CHAPTER 9
File I/O

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of Computer Science and Information Engineering
National Cheng Kung University

Streams

- A *stream* is an object that enables the flow of data between a program and some I/O device or file
  - If the data flows into a program, then the stream is called an *input stream*
  - If the data flows out of a program, then the stream is called an *output stream*

Streams

- Input streams can flow from the keyboard or from a file
  - **`System.in`** is an input stream that connects to the keyboard

```
Scanner keyboard = new Scanner(System.in);
```

- Output streams can flow to a screen or to a file
  - **`System.out`** is an output stream that connects to the screen

```
System.out.println("Output stream");
```

Writing to a Text File

- The class **`PrintWriter`** is a stream class that can be used to write to a text file
  - An object of the class **`PrintWriter`** has the methods **`print`** and **`println`**
  - These are similar to the **`System.out`** methods of the same names, but are used for text file output, not screen output

Writing to a Text File

- All the file I/O classes that follow are in the package **`java.io`**, so a program that uses **`PrintWriter`** will start with a set of **`import`** statements:

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
```

- A stream of the class **`PrintWriter`** is created and connected to a text file for writing as follows:

```
PrintWriter outputStreamName;
outputStreamName = new PrintWriter(new
    FileOutputStream(FileName));
```

Writing to a Text File

- This produces an object of the class **`PrintWriter`** that is connected to the file ***`FileName`***
  - The process of connecting a stream to a file is called *opening the file*
  - If the file already exists, then doing this causes the old contents to be lost
  - If the file does not exist, then a new, empty file named ***`FileName`*** is created

- After doing this, the methods **`print`** and **`println`** can be used to write to the file

Writing to a Text File

- When a text file is opened in this way, a **`FileNotFoundException`** can be thrown
  - In this context it actually means that the file could not be created
  - This type of exception can also be thrown when a program attempts to open a file for reading and there is no such file
- It is therefore necessary to enclose this code in exception handling blocks
  - The file should be opened inside a **`try`** block
  - A **`catch`** block should catch and handle the possible exception
  - The variable that refers to the **`PrintWriter`** object should be declared outside the block (and initialized to **`null`**) so that it is not local to the block

Writing to a Text File

- When a program is finished writing to a file, it should always close the stream connected to that file

```
outputStreamName.close();
```

  - This allows the system to release any resources used to connect the stream to the file
  - If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly

Writing to a Text File

- Output streams connected to files are usually *buffered*
  - Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (*buffer*)
  - When enough data accumulates, or when the method `flush` is invoked, the buffered data is written to the file all at once
  - This is more efficient, since physical writes to a file can be slow

Writing to a Text File

- The method `close` invokes the method `flush`, thus insuring that all the data is written to the file
  - If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file
  - Also, if a program writes to a file and later reopens it to read from the same file, it will have to be closed first anyway
  - The sooner a file is closed after writing to it, the less likely it is that there will be a problem

Lab

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class FileTest {

  public static void main(String[] args) {

    try {
      PrintWriter writer = new PrintWriter(new FileOutputStream("d:\\output.txt"));

      writer.println("Dear Alan,");
      writer.println("How are you?");
      writer.println("Joe");

      writer.flush();
      writer.close();

    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }

  }
}
```

Pitfall: a `try` Block is a Block

- Since opening a file can result in an exception, it should be placed inside a **`try`** block
- If the variable for a **`PrintWriter`** object needs to be used outside that block, then the variable must be declared outside the block
  - Otherwise it would be local to the block, and could not be used elsewhere
  - If it were declared in the block and referenced elsewhere, the compiler will generate a message indicating that it is an undefined identifier

Lab

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class FileTest {

  public static void main(String[] args) {

    try {
      PrintWriter writer = new PrintWriter(new FileOutputStream("d:\\output.txt"));

      writer.println("Dear Alan,");
      writer.println("How are you?");
      writer.println("Joe");

      writer.flush();
      writer.close();

    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }

      writer.println(“Some other words here.");

  }
}
```

Compile error!

Appending to a Text File

- To create a **`PrintWriter`** object and connect it to a text file for *appending*, a second argument, set to **`true`**, must be used in the constructor for the **`FileOutputStream`** object

```
outputStreamName = new PrintWriter(new
  FileOutputStream(FileName, true));
```

  - After this statement, the methods **`print`**, **`println`** and/or **`printf`** can be used to write to the file
  - The new text will be written *after the old text* in the file

Lab

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintWriter;

public class FileTest {

  public static void main(String[] args) {

    try {
      PrintWriter writer = new PrintWriter(new FileOutputStream("d:\\output.txt", true));

      writer.println("Dear Alan,");
      writer.println("How are you?");
      writer.println("Joe");

      writer.flush();
      writer.close();

    } catch (FileNotFoundException e) {
      e.printStackTrace();
    }

  }
}
```

Run this program 3 times…

Reading From a Text File Using Scanner

- The class **Scanner** can be used for reading from the keyboard as well as reading from a text file
  - Simply replace the argument **System.in** (to the **Scanner** constructor) with a suitable stream that is connected to the text file

```
Scanner StreamObject =
    new Scanner(new FileInputStream(FileName));
```

- Methods of the **Scanner** class for reading input behave the same whether reading from the keyboard or reading from a text file
  - For example, the **nextInt** and **nextLine** methods

Testing for the End of a Text File with `Scanner`

- A program that tries to read beyond the end of a file using methods of the **`Scanner`** class will cause an exception to be thrown
- However, instead of having to rely on an exception to signal the end of a file, the **`Scanner`** class provides methods such as **`hasNextInt`** and **`hasNextLine`**
  - These methods can also be used to check that the next token to be input is a suitable element of the appropriate type

Lab

```
import java.io.FileInputStream;
import java.util.Scanner;
public class ReadFileTest {

  public static void main(String[] args) {
    try{
      Scanner scanner = new Scanner(new FileInputStream("d:\\test.txt"));
      while(scanner.hasNextLine()){
        String aline = scanner.nextLine();
        System.out.println(aline);
      }

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

Writing Data to a Binary File

- Binary files store data in the same format used by computer memory to store the values of variables
  - No conversion needs to be performed when a value is stored or retrieved from a binary file
- Java binary files, unlike other binary language files, are portable
  - A binary file created by a Java program can be moved from one computer to another
  - These files can then be read by a Java program, but only by a Java program

Writing Data to a Binary File

- The class **`ObjectOutputStream`** is a stream class that can be used to write to a binary file
  - An object of this class has methods to write strings, values of primitive types, and **objects to a binary file**
- A program using **`ObjectOutputStream`** needs to import several classes from package **`java.io`**:

```
import java.io.ObjectOutputStream;
import java.io.FileOutStream;
import java.io.IOException;
```

Opening a Binary File for Output

- An **`ObjectOutputStream`** object is created and connected to a binary file as follows:

```
ObjectOutputStream oos = new ObjectOutputStream(new
                         FileOutputStream(FileName));
```

Some Methods in the Class `ObjectOutputStream` (Part 1 of 5)

Display 10.14 Some Methods in the Class `ObjectOutputStream`

`ObjectOutputStream` and `FileOutputStream` are in the `java.io` package.

```
public ObjectOutputStream(OutputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

```
new ObjectOutputStream(new FileOutputStream(File_Name))
```

This creates a blank file. If there already is a file named *File_Name*, then the old contents of the file are lost.
If you want to create a stream using an object of the class `File`, you use

```
new ObjectOutputStream(new FileOutputStream(File_Object))
```

The constructor for `FileOutputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileOutputStream` constructor succeeds, then the constructor for `ObjectOutputStream` may throw a different `IOException`.

(continued)

Some Methods in the Class `ObjectOutputStream` (Part 2 of 5)

Display 10.14 Some Methods in the Class `ObjectOutputStream`

```
public void writeInt(int n) throws IOException
```

Writes the `int` value `n` to the output stream.

```
public void writeShort(short n) throws IOException
```

Writes the `short` value `n` to the output stream.

```
public void writeLong(long n) throws IOException
```

Writes the `long` value `n` to the output stream.

```
public void writeDouble(double x) throws IOException
```

Writes the `double` value `x` to the output stream.

(continued)

Some Methods in the Class `ObjectOutputStream` (Part 3 of 5)

Display 10.14 Some Methods in the Class `ObjectOutputStream`

```
public void writeFloat(float x) throws IOException
```

Writes the `float` value x to the output stream.

```
public void writeChar(int n) throws IOException
```

Writes the `char` value n to the output stream. Note that it expects its argument to be an `int` value. However, if you simply use the `char` value, then Java will automatically type cast it to an `int` value. The following are equivalent:

```
outputStream.writeChar((int)'A');
```

and

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

(continued)

Some Methods in the Class `ObjectOutputStream` (Part 4 of 5)

Display 10.14 Some Methods in the Class `ObjectOutputStream`

```
public void writeBoolean(boolean b) throws IOException
```

Writes the `boolean` value `b` to the output stream.

```
public void writeUTF(String aString) throws IOException
```

Writes the `String` value `aString` to the output stream. UTF refers to a particular method of encoding the string. To read the string back from the file, you should use the method `readUTF` of the class `ObjectInputStream`.

(continued)

Some Methods in the Class `ObjectOutputStream` (Part 5 of 5)

Display 10.14 Some Methods in the Class `ObjectOutputStream`

```
public void writeObject(Object anObject) throws IOException
```

Writes its argument to the output stream. The object argument should be an object of a serializable class, a concept discussed later in this chapter. Throws various `IOExceptions`.

```
public void close() throws IOException
```

Closes the stream's connection to a file. This method calls `flush` before closing the file.

```
public void flush() throws IOException
```

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

Lab

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class ObjectOutputStreamTest {

  public static void main(String[] args) {
    try {
      ObjectOutputStream oos = new ObjectOutputStream(new
                                    FileOutputStream("d:\\output2.obj"));
      oos.writeInt(123);
      oos.flush();
      oos.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

}
```

The `Serializable` Interface

- In order to make a class serializable, simply add **`implements Serializable`** to the heading of the class definition

```
public class SomeClass implements Serializable
```

- When a serializable class has instance variables of a class type, then all those classes must be serializable also
  - A class is not serializable unless the classes for all instance variables are also serializable for all levels of instance variables within classes

Lab

```
import java.io.Serializable;

public class Student implements Serializable{
  private String name;
  private int age;

  public Student(String name, int age){
    this.name = name;
    this.age = age;
  }
  public String getName(){
    return name;
  }
  public int getAge(){
    return age;
  }
}
```

Lab

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class ObjectOutputStreamTest {

  public static void main(String[] args) {

    Student stu = new Student("Hacker", 20);

    try {
      ObjectOutputStream oos = new ObjectOutputStream(new
                                          FileOutputStream("d:\\output2.obj"));
      oos.writeInt(123);
      oos.writeObject(stu);
      oos.flush();
      oos.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Reading Data from a Binary File

- The class **`ObjectInputStream`** is a stream class that can be used to read from a binary file
  - An object of this class has methods to read strings, values of primitive types, and objects from a binary file
- A program using **`ObjectInputStream`** needs to import several classes from package **`java.io`**:

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

Opening a Binary File for Reading

- An **`ObjectInputStream`** object is created and connected to a binary file as follows:

```
ObjectInputStream ois = new ObjectInputStream(new
                                    FileInputStream(FileName));
```

Opening a Binary File for Reading

- After opening the file, **`ObjectInputStream`** methods can be used to read to the file
  - Methods used to input primitive values include **`readInt`**, **`readDouble`**, **`readChar`**, and **`readBoolean`**
  - The method **`readUTF`** is used to input values of type **`String`**
- If the file contains multiple types, each item type must be read **in exactly the same order** it was written to the file

Some Methods in the Class `ObjectInputStream` (Part 1 of 5)

Display 10.15 Some Methods in the Class `ObjectInputStream`

The classes `ObjectInputStream` and `FileInputStream` are in the `java.io` package.

```
public ObjectInputStream(InputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

```
new ObjectInputStream(new FileInputStream(File_Name))
```

Alternatively, you can use an object of the class `File` in place of the *File_Name*, as follows:

```
new ObjectInputStream(new FileInputStream(File_Object))
```

The constructor for `FileInputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileInputStream` constructor succeeds, then the constructor for `ObjectInputStream` may throw a different `IOException`.

(continued)

Some Methods in the Class `ObjectInputStream` (Part 2 of 5)

Display 10.15 Some Methods in the Class `ObjectInputStream`

```
public int readInt() throws IOException
```

Reads an `int` value from the input stream and returns that `int` value. If `readInt` tries to read a value from the file and that value was not written using the method `writeInt` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public int readShort() throws IOException
```

Reads a `short` value from the input stream and returns that `short` value. If `readShort` tries to read a value from the file and that value was not written using the method `writeShort` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

(continued)

Some Methods in the Class `ObjectInputStream` (Part 3 of 5)

Display 10.15 Some Methods in the Class `ObjectInputStream`

```
public long readLong() throws IOException
```

Reads a `long` value from the input stream and returns that `long` value. If `readLong` tries to read a value from the file and that value was not written using the method `writeLong` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public double readDouble() throws IOException
```

Reads a `double` value from the input stream and returns that `double` value. If `readDouble` tries to read a value from the file and that value was not written using the method `writeDouble` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public float readFloat() throws IOException
```

Reads a `float` value from the input stream and returns that `float` value. If `readFloat` tries to read a value from the file and that value was not written using the method `writeFloat` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

(continued)

Some Methods in the Class `ObjectInputStream` (Part 4 of 5)

Display 10.15 Some Methods in the Class `ObjectInputStream`

```
public char readChar() throws IOException
```

Reads a `char` value from the input stream and returns that `char` value. If `readChar` tries to read a value from the file and that value was not written using the method `writeChar` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public boolean readBoolean() throws IOException
```

Reads a `boolean` value from the input stream and returns that `boolean` value. If `readBoolean` tries to read a value from the file and that value was not written using the method `writeBoolean` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

(continued)

Some Methods in the Class `ObjectInputStream` (Part 5 of 5)

Display 10.15 Some Methods in the Class `ObjectInputStream`

```
public String readUTF() throws IOException
```

Reads a `String` value from the input stream and returns that `String` value. If `readUTF` tries to read a value from the file and that value was not written using the method `writeUTF` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
Object readObject( ) throws ClassNotFoundException, IOException
```

Reads an object from the input stream. The object read should have been written using `writeObject` of the class `ObjectOutputStream`. Throws a `ClassNotFoundException` if a serialized object cannot be found. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown. May throw various other `IOExceptions`.

```
public int skipBytes(int n) throws IOException
```

Skips n bytes.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

Binary I/O of Objects

- Objects can also be input and output from a binary file
  - Use the **`writeObject`** method of the class **`ObjectOutputStream`** to write an object to a binary file
  - Use the **`readObject`** method of the class **`ObjectInputStream`** to read an object from a binary file
  - In order to use the value returned by **`readObject`** as an object of a class, it must be type cast first:

```
SomeClass someObject =
   (SomeClass)objectInputStream.readObject();
```

Lab

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class ObjectInputStreamTest {

  public static void main(String[] args) {
    try {
      ObjectInputStream ois = new ObjectInputStream(new
                                          FileInputStream("d:\\output2.obj"));
      int num = ois.readInt();
      System.out.println("num=" + num);

      Student stu = (Student)ois.readObject();
      System.out.println("Student name="+stu.getName());
      System.out.println("Student age="+stu.getAge());

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

Array Objects in Binary Files

- Since an array is an object, arrays can also be read and written to binary files using **`readObject`** and **`writeObject`**
  - If the base type is a class, then it must also be serializable, just like any other class type
  - Since **`readObject`** returns its value as type **`Object`** (like any other object), it must be type cast to the correct array type:

```
SomeClass[] someObject =
 (SomeClass[])objectInputStream.readObject();
```

Lab

```
public class ObjectOutputStreamTest {

public static void main(String[] args) {
  Student[] stu = new Student[2];
  stu[0] = new Student("Hacker1", 21);
  stu[1] = new Student("Hacker2", 22);

  try {
    ObjectOutputStream oos = new ObjectOutputStream(new
                                        FileOutputStream("d:\\output2.obj"));
    oos.writeInt(123);
    oos.writeObject(stu);
    oos.flush();
    oos.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}

}
```

Lab

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class ObjectInputStreamTest {

  public static void main(String[] args) {
    try {
      ObjectInputStream ois = new ObjectInputStream(new
                                              FileInputStream("d:\\output2.obj"));
      int num = ois.readInt();
      System.out.println("num=" + num);

      Student[] stu = (Student[])ois.readObject();
      System.out.println("Student1 name="+stu[0].getName());
      System.out.println("Student1 age="+stu[0].getAge());
      System.out.println("Student2 name="+stu[1].getName());
      System.out.println("Student2 age="+stu[1].getAge());

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

try-catch Mechanism

- The basic way of handling exceptions in Java consists of the ***try-throw-catch*** trio
- The **try** block contains the code for the basic algorithm
  - It tells what to do when everything goes smoothly
- It is called a **try** block because it "tries" to execute the case where all goes as planned
  - It can also contain code that throws an exception if something unusual happens

```
try
{
   CodeThatMayThrowAnException
}
```

try-catch Mechanism

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

- When an exception is thrown, the **catch** block begins execution
  - The **catch** block has one parameter
  - The exception object thrown is plugged in for the **catch** block parameter
- The execution of the **catch** block is called *catching the exception*, or *handling the exception*
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by some **catch** block

The `finally` Block

- The **`finally`** block contains code to be executed whether or not an exception is thrown in a **`try`** block
  - If it is used, a **`finally`** block is placed after a **`try`** block and its following **`catch`** blocks

```
try
{  . . .  }
catch(ExceptionClass1 e)
{  . . .  }
  . . .
catch(ExceptionClassN e)
{  . . .  }
finally
{
  CodeToBeExecutedInAllCases
}
```

Lab

```
import java.io.FileInputStream;
import java.util.Scanner;
public class ReadFileTest {

  public static void main(String[] args) {
    try{
      Scanner scanner = new Scanner(new FileInputStream("d:\\test.txt"));
      while(scanner.hasNextLine()){
        String aline = scanner.nextLine();
        System.out.println(aline);
      }

    }catch(Exception e){
      e.printStackTrace();
    }finally{
      System.out.println("Print this message whether the file could be read or not.");
    }
  }
}
```

Lab

An _____ allows data to flow into your program.

(a)input stream

(b)output stream

(c)file name

(d)all of the above

Lab

An _____ allows data to flow from your program.

(a)input stream

(b)output stream

(c)file name

(d)all of the above

Lab

The output stream connected to the computer screen is:

(a)System.exit

(b)System.quit

(c)System.in

(d)System.out

Lab

The scanner class has a series of methods that checks to see if there is any more well-formed input of the appropriate type. These methods are called _____ methods:

(a)nextToken

(b)hasNext

(c)getNext

(d)testNext

Lab

The method _____ from the File class forces a physical write to the file of any data that is buffered.

(a)close()
(b)flush()
(c)writeUTF()
(d)writeObject()

Reference

- “Absolute Java”. Walter Savitch and Kenrick Mock. Addison-Wesley; 5 edition. 2012
- “Java How to Program”. Paul Deitel and Harvey Deitel. Prentice Hall; 9 edition. 2011.
- “A Programmers Guide To Java SCJP Certification: A Comprehensive Primer 3rd Edition”. Khalid Mughal, Rolf Rasmussen. Addison-Wesley Professional. 2008