Java学习

一、 Java基础语法

1.1 基本语法

一个 Java 程序可以认为是一系列对象的集合，而这些对象通过调用彼此的方法来协同工作。下面简要介绍下类、对象、方法和实例变量的概念。

- **对象**：对象是类的一个实例，有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- **类**：类是一个模板，它描述一类对象的行为和状态。
- **方法**：方法就是行为，一个类可以有很多方法。逻辑运算、数据修改以及所有动作都是在方法中完成的。
- **实例变量**：每个对象都有独特的实例变量，对象的状态由这些实例变量的值决定。

编写 Java 程序时，应注意以下几点：

---大小写敏感：Java 是大小写敏感的，这就意味着标识符 Hello 与 hello 是不同的。

---类名：对于所有的类来说，类名的首字母应该大写。如果类名由若干单词组成，那么每个单词的首字母应该大写，例如 **MyFirstJavaClass** 。

---方法名：所有的方法名都应该以小写字母开头。如果方法名含有若干单词，则后面的每个单词首字母大写。

---源文件名：源文件名必须和类名相同。当保存文件的时候，你应该使用类名作为文件名保存（切记 Java 是大小写敏感的），文件名的后缀为 **.java**。（如果文件名和类名不相同则会导致编译错误）。

---主方法入口：所有的 Java 程序由 **public static void main(String[] args)** 方法开始执行。

1.2 Java标识符

Java 所有的组成部分都需要名字。类名、变量名以及方法名都被称为标识符。

关于 Java 标识符，有以下几点需要注意：

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

- 非法标识符举例：123abc、-salary

1.3 Java修饰符

像其他语言一样，Java可以使用修饰符来修饰类中方法和属性。主要有两类修饰符：

- 访问控制修饰符 : default, public , protected, private
- 非访问控制修饰符 : final, abstract, static, synchronized

在后面的章节中我们会深入讨论 Java 修饰符。

1.4 Java变量

Java 中主要有如下几种类型的变量

- 局部变量
- 类变量（静态变量）
- 成员变量（非静态变量）

1.5 Java数组

数组是储存在堆上的对象，可以保存多个同类型变量。在后面的章节中，我们将会学到如何声明、构造以及初始化一个数组。

1.6 Java枚举

ava 5.0引入了枚举，枚举限制变量只能是预先设定好的值。使用枚举可以减少代码中的bug。

例如，我们为果汁店设计一个程序，它将限制果汁为小杯、中杯、大杯。这就意味着它不允许顾客点除了这三种尺寸外的果汁。

```
class FreshJuice{
    enum FreshJuiceSize{ SMALL, MEDIUM, LARGE}
    FreshJuiceSize size;
}
public static FreshJuiceTest{
    public static void main(String[] args){
        FreshJuice juice = new FreshJuice();
        juice.size = FreshJuice.FresheJuiceSize.MEDIUM;
    }
}
```

注意：枚举可以单独声明或者声明在类里面。方法、变量、构造函数也可以在枚举中定义。

1.7 Java关键字

1.7.1 访问控制

private protected public default

1.7.2 类、方法和变量修饰符

abstract--声明抽象 class--类 extends--扩充、继承 final--最终值，不可改变

implements--实现（接口） interface--接口 native--本地，原生方法（非java实现）

new--新，创建 static--静态 strictfp--严格、精准 synchronized-线程，同步

transient--短暂 volatile--易失

1.7.3 程序控制语句

break--跳出循环 case--定义一个值以供switch选择 continue--继续

default--默认 do--运行 else--否则 for--循环 if--如果

instanceof--实例 return--返回 switch--根据值选择执行 while--循环

1.7.4 错误处理

assert--断言表达式是否为真 catch--捕捉异常 finally--有没有异常都执行

throw--抛出一个异常对象 throws--声明一个异常可能被抛出

try--捕获异常

1.7.5 包相关

import--引入、导入 package--包

1.7.6 基本类型

boolean--布尔型 byte--字节型 char--字符型 double--双精度浮点 float--单精度浮点

int--整型 long--长整型 short--短整型

1.7.7 变量引用

super--父类、超类 this--本类 void--无返回值

1.7.8 保留关键字

goto--是关键字，但不能使用 const--是关键字，但不能使用 null--空

1.8 Java注释

类似于 C/C++、Java 也支持单行以及多行注释。注释中的字符将被 Java 编译器忽略。

```
public class HelloWorld {
   /* 这是第一个Java程序
    * 它将打印Hello World
    * 这是一个多行注释的示例
    */
    public static void main(String[] args){
       // 这是单行注释的示例
       /* 这个也是单行注释的示例 */
       System.out.println("Hello World");
    }
}
```

1.9 Java空行

空白行或者有注释的行，Java 编译器都会忽略掉。

1.10 Java继承

在 Java 中，一个类可以由其他类派生。如果你要创建一个类，而且已经存在一个类具有你所需要的属性或方法，那么你可以将新创建的类继承该类。

利用继承的方法，可以重用已存在类的方法和属性，而不用重写这些代码。被继承的类称为超类（super class），派生类称为子类（subclass）。

1.11 Java接口

在 Java 中，接口可理解为对象间相互通信的协议。接口在继承中扮演着很重要的角色。

接口只定义派生要用到的方法，但是方法的具体实现完全取决于派生类。

二、 Java对象和类

Java作为一种面向对象语言。支持以下基本概念：

- 多态
- 继承
- 封装

- 抽象
- 类
- 对象
- 实例
- 方法
- 重载

本节我们重点研究对象和类的概念。

- **对象**：对象是类的一个实例（**对象不是找个女朋友**），有状态和行为。例如，一条狗是一个对象，它的状态有：颜色、名字、品种；行为有：摇尾巴、叫、吃等。
- **类**：类是一个模板，它描述一类对象的行为和状态。

2.1 Java中的对象

现在让我们深入了解什么是对象。看看周围真实的世界，会发现身边有很多对象，车，狗，人等等。所有这些对象都有自己的状态和行为。

拿一条狗来举例，它的状态有：名字、品种、颜色，行为有：叫、摇尾巴和跑。

对比现实对象和软件对象，它们之间十分相似。

软件对象也有状态和行为。软件对象的状态就是属性，行为通过方法体现。

在软件开发中，方法操作对象内部状态的改变，对象的相互调用也是通过方法来完成。

2.2 Java中的类

类可以看成是创建 Java 对象的模板。

通过下面一个简单的类来理解下 Java 中类的定义：

```
public class Dog{
    String breed;
    int age;
    String color;
    void barking(){}
    void hungry(){}
    void sleeping(){}
}
```

一个类可以包含以下类型变量：

- **局部变量**：在方法、构造方法或者语句块中定义的变量被称为局部变量。变量声明和初始化都是在方法中，方法结束后，变量就会自动销毁。
- **成员变量**：成员变量是定义在类中，方法体之外的变量。这种变量在创建对象的时候实例化。成员变量可以被类中方法、构造方法和特定类的语句块访问。
- **类变量**：类变量也声明在类中，方法体之外，**但必须声明为 static 类型。**

一个类可以拥有多个方法，在上面的例子中：barking()、hungry() 和 sleeping() 都是 Dog 类的方法。

2.3 构造方法

每个类都有构造方法。如果没有显式地为类定义构造方法，Java 编译器将会为该类提供一个默认构造方法。

在创建一个对象的时候，至少要调用一个构造方法。构造方法的名称必须与类同名，一个类可以有多个构造方法。

下面是一个构造方法示例：

```
public class Puppy{
    public Puppy(){

    }
    public Puppy(String name){
        // 这个构造器仅有一个参数：name
    }
}
```

2.4 创建对象

对象是根据类创建的。在Java中，使用**关键字 new** 来创建一个新的对象。创建对象需要以下三步：

- **声明**：声明一个对象，包括对象名称和对象类型。
- **实例化**：使用关键字 new 来创建一个对象。
- **初始化**：使用 new 创建对象时，会调用构造方法初始化对象。

下面是一个创建对象的例子：

```
public class Puppy{
    public Puppy(String name){
        // 这个构造器仅有一个参数：name
        System.out.println("小狗的名字是："+name);
    }
    public static void main(String[] args){
        Puppy myPuppy = new Puppy("Tom");       // 创建一个
mPuppy对象
    }
}
//       输出结果
小狗的名字是 ： Tom
```

2.5 访问实例变量和方法

通过已创建的对象来访问成员变量和成员方法，如下所示：

```
// 实例化对象
Object refrenceVarible = new Constructor();
// 访问类中变量
referenceVariable.variableName;
// 访问类中方法
referenceVariable.methodName();
```

举个例子：

```
public class Puppy{
    public Puppy(String name){
        // 这个构造器仅有一个参数：name
        System.out.println("小狗的名字是："+name);
    }
    public void setAge(int age){
        puppyAge = age;
    }
    public int getAge(){
        System.out.println("小狗的年龄为："+puppyAge);
        return puppyAge;
    }
    public static void main(String[] args){
        Puppy myPuppy = new Puppy("Tom");       // 创建一个
mPuppy对象
        myPuppy.setAge(10);
        myPuppy.getAge();
        System.out.println("变量值："+myPuppy.puppyAge);
    }
}
```

编译并运行上面的程序，产生如下结果：

```
小狗的名字是 : Tom
小狗的年龄为 : 10
变量值 : 10
```

2.6源文件声明规则

在本节的最后部分，我们将学习源文件的声明规则。当在一个源文件中定义多个类，并且还有import语句和package语句时，要特别注意这些规则。

- 一个源文件中只能有一个 public 类
- 一个源文件可以有多个非 public 类
- **源文件的名称应该和 public 类的类名保持一致。**例如：源文件中 public 类的类名是 Employee，那么源文件应该命名为Employee.java。
- **如果一个类定义在某个包中，那么 package 语句应该在源文件的首行。**
- 如果源文件包含 import 语句，那么应该放在 package 语句和类定义之间。如果没有 package 语句，那么 import 语句应该在源文件中最前面。
- import 语句和 package 语句对源文件中定义的所有类都有效。在同一源文件中，不能给不同的类不同的包声明。

类有若干种访问级别，并且类也分不同的类型：抽象类和 final 类等。这些将在访问控制章节介绍。

除了上面提到的几种类型，Java 还有一些特殊的类，如：内部类、匿名类。

2.7 Java 包

包主要用来对类和接口进行分类。当开发 Java 程序时，可能编写成百上千的类，因此很有必要对类和接口进行分类。

2.8 import 语句

在 Java 中，如果给出一个完整的限定名，包括包名、类名，那么 Java 编译器就可以很容易地定位到源代码或者类。import 语句就是用来提供一个合理的路径，使得编译器可以找到某个类。

例如，下面的命令行将会命令编译器载入 java_installation/java/io 路径下的所有类

```
import java.io.*;
```

2.9 一个简单的例子

在该例子中，我们创建两个类：**Employee** 和 **EmployeeTest**。

首先打开文本编辑器，把下面的代码粘贴进去。注意将文件保存为 Employee.java。

Employee 类有四个成员变量：name、age、designation 和 salary。该类显式声明了一个构造方法，该方法只有一个参数。

```
/************** Employee.java 文件代码********************/
import java.io.*;
public class Employee{
    String name;
    int age;
    String designation;
    double salary;
    // Employee 类的构造器
    Employee(String name){
        this.name = name;
    }
    // 设置age的值
    public void empAge(int empAge){
        age = empAge;
    }
    // 设置designation的值
    public void empDesignation(String empDesig){
        designation = empDesig;
    }
    // 设置salary的值
    public void empSalary(int empSalary){
        salary = empSalary;
```

```
    }
    // 打印信息
    public void printEmployee(){
        System.out.println("名字: "+name);
        System.out.println("年龄:" + age );
        System.out.println("职位:" + designation );
        System.out.println("薪水:" + salary);
    }
}

/**********************************************************************
**********
程序都是从main方法开始执行。为了能运行这个程序，必须包含main方法并且创建一个实例对象。
下面给出EmployeeTest类，该类实例化2个 Employee 类的实例，并调用方法设置变量的值。
将下面的代码保存在 EmployeeTest.java文件中。
**************** EmployeeTest.java 文件代码
*****************************/
import java.io.*;
public class EmployeeTest{
    public static void main(String[] args){
        // 使用构造器创建两个对象
        Employee emp1 = Employee("RUNOOB1");
        Employee emp2 = Employee("RUNOOB2");
        // 调用这两个对象的成员方法
        emp1.empAge(26);
        emp1.empDesignation("高级程序员");
        emp1.empSalary(1000);
        emp1.printEmployee();

        emp2.empAge(21);
        emp2.empDesignation("菜鸟程序员");
        emp2.empSalary(500);
        emp2.printEmployee();
    }
}
/**************** 程序输出结果 ******************
$ javac EmployeeTest.java
$ java EmployeeTest
名字:RUNOOB1
年龄:26
职位:高级程序员
薪水:1000.0
名字:RUNOOB2
年龄:21
职位:菜鸟程序员
薪水:500.0
**************************************************/
```

三、 Java 基本数据类型

变量就是申请内存来存储值。也就是说，当创建变量的时候，需要在内存中申请空间。

内存管理系统根据变量的类型为变量分配存储空间，分配的空间只能用来储存该类型数据。

因此，通过定义不同类型的变量，可以在内存中储存整数、小数或者字符。

Java 的两大数据类型:

- 内置数据类型
- 引用数据类型

3.1 内置数据类型

Java语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。

byte：

- byte 数据类型是8位、有符号的，以二进制补码表示的整数；
- 最小值是 **-128**（**-2^7**）；
- 最大值是 **127**（**2^7-1**）；
- 默认值是 **0**；
- byte 类型用在大型数组中节约空间，主要代替整数，因为 byte 变量占用的空间只有 int 类型的四分之一；
- 例子：byte a = 100，byte b = -50。

short：

- short 数据类型是 16 位、有符号的以二进制补码表示的整数
- 最小值是 **-32768**（**-2^15**）；
- 最大值是 **32767**（**2^15 - 1**）；
- Short 数据类型也可以像 byte 那样节省空间。一个short变量是int型变量所占空间的二分之一；
- 默认值是 **0**；
- 例子：short s = 1000，short r = -20000。

int：

- int 数据类型是32位、有符号的以二进制补码表示的整数；
- 最小值是 **-2,147,483,648**（**-2^31**）；
- 最大值是 **2,147,483,647**（**2^31 - 1**）；
- 一般地整型变量默认为 int 类型；

- 默认值是 **0**；
- 例子：int a = 100000, int b = -200000。

long：

- long 数据类型是 64 位、有符号的以二进制补码表示的整数；
- 最小值是 **-9,223,372,036,854,775,808（-2^63）**；
- 最大值是 **9,223,372,036,854,775,807（2^63 -1）**；
- 这种类型主要使用在需要比较大整数的系统上；
- 默认值是 **0L**；
- 例子： long a = 100000L，Long b = -200000L。
  "L"理论上不分大小写，但是若写成"l"容易与数字"1"混淆，不容易分辨。所以最好大写。

float：

- float 数据类型是单精度、32位、符合IEEE 754标准的浮点数；
- float 在储存大型浮点数组的时候可节省内存空间；
- 默认值是 **0.0f**；
- 浮点数不能用来表示精确的值，如货币；
- 例子：float f1 = 234.5f。

double：

- double 数据类型是双精度、64 位、符合IEEE 754标准的浮点数；
- 浮点数的默认类型为double类型；
- double类型同样不能表示精确的值，如货币；
- 默认值是 **0.0d**；
- 例子：double d1 = 123.4。

boolean：

- boolean数据类型表示一位的信息；
- 只有两个取值：true 和 false；
- 这种类型只作为一种标志来记录 true/false 情况；
- 默认值是 **false**；
- 例子：boolean one = true。

char：

- char类型是一个单一的 16 位 Unicode 字符；
- 最小值是 **\u0000**（即为0）；
- 最大值是 **\uffff**（即为65,535）；
- char 数据类型可以储存任何字符；
- 例子：char letter = 'A';。

实例

对于数值类型的基本类型的取值范围，我们无需强制去记忆，因为它们的值都已经以常量的形式定义在对应的包装类中了。请看下面的例子：

```
public class PrimitiveTypeTest {
    public static void main(String[] args) {
        // byte
        System.out.println("基本类型：byte 二进制位数：" +
Byte.SIZE);
        System.out.println("包装类：java.lang.Byte");
        System.out.println("最小值：Byte.MIN_VALUE=" +
Byte.MIN_VALUE);
        System.out.println("最大值：Byte.MAX_VALUE=" +
Byte.MAX_VALUE);
        System.out.println();

        // short
        System.out.println("基本类型：short 二进制位数：" +
Short.SIZE);
        System.out.println("包装类：java.lang.Short");
        System.out.println("最小值：Short.MIN_VALUE=" +
Short.MIN_VALUE);
        System.out.println("最大值：Short.MAX_VALUE=" +
Short.MAX_VALUE);
        System.out.println();

        // int
        System.out.println("基本类型：int 二进制位数：" +
Integer.SIZE);
        System.out.println("包装类：java.lang.Integer");
        System.out.println("最小值：Integer.MIN_VALUE=" +
Integer.MIN_VALUE);
        System.out.println("最大值：Integer.MAX_VALUE=" +
Integer.MAX_VALUE);
        System.out.println();

        // long
        System.out.println("基本类型：long 二进制位数：" +
Long.SIZE);
        System.out.println("包装类：java.lang.Long");
        System.out.println("最小值：Long.MIN_VALUE=" +
Long.MIN_VALUE);
        System.out.println("最大值：Long.MAX_VALUE=" +
Long.MAX_VALUE);
        System.out.println();

        // float
        System.out.println("基本类型：float 二进制位数：" +
Float.SIZE);
        System.out.println("包装类：java.lang.Float");
```

```
34          System.out.println("最小值：Float.MIN_VALUE=" +
    Float.MIN_VALUE);
35          System.out.println("最大值：Float.MAX_VALUE=" +
    Float.MAX_VALUE);
36          System.out.println();
37
38          // double
39          System.out.println("基本类型：double 二进制位数：" +
    Double.SIZE);
40          System.out.println("包装类：java.lang.Double");
41          System.out.println("最小值：Double.MIN_VALUE=" +
    Double.MIN_VALUE);
42          System.out.println("最大值：Double.MAX_VALUE=" +
    Double.MAX_VALUE);
43          System.out.println();
44
45          // char
46          System.out.println("基本类型：char 二进制位数：" +
    Character.SIZE);
47          System.out.println("包装类：java.lang.Character");
48          // 以数值形式而不是字符形式将Character.MIN_VALUE输出到控制台

49          System.out.println("最小值：Character.MIN_VALUE="
50                  + (int) Character.MIN_VALUE);
51          // 以数值形式而不是字符形式将Character.MAX_VALUE输出到控制台

52          System.out.println("最大值：Character.MAX_VALUE="
53                  + (int) Character.MAX_VALUE);
54      }
55  }
56
57  /************************* 运行结果 **************************
58  基本类型：byte 二进制位数：8
59  包装类：java.lang.Byte
60  最小值：Byte.MIN_VALUE=-128
61  最大值：Byte.MAX_VALUE=127
62
63  基本类型：short 二进制位数：16
64  包装类：java.lang.Short
65  最小值：Short.MIN_VALUE=-32768
66  最大值：Short.MAX_VALUE=32767
67
68  基本类型：int 二进制位数：32
69  包装类：java.lang.Integer
70  最小值：Integer.MIN_VALUE=-2147483648
71  最大值：Integer.MAX_VALUE=2147483647
72
73  基本类型：long 二进制位数：64
74  包装类：java.lang.Long
75  最小值：Long.MIN_VALUE=-9223372036854775808
76  最大值：Long.MAX_VALUE=9223372036854775807
77
```

```
78  基本类型：float 二进制位数：32
79  包装类：java.lang.Float
80  最小值：Float.MIN_VALUE=1.4E-45
81  最大值：Float.MAX_VALUE=3.4028235E38
82
83  基本类型：double 二进制位数：64
84  包装类：java.lang.Double
85  最小值：Double.MIN_VALUE=4.9E-324
86  最大值：Double.MAX_VALUE=1.7976931348623157E308
87
88  基本类型：char 二进制位数：16
89  包装类：java.lang.Character
90  最小值：Character.MIN_VALUE=0
91  最大值：Character.MAX_VALUE=65535
92  ********************************************************************/
```

Float和Double的最小值和最大值都是以科学记数法的形式输出的，结尾的"E+数字"表示E之前的数字要乘以10的多少次方。比如3.14E3就是3.14 × 103 =3140，3.14E-3 就是 3.14 x 10-3 =0.00314。

实际上，JAVA中还存在另外一种基本类型 void，它也有对应的包装类 java.lang.Void，不过我们无法直接对它们进行操作。

3.2 类型默认值

下表列出了 Java 各个类型的默认值：

数据类型	默认值
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'u0000'
String (or any object)	null
boolean	false

3.3 引用类型

- 在Java中，引用类型的变量非常类似于C/C++的指针。引用类型指向一个对象，指向对象的变量是引用变量。这些变量在声明时被指定为一个特定的类型，比如 Employee、Puppy 等。变量一旦声明后，类型就不能被改变了。
- 对象、数组都是引用数据类型。
- 所有引用类型的默认值都是null。
- 一个引用变量可以用来引用任何与之兼容的类型。
- 例子：Site site = new Site("Runoob")。

3.4 Java 常量

常量在程序运行时是不能被修改的。

在 Java 中使用 **final 关键字来修饰常量**，声明方式和变量类似：

```
final double PI = 3.1415927;
```

虽然常量名也可以用小写，但为了便于识别，**通常使用大写字母表示常量。**

字面量可以赋给任何内置类型的变量。例如：

```
byte a = 68;
char a = 'A'
```

byte、int、long、和short都可以用十进制、16进制以及8进制的方式来表示。

当使用字面量的时候，前缀 **0** 表示 8 进制，而前缀 **0x** 代表 16 进制, 例如：

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

和其他语言一样，Java的字符串常量也是包含在两个引号之间的字符序列。下面是字符串型字面量的例子：

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

字符串常量和字符常量都可以包含任何Unicode字符。例如：

```
char a = '\u0001';
String a = "\u0001";
```

Java语言支持一些特殊的转义字符序列。

符号	字符含义
\n	换行 (0x0a)
\r	回车 (0x0d)
\f	换页符(0x0c)
\b	退格 (0x08)
\0	空字符 (0x0)
\s	空格 (0x20)
\t	制表符
\"	双引号
\'	单引号

符号	字符含义
\\	反斜杠
\ddd	八进制字符 (ddd)
\uxxxx	16进制Unicode字符 (xxxx)

3.5 自动类型转换

整型、实型（常量）、字符型数据可以混合运算。运算中，不同类型的数据先转化为同一类型，然后进行运算。

转换从低级到高级。

```
低  ------------------------------------>  高
byte,short,char-> int -> long-> float -> double
```

数据类型转换必须满足如下规则：

- a. 不能对boolean类型进行类型转换。
- b. 不能把对象类型转换成不相关类的对象。
- c. 在把容量大的类型转换为容量小的类型时必须使用强制类型转换。
- d. 转换过程中可能导致溢出或损失精度，例如：

```
int i =128;
byte b = (byte)i;
```

因为 byte 类型是 8 位，最大值为127，所以当 int 强制转换为 byte 类型时，值 128 时候就会导致溢出。

- e. **浮点数到整数的转换是通过舍弃小数得到**，而不是四舍五入，例如：

```
(int)23.7 == 23;
(int)-45.89f == -45
```

3.6 自动类型转换

必须满足转换前的数据类型的位数要低于转换后的数据类型，例如: short数据类型的位数为16位，就可以自动转换位数为32的int类型，同样float数据类型的位数为32，可以自动转换为64位的double类型。

实例

```
public class ZiDongLeiZhuan{
    public static void main(String[] args){
        char c1 = 'a';      // 定义一个char类型
        int i1 = c1;        // char类型自动转换为int
        System.out.println("char类型自动转换为int后的值为："+i1);
        char c2 = 'A';
        int i2 = c2;
        System.out.println("char类型自动转换为int后的值为："+i2);
```

```
    }
}

//                          运行结果为
->  char自动类型转换为int后的值等于97
->  char类型和int计算后的值等于66
```

解析：c1 的值为字符 **a** ,查 ASCII 码表可知对应的 int 类型值为 97， A 对应值为 65，所以 **i2=65+1=66**。

3.7 强制类型转换

- \1. 条件是转换的数据类型必须是兼容的。
- \2. 格式：(type)value type是要强制类型转换后的数据类型 实例：

实例：

```
public class QiangZhiZhuanHuan{
    public void main(String[] args){
        int i1 = 123;
        byte b = (byte)i1;          //  强制类型转换为byte
        System.out.println("int类型强制转换为byte后的值等于"+b);
    }
}
//              运行结果
->  int强制类型转换为byte后的值等于123
```

3.8 隐含强制类型转换

- a. 整数的默认类型是 int。
- b. 浮点型不存在这种情况，因为在定义 float 类型时必须在数字后面跟上 F 或者 f。

这一节讲解了 Java 的基本数据类型。下一节将探讨不同的变量类型以及它们的用法。

四、 Java变量类型

在Java语言中，所有的变量在使用前必须声明。声明变量的基本格式如下：

```
type identifier [ = value][, identifier [ = value] ...]
```

格式说明：type为Java数据类型。identifier是变量名。可以使用逗号隔开来声明多个同类型变量。

以下列出了一些变量的声明实例。注意有些包含了初始化过程。

```java
int a, b, c;
int d = 3, e = 4, f = 5;
byte z = 22;
String s = "runoob";
double pi = 3.14159;
char x = 'x';
float f = 1.2f;
```

Java语言支持的变量类型有：

- 类变量：独立于方法之外的变量，用 **static** 修饰。
- 实例变量：独立于方法之外的变量，不过没有 **static** 修饰。
- 局部变量：类的方法中的变量。

实例：

```java
public class Variable{
    static int allClicks = 0;        // 类变量
    String str = "hello world";      // 实例变量
    public void method(){
        int i = 0;                  // 局部变量
    }
}
```

4.1 Java 局部变量

- 局部变量声明在方法、构造方法或者语句块中；
- 局部变量在方法、构造方法、或者语句块被执行的时候创建，当它们执行完成后，变量将会被销毁；
- 访问修饰符不能用于局部变量；
- 局部变量只在声明它的方法、构造方法或者语句块中可见；
- 局部变量是在栈上分配的。
- 局部变量没有默认值，所以局部变量被声明后，必须经过初始化，才可以使用。

实例 1

在以下实例中age是一个局部变量。定义在pupAge()方法中，它的作用域就限制在这个方法中。

```java
package com.runoob.test;
public class Test{
    public void pupAge(){
        int age = 0;
        age = age + 7;
        System.out.println("小狗的年龄是：" + age);
    }
    public static void main(String[] args){
        Test test = new Test();
        test.pupAge();
    }
```

```
}

->  小狗的年龄是：7
```

实例2

```
package com.runoob.test;
public class Test{
    public void pupAge(){
        int age;
        age = age + 7;
        System.out.println("小狗的年龄是：" + age);
    }
    public static void main(String[] args){
        Test test = new Test();
        test.pupAge();
    }
}

->  局部变量age没有经过初始化，编译出错
```

4.2 实例变量

- 实例变量声明在一个类中，但在方法、构造方法和语句块之外；
- 当一个对象被实例化之后，每个实例变量的值就跟着确定；
- 实例变量在对象创建的时候创建，在对象被销毁的时候销毁；
- 实例变量的值应该至少被一个方法、构造方法或者语句块引用，使得外部能够通过这些方式获取实例变量信息；
- 实例变量可以声明在使用前或者使用后；
- 访问修饰符可以修饰实例变量；
- 实例变量对于类中的方法、构造方法或者语句块是可见的。**一般情况下应该把实例变量设为私有**。通过使用访问修饰符可以使实例变量对子类可见；
- **实例变量具有默认值**。数值型变量的默认值是0，布尔型变量的默认值是false，引用类型变量的默认值是null。变量的值可以在声明时指定，也可以在构造方法中指定；
- **实例变量可以直接通过变量名访问。但在静态方法以及其他类中，就应该使用完全限定名：ObejectReference.VariableName**。

实例：

```
import java.io.*;
public class Employee{
    public String name;             // 这个实例变量对子类可见
    private double salary;          // 私有变量，仅在该类可见
    public Employee(String empName){
        name = empName;             // 在构造器中对name赋值
    }
    // 设定salary的值
    public void setSalary(double empSalary){
        salary = empSalary;
    }
```

```
    // 打印信息
    public void printEmp(){
        System.out.println("名字 : " + name );
        System.out.println("薪水 : " + salary);
    }

    public static void main(String[] args){
        Employee empOne = new Employee("RUNOOB");
        empOne.setSalary(1000.0);
        empOne.printEmp();
    }
}
//                  运行结果
$ javac Employee.java
$ java Employee
名字 : RUNOOB
薪水 : 1000.0
```

4.3 类变量（静态变量）

- 类变量也称为静态变量，在类中以 static 关键字声明，但必须在方法之外。
- 无论一个类创建了多少个对象，类只拥有类变量的一份拷贝。
- 静态变量除了被声明为常量外很少使用，静态变量是指声明为 public/private，final 和 static 类型的变量。**静态变量初始化后不可改变。**
- **静态变量储存在静态存储区。经常被声明为常量，很少单独使用static声明变量。**
- **静态变量在第一次被访问时创建，在程序结束时销毁。**
- 与实例变量具有相似的可见性。但为了对类的使用者可见，**大多数静态变量声明为public类型。**
- 默认值和实例变量相似。数值型变量默认值是0，布尔型默认值是false，引用类型默认值是null。变量的值可以在声明的时候指定，也可以在构造方法中指定。**此外，静态变量还可以在静态语句块中初始化。**
- 静态变量可以通过：*ClassName.VariableName*的方式访问。
- **类变量被声明为public static final类型时，类变量名称一般建议使用大写字母。如果静态变量不是public和final类型，其命名方式与实例变量以及局部变量的命名方式一致。**

实例：

```java
import java.io.*;
public class Employee{
    private static double salary;           // salary是静态的私有变量
    // DEPARTMENT是一个常量
    public static final String DEPARTMENT = "开发人员";
    public static void main(String[] args){
        salary = 1000;                      // 静态变量在构造器中初始化
        System.out.println(DEPARTMENT+"平均工资: "+salary);
    }
}

-> 开发人员平均工资:1000.0
```

4.4 Java 中静态变量和实例变量区别

- 静态变量属于类，该类不生产对象，通过类名就可以调用静态变量。
- 实例变量属于该类的对象，必须产生该类对象，才能调用实例变量。

在程序运行时的区别：

- 实例变量属于某个对象的属性，必须创建了实例对象，其中的实例变量才会被分配空间，才能使用这个实例变量。
- 静态变量不属于某个实例对象，而是属于类，所以也称为类变量，只要程序加载了类的字节码，不用创建任何实例对象，静态变量就会被分配空间，静态变量就可以被使用了。

总之，实例变量必须创建对象后才可以通过这个对象来使用，静态变量则可以直接使用类名来引用。

例如，对于下面的程序，无论创建多少个实例对象， 永远都只分配了一个 staticInt 变量，并且每创建一个实例对象，这个 staticInt 就会加 1；但是，每创建一个实例对象，就会分配一个 random， 即可能分配多个 random ，并且每个 random 的值都只自加了1次。

```java
public class StaticTest {
    private static int staticInt = 2;
    private int random = 2;

    public StaticTest() {
        staticInt++;
        random++;
        System.out.println("staticInt = "+staticInt+"  random = "+random);
    }

    public static void main(String[] args) {
        StaticTest test = new StaticTest();
        StaticTest test2 = new StaticTest();
    }
}
```

执行以上程序，输出结果为：

```
staticInt = 3  random = 3
staticInt = 4  random = 3
```

4.5 成员变量、局部变量、静态变量的区别

	成员变量	局部变量	静态变量
定义位置	在类中,方法外	方法中,或者方法的形式参数	在类中,方法外
初始化值	有默认初始化值	无,先定义,赋值后才能使用	有默认初始化值
调用方式	对象调用	---	对象调用，类名调用
存储位置	堆中	栈中	方法区
生命周期	与对象共存亡	与方法共存亡	与类共存亡
别名	实例变量	---	类变量

五、 Java修饰符

Java语言提供了很多修饰符，主要分为以下两类：

- 访问修饰符
- 非访问修饰符

修饰符用来定义类、方法或者变量，通常放在语句的最前端。我们通过下面的例子来说明：

```
public class className{
    // ...
    private boolean mFlag;
    static final double weeks = 9.5;
    protected static final int BOXWIDTH = 42;
    public static void main(String[] args){
    // 方法体
    }
}
```

5.1 访问控制修饰符

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

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

我们可以通过以下表来说明访问权限：

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

5.1.1默认访问修饰符-不使用任何关键字

使用默认访问修饰符声明的变量和方法，对同一个包内的类是可见的。接口里的变量都隐式声明为 **public static final**,而接口里的方法默认情况下访问权限为 **public**。

如下例所示，变量和方法的声明可以不使用任何修饰符。

```
String version = "1.5.1";
boolean processOrder(){
    return true;
}
```

5.1.2 私有访问修饰符-private

私有访问修饰符是最严格的访问级别，所以被声明为 **private** 的方法、变量和构造方法只能被所属类访问，并且类和接口不能声明为 **private**。

声明为私有访问类型的变量只能通过类中公共的 getter 方法被外部类访问。

Private 访问修饰符的使用主要用来隐藏类的实现细节和保护类的数据。

下面的类使用了私有访问修饰符：

```
public class Logger{
    private String format;
    public String getFormat(){
        return format;
    }
    public void setFormat(String format){
        this.format = format;
    }
}
```

实例中，Logger 类中的 format 变量为私有变量，所以其他类不能直接得到和设置该变量的值。为了使其他类能够操作该变量，定义了两个 public 方法：getFormat() （返回 format 的值）和 setFormat(String)（设置 format 的值）

5.1.3 公有访问修饰符-public

被声明为 public 的类、方法、构造方法和接口能够被任何其他类访问。

如果几个相互访问的 public 类分布在不同的包中，则需要导入相应 public 类所在的包。由于类的继承性，类所有的公有方法和变量都能被其子类继承。

以下函数使用了公有访问控制：

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

5.1.4 受保护的访问修饰符-protected

protected 需要从以下两个点来分析说明：

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

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

接口及接口的成员变量和成员方法不能声明为 protected。

子类能访问 protected 修饰符声明的方法和变量，这样就能保护不相关的类使用这些方法和变量。

下面的父类使用了 protected 访问修饰符，子类重写了父类的 openSpeaker() 方法。

```
class AudioPlayer{
    protected boolean openSpeaker(Speaker sp){
        // 实现细节
    }
}
class StreamingAudioPlayer extends AudioPlayer{
    protected boolean openSpeaker(Speaker sp){
        // 实现细节
    }
}
```

如果把 openSpeaker() 方法声明为 private，那么除了 AudioPlayer 之外的类将不能访问该方法。

如果把 openSpeaker() 声明为 public，那么所有的类都能够访问该方法。

如果我们只想让该方法对其所在类的子类可见，则将该方法声明为 protected。

5.1.5 访问控制和继承

请注意以下方法继承的规则：

- 父类中声明为 public 的方法在子类中也必须为 public。
- 父类中声明为 protected 的方法在子类中要么声明为 protected，要么声明为 public，不能声明为 private。
- 父类中声明为 private 的方法，不能够被继承。

5.2 非访问修饰符

为了实现一些其他的功能，Java 也提供了许多非访问修饰符。

static 修饰符：用来修饰类方法和类变量。

final 修饰符：用来修饰类、方法和变量，**final 修饰的类不能够被继承，修饰的方法不能被继承类重新定义，修饰的变量为常量，是不可修改的。**

abstract 修饰符：用来创建抽象类和抽象方法。

synchronized 和 volatile 修饰符，主要用于线程的编程。

5.2.1 static 修饰符

- **静态变量：**

  static 关键字用来声明独立于对象的静态变量，无论一个类实例化多少对象，它的静态变量只有一份拷贝。 静态变量也被称为类变量。局部变量不能被声明为 static 变量。

- **静态方法：**

  static 关键字用来声明独立于对象的静态方法。**静态方法不能使用类的非静态变量**。静态方法从参数列表得到数据，然后计算这些数据。

对类变量和方法的访问可以直接使用 **classname.variablename** 和 **classname.methodname** 的方式访问。

如下例所示，static修饰符用来创建类方法和类变量。

```
public class InstanceCounter{
    private static int numInstances = 0;
    protected static int getCount(){
        return numInstances;
    }
    private static void addInstance(){
        numInstances++;
    }
    InstanceCounter(){
        InstanceCounter.addInstance();
    }
    public static void main(String[] args){
```

```
        System.out.println("Strating with:" +
InstanceCounter.getCount() +
                           " instances");
        for(int i = 0; i < 500; i++){
            new InstanceCounter();
        }
        System.out.println("Created " +
InstanceCounter.getCount() + " instances");
    }
}
//                  Output
->  Starting with 0 instances
->  Created 500 instances
```

5.2.2 final 修饰符

final 变量：

final 表示"最后的、最终的"含义，变量一旦赋值后，不能被重新赋值。**被 final 修饰的实例变量必须显式指定初始值。**

final 修饰符通常和 static 修饰符一起使用来创建类常量。

实例：

```
public class Test{
    final int value = 10;
    // 下面是声明常量的实例
    public static final int BOXWIDTH = 6;
    static final String TITLE = "Manager";
    public void changeValue(){
        value = 12;         //  非静态方法无法访问静态类变量，会报错
    }
}
```

final 方法

父类中的 final 方法可以被子类继承，但是**不能被子类重写。**

声明 final 方法的主要目的是防止该方法的内容被修改。

如下所示，使用 final 修饰符声明方法。

```
public class Test{
    public final void changeName(){
        // 方法体
    }
}
```

final 类

final 类不能被继承，没有类能够继承 final 类的任何特性。

```
public final class Test{
    // 类体
}
```

5.2.3 abstract 修饰符

抽象类：

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。

一个类不能同时被 **abstract** 和 **final** 修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。

抽象类可以包含抽象方法和非抽象方法。

实例：

```
abstract class Caravan{
    private double price;
    private String model;
    private String year;
    public abstract void goFast();
    public abstract void changeColor();
}
```

(个人理解： abstract修饰符，抽象这个概念类似于C++中的virtual修饰符。)

抽象方法

抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。

抽象方法不能被声明成 final 和 static。

任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。

如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

抽象方法的声明以分号结尾，例如：**public abstract sample();**。

实例：

```
public abstract class superClass{
    abstract void m();  // 抽象方法
}
class subClass extends superClass{
    // 实现抽象方法
    void m(){
        // ... ...
    }
}
```

5.2.4 synchronized 修饰符

synchronized 关键字声明的方法同一时间只能被一个线程访问。synchronized 修饰符可以应用于四个访问修饰符。

实例：

```
public synchronized void showDetials(){
   // ... ...
}
```

5.2.5 transient 修饰符

序列化的对象包含被 transient 修饰的实例变量时，java 虚拟机(JVM)跳过该特定的变量。

该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

实例：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
//定义一个需要序列化的类
class People implements Serializable{
    String name; //姓名
    transient Integer age; //年龄
    public People(String name,int age){
        this.name = name;
        this.age = age;
    }
    public String toString(){
        return "姓名 = "+name+" ,年龄 = "+age;
    }
}
public class TransientPeople {
```

```
21      public static void main(String[] args) throws
    FileNotFoundException, IOException, ClassNotFoundException {
22          People a = new People("李雷",30);
23          System.out.println(a); //打印对象的值
24          ObjectOutputStream os = new ObjectOutputStream(new
    FileOutputStream("d://people.txt"));
25          os.writeObject(a);//写入文件(序列化)
26          os.close();
27          ObjectInputStream is = new ObjectInputStream(new
    FileInputStream("d://people.txt"));
28          a = (People)is.readObject();//将文件数据转换为对象（反序列
    化）
29          System.out.println(a); // 年龄 数据未定义
30          is.close();
31      }
32  }
33
34  /******************* 输出结果 ****************************/
35  ->  姓名 = 李雷 ,年龄 = 30
36  ->  姓名 = 李雷 ,年龄 = null
```

5.2.6 volatile 修饰符

volatile 修饰的成员变量在每次被线程访问时，都强制从共享内存中重新读取该成员变量的值。而且，当成员变量发生变化时，会强制线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

一个 volatile 对象引用可能是 null。

实例：

```
1   public class MyRunnable implements Runnable{
2       private volatile boolean active;
3       public void run(){
4           active = true;
5           while(avtive){          // 第一行
6               // ...
7           }
8       }
9       public void stop(){
10          avtive = false;         // 第二行
11      }
12  }
```

通常情况下，在一个线程调用 run() 方法（在 Runnable 开启的线程），在另一个线程调用 stop() 方法。 如果 **第一行** 中缓冲区的 active 值被使用，那么在 **第二行** 的 active 值为 false 时循环不会停止。

但是以上代码中我们使用了 volatile 修饰 active，所以该循环会停止。

volatile可以用在任何变量前面，但不能用于final变量前面，因为final型的变量是禁止修改的。

使用的场景之一，单例模式中采用DCL双锁检测（double checked locking）机制，在多线程访问的情况下，可使用volatitle修改，保证多线程下的可见性。缺点是性能有损失，因此单线程情况下不必用此修饰符。

```
class Singleton{
    private volatile static Singleton instance = null;
    private Singleton(){

    }
    public static Singleton getInstance(){
        if(instance==null){
            synchronized(Singleton.class){
                if(instance==null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

六、 Java 运算符

计算机的最基本用途之一就是执行数学运算，作为一门计算机语言，Java也提供了一套丰富的运算符来操纵变量。我们可以把运算符分成以下几组：

- 算术运算符
- 关系运算符
- 位运算符
- 逻辑运算符
- 赋值运算符
- 其他运算符

6.1 算术运算符

算术运算符用在数学表达式中，它们的作用和在数学中的作用一样。下表列出了所有的算术运算符。

表格中的实例假设整数变量A的值为10，变量B的值为20：

操作符	描述	例子
+	加法 - 相加运算符两侧的值	A + B 等于 30
-	减法 - 左操作数减去右操作数	A – B 等于 -10
*	乘法 - 相乘操作符两侧的值	A * B等于200
/	除法 - 左操作数除以右操作数	B / A等于2
%	取余 - 左操作数除以右操作数的余数	B%A等于0
++	自增: 操作数的值增加1	B++ 或 ++B 等于 21（区别详见下文）
--	自减: 操作数的值减少1	B-- 或 --B 等于 19（区别详见下文）

6.2 关系运算符

下表为Java支持的关系运算符

表格中的实例整数变量A的值为10，变量B的值为20：

运算符	描述	例子
==	检查如果两个操作数的值是否相等，如果相等则条件为真。	（A == B）为假。
!=	检查如果两个操作数的值是否相等，如果值不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是那么条件为真。	（A> B）为假。
<	检查左操作数的值是否小于右操作数的值，如果是那么条件为真。	（A <B）为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是那么条件为真。	（A> = B）为假。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是那么条件为真。	（A <= B）为真。

6.3 位运算符

Java定义了位运算符，应用于整数类型(int)，长整型(long)，短整型(short)，字符型(char)，和字节型(byte)等类型。

位运算符作用在所有的位上，并且按位运算。假设a = 60，b = 13;它们的二进制格式表示将如下：

```
A = 0011 1100
B = 0000 1101
-----------------
A&B = 0000 1100
A | B = 0011 1101
A ^ B = 0011 0001
~A= 1100 0011
```

下表列出了位运算符的基本运算，假设整数变量 A 的值为 60 和变量 B 的值为 13：

操作符	描述	例子
&	如果相对应位都是1，则结果为1，否则为0	（A&B），得到12，即0000 1100
|	如果相对应位都是 0，则结果为 0，否则为 1	（A | B）得到61，即0011 1101
^	如果相对应位值相同，则结果为0，否则为1	（A ^ B）得到49，即0011 0001
~	按位取反运算符翻转操作数的每一位，即0变成1，1变成0。	（~A）得到-61，即1100 0011

操作符	描述	例子
<<	按位左移运算符。左操作数按位左移右操作数指定的位数。	A << 2得到240，即1111 0000
>>	按位右移运算符。左操作数按位右移右操作数指定的位数。	A >> 2得到15即 1111
>>>	按位右移补零操作符。左操作数的值按右操作数指定的位数右移，移动得到的空位以零填充。	A>>>2得到15即0000 1111

6.4 逻辑运算符

下表列出了逻辑运算符的基本运算，假设布尔变量A为真，变量B为假

操作符	描述	例子
&&	称为逻辑与运算符。当且仅当两个操作数都为真，条件才为真。	（A && B）为假。
||	称为逻辑或操作符。如果任何两个操作数任何一个为真，条件为真。	（A || B）为真。
!	称为逻辑非运算符。用来反转操作数的逻辑状态。如果条件为true，则逻辑非运算符将得到false。	！（A && B）为真。

短路逻辑运算符

当使用与逻辑运算符时，在两个操作数都为true时，结果才为true，但是当得到第一个操作为false时，其结果就必定是false，这时候就不会再判断第二个操作了。

6.5 赋值运算符

下面是Java语言支持的赋值运算符：

操作符	描述	例子
=	简单的赋值运算符，将右操作数的值赋给左侧操作数	C = A + B将把A + B得到的值赋给C
+ =	加和赋值操作符，它把左操作数和右操作数相加赋值给左操作数	C + = A等价于C = C + A
- =	减和赋值操作符，它把左操作数和右操作数相减赋值给左操作数	C - = A等价于C = C - A
* =	乘和赋值操作符，它把左操作数和右操作数相乘赋值给左操作数	C * = A等价于C = C * A
/ =	除和赋值操作符，它把左操作数和右操作数相除赋值给左操作数	C / = A，C 与 A 同类型时等价于 C = C / A
(%)=	取模和赋值操作符，它把左操作数和右操作数取模后赋值给左操作数	C%= A等价于C = C%A
<< =	左移位赋值运算符	C << = 2等价于C = C << 2

操作符	描述	例子
>> =	右移位赋值运算符	C >> = 2等价于C = C >> 2
&=	按位与赋值运算符	C&= 2等价于C = C&2
^ =	按位异或赋值操作符	C ^ = 2等价于C = C ^ 2
| =	按位或赋值操作符	C | = 2等价于C = C | 2

```
public class Test{
    public static void main(String[] args){
        int a = 10;
        int b = 0;
        b %= a;
        System.out.println("b %= a = " + b);
    }
}
//                  Output
->  b %= a = 10
```

6.6 条件运算符（?:）

条件运算符也被称为三元运算符。该运算符有3个操作数，并且需要判断布尔表达式的值。该运算符的主要是决定哪个值应该赋值给变量。

```
variable x = (expression) ? value if true : value if false
```

6.7 instanceof 运算符

该运算符用于操作对象实例，检查该对象是否是一个特定类型（类类型或接口类型）。

instanceof运算符使用格式如下：

```
( Object reference variable ) instanceof  (class/interface type)
```

如果运算符左侧变量所指的对象，是操作符右侧类或接口(class/interface)的一个对象，那么结果为真。

下面是一个例子：

```
String name = "James";
boolean result = name instanceof String; // 由于 name 是 String 类型，所以返回真
```

如果被比较的对象兼容于右侧类型,该运算符仍然返回true。

看下面的例子：

```
class Vehicle {}

public class Car extends Vehicle {
```

```
    public static void main(String args[]){
        Car c1 = new Car();       // c1 是 Car 类型
        Vehicle v2 = new Car();    // v2 是 Car 类型
        Vehicle v3 = new Vehicle();// v3 是 Vehicle 类型

        //  Car 是 Vehicle类型, Vehicle 不是 Car 类型
        boolean result1 =  c1 instanceof Vehicle;    // true
        boolean result2 =  v2 instanceof Car;        // true
        boolean result3 =  v2 instanceof Vehicle;    // true
        boolean result4 =  v3 instanceof Car;          // false

        System.out.println(result1);
        System.out.println(result2);
        System.out.println(result3);
        System.out.println(result4);
    }
}
```

子类是父类的类型，但父类不是子类的类型。

子类的实例可以声明为父类型，但父类的实例不能声明为子类型。

6.8 Java运算符优先级

当多个运算符出现在一个表达式中，谁先谁后呢？这就涉及到运算符的优先级别的问题。在一个多运算符的表达式中，运算符优先级不同会导致最后得出的结果差别甚大。

例如，（1+3）＋（3+2）*2，这个表达式如果按加号最优先计算，答案就是 18，如果按照乘号最优先，答案则是 14。

再如，x = 7 + 3 * 2;这里x得到13，而不是20，因为乘法运算符比加法运算符有较高的优先级，所以先计算3 * 2得到6，然后再加7。

下表中具有最高优先级的运算符在的表的最上面，最低优先级的在表的底部。

类别	操作符	关联性
后缀	() [] . (点操作符)	左到右
一元	expr++、 expr--	从左到右
一元	++expr、 --expr 、+、 - 、～ 、!	从右到左
乘性	* 、%	左到右
加性	+ 、-	左到右
移位	>>、 >>>、 <<	左到右
关系	>、 >=、 < 、<=	左到右
相等	==、 !=	左到右
按位与	&	左到右
按位异或	^	左到右
按位或	|	左到右
逻辑与	&&	左到右

类别	操作符	关联性
逻辑或	||	左到右
条件	? :	从右到左
赋值	=、+=、-=、*=、/=、%=、>>=、<<=、&=、^=、|=	从右到左
逗号	,	左到右

C + = A 与 **C = C + A** 是有区别的一句话总结： **+=** 运算符既可以实现运算，又不会更改 **s** 的数据类型；而后者，**C** 和 **A** 必须是同一类型，否则无法直接运算。

Java 中的运算符的左右结合性是什么意思

```
a?b:c?d:e       // 从右向左--->先计算c?d:e，再计算 a?b:(c?d:e)
```

& 和 **&&**、| 和 || 的区别

区别 **1**： & 和 | 可以进行位运算，后者不能。

区别 **2**： && 和 || 进行运算时有短路性，前者无。

整数运算：

- 1、如果两个操作数有一个为 long，则结果也为 long。
- 2、没有 long 时，结果为 int。即使操作数全为 short、byte，结果也是 int。

```
public class Main {
    public static void main(String[] args) {
        byte a = 1;
        int b = 2;
        long b2 = 3;
        byte c = a + b;  // 报错
        int c2 = b2 + b;  // 报错
    }
}
```

七、 Java 循环结构

顺序结构的程序语句只能被执行一次。如果您想要同样的操作执行多次,，就需要使用循环结构。

Java中有三种主要的循环结构：

- **while** 循环
- **do...while** 循环
- **for** 循环

7.1 while 循环

while是最基本的循环，它的结构为：

```
while( 布尔表达式 ) {
    //循环内容
}
```

7.2 do...while 循环

对于 while 语句而言，如果不满足条件，则不能进入循环。但有时候我们需要即使不满足条件，也至少执行一次。

do...while 循环和 while 循环相似，不同的是，do...while 循环至少会执行一次。

```
do {
        //代码语句
}while(布尔表达式);
```

注意：布尔表达式在循环体的后面，所以语句块在检测布尔表达式之前已经执行了。 如果布尔表达式的值为 true，则语句块一直执行，直到布尔表达式的值为 false。

7.3 for循环

虽然所有循环结构都可以用 while 或者 do...while表示，但 Java 提供了另一种语句 —— for 循环，使一些循环结构变得更加简单。

for循环执行的次数是在执行前就确定的。语法格式如下：

```
for(初始化; 布尔表达式; 更新) {
    //代码语句
}

// 实例
public class Test{
    public static void main(String[] args){
        for(int i = 0; i < 20; i++){
            System.out.println("value of i is : " + i + '\n');
        }
    }
}
```

关于 for 循环有以下几点说明：

- 最先执行初始化步骤。可以声明一种类型，但**可初始化一个或多个循环控制变量**，也可以是空语句。
- 然后，检测布尔表达式的值。如果为 true，循环体被执行。如果为false，循环终止，开始执行循环体后面的语句。
- 执行一次循环后，更新循环控制变量。
- 再次检测布尔表达式。循环执行上面的过程。

7.4 Java 增强 for 循环

Java5 引入了一种主要用于数组的增强型 for 循环。

Java 增强 for 循环语法格式如下:

```
for(声明语句 : 表达式) {
    //代码句子
}

//  实例
public class Test{
    public static void main(String[] args){
        int [] nums = {1, 2, 3, 4, 5};
        for (int x : nums){
            System.out.print(x + ", ");
        }
        System.out.print('\n');
        String[] names = {"James", "Larry", "Kobe", "Jordan"};
        for (String it : names){
            System.out.print(it + ", ");
        }
        System.out.print('\n');
    }
}
```

声明语句：声明新的局部变量，该变量的类型必须和数组元素的类型匹配。其作用域限定在循环语句块，其值与此时数组元素的值相等。

表达式：表达式是要访问的数组名，或者是返回值为数组的方法。

7.5 break 关键字

break 主要用在循环语句或者 switch 语句中，用来跳出整个语句块。

break 跳出最里层的循环，并且继续执行该循环下面的语句。

语法

break 的用法很简单，就是循环结构中的一条语句：

```
break;

// 实例
public class Test {
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         // x 等于 30 时跳出循环
```

```
            if( x == 30 ) {
               break;
            }
            System.out.print( x );
            System.out.print("\n");
         }
      }
}
以上实例编译运行结果如下：
10
20
```

7.6 continue 关键字

continue 适用于任何循环控制结构中。作用是让程序立刻跳转到下一次循环的迭代。

在 for 循环中，continue 语句使程序立即跳转到更新语句。

在 while 或者 do...while 循环中，程序立即跳转到布尔表达式的判断语句。

```
public class Test {
   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         if( x == 30 ) {
        continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
以上实例编译运行结果如下：

10
20
40
50
```

八、 Java条件语句

8.1 Java 条件语句 - if...else

一个 if 语句包含一个布尔表达式和一条或多条语句。

语法

if 语句的语法如下：

```
if(布尔表达式1) {
    //如果布尔表达式1为true将执行的语句
}else if(布尔表达式2){
    //如果布尔表达式2为true将执行的语句
}else{
    //如果以上布尔表达式的值都为false
}
```

8.2 嵌套的 if...else 语句

使用嵌套的 if...else 语句是合法的。也就是说你可以在另一个 if 或者 else if 语句中使用 if 或者 else if 语句。

语法

嵌套的 if...else 语法格式如下：

```
if(布尔表达式 1){
    ////如果布尔表达式 1的值为true执行代码
    if(布尔表达式 2){
        ////如果布尔表达式 2的值为true执行代码
    }
}
```

九、 Java switch case语句

switch case 语句判断一个变量与一系列值中某个值是否相等，每个值称为一个分支。

语法

switch case 语句语法格式如下：

```
switch(expression){
    case value :
        //语句
        break;          //可选
    case value :
        //语句
        break;          //可选
    //你可以有任意数量的case语句
    default : //可选
        //语句
}
```

switch case 语句有如下规则：

- switch 语句中的变量类型可以是： byte、short、int 或者 char。从 Java SE 7 开始，switch 支持字符串 String 类型了，同时 case 标签必须为字符串常量或字面量。
- switch 语句可以拥有多个 case 语句。每个 case 后面跟一个要比较的值和冒号。
- case 语句中的值的数据类型必须与变量的数据类型相同，而且只能是常量或者字面常量。
- 当变量的值与 case 语句的值相等时，那么 case 语句之后的语句开始执行，直到 break 语句出现才会跳出 switch 语句。
- **当遇到 break 语句时，switch 语句终止。程序跳转到 switch 语句后面的语句执行。**case 语句不必须要包含 break 语句。如果没有 break 语句出现，程序会继续执行下一条 case 语句，直到出现 break 语句。
- switch 语句可以包含一个 default 分支，该分支一般是 switch 语句的最后一个分支（可以在任何位置，但建议在最后一个）。default 在没有 case 语句的值和变量值相等的时候执行。**default 分支不需要 break 语句。**

switch case 执行时，一定会先进行匹配，匹配成功返回当前 case 的值，再根据是否有 break，判断是否继续输出，或是跳出判断

如果 case 语句块中没有 break 语句时，匹配成功后，从当前 case 开始，后续所有 case 的值都会输出。

```
public class Test {
   public static void main(String args[]){
      int i = 1;
      switch(i){
         case 0:
            System.out.println("0");
         case 1:
            System.out.println("1");
         case 2:
            System.out.println("2");
         default:
            System.out.println("default");
      }
   }
}
以上代码编译运行结果如下：

1
2
default
```

如果当前匹配成功的 case 语句块没有 break 语句，则从当前 case 开始，后续所有 case 的值都会输出，如果后续的 case 语句块有 break 语句则会跳出判断。

```
public class Test {
   public static void main(String args[]){
      int i = 1;
      switch(i){
         case 0:
            System.out.println("0");
         case 1:
```

```
            System.out.println("1");
         case 2:
            System.out.println("2");
         case 3:
            System.out.println("3"); break;
         default:
            System.out.println("default");
      }
   }
}
以上代码编译运行结果如下：

1
2
3
```

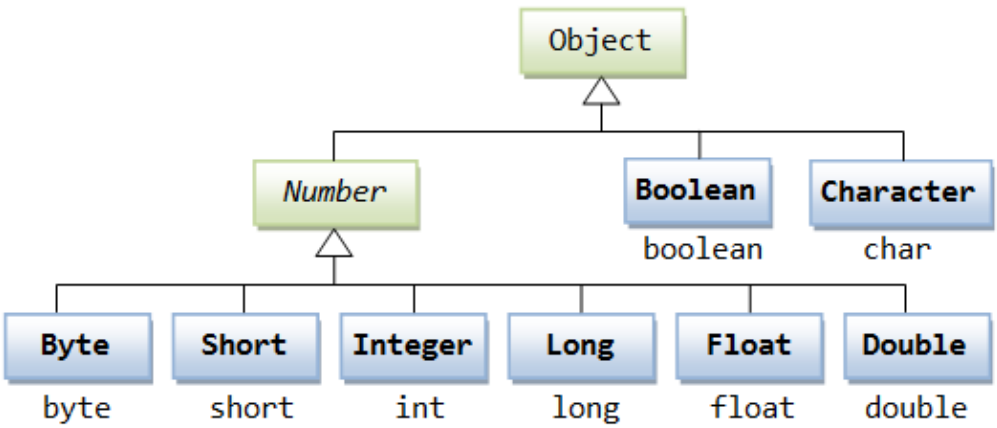
十、 Java Number & Math 类

10.1 Number类

一般地，当需要使用数字的时候，我们通常使用内置数据类型，如：**byte**、**int**、**long**、**double** 等。然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情形。为了解决这个问题，Java 语言为每一个内置数据类型提供了对应的包装类。

所有的包装类（**Integer**、**Long**、**Byte**、**Double**、**Float**、**Short**）都是抽象类 Number 的子类。

包装类	基本数据类型
Boolean	boolean
Byte	byte
Short	short
Integer	int
Long	long
Character	char
Float	float
Double	double



这种由编译器特别支持的包装称为装箱，所以当内置数据类型被当作对象使用的时候，编译器会把内置类型装箱为包装类。相似的，编译器也可以把一个对象拆箱为内置类型。
Number 类属于 java.lang 包。

下面是一个使用 Integer 对象的实例：

```
public class Test{
    public static void main(String[] args){
        Integer x = 5;
        x = x + 10;
        System.out.println(x);
    }
}

->  15
```

10.2 Math 类

Java 的 Math 包含了用于执行基本数学运算的属性和方法，如初等指数、对数、平方根和三角函数。

Math 的方法都被定义为 static 形式，通过 Math 类可以在主函数中直接调用。

```
public class Test{
    public static void main(String[] args){
        System.out.println("90 度的正弦值：" +
Math.sin(Math.PI/2));
        System.out.println("0度的余弦值：" + Math.cos(0));
        System.out.println("60度的正切值：" +
Math.tan(Math.PI/3));
        System.out.println("1的反正切值：  " + Math.atan(1));
        System.out.println("π/2的角度值：" +
Math.toDegrees(Math.PI/2));
        System.out.println(Math.PI);
    }
}
以上实例编译运行结果如下：

90 度的正弦值：1.0
0度的余弦值：1.0
60度的正切值：1.7320508075688767
1的反正切值：  0.7853981633974483
π/2的角度值：90.0
3.141592653589793
```

10.3 Number & Math 类方法

下面的表中列出的是 Number & Math 类常用的一些方法：

序号	方法与描述

序号	方法与描述
1	xxxValue() 将 Number 对象转换为xxx数据类型的值并返回。
2	compareTo() 将number对象与参数比较。
3	equals() 判断number对象是否与参数相等。
4	valueOf() 返回一个 Number 对象指定的内置数据类型
5	toString() 以字符串形式返回值。
6	parseInt() 将字符串解析为int类型。
7	abs() 返回参数的绝对值。
8	ceil() 返回大于等于(>=)给定参数的的最小整数，类型为双精度浮点型。
9	floor() 返回小于等于（<=）给定参数的最大整数 。
10	rint() 返回与参数最接近的整数。返回类型为double。
11	round() 它表示四舍五入，算法为 **Math.floor(x+0.5)**，即将原来的数字加上 0.5 后再向下取整，所以，Math.round(11.5) 的结果为12，Math.round(-11.5) 的结果为-11。
12	min() 返回两个参数中的最小值。
13	max() 返回两个参数中的最大值。
14	exp() 返回自然数底数e的参数次方。
15	log() 返回参数的自然数底数的对数值。
16	pow() 返回第一个参数的第二个参数次方。
17	sqrt() 求参数的算术平方根。
18	sin() 求指定double类型参数的正弦值。
19	cos() 求指定double类型参数的余弦值。
20	tan() 求指定double类型参数的正切值。
21	asin() 求指定double类型参数的反正弦值。
22	acos() 求指定double类型参数的反余弦值。
23	atan() 求指定double类型参数的反正切值。
24	atan2() 将笛卡尔坐标转换为极坐标，并返回极坐标的角度值。
25	toDegrees() 将参数转化为角度。
26	toRadians() 将角度转换为弧度。
27	random() 返回一个随机数。

十一、 Java Character 类

Character 类用于对单个字符进行操作。
Character 类在对象中包装一个基本类型 **char** 的值

```
char ch = 'a';

// Unicode 字符表示形式
char uniChar = '\u039A';

// 字符数组
char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };
```

然而，在实际开发过程中，我们经常会遇到需要使用对象，而不是内置数据类型的情况。为了解决这个问题，Java语言为内置数据类型char提供了包装类Character类。

Character类提供了一系列方法来操纵字符。你可以使用Character的构造方法创建一个Character类对象，例如：

```
Character ch = new Character('a');
```

在某些情况下，Java编译器会自动创建一个Character对象。

例如，将一个char类型的参数传递给需要一个Character类型参数的方法时，那么编译器会自动地将char类型参数转换为Character对象。 这种特征称为装箱，反过来称为拆箱。

```
// 原始字符 'a' 装箱到 Character 对象 ch 中
Character ch = 'a';

// 原始字符 'x' 用 test 方法装箱
// 返回拆箱的值到 'c'
char c = test('x');
```

转义序列

前面有反斜杠（\）的字符代表转义字符，它对编译器来说是有特殊含义的。

下面列表展示了Java的转义序列：

转义序列	描述
\t	在文中该处插入一个tab键
\b	在文中该处插入一个后退键
\n	在文中该处换行
\r	在文中该处插入回车
\f	在文中该处插入换页符
'	在文中该处插入单引号
"	在文中该处插入双引号
\	在文中该处插入反斜杠

Character 方法

下面是Character类的方法：

序号	方法与描述
1	isLetter() 是否是一个字母
2	isDigit() 是否是一个数字字符
3	isWhitespace() 是否是一个空白字符
4	isUpperCase() 是否是大写字母
5	isLowerCase() 是否是小写字母
6	toUpperCase() 指定字母的大写形式

序号	方法与描述
7	toLowerCase() 指定字母的小写形式
8	toString() 返回字符的字符串形式，字符串的长度仅为1

提取字符中的大小写字母：

```
public class Test{
    public static void main(String[] args){
        String strA = "I am a Student, and I love Kobe Bryant";
        String strB = "";
        String strC = "";
        for (int i = 0; i < strA.length(); i++){
            if(Character.isUpperCase(strA.charAt(i)))
                strB += strA.charAt(i) + " ";
            if(Character.isLowerCase(strA.charAt(i)))
                strC += strA.charAt(i) + " ";
        }
        System.out.println("大写字母有: " + strB);
        System.out.println("小写字母有: " + strC);
    }
}
```

十二、 Java String类

字符串广泛应用 在 Java 编程中，在 Java 中字符串属于对象，Java 提供了 String 类来创建和操作字符串。

12.1 创建字符串

创建字符串最简单的方式如下:

```
String greeting = "菜鸟教程";
```

在代码中遇到字符串常量时，这里的值是 "菜鸟教程"，编译器会使用该值创建一个 String 对象。和其它对象一样，可以使用关键字和构造方法来创建 String 对象。String 类有 11 种构造方法，这些方法提供不同的参数来初始化字符串，比如提供一个字符数组参数:

```
public class StringDemo{
    public static void main(String []args){
        char[] helloArray = {'r', 'u', 'n', 'o', 'o', 'b'};
        String helloString = new String(helloArray);
        System.out.println(helloString);
    }
}
以上实例编译运行结果如下:
->  runoob
```

注意:String 类是不可改变的，所以你一旦创建了 String 对象，那它的值就无法改变了（详看笔记部分解析）。

如果需要对字符串做很多修改，那么应该选择使用 StringBuffer & StringBuilder 类。

12.2 字符串长度

用于获取有关对象的信息的方法称为访问器方法。

String 类的一个访问器方法是 length() 方法，它返回字符串对象包含的字符数。

```
public class StringDemo{
    public static void main(String[] args){
        String str = "www.baidu.com";
        int len = str.length();
        System.out.println("String length is: " + len);
    }
}
以上实例编译运行结果如下：

菜鸟教程网址长度 : 13
```

12.3 连接字符串

String 类提供了连接两个字符串的方法：

string1.concat(string2);

返回 string2 连接 string1 的新字符串。也可以对字符串常量使用 concat() 方法，如：

```
"我的名字是 ".concat("Runoob");
```

更常用的是使用 '+' 操作符来连接字符串，如：

```
"Hello," + " runoob" + "!"
```

结果如下：

```
"Hello, runoob!"
```

12.4 创建格式化字符串

我们知道输出格式化数字可以使用 printf() 和 format() 方法。

String 类使用静态方法 format() 返回一个String 对象而不是 PrintStream 对象。

String 类的静态方法 format() 能用来创建可复用的格式化字符串，而不仅仅是用于一次打印输出。

如下所示：

```
System.out.printf("浮点型变量的值为 " +
                  "%f, 整型变量的值为 " +
                  " %d, 字符串变量的值为 " +
                  "is %s", floatVar, intVar, stringVar);
String fs;
fs = String.format("浮点型变量的值为 " +
                   "%f, 整型变量的值为 " +
                   " %d, 字符串变量的值为 " +
                   " %s", floatVar, intVar, stringVar);
```

12.5 String 方法

下面是 String 类支持的方法，更多详细，参看 Java String API 文档:

SN(序号)	方法描述
1	char charAt(int index) 返回指定索引处的 char 值。
2	int compareTo(Object o) 把这个字符串和另一个对象比较。 如果参数字符串等于此字符串，则返回值 0； 如果此字符串小于字符串参数，则返回一个小于 0 的值； 如果此字符串大于字符串参数，则返回一个大于 0 的值。
3	int compareTo(String anotherString) 按字典顺序比较两个字符串。
4	int compareToIgnoreCase(String str) 按字典顺序比较两个字符串，不考虑大小写。
5	String concat(String str) 将指定字符串连接到此字符串的结尾。
6	boolean contentEquals(StringBuffer sb) 当且仅当字符串与指定的StringBuffer有相同顺序的字符时候返回真。
7	static String copyValueOf (char[ ] data) 返回指定数组中表示该字符序列的 String。
8	static String copyValueOf(char[ ] data, int offset, int count) 返回指定数组中表示该字符序列的 String。
9	boolean endsWith(String suffix) 测试此字符串是否以指定的后缀结束。
10	boolean equals(Object anObject) 将此字符串与指定的对象比较。
11	boolean equalsIgnoreCase(String anotherString) 将此 String 与另一个 String 比较，不考虑大小写。
12	[byte] getBytes() 使用平台的默认字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
13	[byte] getBytes(String charsetName) 使用指定的字符集将此 String 编码为 byte 序列，并将结果存储到一个新的 byte 数组中。
14	[void getChars(int srcBegin, int srcEnd, char] dst, int dstBegin) 将字符从此字符串复制到目标字符数组。
15	int hashCode() 返回此字符串的哈希码。
16	int indexOf(int ch) 返回指定字符在此字符串中第一次出现处的索引。
17	int indexOf(int ch, int fromIndex) 返回在此字符串中第一次出现指定字符处的索引，从指定的索引开始搜索。
18	int indexOf(String str) 返回指定子字符串在此字符串中第一次出现处的索引。

SN(序号)	方法描述
19	int indexOf(String str, int fromIndex) 返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始。
20	String intern() 返回字符串对象的规范化表示形式。
21	int lastIndexOf(int ch) 返回指定字符在此字符串中最后一次出现处的索引。
22	int lastIndexOf(int ch, int fromIndex) 返回指定字符在此字符串中最后一次出现处的索引，从指定的索引处开始进行反向搜索。
23	int lastIndexOf(String str) 返回指定子字符串在此字符串中最右边出现处的索引。
24	int lastIndexOf(String str, int fromIndex) 返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。
25	int length() 返回此字符串的长度。
26	boolean matches(String regex) 告知此字符串是否匹配给定的正则表达式。
27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) 测试两个字符串区域是否相等。
28	boolean regionMatches(int toffset, String other, int ooffset, int len) 测试两个字符串区域是否相等。
29	String replace(char oldChar, char newChar) 返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。
30	String replaceAll(String regex, String replacement) 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。
31	String replaceFirst(String regex, String replacement) 使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。
32	[String] split(String regex) 根据给定正则表达式的匹配拆分此字符串。
33	[String] split(String regex, int limit) 根据匹配给定的正则表达式来拆分此字符串。
34	boolean startsWith(String prefix) 测试此字符串是否以指定的前缀开始。
35	boolean startsWith(String prefix, int toffset) 测试此字符串从指定索引开始的子字符串是否以指定前缀开始。
36	CharSequence subSequence(int beginIndex, int endIndex) 返回一个新的字符序列，它是此序列的一个子序列。
37	String substring(int beginIndex) 返回一个新的字符串，它是此字符串的一个子字符串。
38	String substring(int beginIndex, int endIndex) 返回一个新字符串，它是此字符串的一个子字符串。
39	[char] toCharArray() 将此字符串转换为一个新的字符数组。
40	String toLowerCase() 使用默认语言环境的规则将此 String 中的所有字符都转换为小写。
41	String toLowerCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为小写。
42	String toString() 返回此对象本身（它已经是一个字符串！）。
43	String toUpperCase() 使用默认语言环境的规则将此 String 中的所有字符都转换为大写。
44	String toUpperCase(Locale locale) 使用给定 Locale 的规则将此 String 中的所有字符都转换为大写。
45	String trim() 返回字符串的副本，忽略前导空白和尾部空白。

SN(序号)	方法描述
46	static String valueOf(primitive data type x) 返回给定data type类型x参数的字符串表示形式。
47	contains(CharSequence chars) 判断是否包含指定的字符系列。
48	isEmpty() 判断字符串是否为空。

十三、 Java StringBuffer类和StringBuilder类

当对字符串进行修改的时候，需要使用 StringBuffer 和 StringBuilder 类。
和 String 类不同的是，StringBuffer 和 StringBuilder 类的对象能够被多次的修改，并且不产生新的未使用对象。
StringBuilder 类在 Java 5 中被提出，它和 StringBuffer 之间的最大不同在于 StringBuilder 的方法不是线程安全的（不能同步访问）。
由于 StringBuilder 相较于 StringBuffer 有速度优势，所以多数情况下建议使用 StringBuilder 类。然而在应用程序要求线程安全的情况下，则必须使用 StringBuffer 类。

```
1  public class Test {
2      public static void main(String args[]) {
3          StringBuffer sBuffer = new StringBuffer("Cai Niao Jiao
   Cheng->");
4          sBuffer.append("www");
5          sBuffer.append(".runoob");
6          sBuffer.append(".com");
7          System.out.println(sBuffer);
8      }
9  }
```

StringBuffer 方法

以下是 StringBuffer 类支持的主要方法：

序号	方法描述
1	public StringBuffer append(String s) 将指定的字符串追加到此字符序列。
2	public StringBuffer reverse() 将此字符序列用其反转形式取代。
3	public delete(int start, int end) 移除此序列的子字符串中的字符。
4	public insert(int offset, int i) 将 `int` 参数的字符串表示形式插入此序列中。
5	replace(int start, int end, String str) 使用给定 `String` 中的字符替换此序列的子字符串中的字符。

下面的列表里的方法和 String 类的方法类似：

序号	方法描述
1	int capacity() 返回当前容量。
2	char charAt(int index) 返回此序列中指定索引处的 `char` 值。
3	void ensureCapacity(int minimumCapacity) 确保容量至少等于指定的最小值。

序号	方法描述
4	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 将字符从此序列复制到目标字符数组 `dst`。
5	int indexOf(String str) 返回第一次出现的指定子字符串在该字符串中的索引。
6	int indexOf(String str, int fromIndex) 从指定的索引处开始，返回第一次出现的指定子字符串在该字符串中的索引。
7	int lastIndexOf(String str) 返回最右边出现的指定子字符串在此字符串中的索引。
8	int lastIndexOf(String str, int fromIndex) 返回 String 对象中子字符串最后出现的位置。
9	int length() 返回长度（字符数）。
10	void setCharAt(int index, char ch) 将给定索引处的字符设置为 `ch`。
11	void setLength(int newLength) 设置字符序列的长度。
12	CharSequence subSequence(int start, int end) 返回一个新的字符序列，该字符序列是此序列的子序列。
13	String substring(int start) 返回一个新的 `String`，它包含此字符序列当前所包含的字符子序列。
14	String substring(int start, int end) 返回一个新的 `String`，它包含此序列当前所包含的字符子序列。
15	String toString() 返回此序列中数据的字符串表示形式。

笔记

1、

String 长度大小不可变

StringBuffer 和 StringBuilder 长度可变

StringBuffer 线程安全 StringBuilder 线程不安全

StringBuilder 速度快

2、**JAVA 中的 StringBuilder 和 StringBuffer 适用的场景是什么？**

最简单的回答是，stringbuffer 基本没有适用场景，你应该在所有的情况下选择使用 stringbuiler，除非你真的遇到了一个需要线程安全的场景，如果遇到了，请务必在这里留言通知我。

然后，补充一点，关于线程安全，即使你真的遇到了这样的场景，很不幸的是，恐怕你仍然有 99.99....99% 的情况下没有必要选择 stringbuffer，因为 stringbuffer 的线程安全，仅仅是保证 jvm 不抛出异常顺利的往下执行而已，它可不保证逻辑正确和调用顺序正确。大多数时候，我们需要的不仅仅是线程安全，而是锁。

最后，为什么会有 stringbuffer 的存在，如果真的没有价值，为什么 jdk 会提供这个类？答案太简单了，因为最早是没有 stringbuilder 的，sun 的人不知处于何种愚蠢的考虑，决定让 stringbuffer 是线程安全的，然后大约 10 年之后，人们终于意识到这是一个多么愚蠢的决定，意识到在这 10 年之中这个愚蠢的决定为 java 运行速度慢这样的流言贡献了多大的力量，于是，在 jdk1.5 的时候，终于决定提供一个非线程安全的 stringbuffer 实现，并命名为 stringbuilder。顺便，javac 好像大概也是从这个版本开始，把所有用加号连接的 string 运

算都隐式的改写成 stringbuilder，也就是说，从 jdk1.5 开始，用加号拼接字符串已经没有任何性能损失了。

如诸多评论所指出的，我上面说，"用加号拼接字符串已经没有任何性能损失了"并不严谨，严格的说，如果没有循环的情况下，单行用加号拼接字符串是没有性能损失的，java 编译器会隐式的替换成 stringbuilder，但在有循环的情况下，编译器没法做到足够智能的替换，仍然会有不必要的性能损耗，因此，用循环拼接字符串的时候，还是老老实实的用 stringbuilder 吧。

十四、 Java数组

数组对于每一门编程语言来说都是重要的数据结构之一，当然不同语言对数组的实现及处理也不尽相同。
Java 语言中提供的数组是用来存储固定大小的同类型元素。
你可以声明一个数组变量，如 numbers[100] 来代替直接声明 100 个独立变量 number0，number1，....，number99。
本教程将为大家介绍 Java 数组的声明、创建和初始化，并给出其对应的代码。

14.1 声明数组变量

首先必须声明数组变量，才能在程序中使用数组。下面是声明数组变量的语法：

```
dataType[] arrayRefVar;          // 首选方法
//或
dataType arratRefVar[];          // 效果相同，但不是首选方法

// 举个例子
double[] num;
int numm[];
```

注意：建议使用 **dataType[] arrayRefVar** 的声明风格声明数组变量。 dataType arrayRefVar[] 风格是来自 C/C++ 语言 ，在Java中采用是为了让 C/C++ 程序员能够快速理解java语言。

14.2 创建数组

Java语言使用new操作符来创建数组，语法如下：

```
arrayRefVar = new dataType[arraySize];
```

上面的语法语句做了两件事：

- 一、使用 dataType[arraySize] 创建了一个数组。
- 二、把新创建的数组的引用赋值给变量 arrayRefVar。

数组变量的声明，和创建数组可以用一条语句完成，如下所示：

```
dataType[] arrayRefVar = new dataType[arraySize];
```

另外，你还可以使用如下的方式创建数组。

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

数组的元素是通过索引访问的。数组索引从 0 开始，所以索引值从 0 到 arrayRefVar.length-1。

14.3 处理数组

数组的元素类型和数组的大小都是确定的，所以当处理数组元素时候，我们通常使用基本循环或者 For-Each 循环。

示例

该实例完整地展示了如何创建、初始化和操纵数组：

```
public class Test{
    // 数组作为函数参数
    public static void printArray(double[] array){
        for (double i : array){
            System.out.print(i + " ");
        }
        System.out.println();
    }
    // 数组作为函数返回值
    public static double[] createArray(int size){
        return new double[size];
    }
    public static void main(String []args){
        double[] a = {1.2, 3.2, 4, 5.2, 5.7};
        // Print all data
        printArray(a);
        // Caculate the sum of a
        double sum = 0.0;
        for (double x : a){
            sum += x;
        }
        System.out.println("The sum of a is: " + sum);
        // Find the max
        double max = a[0];
        for (int i = 0; i < a.length; i++){
            if (a[i] > max){
                max = a[i];
            }
        }
        System.out.println("Max is: " + max);
        // 通过createArray函数创建新数组
        double[] arr = createArray(10);
        System.out.println("The new array is: " + arr);
        System.out.println("The size of new array: " +
arr.length);
```

```
    }
}
```

14.4 多维数组

多维数组可以看成是数组的数组，比如二维数组就是一个特殊的一维数组，其每一个元素都是一个一维数组，例如：

```
String str[][] = new String[3][4];
```

多维数组的动态初始化（以二维数组为例）

1. 直接为每一维分配空间，格式如下：
2. 从最高维开始，分别为每一维分配空间，例如：

```
type[][] typeName = new type[typeLength1][typeLength2];

// 示例1
int a[][] = new int[2][3];
// 示例2
String[][] s = new String[2][];
s[0] = new String[2];
s[1] = new String[3];
s[0][0] = new String("Good");
s[0][1] = new String("Luck");
s[1][0] = new String("to");
s[1][1] = new String("you");
s[1][2] = new String("!");
```

多维数组的引用（以二维数组为例）

对二维数组中的每个元素，引用方式为 **arrayName[index1][index2]**，例如：

```
num[1][0]
```

14.5 Arrays 类

java.util.Arrays 类能方便地操作数组，它提供的所有方法都是静态的。

具有以下功能：

- 给数组赋值：通过 fill 方法。
- 对数组排序：通过 sort 方法,按升序。
- 比较数组：通过 equals 方法比较数组中元素值是否相等。
- 查找数组元素：通过 binarySearch 方法能对排序好的数组进行二分查找法操作。

具体说明请查看下表：

序号	方法和说明

序号	方法和说明
1	**public static int binarySearch(Object[] a, Object key)** 用二分查找算法在给定数组中搜索给定值的对象(Byte,Int,double等)。数组在调用前必须排序好的。如果查找值包含在数组中，则返回搜索键的索引；否则返回 (-(*插入点*) - 1)。
2	**public static boolean equals(long[] a, long[] a2)** 如果两个指定的 long 型数组彼此*相等*，则返回 true。如果两个数组包含相同数量的元素，并且两个数组中的所有相应元素对都是相等的，则认为这两个数组是相等的。换句话说，如果两个数组以相同顺序包含相同的元素，则两个数组是相等的。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。
3	**public static void fill(int[] a, int val)** 将指定的 int 值分配给指定 int 型数组指定范围中的每个元素。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。
4	**public static void sort(Object[] a)** 对指定对象数组根据其元素的自然顺序进行升序排列。同样的方法适用于所有的其他基本数据类型（Byte，short，Int等）。

选择排序

```
public class Test{
    public static void selectionSort(int[] num){
        if (num.length==0 || num.length==1)
            return;
        for(int i=0; i<num.length; i++){
            for(int j=i+1; j<num.length; j++){
                if (num[j]<num[i]){
                    int temp = num[i];
                    num[i] = num[j];
                    num[j] = temp;
                }
            }
        }
    }
    public static void main(String[] args){
        int[] arr =
{20,60,51,81,285,12,165,51,81,318,186,9,70};
        selectionSort(arr);
        for(int x : arr)
            System.out.print(x + " ");
    }
}
```

十五、 Java 日期时间

java.util 包提供了 Date 类来封装当前的日期和时间。 Date 类提供两个构造函数来实例化 Date 对象。
第一个构造函数使用当前日期和时间来初始化对象。

```
Date( )
```

第二个构造函数接收一个参数，该参数是从1970年1月1日起的毫秒数。

```
Date(long millisec)
```

Date对象创建以后，可以调用下面的方法。

序号	方法和描述
1	**boolean after(Date date)** 若当调用此方法的Date对象在指定日期之后返回true,否则返回false。
2	**boolean before(Date date)** 若当调用此方法的Date对象在指定日期之前返回true,否则返回false。
3	**Object clone( )** 返回此对象的副本。
4	**int compareTo(Date date)** 比较当调用此方法的Date对象和指定日期。两者相等时候返回0。调用对象在指定日期之前则返回负数。调用对象在指定日期之后则返回正数。
5	**int compareTo(Object obj)** 若obj是Date类型则操作等同于compareTo(Date) 。否则它抛出ClassCastException。
6	**boolean equals(Object date)** 当调用此方法的Date对象和指定日期相等时候返回true,否则返回false。
7	**long getTime( )** 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
8	**int hashCode( )** 返回此对象的哈希码值。
9	**void setTime(long time)** 用自1970年1月1日00:00:00 GMT以后time毫秒数设置时间和日期。
10	**String toString( )** 把此 Date 对象转换为以下形式的 String： dow mon dd hh:mm:ss zzz yyyy 其中： dow 是一周中的某一天 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)。

15.1 获取当前日期时间

Java中获取当前日期和时间很简单，使用 Date 对象的 toString() 方法来打印当前日期和时间，如下所示：

```
import java.util.Date;
public class Test{
    public static void main(String[] args){
        Date dat = new Date();
        System.out.println(dat.toString());
    }
}
```

15.2 日期比较

Java使用以下三种方法来比较两个日期：

- 使用 getTime() 方法获取两个日期（自1970年1月1日经历的毫秒数值），然后比较这两个值。
- 使用方法 before()，after() 和 equals()。例如，一个月的12号比18号早，则 new Date(99, 2, 12).before(new Date (99, 2, 18)) 返回true。

- 使用 compareTo() 方法，它是由 Comparable 接口定义的，Date 类实现了这个接口。

15.3 使用 SimpleDateFormat 格式化日期

SimpleDateFormat 是一个以语言环境敏感的方式来格式化和分析日期的类。
SimpleDateFormat 允许你选择任何用户自定义日期时间格式来运行。例如：

```
1  import java.util.*;
2  import java.text.*;
3
4  public class Test{
5      public static void main(String[] args){
6          Date dat = new Date();
7          SimpleDateFormat ft = new SimpleDateFormat("yyyy-MM-dd
   HH:mm:ss");
8          System.out.println(ft.format(dat));
9      }
10 }
11 //              Output
12 -》  2020-10-03 04:00:22
```

这一行代码确立了转换的格式，其中 yyyy 是完整的公元年，MM 是月份，dd 是日期，HH:mm:ss 是时、分、秒。

注意:有的格式大写，有的格式小写，例如 MM 是月份，mm 是分；HH 是 24 小时制，而 hh 是 12 小时制。

15.4 日期和时间的格式化编码

时间模式字符串用来指定时间格式。在此模式中，所有的 ASCII 字母被保留为模式字母，定义如下：

字母	描述	示例
G	纪元标记	AD
y	四位年份	2001
M	月份	July or 07
d	一个月的日期	10
h	A.M./P.M. (1~12)格式小时	12
H	一天中的小时 (0~23)	22
m	分钟数	30
s	秒数	55
S	毫秒数	234
E	星期几	Tuesday
D	一年中的日子	360
F	一个月中第几周的周几	2 (second Wed. in July)
w	一年中第几周	40

字母	描述	示例
W	一个月中第几周	1
a	A.M./P.M. 标记	PM
k	一天中的小时(1~24)	24
K	A.M./P.M. (0~11)格式小时	10
z	时区	Eastern Standard Time
'	文字定界符	Delimiter
"	单引号	`

15.5 使用printf格式化日期

printf 方法可以很轻松地格式化时间和日期。使用两个字母格式，它以 **%t** 开头并且以下面表格中的一个字母结尾。

转 换 符	说 明	示 例
c	包括全部日期和时间信息	星期六 十月 27 14:21:20 CST 2007
F	"年-月-日"格式	2007-10-27
D	"月/日/年"格式	10/27/07
r	"HH:MM:SS PM"格式（12时制）	02:25:51 下午
T	"HH:MM:SS"格式（24时制）	14:28:16
R	"HH:MM"格式（24时制）	14:28

实例：

```
1  import java.util.Date;
2
3  /**
4   * 使用printf输出
5   */
6  /**关键技术点
7   * 使用java.io.PrintStream的printf方法实现C风格的输出
8   * printf 方法的第一个参数为输出的格式,第二个参数是可变长的,表示待输出的
    数据对象
9   */
10 public class Printf {
11
12         public static void main(String[] args) {
13                 /*** 输出字符串 ***/
14                 // %s表示输出字符串，也就是将后面的字符串替换模式中的%s
15                 System.out.printf("%s", new Integer(1212));
16                 // %n表示换行
17                 System.out.printf("%s%n", "end line");
18                 // 还可以支持多个参数
19                 System.out.printf("%s = %s%n", "Name",
    "Zhangsan");
20                 // %S将字符串以大写形式输出
21                 System.out.printf("%S = %s%n", "Name",
    "Zhangsan");
```

```
22                  // 支持多个参数时，可以在%s之间插入变量编号，1$表示第一个字符串，3$表示第3个字符串
23                  System.out.printf("%1$s = %3$s %2$s%n", "Name", "san", "Zhang");
24
25                  /*** 输出boolean类型 ***/
26                  System.out.printf("true = %b; false = ", true);
27                  System.out.printf("%b%n", false);
28
29                  /*** 输出整数类型***/
30                  Integer iObj = 342;
31                  // %d表示将整数格式化为10进制整数
32                  System.out.printf("%d; %d; %d%n", -500, 2343L, iObj);
33                  // %o表示将整数格式化为8进制整数
34                  System.out.printf("%o; %o; %o%n", -500, 2343L, iObj);
35                  // %x表示将整数格式化为16进制整数
36                  System.out.printf("%x; %x; %x%n", -500, 2343L, iObj);
37                  // %X表示将整数格式化为16进制整数，并且字母变成大写形式
38                  System.out.printf("%X; %X; %X%n", -500, 2343L, iObj);
39
40                  /*** 输出浮点类型***/
41                  Double dObj = 45.6d;
42                  // %e表示以科学技术法输出浮点数
43                  System.out.printf("%e; %e; %e%n", -756.403f, 7464.232641d, dObj);
44                  // %E表示以科学技术法输出浮点数，并且为大写形式
45                  System.out.printf("%E; %E; %E%n", -756.403f, 7464.232641d, dObj);
46                  // %f表示以十进制格式化输出浮点数
47                  System.out.printf("%f; %f; %f%n", -756.403f, 7464.232641d, dObj);
48                  // 还可以限制小数点后的位数
49                  System.out.printf("%.1f; %.3f; %f%n", -756.403f, 7464.232641d, dObj);
50
51                  /*** 输出日期类型***/
52                  // %t表示格式化日期时间类型，%T是时间日期的大写形式，在%t之后用特定的字母表示不同的输出格式
53                  Date date = new Date();
54                  long dataL = date.getTime();
55                  // 格式化年月日
56                  // %t之后用y表示输出日期的年份（2位数的年，如99）
57                  // %t之后用m表示输出日期的月份，%t之后用d表示输出日期的日号
58                  System.out.printf("%1$ty-%1$tm-%1$td; %2$ty-%2$tm-%2$td%n", date, dataL);
59                  // %t之后用Y表示输出日期的年份（4位数的年），
```

```
60                  // %t之后用B表示输出日期的月份的完整名， %t之后用b表示输出日期的月份的简称
61                  System.out.printf("%1$tY-%1$tB-%1$td; %2$tY-%2$tb-%2$td%n", date, dataL);
62
63                  // 以下是常见的日期组合
64                  // %t之后用D表示以 "%tm/%td/%ty"格式化日期
65                  System.out.printf("%1$tD%n", date);
66                  //%t之后用F表示以"%tY-%tm-%td"格式化日期
67                  System.out.printf("%1$tF%n", date);
68
69                  /*** 输出时间类型***/
70                  // 输出时分秒
71                  // %t之后用H表示输出时间的时（24进制），%t之后用I表示输出时间的时（12进制），
72                  // %t之后用M表示输出时间的分，%t之后用S表示输出时间的秒
73                  System.out.printf("%1$tH:%1$tM:%1$tS; %2$tI:%2$tM:%2$tS%n", date, dataL);
74                  // %t之后用L表示输出时间的秒中的毫秒
75                  System.out.printf("%1$tH:%1$tM:%1$tS %1$tL%n", date);
76                  // %t之后p表示输出时间的上午或下午信息
77                  System.out.printf("%1$tH:%1$tM:%1$tS %1$tL %1$tp%n", date);
78
79                  // 以下是常见的时间组合
80                  // %t之后用R表示以"%tH:%tM"格式化时间
81                  System.out.printf("%1$tR%n", date);
82                  // %t之后用T表示以"%tH:%tM:%tS"格式化时间
83                  System.out.printf("%1$tT%n", date);
84                  // %t之后用r表示以"%tI:%tM:%tS %Tp"格式化时间
85                  System.out.printf("%1$tr%n", date);
86
87                  /*** 输出星期***/
88                  // %t之后用A表示得到星期几的全称
89                  System.out.printf("%1$tF %1$tA%n", date);
90                  // %t之后用a表示得到星期几的简称
91                  System.out.printf("%1$tF %1$ta%n", date);
92
93                  // 输出时间日期的完整信息
94                  System.out.printf("%1$tc%n", date);
95          }
96  }
97  /**
98   *printf方法中,格式为"%s"表示以字符串的形式输出第二个可变长参数的第一个参数值;
99   *格式为"%n"表示换行;格式为"%S"表示将字符串以大写形式输出;在"%s"之间用"n$"表示
100  *输出可变长参数的第n个参数值.格式为"%b"表示以布尔值的形式输出第二个可变长参数
101  *的第一个参数值.
102  */
```

```
/**
 * 格式为"%d"表示以十进制整数形式输出;"%o"表示以八进制形式输出;"%x"表示以十六进制
 * 输出;"%X"表示以十六进制输出,并且将字母(A、B、C、D、E、F)换成大写.格式为"%e"表
 * 以科学计数法输出浮点数;格式为"%E"表示以科学计数法输出浮点数,而且将e大写;格式为
 * "%f"表示以十进制浮点数输出,在"%f"之间加上".n"表示输出时保留小数点后面n位.
 */
/**
 * 格式为"%t"表示输出时间日期类型."%t"之后用y表示输出日期的二位数的年份(如99)、用m
 * 表示输出日期的月份,用d表示输出日期的日号;"%t"之后用Y表示输出日期的四位数的年份
 * (如1999)、用B表示输出日期的月份的完整名,用b表示输出日期的月份的简称."%t"之后用D
 * 表示以"%tm/%td/%ty"的格式输出日期、用F表示以"%tY-%tm-%td"的格式输出日期.
 */
/**
 * "%t"之后用H表示输出时间的时(24进制),用I表示输出时间的时(12进制),用M表示输出时间
 * 分,用S表示输出时间的秒,用L表示输出时间的秒中的毫秒数、用 p 表示输出时间的是上午还是
 * 下午."%t"之后用R表示以"%tH:%tM"的格式输出时间、用T表示以"%tH:%tM:%tS"的格式输出
 * 时间、用r表示以"%tI:%tM:%tS %Tp"的格式输出时间.
 */
/**
 * "%t"之后用A表示输出日期的全称,用a表示输出日期的星期简称.
```

15.6 解析字符串为时间

SimpleDateFormat 类有一些附加的方法，特别是parse()，它试图按照给定的 SimpleDateFormat 对象的格式化存储来解析字符串。例如：

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

public class Test{
    public static void main(String args[]){
        SimpleDateFormat ft = new SimpleDateFormat("yyyy-MM-dd");
        String input = args.length==0? "1818-11-11" : args[0];
        System.out.print(input + "Parses as ");
        Date t;
        try{
            t = ft.parse(input);
            System.out.println(t);
        }catch(ParseException e){
```

```
            System.out.println("Unparseable using " + ft );
        }
    }
}
```

15.7 Java 休眠(sleep)

sleep()使当前线程进入停滞状态（阻塞当前线程），让出CPU的使用、目的是不让当前线程独自霸占该进程所获的CPU资源，以留一定时间给其他线程执行的机会。

你可以让程序休眠一毫秒的时间或者到您的计算机的寿命长的任意段时间。例如，下面的程序会休眠3秒：

```
import java.util.*;
public class Test{
    public static void main(String[] args){
        try{
            System.out.println(new Date() + "\n");
            Thread.sleep(1000*3);               // sleep 3
seconds
            System.out.println(new Date() + "\n");
        }catch(Exception e){
            System.out.println("Got an exception!");
        }
    }
}
```

15.8 测量时间

下面的一个例子表明如何测量时间间隔（以毫秒为单位）：

```
import java.util.*;
public class Test{
    public static void maim(String[] args){
        try{
            long start = System.currentTimeMillis();
            System.out.println(new Date() + "\n");
            Thread.sleep(5*60*10);
            System.out.println(new Date() + "\n");
            long end = System.currentTimeMillis();
            long diff = end - start;
            System.out.println("Difference is : "+ diff);
        }catch(Exception e){
            System.out.println("Got an exception");
        }
    }
}
```

15.9 Calendar类

我们现在已经能够格式化并创建一个日期对象了，但是我们如何才能设置和获取日期数据的特定部分呢，比如说小时，日，或者分钟？我们又如何在日期的这些部分加上或者减去值呢？答案是使用Calendar 类。

Calendar类的功能要比Date类强大很多，而且在实现方式上也比Date类要复杂一些。

Calendar类是一个抽象类，在实际使用时实现特定的子类的对象，创建对象的过程对程序员来说是透明的，只需要使用getInstance方法创建即可。

创建一个代表系统当前日期的Calendar对象

```
Calendar c = Calendar.getInstance();//默认是当前日期
```

创建一个指定日期的Calendar对象

使用Calendar类代表特定的时间，需要首先创建一个Calendar的对象，然后再设定该对象中的年月日参数来完成。

```
//创建一个代表2009年6月12日的Calendar对象
Calendar c1 = Calendar.getInstance();
c1.set(2009, 6 - 1, 12);
```

Calendar类对象字段类型

Calendar类中用以下这些常量表示不同的意义，jdk内的很多类其实都是采用的这种思想

常量	描述
Calendar.YEAR	年份
Calendar.MONTH	月份
Calendar.DATE	日期
Calendar.DAY_OF_MONTH	日期，和上面的字段意义完全相同
Calendar.HOUR	12小时制的小时
Calendar.HOUR_OF_DAY	24小时制的小时
Calendar.MINUTE	分钟
Calendar.SECOND	秒
Calendar.DAY_OF_WEEK	星期几

Calendar类对象信息的设置

```
// **********************Set设置，如：
Calendar c = Calendar.getInstance();
// 调用
public final void set(int year, int month, int date);
c.set(2020, 10, 3);
// 利用字段类型设置
// 如果只设定某个字段，例如日期的值，则可以使用如下set方法：
public void set(int field, int value);
```

```
// 把c对象代表的日期设置为29号， 其他所有的数值会被重新计算
c.set(Calendar.DATE, 29);
// 把c对象代表的年份设置为2019年， 其他所有的数值会被重新计算
c.set(Calendar.YEAR, 2019);
// ****************************Add设置
// 把c对象的日期加上10，也就是c表示为10天后的日期，其它所有的数值会被重新计算
c.add(Calendar.DATE, 10);
```

Calendar类对象信息的获得

```
Calendar c = Calendar.getInstance();
// 获得年份
int year = c1.get(Calendar.YEAR);
// 获得月份
int month = c1.get(Calendar.MONTH) + 1;
// 获得日期
int date = c1.get(Calendar.DATE);
// 获得小时
int hour = c1.get(Calendar.HOUR_OF_DAY);
// 获得分钟
int minute = c1.get(Calendar.MINUTE);
// 获得秒
int second = c1.get(Calendar.SECOND);
// 获得星期几（注意（这个与Date类是不同的）：1代表星期日、2代表星期1、3代表星期二，以此类推）
int day = c1.get(Calendar.DAY_OF_WEEK);
```

15.10 GregorianCalendar类

Calendar类实现了公历日历，GregorianCalendar是Calendar类的一个具体实现。

Calendar 的getInstance（）方法返回一个默认用当前的语言环境和时区初始化的GregorianCalendar对象。GregorianCalendar定义了两个字段：AD和BC。这是代表公历定义的两个时代。

下面列出GregorianCalendar对象的几个构造方法：

序号	构造函数和说明
1	**GregorianCalendar()** 在具有默认语言环境的默认时区内使用当前时间构造一个默认的 GregorianCalendar。
2	**GregorianCalendar(int year, int month, int date)** 在具有默认语言环境的默认时区内构造一个带有给定日期设置的 GregorianCalendar
3	**GregorianCalendar(int year, int month, int date, int hour, int minute)** 为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。
4	**GregorianCalendar(int year, int month, int date, int hour, int minute, int second)** 为具有默认语言环境的默认时区构造一个具有给定日期和时间设置的 GregorianCalendar。

序号	构造函数和说明
5	**GregorianCalendar(Locale aLocale)** 在具有给定语言环境的默认时区内构造一个基于当前时间的 GregorianCalendar。
6	**GregorianCalendar(TimeZone zone)** 在具有默认语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。
7	**GregorianCalendar(TimeZone zone, Locale aLocale)** 在具有给定语言环境的给定时区内构造一个基于当前时间的 GregorianCalendar。

这里是GregorianCalendar 类提供的一些有用的方法列表：

序号	方法和说明
1	**void add(int field, int amount)** 根据日历规则，将指定的（有符号的）时间量添加到给定的日历字段中。
2	**protected void computeFields()** 转换UTC毫秒值为时间域值
3	**protected void computeTime()** 覆盖Calendar ，转换时间域值为UTC毫秒值
4	**boolean equals(Object obj)** 比较此 GregorianCalendar 与指定的 Object。
5	**int get(int field)** 获取指定字段的时间值
6	**int getActualMaximum(int field)** 返回当前日期，给定字段的最大值
7	**int getActualMinimum(int field)** 返回当前日期，给定字段的最小值
8	**int getGreatestMinimum(int field)** 返回此 GregorianCalendar 实例给定日历字段的最高的最小值。
9	**Date getGregorianChange()** 获得格里高利历的更改日期。
10	**int getLeastMaximum(int field)** 返回此 GregorianCalendar 实例给定日历字段的最低的最大值
11	**int getMaximum(int field)** 返回此 GregorianCalendar 实例的给定日历字段的最大值。
12	**Date getTime()** 获取日历当前时间。
13	**long getTimeInMillis()** 获取用长整型表示的日历的当前时间
14	**TimeZone getTimeZone()** 获取时区。
15	**int getMinimum(int field)** 返回给定字段的最小值。
16	**int hashCode()** 重写hashCode.
17	**boolean isLeapYear(int year)** 确定给定的年份是否为闰年。
18	**void roll(int field, boolean up)** 在给定的时间字段上添加或减去（上/下）单个时间单元，不更改更大的字段。
19	**void set(int field, int value)** 用给定的值设置时间字段。
20	**void set(int year, int month, int date)** 设置年、月、日的值。
21	**void set(int year, int month, int date, int hour, int minute)** 设置年、月、日、小时、分钟的值。
22	**void set(int year, int month, int date, int hour, int minute, int second)** 设置年、月、日、小时、分钟、秒的值。
23	**void setGregorianChange(Date date)** 设置 GregorianCalendar 的更改日期。
24	**void setTime(Date date)** 用给定的日期设置Calendar的当前时间。

序号	方法和说明
25	**void setTimeInMillis(long millis)** 用给定的long型毫秒数设置Calendar的当前时间。
26	**void setTimeZone(TimeZone value)** 用给定时区值设置当前时区。
27	**String toString()** 返回代表日历的字符串。

实例：

```
import java.util.*;
public class Test {
   public static void main(String args[]) {
      String months[] = {
      "Jan", "Feb", "Mar", "Apr",
      "May", "Jun", "Jul", "Aug",
      "Sep", "Oct", "Nov", "Dec"};
      int year;
      GregorianCalendar gcalendar = new GregorianCalendar();
      System.out.print("Date: ");
      System.out.print(months[gcalendar.get(Calendar.MONTH)]);
      System.out.print(" " + gcalendar.get(Calendar.DATE) + "
");
      System.out.println(year = gcalendar.get(Calendar.YEAR));
      System.out.print("Time: ");
      System.out.print(gcalendar.get(Calendar.HOUR) + ":");
      System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
      System.out.println(gcalendar.get(Calendar.SECOND));
      if(gcalendar.isLeapYear(year)) {
         System.out.println("Current is RunYear");
      }
      else {
         System.out.println("Current isn't RunYear");
      }
   }
}
```

十六、 Java 正则表达式

正则表达式定义了字符串的模式。
正则表达式可以用来搜索、编辑或处理文本。
正则表达式并不仅限于某一种语言，但是在每种语言中有细微的差别。

16.1 正则表达式实例

一个字符串其实就是一个简单的正则表达式，例如 **Hello World** 正则表达式匹配 "Hello World" 字符串。
.（点号）也是一个正则表达式，它匹配任何一个字符如："a" 或 "1"。
下表列出了一些正则表达式的实例及描述：

正则表达式	描述

正则表达式	描述
this is text	匹配字符串 "this is text"
this\s+is\s+text	注意字符串中的 **\s+**。匹配单词 "this" 后面的 **\s+** 可以匹配多个空格，之后匹配 is 字符串，再之后 **\s+** 匹配多个空格然后再跟上 text 字符串。可以匹配这个实例：this is text
^\d+(\.\d+)?	^ 定义了以什么开始、\d+ 匹配一个或多个数字、? 设置括号内的选项是可选的 \. 匹配 "." 可以匹配的实例："5", "1.5" 和 "2.21"。

Java 正则表达式和 Perl 的是最为相似的。

java.util.regex 包主要包括以下三个类：

- Pattern 类：

  pattern 对象是一个正则表达式的编译表示。Pattern 类没有公共构造方法。要创建一个 Pattern 对象，你必须首先调用其公共静态编译方法，它返回一个 Pattern 对象。该方法接受一个正则表达式作为它的第一个参数。

- Matcher 类：

  Matcher 对象是对输入字符串进行解释和匹配操作的引擎。与Pattern 类一样，Matcher 也没有公共构造方法。你需要调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。

- PatternSyntaxException：

  PatternSyntaxException 是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

以下实例中使用了正则表达式 **.*runoob.*** 用于查找字符串中是否包了 **runoob** 子串：

```
import java.util.regex.*;
class Test{
    public static void main(String[] args){
        String content = "I am noob " + "from runoob.com";
        String pattern = ".*runoob.*";
        boolean isMatch = Pattern.matches(pattern, content);
        System.out.println("Is str contain the pattern?\n"+
isMatch);
    }
}
```

16.2 捕获组

捕获组是把多个字符当一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。

例如，正则表达式 (dog) 创建了单一分组，组里包含"d"，"o"，和"g"。

捕获组是通过从左至右计算其开括号来编号。例如，在表达式（（A）（B（C））），有四个这样的组：

- ((A)(B(C)))
- (A)

- (B(C))
- (C)

可以通过调用 matcher 对象的 groupCount 方法来查看表达式有多少个分组。groupCount 方法返回一个 int 值，表示matcher对象当前有多个捕获组。

还有一个特殊的组（group(0)），它总是代表整个表达式。该组不包括在 groupCount 的返回值中。

实例

下面的例子说明如何从一个给定的字符串中找到数字串：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Test{
    public static void main(String[] args){
        String line = "This order was placed for QT3000! OK?";
        String pattern = "(\\D*)(\\d+)(.*)";
        // Create a object of Pattern
        Pattern r = Pattern.compile(pattern);
        // Create a object of Matcher
        Matcher m = r.matcher(line);
        if(m.find()){
            System.out.println("Found value: " + m.group(0));
            System.out.println("Found value: " + m.group(1));
            System.out.println("Found value: " + m.group(2));
            System.out.println("Found value: " + m.group(3));
        }else{
            System.out.println("No Match!");
        }
    }
}
//  以上实例编译运行结果如下：

Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT
Found value: 3000
Found value: ! OK?
```

16.3 正则表达式语法

在其他语言中，\\ 表示：我想要在正则表达式中插入一个普通的（字面上的）反斜杠，请不要给它任何特殊的意义。

在 Java 中，\\ 表示：我要插入一个正则表达式的反斜线，所以其后的字符具有特殊的意义。

所以，在其他的语言中（如Perl），一个反斜杠 \ 就足以具有转义的作用，而在 Java 中正则表达式中则需要有两个反斜杠才能被解析为其他语言中的转义作用。也可以简单的理解在 Java 的正则表达式中，两个 \\ 代表其他语言中的一个 \，这也就是为什么表示一位数字的正则表达式是 \\d，而表示一个普通的反斜杠是 \\\\。

字符	说明
\	将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，"n"匹配字符"n"。"\n"匹配换行符。序列"\\"匹配"\"，"\("匹配"("。
^	匹配输入字符串开始的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与"\n"或"\r"之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与"\n"或"\r"之前的位置匹配。
*	零次或多次匹配前面的字符或子表达式。例如，zo* 匹配"z"和"zoo"。* 等效于 {0,}。
+	一次或多次匹配前面的字符或子表达式。例如，"zo+"与"zo"和"zoo"匹配，但与"z"不匹配。+ 等效于 {1,}。
?	零次或一次匹配前面的字符或子表达式。例如，"do(es)?"匹配"do"或"does"中的"do"。? 等效于 {0,1}。
{*n*}	*n* 是非负整数。正好匹配 *n* 次。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。
{*n*,}	*n* 是非负整数。至少匹配 *n* 次。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"fooooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o*"。
{*n*,*m*}	*m* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。
?	当此字符紧随任何其他限定符（、+、*?*、*{n}*、*{n,}*、*{n,m*}）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。
.(点号)	匹配除"\r\n"之外的任何单个字符。若要匹配包括"\r\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。
(*pattern*)	匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **$0…$9** 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"("或者")"。
(?:*pattern*)	匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。
(?=*pattern*)	执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
(?!*pattern*)	执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的"Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。
x|*y*	匹配 *x* 或 *y*。例如，'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。
[*xyz*]	字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。

字符	说明
[^*xyz*]	反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p"，"l"，"i"，"n"。
[*a*-*z*]	字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。
[^*a*-*z*]	反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。
\b	匹配一个字边界，即字与空格间的位置。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。
\B	非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。
\c*x*	匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。
\d	数字字符匹配。等效于 [0-9]。
\D	非数字字符匹配。等效于 [^0-9]。
\f	换页符匹配。等效于 \x0c 和 \cL。
\n	换行符匹配。等效于 \x0a 和 \cJ。
\r	匹配一个回车符。等效于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等。与 [\f\n\r\t\v] 等效。
\S	匹配任何非空白字符。与 【^\f\n\r\t】 等效。
\t	制表符匹配。与 \x09 和 \cI 等效。
\v	垂直制表符匹配。与 \x0b 和 \cK 等效。
\w	匹配任何字类字符，包括下划线。与"[A-Za-z0-9_]"等效。
\W	与任何非单词字符匹配。与"【A-Za-z0-9_】"等效。
\x*n*	匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。
\num	匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。
\n	标识一个八进制转义码或反向引用。如果 *n* 前面至少有 *n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。
\nm	标识一个八进制转义码或反向引用。如果 *nm* 前面至少有 *nm* 个捕获子表达式，那么 *nm* 是反向引用。如果 *nm* 前面至少有 *n* 个捕获，则 *n* 是反向引用，后面跟有字符 *m*。如果两种前面的情况都不存在，则 *nm* 匹配八进制值 *nm*，其中 *n* 和 *m* 是八进制数字 (0-7)。
\nml	当 *n* 是八进制数 (0-3)，*m* 和 *l* 是八进制数 (0-7) 时，匹配八进制转义码 *nml*。
\u*n*	匹配 *n*，其中 *n* 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (©)。

根据 Java Language Specification 的要求，Java 源代码的字符串中的反斜线被解释为 Unicode 转义或其他字符转义。因此必须在字符串字面值中使用两个反斜线，表示正则表达式受到保护，不被 Java 字节码编译器解释。例如，当解释为正则表达式时，字符串字面值 "\b" 与单个退格字符匹配，而 "\\b" 与单词边界匹配。字符串字面值 "(hello)" 是非法的，将导致编译时错误；
要与字符串 (hello) 匹配，必须使用字符串字面值 "\\(hello\\)"。

16.4 Matcher 类的方法

索引方法

索引方法提供了有用的索引值，精确表明输入字符串中在哪能找到匹配：

序号	方法及说明
1	**public int start()** 返回以前匹配的初始索引。
2	**public int start(int group)** 返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引
3	**public int end()** 返回最后匹配字符之后的偏移量。
4	**public int end(int group)** 返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。

查找方法

查找方法用来检查输入字符串并返回一个布尔值，表示是否找到该模式：

序号	方法及说明
1	**public boolean lookingAt()** 尝试将从区域开头开始的输入序列与该模式匹配。
2	**public boolean find()** 尝试查找与该模式匹配的输入序列的下一个子序列。
3	**public boolean find(int start**）** 重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。
4	**public boolean matches()** 尝试将整个区域与模式匹配。

替换方法

替换方法是替换输入字符串里文本的方法：

序号	方法及说明
1	**public Matcher appendReplacement(StringBuffer sb, String replacement)** 实现非终端添加和替换步骤。
2	**public StringBuffer appendTail(StringBuffer sb)** 实现终端添加和替换步骤。
3	**public String replaceAll(String replacement)** 替换模式与给定替换字符串相匹配的输入序列的每个子序列。
4	**public String replaceFirst(String replacement)** 替换模式与给定替换字符串匹配的输入序列的第一个子序列。
5	**public static String quoteReplacement(String s)** 返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给Matcher类的appendReplacement 方法一个字面字符串一样工作。

16.4.1 start 和 end 方法

下面是一个对单词 "cat" 出现在输入字符串中出现次数进行计数的例子：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class Test{
    private static final String REGEX = "\\bcat\\b";
    private static final String INPUT = "cat cat cat cattie
cat";
    public static void main(String[] args){
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT);
        int count = 0;
        while(m.find()){
            count++;
            System.out.println("Match number "+count);
            System.out.println("start(): "+m.start());
            System.out.println("end(): "+m.end());
        }
    }
}
//以上实例编译运行结果如下：
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22
```

可以看到这个例子是使用单词边界，以确保字母 "c" "a" "t" 并非仅是一个较长的词的子串。它也提供了一些关于输入字符串中匹配发生位置的有用信息。
Start 方法返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引，end 方法最后一个匹配字符的索引加 1。

16.4.2 matches 和 lookingAt 方法

matches 和 lookingAt 方法都用来尝试匹配一个输入序列模式。它们的不同是 matches 要求整个序列都匹配，而lookingAt 不要求。

lookingAt 方法虽然不需要整句都匹配，但是需要从第一个字符开始匹配。

这两个方法经常在输入字符串的开始使用。

我们通过下面这个例子，来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class Test{
    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static final String INPUT2 = "ooooofooooooooooo";
    private static final String INPUT3 = "foo";
    private static Pattern pattern;
    private static Matcher m;
    private static Matcher m2;
    private static Matcher m3;
    public static void main(String[] args){
        pattern = Pattern.compile(REGEX);
        m = pattern.matcher(INPUT);
        m2 = pattern.matcher(INPUT2);
        m3 = pattern.matcher(INPUT3);

        System.out.println("Current REGEX is "+REGEX);
        System.out.println("Current INPUT is "+INPUT);
        System.out.println("Current INPUT2 is "+INPUT2);


        System.out.println("lookingAt(): "+m.lookingAt());
        System.out.println("matches() : "+m.matches());
        System.out.println("lookingAt(): "+m2.lookingAt());
        System.out.println("m3.matches() : "+m3.matches());
    }
}
//      输出结果
Current REGEX is foo
Current INPUT is fooooooooooooooooo
Current INPUT2 is ooooofooooooooooo
lookingAt(): true
matches() : false
lookingAt(): false              // REGEX不需要和String完全一样，但必须从头开始匹配
m3.matches() : true             // String要和REGEX完全一样，才会返回true
```

16.4.3 appendReplacement 和 appendTail 方法

Matcher 类也提供了appendReplacement 和 appendTail 方法用于文本替换：

看下面的例子来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class Test{
    private static String REGEX = "a*b";    // 可以匹配到b、ab...（n个a）+b
    private static String INPUT = "aabfooaabfooabfoobkkk";
```

```
    private static String REPLACE = "-";

    public static void main(String[] args){
        Pattern p = Pattern.compile(REGEX);
        Matcher m = p.matcher(INPUT);
        StringBuffer sb = new StringBuffer();
        while(m.find()){
            m.appendReplacement(sb, REPLACE);
        }
        m.appendTail(sb);
        System.out.println(sb.toString());
    }
}
//              输出结果
-foo-foo-foo-kkk
```

16.4.4 replaceFirst 和 replaceAll 方法

replaceFirst 和 replaceAll 方法用来替换匹配正则表达式的文本。不同的是，replaceFirst 替换首次匹配，replaceAll 替换所有匹配。

下面的例子来解释这个功能：

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Test
{
    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. All dogs say meow.";
    private static String REPLACE = "cat";
    private static String RES = "";

    public static void main(String[] args) {
        Pattern p = Pattern.compile(REGEX);
        // get a matcher object
        Matcher m = p.matcher(INPUT);
        INPUT = m.replaceAll(REPLACE);
        System.out.println("ALL: " + INPUT);
        RES = m.replaceFirst(REPLACE);
        System.out.println("Firsrt: "+RES);
   }
}
//  以上实例编译运行结果如下：
ALL: The cat says meow. All cats say meow.
Firsrt: The cat says meow. All dogs say meow.
```

16.5 PatternSyntaxException 类的方法

PatternSyntaxException 是一个非强制异常类，它指示一个正则表达式模式中的语法错误。

PatternSyntaxException 类提供了下面的方法来帮助我们查看发生了什么错误。

序号	方法及说明
1	**public String getDescription()** 获取错误的描述。
2	**public int getIndex()** 获取错误的索引。
3	**public String getPattern()** 获取错误的正则表达式模式。
4	**public String getMessage()** 返回多行字符串，包含语法错误及其索引的描述、错误的正则表达式模式和模式中错误索引的可视化指示。

16.6 笔记（实战）

```
// 校验QQ号，要求：必须是5~15位数字，0不能开头。没有正则表达式之前
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class Test{
    public static void checkQQ(String str){
        String reg = "[1-9][0-9]{4,14}";
        System.out.println(str.matches(reg)?"Valid":"Invalid");
    }
    public static void main(String[] args){
        String qq = "1570285527";
        checkQQ(qq);
    }
}
//              输出结果
Valid
```