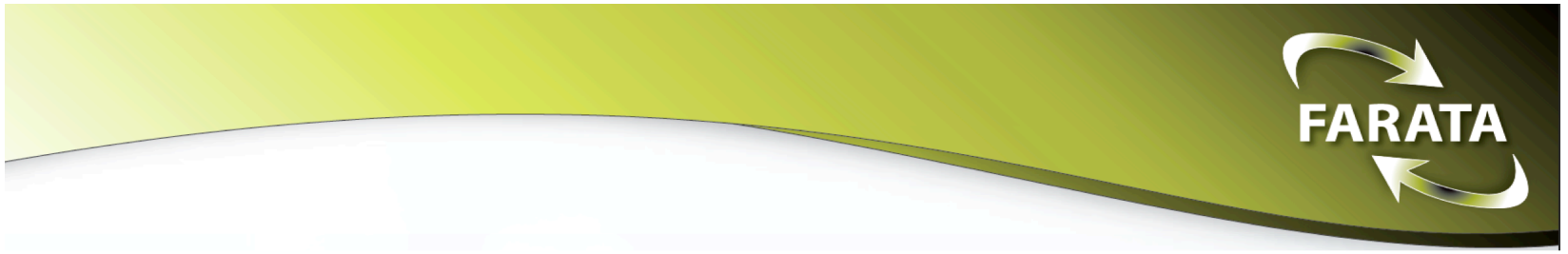

Java Programming
Unit 9

Working with I/O Streams.
Java Serialization.
Basic Networking.

Java I/O Streams From java.io

not to be confused with Java Streams API from java.util.stream

Input and Output Streams

- A Java program can read sequences of bytes from an *input I/O stream* (or write into an *output stream*): byte after byte, character after character, primitive after primitive.

- Some classes are meant for reading character streams such as `Reader` and `Writer`.

  `DataInputStream` and `DataOutputStream` can read and write Java primitives.

  To work with files you may use such classes as `FileInputStream`, `FileReader`, and others.

Classes that work with streams are located in two packages: *java.io* and *java.nio (non blocking i/o)*.

Three Steps of Working With I/O Streams

1. Open a stream that points at a specific data source: a file, a socket, a URL, and so on.

2. Read or write data from/to this stream.

3. Close the stream unless it's auto-closable.

Reading From `FileInputStream`

```
FileInputStream myFile = null;
  try {
       myFile = new  FileInputStream("abc.dat");
       boolean eof = false;

       while (!eof) {
          int byteValue = myFile.read();
          System.out.print(byteValue + " ");
          if (byteValue  == -1){
                    eof = true;
           }
       }
       // myFile.close(); // do not do it here!!!

  } catch (IOException e) {
    System.out.println("Could not read file: " +
                                  e.toString());
  } finally{
     if (myFile !=null){
        try{
            myFile.close();
        } catch (Exception e1){
             e1.printStackTrace();
        }
    }
  }
}
```

While reading with `FileInputStream`, the end of the file is represented by a negative 1.

Writing Into `FileOutputStream`

```
// byte values are represented by integers from 0 to 255
  int somedata[]= {56,230,123,43,11,37};
  FileOutputStream myFile = null;

  try {
     myFile = new  FileOutputStream("xyz.dat");
     for (int i = 0; i <somedata.length; i++){
        myFile.write(somedata[i]);
     }
 } catch (IOException e) {
       System.out.println("Could not write to a  file: " + e.toString());
  } finally{

     if (myFile !=null){
        try{
          myFile.close();
        } catch (Exception e1){
          e1.printStackTrace();
        }

     }
  }
}
```

Buffered I/O Streams

To minimize the number of times the disk is accessed use *buffers*, which serve as reservoirs of data.

You don't want to access disk 1000 time to read 1000 bytes.

The class `BufferedInputStream` works as a middleman between `FileInputStream` and the file itself. It reads a big chunk of bytes from a file into memory in one shot, and the `FileInputStream` object then reads single bytes from there.

Chaining I/O Streams

```
FileInputStream myFile = null;
BufferedInputStream buff =null;

try {
     myFile = new  FileInputStream("abc.dat");
     buff = new BufferedInputStream(myFile);
            boolean eof = false;
            while (!eof) {
                int byteValue = buff.read();
                System.out.print(byteValue + " ");
                if (byteValue  == -1)
                    eof = true;
             }
} catch (IOException e) {
  // . . .
} finally{
     if (myFile !=null){
        try{
          buff.close();
          myFile.close();
        } catch (Exception e1){
          e1.printStackTrace();
        }
     }
 }
```

Reading Character Streams

Text in Java is represented as a set of `char` values (two-byte characters), which are based on the Unicode Standard.

The Java classes `FileReader` and `FileWriter` were created to work with text files.

The recommended way is to pipe the class `InputStreamReader` with specified encoding and the `FileInputStream`.

```
// Can use either StringBuffer or StringBuilder here
StringBuffer buffer = new StringBuffer();

    try {
     FileInputStream myFile =
                    new FileInputStream("abc.txt");
     InputStreamReader inputStreamReader =
             new InputStreamReader(myFile, "UTF8");

      Reader reader =
              new BufferedReader(inputStreamReader);

      int ch;  // the code of one character

      while ((ch = reader.read()) > -1) {
          buffer.append((char)ch);
      }

      String result = buffer.toString();

    } catch (IOException e) {
        . . .
    }
```

Writing Into Character Streams

To write characters to a file, pipe `FileOutputStream` and `OutputStreamWriter`.

For efficiency, use `BufferedWriter`:

```
try{
 String myAddress = "123 Broadway, New York, NY 10011";
 FileOutputStream myFile = new FileOutputStream("abc.txt");

 Writer out
    = new BufferedWriter(new OutputStreamWriter(myFile, "UTF8"));

 out.write(myAddress);

} catch(IOException e){
      // . . .
}
```

Walkthrough 1

1. Download and import into Eclipse the source code of Lesson 16 from the textbook. Open the project properties and change the Compiler to use JDK 1.7 and replace the library in Java Build Path to be JRE 1.7.

2. Review the code and run the TestFileInputStream program. Modify the code to use try-with-resources.

Review the code and run TestBufferedInputStream program. Modify the code to use try-with-resources:

```
try ( FileInputStream myFile = new  FileInputStream("abc.dat");
      BufferedInputStream buff = new BufferedInputStream(myFile);)
```

3. Review the code and run TaxGUIFile program.

You can find the solutions for this walkthrough at https://github.com/yfain/javacodesamples

Files API

The class Files simplifies such operations as create, update, delete, and copy of files.
The class Path is a programmatic representation of a path in a file system.

```
Path pathCustomers= FileSystems.getDefault().getPath(".","c:\\practicalJava\\Customers.txt");
Path pathCustomers= FileSystems.getDefault().getPath(".","/practicalJava/Customers.txt");
```

Read file as strings:

```
List<String> customers=Files.readAllLines(pathCustomers, Charset.defaultCharset());
```

Read file as bytes:

```
byte[] customers=Files.readAllBytes(pathCustomers);
```

You still can use buffered reads for efficiency:

```
Reader reader=Files.newBufferedReader(pathCustomers, Charset.defaultCharset());
```

To create a file Customers.txt:

```
Path fileName=  Paths.get("c:\\practicalJava\\Customers.txt");
Path customers=Paths.createFile(fileName);
```

Java Serialization

Java Object Serialization

Object Serialization supports the encoding of objects into a stream of bytes. Serialization also supports the reconstruction of the object graph from a stream.

Serialization is used for communication via sockets or Java Remote Method Invocation (RMI).

Sample Use Case

1. Create an instance of a class Tax, and then serialize it into a sequence of bytes.
2. Store this sequence of bytes on a disk or send it over the network (**serialization**).
3. Recreate the instance of the class (**de-serialization**) Tax in memory of another JVM.

Marker Interface `Serializable`

A ClassA creates an instance of the object Employee, which has the fields firstName, lastName, salary, etc.

The values of these fields (a.k.a. *object state*) have to be saved in a stream.

The ClassB, which needs these data, has to re-create the object Employee in memory. The instances of ClassA and ClassB usually live in two different JVMs running on different computers.

```
class Employee implements java.io.Serializable{

     String lastName;
     String  firstName;
     double salary;
}
```

Rather than sending to a stream *one property* at a time, send *one object* at a time using `ObjectOutputStream` and `ObjectInputStream`.

ObjectOutputStream

Serializing the Employee object into the file c:\practicalJava\BestEmployee.ser.

```
class ClassA {
  public static void main(String args[]){
       Employee emp = new Employee();
       emp.lName = "John";
       emp.fName = "Smith";
       emp.salary = 50000;

       FileOutputStream fOut=null;
       ObjectOutputStream oOut=null;

   try{
    fOut = new  FileOutputStream(
    "c:\\practicalJava\\BestEmployee.ser");

     oOut = new ObjectOutputStream(fOut);

     oOut.writeObject(emp); // serialization

     } catch (IOException e){….}
}}
```

To serialize an object do this:

1. Open an output stream.
2. Chain it with the `ObjectOutputStream`.
3. Call the method `writeObject()`, providing the instance of the `Serializable` object.
4. Close the stream.

ObjectInputStream

To deserialize an object perform the following steps:

1. Open an input stream.
2. Chain it with the `ObjectInputStream`.
3. Call the method `readObject()` and cast the returned object to the class that is being deserialized.
4. Close the stream.

```
class ClassB {

  public static void main(String args[]){

    FileInputStream fIn=null;
    ObjectInputStream oIn=null;

    try{
     fIn = new  FileInputStream("c:\\practicalJava\
\BestEmployee.ser");

     oIn = new ObjectInputStream(fIn);

     Employee bestEmp = (Employee) oIn.readObject();

   } catch (ClassNotFoundException cnf){
            cnf.printStackTrace();
   } catch (IOException e){…}}}
```

The class that de-serializes an object has to have access to its declaration, or the `ClassNotFoundException` will be thrown.

The `transient` keyword

If you do not want to serialize some sensitive data (e.g. salary), use the keyword `transient`.

```
transient double salary;
```

If you declare the property salary of a class with the `transient` qualifier, its value won't be serialized.

Serialization and Versioning

Declarations of serializable classes may change over time. The variable `serialVersionUID` helps to ensure that the declaration of the class in both JVM is the same.

```
private static final long serialVersionUID = 1L;
```

`InvalidClassException` the declaration indicates that the serialVersionUID are different in JVM that serializes and deserializes the object.

Walkthrough 2

1 Download and import into Eclipse the code of the Lesson 17 from the textbook Web site.

2 Review the code of the classes and run the ClassA program to serialize the object Employee into a file.

3 Open the file Employee.ser in any text editor and review its content.

4 Add the statement to ClassB to print the name and the salary of the deserialized Employee object. Run the program ClassB to deserialize Employee from the file Employee.ser.

5 Change the declaration of the class Employee to make salary transient:
transient double salary;

6 Run the program ClassA again and review the code of the Employee.ser. See the différence?

`Externalizable` Interface

The Employee class from last walkthrough had only 3 fields.

Now Imagine a `TradeOrder` class with 50 fields, but you need to serialize only 10 of them.

To minimize the number of bytes going over the network use `Externalizable` interface. It requires more coding, but allows selectively serialize data by implementing the methods `writeExternal()` and `readExternal()`.

Implementing Externalizable

```
class Employee2 implements java.io.Externalizable {
       String lName;
       String fName;
       String address;
       Date hireDate;
       int id;
       double salary;

   public void writeExternal(ObjectOutput stream)
                        throws java.io.IOException {
    // Serializing only the salary and id
    stream.writeDouble(salary);
    stream.writeInt(id);
   }

   public void readExternal(ObjectInput stream)
                        throws java.io.IOException {
     // Order or reads must be the
     // same as the order of writes
     salary = stream.readDouble();
     id  = stream.readInt();
   }}
```

Externalizing the object Employee

```
public class EmpProcessor {

  public static void main(String[] args) {
    Employee2 emp = new Employee2();
    emp.fName = "John";
    emp.lName = "Smith";
    emp.salary = 50000;
    emp.address = "12 main street";
    emp.hireDate = new Date();
    emp.id=123;

    FileOutputStream fOut=null;
    ObjectOutputStream oOut=null;
    try{
     fOut= new FileOutputStream("NewEmployee2.ser");
     oOut = new ObjectOutputStream(fOut);
     oOut.writeObject(emp); //serializing employee

     System.out.println("An employee is externalized into" +
                "NewEmployee2.ser");
    }catch(IOException e){
                    e.printStackTrace();
             }
     }
}
```

Network Programming with the class `java.net.URL`

Some Terminology

- Computers can communicate with each other if they agree on the rules of communication (a.k.a. ***protocols):*** TCP/IP, UDP/IP, FTP, HTTP et al.)
- ***Local area network (LAN)*** is a computer network connecting devices in a small area—the same office or house, or a rack.

- Interconnected computers located farther apart or that belong to different companies are part of a ***wide area network (WAN)***.

- World Wide Web (WWW) uses ***uniform resource locators (URLs)*** to identify online resources, for example http://www.mycompany.com:80/training.html

IP Address

The host name is automatically converted to the IP address of the physical device by your *Internet service provider (ISP)*, aka your hosting company.

The IP address can be a group of four numbers (e.g. 122.65.98.11) or 8-numbers (e.g. 2001:db8:0:1234:0:567:8:1)

Most people are connected to the Internet are getting *dynamic* IP addresses assigned to their home computers, but for a fee you can request a static IP address to be assigned to any computer located in your basement, office, or garage.

In enterprises, network computers have static (permanent) IP addresses.

Reading with `java.net.URL`

If you know the URL of the resource, and the Web server there is up and running, you can use URL class for reading the data located there.

By default, web servers are listening to all HTTP requests on port 80, and secure HTTPS requests are directed to port 443.

1 Create an instance of the class `URL`.

2 Create an instance of the `URLConnection` class and open a connection using the URL from Step 1.

3 Get a reference to an input stream of this object by calling the method `URLConnection.getInputStream()`.

4 Read the data from the stream.
Use a buffered reader to speed up the reading process.

Reading Google.com

```
public class WebSiteReader {

 public static void main(String args[]){

    String nextLine;
    URL url = null;
    URLConnection urlConn = null;

    InputStreamReader  inStream = null;
    BufferedReader buff = null;

    try{

     // index.html  or index.jsp or some other default URL at Google
     url  = new URL("http://www.google.com" );
     urlConn = url.openConnection();

     inStream = new InputStreamReader(urlConn.getInputStream(),
                                                        "UTF8");

     buff  = new BufferedReader(inStream);
```

```
// Read and print the lines from index.html
      while (true){
        nextLine =buff.readLine();

        if (nextLine !=null){
          System.out.println(nextLine);
        }else{
         break;
        }
      }

    } catch(MalformedURLException e){
      System.out.println("Please check the URL:" + e.toString() );
    } catch(IOException  e1){

      System.out.println("Can't read  from the Internet: "+
                                                  e1.toString() );
    } finally{
       if (inStream != null){
        try{
          inStream.close();
          buff.close();
        } catch(IOException e1){
          System.out.println("Can't close the streams: "+
                    e1.getMessage());
        }
       }
    }
 }
}
```

Walkthrough 3

- Download and import the source code from Lesson 18
- Review the code of the WebSiteReader and run the program
- Review the output on the system console

Downloading files from the Internet

If you can open a stream pointing at an unprotected file located on the remote computer, you can read it as easy as the local one.

```
class FileDownload{

 public static void main(String args[]){

  if (args.length!=2){

    System.out.println(
      "Proper Usage: java FileDownload URL OutputFileName");
    System.exit(0);

  }

 InputStream in=null;
 FileOutputStream fOut=null;

 try{

  URL remoteFile=new URL(args[0]);
 URLConnection fileStream=remoteFile.openConnection();

  // Open output and input streams
  fOut=new FileOutputStream(args[1]);
  in=fileStream.getInputStream();
```

```
  // Save the file

    int data;

    while((data=in.read())!=-1){
        fOut.write(data);
    }

  } catch (Exception e){
     e.printStackTrace();
  } finally{
            System.out.println("The file " + args[0] +
                    " has been downloaded successfully as " + args[1]);
     try{
       in.close();
       fOut.flush();
       fOut.close();

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

Walkthrough 3 (start)

- The goal is to download one of the Yakov's podcasts located at http://americhka.us.

- Review the code of the *FileDownload.java* and run the program. Explain the message on the system console
- Visit http://americhka.us and get a URL of any mp3 file there - use it as the first command line arg for *FileDownload.java*.
- Configure two program arguments for *FileDownload.java* (right-click | Run Configurations | (x)=Arguments), for example: http://TheFilenameYouFiguredOut.mp3 bestPodcast.mp3

Walkthrough 3 (End)

- Add the following statement above the line with try statement: System.out.println("Downloading…");

- Run the program *FileDownload* again. When you see the message that the file has been downloaded successfully, refresh your Eclipse project Lesson18 (Right-click | Refresh). You'll see the file bestPodcast.mp3 there.

- Double-click on this mp3 file and enjoy listening to the podcast (if you understand the language).

Homework

1. Modify the code from Lesson16 project to use try-with-resources syntax as shown in the video about Error Handling (Lesson 7). You'll need to replace JRE 1.6 with JRE 1.7 or later in Eclipse project Lesson16.

2. Do the assignments from the Try It sections of 17 (serialization)

3. Read the Networking Basics tutorial at http://bit.ly/1lh3aMk

4. For extra credit modify the FileDownload program so it can download several podcasts from americhka.us. After downloading as separate files works, see if you can get them as one zip file.

Additional Materials

Study the tutorial on working with files:
http://bit.ly/NKwb3X

Read Oracle's tutorial on Serialization: http://bit.ly/1hgs9IH

Read about the use of `serialVersionID`:
http://www.javablogging.com/what-is-serialversionuid