

03 – Streams & File I/O



JAVA: An Introduction to Problem Solving & Programming, 6th Ed. By Walter Savitch ISBN 0132162709 © 2012 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Objectives



Describe the concept of an I/O stream

- Explain the difference between text and binary files
- Save data, including objects, to a file
- Read data, including objects, from a file



Overview: Outline

Why Use Files for I/O?
Text Files and Binary Files
The Concept of a Stream

Why Use Files for I/O

Keyboard input, screen output deal with temporary data

- When program ends, data is gone
- Data in a file remains after program ends
 - Can be used next time program runs
 - Can be used by another program



Text Files and Binary Files

Files treated as sequence of characters called text files

- Java program source code
- Can be viewed, edited with text editor
- □All other files are called *binary files*
 - Movie, music files
 - Access requires specialized program

Background



- The Java platform includes a number of packages that are concerned with the movement of data into and out of programs. These packages differ in the kinds of abstractions they provide for dealing with I/O (input/output).
- The java.io package defines I/O in terms of *streams*. Streams are ordered sequences of data that have a *source* (input streams) or *destination* (output streams). The I/O classes isolate programmers from the specific details of the underlying operating system, while enabling access to system resources through files and other means.
- The best way to understand the I/O package is to start with the basic interfaces and abstract classes.

Background



- An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, remote data sources etc.
- Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.
- No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data.



The Concept of a Stream

Use of files in general

- Store Java classes, programs
- Store pictures, music, videos
- Can also use files to store program I/O
- A *stream* is a flow of input or output data
 - Characters
 - Numbers
 - Bytes





Streams are implemented as objects of special stream classes

- Class Scanner / PrintWriter
- Object System.out





The Concept of a Stream

A program uses an *input stream* to read data from a source, one item at time:



A program uses an *output stream* to write data to a destination, one item at time:





Class **PrintWriter** defines methods needed to create and write to a text file

- Must import package java.io
- □To open the file
 - Declare stream variable for referencing the stream
 - Invoke PrintWriter constructor, pass file name as argument
 - Requires try and catch blocks



Creating a Text File

□ File is empty initially

May now be written to with method println

Data goes initially to memory buffer

- When buffer full, goes to file
- Closing file empties buffer, disconnects from stream





class TextFileOutputDemo

public class TextFileOutputDemo

ł

}

```
public static void main(String[] args)
ť
   String fileName = "out.txt"; //The name could be read from
                                 //the keyboard.
   PrintWriter outputStream = null;
   try {
        outputStream = new PrintWriter(fileName);
    }
    catch(FileNotFoundException e){
        System.out.println("Error opening the file " + fileName);
        System.exit(0);
    }
                                                                            Writing out
                                                                          to the text file
    System.out.println("Enter three lines of text:");
    Scanner keyboard = new Scanner(System.in);
    for (int count = 1; count <= 3; count++) {</pre>
        String line = keyboard.nextLine( );
        outputStream.println(count + " " + line);
    outputStream.close( );
    System.out.println("Those lines were written to " +
```

fileName);



Creating a Text File

When creating a file

- Inform the user of ongoing I/O events, program should not be "silent"
- □ A file has two names in the program
 - File name used by the operating system
 - The stream name variable
- Opening, writing to file overwrites preexisting file in directory



Opening a file new begins with an empty file

- If already exists, will be overwritten
- Some situations require appending data to existing file
- Command could be
 - outputStream =
 - new PrintWriter(

new FileOutputstream(fileName, true));
Method println would append data at end



Reading from a Text File

□View sample program

- class TextFileInputDemo
- Reads text from file, displays on screen

Note

- Statement which opens the file
- Use of Scanner object (not PrintWriter object)
- Boolean statement which reads the file and terminates reading loop

Reading from a Text File







class TextFileInputDemo

```
public class TextFileInputDemo
4
    public static void main(String[] args)
9
         String fileName = "out.txt";
         Scanner inputStream = null;
         System.out.println("The file " + fileName + "\ncontains the following lines:\n");
       try {
            inputStream = new Scanner(new File(fileName));
        }
        catch(FileNotFoundException e) {
            System.out.println("Error opening the file " + fileName);
           System.exit(0);
                                                                             Reading in from
        }
                                                                                the text file
       while (inputStream.hasNextLine())
       k
            String line = inputStream.nextLine();
           System.out.println(line);
        inputStream.close();
     }
}
```



Working with Binary files

□ This Section will cover the following :

- Introduce a *FileDialog* object which allows the user to specify a *File* (via a GUI)
- Write *bytes* to a *File* and read them back from the *File* using *FileOutputStream* and *FileInputStream*
- Write values of primitive data types to/from a *File* using *DataOutputStream* and *DataInputStream*
- Write **Objects** to/from a *File* using *ObjectOutputStream* and *ObjectInputStream*
- Write exception-handling routines using the try-catch block



The Class File

Class provides a way to represent file names in a general way

A File object represents the name of a file
 The object (myFile) in the statement
 File myFile = new File ("sample.dat");
 is not simply a string

It is an object that *knows* it is supposed to name a file

The Class File



Suppose we want to read the contents of the file "sample.dat".

- 1st we must create a File object (from the java.io package)
- 2nd we must associate the File object with the file itself

This is achieved as follows :

File myFile = new File ("sample.dat");
Note : This assumes the file sample.dat is stored in
the current directory.



- Files opened in our examples assumed to be in same folder as where program run
- Possible to specify path names
 - Full path name
 - Relative path name
- Be aware of differences of pathname styles in different operating systems





- The argument to the constructor ("sample.dat"), specifies the file to access.
- To open a file that is stored in a directory other than the current directory you must also specify a path name.
 N.B.
 - File myFile = new File("C:\\docs", "sample.dat");
- As a rule, you should also check to see if a File object has correctly been associated with an existing file, by calling its *exists* method.
 - if(myFile.exists())

□ If a vaild association is established, we say *the file is opened*, and we can now proceed with I/O.



Some commonly used File methods.

Method	Description
boolean canRead()	Returns true if a file is readable; false otherwise.
<pre>boolean canWrite()</pre>	Returns true if a file is writable; false otherwise.
boolean exists()	Returns true if the name specified as the argument to the File
	constructor is a file or directory in the specified path; false
	otherwise.
boolean isFile()	Returns true if the name specified as the argument to the File
	constructor is a file; false otherwise.
boolean	Returns true if the name specified as the argument to the File
isDirectory()	constructor is a directory; false otherwise.
boolean	Returns true if the arguments specified to the File constructor
isAbsolute()	indicate an absolute path to a file or directory; false otherwise.
String	Returns a String with the absolute path of the file or directory.
getAbsolutePath()	
<pre>String getName()</pre>	Returns a String with the name of the file or directory.
String getPath()	Returns a String with the path of the file or directory.
<pre>String getParent()</pre>	Returns a String with the parent directory of the file or
	directory—that is, the directory in which the file or directory can be
	found.
long length()	Returns the length of the file in bytes. If the File object represents
	a directory, 0 is returned.
long	Returns a platform-dependent representation of the time at which the
lastModified()	file or directory was last modified. The value returned is only useful
	for comparison with other values returned by this method.
String[] list()	Returns an array of Strings representing the contents of a
	directory.



- We can let the user select a file or a directory, via a GUI, using a *FileDialog* object.
- □ The object has 2 modes :
 - LOAD : to read data from the specified file
 - SAVE : to write data to the specified/selected file
- The Following code is used to display the Dialog Box below (on the next slide) for <u>Opening a File</u>

```
FileDialog dialog;
Frame frame= new Frame("My Frame"); //required to hold the dialog
dialog = new FileDialog(frame,"Open File",FileDialog.LOAD);
dialog.setDirectory("C:\\env");
dialog.setFile("*.dat");
dialog.setVisible (true);
```





03 File Input & Output



To retrieve the name of the file the user has selected, we use the getFile() method.

```
String fileName = dialog.getFile();
```

- If the user has selected 'Cancel' the method returns a null string.
- Once we have a filename we can create a new File object.

```
File myFile = new File(fileName); // Current Directory
Of
```

File myFile = new File(dialog.getDirectory(), fileName);

To select a file for saving data, we open the *FileDialog* in SAVE mode:

```
dialog.setTitle("File Save As...");
dialog.setMode(FileDialog.SAVE);
dialog.setVisible (true);
```





03 File Input & Output

Low-Level File I/O (1) – *Byte Streams*



- Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from *InputStream* and *OutputStream*.
- Once a file is opened (associated properly with a File object), the actual file access can commence.
- In order to read/write from/to a file, we must create one of the Java stream objects and attach it to a file.
- There are many byte stream classes available to us. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, *FileInputStream* and *FileOutputStream*.
 - Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.
- To actually read the data from a file, we attach one of the Java Input Stream objects to the file (ditto for writing)

Low-Level File I/O (2)



- As already mentioned, the two objects that provide low-level (byte) file access are:
 - FileOutputStream
 - FileInputStream
- With these objects, we can only input/output a sequences of bytes i.e. values of data type byte (we will look at writing other data types and even objects later on)
- To <u>write</u> information to a file we can do the following:
 - First, create our File object
 - * File myFile = new File("sample.dat"); // or use FileDialog
 - Second, associate a new FileOutputStream object to a File
 - * FileOutputStream fos = new FileOutputStream(myFile);
 - Now we are ready for output

Low-Level File I/O (3)



Consider the following byte array: byte byteArray[] = $\{10, 20, 30, 40, 50, 60\};$ We can write out the whole array at once as follows: fos.write(byteArray); Or elements Individually: fos.write(byteArray[0]); Once the values have been written to the memory buffer, we must close the stream, to actually write the data to the file: fos.close(); If a stream is not closed, what are the implications?

Low-Level File I/O (4)



- To <u>read</u> data into a program, we reverse the steps in the output routine:
 - First, create our File object (if one doesn't already exist)

* File myFile = new File("sample.dat");

- Second, associate a new FileInputStream object to a File
 - * FileInputStream fis = new FileInputStream(myFile);
- Before we can read in the data, we must first declare and create a byteArray:

int filesize = (int) myFile.length()

byte byteArray[] = new byte[filesize];

- We use the *length()* method of the File class to determine the size of the file (the number of bytes in the file).
- We can then read in the data as follows:

```
fis.read(byteArray);
```

Mid-Level File I/O (1) – Data Streams



- Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values.
- All data streams implement either the *DataInput* interface or the *DataOutput* interface.
- □ This section focuses on the most widely-used implementations of these interfaces, *DataInputStream* and *DataOutputStream*.
- The Following code creates a *DataOutputStream* object:

File myFile = new File("Sample1.dat");

FileOutputStream fos = new FileOutputStream(myFile);

DataOutputStream dos = new DataOutputStream(fos);

N.B. - Since a DataOutputStream can only be created as a wrapper for an existing byte stream object, the argument to the DataOutputStream constructor is a FileOutputStream object. A DataOutputStream object does not get connected to a file directly – it's connected via a FileOutputStream object (ditto for reading...)

Mid-Level File I/O (2)



The DataOutputStream object has a number of methods for writing the primitive data types:

```
dos.writeInt(12345689);
```

dos.writeDouble(12345689.99);

```
dos.writeChar(`A');
```

dos.writeBoolean(true);

Don't forget to close the stream

```
dos.close();
```

Mid-Level File I/O (3)



```
To read the data back from the file, we reverse the operation:
    File myFile = new File("sample.dat");
    FileInputStream fis = new FileInputStream(myFile);
    DataInputStream dis = new DataInputStream(fis);
    ....
    int X = dis.readInt();
    Double D = dis.readDouble();
    char Y = dis.readChar();
    Boolean B = dis.readBoolean();
    .....
    dis.close();
    Note : the order of write % read eperations must match. When
```

Note : the order of write & read operations must match – Why?

File I/O : Exceptions (1)



- Exceptions are usually handled by *catching* a *thrown* exception and providing exception-handling code to process the thrown exception.
- There are two approaches to handling thrown exceptions:
 - Add a *throw* clause to the method header or
 - Use a *try/catch* block in your code
- □ For simple programs, the first approach may be acceptable, but in general you should use the second approach, in which you write a response code to handle a thrown exception.

File I/O : Exceptions (2)



□ Take the following example :

```
class FindSum
{
    private int result;
    private boolean success;

    public int GetSum()
        { return result; }

    public boolean isSuccess()
        { return success; }
```

File I/O : Exceptions (3)



```
void computeSum(String Filename)
    success = true;
       try {
             File myFile = new File(FileName);
             FileInputStream fis = new FileInputStream(myFile);
             DataInputStream dis = new DataInputStream(fis);
             int i = dis.readInt();
             int j = dis.readInt();
             int k = dis.readInt();
             result = i + j + k;
             dis.close();
           } // end of try block
     catch( IOException e )
                  { success = false; }
       } // end of method computeSum
} // end of class FindSum
```

File I/O : Exceptions (4)



// In a Program

File I/O: Exceptions (5)



- If an exception is thrown (i.e. an error occurs) during the execution of the *try* block, then the *catch* block is executed and the variable *success* is set to false.
- On a call to the method isSuccess(), a false value is returned and an error message is displayed.
- We can modify the catch block to output an error message itself, as well as setting the success variable:

```
try {...}
catch (IOException e)
  {
   success = false;
   JOptionPane.showMessageDialog(null,e.toString(),"Error
   Message",JOptionPane.ERROR_MESSAGE);
  }
}
```

File I/O: Exceptions (6)



- There are a number of useful exception classes available, when working with files :
 - You can throw a *FileNotFoundException* if you're trying to open a file that does not exist.
 - You can throw a *EOFException* if you're trying to read beyond the end of a file
- □ If you want to include statements that will catch any type of exception you can use the following:

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



High-Level File I/O - *Object Streams*

Binary-File I/O with Objects of a Class

Storing Array Objects in Binary Files

High-Level File I/O



Consider the need to write/read objects other than Strings

- Possible to write the individual instance variable values
- Then reconstruct the object when file is read
- A better way is provided by Java
 - Object serialization represent an object as a sequence of bytes to be written/read
 - Possible for any class implementing
 Serializable



High-Level File I/O – *Case Study*

- We will take a *Person* Object as an example to illustrate.
- □ We assume a Person Object consists of :
 - A Name (String)
 - An Age (int)
 - A Gender (char), ('M' or 'F')

High-Level File I/O – *Case Study*



□ Interface Serializable is an empty interface

- No need to implement additional methods
- Tells Java to make the class serializable (class objects convertible to sequence of bytes)

E.G - class Person

```
import java.io.*; // we need this for class Serializable
```

```
class Person implements Serializable
{
     // All the declarations / methods
}
```





Once we have a class that is specified as Serializable we can write objects to a binary file

Use method writeObject

Read objects with method readObject();

Also required to use typecast of the object

High-Level File I/O – *Case Study*



To write a Person object to a file, we first create an ObjectOutputStream object:

```
File myFile = new File("objects.dat");
```

```
FileOutputStream fos = new FileOutputStream(myFile);
```

```
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

To save a Person object, we write :
 Person p = new Person("Joe Bloggs", 25, 'M');
 oos.writeObject(p);

Note : You can save different types of objects to the same file using the writeObject() method.





To read a Person object from a file, we first create an ObjectInputStream object:

File myFile = new File("objects.dat");
FileInputStream fis = new FileInputStream(myFile);
ObjectInputStream ois = new ObjectInputStream(fis);

□ Then, to read a single Person object from the file, we write :

```
Person p = (Person) OIStream.readObject();
```

Note : If you save different types of objects to the same file using the *writeObject(*) method you must ensure that the objects are read back in the <u>correct order</u>, using <u>readObject(</u>) Array Objects in Binary Files



Since an array is an object itself, it is possible to use writeObject with an entire array

Similarly use readObject to read entire array

E.G. - **class PersonIODemo** (later slides)

High-Level File I/O – *Case Study*



Consider the following array of Person objects where N represents some integer value.

```
Person group[] = new Person[N];
```

Assuming that all N Person objects are in the Array, we can store them to a file a follows:

```
//Save the size of the array first
oos.writeInt(group.length);
```

We store the size of the array first, so we know how many Person objects to read back.





int N = OIStream.readInt();
for(int i = 0; i< N ; i++)
group[i] = (Person) ois.readObject();</pre>

However, since an array itself is an object, we can actually store the whole array at once :

```
oos.writeObject(Group);
```

□ And read the whole array back at once :

group = (Person[])ois.readObject();

□ Note the type casting of *an array* of Person objects

Case Study : PersonIODemo public class PersonIODemo { Output on 1st Run File myFile = new File("objects.dat"); ObjectOutputStream oos; ObjectInputStream ois; int size = 0; Person[] group = new Person[10]; public static void main(String[] args) ł

😑 Console 🖾

<terminated> PersonIODemo [Java Application] C:\env\appdev\jdk1.7.

File Empty/No File....Writing Data to Disk....

Э

}

new PersonIODemo();

Output on 2nd Run E Console 🔀 <terminated> PersonIODemo [Java Application] C:\env\appdev\jdk1.7.(File Not Empty....Reading Data From Disk.... Data of Size [2] Read in is.... Person [name=Jack, age=21, gender=M] Person [name=Gill, age=20, gender=F]

Case Study : PersonIODemo



Case Study : PersonIODemo



```
public void writeOut(Person[] p) throws FileNotFoundException, IOException
{
    oos = new ObjectOutputStream(new FileOutputStream(myFile));
    oos.writeInt(size);
    oos.writeObject(p);
    oos.close();
}
public Person[] readIn() throws FileNotFoundException, IOException, ClassNotFoundException
ł
    Person[] temp = null;
        ois = new ObjectInputStream(new FileInputStream(myFile));
        size = ois.readInt();
        temp = (Person[]) ois.readObject();
        ois.close();
    return temp;
}
```

Summary



□ Files with characters are text files

- Other files are binary files
- Programs can use **PrintWriter** and **Scanner** for I/O
- Always check for end of file
- File name can be literal string or variable of type
 String
- Class File gives additional capabilities to deal with file names

Summary



Use ObjectOutputStream and ObjectInputStream classes enable writing to, reading from binary files

- Use **writeObject** to write class objects to binary file
- Use **readObject** with type cast to read objects from binary file

Classes for binary I/O must be **Serializable**



Questions?

03 File Input & Output