern("E, MMMM/dd/yyyy HH:mm", Locale.US);
        System.out.println(usFormatter.format(zdt));
    }
}
```

在格式化字符串中，如果需要输出固定字符，可以用'xxx'表示。

运行上述代码，分别以默认方式、中国地区和美国地区对当前时间进行显示，结果如下：

```
2019-09-15T23:16 GMT+08:00
2019 9月 15 周日 23:16
Sun, September/15/2019 23:16
```

当我们直接调用System.out.println()对一个ZonedDateTime或者LocalDateTime实例进行打印的时候，实际上，调用的是它们的toString()方法，默认的toString()方法显示的字符串就是按照ISO 8601格式显示的，我们可以通过DateTimeFormatter预定义的几个静态变量来引用：

```
var ldt = LocalDateTime.now();
System.out.println(DateTimeFormatter.ISO_DATE.format(ldt));
System.out.println(DateTimeFormatter.ISO_DATE_TIME.format(ldt));
```

得到的输出和toString()类似：

```
2019-09-15
2019-09-15T23:16:51.56217
```

### 小结

对ZonedDateTime或LocalDateTime进行格式化，需要使用DateTimeFormatter类；

`DateTimeFormatter`可以通过格式化字符串和`Locale`对日期和时间进行定制输出。

我们已经讲过，计算机存储的当前时间，本质上只是一个不断递增的整数。Java提供的`System.currentTimeMillis()`返回的就是以毫秒表示的当前时间戳。

## Instant

这个当前时间戳在`java.time`中以`Instant`类型表示，我们用`Instant.now()`获取当前时间戳，效果和`System.currentTimeMillis()`类似：

```
import java.time.*;
----
public class Main {
    public static void main(String[] args) {
        Instant now = Instant.now();
        System.out.println(now.getEpochSecond()); // 秒
        System.out.println(now.toEpochMilli()); // 毫秒
    }
}
```

打印的结果类似：

```
1568568760
1568568760316
```

实际上，`Instant`内部只有两个核心字段：

```
public final class Instant implements ... {
    private final long seconds;
    private final int nanos;
}
```

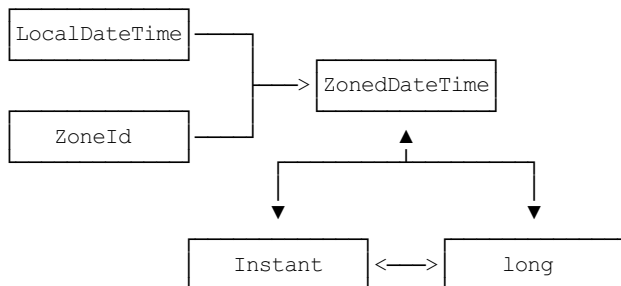
一个是以秒为单位的时间戳，一个是更精确的纳秒精度。它和`System.currentTimeMillis()`返回的`long`相比，只是多了更高精度的纳秒。

既然`Instant`就是时间戳，那么，给它附加上一个时区，就可以创建出`ZonedDateTime`：

```
// 以指定时间戳创建Instant：
Instant ins = Instant.ofEpochSecond(1568568760);
ZonedDateTime zdt = ins.atZone(ZoneId.systemDefault());
System.out.println(zdt); // 2019-09-16T01:32:40+08:00[Asia/Shanghai]
```

可见，对于某一个时间戳，给它关联上指定的`ZoneId`，就得到了`ZonedDateTime`，继而可以获得了对应时区的`LocalDateTime`。

所以，`LocalDateTime`，`ZoneId`，`Instant`，`ZonedDateTime`和`long`都可以互相转换：



转换的时候，只需要留意`long`类型以毫秒还是秒为单位即可。

### 小结

`Instant`表示高精度时间戳，它可以和`ZonedDateTime`以及`long`互相转换。



由于Java提供了新旧两套日期和时间的API，除非涉及到遗留代码，否则我们应该坚持使用新的API。

## 最佳实践

如果需要与遗留代码打交道，如何在新旧API之间互相转换呢？

### 旧API转新API

如果要把旧式的Date或Calendar转换为新API对象，可以通过toInstant()方法转换为Instant对象，再继续转换为ZonedDateTime：

```
// Date -> Instant:
Instant ins1 = new Date().toInstant();

// Calendar -> Instant -> ZonedDateTime:
Calendar calendar = Calendar.getInstance();
Instant ins2 = Calendar.getInstance().toInstant();
ZonedDateTime zdt = ins2.atZone(calendar.getTimeZone().toZoneId());
```

从上面的代码还可以看到，旧的TimeZone提供了一个toZoneId()，可以把自己变成新的ZoneId。

### 新API转旧API

如果要把新的ZonedDateTime转换为旧的API对象，只能借助long型时间戳做一个“中转”：

```
// ZonedDateTime -> long:
ZonedDateTime zdt = ZonedDateTime.now();
long ts = zdt.toEpochSecond() * 1000;

// long -> Date:
Date date = new Date(ts);

// long -> Calendar:
Calendar calendar = Calendar.getInstance();
calendar.clear();
calendar.setTimeZone(TimeZone.getTimeZone(zdt.getZone().getId()));
calendar.setTimeInMillis(zdt.toEpochSecond() * 1000);
```

从上面的代码还可以看到，新的ZoneId转换为旧的TimeZone，需要借助ZoneId.getId()返回的String完成。

## 在数据库中存储日期和时间

除了旧式的java.util.Date，我们还可以找到另一个java.sql.Date，它继承自java.util.Date，但会自动忽略所有时间相关信息。这个奇葩的设计原因要追溯到数据库的日期与时间类型。

在数据库中，也存在几种日期和时间类型：

- DATETIME：表示日期和时间；
- DATE：仅表示日期；
- TIME：仅表示时间；
- TIMESTAMP：和DATETIME类似，但是数据库会在创建或者更新记录的时候同时修改TIMESTAMP。

在使用Java程序操作数据库时，我们需要把数据库类型与Java类型映射起来。下表是数据库类型与Java新旧API的映射关系：

数据库	对应Java类（旧）	对应Java类（新）
DATETIME	java.util.Date	LocalDateTime
DATE	java.sql.Date	LocalDate
TIME	java.sql.Time	LocalTime
TIMESTAMP	java.sql.Timestamp	LocalDateTime

实际上，在数据库中，我们需要存储的最常用的是时刻（Instant），因为有了时刻信息，就可以根据用户自己选择的时区，显示出正确的本地时间。所以，最好的方法是直接用长整数long表示，在数据库中存储为BIGINT类型。

通过存储一个long型时间戳，我们可以编写一个timestampToString()的方法，非常简单地为不同用户以不同的偏好来显示不同的本地时间：

```
import java.time.*;
import java.time.format.*;
import java.util.Locale;
----
public class Main {
    public static void main(String[] args) {
        long ts = 1574208900000L;
        System.out.println(timestampToString(ts, Locale.CHINA, "Asia/Shanghai"));
        System.out.println(timestampToString(ts, Locale.US, "America/New_York"));
    }

    static String timestampToString(long epochMilli, Locale lo, String zoneId) {
        Instant ins = Instant.ofEpochMilli(epochMilli);
        DateTimeFormatter f = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM, FormatStyle.SHORT);
        return f.withLocale(lo).format(ZonedDateTime.ofInstant(ins, ZoneId.of(zoneId)));
    }
}
```

对上述方法进行调用，结果如下：

```
2019年11月20日 上午8:15
Nov 19, 2019, 7:15 PM
```

## 小结

处理日期和时间时，尽量使用新的java.time包；

在数据库中存储时间戳时，尽量使用long型时间戳，它具有省空间，效率高，不依赖数据库的优点。

本节我们介绍Java平台最常用的测试框架JUnit，并详细介绍如何编写单元测试。

## 单元测试

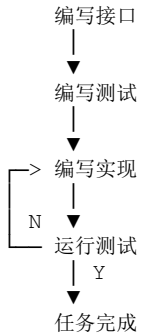


什么是单元测试呢？单元测试就是针对最小的功能单元编写测试代码。Java程序最小的功能单元是方法，因此，对Java程序进行单元测试就是针对单个Java方法的测试。

## 编写JUnit测试

单元测试有什么好处呢？在学习单元测试前，我们可以先了解一下测试驱动开发。

所谓测试驱动开发，是指先编写接口，紧接着编写测试。编写完测试后，我们才开始真正编写实现代码。在编写实现代码的过程中，一边写，一边测，什么时候测试全部通过了，那就表示编写的实现完成了：



这就是传说中的.....



当然，这是一种理想情况。大部分情况是我们已经编写了实现代码，需要对已有的代码进行测试。

我们先通过一个示例来看如何编写测试。假定我们编写了一个计算阶乘的类，它只有一个静态方法来计算阶乘：

$n! = 1 \times 2 \times 3 \times \dots \times n$

代码如下：

```
public class Factorial {
    public static long fact(long n) {
        long r = 1;
        for (long i = 1; i <= n; i++) {
            r = r * i;
        }
        return r;
    }
}
```

要测试这个方法，一个很自然的想法是编写一个main()方法，然后运行一些测试代码：

```
public class Test {
    public static void main(String[] args) {
        if (fact(10) == 3628800) {
            System.out.println("pass");
        } else {
            System.out.println("fail");
        }
    }
}
```

这样我们就可以通过运行main()方法来运行测试代码。

不过，使用main()方法测试有很多缺点：

一是只能有一个main()方法，不能把测试代码分离，二是没有打印出测试结果和期望结果，例如，expected: 3628800, but actual: 123456，三是很难编写一组通用的测试代码。

因此，我们需要一种测试框架，帮助我们编写测试。

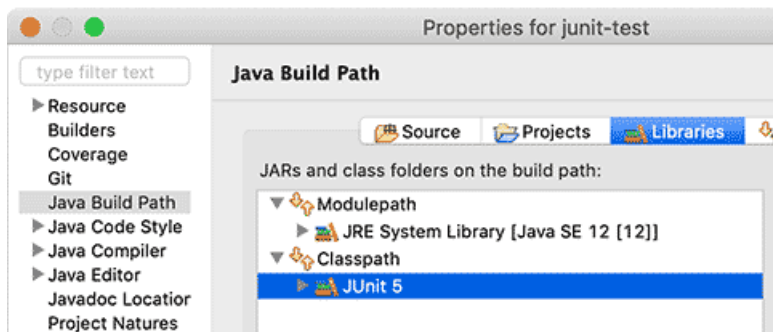
## JUnit

JUnit是一个开源的Java语言的单元测试框架，专门针对Java设计，使用最广泛。JUnit是事实上的单元测试的标准框架，任何Java开发者都应当学习并使用JUnit编写单元测试。

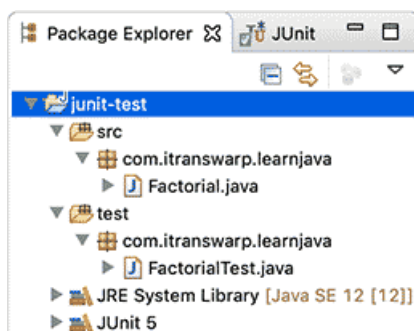
使用JUnit编写单元测试的好处在于，我们可以非常简单地组织测试代码，并随时运行它们，JUnit就会给出成功的测试和失败的测试，还可以生成测试报告，不仅包含测试的成功率，还可以统计测试的代码覆盖率，即被测试的代码本身有多少经过了测试。对于高质量的代码来说，测试覆盖率应该在80%以上。

此外，几乎所有的IDE工具都集成了JUnit，这样我们就可以直接在IDE中编写并运行JUnit测试。JUnit目前最新版本是5。

以Eclipse为例，当我们已经编写了一个Factorial.java文件后，我们想对其进行测试，需要编写一个对应的FactorialTest.java文件，以Test为后缀是一个惯例，并分别将其放入src和test目录中。最后，在Project - Properties - Java Build Path - Libraries中添加JUnit 5的库：



整个项目结构如下：



我们来看一下FactorialTest.java的内容：

```
package com.itranswarp.learnjava;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class FactorialTest {

    @Test
    void testFact() {
        assertEquals(1, Factorial.fact(1));
        assertEquals(2, Factorial.fact(2));
        assertEquals(6, Factorial.fact(3));
    }
}
```

```

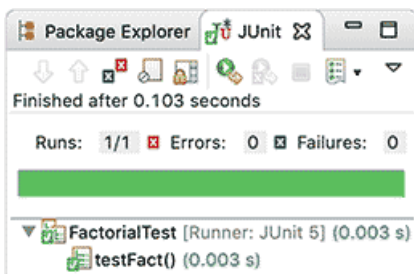
        assertEquals(3628800, Factorial.fact(10));
        assertEquals(2432902008176640000L, Factorial.fact(20));
    }
}

```

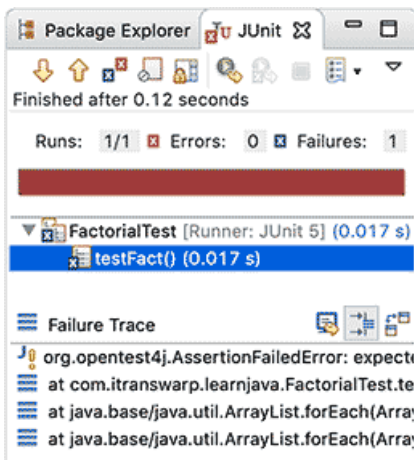
核心测试方法`testFact()`加上了`@Test`注解，这是JUnit要求的，它会把带有`@Test`的方法识别为测试方法。在测试方法内部，我们用`assertEquals(1, Factorial.fact(1))`表示，期望`Factorial.fact(1)`返回1。`assertEquals(expected, actual)`是最常用的测试方法，它在`Assertion`类中定义。`Assertion`还定义了其他断言方法，例如：

- `assertTrue()`:期待结果为`true`
- `assertFalse()`:期待结果为`false`
- `assertNotNull()`:期待结果为非`null`
- `assertArrayEquals()`:期待结果为数组并与期望数组每个元素的值均相等
- ...

运行单元测试非常简单。选中`Factorial.java`文件，点击Run - Run As - JUnit Test，Eclipse会自动运行这个JUnit测试，并显示结果：



如果测试结果与预期不符，`assertEquals()`会抛出异常，我们会得到一个测试失败的结果：



在Failure Trace中，JUnit会告诉我们详细的错误结果：

```

org.opentest4j.AssertionFailedError: expected: <3628800> but was: <362880>
    at org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
    at org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:195)
    at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:168)
    at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:163)
    at org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:611)
    at com.itranswarp.learnjava.FactorialTest.testFact(FactorialTest.java:14)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at ...

```

第一行的失败信息的意思是期待结果3628800但是实际返回是362880，此时，我们要么修正实现代码，要么修正测试代码，直到测试通过为止。

使用浮点数时，由于浮点数无法精确地进行比较，因此，我们需要调用`assertEquals(double expected, double`

`actual, double delta)`这个重载方法，指定一个误差值：

```
assertEquals(0.1, Math.abs(1 - 9 / 10.0), 0.0000001);
```

## 单元测试的好处

单元测试可以确保单个方法按照正确预期运行，如果修改了某个方法的代码，只需确保其对应的单元测试通过，即可认为改动正确。此外，测试代码本身就可以作为示例代码，用来演示如何调用该方法。

使用JUnit进行单元测试，我们可以使用断言（Assertion）来测试期望结果，可以方便地组织和运行测试，并方便地查看测试结果。此外，JUnit既可以直接在IDE中运行，也可以方便地集成到Maven这些自动化工具中运行。

在编写单元测试的时候，我们要遵循一定的规范：

一是单元测试代码本身必须非常简单，能一下看明白，决不能再为测试代码编写测试；

二是每个单元测试应当互相独立，不依赖运行的顺序；

三是测试时不但要覆盖常用测试用例，还要特别注意测试边界条件，例如输入为0，null，空字符串""等情况。

## 练习

[JUnit测试](#)

## 小结

JUnit是一个单元测试框架，专门用于运行我们编写的单元测试：

一个JUnit测试包含若干@Test方法，并使用Assertions进行断言，注意浮点数assertEquals()要指定delta。

在一个单元测试中，我们经常编写多个@Test方法，来分组、分类对目标代码进行测试。

## 使用Fixture

在测试的时候，我们经常遇到一个对象需要初始化，测试完可能还需要清理的情况。如果每个@Test方法都写一遍这样的重复代码，显然比较麻烦。

JUnit提供了编写测试前准备、测试后清理的固定代码，我们称之为**Fixture**。

我们来看一个具体的Calculator的例子：

```
public class Calculator {
    private long n = 0;

    public long add(long x) {
        n = n + x;
        return n;
    }

    public long sub(long x) {
        n = n - x;
        return n;
    }
}
```

这个类的功能很简单，但是测试的时候，我们要先初始化对象，我们不必在每个测试方法中都写上初始化代码，而是通过@BeforeEach来初始化，通过@AfterEach来清理资源：

```
public class CalculatorTest {

    Calculator calculator;

    @BeforeEach
    public void setUp() {
        this.calculator = new Calculator();
    }

    @AfterEach
    public void tearDown() {
        this.calculator = null;
    }

    @Test
    void testAdd() {
        assertEquals(100, this.calculator.add(100));
        assertEquals(150, this.calculator.add(50));
        assertEquals(130, this.calculator.add(-20));
    }

    @Test
    void testSub() {
        assertEquals(-100, this.calculator.sub(100));
        assertEquals(-150, this.calculator.sub(50));
        assertEquals(-130, this.calculator.sub(-20));
    }
}
```

在CalculatorTest测试中，有两个标记为@BeforeEach和@AfterEach的方法，它们会在运行每个@Test方法前后自动运行。

上面的测试代码在JUnit中运行顺序如下：

```
for (Method testMethod : findTestMethods(CalculatorTest.class)) {
    var test = new CalculatorTest(); // 创建Test实例
    invokeBeforeEach(test);
    invokeTestMethod(test, testMethod);
}
```



```
    invokeAfterEach(test);  
}
```

可见，@BeforeEach和@AfterEach会“环绕”在每个@Test方法前后。

还有一些资源初始化和清理可能更加繁琐，而且会耗费较长的时间，例如初始化数据库。JUnit还提供了@BeforeAll和@AfterAll，它们在运行所有@Test前后运行，顺序如下：

```
invokeBeforeAll(CalculatorTest.class);  
for (Method testMethod : findTestMethods(CalculatorTest.class)) {  
    var test = new CalculatorTest(); // 创建Test实例  
    invokeBeforeEach(test);  
        invokeTestMethod(test, testMethod);  
    invokeAfterEach(test);  
}  
invokeAfterAll(CalculatorTest.class);
```

因为@BeforeAll和@AfterAll在所有@Test方法运行前后仅运行一次，因此，它们只能初始化静态变量，例如：

```
public class DatabaseTest {  
    static Database db;  
  
    @BeforeAll  
    public static void initDatabase() {  
        db = createDb(...);  
    }  
  
    @AfterAll  
    public static void dropDatabase() {  
        ...  
    }  
}
```

事实上，@BeforeAll和@AfterAll也只能标注在静态方法上。

因此，我们总结出编写Fixture的套路如下：

1. 对于实例变量，在@BeforeEach中初始化，在@AfterEach中清理，它们在各个@Test方法中互不影响，因为是不同的实例；
2. 对于静态变量，在@BeforeAll中初始化，在@AfterAll中清理，它们在各个@Test方法中均是唯一实例，会影响各个@Test方法。

大多数情况下，使用@BeforeEach和@AfterEach就足够了。只有某些测试资源初始化耗费时间太长，以至于我们不得不尽量“复用”时才会用到@BeforeAll和@AfterAll。

最后，注意到每次运行一个@Test方法前，JUnit首先创建一个XxxTest实例，因此，每个@Test方法内部的成员变量都是独立的，不能也无法把成员变量的状态从一个@Test方法带到另一个@Test方法。

## 练习

[使用Fixture](#)

## 小结

编写Fixture是指针对每个@Test方法，编写@BeforeEach方法用于初始化测试资源，编写@AfterEach用于清理测试资源；

必要时，可以编写@BeforeAll和@AfterAll，使用静态变量来初始化耗时的资源，并且在所有@Test方法的运行前后仅执行一次。

在Java程序中，异常处理是非常重要的。

## 异常测试

我们自己编写的方法，也经常抛出各种异常。对于可能抛出的异常进行测试，本身就是测试的重要环节。

因此，在编写JUnit测试的时候，除了正常的输入输出，我们还要特别针对可能导致异常的情况进行测试。

我们仍然用Factorial举例：

```
public class Factorial {
    public static long fact(long n) {
        if (n < 0) {
            throw new IllegalArgumentException();
        }
        long r = 1;
        for (long i = 1; i <= n; i++) {
            r = r * i;
        }
        return r;
    }
}
```

在方法入口，我们增加了对参数n的检查，如果为负数，则直接抛出IllegalArgumentException。

现在，我们希望对异常进行测试。在JUnit测试中，我们可以编写一个@Test方法专门测试异常：

```
@Test
void testNegative() {
    assertThrows(IllegalArgumentException.class, new Executable() {
        @Override
        public void execute() throws Throwable {
            Factorial.fact(-1);
        }
    });
}
```

JUnit提供assertThrows()来期望捕获一个指定的异常。第二个参数Executable封装了我们要执行的会产生异常的代码。当我们执行Factorial.fact(-1)时，必定抛出IllegalArgumentException。assertThrows()在捕获到指定异常时表示通过测试，未捕获到异常，或者捕获到的异常类型不对，均表示测试失败。

有些童鞋会觉得编写一个Executable的匿名类太繁琐了。实际上，Java 8开始引入了函数式编程，所有单方法接口都可以简写如下：

```
@Test
void testNegative() {
    assertThrows(IllegalArgumentException.class, () -> {
        Factorial.fact(-1);
    });
}
```

上述奇怪的->语法就是函数式接口的实现代码，我们会在后面详细介绍。现在，我们只需要通过这种固定的代码编写能抛出异常的语句即可。

### 练习

观察Factorial.fact()方法，注意到由于long型整数有范围限制，当我们传入参数21时，得到的结果是-4249290049419214848，而不是期望的51090942171709440000，因此，当传入参数大于20时，Factorial.fact()方法应当抛出ArithmeticException。请编写测试并修改实现代码，确保测试通过。

[异常测试](#)

### 小结

测试异常可以使用`assertThrows()`，期待捕获到指定类型的异常；

对可能发生的每种类型的异常都必须进行测试。

在运行测试的时候，有些时候，我们需要排除某些@Test方法，不要让它运行，这时，我们就可以给它标记一个@Disabled:

## 条件测试

```
@Disabled
@Test
void testBug101() {
    // 这个测试不会运行
}
```

为什么我们不直接注释掉@Test，而是要加一个@Disabled? 这是因为注释掉@Test，JUnit就不知道这是个测试方法，而加上@Disabled，JUnit仍然识别出这是个测试方法，只是暂时不运行。它会在测试结果中显示:

Tests run: 68, Failures: 2, Errors: 0, Skipped: 5

类似@Disabled这种注解就称为条件测试，JUnit根据不同的条件注解，决定是否运行当前的@Test方法。

我们来看一个例子:

```
public class Config {
    public String getConfigFile(String filename) {
        String os = System.getProperty("os.name").toLowerCase();
        if (os.contains("win")) {
            return "C:\\\" + filename;
        }
        if (os.contains("mac") || os.contains("linux") || os.contains("unix")) {
            return "/usr/local/" + filename;
        }
        throw new UnsupportedOperationException();
    }
}
```

我们想要测试getConfigFile()这个方法，但是在Windows上跑，和在Linux上跑的代码路径不同，因此，针对两个系统的测试方法，其中一个只能在Windows上跑，另一个只能在Mac/Linux上跑:

```
@Test
void testWindows() {
    assertEquals("C:\\test.ini", config.getConfigFile("test.ini"));
}

@Test
void testLinuxAndMac() {
    assertEquals("/usr/local/test.cfg", config.getConfigFile("test.cfg"));
}
```

因此，我们给上述两个测试方法分别加上条件如下:

```
@Test
@EnabledOnOs(OS.WINDOWS)
void testWindows() {
    assertEquals("C:\\test.ini", config.getConfigFile("test.ini"));
}

@Test
@EnabledOnOs({ OS.LINUX, OS.MAC })
void testLinuxAndMac() {
    assertEquals("/usr/local/test.cfg", config.getConfigFile("test.cfg"));
}
```

@EnableOnOs就是一个条件测试判断。

我们来看一些常用的条件测试:

不在Windows平台执行的测试，可以加上@DisabledOnOs(OS.WINDOWS):

```

@Test
@DisabledOnOs(OS.WINDOWS)
void testOnNonWindowsOs() {
    // TODO: this test is disabled on windows
}

```

只能在Java 9或更高版本执行的测试，可以加上@DisabledOnJre(JRE.JAVA\_8)：

```

@Test
@DisabledOnJre(JRE.JAVA_8)
void testOnJava9OrAbove() {
    // TODO: this test is disabled on java 8
}

```

只能在64位操作系统上执行的测试，可以用@EnabledIfSystemProperty判断：

```

@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void testOnlyOn64bitSystem() {
    // TODO: this test is only run on 64 bit system
}

```

需要传入环境变量DEBUG=true才能执行的测试，可以用@EnabledIfEnvironmentVariable：

```

@Test
@EnabledIfEnvironmentVariable(named = "DEBUG", matches = "true")
void testOnlyOnDebugMode() {
    // TODO: this test is only run on DEBUG=true
}

```

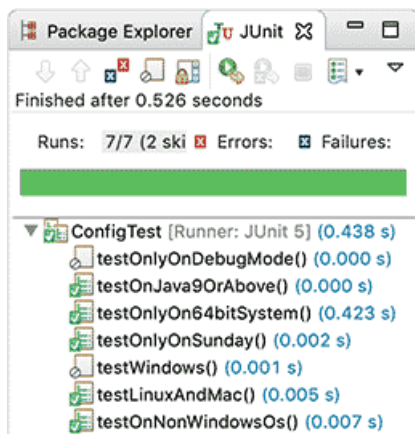
最后，万能的@EnableIf可以执行任意Java语句并根据返回的boolean决定是否执行测试。下面的代码演示了一个只能在星期日执行的测试：

```

@Test
@EnabledIf("java.time.LocalDate.now().getDayOfWeek()==java.time.DayOfWeek.SUNDAY")
void testOnlyOnSunday() {
    // TODO: this test is only run on Sunday
}

```

当我们在JUnit中运行所有测试的时候，JUnit会给出执行的结果。在IDE中，我们能很容易地看到没有执行的测试：



带有⦿标记的测试方法表示没有执行。

## 练习

[条件测试](#)。

## 小结

条件测试是根据某些注解在运行期让JUnit自动忽略某些测试。

如果待测试的输入和输出是一组数据： 可以把测试数据组织起来 用不同的测试数据调用相同的测试方法

## 参数化测试

参数化测试和普通测试稍微不同的地方在于，一个测试方法需要接收至少一个参数，然后，传入一组参数反复运行。

`JUnit`提供了一个`@ParameterizedTest`注解，用来进行参数化测试。

假设我们想对`Math.abs()`进行测试，先用一组正数进行测试：

```
@ParameterizedTest
@ValueSource(ints = { 0, 1, 5, 100 })
void testAbs(int x) {
    assertEquals(x, Math.abs(x));
}
```

再用一组负数进行测试：

```
@ParameterizedTest
@ValueSource(ints = { -1, -5, -100 })
void testAbsNegative(int x) {
    assertEquals(-x, Math.abs(x));
}
```

注意到参数化测试的注解是`@ParameterizedTest`，而不是普通的`@Test`。

实际的测试场景往往没有这么简单。假设我们自己编写了一个`StringUtils.capitalize()`方法，它会把字符串的第一个字母变为大写，后续字母变为小写：

```
public class StringUtils {
    public static String capitalize(String s) {
        if (s.length() == 0) {
            return s;
        }
        return Character.toUpperCase(s.charAt(0)) + s.substring(1).toLowerCase();
    }
}
```

要用参数化测试的方法来测试，我们不但要给出输入，还要给出预期输出。因此，测试方法至少需要接收两个参数：

```
@ParameterizedTest
void testCapitalize(String input, String result) {
    assertEquals(result, StringUtils.capitalize(input));
}
```

现在问题来了：参数如何传入？

最简单的方法是通过`@MethodSource`注解，它允许我们编写一个同名的静态方法来提供测试参数：

```
@ParameterizedTest
@MethodSource
void testCapitalize(String input, String result) {
    assertEquals(result, StringUtils.capitalize(input));
}

static List<Arguments> testCapitalize() {
    return List.of( // arguments:
        Arguments.arguments("abc", "Abc"), //
        Arguments.arguments("APPLE", "Apple"), //
        Arguments.arguments("good", "Good"));
}
```

上面的代码很容易理解：静态方法`testCapitalize()`返回了一组测试参数，每个参数都包含两个`String`，正好作为测试方法的两个参数传入。

如果静态方法和测试方法的名称不同，`@MethodSource`也允许指定方法名。但使用默认同名方法最方便。

另一种传入测试参数的方法是使用@CsvSource，它的每一个字符串表示一行，一行包含的若干参数用,分隔，因此，上述测试又可以改写如下：

```
@ParameterizedTest
@CsvSource({ "abc, Abc", "APPLE, Apple", "gooD, Good" })
void testCapitalize(String input, String result) {
    assertEquals(result, StringUtils.capitalize(input));
}
```

如果有成百上千的测试输入，那么，直接写@CsvSource就很不方便。这个时候，我们可以把测试数据提到一个独立的CSV文件中，然后标注上@CsvFileSource：

```
@ParameterizedTest
@CsvFileSource(resources = { "/test-capitalize.csv" })
void testCapitalizeUsingCsvFile(String input, String result) {
    assertEquals(result, StringUtils.capitalize(input));
}
```

JUnit只在classpath中查找指定的CSV文件，因此，test-capitalize.csv这个文件要放到test目录下，内容如下：

```
apple, Apple
HELLO, Hello
JUnit, Junit
reSource, Resource
```

## 练习

[参数化测试StringUtils](#)

## 小结

使用参数化测试，可以提供一组测试数据，对一个测试方法反复测试。

参数既可以在测试代码中写死，也可以通过@CsvFileSource放到外部的CSV文件中。