Chapter 9

Streams and File I/O

- Overview of Streams and File I/O
- Text File I/O
- Binary File I/O
- File Objects and File Names

I/O Overview

- *I/O* = Input/Output
- In this context it is input to and output from programs
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
  - » permanent copy
  - » output from one program can be input to another
  - » input can be automated (rather than entered manually)

Note: Since the sections on text file I/O and binary file I/O have some similar information, and can be covered with either one first, some duplicate (or nearly duplicate) slides are included.

Streams

- *Stream*: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - » it acts as a buffer between the data source and destination
- *Input stream*: a stream that provides input to a program
- *Output stream*: a stream that accepts output from a program
  - » `System.out` is an output stream
  - » `SavitchIn` is an input stream
- A stream connects a program to an I/O object
  - » `System.out` connects a program to the screen
  - » `SavitchIn` connects a program to the keyboard

Binary Versus Text Files

- *All* data and programs are ultimately just zeros and ones
  - » each digit can have one of two values, hence *binary*
  - » *bit* is one binary digit
  - » *byte* is a group of eight bits
- *Text files*: the bits represent printable characters
  - » one byte per character for ASCII, the most common code
  - » for example, Java source files are text files
  - » so is any file created with a "text editor"
- *Binary files*: the bits represent other types of encoded information, such as executable instructions or numeric data
  - » these files are easily read by the computer but not humans
  - » they are *not* "printable" files
    - – actually, you *can* print them, but they will be unintelligible
    - – "printable" means "easily readable by humans when printed"

Java: Text Versus Binary Files

- Text files are more readable by humans
- Binary files are more efficient
  - » computers read and write binary files more easily than text
- Java binary files are portable
  - » they can be used by Java on different machines
  - » Reading and writing binary files is normally done by a program
  - » text files are used only to communicate with humans

Java Text Files

- Source files
- Occasionally input files
- Occasionally output files

Java Binary Files

- Executable files (created by compiling source files)
- Usually input files
- Usually output files

Text File I/O

- Important classes for text file **output** (to the file)
  - » **`PrintWriter`**
  - » **`FileOutputStream`**
- Important classes for text file **input** (from the file):
  - » **`BufferedReader`**
  - » **`FileReader`**
- Note that **`FileOutputStream`** and **`FileReader`** are used only for their constructors, which can take file names as arguments.
  - » **`PrintWriter`** and **`BufferedReader`** cannot take file names as arguments for their constructors.
- To use these classes your program needs a line like the following:

```
import java.io.*;
```

Every File Has Two Names

- The code to open the file creates two names for an output file
  - » the name used by the operating system
    - – `out.txt` in the example
  - » the stream name
    - – `outputStream` in the example
- Java programs use the *stream* name
  - » `outputStream` in the example

Text File Output

- Binary files are more efficient for Java, but text files are readable by humans
  - » so occasionally text rather than binary files are used
- Java allows both binary and text file I/O
- To open a text file for output: connect a text file to a stream for writing
  - » create a stream of the class `PrintWriter` and connect it to a text file
- For example:

```
PrintWriter outputStream =
  new PrintWriter(new FileOutputStream("out.txt");
```

- Then you can use `print` and `println` to write to the file
  - » The text lists some other useful `PrintWriter` methods

TextFileOutputDemo

Part 1

```
public static void main(String[] args)
{
    PrintWriter outputStream = null;
    try
    {
        outputStream =
             new PrintWriter(new FileOutputStream("out.txt"));
    }
    catch(FileNotFoundException e)
    {
        System.out.println("Error openin
        System.exit(0);
    }
```

`outputStream` would not be accessible to the rest of the program if it were declared inside the `try`-block

Opening the file

Creating a file can cause the `FileNotFound-Exception` if the new file cannot be made.

TextFileOutputDemo

Part 2

```
System.out.println("Enter three lines of text:");
String line = null;
int count;
   for (count = 1; count <= 3; count++)
   {
       line = SavitchIn.readLine();
       outputStream.println(count + " " + line);
   }
   outputStream.close();
   System.out.println("... written to out.txt.");
}
```

Writing to the file

Closing the file

The `println` method is used with two different streams: `outputStream` and `System.out`

Closing a File

- An output file should be closed when you are done writing to it (and an input file should be closed when you are done reading from it).
- Use the `close` method of the class `PrintWriter` (`BufferedReader` also has a `close` method).
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open

If It Is Done Automatically, Why Explicitly Close Files?

If a program automatically closes files when it ends normally, why close them with explicit calls to `close`?

Two reasons:

1. To make sure it is closed if a program ends abnormally (it could get damaged if it is left open).
2. A file open for writing must be closed before it can be opened for reading.
   - Although Java does have a class that opens a file for both reading and writing, it is not used in this text

Text File Input

- To open a text file for input: connect a text file to a stream for reading
  - » use a stream of the class `BufferedReader` and connect it to a text file
  - » use the `FileReader` class to connect the `BufferedReader` object to the text file
- For example:

```
BufferedReader inputStream =
    new BufferedReader(new FileReader("data.txt"));
```

- Then:
  - » read lines (`Strings`) with `readLine`
  - » `BufferedReader` has no methods to read numbers directly, so read numbers as `Strings` and then convert them
  - » read a `char` with `read`

Reading Words in a String: Using **`StringTokenizer`** Class

- There are `BufferedReader` methods to read a line and a character, but not just a single word
- `StringTokenizer` can be used to parse a line into words
  - » it is in the `util` library so you need to import `java.util.*`
  - » some of its useful methods are shown in the text
    - – e.g. test if there are more tokens
  - » you can specify *delimiters* (the character or characters that separate words)
    - – the default delimiters are "white space" (space, tab, and newline)

Example: **`StringTokenizer`**

- Display the words separated by any of the following characters: space, new line (\n), period (.) or comma (,).

```
String inputLine = SavitchIn.readLine();
StringTokenizer wordFinder =
new StringTokenizer(inputLine, " \n.,");
//the second argument is a string of the 4 delimiters
while(wordFinder.hasMoreTokens())
{
   System.out.println(wordFinder.nextToken());
}
```

Entering "`Question,2b.or !tooBee.`" gives this output:

```
Question
2b
or
!tooBee
```


Testing for End of File in a Text File

- There are several ways to test for end of file. For reading text files in Java you can use this one:

  Test for a special character that signals the end of the file

- When `readLine` tries to read beyond the end of a text file it returns the special value *`null`*
  - » so you can test for `null` to stop processing a text file

- `read` returns -1 when it tries to read beyond the end of a text file
  - » the `int` value of all ordinary characters is nonnegative

- Neither of these two methods (`read` and `readLine`) will throw an `EOFException`.

Example: Using Null to Test for End-of-File in a Text File

When using **`readLine`** test for `null`

Excerpt from `TextEOFDemo`

```
int count = 0;
String line = inputStream.readLine();
while (line != null)
{
  count++;
  outputStream.println(count + " " + line);
  line = inputStream.readLine();
}
```

When using **`read`** test for -1

Binary File I/O

- Important classes for binary file **output** (to the file)
  - » **`DataOutputStream`**
  - » **`FileOutputStream`**
- Important classes for binary file **input** (from the file):
  - » **`DataInputStream`**
  - » **`FileInputStream`**
- Note that **`FileOutputStream`** and **`FileInputStream`** are used only for their constructors, which can take file names as arguments.
  - » **`DataOutputStream`** and **`DataInputStream`** cannot take file names as arguments for their constructors.
- To use these classes your program needs a line like the following:

```
import java.io.*;
```

Java File I/O: Stream Classes

- `DataInputStream` and `DataOutputStream`:
  - » have methods to either read or write data one byte at a time
  - » automatically convert numbers and characters into binary
    - – binary-encoded numeric files (files with numbers) are not readable by a text editor, but store data more efficiently
- Remember:
  - » *input* means data into a program, not the file
  - » similarly, *output* means data out of a program, not the file

When Using **`DataOutputStream`** to Output Data to Files:

- The output files are binary and can store any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type
- The files created can be read by other Java programs but are not printable
- The Java I/O library must be imported by including the line:
  `import java.io.*;`
  - » it contains `DataOutputStream` and other useful class definitions
- An `IOException` might be thrown

Handling **`IOException`**

- `IOException` cannot be ignored
  - » either handle it with a catch block
  - » or defer it with a `throws`-clause

We will put code to open the file and write to it in a `try`-block and write a `catch`-block for this exception :

```
catch(IOException e)
{
  System.out.println("Problem with output...";
}
```


Opening a New Output File

- The file name is given as a `String`
  - » file name rules are determined by your operating system

- Opening an output file takes two steps
  1. Create a `FileOutputStream` object associated with the file name `String`
  2. Connect the `FileOutputStream` to a `DataOutputStream` object

  This can be done in one line of code

Example: Opening an Output File

To open a file named `numbers.dat`:

```
DataOutputStream outputStream =
  new DataOutputStream(new
  FileOutputStream("numbers.dat"));
```

- The constructor for `DataOutputStream` requires a `FileOutputStream` argument
- The constructor for `FileOutputStream` requires a `String` argument
  - » the `String` argument is the output file name
- The following two statements are equivalent to the single statement above:

```
FileOutputStream middleman =
  new FileOutputStream("numbers.dat");
DataOutputStream outputStream =
  new DataOutputSteam(middleman);
```


Every File Has Two Names

- The code to open the file creates two names for an output file
  - » the name used by the operating system
    - – `numbers.dat` in the example
  - » the stream name
    - – `outputStream` in the example
- Java programs use the *stream* name
  - » `outputStream` in the example

Some `DataOutputStream` Methods

- You can write data to an output file after it is connected to a stream class
  - » Use methods defined in `DataOutputStream`
    - – `writeInt(int n)`
    - – `writeDouble(double x)`
    - – `writeBoolean(boolean b)`
    - – etc.
    - – See the text for more
- Note that each write method throws `IOException`
  - » eventually we will have to write a catch block for it
- Also note that each write method includes the modifier `final`
  - » `final` methods cannot be redefined in derived classes

Closing a File

- An Output file should be closed when you are done writing to it
- Use the `close` method of the class `DataOutputStream`
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open

If It Is Done Automatically, Why Explicitly Close Files?

If a program automatically closes files when it ends normally, why close them with explicit calls to `close`?

Two reasons:

1. To make sure it is closed if a program ends abnormally (it could get damaged if it is left open).

2. A file open for writing must be closed before it can be opened for reading.
   - Although Java does have a class that opens a file for both reading and writing, it is not used in this text

Writing a Character to a File: an Unexpected Little Complexity

- The method `writeChar` has an annoying property:
  - » it takes an `int`, not a `char`, argument

- But it is easy to fix:
  - » just cast the character to an int

- For example, to write the character 'A' to the file opened previously:

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

Writing a **`boolean`** Value to a File

- `boolean` values can be either of two values, `true` or `false`
- `true` and `false` are not just names for the values, they actually are of type `boolean`
- For example, to write the `boolean` value `false` to the output file:

```
outputStream.writeBoolean(false);
```


Writing Strings to a File: Another Little Unexpected Complexity

- Use the `writeUTF` method to output a value of type `String`
  - » there is no `writeString` method
- UTF stands for Unicode Text Format
  - » a special version of Unicode
- Unicode: a text (printable) code that uses 2 bytes per character
  - » designed to accommodate languages with a different alphabet or no alphabet (such as Chinese and Japanese)
- ASCII: also a text (printable) code, but it uses just 1 byte per character
  - » the most common code for English and languages with a similar alphabet
- UTF is a modification of Unicode that uses just one byte for ASCII characters
  - » allows other languages without sacrificing efficiency for ASCII files

Warning: Overwriting a File

- Opening a file creates an empty file
- Opening a file creates a new file if it does not already exist
- Opening a file that already exists eliminates the old file and creates a new, empty one
  - » data in the original file is lost
- How to test for the existence of a file and avoid overwriting it will be covered later (it is in Section 9.3 of the text, which discusses the `File` class)

When Using `DataInputStream` to Read Data from Files:

- Input files are binary and contain any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type
- The files can be read by Java programs but are not printable
- The Java I/O library must be imported including the line:
  `import java.io.*;`
  - » it contains `DataInputStream` and other useful class definitions
- An `IOException` might be thrown

Opening a New Input File

- Similar to opening an output file, but replace "output" with "input"
- The file name is given as a `String`
  - » file name rules are determined by your operating system
- Opening a file takes two steps
  1. Creating a `FileInputStream` object associated with the file name `String`
  2. Connecting the `FileInputStream` to a `DataInputStream` object
- This can be done in one line of code

Example: Opening an Input File

To open a file named `numbers.dat`:

```
DataInputStream inStream =
  new DataInputStream(new
  FileInputStream("numbers.dat"));
```

- The constructor for `DataInputStream` requires a `FileInputStream` argument
- The constructor for `FileInputStream` requires a `String` argument
  - » the `String` argument is the input file name
- The following two statements are equivalent:

```
FileInputStream middleman =
  new FileInputStream("numbers.dat");
DataInputStream inputStream =
  new DataInputSteam(middleman);
```

Some **`DataInputStream`** Methods

- For every output file method there is a corresponding input file method
- You can read data from an input file after it is connected to a stream class
  - » Use methods defined in `DataInputStream`
    - `readInt()`
    - `readDouble()`
    - `readBoolean()`
    - etc.
    - See the text for more
- Note that each write method throws `IOException`
- Also note that each write method includes the modifier `final`

Input File Exceptions

- A `FileNotFoundException` is thrown if the file is not found when an attempt is made to open a file
- Each read method throws `IOException`
  - » we still have to write a catch block for it
- If a read goes beyond the end of the file an `EOFException` is thrown

Avoiding Common **`DataInputStream`** File Errors

There is no error message (or exception) if you read the wrong data type!

- Input files can contain a mix of data types
  - » it is up to the programmer to know their order and use the correct read method
- `DataInputStream` works with binary, not text files
- As with an output file, close the input file when you are done with it

Example: Reading a File Name from the Keyboard

reading a file name from the keyboard

using the file name read from the keyboard

reading data from the file

closing the file

`FileNameDemo`

```
public static void main(String[] args)
  {
    String fileName = null;
    try
    {
      System.out.println("Enter file name:");
      fileName = SavitchIn.readLineWord();
      DataInputStream inputStream =
        new DataInputStream(new FileInputStream(fileName));
      int n;
      System.out.println("Reading the nonnegative integers");
      System.out.println("in the file " + fileName);
      n = inputStream.readInt();
      while (n >= 0)
      {
        System.out.println(n);
        n = inputStream.readInt();
      }
      System.out.println("End of reading from file.");
      inputStream.close();
    }
    catch(IOException e)
    {
      System.out.println("Problem with output to file " + fileName);
    }
  }
```

Exception Handling with File I/O

Catching IOExceptions

- `IOException` is a predefined class
- File I/O done as described here might throw an `IOException`
- You should catch the exception in a catch block that at least prints an error message and ends the program
- `FileNotFoundException` is derived from `IOException`
  - » therefor any catch block that catches `IOExceptions` also catches `FileNotFoundExceptions`
  - » errors can be isolated better if they have different messages
  - » so create different catch blocks for each exception type
  - » put the more specific one first (the derived one) so it catches specifically file-not-found exceptions
  - » then you will know that an I/O error is something other than file-not-found

Common Methods to Test for the End of an Input File

- A common programming situation is to read data from an input file but not know how much data the file contains
- In these situations you need to check for the end of the file
- There are three common ways to test for the end of a file:
  1. Put a sentinel value at the end of the file and test for it.
  2. Throw and catch an end-of-file exception.
  3. Test for a special character that signals the end of the file (text files often have such a character).

The **`EOFException`** Class

- Many (but not all) methods that read from a file throw an end-of-file exception (`EOFException`) when they try to read beyond the file
  - » all the `DataInputStream` methods in Display 9.3 do throw it
- The end-of-file exception can be used in an "infinite" (`while(true)`) loop that reads and processes data from the file
  - » the loop terminates when an `EOFException` is thrown
- The program is written to continue normally after the `EOFException` has been caught

Using **`EOFException`**

`main` method from `EOFExceptionDemo`

```
try
{
   DataInputStream inputStream =
     new DataInputStream(new FileInputStream("numbers.dat"));
   int n;

   System.out.println("Reading ALL the integers");
   System.out.println("in the file numbers.dat.");
   try
   {
      while (true)
      {
         n = inputStream.readInt();
         System.out.println(n);
      }
   }
   catch(EOFException e)
   {
      System.out.println("End of reading from file.");
   }
   inputStream.close();
}
catch(FileNotFoundException e)
{
   System.out.println("Cannot find file numbers.dat.");
}
catch(IOException e2)
{
   System.out.println("Problem with input from file numbers.dat.");
}
```

Intentional "infinite" loop to process data from input file

Loop exits when end-of-file exception is thrown

Processing continues after `EOFException`: the input file is closed

Note order of catch blocks: the most specific is first and the most general last

Adding File I/O Capability to Classes

- Classes that perform I/O using the keyboard and screen can (and usually should) be generalized to do file I/O
- Overload the methods to read input and write output to include methods with input and output stream arguments
  - » the default (methods with no arguments) it to use the keyboard and screen for I/O
- See the `Species` program in the text for an example
- Note: file names alone are presumed in the same directory/folder as the program
  - » you can also use a full or relative path name if the file is in a different directory/folder

The **`File`** Class

- Acts like a wrapper class for file names
- A file name like "`numbers.dat`" has only `String` properties
- But a file name of type `File` has some very useful methods
  - » `exists`: tests to see if a file already exists
  - » `canRead`: tests to see if the operating system will let you read a file
- `FileInputStream` and `FileOutputStream` have constructors that take a `File` argument as well as constructors that take a `String` argument
- The text shows some additional useful `File` methods

Using Path Names

- *Path name*—gives name of file and tells which directory the file is in
- *Relative path name*—gives the path starting with the directory that the program is in
- Typical UNIX path name:

```
/user/smith/home.work/java/FileClassDemo.java
```

- Typical Windows path name:

```
D:\Work\Java\Programs\FileClassDemo.java
```

- When a backslash is used in a quoted string it must be written as two backslashes since backslash is the escape character:

```
"D:\\Work\\Java\\Programs\\FileClassDemo.java"
```

- Java will accept path names in UNIX or Windows format, regardless of which operating system it is actually running on.

Summary
Part 1

- *Text files* contain strings of printable characters; they look intelligible to humans when opened in a text editor.
- *Binary files* contain numbers or data in non-printable codes; they look *un*intelligible to humans when opened in a text editor.
- Java can process both binary and text files, but binary files are more common when doing file I/O.
- The class `DataOutputStream` is used to write output to a binary file.

Summary
Part 2

- The class `DataInputStream` is used to read input from a binary file.
- Always check for the end of the file when reading from a file. The way you check for end-of-file depends on the method you use to read from the file.
- A file name can be read from the keyboard into a `String` variable and the variable used in place of a file name.
- The class `File` has methods to test if a file exists and if it is read- and/or write-enabled.