

Chp2 Java 语言基础

本章导读

在上一章里，我们介绍了 Java 语言的一些背景，并介绍了 Java 语言环境的配置，为后面进一步的学习打下良好的基础。

从本章开始，我们将逐步开始介绍 Java 语言的语言特性和语法规则。

1 注释

注释指的是一些描述代码的文字。

我们可以把对代码的各方面描述，都写成代码的注释。注释中典型的内容包括：这段代码是如何工作的、这段代码使用了什么算法、这段代码执行的流程如何，等等。

注释不是 Java 代码的一部分，编译时，编译器会把 Java 代码翻译成字节码，而注释则会被编译器自动忽略。因此，代码中有没有注释，都不会影响到代码运行的结果。

但是，注释却是编程中必不可少的内容。有良好注释的代码，能够极大的增强代码的**可读性**。也就是说，加上注释的代码更容易让人读懂。

对于初学者而言，一开始可能并不会意识到注释的意义。然而我们要知道，Java 代码是给机器看的，同时也是给人看的，事实上，程序员看代码的时间远远超过 Java 编译器看代码的时间（想象一下你调试和写程序需要多久，而编译器编译程序需要多久）。因此，我们写程序应当注重“可读性”，也就是说，我们写出的代码应当尽可能让人容易读懂。随着工作量和代码量的增加，越来越多的代码和越来越复杂的逻辑会让程序员很难读懂、修改和维护代码。也许有一天，在看到自己写的代码的时候，你也会发出这样的感叹：“咦，这段代码真的是我写的么？我怎么一点都看不懂了？”（当然，发出这样的感叹说明你确实积累了不少代码了）。这时候，注释能给程序员带来最大的帮助。

Java 中的注释从语法上来说主要有三种：单行注释、多行注释和 javadoc 注释。

1.1 单行和多行注释

单行注释和多行注释，是很多编程语言中常见的注释语法。

在 Java 中，单行注释以“//”开头，直到遇到一个换行符为止。以下为合法的单行注释：

```
Eg 1:  //This is a comment.
Eg 2:
System.out.println("ABC"); // This is another comment
Eg 3:
if (XXX) { //This is comment
} else { //This is else
}
```

多行注释以“/*”开头，以“*/”结尾，在/*和*/之间的所有内容，均作为注释。

以下为多行注释的例子：

```
Eg 1  /* This is Comment */
```

多行注释可以跨行，例如：

```
Eg 2
      /* This is a example
      Of multi-line comment */
```

特别要注意的是，多行注释不能嵌套。例如以下程序片段：

```
/* This is comment
   /*can not have inside comment*/
*/
```

这个程序片段在多行注释中包含了另一个多行注释，会引发一个编译错误。

1.2 javadoc 注释

javadoc 注释，是 Java 中一种比较特殊的注释。这种注释用来生成 api 文档。

程序员在对外发布 Java 代码的时候，还应当对外公布这些代码的用法。这样，才能让其他的程序员能够更加方便的利用已有的代码。在 Java 中，我们往往使用 api 文档，来向其他程序员介绍代码的组成和用法。

例如，Sun 公司发布的 JDK，包括了一个庞大的类库，为了说明这些类的用法，Sun 公司还提供了相应的 api 文档。

Java1.6 的文档链接如下：

<http://java.sun.com/javase/6/docs/api/>

下图显示了 java.lang.Object 类的 api 文档：



对于程序员来说，如何来生成这样的 api 文档呢？很显然，让程序员直接手工编写，是一件非常麻烦的事情。为此，Java 提供了一种相对简单的机制。

首先，在代码中，我们可以使用 javadoc 注释。从语法上说，这种注释由 “/**” 开头，以 “*/” 结尾，在 “/**” 和 “*/” 之间可以有多行文本，并且与多行注释一样，javadoc 注释也不允许嵌套。这是一种特殊的多行注释，可以在代码中描述类、函数等等。

然后，在 JDK 中，有一个 javadoc 命令。这个命令能够从源文件中获得 javadoc 注释，并根据 javadoc 注释，自动生成 api 文档。

例如，我们写出如下代码：

```
/** 这是对类的注释 */  
public class TestJavaDoc{  
    /** 这是对主方法的注释  
    并且有多行 */  
    public static void main(String args[]){  
        System.out.println("test java doc");  
    }  
}
```

上述代码，在类和方法前面，都加上了 javadoc 注释。

然后，进入命令行。在命令行上输入如下命令：

```
javadoc -d doc TestJavaDoc.java
```

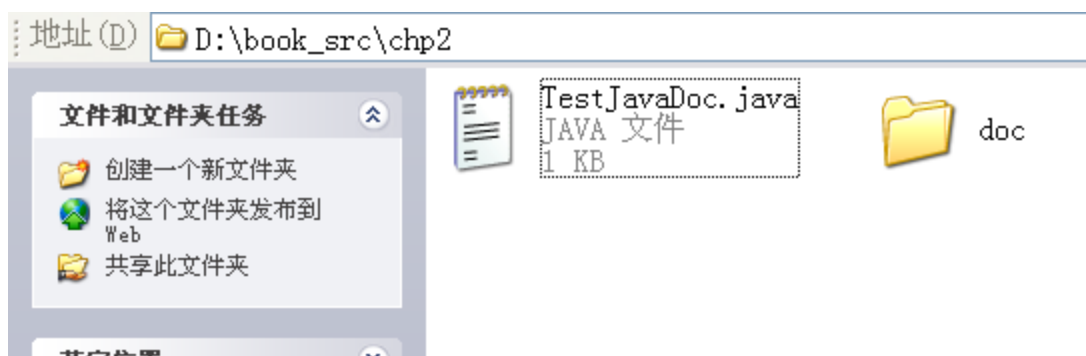
javadoc 命令能够根据代码中的 javadoc 注释生成文档。“-d 文件夹名”是 javadoc 命令的一个选项，这个选项表示的是，生成的文档要放在后面指定的文件夹下。例如，上面“-d doc”，就表示生成的 javadoc 文档要放在 doc 目录下。javadoc 命令最后一个参数，是一个 java 源文件的名称，表示要生成哪一个源文件的文档。

使用该命令之后，命令行上运行的结果如下：

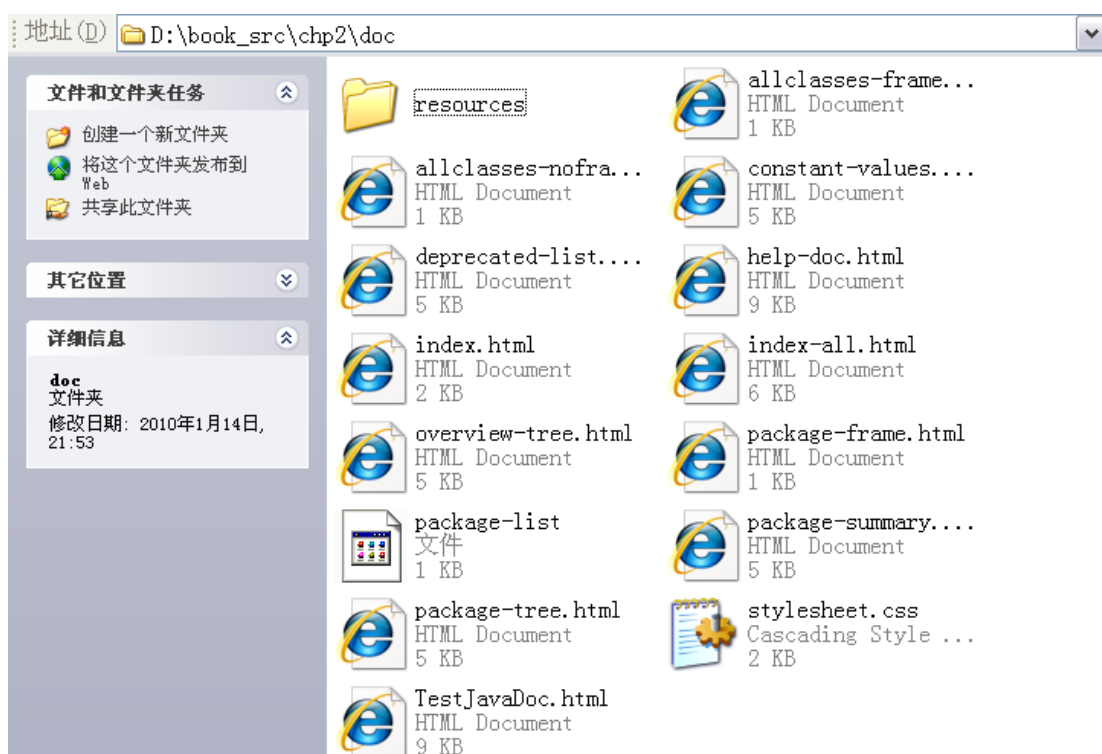


```
D:\book_src\chp2>javadoc -d doc TestJavaDoc.java  
Creating destination directory: "doc\  
Loading source file TestJavaDoc.java...  
Constructing Javadoc information...  
Standard Doclet version 1.5.0  
Building tree for all the packages and classes...  
Generating doc\TestJavaDoc.html...  
Generating doc\package-frame.html...  
Generating doc\package-summary.html...  
Generating doc\package-tree.html...  
Generating doc\constant-values.html...  
Building index for all the packages and classes...  
Generating doc\overview-tree.html...  
Generating doc\index-all.html...  
Generating doc\deprecated-list.html...  
Building index for all classes...  
Generating doc\allclasses-frame.html...  
Generating doc\allclasses-noframe.html...  
Generating doc\index.html...  
Generating doc\help-doc.html...  
Generating doc\stylesheet.css...  
D:\book_src\chp2>
```

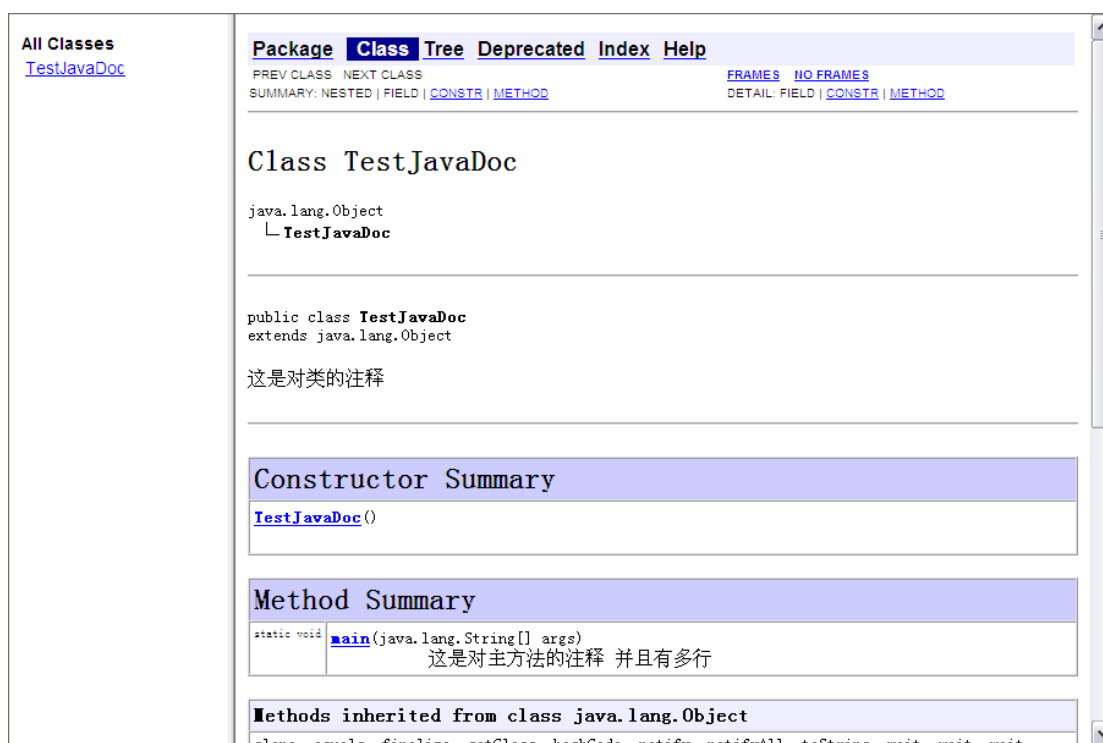
然后，在 D:\book_src\chp2 目录下，生成了一个 doc 目录，如下：



在 doc 目录下，生成了相当多的 html 网页以及一些其他的文件，如下图：



在生成的一系列文件中，可以用浏览器打开 index.html 网页，显示如下：



javadoc 命令自动为我们生成了上图显示的文档。在文档中，“这是对类的注释”，以及“这是对主方法的注释 并且有多行”，这些内容都来源于我们在 TestJavaDoc.java 中写的 javadoc 注释。

上面，我们介绍了如何用 javadoc 注释自动生成 api 文档。关于这种注释的更详细的使用，可以参考 <http://java.sun.com/j2se/javadoc/>。

2 包

随着代码的大量增加，程序员写的.java 源文件以及编译产生的.class 字节码文件会大量的增加。如果任由这种情况发生的话，无论是查询还是管理都会非常的不方便。为了解决这方面的问题，Java 提供了“包”来帮助我们组织和管理类。

本节内容介绍包的使用，主要包括 package 语句和 import 语句。

2.1 package 语句

在操作系统中，如果有大量的文件，为了方便管理，我们往往会按照某种规则，创建结构合理的文件夹结构。例如，如果有大量的 mp3 音乐文件，用户可以把这些文件按照歌手、风格、专辑等，创建相应的文件夹，分门别类的进行管理。

类似的，在 Java 中，为了更好的管理大量的类，引入了“包”的概念。

使用 package 语句可以用来将一个特定的类放入包中。要注意的是，如果要使用 package 语句，则这个语句必须作为.java 文件的第一个语句，并写在任何类的外面。例如，对我们之前写的 HelloWorld 类放入 book.corejava.chp2 包，则代码如下：

```
package book.corejava.chp2;

public class HelloWorld{
```

```

        public static void main(String args[]){
            System.out.println("Hello World");
        }
    }
}

```

上面的这段代码，就把 HelloWorld 类放在了 book.corejava.chp2 包下。要注意的是，在加包之后，使用 HelloWorld 类时必须加上包名作为前缀，因此完整的写法应当是：book.corejava.chp2.HelloWorld。这种在类名前面加上包名的写法称为类的**全限定名**。

假设上述代码存在一个 HelloWorld.java 文件中，而这个文件在硬盘的 C:\JavaFile 目录下，则可以使用编译命令：

```
javac HelloWorld.java
```

该命令在 C:\JavaFile 文件夹下生成 HelloWorld.class 文件。

到现在为止，除了加了一句代码之外，似乎没有别的事情发生。然而接下来，运行.class 文件时，会产生一个错误，结果如下：

```

C:\JavaFile>javac HelloWorld.java

C:\JavaFile>java HelloWorld
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld (wrong name
e: book/corejava/chp2/HelloWorld)
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(Unknown Source)
    at java.security.SecureClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.defineClass(Unknown Source)
    at java.net.URLClassLoader.access$000(Unknown Source)
    at java.net.URLClassLoader$1.run(Unknown Source)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClassInternal(Unknown Source)
Could not find the main class: HelloWorld. Program will exit.

C:\JavaFile>

```

出现这个错误的原因，显然跟我们刚刚给 HelloWorld 加包有关。

在加包之后，使用 HelloWorld 类时，必须使用全限定名运行 HelloWorld 的.class 文件。即，命令必须是

```
java book.corejava.chp2.HelloWorld
```

但是，这样运行，结果依然出错。运行结果如下：

```

C:\JavaFile>java book.corejava.chp2.HelloWorld
Exception in thread "main" java.lang.NoClassDefFoundError: book/corejava/chp2/HelloWorld
Caused by: java.lang.ClassNotFoundException: book.corejava.chp2.HelloWorld
    at java.net.URLClassLoader$1.run(Unknown Source)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClass(Unknown Source)
    at java.lang.ClassLoader.loadClassInternal(Unknown Source)
Could not find the main class: book.corejava.chp2.HelloWorld. Program will exit.

C:\JavaFile>

```

出现这个错误的原因是，在使用包时，**包结构必须和硬盘上的文件夹结构一致**。也就是说，在上述例子中，`HelloWorld.class` 文件，必须放在类路径下的 `book/corejava/chp2/` 目录下。

在 `C:\JavaFile` 文件夹下，建立如下目录结构，并移动.class 文件到特定文件夹下：

```
C:\JavaFile\  
|--HelloWorld.java  
|--book  
    |--corejava  
        |--chp2  
            |--HelloWorld.class
```

示意如下：



此时，使用 `java` 命令：

```
java book.corejava.chp2.HelloWorld
```

才能正确执行。运行结果如下：

```
C:\JavaFile>java book.corejava.chp2.HelloWorld  
Hello World  
  
C:\JavaFile>_
```

特别提醒，执行 `java` 命令时，当前路径必须在字节码文件包结构的上层目录（在本例中，是 `C:\JavaFile` 目录）下，否则将会出现运行时错误。这是因为我们在配置环境变量的时候，将 `CLASSPATH` 这个变量的值设为了“.”。也就是说，JVM 会到当前目录下去寻找所需要的类文件，而在哪个目录下能够找到 `book/corejava/chp2/HelloWorld.class` 这个文件呢？当然是 `book` 目录的上层目录。因此，我们在运行时必须以这个目录作为当前目录。

也可以使用 `javac` 命令的 `-d` 选项，要求编译器把编译生成的.class 文件按照包结构来存放。编译器会按照类的包结构，自动生成对应的目录结构。用法如下：

```
javac -d 目标目录 源文件名
```

其中的目标目录指的是：把生成的目录放在哪个目录下作为子目录。

例如，如果编译之前，硬盘文件结构如下：

```
C:\JavaFile  
|--HelloWorld.java
```

我们把 `C:\JavaFile` 目录作为当前目录，使用命令

```
javac -d . HelloWorld.java
```

这就表示把编译成的字节码连同其目录放入当前目录下。于是，硬盘文件结构变为：

```
C:\JavaFile\  
|--HelloWorld.java  
|--book
```

```
|--corejava
    |--chp2
        |--HelloWorld.class
```

使用包的主要目的是为了避免类名冲突。例如，假设有两个程序员不约而同的都使用 `HelloWorld` 作为类名。如果这两个类都不使用包的话，一方面，两个类的类名相同，因此在使用上会产生歧义。另一方面，在把两个程序员写完的 `.class` 字节码文件放在一起运行的时候，在同一个文件夹下会有两个同名的字节码文件，从而产生文件冲突。而如果这两个程序员使用不同的包 `p1` 和 `p2`，一方面生成的 `HelloWorld.class` 字节码文件一个放在 `p1` 目录下，另一个放在 `p2` 目录下，不会有文件名的冲突。另一方面，全限定名不同，在使用这两个类时，一个被称为 `p1>HelloWorld`，另一个被称为 `p2>HelloWorld`，从而避免了类名的冲突。

为了保持包名的绝对唯一性，Sun 公司建议将公司的 Internet 网址（必然是唯一的）的逆序作为包名，并在不同的项目中使用不同的子包。例如，假设公司的网址为 `abc.com`，则应当使用 `com.abc` 作为包名，而这个公司创建的 `corejava` 项目的包名则可以设定为 `com.abc.corejava`。

2.2 import 语句

把类加上包之后，除了运行时之外，在源代码中使用这个类时也必须使用类的全限定名。例如，假设我们要使用一个 `HelloWorld` 类型的变量，如果用下面的代码

```
public class Welcome{
    public static void main(String args[]){
        HelloWorld hello;
    }
}
```

则编译时会在第三行出错，如下：

```
C:\JavaFile>javac Welcome.java
Welcome.java:3: cannot access HelloWorld
bad class file: .\HelloWorld.java
file does not contain class HelloWorld
Please remove or make sure it appears in the correct subdirectory of the classpath.

        HelloWorld hello;
        ^
1 error

C:\JavaFile>
```

因为在我们使用类的时候，由于 `HelloWorld` 类加了包，因此必须使用全限定名。必须把第三行代码替换成

```
book.corejava.chp2.HelloWorld hello;
```

编译才能通过。

如果要多次使用 `HelloWorld` 类怎么办？例如要定义三个变量，`hello1`，`hello2`，`hello3`，则代码如下：

```
public class UseHelloWorld1{
    public static void main(String args[]){
        book.corejava.chp2.HelloWorld hello1;
```



```

        book.corejava.chp2.HelloWorld hello2;
        book.corejava.chp2.HelloWorld hello3;
    }
}

```

可以看到，冗长的包名出现了三次！这样，我们的代码有大量的冗余，清晰程度降低，可读性降低。

为了解决这个问题，Java 为我们提供了 **import** 语句。

import 语句表示导入特定的类。在使用 **import** 语句导入一个类之后，使用时就能简单的使用类名。例如：

```

import book.corejava.chp2.HelloWorld;
public class Welcome{
    public static void main(String args[]){
        HelloWorld hello;
    }
}

```

前面的 **import** 语句，就好比是一个声明。这个声明表明，在程序接下来的代码中，如果遇到 **HelloWorld** 类，则指的都是之前引入的 **book.corejava.chp2.HelloWorld** 类。这样在后面我们定义 **hello** 这个变量的时候，就省略了冗长的包名。

在使用 **import** 语句之后，如果我们需要三个 **HelloWorld** 类型的变量，则代码如下：

```

import book.corejava.chp2.HelloWorld;
public class UseHelloWorld2{
    public static void main(String args[]){
        HelloWorld hello1;
        HelloWorld hello2;
        HelloWorld hello3;
    }
}

```

可以看出，在使用了 **import** 语句之后，程序的冗余代码大大减少，变得相当的简洁和清晰。

从语法上说，如果要使用 **import** 语句，则这个语句必须放在任何类的外部，仅能在 **package** 语句之后。例如，以下均是 **import** 语句的正确用法：

Eg1

```

// 如果有 package 语句，则 import 语句应在 package 语句之后，
// class 定义之前
package book.corejava.chp2.hello;
import book.corejava.chp2.HelloWorld;
public class ImportTest{
    .....
}

```

Eg 2

```

// 如果没有 package 语句，则 import 语句应为第一个语句
import book.corejava.chp2.HelloWorld;
public class AnotherTest{

```

```
.....  
}
```

此外，一个.java文件中可以有多个import语句，用于导入多个类。

如果想要一次导入某个包下的所有类，则可以使用*通配符。例如，下面这个语句

```
import java.util.*;
```

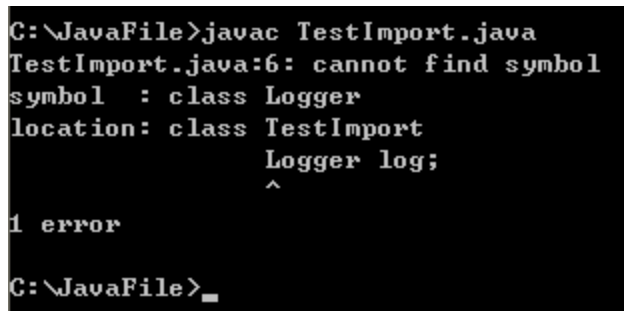
表示导入 java.util 包内的所有类。也就是说，当我们用到这个包里面任何一个类的时候，都可以省略包名。特别要提醒的是，上述语句并不能导入 util 包的子包，也就是说，java.util.jar、java.util.logging 等包下的类并未导入。

例如，我们要使用下面这几个类：

java.util.ArrayList, java.util.HashSet, java.util.logging.Logger, 下面的代码：

```
import java.util.*;  
public class TestImport{  
    public static void main(String args[]){  
        ArrayList list;  
        HashSet set;  
        Logger log;  
    }  
}
```

由于 import java.util.*能够导入 java.util.ArrayList 和 java.util.HashSet 类，但不能导入 java.util 的子包中的类，因此 java.util.logging.Logger 类没有导入，编译器找不到这个类，因此编译出错，如下图所示。



```
C:\JavaFile>javac TestImport.java  
TestImport.java:6: cannot find symbol  
symbol : class Logger  
location: class TestImport  
    Logger log;  
    ^  
1 error  
C:\JavaFile>
```

另外，只能用*导入一个包，而不能使用 java.*或 java.*.*等语法，试图导入所有以 java 为前缀的包。

最后，java.lang 包为标准 Java 库中的最常用的包，这个包中的类会在所有 java 程序中自动被导入。也可以理解为，在所有.java 文件中，import java.lang.*;这个语句都不用写，编译器会自动导入这个包下的类。

3 编码规范

这一小节主要介绍一些简单的编码规范。在实际工程中，一个团队一定会要求团队成员在写代码的时候，必须遵循统一的规范。只有这样，才能够让整个团队的代码风格统一，才能够让写出的代码有着比较好的可读性。

常见的编码规范有：

- 1、良好的注释
- 2、良好的标识符命名
- 3、良好的缩进

作为一名初学者，一定要在一开始的时候就养成良好的编码习惯。首先介绍标识符的命名规范。

下面，我们对“标识符和命名”以及“缩进”做进一步的介绍。

3.1 标识符命名

标识符，指的是程序员为程序组件起的名字。起名字是一门艺术，这一点对标识符也一样。良好的标识符命名风格和习惯，能够大大增加代码的可读性。

我们首先来介绍一下 Java 中标识符的分类。标识符为程序组件的名字，而所有的 Java 程序组件基本分为 5 大类：包、类、变量、函数、常量。首先我们介绍一下 Java 标识符的命名规则。

在 Java 中，关于标识符命名有三条规则。如果我们定义的标识符违背了语法规则，编译时将不能通过。

标识符命名的规则如下：

一、Java 标识符由字母、数字、下划线(_)、货币符号(\$)组成，其中数字不能开头。

要注意的是，所谓“字母”，从技术上说，是一个 unicode 字符，包括中文字符。换句话说，Java 标识符能够使用中文。例如，你可以写一个类叫做“学生”，写一个变量叫做“成绩”等等，Java 程序照样认得。但是在实际编程中，为了避免一些不必要的麻烦，所有 Java 程序员都会使用英文字母起名。对大部分程序员来说，代码里面出现中文总觉得怪怪的。

以下的标识符都是合法的：

abc _score a\$b

但以下的标识符是非法的：

a#(出现了非法字符 #) 2a(不能以数字开头)

二、Java 标识符区分大小写。

也就是说，helloWorld，HelloWorld，HELLOWORLD 这三个标识符，对于 Java 来说是完全不同的三个名字。

三、不能与 Java 关键字重名。

首先我们罗列一下 Java 中的一些关键字：

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

有些关键字我们已经介绍过，例如 class，package 等，其余关键字随着我们的学习，会逐一进行详细讲解。

此外，Java 中有两个单词：goto / const，他们在 Java 中没有特殊的含义，但是由于这两个单词在其他语言中（例如 C 语言）有特殊含义，为了避免其他语言的程序员学习 Java 时产生混淆和误会，Java 语言不允许程序员使用这两个单词。也就是说，虽然在 Java 语言中。goto 和 const 这两个单词没有特殊含义，但是程序员在给程序组件起名字时，依然不能用这两个单词。

严格的说，“true”和“false”也不能称为 Java 语言的关键字。这两个单词是 boolean 类型的字面值，我们稍后再介绍。

除了语法规则之外，还有一些标识符命名方面的习惯。违反了这些习惯的标识符可能是符合语法，从而能够编译通过的。但使用了这样的标识符，会被认为是不良好，不规范的。在 Java 语言中，标识符命名应该注意两大习惯：

一、望文生义

这指的是说，标识符的名字应当起的有意义，最好能通过名字，让人一眼就能看出标识符的作用。例如，变量 `totalScore` 肯定是用来统计总分，函数 `addStudent` 肯定是用来增加一个学生。这样的名字是比较好的名字。它们很容易让人理解这个标识符的意义，从而提高程序的可读性。

二、大小写规范

相比上一条规范，这一条显得非常教条。Java 语言中，对于不同的程序组件，有着不同的大小写规范。罗列一下：

包名：全小写。例如 `book.corejava`

类名：每个单词首字母大写。例如 `HelloWorld`

变量/函数名：首单词小写，后面每个单词首字母大写。例如 `helloWorld`

常量名：全大写，单词之间用下划线分隔。例如 `HELLO_WORLD`

我们可以参考 Java SE 类库中的命名方式，会发现所有的标识符都符合上述两个习惯。这无疑是 Java 世界中约定俗成的，所有程序员都恪守的准则。如果你是一个 Java 初学者，也请从一开始就养成良好的标识符命名习惯，千万不要轻视它们！

3.2 缩进

缩进，指的是在写代码的时候，在某些行的行首，会留出一些空白字符。良好、规范的缩进能够极大的提高代码的清晰程度，让可读性大大增加。

例如，下面的代码，有着混乱的换行和缩进。

```
import book.corejava.chp2.HelloWorld; class BadCode
    {public static void main(String
args[]){System.out.println("welcome")
;HelloWorld
hw;}}
```

然而神奇的是，这一段代码能够编译通过。但是，这样的代码，可读性简直差到了极致！不仔细看几遍是很难读懂这段代码的。

而实际上，对上面的代码简单调整一下换行和缩进，马上就焕然一新：

```
import book.corejava.chp2.HelloWorld;
class GoodCode{
    public static void main(String args[]){
        System.out.println("welcome");
        HelloWorld hw;
    }
}
```

对于缩进的使用，应当注意下面两条：

第一，对于每一个代码块，其内容都应当缩进。例如，`class` 顶格写，而 `class` 的内容，相对于 `class` 都应当有缩进。类似的，主方法有一级缩进；而主方法的所有内容，都应当在主方法的基础上，再缩进一级。

第二，缩进应当统一。也就是说，每一级缩进的长度应当一致。比较推荐的缩进长度是四个空格或者一个制表符（也就是键盘上的 `tab` 键所产生的空白）。

4 变量

4.1 变量的含义

变量是编程中最基本的概念之一。如果你已经熟悉了变量的定义，可以跳过这一节。如果不是，也不用担心。只要把这一小节的内容读完，然后简单的写两个程序，相信你很快就会掌握。

对于 Java 语言而言，每一个变量都代表着内存中的一小块区域，而这块区域能够用来存放某个数据。例如，在 Java 中，我们需要存放一个整数，那就可以定义一个变量，代码如下：

```
int a;
```

上面的这行代码定义了一个 `int` 类型的变量 `a`。首先简单介绍一下 `int` 类型，这种类型表示整数，一个 `int` 类型的变量占 4 个字节的空间。

定义一个 `int` 类型的变量，实际上完成了下面这样两个步骤：

- 1、在 JVM 中，分配 4 个字节的空间，用来存放一个整数。
- 2、为这 4 个字节的空间起了一个名字 `a`，供程序员使用。

这样，在程序中，我们就可以用“`a`”这个名字来指代这四个字节的空间，从而访问到这个空间中所存放的整数。下面是对变量操作的一些例子：

```
public class TestVariable{
    public static void main(String args[]){
        int a;
        a = 10;
        System.out.println(a);
        a = 15;
        System.out.println(a);
        int b;
        b = a;
        System.out.println(b);
    }
}
```

我们仔细分析上面这段代码。

- 1) `a = 10;`

这被称之为变量的赋值。这句话的意思是，把 `a` 所表示的 4 个字节的空间，设置为

数值 10。

2) `System.out.println(a);`

在这个语句中，打印 `a`，就是指打印 `a` 变量的值，也就是 `a` 所代表的 4 个字节的值。此时，`a` 的值为 10，因此打印结果为 10。

3) `a = 15;`

再次为 `a` 变量赋值。这时，把 `a` 所表示的 4 个字节的空間，设置为 15。原来 `a` 的值 10 被新值 15 所替代。

4) `System.out.println(a);` 再次输出时，输出结果为新值 15

5) `int b;` 定义一个新变量 `b`。这个变量同样为 `int` 类型。

6) `b = a;`

这句话的含义是，把 `a` 变量的值赋值给 `b` 变量。`a` 变量的值是 15，通过赋值，把 `b` 变量的值也设置为 15。

7) 输出 `b` 变量，此时输出结果为 15

运行结果如下：

```
C:\JavaFile>javac TestVariable.java

C:\JavaFile>java TestVariable
10
15
15

C:\JavaFile>
```

我们再来进一步研究一下 Java 中变量的一些基本操作。

```
01: public class TestVariable{
02:     public static void main(String args[]){
03:         int a;
04:         a = 10;
05:         int b = 20;
06:         int c, d;
07:         int e = 30, f = 40;
08:         c = a + b;
09:         d = 5;
10:         d = d + c;

        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
        System.out.println(e);
        System.out.println(f);
    }
}
```

第 3、4 行，定义了一个 `a` 变量，并赋值为 10；

第 5 行，在定义一个变量 `b` 的同时，为其赋值为 20。这说明可以在定义一个变量的同

时赋值。这样的写法，和先定义变量之后再赋值，最终的结果是一样的。也就是说，第 5 行的代码，可以等价的写成：

```
int b;  
b = 20;
```

相对而言，在定义变量的同时赋值，能够让代码显得更加简洁。

第 6 行，在一个语句中定义了两个变量 c 和 d。这说明在一个语句中可以定义多个变量，变量之间用逗号隔开。

第 7 行，在一个语句中定义了两个变量 e 和 f，并同时对这两个变量赋值。

第 8 行，把 a 和 b 变量的值相加，并把结果保存在 c 变量中。由于 a 变量是 10，b 变量是 20，因此最后 c 变量结果是 30。

第 9 行，把 d 的值设为 5。

第 10 行，把 d 和 c 的值相加，赋值给 d。怎么来理解呢？在进行计算的时候，首先会计算“=”右边。此时，需要计算 d+c 的结果。由于此时 d 的结果为 5，c 的结果为 30，因此 d+c 的值为 35。

然后，把 35 赋值给 d。此时，d 的结果为 35。

程序的输出结果如下：



```
C:\JavaFile>javac TestVariable.java  
  
C:\JavaFile>java TestVariable  
10  
20  
30  
35  
30  
40  
  
C:\JavaFile>_
```

此外，在 Java 中，变量是**强类型**的。所谓强类型，指的是每一个变量都有自身的类型。不同的变量类型，大小不同，所能存放的数据也不同。例如，int 类型的变量只能存放整数数据，下面的代码是错误的：

```
int a = 123.45;
```

相比强类型来说，有些编程语言是弱类型的。用户只需要定义一个变量，就可以存入任意类型的数据。显然这样会引起混乱，甚至我们可能自己都很难搞清楚一个变量中究竟装的是哪种数据。很庆幸，Java 不是这样的。

可以把变量想象成不同大小的盒子。不同类型的盒子，具有不同的空间大小，被用来存放不同类型的东西。饭盒和鞋盒，不仅大小是不同的，而且装的东西也不一样。我们绝不会把一双皮鞋放在饭盒里，是吧？

4.2 变量的类型

下面我们就来介绍一下 Java 语言中的变量类型。在 Java 中，所有变量类型分为两类，一类为基本类型，一类为对象类型。本章主要介绍的是**基本类型**。

基本类型，英语叫 **primitive type**，也有人翻译成“原始类型”、“简单类型”等等。这类变量属于编程语言中比较基础的组成部分，因此也被称之为“基本类型”。

基本类型总共分为 8 种，分别为 byte、short、int、long、float、double、char、boolean。

下面让我们学习这八种基本类型。

4.2.1 整数类型

有四种类型都用来表示整数，他们是 `byte`、`short`、`int`、`long`，他们之间的区别在于他们所占的内存空间和表示范围。下表为这四种基本类型的参数。

类型名称	所占空间	表示范围
<code>byte</code>	1 个字节	-128 ~ 127
<code>short</code>	2 个字节	-32768 ~ 32767
<code>int</code>	4 个字节	-2147483648 ~ 2147483647
<code>long</code>	8 个字节	-9223372036854775808 ~ 9223372036854775807

可以把这四种基本类型想象成四个大小不同的饭盒，虽然空间不同，但是所装的数据基本上是一类的。要注意的是，表示范围小的类型可以直接赋值给表示范围大的类型，而反之不行。例如：

```
int a = 10;
byte b = 10;
a = b; //可以，表示范围小的类型赋值给范围更大的类型
b = a; //编译错误！表示范围大的类型不能赋值给范围小的类型！
```

也不能给一个变量赋一个超过其表示范围的值。例如：

```
byte b1 = 100; //可以赋值
byte b2 = 150; //编译错误！150 超过了 byte 类型的表示范围！
```

字面值

字面值，指的是某个类型的合法取值，或者说，可以为该类型的变量赋值的数据。例如，“`int a = 5;`”，在这个代码中，`a` 就是变量，`5` 就是字面值。

要注意的是，字面值同样有类型。对于 `1`、`5`、`10`、`-99` 等整数字面值来说，其类型都是 `int` 类型。

但是，下面的代码能够正常执行：

```
byte b = 100;
```

在上面的这一行代码中，虽然 `100` 是一个 `int` 类型的字面值，但是由于 `100` 在 `byte` 类型的表示范围中，因此程序能够自动把 `100` 这个 `int` 类型转成 `byte` 类型。

另外，如果需要 `long` 类型字面值，我们可以用在数值后面加 `L` 的方式（大小写均可）。例如，`1000` 是一个 `int` 类型的字面值，而 `1000L` 是一个 `long` 类型的字面值。例如，下面的代码都是正确的：

```
long l = 1000L;
long l = 1000l;
```

当然，小写的“`l`”容易和数字“`1`”混淆，因此最好还是用大写的“`L`”。

4.2.2 浮点数类型

在计算机术语中，小数被称为浮点数。在 Java 语言中，浮点数有两种，分别为 `float` 和 `double`。两者相关参数如下：

类型名称	所占空间	表示范围
<code>float</code>	4 个字节	3.4028235×10^{38} $\sim 1.4 \times 10^{-45}$
<code>double</code>	8 个字节	$1.7976931348623157 \times 10^{308}$ $\sim 4.9 \times 10^{-324}$

浮点数类型的符号可以是正的，也可以是负的。

字面值

浮点数的字面值有两种。第一种是直接给出小数，例如 1.5，-0.38 等。需要注意的是，这样给出的字面值都是 `double` 类型，如果需要 `float` 类型的字面值，需要在数值后面写一个字母 `f`（大小写均可）。例如：1.6f，-10.39F。事实上，为了更明确的表示 `double` 类型的字面值，也可以在数值后面写一个字母 `d`（大小写均可）。例如：1.6d，-10.39D

第二种是用科学计数法表示。例如， -1.5×10^{23} ，就可以用 -1.5e23 来表示。而 3.8×10^{-5} ，则可以用 3.8e-5 来表示。而 `float` 类型的字面值，则在数值后面再加一个 `f`。例如下面的代码

```
double d = 2.67e13;
float f = 1.57e-3f;
```

计算机表示小数，自然就涉及到表示精度的问题。由于计算机内部使用二进制表示小数，与我们通常的十进制表示法不同，因此小数在表示过程中有可能会有精度方面的损失。例如下面的程序：

```
public class TestFloat {
    public static void main(String[] args) {
        double a = 2.0 - 1.1;
        System.out.println(a);
    }
}
```

这段代码会输出：

0.8999999999999999

在上面的程序中，显示的结果并不是我们期望的“0.9”，这就是因为计算机内部用二进制表示这个数的时候，产生了精度上的问题。这就类似于，用十进制表示数，很难精确的表示 $1/3$ 一样。

4.2.3 字符类型

Java 中的字符类型为 `char` 类型，其本质为一个无符号整数。相关参数如下：

类型名称	所占空间	表示范围
<code>char</code>	2 个字节	0 ~ 65536

在计算机中，一个字符是由一个正整数表示，这个整数被称为字符的编码。Java 中的

`char` 类型存放的就是字符的编码。例如，大写字母 ‘A’ 的编码为 65，用 16 进制表示为 0x41。给一个字符变量赋值总共有三种方式：字面值赋值，编码赋值，`unicode` 赋值。三种赋值方式的语法如下：

```
char ch1 = 'A'; //字面值赋值
char ch2 = 65; //编码赋值
char ch3 = '\u0041'; //unicode 赋值
```

字面值赋值：直接给出用单引号包围的单个字符。注意，Java 中单引号和双引号有不同的含义，单引号用来包围字符，双引号用来包围字符串。

编码赋值：直接给出字符的编码值，例如给出大写字母 ‘A’ 的编码 65。

Unicode 赋值：Unicode 是一种国际字符编码规范，能够处理全世界各国的语言。如果要使用这种赋值方式，则需要在一对单引号之下给出 `\u` 和 4 位 16 进制的 `unicode` 编码。在这个例子中，十六进制数 0041 表示成十进制为数字 65，因此 `ch3` 表示的也是大写字母 ‘A’。

由于 Java 中的 `char` 类型采用 Unicode 编码方式，前面介绍过，Unicode 能够处理世界各国的语言字符，因此，`char` 类型可以用来处理中文。例如：

```
char ch1 = '中';
```

另外，有一些字符在 java 中有特殊含义，在使用时要通过反斜杠 “\” 来转义。例如，想表示一个单引号字符 (')，则必须使用 (\') 的语法。除了单引号 (\') 之外，双引号 (\")、反斜杠 (\\)、换行符 (\n)、制表符 (\t) 都是一些常用的转义字符。

4.2.4 布尔类型

布尔类型用来进行逻辑运算。Java 中的布尔类型为 `boolean` 类型，这种类型只有两种字面值：`true` 或者 `false`。要注意的是布尔类型无法跟其他类型进行运算，也无法自动转化为其他类型。例如：

```
boolean flag = true; // 正确
boolean flag = 1; //错误! 1 不能转换为 boolean 类型!
```

4.3 变量的基本操作

介绍完八种基本类型之后，接下来我们要介绍的是对变量的一些基本操作。

4.3.1 强制类型转换

我们在上一节中介绍，表示范围大的类型，不能给表示范围小的类型赋值。例如，下面的代码就会产生一个编译错误：

```
int a = 10;
byte b;
b = a; //!编译错误，int 类型不能直接给 byte 类型赋值
```

编译出错的原因，在于 `int` 类型中能够保存的数值，有可能远大于 `byte` 类型的表示范围。就好像如果要把一个大汤盆中的东西倒到一个小碗中，很有可能会出问题。这时候，编译器就会阻拦我们的这个操作。

但是，有些情况下，大汤盆中的东西只剩一点儿了，这个时候完全可以把大汤盆中的

东西倒到小碗中。对于上面的代码来说，虽然 `int` 的表示范围远大于 `byte` 类型，但是我们可以确认，`int` 变量 `a` 中的数据，完全可以让一个 `byte` 变量来保存。这种情况，我们不希望编译器阻拦我们。因此，可以使用强制类型转换。

强制类型转换的语法如下：

(类型) 变量

这个语法表示，把变量的值强制转换为某个特定的类型。例如上面的代码，我们要把 `a` 的值强制转换为 `byte` 类型，再给 `b` 赋值，则可以把代码写成：

```
b = (byte) a;
```

需要注意的是，上面的语法转换的是 `a` 变量的值。在执行过程中，会先分配一个 `byte` 类型的临时空间，然后把 `a` 的值转换成 `byte` 类型，然后放到这个临时空间去，最后把临时空间的值赋值给 `b`。在转换过程中，并没有改变 `a` 变量的类型和值。

那 `a` 变量有 4 个字节，怎么转换成为 1 个字节的 `byte` 类型呢？Java 会把 `a` 变量 4 个字节中，最后的一个字节取出来，然后把这个字节的值，放到临时空间去。而另外的 3 个字节的内容，则在强制类型转换的时候被舍弃。

因此，如果 `a` 中数值的范围超过了 `b` 所表示的范围，使用强制类型转换也能正常编译，但是转换的结果会有问题。例如

```
int a = 150;
byte b = (byte) a; //编译通过，但是 b 的值为一个错误值
System.out.println(b); //输出为-106
```

为什么一个超过表示范围的正数，转换之后会变成一个负数呢？这涉及到在计算机中如何用二进制表示正数。例如，下面的代码：

```
short s = 450;
byte b = (byte) s;
```

首先我们来看 `short` 类型的变量 `s`。一个 `short` 类型占 2 个字节，一个字节是 8 个二进制位，因此一个 `short` 变量占 16 个二进制位。这 16 个二进制位中，有 15 个字节是表示数值，最高的一个二进制位表示的是数值的符号。因此，整数的最高位也被称之为符号位。如果一个整数是正数，则它的符号位为 0；相反，负数的符号位为 1。

450 是正数，因此其符号位为 0。然后，把十进制数 450，转换成二进制之后，得到结果 111000010。然后，除了符号和数值位之外，在其余位上补 0。最后，450 在内存中表示如下：

0	0	0	0	0	0	0	1	1	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

然后，把 `s` 强制转换成 `byte` 类型。此时，会开辟 1 个字节的空間，并把 `short` 类型最低 8 位取出来，得到结果如下：

1	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---

可以看到，在转换过程中，舍弃了 `short` 变量的部分内容。

然后，11000010 这个数字，被当做 `byte` 类型时，首先要判断它的符号。由于这个数字的最高位为 1，因此 java 会把这个数当做是负数来处理。负数的表示方式与正数不同，相对比较复杂，在此不多介绍。需要明确的是，在强制类型转换时，如果对超过表示范围的数做强制类型转换，有可能产生一些意想不到的情况，例如正数变成负数，或者负数变成正数。因此，使用强制类型转换应当要注意，避免发生类似的错误。

4.3.2 自动类型提升

在正式介绍 Java 中的各种基本运算符之前，先给大家介绍 Java 中很特别的一个语法性质：自动类型提升。请看下面的一段代码：

```
byte a = 10, b = 20;  
byte c = a+b;
```

对于大部分语言而言，这都应该是一段合法的代码，而对于 Java 语言来说，这段代码会出一个编译时错误：



```
C:\JavaFile>javac TestVariable.java  
TestVariable.java:4: 可能损失精度  
找到: int  
需要: byte  
byte c = a + b;  
                ^  
1 错误
```

要理解产生这个错误的原因，就要了解 Java 的自动类型提升特性。我们首先来看下面这个表达式：

```
byte a = 10, b = 20, c = 40;  
byte d = a + b + c;
```

在运行上面的代码时，首先把 a、b、c 三个变量的值分别设为 10、20 和 40。然后，在计算 d 的值时，首先需要计算“=”右边表达式的值。由于这个表达式有两个“+”，因此在计算的时候，需要进行两次加法操作：首先计算 a+b 的值，在计算出结果之后，再把结果和 c 的值相加。为此，Java 会首先获得 a 变量的值 10，然后获得 b 变量的值 20，之后进行第一次加法，计算出结果 30。因为这个结果在后面的运算中还要使用，所以，Java 会把这个数值保存在一个临时变量中。然后，Java 进行第二个加法运算时，首先取出临时变量的值 30，然后取出 c 变量的值 40，再计算出结果是 70。70 作为整个“a+b+c”表达式的值，也会存在一个临时变量里。最后，把这个临时变量的值赋值给 d 变量。

可以这么来理解上述的过程：就好比我们在做 a+b+c 这道数学题时，首先，会把 a+b 的值计算出来，计算出来之后，会把这个值暂时写在草稿纸上。再然后，计算这个写在草稿纸上的值和 c 相加的结果，并把所得到的最终结果也写在草稿纸上。最后，再把草稿纸上的值，重新写回到考卷上。

因此，当 Java 遇到例如 c = a + b 这样的式子时，会首先计算等号右边的 a+b 的值，然后赋值给等号左边的变量 c。在计算过程中，Java 会创建一个临时变量来保存 a+b 这个表达式的计算结果，然后把这个临时变量中的值赋值给 c。基本过程如下：

计算 a+b 的值 --> 保存到临时变量 --> 把临时变量的值赋给 c。

问题在于，尽管 a 和 b 两个变量都是 byte 类型，但是 Java 为临时变量选择类型时，会将这个类型“自动的提升”为 int 类型，大小为 4 个字节。于是，上述过程中的第三步，就成了把一个 int 类型赋值给 byte 类型的操作，从而产生一个编译时错误。这就是 Java 语言中的自动类型提升特性。

要避免这个问题，只需要对其结果进行强制类型转换即可。即把原代码改为：

```
byte c = (byte) (a+b);
```

需要注意的是，由于是对 a+b 的结果进行强制转换，因此要对 a+b 这个表达式加上括号。Java 自动类型提升的规则如下：

1. 如果运算数中存在 double，则自动类型提升为 double

2. 如果运算数中没有 `double` 但存在 `float`，则自动类型提升为 `float`
3. 如果运算数中没有浮点类型，但存在 `long`，则自动类型提升为 `long`
4. 其他所有情况，自动类型提升为 `int`。

换言之，`byte + byte`，`byte + short` 之类的运算，都会被自动提升为 `int` 类型。需要说明的是，`char` 类型也能进行运算，并且 `char` 类型与其他类型运算时，也会进行相应的自动类型提升。

4.3.3 运算符

在本节中，我们将详细给大家介绍 Java 中的常用运算符。

在介绍运算符之前，首先介绍一个非常基本的概念：表达式。

所谓的表达式，指的是用运算符连接变量或字面值所形成的式子，例如：`a+b`，`2+c` 等。需要强调的一点是，任何一个表达式都会有一个值。也可以理解为，所有表达式都会返回一个结果，例如 `1+2` 会返回 3 作为表达式的结果。

表达式的值也有不同类型。例如，布尔表达式，就说明这个表达式的值的类型为 `boolean` 类型。因此，在写程序的过程中，一定要明确表达式的值，以及这个值的类型。

理解了表达式的概念，我们来关注形成表达式的关键元素：运算符

◆ 赋值号 (=)

赋值操作是编程中最常用的操作之一。Java 中的赋值号为一个等号 (=)。赋值号具有右结合性，也就是说，会先计算赋值号右边的内容，然后把计算结果赋值给左边的变量。

此外，`a = b` 构成一个赋值表达式，其作用就是将变量 `b` 中的值赋值给变量 `a`，这个表达式也有值，表达式的值为赋值号右边的计算结果。

由这个特性，在 Java 中可以进行连等操作，即：`a = b = c = 10` 这样的赋值操作。

◆ 基本数学运算

基本数学运算包括加 (+) 减 (-) 乘 (*) 除 (/) 以及取余 (%) 操作。这些操作和数学上的定义没有区别。

需要注意的是，类似于 `3/2` 这样的表达式。由于 3 和 2 都是整数类型，因此根据自动类型提升的规则，结果也一定是整数类型。因此，`3/2` 这个表达式的值是 1，如果希望得到数学上精确的结果 1.5，则需要使用 `3.0/2` 这样的表达式，来保证结果数据是 `double` 类型的。

此外，Java 中的数学运算符符合先乘除，后加减的规则。例如：`2+3*4` 这个表达式的值为 14。

◆ +=, -=, *=, /=

在实际编程中，经常会写出类似于 `a = a + 2` 之类的表达式，表示把 `a` 在原有值的基础上增加 2。这种写法有一种更方便的简写形式：`a += 2`。

`-=`，`*=`，`/=` 等运算符与 `+=` 类似。

◆ ++与--

对于 `a+=1` 这样的表达式而言，还有一种更加简单的运算符：`++`，与之类似的，`a-=1` 可以用 `a--` 来代替。

要注意的是，使用 `++`（或 `--`）运算符有两种方式：前缀式或后缀式。例如：

```
a++; //后缀式
++a; //前缀式
```

要注意的是这两种方式的区别和联系。首先，对于 `a++` 和 `++a` 这两种方式而言，对 `a` 的操作是完全一样的，都会在表达式运算结束之后把 `a` 的值加 1。这两种方式所不同的地方在于表达式的值不同。

例如，假设 `a = 5`，则不管执行 `a++` 还是 `++a`，执行之后 `a` 的值均为 6。所不同的是，`a++` 这个表达式的值为 5（即 `a` 加 1 之前的值），而 `++a` 这个表达式的值为 6（即 `a` 加 1 之后的值）。

`a--` 和 `--a` 有类似的关系。

◆ 位运算符

Java 语言中包含了四种位运算符：按位与（&），按位或（|），按位异或（^），取反~。位运算符主要用于对数据的每个二进制位进行运算。

首先，从逻辑上说，与、或、异或的逻辑运算如下表所示

运算值 1	运算值 2	与	或	异或
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

可以来这么记忆：与运算只有在两者都为 1 时结果才为 1，或运算只要有一个为 1 结果就为 1，异或运算两者不同结果为 1。

要计算一个位运算的表达式结果，先将十进制的运算数转化为二进制，然后再进行按位操作。

例如，对于十进制数 10 和 6，转化为二进制之后为 1010 和 0110。对其分别进行与、或和异或的操作：

```

      1010      1010      1010
    & 0110    |  0110    ^  0110
    -----
      0010      1110      1100
  
```

因此计算所得的结果分别为：2、14、12。

~操作对整数按位求反。例如下面的代码：

```

int i = 10;
i = ~i;
  
```

这段代码中，首先定义了一个 `int` 类型的变量 `i`。这个变量占 32 位（4 个字节），在内存中表示如下：

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

然后，`~i` 表示按位求反，原数位为 1 的转为 0，0 转为 1。得到的结果如下：

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

这个数表示十进制的数 -11。

还要注意的是，这四种位操作一般只用来操作整数。而与、或这两种运算符，除了能进行整数操作之外，还能进行 `boolean` 类型的操作。关于这一点，我们会在讲述布尔运算时进行更详细的阐述。

◆ 移位操作

Java 中的移位运算符有三种：算术右移（>>），逻辑右移（>>>）和左移（<<）。

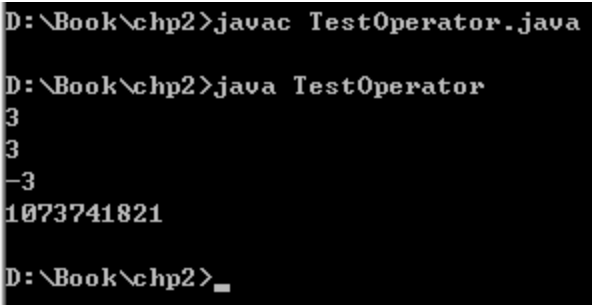
移位操作是指，把一个整数的二进制表示形式，向某个方向（左或者右）移动，并按照一定的规律，丢弃和补充相应位上的值。

例如，对于十进制数 6（0110），进行右移 1 位运算，结果为 3（0011），右移 2 位结果

为 1 (0001)。可以看到，右移 1 位于除 2 的结果一致，右移 n 位与除 2^n 结果一致。而左移与右移相反，对于 6 (0110)，左移 1 位结果为 12 (1100)，左移两位结果为 24 (11000)。因此可以看出，左移 n 位与乘以 2^n 结果一致。

逻辑右移与算术右移的区别很微妙。对于 $n \ggg m$ (逻辑右移) 与 $n \gg m$ (算术右移) 来说，当 n 为正数时，两种运算的结果是一样的。所不同的是，当 n 为负数时，算术右移的结果为一个负数，并且是基本符合算术规律的 (所以它叫算术右移)，而逻辑右移的结果为一个正数。例如下面的代码：

```
int n = 12;
System.out.println(n>>2); //结果为 3
System.out.println(n>>>2); //结果也为 3
n = -12;
System.out.println(n>>2); //结果为负
System.out.println(n>>>2); //结果为正!!!
运行结果如下：
```



产生这种不同的原因，与计算机中表示整数的方式有关。之前介绍过，在计算机中，保存一个整数时，整数的最高位用来表示符号。其中，符号位为 0 表示正数，1 表示负数。在使用算术右移时，移动数值时不会改变符号位的值，因此正数移动之后，符号位为 0，负数移动之后，符号位依然是 1。这样，对一个整数进行算术右移之后，正数依然是正数，负数依然是负数。

然而使用逻辑右移时，最高位总会补上 0。对于正数来说，符号位没有改变；而对于负数来说，符号位由 1 变成了 0。因此，使用逻辑右移时，所得到的结果总是正数。

◆ 布尔运算

我们首先把所有布尔运算符分为两大类。

第一类运算符为：> (大于), >= (大于等于), < (小于), <= (小于等于), == (相等), != (不相等)，这些运算符都接受两个参数，返回一个布尔值，表示判断结果。需要注意的是判断相等 (==) 是两个等号，一定要把这个布尔运算和赋值号 (=) 区分开来。

第二类运算符为：与 (&&)、或 (||)、非 (!)，这些运算符只能接受两个布尔类型的运算数。非运算符表示取反，即!true 结果为 false，而!false 结果为 true。对于与运算和或运算，运算规则见下表：

运算数 1	运算数 2	与	或
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

布尔运算与 (&&) 和位运算与 (&) 具有类似的运算结果，而布尔运算或 (||) 和位运算或 (|) 也具有类似的运算结果。所不同的有两点：

1、布尔运算（&&和||）只能接受布尔值作为运算数，而位运算除了能进行布尔值的运算之外，还能进行整数运算；

2、布尔运算具有短路特性。

什么是短路特性呢？对于与运算而言，如果运算数 1 的值为 `false`，则无论运算数 2 的值是 `true` 还是 `false`，结果一定是 `false`。因此，如果计算机遇到第一个运算数为 `false` 的与（&&）操作，则不会去查看第二个运算数而直接返回。这就是所谓的短路特性。

“与运算”能被 `false` 短路，相对应的，“或运算”能被 `true` 短路（如果第一个操作数为 `true`，则不管第二个操作数如何，结果一定为 `true`）。在 Java 中，布尔运算（&&和||）具有短路特性，而位运算（&和|）不具有短路特性。

短路特性和++（--）运算符结合在一起会产生很多有趣的式子。例如下面的式子：

```
int a = 4, b = 5;
```

```
boolean flag = (a++>4) && (b++>3);
```

请读者思考一下，这段代码运行结束之后，`flag`、变量 `a`、变量 `b` 的值分别是什么？

答案：`flag` 为 `false`，`a` 的值为 5，`b` 的值也为 5。

说明：

在上述代码中，有两个括号，这两个括号的值会由左向右依次执行。

首先，计算 `a++>4`。在计算这个式子时，首先会执行 `a++`，执行完这个表达式之后，`a` 的值变为 5。但是，`a++` 这个表达式的值是 4，而 `4>4` 为 `false`，因此第一个括号中，表达式的值为 `false`。

之后，由于后面遇到的是“&&”运算符，这个运算符能够被 `false` 短路，因此，第一个括号中的表达式计算出是 `false` 之后，第二个括号中的代码不会执行，因此 `b++` 这个代码没有被执行，因此 `b` 的值没有被改变。

可是，如果表达式变为：

```
int a = 4, b = 5;
```

```
boolean flag = (a++>4) & (b++>3);
```

由于位运算符&不具有短路特性，因此，尽管第一个表达式已经被计算出是 `false`，第二个括号中的代码依然会执行。最终的结果，`b` 的值会变化为 6。

◆ 三元操作符？：

Java 中只有唯一的一个三元运算符，其基本语法如下：

布尔表达式 ? 表达式 1 : 表达式 2

说这个运算符是三元运算符，指的是这个运算符在使用时，能够接受三个部分参与运算。

从语法上说，第一部分是一个布尔表达式，第二、第三个部分分别是一个表达式。第一部分和第二部分用“？”隔开，第二部分和第三部分用“：”隔开。这三个部分构成一个完整的三元表达式。

第一部分布尔表达式有一个值，这个值要么是 `true`，要么是 `false`。而表达式 1 有一个值，表达式 2 也有一个值。这三个值参与运算，最后能够得出整个三元表达式的值。

整个表达式的值由下面的原则确定：

如果布尔表达式的值为 `true`，则整个三元表达式的值为表达式 1 的值；如果布尔表达式的值为 `false`，则整个三元表达式的值为表达式 2 的值。例如下面的代码：

```
c = a>b ? a-b : b-a;
```

上面的代码，`a>b` 是布尔表达式，`a-b` 是表达式 1，`b-a` 是表达式 2。而整个表达式的值要么是 `a-b` 的值，要么是 `b-a` 的值。得到结果之后，再把计算所得的值赋值给变量 `c`。

那么上面的代码完成什么功能呢？我们分情况讨论。如果 `a` 大于 `b`，则`(a>b)`这个表达式

的值为 `true`，这样，整个表达式的值就是 `a-b`。如果 `a` 小于 `b`，则`(a<b)`这个表达式的值为 `false`。因此，整个三元表达式的值就是 `b-a`。

综上所述，无论 `a` 和 `b` 的值是多少，上述代码都能让 `a` 和 `b` 这两个变量中，较大的变量减去较小的变量，并把获得的差赋值给 `c` 变量。

◆ 运算符的优先级

在我们小学学习四则混合运算的时候，老师曾经反复的跟我们说，一定要注意，先乘除，后加减。对于下面的式子：

$$2 + 3 * 2$$

这个式子要先计算 `3*2`，再把所得到的结果与 `2` 相加。在这个运算的过程中，“先乘除，后加减”，体现的就是运算符优先级的思想：乘除法的优先级比较高，如果有乘除运算的话，应当先计算。

在 `Java` 中，同样有运算符优先级的概念。我们把本章提到的运算符的优先级罗列如下：

优先级	运算符	说明
1	()	最高优先级
2	!, ~, ++, --	除了括号外，一元操作符优先级最高
3	*, /, %	
4	+, -	<code>Java</code> 中同样满足先乘除，后加减
5	<<, >>, >>>	移位操作
6	<, <=, >, >=	比大小
7	==, !=	判断是否相等
8	&	按位与
9	^	异或
10		按位或
11	&&	与（逻辑操作）
12		或（逻辑操作）
13	?:	三元操作
14	=, +=, -=, *=, ...	所有的赋值操作

在 `Java` 中进行计算时，会先进行优先级高的运算，再进行优先级低的运算。但是，圆括号“`()`”的优先级是最高的，如果有圆括号的话，就先计算圆括号中的值。

在写代码的时候，一般不用刻意去记忆操作符的优先级。如果不能确定运算符的优先级，可以使用圆括号`()`来指定运算的顺序。

4.4 String 类型初探

`String` 类型不是八种基本类型的一种，从分类上说，`String` 类型是一种对象类型。但是 `String` 类型和普通的对象类型又不尽相同，作为一个比较基本的数据类型，它有一些区别于其他对象类型的特点。

`String` 类是 `Java` 语言中比较重要的一个类型，随着学习的深入，我们会对 `String` 类型的了解越来越深入。在本章中，我们主要介绍 `String` 类型的三个特点。

一、`String` 类型有字面值。

关于这一点，我们之前已经有过接触：`String` 类型的字面值为双引号（`""`）中包含的内

容。例如：

```
String str = "Hello World";
```

要注意的是单引号和双引号的区别。单引号用在字符类型上，而双引号用在字符串上。

例如：

```
char ch1 = 'A'; //单引号用在字符类型字面值上
char ch2 = 'AB'; //错误！char 类型不能表示多个字符
char ch3 = "A"; //错误！
String str1 = "A"; //表示只有一个字符的字符串
String str2 = "ABC"; //表示有多个字符的字符串
String str3 = 'A'; //错误！
```

二、String 类型支持加法运算。

字符串类型是对象类型中唯一支持加法运算的类型。字符串加法表示字符串的连接。例如：

如：

```
String str1 = "Hello World";
String str2 = " Welcome";
// 字符串加法，输出结果为：Hello World Welcome
System.out.println(str1 + str2);
```

需要注意的是字符串加法和字符加法之间的区别。字符串加法表示连接，而字符的加法表示字符编码数值的相加，两者有本质的不同。例如：

```
System.out.println("a" + "b"); //字符串加法，输出 ab
System.out.println('a' + 'b'); //字符加法，将 a 和 b 的编码相加，输出一个整数，由于'a'的编码为 97，'b'的编码为 98，因此将输出 195。
```

三、任何类型与 String 类型相加，结果都为 String 类型。

这个特性可以当做是 String 类型的“自动类型提升”。例如：

```
int i = 10;
System.out.println("1234" + i);
```

输出 123410，整数 i 会被自动“提升”为字符串“10”，其结果就是“1234”和“10”这两个字符串的连接结果。

4.5 局部变量

局部变量是 Java 中很重要的一个概念，我们首先来看局部变量的定义。

局部变量是指：在方法内部定义的变量。在目前，我们能接触到的所有变量都被定义在主方法内部，因此目前我们接触到的变量都是局部变量。

关于局部变量，Java 有三条规则：

一、先赋值，后使用。

关于这条规则，可以看以下这个例子：

```
public static void main(String args[]){
    int a;
    System.out.println(a); // 编译时错误！因为 a 没有被赋值
}
```

在 Java 语言中，任何一个局部变量都必须先被赋值过之后，才能使用。这一点和很多其他语言不同。

二、局部变量有作用范围，其作用范围从定义的位置开始，到包含它的代码块结束。

我们看下面两段代码的例子：

```
public static void main(String args[]){
    System.out.println(a); //编译错误！
    int a = 10;
}
```

```
public static void main(String args[]){
    {
        int a = 10;
    }
    System.out.println(a); //编译错误！
}
```

这两段代码都会产生一个编译错误。对于第一个例子而言，输出语句在定义 **a** 变量之前就是用了 **a**，显然不在 **a** 变量作用范围之内，因此是一个编译错误。对于第二个例子而言，**a** 变量作用范围为代码块内部，而输出语句在代码块外部，因此也会产生一个编译错误。

三、重合范围内，两个局部变量不能有命名冲突。

请看下面这段代码的例子：

```
01: public static void main(String args[]){
02:     int a = 10;
03:     {
04:         int a = 20;
05:         System.out.println(a); //编译错误！
06:     }
07:     System.out.println(a);
08: }
```

这段代码会产生一个编译时错误。产生错误的原因，是由于在 **main** 方法中定义的第一个变量 **a**，其作用范围为 2~8 行，而第二个 **a** 变量作用范围为 4~6 行。因而在 4~6 行，有两个局部变量重名，则会产生一个编译时错误。

要修改也简单，可以把代码改成：

```
01: public static void main(String args[]){
02:     {
03:         int a = 20;
04:         System.out.println(a);
05:     }
06:     int a = 10;
07:     System.out.println(a);
08: }
```

这样，两个局部变量虽然重名，但是作用范围没有交集，也就不存在上述的问题。

本章小结

在这一章中，我们给读者介绍了 **Java** 语言中的一些基本的语法知识。

注释是我们学习的第一个语法点，希望读者能够在编程过程中，能够从一开始就养成良

好注释的习惯。

包是我们学习的第二个语法点。实际工作中，几乎每一个程序都会处于特定的包下，从而能避免名字冲突，并且更加利于工程管理。从规范的角度而言，希望大家能够逐渐养成使用包的良好习惯。

还有一个与规范相关的知识点，是我们提到的标识符命名规范。只有使用了正确且规范的标识符命名，才能使我们的程序具有更高的可读性。为此，强烈要求读者严格的按照标识符命名规范的要求写程序。

除了一些基本的程序规范之外，本章的重点是 Java 中一些关于变量的基本知识。希望读者通过本章的学习，能掌握 8 种基本类型和 `String` 类型的一些基本操作，掌握自动类型提升的概念和原则，掌握运算符的用法以及局部变量的概念和使用的原则。