湖南石油化工職業技術學院
HUNAN PETROCHEMICAL VOCATIONAL TECHNOLOGY COLLEGE

《Java 语言编程基础》实验指导书

院系名称：经管信息学院

课程代码：311010512

总学时数：64

适用专业：大数据技术与应用专业

编 制 人：杨超

编制日期：2019 年 11 月

审 核 人：

审 定 人：

目录

一、 课程性质和教学目的

本课程是软件技术专业的专业基础课程，是培养学生 Java 程序设计能力的支撑课程。本课程主要培养学生的 Java 开发能力。通过“教、学、做”理论与实践一体化教学，使学生掌握 Java 程序编写的基本方法，并逐步形成正确的 Java 程序设计思想,能够熟练地使用 Java 语言编程进行程序设计，为 Web 开发后续课程打下基础。

通过本课程的学习，使学生逐步建立和掌握 Java 程序设计的思想方法，具有分析问题和解决问题的能力，能够使用 Java 语言编写基本程序解决实际问题，具备吃苦耐劳、团结协作的良好品质

二、实验目的

上机实验的目的不仅是为了验证教材和讲课的内容,或者验证自己所编写的程序正确与否。学习程序设计上机实验的目的是：

1．加深对讲授内容的理解，尤其是一些语法规定，课堂讲授即枯燥无味又难以记忆，但它们都很重要。能过多次上机就能自然地、熟练地掌握。通过上机掌握语法是行之有效的方法。

2．学会上机调试程序。即善于发现程序中的错误，并且能很快排除这些错误， 使程序能正确运行。要真正掌握这门课程，不仅应当了解和熟悉有关理论和方法， 还要求自己动手实现即会编程并上机调试通过。故应给予充分重视。调试程序固然可以借鉴他人的现成经验,但更重要的是通过自己的直接实践来累积经验， 而且有些经验是只能意会难以言传。调试程序的能力是每个程序设计人员应当掌握的一项基本功。

3．做实验时不要在程序通过后就认为搞定、完成任务了，而应当在已通过的程序上作一些改动（例如修改一些参数、增加程序一些功能、改变输入数据的方法等），以观察和分析所出现的情况。

三、上机实验前的准备工作

实验前应做好准备工作，以充分利用有限的上机时间。准备工作至少包括：

1．复习和掌握本实验有关的教学内容。

2．准备好上机所需的程序。初学者切忌不编写程序或抄别人的程序去上机，

应从一开始就养成严谨的科学作风。

3. 对运行上可能出现的问题应事先做出估计；对程序中自己有疑问的地方，应作上记号，以便在上机时给予注意。

4. 根据实验内容认真准备实验程序及调试时所需的输入数据。

5. 在上实验课之前必须写好预习报告（编程题源程序用纸写好或画好程序流程图）

6. 填空与改错题等题要预先做好，上机时的工作只能是输入源程序和调试修改。

7.认真完成实验内容， 得出正确的实验结果。 实验结束后总结实验内容、书写实验报告。

8.遵守实验室规章制度、不缺席、按时上、下机。

四、实验环境

代码编写环境：可根据机房主机条件自己决定。推荐 notepad++、 IDEA 等。

实验 1　JAVA 开发环境配置

实验目的

1. window 系统安装 java
2. 运行 Java 简单程序

实验内容

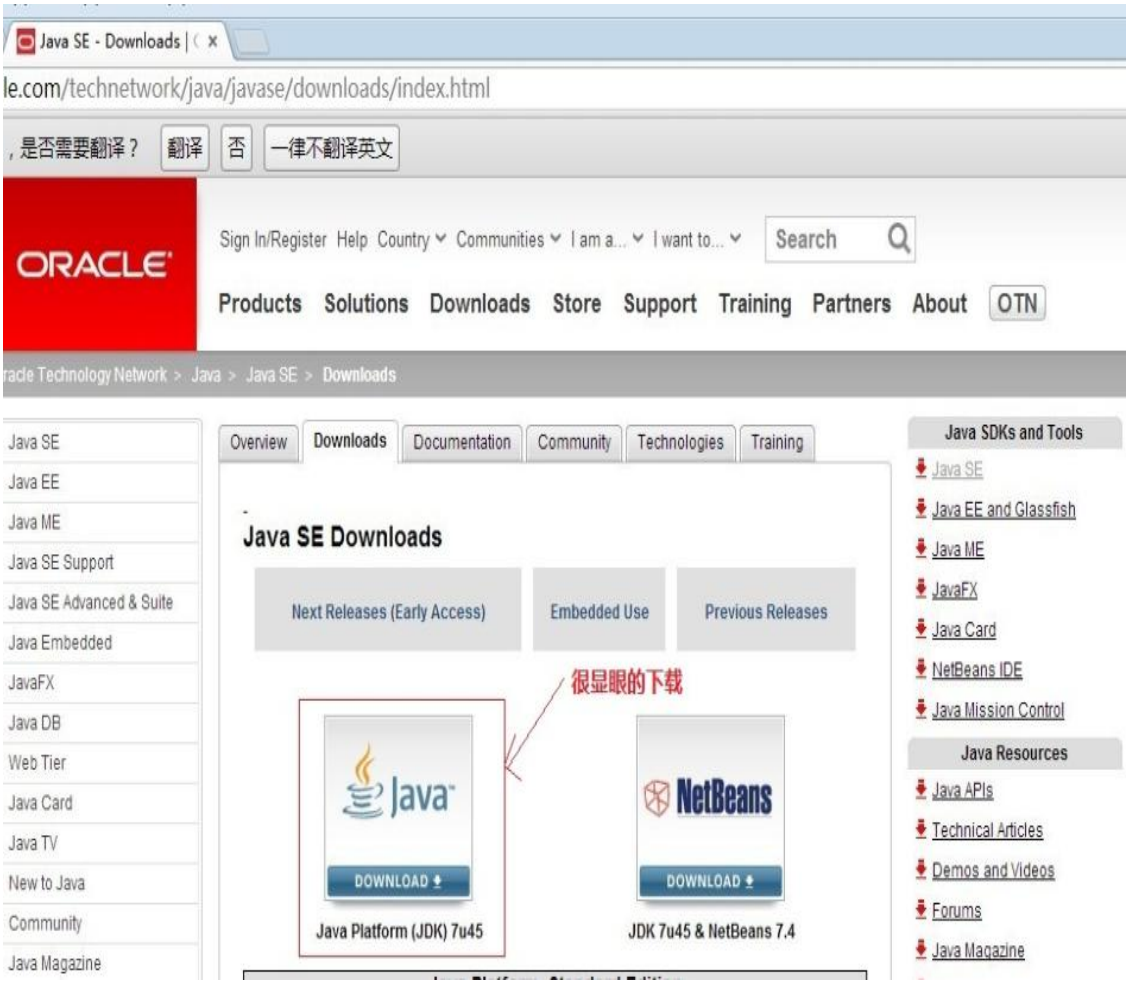
window 系统安装 java

下载 JDK

首先我们需要下载 java 开发工具包 JDK，下载地址：

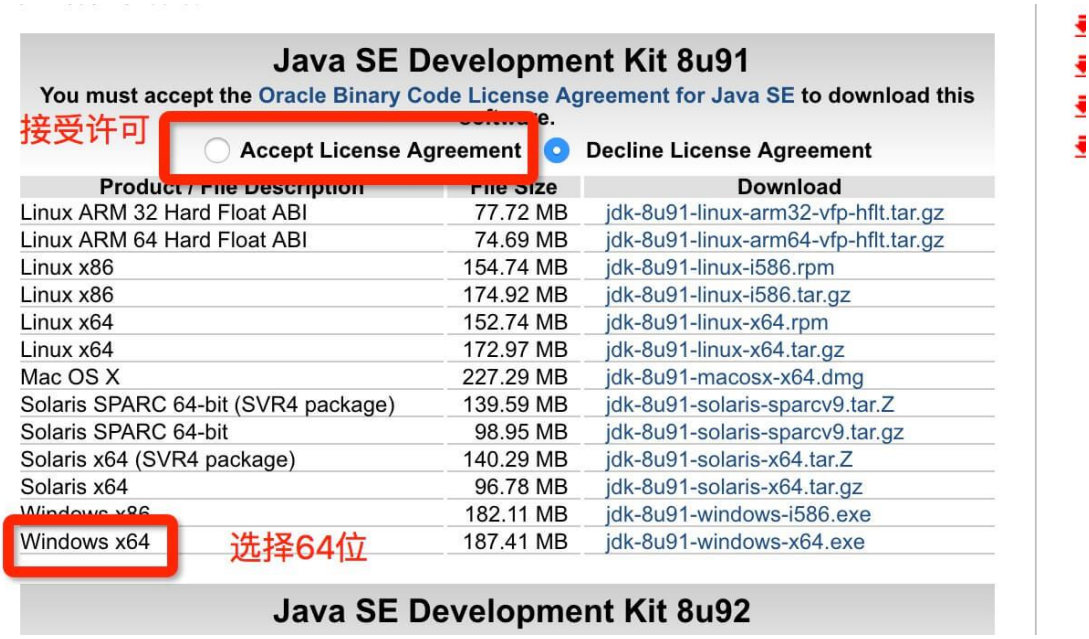
http://www.oracle.com/technetwork/java/javase/downloads/index.html

点击如下下载按钮：



在下载页面中你需要选择接受许可，并根据自己的系统选择对应的版本，本

文以 Window 64 位系统为例：



下载后 JDK 的安装根据提示进行，还有安装 JDK 的时候也会安装 JRE，一并安装就可以了。

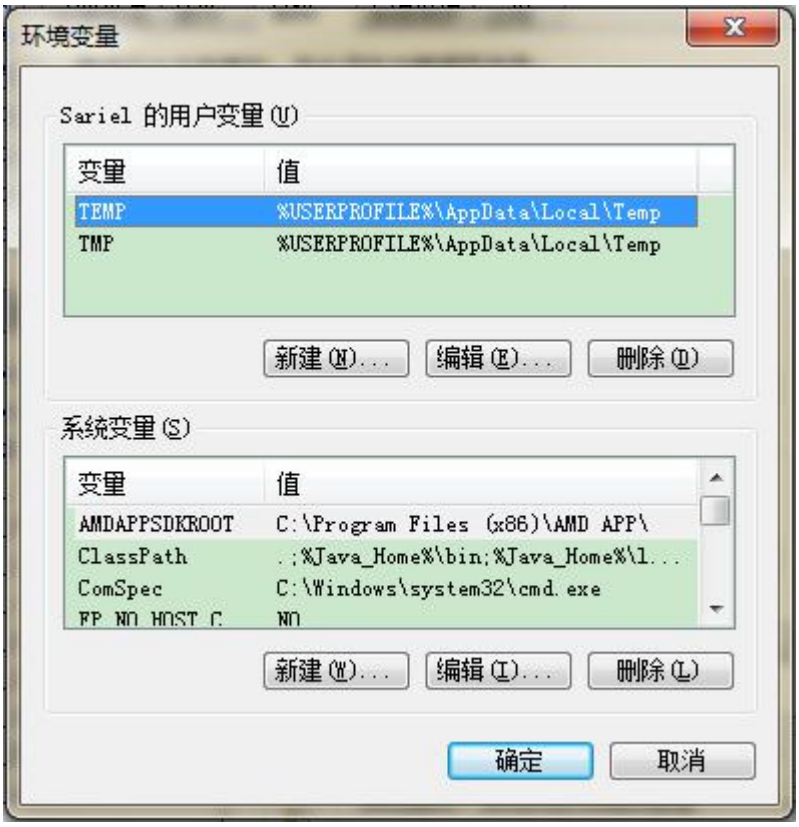
安装 JDK，安装过程中可以自定义安装目录等信息，例如我们选择安装目录为 C:\Program Files (x86)\Java\jdk1.8.0_91。

配置环境变量

1.安装完成后，右击"我的电脑"，点击"属性"，选择"高级系统设置"；

2.选择"高级"选项卡，点击"环境变量"；

然后就会出现如下图所示的画面：



在"系统变量"中设置 3 项属性，JAVA_HOME,PATH,CLASSPATH(大小写无所谓)，若已存在则点击"编辑"，不存在则点击"新建"。

变量设置参数如下：

变量名：JAVA_HOME

变量值：C:\Program Files (x86)\Java\jdk1.8.0_91 // 要根据自己的实际路径配置

变量名：CLASSPATH

变量值:.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar; //记得前面有个"."

变量名：Path

变量值：%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin;

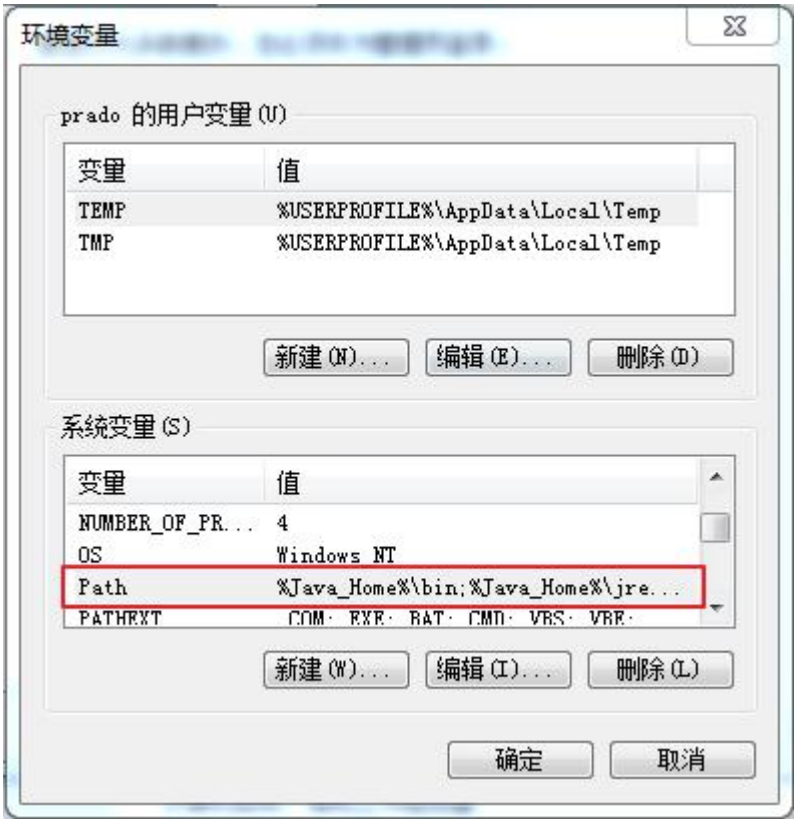
JAVA_HOME 设置


新建系统变量
变量名(N):
JAVA_HOME
变量值(V):
'rogram Files (x86)\Java\jdk1.8.0_91
确定
取消


环境变量
编辑系统变量
变量名(N):
JAVA_HOME
变量值(V):
'rogram Files (x86)\Java\jdk1.8.0_91
确定
取消
系统变量(S)
变量
值
JAVA_HOME
C:\Program Files (x86)\Java\jdk...
MYSQL_HOME
.;D:\Lamp\Mysql57\bin
新建(W)...
编辑(I)...
删除(L)
确定
取消

PATH 设置


编辑系统变量
变量名(N):
Path
变量值(V):
%Java_Home%\bin;%Java_Home%\jre\bin
确定
取消



CLASSPATH 设置

这是 Java 的环境配置，配置完成后，你可以启动 Eclipse 来编写代码，它会自动完成 java 环境的配置。

注意：如果使用 1.5 以上版本的 JDK，不用设置 CLASSPATH 环境变量，也可以正常编译和运行 Java 程序。

测试 JDK 是否安装成功

1、"开始"->"运行"，键入"cmd"；

2、键入命令： java -version、java、javac 几个命令，出现以下信息，

说明环境变量配置成功；

```
C:\Users\prado>java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b14, mixed mode)
```

Linux，UNIX，Solaris，FreeBSD 环境变量设置

环境变量 PATH 应该设定为指向 Java 二进制文件安装的位置。如果设置遇到困难，请参考 shell 文档。

例如，假设你使用 bash 作为 shell，你可以把下面的内容添加到你的 .bashrc 文件结尾：export PATH=/path/to/java:$PATH

流行 JAVA 开发工具

正所谓工欲善其事必先利其器，我们在开发 java 语言过程中同样需要一款不错的开发工具，目前市场上的 IDE 很多，本文为大家推荐以下下几款 java 开发工具：

Eclipse（推荐）：另一个免费开源的 java IDE，下载地址：http://www.eclipse.org/

选择 Eclipse IDE for Java Developers：

Notepad++ ：Notepad++ 是在微软视窗环境之下的一个免费的代码编辑器，下载地址：http://notepad-plus-plus.org/

使用 Eclipse 运行第一个 Java 程序

HelloWorld.java 文件代码：

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

实验 2 基本数据类型的使用

实验目的

1．掌握数据的定义

2．熟悉各种数据类型的使用

实验内容

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

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。

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");
        System.out.println("  最  小  值  :  Float.MIN_VALUE="  +
Float.MIN_VALUE);
```

```
        System.out.println("  最  大  值  :  Float.MAX_VALUE="  +
Float.MAX_VALUE);
        System.out.println();

        // double
        System.out.println("基本类型：double 二进制位数：" +
Double.SIZE);
        System.out.println("包装类：java.lang.Double");
        System.out.println("  最  小  值  :  Double.MIN_VALUE="  +
Double.MIN_VALUE);
        System.out.println("  最  大  值  :  Double.MAX_VALUE="  +
Double.MAX_VALUE);
        System.out.println();

        // char
        System.out.println("基本类型：char 二进制位数：" +
Character.SIZE);
        System.out.println("包装类：java.lang.Character");
        // 以数值形式而不是字符形式将Character.MIN_VALUE输出到控制台
        System.out.println("最小值：Character.MIN_VALUE="
                + (int) Character.MIN_VALUE);
        // 以数值形式而不是字符形式将Character.MAX_VALUE输出到控制台
        System.out.println("最大值：Character.MAX_VALUE="
                + (int) Character.MAX_VALUE);
    }
    }
```

编译以上代码输出结果如下所示：

```
基本类型：byte 二进制位数：8
包装类：java.lang.Byte
```

最小值：Byte.MIN_VALUE=-128

最大值：Byte.MAX_VALUE=127

基本类型：short 二进制位数：16

包装类：java.lang.Short

最小值：Short.MIN_VALUE=-32768

最大值：Short.MAX_VALUE=32767

基本类型：int 二进制位数：32

包装类：java.lang.Integer

最小值：Integer.MIN_VALUE=-2147483648

最大值：Integer.MAX_VALUE=2147483647

基本类型：long 二进制位数：64

包装类：java.lang.Long

最小值：Long.MIN_VALUE=-9223372036854775808

最大值：Long.MAX_VALUE=9223372036854775807

基本类型：float 二进制位数：32

包装类：java.lang.Float

最小值：Float.MIN_VALUE=1.4E-45

最大值：Float.MAX_VALUE=3.4028235E38

基本类型：double 二进制位数：64

包装类：java.lang.Double

最小值：Double.MIN_VALUE=4.9E-324

最大值：Double.MAX_VALUE=1.7976931348623157E308

基本类型：char 二进制位数：16

包装类：java.lang.Character

最小值：Character.MIN_VALUE=0

最大值：Character.MAX_VALUE=65535

实验 3 对象的应用

实验目的

1．熟悉 Java 对象的使用

2．熟悉类方法定义使用

实验内容

Java 对象和类

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，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。

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。

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。

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
    }
```

```
}
```

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("Passed Name is :" + name );
   }
   public static void main(String []args){
      // 下面的语句将创建一个 Puppy 对象
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

编译并运行上面的程序，会打印出下面的结果：

```
Passed Name is :tommy
```

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 */
ObjectReference = new Constructor();/* 访问其中的变量 */
ObjectReference.variableName;/* 访问类中的方法 */
ObjectReference.MethodName();
```

实例

下面的例子展示如何访问实例变量和调用成员方法：

```
public class Puppy{
   int puppyAge;
   public Puppy(String name){
      // 这个构造器仅有一个参数：name
      System.out.println("Passed Name is :" + name );
   }

   public void setAge( int age ){
       puppyAge = age;
   }

   public int getAge( ){
       System.out.println("Puppy's age is :" + puppyAge );
       return puppyAge;
   }

   public static void main(String []args){
      /* 创建对象 */
      Puppy myPuppy = new Puppy( "tommy" );
      /* 通过方法来设定 age */
      myPuppy.setAge( 2 );
      /* 调用另一个方法获取 age */
      myPuppy.getAge( );
      /*你也可以像下面这样访问成员变量 */
      System.out.println("Variable Value :" + myPuppy.puppyAge );
   }
}
```

编译并运行上面的程序，产生如下结果：

```
Passed Name is :tommy
Puppy's age is :2
Variable Value :2
```

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。

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。

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.*;

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。

```
import java.io.*;public class Employee{
   String name;
   int age;
   String designation;
   double salary;
   // Employee 类的构造器
   public 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(double empSalary){
      salary = empSalary;
```

```
    }
    /* 打印信息 */
    public void printEmployee(){
       System.out.println("Name:"+ name );
       System.out.println("Age:" + age );
       System.out.println("Designation:" + designation );
       System.out.println("Salary:" + 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。

```
import java.io.*;public class EmployeeTest{

   public static void main(String args[]){
      /* 使用构造器创建两个对象 */
      Employee empOne = new Employee("James Smith");
      Employee empTwo = new Employee("Mary Anne");

      // 调用这两个对象的成员方法
      empOne.empAge(26);
      empOne.empDesignation("Senior Software Engineer");
      empOne.empSalary(1000);
      empOne.printEmployee();

      empTwo.empAge(21);
      empTwo.empDesignation("Software Engineer");
```

```
        empTwo.empSalary(500);
        empTwo.printEmployee();
    }
}
```

编译这两个文件并且运行 EmployeeTest 类，可以看到如下结果：

```
C :> javac Employee.java
C :> vi EmployeeTest.java
C :> javac  EmployeeTest.java
C :> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

实验 4 运算符的使用

实验目的

1．运算符的应用

2．数据类型的转换

实验内容

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

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
%	取模 - 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
--	自减 - 操作数的值减少 1	B-- 或 --B 等于 19

实例

下面的简单示例程序演示了算术运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     int c = 25;
     int d = 25;
     System.out.println("a + b = " + (a + b) );
     System.out.println("a - b = " + (a - b) );
     System.out.println("a * b = " + (a * b) );
     System.out.println("b / a = " + (b / a) );
     System.out.println("b % a = " + (b % a) );
     System.out.println("c % a = " + (c % a) );
     System.out.println("a++   = " +  (a++) );
     System.out.println("a--   = " +  (a--) );
     // 查看  d++ 与 ++d 的不同
     System.out.println("d++   = " +  (d++) );
     System.out.println("++d   = " +  (++d) );
  }
}
```

以上实例编译运行结果如下：

```
a + b = 30
```

a - b = -10

a * b = 200

b / a = 2

b % a = 0

c % a = 5

a++ = 10

a-- = 11

d++ = 25

++d = 27

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(非真)。
!=	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）非真。
<	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。

实例

下面的简单示例程序演示了关系运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;
     System.out.println("a == b = " + (a == b) );
     System.out.println("a != b = " + (a != b) );
     System.out.println("a > b = " + (a > b) );
     System.out.println("a < b = " + (a < b) );
System.out.println("b >= a = " + (b >= a) );
     System.out.println("b <= a = " + (b <= a) );   } }
```

以上实例编译运行结果如下：

```
a == b = false
a != b = true
a > b = false
a < b = true b >= a = true
b <= a = false
```

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，和 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
&	按位与操作符,当且仅当两个操作数的某一位都非0时候结果的该位才为1。	（A&B），得到12，即0000 1100
|	按位或操作符,只要两个操作数的某一位有一个非0时候结果的该位就为1。	（A | B）得到61，即 0011 1101
^	按位异或操作符,两个操作数的某一位不相同时候结果的该位就为1。	（A ^ B）得到49，即 0011 0001
~	按位补运算符翻转操作数的每一位。	（~A）得到-61，即1100 0011
<<	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

实例

下面的简单示例程序演示了位运算符。复制并粘贴下面的Java程序并保存

为 Test.java 文件，然后编译并运行这个程序：

```
    public class Test {
       public static void main(String args[]) {
            int a = 60; /* 60 = 0011 1100 */
             int b = 13; /* 13 = 0000 1101 */
             int c = 0;
             c = a & b;              /* 12 = 0000 1100 */
             System.out.println("a & b = " + c );

             c = a | b;              /* 61 = 0011 1101 */
             System.out.println("a | b = " + c );

             c = a ^ b;              /* 49 = 0011 0001 */
             System.out.println("a ^ b = " + c );

             c = ~a;                    /*-61 = 1100 0011 */
             System.out.println("~a = " + c );

             c = a << 2;          /* 240 = 1111 0000
*/           System.out.println("a << 2 = " + c );           c = a >>
2;          /* 215 = 1111 */
             System.out.println("a >> 2   = " + c );

            c = a >>> 2;          /* 215 = 0000 1111 */
             System.out.println("a >>> 2 = " + c );
       }
    }
```

以上实例编译运行结果如下：

```
a & b = 12
```

a | b = 61

a ^ b = 49

~a = -61

a << 2 = 240

a >> 2 = 15

a >>> 2 = 15

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。

实例

下面的简单示例程序演示了逻辑运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
public class Test {
   public static void main(String args[]) {
         boolean a = true;
         boolean b = false;
         System.out.println("a && b = " + (a&&b));
         System.out.println("a || b = " + (a||b) );
         System.out.println("!(a && b) = " + !(a && b));
   }
```

```
}
```

以上实例编译运行结果如下：

```
a && b = false
a || b = true
!(a && b) = true
```

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 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
>> =	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

实例

面的简单示例程序演示了赋值运算符。复制并粘贴下面的 Java 程序并保存为 Test.java 文件，然后编译并运行这个程序：

```
public class Test {
   public static void main(String args[]) {
         int a = 10;
         int b = 20;
         int c = 0;
         c = a + b;
         System.out.println("c = a + b = " + c );
         c += a ;
         System.out.println("c += a   = " + c );
         c -= a ;
         System.out.println("c -= a = " + c );
         c *= a ;
         System.out.println("c *= a = " + c );
         a = 10;
         c = 15;
         c /= a ;
         System.out.println("c /= a = " + c );
         a = 10;
```

```
            c = 15;
            c %= a ;
            System.out.println("c %= a   = " + c );
            c <<= 2 ;          System.out.println("c <<= 2 = " +
c );          c >>= 2 ;
            System.out.println("c >>= 2 = " + c );
            c >>= 2 ;
            System.out.println("c >>= a = " + c );
            c &= a ;
            System.out.println("c &= a  = " + c );
            c ^= a ;
            System.out.println("c ^= a  = " + c );
            c |= a ;
            System.out.println("c |= a  = " + c );
        }
    }
```

以上实例编译运行结果如下：

```
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20 c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
```

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

实例

```
public class Test {
   public static void main(String args[]){
      int a , b;
      a = 10;
b = (a == 1) ? 20: 30;
System.out.println( "Value of b is : " +  b );
      b = (a == 10) ? 20: 30;
     System.out.println( "Value of b is : " + b );
   }
}
```

以上实例编译运行结果如下：

```
Value of b is : 30
Value of b is : 20
```

---

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 Strine 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[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result);
   }
}
```

以上实例编译运行结果如下:

```
true
```

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	+ + - ! ~	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
逻辑或	||	左到右
条件	? :	从右到左
赋值	= + = - = * = / =%= >> = << =&= ^ = | =	从右到左
逗号	,	左到右

实验 5 循环结构的应用

实验目的

1. 掌握循环结构的使用

2. 掌握条件结构的使用

实验内容

Java 循环结构 - for, while 及 do...while

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

在 Java5 中引入了一种主要用于数组的增强型 for 循环。

---

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
}
```

只要布尔表达式为 true, 循环体会一直执行下去。

实例

```
public class Test {
   public static void main(String args[]) {
      int x = 10;
      while( x < 20 ) {          System.out.print("value of x : " + x );          x++;          System.out.print("\n");       }    } }
```

以上实例编译运行结果如下:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
```

```
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

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。

实例

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      do{
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }while( x < 20 );    } }
```

以上实例编译运行结果如下：

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

---

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
}
```

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。

实例

```
public class Test {
```

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

           for(int x = 10; x < 20; x = x+1)
{           System.out.print("value of x : " + x );
System.out.print("\n");       }    } }
```

以上实例编译运行结果如下：

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

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
}
```

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。

实例

```
public class Test {
```

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

      for(int x : numbers ){
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names ={"James", "Larry", "Tom", "Lacy"};
      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

以上实例编译运行结果如下：

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

---

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 ) {
         if( x == 30 ) {
         break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

以上实例编译运行结果如下：

```
1020
```

---

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。

语法

continue 就是循环体中一条简单的语句：

```
continue;
```

实例

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

实验 6 包装类的应用

实验目的

1. 熟悉包装类

2. 熟悉包装类的使用

实验内容

Java Number 类

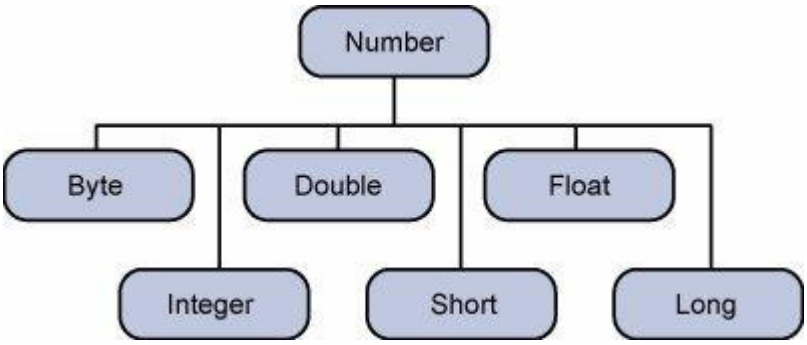
一般地，当需要使用数字的时候，我们通常使用内置数据类型，如：byte、int、long、double 等。

实例

```
int i = 5000;float gpa = 13.65;byte mask = 0xaf;
```

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。



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。

下面是一个装箱与拆箱的例子：

```
public class Test{

    public static void main(String args[]){
        Integer x = 5; // boxes int to an Integer o
```

```
bject
                x = x + 10;      // unboxes the Integer to
  a int
                System.out.println(x);
            }
    }
```

以上实例编译运行结果如下：

```
15
```

当 x 被赋为整型值时，由于 x 是一个对象，所以编译器要对 x 进行装箱。然后，为了使 x 能进行加运算，所以要对 x 进行拆箱。

Java 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。

实例

```
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:1.00 度的余弦值:1.060 度的正切值:1.73205080756887671
的反正切值： 0.7853981633974483
π/2 的角度值：90.03.141592653589793
```

Number & Math 类方法

下面的表中列出的是常用的 Number 类和 Math 类的方法：

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 Integer 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() 对整形变量向左取整，返回类型为 double 型。

9	floor() 对整型变量向右取整。返回类型为 double 类型。
10	rint() 返回与参数最接近的整数。返回类型为 double。
11	round() 返回一个最接近的 int、long 型值。
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。

实验 7 STRING 的应用

实验目的

1. String 对象的灵活运用

实验内容

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。

创建字符串

创建字符串最简单的方式如下:

```
String greeting = "Hello world!";
```

在代码中遇到字符串常量时，这里的值是"Hello world!"，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 = { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

以上实例编译运行结果如下:

```
hello.
```

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。如果需要对字符串做很多修改,那么应该选择使用 StringBuffer

& StringBuilder 类。

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。

下面的代码执行后，len 变量等于 17:

```
public class StringDemo {

   public static void main(String args[]) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println( "String Length is : " + len );
   }
}
```

以上实例编译运行结果如下：

```
String Length is : 17
```

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：

```
"My name is ".concat("Zara");
```

更常用的是使用'+'操作符来连接字符串，如：

```
"Hello," + " world" + "!"
```

结果如下：

```
"Hello, world!"
```

下面是一个例子:

```
public class StringDemo {
   public static void main(String args[]) {
   String string1 = "saw I was ";
   System.out.println("Dot " + string1 + "Tod");
}
}
```

以上实例编译运行结果如下：

```
Dot saw I was Tod
```

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("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);
```

你也可以这样写

```
String fs;
fs = String.format("The value of the float variable is " +
                   "%f, while the value of the integer " +
                   "variable is %d, and the string " +
                   "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

String 方法

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

SN(序	方法描述

号)	
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。
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StringButter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。
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。
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。

验 8 数组的应用

实验目的

1. 掌握数组的使用

2. 掌握数组的方法

实验内容

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。

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;       //  首选的方法
```

或

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

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。

实例

下面是这两种语法的代码示例：

```
double[]  myList;                    //  首选的方法
```

或

```
double  myList[];                    //    效果相同，但不是首选方法
```

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};
```

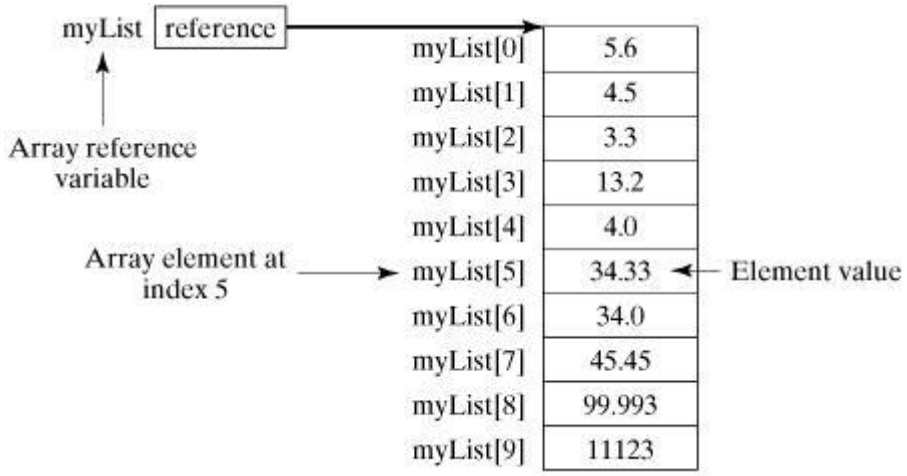
数组的元素是通过索引访问的。数组索引从 0 开始，所以索引值从 0 到 arrayRefVar.length-1。

实例

下面的语句首先声明了一个数组变量 myList，接着创建了一个包含 10 个 double 类型元素的数组，并且把它的引用赋值给 myList 变量。

```
double[] myList = new double[10];
```

下面的图片描绘了数组 myList。这里 myList 数组里有 10 个 double 元素，它的下标从 0 到 9。



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 foreach 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 TestArray {

    public static void main(String[] args) {
        double[] myList = {1.9, 2.9, 3.4, 3.5};

        // 打印所有数组元素
        for (int i = 0; i < myList.length; i++) {
```

```
            System.out.println(myList[i] + " ");
        }
        // 计算所有元素的总和
        double total = 0;
        for (int i = 0; i < myList.length; i++) {
            total += myList[i];
        }
        System.out.println("Total is " + total);
        // 查找最大元素
        double max = myList[0];
        for (int i = 1; i < myList.length; i++) {
            if (myList[i] > max) max = myList[i];
        }
        System.out.println("Max is " + max);
    }
}
```

以上实例编译运行结果如下：

```
1.92.93.43.5
Total is 11.7
Max is 3.5
```

---

foreach 循环

JDK 1.5 引进了一种新的循环类型，被称为 foreach 循环或者加强型循环，它能在不使用下标的情况下遍历数组。

示例

该实例用来显示数组 myList 中的所有元素：

```
public class TestArray {

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

```
            double[] myList = {1.9, 2.9, 3.4, 3.5};

            // 打印所有数组元素
            for (double element: myList) {
                System.out.println(element);
            }
        }
    }
```

以上实例编译运行结果如下：

```
1.92.93.43.5
```

数组作为函数的参数

数组可以作为参数传递给方法。例如，下面的例子就是一个打印 int 数组中元素的方法。

```
public static void printArray(int[] array) {
    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
}
```

下面例子调用 printArray 方法打印出 3，1，2，6，4 和 2：

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

数组作为函数的返回值

```
public static int[] reverse(int[] list) {
    int[] result = new int[list.length];

    for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
        result[j] = list[i];
```

```
        }
        return  result;
    }
```

以上实例中 result 数组作为函数的返回值。

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
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）。

实验 9　日期时间的应用

实验目的

1. 掌握日期时间的应用

实验内容

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 日起的微秒数。

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( ) 转换 Date 对象为 String 表示形式，并返回该字符串。

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 DateDemo {
   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 使用 toString() 函数显示日期时间
       System.out.println(date.toString());
```

```
    }
}
```

以上实例编译运行结果如下:

```
Mon May 04 09:51:52 CDT 2013
```

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 日经历的微秒数值),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。

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:

```
import java.util.*;import java.text.*;
public class DateDemo {
   public static void main(String args[]) {

      Date dNow = new Date( );
      SimpleDateFormat ft =
      new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

      System.out.println("Current Date: " + ft.format(dNow));
   }
}
```

以上实例编译运行结果如下:

Current Date: Sun 2004.07.18 at 04:14:09 PM PDT

简单的DateFormat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	微妙数	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
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	`

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。例如：

```
import java.util.Date;
public class DateDemo {

  public static void main(String args[]) {
     // 初始化 Date 对象
     Date date = new Date();

     // 使用 toString()显示日期和时间
     String str = String.format("Current Date/Time : %tc", date );

     System.out.printf(str);
  }
}
```

以上实例编译运行结果如下：

```
Current Date/Time : Sat Dec 15 16:37:57 MST 2012
```

如果你需要重复提供日期，那么利用这种方式来格式化它的每一部分就有点复杂了。因此，可以利用一个格式化字符串指出要被格式化的参数的索引。

索引必须紧跟在%后面，而且必须以$结束。例如：

```
import java.util.Date;
```

```
  public class DateDemo {

   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 使用 toString()显示日期和时间
       System.out.printf("%1$s %2$tB %2$td, %2$tY",
                         "Due date:", date);
   }
}
```

以上实例编译运行结果如下:

```
Due date: February 09, 2004
```

或者，你可以使用<标志。它表明先前被格式化的参数要被再次使用。例如:

```
import java.util.Date;
  public class DateDemo {

   public static void main(String args[]) {
       // 初始化 Date 对象
       Date date = new Date();

       // 显示格式化时间
       System.out.printf("%s %tB %<te, %<tY",
                         "Due date:", date);
   }
}
```

以上实例编译运行结果如下:

```
Due date: February 09, 2004
```

日期和时间转换字符

字符	描述	例子
c	完整的日期和时间	Mon May 04 09:51:52 CDT 2009
F	ISO 8601 格式日期	2004-02-09
D	U.S. 格式日期 (月/日/年)	02/09/2004
T	24 小时时间	18:05:19
r	12 小时时间	06:05:19 pm
R	24 小时时间，不包含秒	18:05
Y	4 位年份(包含前导 0)	2004
y	年份后 2 位(包含前导 0)	04
C	年份前 2 位(包含前导 0)	20
B	月份全称	February
b	月份简称	Feb
n	2 位月份(包含前导 0)	02
d	2 位日子(包含前导 0)	03
e	2 位日子(不包含前导 0)	9
A	星期全称	Monday
a	星期简称	Mon
j	3 位年份(包含前导 0)	069
H	2 位小时(包含前导 0)，00 到 23	18

k	2 位小时(不包含前导 0)， 0 到 23	18
I	2 位小时(包含前导 0)，01 到 12	06
l	2 位小时(不包含前导 0)， 1 到 12	6
M	2 位分钟(包含前导 0)	05
S	2 位秒数(包含前导 0)	19
L	3 位毫秒(包含前导 0)	047
N	9 位纳秒(包含前导 0)	047000000
P	大写上下午标志	PM
p	小写上下午标志	pm
z	从 GMT 的 RFC 822 数字偏移	-0800
Z	时区	PST
s	自 1970-01-01 00:00:00 GMT 的秒数	1078884319
Q	自 1970-01-01 00:00:00 GMT 的毫妙	1078884319047

还有其他有用的日期和时间相关的类。对于更多的细节，你可以参考到 Java 标准文档。

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 DateDemo {

   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);
        }
    }
}
```

以上实例编译运行结果如下:

```
$ java DateDemo1818-11-11 Parses as Wed Nov 11 00:00:00 GMT 1818
$ java DateDemo 2007-12-012007-12-01 Parses as Sat Dec 01 00:00:00
GMT 2007
```

Java 休眠(sleep)

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 SleepDemo {
   public static void main(String args[]) {
      try {
         System.out.println(new Date( ) + "\n");
         Thread.sleep(5*60*10);
         System.out.println(new Date( ) + "\n");
```

```
      } catch (Exception e) {
          System.out.println("Got an exception!");
      }
   }
}
```

以上实例编译运行结果如下:

```
Sun May 03 18:04:41 GMT 2009

Sun May 03 18:04:44 GMT 2009
```

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 DiffDemo {

   public static void main(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!");
      }
   }
}
```

以上实例编译运行结果如下:

```
Sun May 03 18:16:51 GMT 2009

Sun May 03 18:16:57 GMT 2009

Difference is : 5993
```

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 类中用一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 c1 = Calendar.getInstance();
```

调用：

```
public final void set(int year,int month,int date)
```

```
c1.set(2009,6－1,12);//把Calendar对象c1的年月日分别设这为:2009、6、12
```

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 c1 对象代表的日期设置为 10 号，其它所有的数值会被重新计算

```
c1.set(Calendar.DATE,10);
```

把 c1 对象代表的年份设置为 2008 年，其他的所有数值会被重新计算

```
c1.set(Calendar.YEAR,2008);
```

其他字段属性 set 的意义以此类推

Add 设置

```
Calendar c1 = Calendar.getInstance();
```

把 c1 对象的日期加上 10，也就是 c1 所表的日期的 10 天后的日期，其它所有的数值会被重新计算

c1.add(Calendar.DATE, 10);

把 c1 对象的日期减去 10，也就是 c1 所表的日期的 10 天前的日期，其它所有的数值会被重新计算

c1.add(Calendar.DATE, -10);

其他字段属性的 add 的意义以此类推

Calendar 类对象信息的获得

Calendar c1 = 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);

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。这些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。
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。
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 GregorianCalendarDemo {

   public static void main(String args[]) {
      String months[] = {
      "Jan", "Feb", "Mar", "Apr",
      "May", "Jun", "Jul", "Aug",
      "Sep", "Oct", "Nov", "Dec"};

      int year;
      // 初始化 Gregorian 日历
      // 使用当前时间和日期
      // 默认为本地时间和时区
      GregorianCalendar gcalendar = new GregorianCalendar();
      // 显示当前时间和日期的信息
      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));
```

```
        // 测试当前年份是否为闰年
        if(gcalendar.isLeapYear(year)) {
           System.out.println("当前年份是闰年");
        }
        else {
           System.out.println("当前年份不是闰年");
        }
     }
}
```

实验 10 JAVA 流(STREAM)、文件(FILE)和 IO

实验目的

1. 掌握 Java IO 流

2. 掌握 Java 文件流

实验内容

Java 流(Stream)、文件(File)和 IO

Java.io 包几乎包含了所有操作输入、输出需要的类。所有这些流类代表了输入源和输出目标。

Java.io 包中的流支持很多种格式，比如：基本类型、对象、本地化字符集等等。

一个流可以理解为一个数据的序列。输入流表示从一个源读取数据，输出流表示向一个目标写数据。

Java 为 I/O 提供了强大的而灵活的支持，使其更广泛地应用到文件传输和网络编程中。

但本节讲述最基本的和流与 I/O 相关的功能。我们将通过一个个例子来学习这些功能。

读取控制台输入

Java 的控制台输入由 System.in 完成。

为了获得一个绑定到控制台的字符流，你可以把 System.in 包装在一个 BufferedReader 对象中来创建一个字符流。

下面是创建 BufferedReader 的基本语法：

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

BufferedReader 对象创建后，我们便可以使用 read()方法从控制台读取一个字符，或者用 readLine()方法读取一个字符串。

从控制台读取多字符输入

从 BufferedReader 对象读取一个字符要使用 read()方法，它的语法如下：

```
int read() throws IOException
```

每次调用 read()方法，它从输入流读取一个字符并把该字符作为整数值返回。 当流结束的时候返回-1。该方法抛出 IOException。

下面的程序示范了用 read()方法从控制台不断读取字符直到用户输入"q"。

```
// 使用 BufferedReader 在控制台读取字符
import java.io.*;
public class BRRead {
      public static void main(String args[]) throws IOException
      {
            char c;
            // 使用 System.in 创建 BufferedReader
            BufferedReader br = new BufferedReader(new
                                                InputStreamReader(System.in));
            System.out.println("Enter characters, 'q' to quit.");
            // 读取字符
            do {
                  c = (char) br.read();
                  System.out.println(c);
            } while(c != 'q');
      }
}
```

以上实例编译运行结果如下：

```
Enter characters, 'q' to quit.123abcq123
a
b
```

c

q

从控制台读取字符串

从标准输入读取一个字符串需要使用 BufferedReader 的 readLine()方法。

它的一般格式是:

String readLine() throws IOException

下面的程序读取和显示字符行直到你输入了单词"end"。

```
// 使用 BufferedReader 在控制台读取字符
import java.io.*;public class BRReadLines {
        public static void main(String args[]) throws IOExcep
tion
        {
              // 使用 System.in 创建 BufferedReader
              BufferedReader br = new BufferedReader(new
                                                    Input
StreamReader(System.in));
              String str;
              System.out.println("Enter lines of text.");
              System.out.println("Enter 'end' to quit.");
              do {
                    str = br.readLine();
                    System.out.println(str);
              } while(!str.equals("end"));
        }
}
```

以上实例编译运行结果如下:

```
Enter lines of text.
Enter 'end' to quit.
```

```
This  is  line  one
This  is  line  one
This  is  line  two
This  is  line  two
end
end
```

控制台输出

在此前已经介绍过，控制台的输出由 print() 和 println()完成。这些方法都由类 PrintStream 定义，System.out 是该类对象的一个引用。

PrintStream 继承了 OutputStream 类，并且实现了方法 write()。这样，write()也可以用来往控制台写操作。

PrintStream 定义 write()的最简单格式如下所示：

```
void  write(int  byteval)
```

该方法将 byteval 的低八位字节写到流中。

实例

下面的例子用 write()把字符"A"和紧跟着的换行符输出到屏幕：

```
import  java.io.*;
//  演示  System.out.write().public  class  WriteDemo  {
      public  static  void  main(String  args[])  {
            int  b;
            b  =  'A';
            System.out.write(b);
            System.out.write('\n');
      }
}
```

运行以上实例在输出窗口输出"A"字符

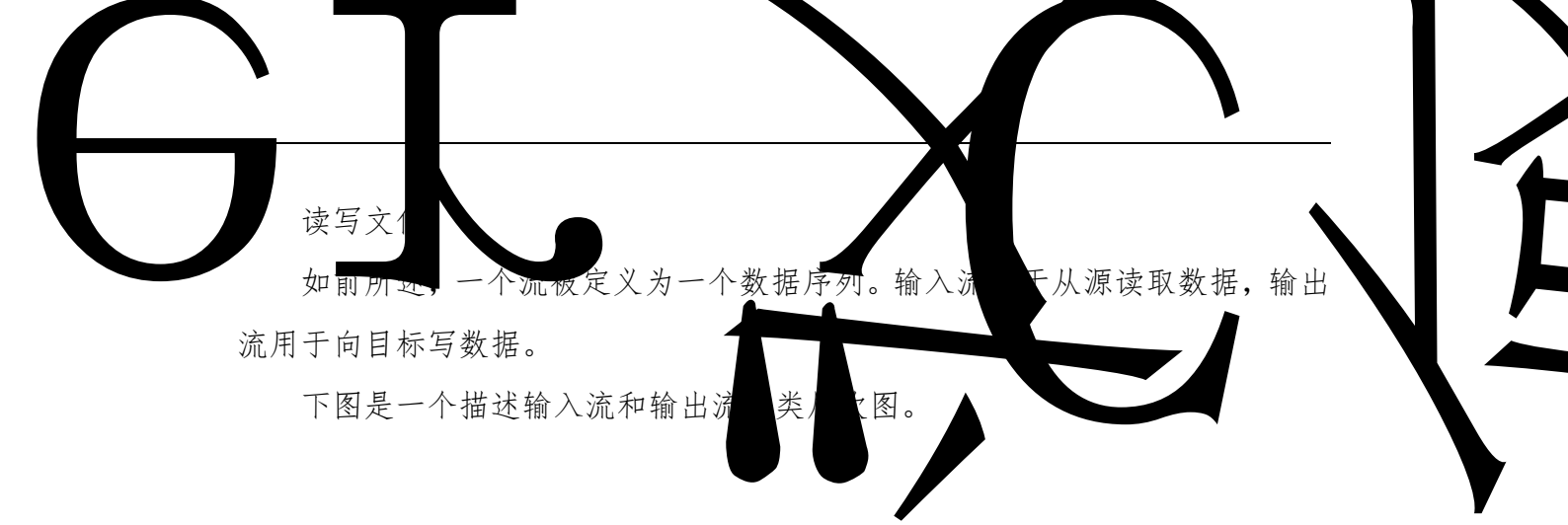
```
A
```

注意：write()方法不经常使用，因为 print()和 println()方法用起来更为方便。

读写文件

如前所述，一个流被定义为一个数据序列。输入流用于从源读取数据，输出流用于向目标写数据。

下图是一个描述输入流和输出流的类层次图。



下面将要讨论的两个重要的流是 FileInputStream 和 FileOutputStream:

FileInputStream

该流用于从文件读取数据，它的对象可以用关键字 new 来创建。

有多种构造方法可用来创建对象。

可以使用字符串类型的文件名来创建一个输入流对象来读取文件:

```
InputStream f = new FileInputStream("C:/java/hello");
```

也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用 File()() 对象的台

	回下一字节数据，如果已经到结尾则返回-1。
4	public int read(byte[] r) throws IOException{} 这个方法从输入流读取 r.length 长度的字节。返回读取的字节数。如果是文件结尾则返回-1。
5	public int available() throws IOException{} 返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取的字节数。返回一个整数值。

除了 InputStream 外，还有一些其他的输入流，更多的细节参考下面链接：

ByteArrayInputStream

DataInputStream

FileOutputStream

该类用来创建一个文件并向文件中写数据。

如果该流在打开文件进行输出前，目标文件不存在，那么该流会创建该文件。

有两个构造方法可以用来创建 FileOutputStream 对象。

使用字符串类型的文件名来创建一个输出流对象：

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

也可以使用一个文件对象来创建一个输出流来写文件。我们首先得使用 File()方法来创建一个文件对象：

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

创建 OutputStream 对象完成后，就可以使用下面的方法来写入流或者进行其他的流操作。

序号	方法及描述
1	public void close() throws IOException{} 关闭此文件输入流并释放与此流有关的所有系统资源。抛出 IOException 异常。
2	protected void finalize()throws IOException {} 这个方法清除与该文件的连接。确保在不再引用文件输入流时调用其 close 方法。抛出 IOException 异常。
3	public void write(int w)throws IOException{} 这个方法把指定的字节写到输出流中。
4	public void write(byte[] w) 把指定数组中 w.length 长度的字节写到 OutputStream 中。

除了 OutputStream 外，还有一些其他的输出流，更多的细节参考下面链接：

ByteArrayOutputStream

DataOutputStream

实例

下面是一个演示 InputStream 和 OutputStream 用法的例子：

```
import  java.io.*;
public  class  fileStreamTest{

      public  static  void  main(String  args[]){

      try{
```

```
                byte  bWrite  []  =  {11,21,3,40,5};
                OutputStream  os  =  new  FileOutputStream("test.txt")
;
                for(int  x=0;  x  <  bWrite.length  ;  x++){
            os.write(  bWrite[x]  );  //  writes  the  bytes
      }             os.close();                          InputStream
  is  =  new  FileInputStream("test.txt");             int  size  =
  is.available();               for(int  i=0;  i<  size;  i++){
                System.out.print((char)is.read()  +  "    ");
      }             is.close();        }catch(IOException  e){
        System.out.print("Exception");        }          }  }
```

上面的程序首先创建文件 test.txt，并把给定的数字以二进制形式写进该文件，同时输出到控制台上。

以上代码由于是二进制写入，可能存在乱码，你可以使用以下代码实例来解决乱码问题：

```
//文件名  :fileStreamTest2.javaimport  java.io.*;
public  class  fileStreamTest2{
   public  static  void  main(String[]  args)  throws  IOExceptio
n  {

        File  f  =  new  File("a.txt");
        FileOutputStream  fop  =  new  FileOutputStream(f);
        //  构建FileOutputStream对象,文件不存在会自动新建


  OutputStreamWriter  writer  =  new  OutputStreamWriter(fop,  "UTF
-8");
        //  构建OutputStreamWriter对象,参数可以指定编码,默认为操作
系统默认编码,windows上是gbk
```

```
            writer.append("中文输入");
            //  写入到缓冲区

            writer.append("\r\n");
            //换行

            writer.append("English");
            //  刷新缓存冲,写入到文件,如果下面已经没有写入的内容了,直接 close 也会写入

            writer.close();
            //关闭写入流,同时会把缓冲区内容写入文件,所以上面的注释掉

            fop.close();
            //  关闭输出流,释放系统资源

            FileInputStream  fip  =  new  FileInputStream(f);
            //  构建 FileInputStream 对象


    InputStreamReader  reader  =  new  InputStreamReader(fip,  "UTF-8
");
            //  构建 InputStreamReader 对象,编码与写入相同

            StringBuffer  sb  =  new  StringBuffer();
            while  (reader.ready())  {
                sb.append((char)  reader.read());
                //  转成 char 加到 StringBuffer 对象中
```

```
        }
        System.out.println(sb.toString());
        reader.close();
        //  关闭读取流

        fip.close();
        //  关闭输入流,释放系统资源

    }
}
```

文件和 I/O

还有一些关于文件和 I/O 的类，我们也需要知道：

File Class(类)

FileReader Class(类)

FileWriter Class(类)

实验 11 JAVA 异常处理

实验目的

1. 掌握 Java 异常处理

实验内容

Java 异常处理

异常是程序中的一些错误，但并不是所有的错误都是异常，并且错误有时候是可以避免的。

比如说，你的代码少了一个分号，那么运行出来结果是提示是错误 java.lang.Error；如果你用 System.out.println(11/0)，那么你是因为你用 0 做了除数，会抛出 java.lang.ArithmeticException 的异常。

异常发生的原因有很多，通常包含以下几大类：

用户输入了非法数据。

要打开的文件不存在。

网络通信时连接中断，或者 JVM 内存溢出。

这些异常有的是因为用户错误引起，有的是程序错误引起的，还有其它一些是因为物理错误引起的。-

要理解 Java 异常处理是如何工作的，你需要掌握以下三种类型的异常：

检查性异常：最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。

运行时异常： 运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。

错误： 错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

Exception 类的层次

所有的异常类是从 java.lang.Exception 类继承的子类。

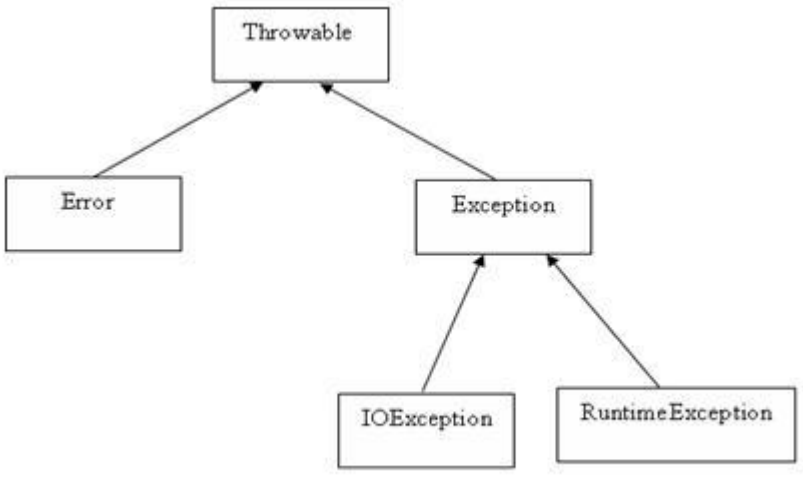
Exception 类是 Throwable 类的子类。除了 Exception 类外，Throwable 还有一个子类 Error 。

Java 程序通常不捕获错误。错误一般发生在严重故障时，它们在 Java 程序处理的范畴之外。

Error 用来指示运行时环境发生的错误。

例如，JVM 内存溢出。一般地，程序不会从错误中恢复。

异常类有两个主要的子类：IOException 类和 RuntimeException 类。



在 Java 内置类中(接下来会说明)，有大部分常用检查性和非检查性异常。

Java 内置异常类

Java 语言定义了一些异常类在 java.lang 标准包中。

标准运行时异常类的子类是最常见的异常类。由于 java.lang 包是默认加载到所有的 Java 程序的，所以大部分从运行时异常类继承而来的异常都可以直接使用。

Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

异常	描述
ArithmeticException	当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。
ArrayIndexOutOfBoundsException	用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。
ArrayStoreException	试图将错误类型的对象存储到一个对象数组时抛出的异常。
ClassCastException	当试图将对象强制转换为不是实例的子类时，抛出该异常。
IllegalArgumentException	抛出的异常表明向方法传递了一个不合法或不正确的参数。
IllegalMonitorStateException	抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。
IllegalStateException	在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环

	境或 Java 应用程序没有处于请求操作所要求的适当状态下。
IllegalThreadStateException	线程没有处于请求操作所要求的适当状态时抛出的异常。
IndexOutOfBoundsException	指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
NegativeArraySizeException	如果应用程序试图创建大小为负的数组，则抛出该异常。
NullPointerException	当应用程序试图在需要对象的地方使用 null 时，抛出该异常
NumberFormatException	当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
SecurityException	由安全管理器抛出的异常，指示存在安全侵犯。
StringIndexOutOfBoundsException	此异常由 String 方法抛出，指示索引或者为负，或者超出字符串的大小。
UnsupportedOperationException	当不支持请求的操作时，抛出该异常。

下面的表中列出了 Java 定义在 java.lang 包中的检查性异常类。

异常	描述
ClassNotFoundException	应用程序试图加载类时，找不到相应的类，抛出该异常。
CloneNotSupportedExceptio	当调用 Object 类中的 clone 方

n	法克隆对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。
IllegalAccessException	拒绝访问一个类的时候，抛出该异常。
InstantiationException	当试图使用 Class 类中的 newInstance 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。
InterruptedException	一个线程被另一个线程中断，抛出该异常。
NoSuchFieldException	请求的变量不存在
NoSuchMethodException	请求的方法不存在

异常方法

下面的列表是 Throwable 类的主要方法:

序号	方法及说明
1	public String getMessage() 返回关于发生的异常的详细信息。这个消息在 Throwable 类的构造函数中初始化了。
2	public Throwable getCause() 返回一个 Throwable 对象代表异常原因。
3	public String toString() 使用 getMessage()的结果返回类的串级名字。
4	public void printStackTrace() 打印 toString()结果和栈层次到 System.err，即错误输出流。

5	public StackTraceElement [] getStackTrace() 返回一个包含堆栈层次的数组。下标为 0 的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。
6	public Throwable fillInStackTrace() 用当前的调用栈层次填充 Throwable 对象栈层次,添加到栈层次任何先前信息中。

捕获异常

使用 try 和 catch 关键字可以捕获异常。try/catch 代码块放在异常可能发生的地方。

try/catch 代码块中的代码称为保护代码，使用 try/catch 的语法如下：

```
try
{
    //  程序代码
}catch(ExceptionName  e1)
{
    //Catch  块
}
```

Catch 语句包含要捕获异常类型的声明。当保护代码块中发生一个异常时，try 后面的 catch 块就会被检查。

如果发生的异常包含在 catch 块中，异常会被传递到该 catch 块，这和传递一个参数到方法是一样。

实例

下面的例子中声明有两个元素的一个数组,当代码试图访问数组的第三个元素的时候就会抛出一个异常。

```
//  文件
名  :  ExcepTest.javaimport  java.io.*;public  class  ExcepTest{

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

```
                try{
                        int  a[]  =  new  int[2];
                        System.out.println("Access  element  three  :"
  +  a[3]);
                }catch(ArrayIndexOutOfBoundsException  e){
                        System.out.println("Exception  thrown    :"  +
  e);
                }
                System.out.println("Out  of  the  block");
        }
    }
```

以上代码编译运行输出结果如下：

```
Exception  thrown    :java.lang.ArrayIndexOutOfBoundsException:
3
Out  of  the  block
```

多重捕获块

一个 try 代码块后面跟随多个 catch 代码块的情况就叫多重捕获。

多重捕获块的语法如下所示：

```
  try{
        //  程序代码
  }catch(异常类型 1  异常的变量名 1){
        //  程序代码
  }catch(异常类型 2  异常的变量名 2){
        //  程序代码
  }catch(异常类型 2  异常的变量名 2){
        //  程序代码
  }
```

上面的代码段包含了 3 个 catch 块。

可以在 try 语句后面添加任意数量的 catch 块。

如果保护代码中发生异常，异常被抛给第一个 catch 块。

如果抛出异常的数据类型与 ExceptionType1 匹配，它在这里就会被捕获。

如果不匹配，它会被传递给第二个 catch 块。

如此，直到异常被捕获或者通过所有的 catch 块。

实例

该实例展示了怎么使用多重 try/catch。

```
try
{
      file  =  new  FileInputStream(fileName);
      x  =  (byte)  file.read();
}catch(IOException  i)
{
      i.printStackTrace();
      return  -1;
}catch(FileNotFoundException  f)  //Not  valid!
{
      f.printStackTrace();
      return  -1;
}
```

---

throws/throw 关键字：

如果一个方法没有捕获一个检查性异常，那么该方法必须使用 throws 关键字来声明。throws 关键字放在方法签名的尾部。

也可以使用 throw 关键字抛出一个异常，无论它是新实例化的还是刚捕获到的。

下面方法的声明抛出一个 RemoteException 异常：

```
import  java.io.*;public  class  className{
      public  void  deposit(double  amount)  throws  RemoteExcept
```

```
ion
        {
            //  Method  implementation
            throw  new  RemoteException();
        }
        //Remainder  of  class  definition
    }
```

一个方法可以声明抛出多个异常，多个异常之间用逗号隔开。

例如，下面的方法声明抛出 RemoteException 和 InsufficientFundsException:

```
    import  java.io.*;public  class  className{
        public  void  withdraw(double  amount)  throws  RemoteExcep
tion,
                                                          Insuf
ficientFundsException
        {
            //  Method  implementation
        }
        //Remainder  of  class  definition
    }
```

---

finally 关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句。

finally 代码块出现在 catch 代码块最后，语法如下:

```
  try{
      //  程序代码
  }catch(异常类型 1  异常的变量名 1){
```

```
        //  程序代码
    }catch(异常类型 2  异常的变量名 2){
        //  程序代码
    }finally{
        //  程序代码
    }
```

实例

```
public  class  ExcepTest{

    public  static  void  main(String  args[]){
        int  a[]  =  new  int[2];
        try{
            System.out.println("Access  element  three  :"
+  a[3]);
        }catch(ArrayIndexOutOfBoundsException  e){
            System.out.println("Exception  thrown    :"  +
e);
        }
        finally{
            a[0]  =  6;
            System.out.println("First  element  value:  "
+a[0]);
            System.out.println("The  finally  statement  is
executed");
        }
    }
```

以上实例编译运行结果如下：

```
Exception  thrown    :java.lang.ArrayIndexOutOfBoundsException:
```

3

```
First element value: 6
The finally statement is executed
```

注意下面事项：

catch 不能独立于 try 存在。

在 try/catch 后面添加 finally 块并非强制性要求的。

try 代码后不能既没 catch 块也没 finally 块。

try, catch, finally 块之间不能添加任何代码。

声明自定义异常

在 Java 中你可以自定义异常。编写自己的异常类时需要记住下面的几点。

所有异常都必须是 Throwable 的子类。

如果希望写一个检查性异常类，则需要继承 Exception 类。

如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

可以像下面这样定义自己的异常类：

```
class  MyException  extends  Exception{
}
```

只继承 Exception 类来创建的异常类是检查性异常类。

下面的 InsufficientFundsException 类是用户定义的异常类，它继承自 Exception。

一个异常类和其它任何类一样，包含有变量和方法。

实例

```
//  文件名 InsufficientFundsException.javaimport  java.io.*;
public  class  InsufficientFundsException  extends  Exception{
      private  double  amount;
      public  InsufficientFundsException(double  amount)
      {
            this.amount  =  amount;
      }
      public  double  getAmount()
      {
            return  amount;
      }
}
```

为了展示如何使用我们自定义的异常类，

在下面的 CheckingAccount 类中包含一个 withdraw()方法抛出一个 InsufficientFundsException 异常。

```
//  文件名称  CheckingAccount.javaimport  java.io.*;
public  class  CheckingAccount{
      private  double  balance;
      private  int  number;
      public  CheckingAccount(int  number)
      {
            this.number  =  number;
```

```
        }
        public  void  deposit(double  amount)
        {
              balance  +=  amount;
        }
        public  void  withdraw(double  amount)  throws
                                                                  Insuf
ficientFundsException
        {
              if(amount  <=  balance)
                {
                      balance  -=  amount;
                }
                else
                {
                      double  needs  =  amount  -  balance;
                      throw  new  InsufficientFundsException(needs)
;
                }
          }
          public  double  getBalance()
          {
                return  balance;
          }
          public  int  getNumber()
          {
                return  number;
          }
    }
```

下面的BankDemo程序示范了如何调用CheckingAccount类的deposit() 和withdraw()方法。

```
//文件名称  BankDemo.javapublic  class  BankDemo{
      public  static  void  main(String  []  args)
      {
            CheckingAccount  c  =  new  CheckingAccount(101);
            System.out.println("Depositing  $500...");
            c.deposit(500.00);
            try
            {
                  System.out.println("\nWithdrawing  $100...");
                  c.withdraw(100.00);
                  System.out.println("\nWithdrawing  $600...");
                  c.withdraw(600.00);
            }catch(InsufficientFundsException  e)
            {
                  System.out.println("Sorry,  but  you  are  shor
t  $"

+  e.getAmount());
                  e.printStackTrace();
            }
        }
}
```

编译上面三个文件，并运行程序BankDemo，得到结果如下所示：

```
Depositing  $500...

Withdrawing  $100...
```

```
Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

实验 12 JAVA 继承

实验目的

1. 掌握 Java 继承的使用

实验内容

Java 继承

继承是 java 面向对象编程技术的一块基石，因为它允许创建分等级层次的类。继承可以理解为一个对象从另一个对象获取属性的过程。

如果类 A 是类 B 的父类，而类 B 是类 C 的父类，我们也称 C 是 A 的子类，类 C 是从类 A 继承而来的。在 Java 中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类

继承中最常使用的两个关键字是 extends 和 implements。

这两个关键字的使用决定了一个对象和另一个对象是否是 IS-A(是一个)关系。

通过使用这两个关键字，我们能实现一个对象获取另一个对象的属性。

所有 Java 的类均是由 java.lang.Object 类继承而来的，所以 Object 是所有类的祖先类，而除了 Object 外，所有类必须有一个父类。

通过过 extends 关键字可以申明一个类是继承另外一个类而来的,一般形式如下：

```
// A.javapublic class A {
    private int i;
    protected int j;

    public void func() {

    }
}
 // B.javapublic class B extends A {
}
```

以上的代码片段说明，B 由 A 继承而来的，B 是 A 的子类。而 A 是 Object 的子类，这里可以不显示地声明。

作为子类，B 的实例拥有 A 所有的成员变量，但对于 private 的成员变量 B 却没有访问权限，这保障了 A 的封装性。

IS-A 关系

IS-A 就是说:一个对象是另一个对象的一个分类。

下面是使用关键字 extends 实现继承。

```
public class Animal{
}
public class Mammal extends Animal{
}
public class Reptile extends Animal{
}
public class Dog extends Mammal{
}
```

基于上面的例子，以下说法是正确的：

Animal 类是 Mammal 类的父类。

Animal 类是 Reptile 类的父类。

Mammal 类和 Reptile 类是 Animal 类的子类。

Dog 类既是 Mammal 类的子类又是 Animal 类的子类。

分析以上示例中的 IS-A 关系，如下：

Mammal IS-A Animal

Reptile IS-A Animal

Dog IS-A Mammal

因此 : Dog IS-A Animal

通过使用关键字 extends，子类可以继承父类的除 private 属性外所有的属性。

我们通过使用 instanceof 操作符，能够确定 Mammal IS-A Animal

实例

```
public class Dog extends Mammal{

   public static void main(String args[]){

      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

以上实例编译运行结果如下：

```
truetruetrue
```

介绍完 extends 关键字之后，我们再来看下 implements 关键字是怎样使用来表示 IS-A 关系。

Implements 关键字使用在类继承接口的情况下， 这种情况不能使用关键字 extends。

实例

```
public interface Animal {}
public class Mammal implements Animal{
}
public class Dog extends Mammal{
}
```

---

instanceof 关键字

可以使用 instanceof 运算符来检验 Mammal 和 dog 对象是否是 Animal

类的一个实例。

```
interface Animal{}
class Mammal implements Animal{}
public class Dog extends Mammal{
   public static void main(String args[]){

      Mammal m = new Mammal();
      Dog d = new Dog();

      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

以上实例编译运行结果如下：

```
truetruetrue
```

HAS-A 关系

HAS-A 代表类和它的成员之间的从属关系。这有助于代码的重用和减少代码的错误。

例子

```
public class Vehicle{}public class Speed{}public class Van extends Vehicle{
   private Speed sp;
}
```

Van 类和 Speed 类是 HAS-A 关系(Van 有一个 Speed)，这样就不用将 Speed 类的全部代码粘贴到Van类中了,并且Speed类也可以重复利用于多个应用程序。

在面向对象特性中，用户不必担心类的内部怎样实现。

Van 类将实现的细节对用户隐藏起来，因此，用户只需要知道怎样调用 Van

类来完成某一功能，而不必知道 Van 类是自己来做还是调用其他类来做这些工作。

Java 只支持单继承，也就是说，一个类不能继承多个类。

下面的做法是不合法的：

public class extends Animal, Mammal{}

Java 只支持单继承（继承基本类和抽象类），但是我们可以用接口来实现（多继承接口来实现）,脚本结构如：

public class Apple extends Fruit implements Fruit1, Fruit2{}

一般我们继承基本类和抽象类用 extends 关键字，实现接口类的继承用 implements 关键字。