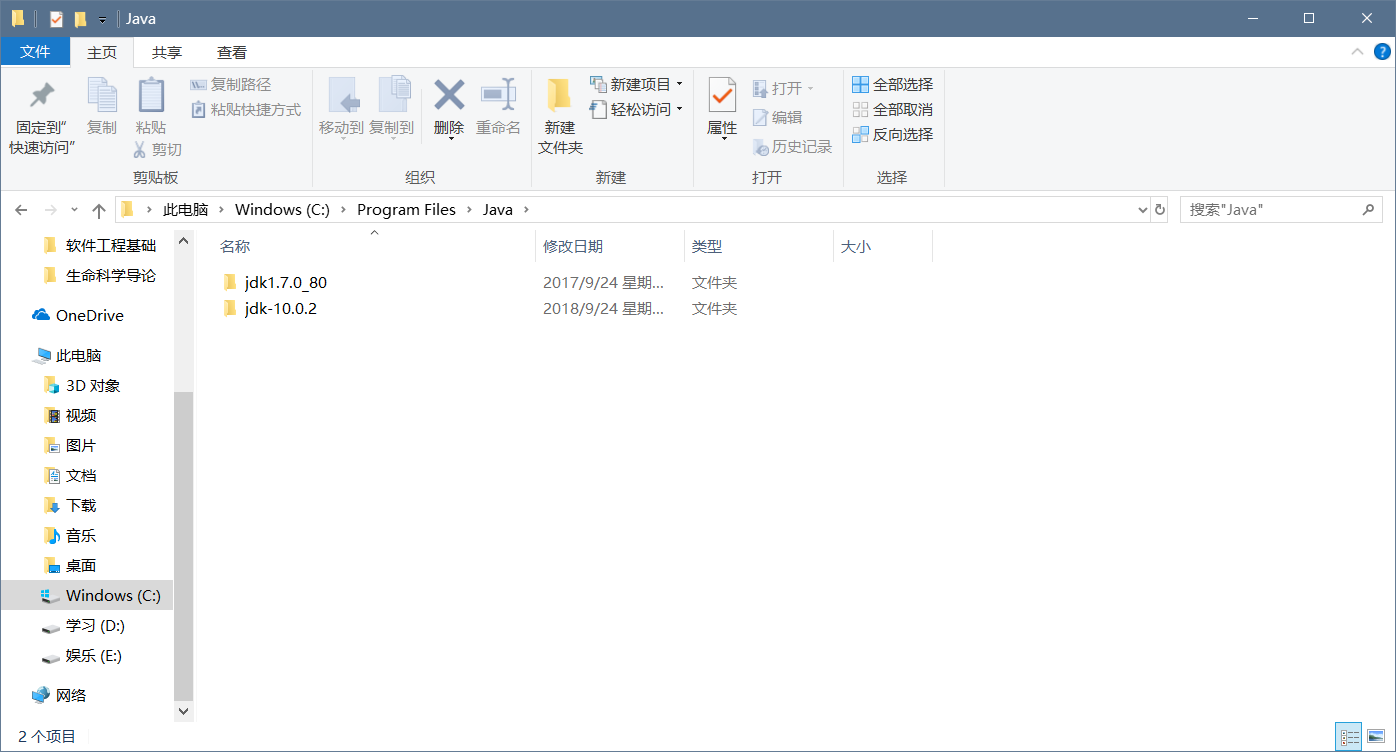
**面向对象程序设计（Java）**

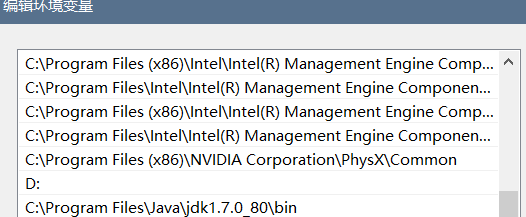
陆圣珩

2017221302009

**第一章：**

1. 下载并安装JDK软件包，尝试阅读其中的JDK文档。





1. 怎样区分Java Application和Applet程序？

Applet程序没有含 main方法的主类，但一定有一个从java.applet.Applet派生的类，它是由Java系统提供的，不能单独运行，必须依附于一个用HTML语言编写的网页。

每个Java Application程序必定含有一个并且只有一个main方法，Java Application程序都含有一个主类。

1. Java语言的特点？

* 平台无关性：编译器所生成的可执行代码是基于抽象处理器─Java虚拟机(JVM：Java Virual Machine)来实现。Java虚拟机就是虚拟运行Java代码的假想计算机，其定义为：运行经过编译的Java目标代码的计算机的实现。编译生成的代码不针对任何具体的硬件体系结构和软件平台的代码--“字节码”。
* 简洁性：由C++衍生而来，其语言风格与C++类似，但进行了很大的简化和改进。抛弃了C++中一些不是绝对必要的东西，如头文件、指针、结构、联合、隐式的类型转换和操作符重载等Java支持单重继承，但接口(interface)能实现多继承，没有多继承混乱、复杂的问题。
* 安全性：对内存访问是通过对象实例变量实现的，防止在网络系统或分布系统环境下特洛伊木马等手段访问对象的私有成员。Java语言不支持指针，避免了指针操作的安全隐患。Java语言提供的内存管理机制，有自动搜集“内存垃圾”程序。

1. Java包含哪三个版本，每个版本有什么功能，各个版本的应用领域？

Java Standard Edition：即Java标准版，是Java所有版本的基本，提供了Java最基本的语言特性；

Java移动版：在此基础上将一些功能进行缩减，并软提高了其运行效率，又发展出了针对便携设备、智能家电的Java Mirco Edition

Java企业版：在Java SE的版本之上，又加上分布计算、网络支持等支持大型企业应用的功能后，又发展出了Java Enterprise Edition。

1. 见实验报告一。
2. 见实验报告一。

**第二章：**

1. Java语言使用的哪一种字符集，储存空间是多少？

Java使用的是Unicode，又称为统一码字符集，使用的是16位存储空间，能支持多种语言，更具有国际化特性。

1. 在Java语言中，数据类型分为两大类，具体的类型有哪些？

整数类型、浮点类型、字符型、 布尔类型、数组、类、接口

1. 简单数据类型所需要的存储开销个是多少，它们的值范围是什么？

|  |  |  |
| --- | --- | --- |
| 类型 | 存储(bit) | 范围 |
| **boolean** | **1** | Ture、Flase |
| **char** | **16** |  |
| **byte** | **8** | -128~127 |
| **short** | **16** | -32768~32767 |
| **int** | **32** | -2147483648~2147483647 |
| **long** | **64** | -9223372036854774808~9223372036854774807 |
| **float** | **32** | 3.402823e+38 ~ 1.401298e-45 |
| **double** | **64** | 1.797693e+308~ 4.9000000e-324 |
| **void** | **–** |  |

1. Java语言的逻辑运算的优化的含义是什么？

&& 运算符：opB1 && opB2

如果opB1值为false，则运算式的值就是false，程序不会访问opB2；

|| 运算符：opB1 || opB2

如果opB1值为true，则运算式的值就是true，程序不会访问opB2；

1. 数据类型转换是什么含义？有哪几种？

自动转换：int a = 10; long b = a;

强制转换： long b = 10; int a = (int)b;

1. 变量的作用域分为几种，各有什么特点？

类级变量在类定义后就已经存在，占用内存空间，可以直接通过类名来访问，不需要实例化。

对象实例级变量就是成员变量，实例化后才能分配内存空间，才能访问。

方法级变量就是在方法内部定义的变量，即局部变量。

块级变量就是定义在一对大括号“{}”中的变量，变量的生命周期就是这个块。

1. 见实验报告。

**第三章：**

1. 在Java语言中，流程控制分为哪些？

Java语言中的流程控制语句提供了控制程序执行顺序的手段。流程控制是程序代码的重要部分。

流程控制语句分为：分支语句、循环语句、异常处理语句和跳转语句。

1. 分支语句if-else和switch在断定条件上各为何种数据类型？

If —— Boolean

Switch —— int

1. 循环语句中由哪几个部分组成，各有何作用？

1、循环初始化部分——初始化循环控制变量、循环体所用到变量；   
2、循环体部分——循环结构的主体；   
3、循环调整部分——循环控制变量的修改、或循环终止条件的检查；   
4、循环控制部分——程序执行的控制转移。

1. 比较break和continue语句的区别。

中断控制流语句break：  
break语句用于循环结构中，当程序执行break语句时，程序流程就结束循环。

break语句也可以带语句标记，它的作用是结束该语句标记的语句块。

continue语句用于循环结构中，当程序执行continue语句时，程序流程就结束本次循环，充当了循环体的最后一条语句作用。

1. 数组的声明形式有哪些，其初始化是什么含义？

它们的声明的形式为：

* + - 1. type[] arrayName; 或 type arrayName[];

声明数组的时候不能指定其大小

* + - 1. intArray0 =**new int**[3]**;**

如果创建数组时不指定大小则必须初始化

* + - 1. intArray1 = **new int**[]{0,1,2}; **int** intArray3 [] = {1,2};

不指定数组大小则必须在创建的同时初始化数组

1. 数组的length属性指的是什么，多维数组length的用法？

数组长度，多维数组a直接a.length是第一维，a[0].length是第二维，以此类推。

1. 见实验报告。
2. 见实验报告。

**第四章：**

1. 面向对象程序设计的三个特性是什么？

封装性、继承性、多态性

2.如何用类的成员来表示对象的属性和行为的？举例说明。

用变量来表示属性，用方法来实现行为，例如：

class Person{

String name;

void talk()

void sleep()

}

3.类的成员变量是如何获得初始化的值的？

创建变量时初始化值、创建时赋初值、在类的构造器中对成员变量赋初值

4.方法的数据传递方式有几种，它们各有什么特点？举例说明。

方法在被调用时，其参数的数据传递是值传递，即实际参数传值给形式参数。

形式参数是简单类型:

在方法调用时，实际参数将其存储单元的数据赋值给形式参数.

形式参数是引用类型:

在方法中，如果引用类型的参数没有发生引用的改变，则形式参数对引用中的变量值的改变会影响实际参数引用中变量的值.

在方法中，如果引用类型的参数发生引用的改变，则不会影响实际参数引用中变量的值.

5.方法过载是指什么，创建方法时如何才能使方法过载？举例说明。

* + 在同一个类中创建的具有相同方法名，但是参数不同的方法。
  + 参数不同：

数量不同；

数量相同，但是对应的类型不同。

* + 方法过载也是多态性表现之一，编译时多态。
  + 方法过载中的方法由调用时的实参决定调用的方法是那个。

示例：

void setData(int a, int b)

void setData(Point p) //个数不同

void setData(float a, float b) //类型不同

void setData() //个数不同

6. 实例成员和类成员有什么不同，它们如何使用？

静态成员是指声明时，带有 static关键字的成员。

实例成员是指声明时，不带 static关键字的成员。

静态成员在调用时，需要使用类名直接调用；而实例成员在调用时，首先需要创建类的对象，然后通过对象来进行调用。

7.对象与类的实例是什么关系，对象的引用是指什么？举例说明。

对象：是一个变量，它的存储开销是一个地址存储单元。

实例：是在存储空间分配的存储堆。是垃圾回收的目标。

引用：对象通过它存储的实例的起始地址对实例实现访问。

8.对象是如何创建和使用的，又是什么情况下回收的？

对象的生命周期：

对象使用

通过对象名对成员的访问：

<对象名>·<成员>

销毁：

垃圾回收：实例开销的回收。

由JVM自动完成。

调用finalize()方法处理。

9. 见实验报告。

10. 见实验报告

11. 见实验报告

**第五章、超类、子类和继承性**

1、java中类的继承性有什么特点

继承性特性如下：

单一继承性：子类只能有一个超类，而超类可以有多个子类；

子类继承超类的所有成员；

子类可以创建自己的成员；

子类不能继承超类的构造器，只能在构造器中通过super()调用超类的构造器；

子类的构造器首先要调用超类的构造器；

多态性之一：子类的成员隐藏和覆盖超类中相同的成员；

多态性之二：超类的对象可以对子类的实例引用；

由abstract和final修饰的类指示该类是否必须或不能被继承。

1. 子类可以继承超类的什么，不能继承的如何在子类中访问

子类继承超类的所有成员，

子类不能继承超类的构造器，只能在构造器中通过super()调用超类的构造器；

3、子类对超类的扩展表现在哪些方面？举例说明

继承，覆盖，添加新方法

4、方法覆盖是指什么，它与方法过载有什么不同？举例说明

覆盖是指在子类中对超类的方法进行重定义，过载是指在同一个类中同一方法名由于参数的不同表现不同

5、在java中有哪些是多态的表现？

继承、重写、父类引用指向子类对象

6、用final和abstract修饰的类各有什么特点？

final类不可被继承

Abstract类必须被继承

7、用final和abstract修饰的方法各有什么特点？

final方法不可被覆盖

Abstract方法必须被覆盖

8、举例说明Object类方法equals()的使用情况

equals方法对于字符串来说是比较内容的，而对于非字符串来说是比较其指向的对象是否相同的。

9. 见实验报告。

10. 见实验报告。

**第六章**

1. 包：包实际上是一组类组成的集合，也称之为类库。包的层次结构与文件系统的文件目录结构是相似的。包名是Java的合法标识符，一般都用小写的字母单词表示。Java语言提供了一些常用的基本类包，如java.io和java.lang。使用时需要使用import语句引入
2. package语句作为Java源文件的第一条语句，指明该文件中定义的类所在的包，若缺省该语句，则指定为无名包。格式：

package pkgName1[.pkgName2[.pkgName3...]];

其中：pkgName1～pkgNameN表示包的目录层次。它对应于文件系统的目录结构。

import类引入语句:

引入语句提供了能使用Java中API或用户已创建的类。引入语句是在包语句(如果有的话)之后的任何条语句。格式为：

import pack1[.pack2...].<className|\*>;

pack1～packN为包的层次结构，它对应着要访问的类所在文件目录结构;

className则指明所要引入的类，如果要从一个包中引入多个类时，则可以用星号(\*)来表示。

使用“\*”引入语句，只表示了源程序中所需要的类会在包中找到并引入，但是对包中其它的类或它下面的包中的类并不引入。

1. 至少需要一个public类

6、

接口提供统一的对外操作方法  
抽象类提供基本统一的可以不对外的共用元素

7. 见实验报告。

8. 见实验报告。

**第七章**

1. 异常就是程序执行过程中出现的不正常现象。非预期情况，错误的参数、网络故障。任何一个程序都可能出现异常，Java使用对象表示对打开的文件不存在、内存不够、数组访问超界等非预期情况。Java使异常处理标准化，使程序设计思路更清楚，理解更容易。发生异常时，就会抛出一个异常，通过捕获这个异常，就可以进行相应异常处理。

|  |
| --- |
| try {  正常程序段;  }  catch(异常类1 异常变量) {  与异常类1有关的处理程序段;  }  catch(异常类2 异常变量) {  与异常类2有关的处理程序段;  }  ......  finally {  退出异常处理程序段;  } |

4、5、

throws是方法可能抛出异常的声明。

语法：[(修饰符)](返回值类型)(方法名)([参数列表])[throws(异常类)]{......}

如：

public void function() throws Exception{......}

当某个方法可能会抛出某种异常时用于throws 声明可能抛出的异常，然后交给上层调用它的方法程序处理。

1. 见实验报告。

**第八章**

2.

|  |
| --- |
| class Local{ static int *local* = 1;  }  class Add implements Runnable{  public void run() {  for(int i=0;i<10;i++) {  System.*out*.println(++Local.*local*);;  }  } }  class Less implements Runnable{  public void run() {  for(int i=0;i<10;i++) {  System.*out*.println(--Local.*local*);;  }  } }  public class Test8\_2 {  public static void main(String args[]) {  Add test1 = new Add();  Less test2 = new Less();  Add test3 = new Add();  Less test4 = new Less();  new Thread(test1).start();  new Thread(test2).start();  new Thread(test3).start();  new Thread(test4).start();  } } |

3

|  |
| --- |
| class Lock{  int lockon;  Lock(){  this.lockon = 1;  } }  class ThTest implements Runnable{  Lock lock;  ThTest(Lock lock){  this.lock = lock;  }  public void run() {  synchronized(lock) {  while(lock.lockon==1) {  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.*out*.println(lock.lockon+"Lock is down.Next it will be on.");  try {  Thread.*sleep*(5);  } catch (InterruptedException e) {  e.printStackTrace();  }  lock.lockon = 1;  lock.notify();  }  } }  class ThRest implements Runnable{  Lock lock;  ThRest(Lock lock){  this.lock = lock;  }  public void run() {  synchronized(lock) {  while(lock.lockon==0) {  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.*out*.println(lock.lockon+"Lock is on.Next it will be down.");  try {  Thread.*sleep*(5);  } catch (InterruptedException e) {  e.printStackTrace();  }  lock.lockon = 0;  lock.notify();  }  } }  public class Test8\_3 {  public static void main(String args[]) {  Lock lock = new Lock();  ThTest thtest = new ThTest(lock);  ThRest threst = new ThRest(lock);  new Thread(thtest).start();  new Thread(threst).start();  } } |

4.

|  |
| --- |
| public class Test1 {  private static Integer *count* = 0;  private static final Integer *FULL* = 10;  private static String *LOCK* = "lock";   public static void main(String[] args) {  Test1 test1 = new Test1();  new Thread(test1.new Producer()).start();  new Thread(test1.new Consumer()).start();  new Thread(test1.new Producer()).start();  new Thread(test1.new Consumer()).start();  new Thread(test1.new Producer()).start();  new Thread(test1.new Consumer()).start();  new Thread(test1.new Producer()).start();  new Thread(test1.new Consumer()).start();  }   class Producer implements Runnable {  @Override  public void run() {  for (int i = 0; i < 10; i++) {  try {  Thread.*sleep*(3000);  } catch (Exception e) {  e.printStackTrace();  }  synchronized (*LOCK*) {  while (*count* == *FULL*) {  try {  *LOCK*.wait();  } catch (Exception e) {  e.printStackTrace();  }  }  *count*++;  System.*out*.println(Thread.*currentThread*().getName() + "生产者生产，目前总共有" + *count*);  *LOCK*.notifyAll();  }  }  }  }   class Consumer implements Runnable {  @Override  public void run() {  for (int i = 0; i < 10; i++) {  try {  Thread.*sleep*(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  synchronized (*LOCK*) {  while (*count* == 0) {  try {  *LOCK*.wait();  } catch (Exception e) {  }  }  *count*--;  System.*out*.println(Thread.*currentThread*().getName() + "消费者消费，目前总共有" + *count*);  *LOCK*.notifyAll();  }  }  }  } } |

5.

|  |
| --- |
| import java.io.IOException; import java.io.PipedInputStream; import java.io.PipedOutputStream;  class MyProducer extends Thread {   private PipedOutputStream outputStream;   private int index = 0;   public MyProducer(PipedOutputStream outputStream) {  this.outputStream = outputStream;  }   @Override  public void run() {  while (true) {  try {  for (int i = 0; i < 5; i++) {  outputStream.write(index++);  }  } catch (IOException e) {  e.printStackTrace();  }   try {  Thread.*sleep*(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } }  class MyConsumer extends Thread {   private PipedInputStream inputStream;   public MyConsumer(PipedInputStream inputStream) {  this.inputStream = inputStream;  }   @Override  public void run() {  while (true) {  try {  Thread.*sleep*(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  int count = inputStream.available();  if (count > 0) {  System.*out*.println("rest product count: " + count);  System.*out*.println("get product: " + inputStream.read());  }  } catch (IOException e1) {  e1.printStackTrace();  }  }  } }  public class PipeTest1 {   public static void main(String[] args) {   PipedOutputStream pos = new PipedOutputStream();  PipedInputStream pis = new PipedInputStream();  try {  pis.connect(pos);  } catch (IOException e) {  e.printStackTrace();  }   new MyProducer(pos).start();  new MyConsumer(pis).start();   } } |

6. 见实验报告

**第九章**

1、流(stream)是指在计算机的输入与输出之间运动的数据序列。

输入流：InputStream

输出流：OutputStream

1. 字节流和字符流

3、

（1）“对象序列化”（Object Serialization）是Java一种特性。类实现 Serializable接口。实现了Serializable接口的对象，可将它们转换成一系列字节，并可在以后完全恢复回原来的样子。这一过程可通过网络进行。这意味着序列化机制能自动补偿操作系统间的差异。可以实现“有限持久化”。意味着对象的“生存时间”并不取决于程序是否正在执行——它存在或“生存”于程序的每一次调用之间。

（2）序列化一个对象：首先要创建某些OutputStream对象，然后将其封装到ObjectOutputStream对象内。再需调用writeObject()即可完成对象的序列化，并将其发送给OutputStream。相反将一个InputStream封装到ObjectInputStream内，然后调用readObject()。和往常一样，最后获得的是指向一个上溯造型Object的句柄，所以必须下溯造型(转换为相应的对象) 。

4、

|  |
| --- |
| import java.util.\*; import java.io.\*;  public class WriteTest{  public void write(String comment){  FileWriter fw=null;  try{  File f=new File("F:\\test.txt");  fw=new FileWriter(f,true);  }  catch(IOException e){e.printStackTrace();}  PrintWriter pw=new PrintWriter(fw);  pw.println(comment);  pw.flush();  try{  fw.flush();  pw.close();  fw.close();  }  catch(IOException e){e.printStackTrace();}  }  public static void main(String args[]){  WriteTest a=new WriteTest();  Scanner input=new Scanner(System.*in*);  String comment=input.nextLine();  a.write(comment);  input.close();  } } |

8**、**InputStreamReader和OutputStreamWriter

9、如果要一次输入或输出多个字节,系统不一定能直接接受,缓冲区可以把多个字节先放在里面,然后系统到里面去读取数据.输入或输出的速度比系统读取速度快,缓冲区起到缓解速度不匹配的问题。

10、见实验报告。

11、

|  |
| --- |
| import java.io.IOException; import java.io.PipedInputStream; import java.io.PipedOutputStream;  class MyProducer extends Thread {   private PipedOutputStream outputStream;   private int index = 0;   public MyProducer(PipedOutputStream outputStream) {  this.outputStream = outputStream;  }   @Override  public void run() {  while (true) {  try {  for (int i = 0; i < 5; i++) {  outputStream.write(index++);  }  } catch (IOException e) {  e.printStackTrace();  }   try {  Thread.*sleep*(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } }  class MyConsumer extends Thread {   private PipedInputStream inputStream;   public MyConsumer(PipedInputStream inputStream) {  this.inputStream = inputStream;  }   @Override  public void run() {  while (true) {  try {  Thread.*sleep*(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  int count = inputStream.available();  if (count > 0) {  System.*out*.println("rest product count: " + count);  System.*out*.println("get product: " + inputStream.read());  }  } catch (IOException e1) {  e1.printStackTrace();  }  }  } }  public class PipeTest1 {   public static void main(String[] args) {   PipedOutputStream pos = new PipedOutputStream();  PipedInputStream pis = new PipedInputStream();  try {  pis.connect(pos);  } catch (IOException e) {  e.printStackTrace();  }   new MyProducer(pos).start();  new MyConsumer(pis).start();   } } |

**第十章**

1、

Accept方法用于产生"阻塞"，直到接受到一个连接，并且返回一个客户端的Socket对象实例。"阻塞"是一个术语，它使程序运行暂时"停留"在这个地方，直到一个会话产生，然后程序继续；通常"阻塞"是由循环产生的。

.getInputStream方法获得网络连接输入，同时返回一个IutputStream对象实例。

.getOutputStream方法连接的另一端将得到输入，同时返回一个OutputStream对象实例。

2、客户端：建立套接字，得到数据流，发送信息，关闭数据流

服务端：监听端口，建立连接，接收数据流，反馈，断开连接

3、

|  |
| --- |
| public static void downLoadFromUrl(String urlStr,String fileName,String savePath) throws IOException{   URL url = new URL(urlStr);  HttpURLConnection conn = (HttpURLConnection)url.openConnection();*//3秒* conn.setConnectTimeout(3\*1000);   conn.setRequestProperty("User-Agent", "Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)");    InputStream inputStream = conn.getInputStream();   byte[] getData = readInputStream(inputStream);   File saveDir = new File(savePath);  if(!saveDir.exists()){  saveDir.mkdir();  }  File file = new File(saveDir+File.separator+fileName);  FileOutputStream fos = new FileOutputStream(file);  fos.write(getData);  if(fos!=null){  fos.close();  }  if(inputStream!=null){  inputStream.close();  }    System.out.println("info:"+url+" download success");   }    public static byte[] readInputStream(InputStream inputStream) throws IOException {  byte[] buffer = new byte[1024];  int len = 0;  ByteArrayOutputStream bos = new ByteArrayOutputStream();  while((len = inputStream.read(buffer)) != -1) {  bos.write(buffer, 0, len);  }  bos.close();  return bos.toByteArray();  }  public static void main(String[] args) {  try{  downLoadFromUrl("testurl");  }catch (Exception e) {  *// TODO: handle exception* }  } |

4**、**

|  |
| --- |
| package com.inetaddress;  import java.io.IOException; import java.net.InetAddress; import java.net.UnknownHostException;  public class Address {  public static void main(String[] args) throws IOException{  InetAddress inetAddress;*//声明InetAddress对象* try {  inetAddress=InetAddress.*getLocalHost*();*//实例化InetAddress对象，返回本地主机* String hostName=inetAddress.getHostName();*//获取本地主机名* String canonicalHostName=inetAddress.getCanonicalHostName();*//获取此 IP地址的完全限定域名* byte[] address=inetAddress.getAddress();*//获取原始IP地址* int a=0;  if(address[3]<0){  a=address[3]+256;  }  String hostAddress=inetAddress.getHostAddress();*//获取本地主机的IP地址* boolean reachable=inetAddress.isReachable(2000);*//获取布尔类型，看是否能到达此IP地址* System.*out*.println(inetAddress.toString());  System.*out*.println("主机名为："+hostName);*//输出本地主机名* System.*out*.println("此IP地址的完全限定域名："+canonicalHostName);*//输出此IP地址的完全限定域名* System.*out*.println("原始IP地址为："+address[0]+"."+address[1]+"."+address[2]+"."+a);*//输出本地主机的原始IP地址* System.*out*.println("IP地址为："+hostAddress);*//输出本地主机的IP地址* System.*out*.println("是否能到达此IP地址："+reachable);  } catch (UnknownHostException e) {  *// TODO Auto-generated catch block* e.printStackTrace();  }  } } |

6. 见实验报告