Text File I/O

Text Files and Binary Files

- Files that are designed to be read by human beings, and that can be read or written with an editor are called *text files*
  - Text files can also be called ASCII files because the data they contain uses an ASCII encoding scheme
  - An advantage of text files is that the are usually the same on all computers, so that they can move from one computer to another

Text Files and Binary Files

- Files that are designed to be read by programs and that consist of a sequence of binary digits are called *binary files*
  - Binary files are designed to be read on the same type of computer and with the same programming language as the computer that created the file
  - An advantage of binary files is that they are *more efficient to process* than text files
  - Unlike most binary files, Java binary files have the advantage of being platform independent also
- For this course, we will deal only with text files

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");
```

`System.in`, `System.out`, and `System.err`

- The standard streams **`System.in`**, **`System.out`**, and **`System.err`** are automatically available to every Java program
  - **`System.out`** is used for normal screen output
  - **`System.err`** is used to output error messages to the screen
- The **`System`** class provides three methods (**`setIn`**, **`setOut`**, and **`setErr`**) for redirecting these standard streams:

```
public static void setIn(InputStream inStream)
public static void setOut(PrintStream outStream)
public static void setErr(PrintStream outStream)
```

File Names

- The rules for how file names should be formed depend on a given operating system, not Java
  - When a file name is given to a java constructor for a stream, it is just a string, not a Java identifier (e.g., `"fileName.txt"`)
  - Any suffix used, such as `.txt` has no special meaning to a Java program

Path Names

- When a file name is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run
- If it is not in the same directory, the full or relative path name must be given

Path Names

- A *path name* not only gives the name of the file, but also the directory or folder in which the file exists
- A *full path name* gives a complete path name, starting from the root directory
- A *relative path name* gives the path to the file, starting with the directory in which the program is located

Path Names

- The way path names are specified depends on the operating system
  - A typical Windows path name that could be used as a file name argument is

    `"C:\\user\\data\\data.txt"`
  - A Java program will accept a path name written in either Windows or Unix format regardless of the operating system on which it is run

A File Has Two Names

Every input file and every output file used by a program has two names:

1. The real file name used by the operating system
2. The name of the stream that is connected to the file

The actual file name is used to connect to the stream

The stream name serves as a temporary name for the file, and is the name that is primarily used within the program

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;
```

- The class **`PrintWriter`** has no constructor that takes a file name as its argument
  - It uses another class, **`FileOutputStream`**, to convert a file name to an object that can be used as the argument to its (the **`PrintWriter`**) constructor

Writing to a Text File

- 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));
```

  - The class **`FileOutputStream`** takes a string representing the file name as its argument
  - The class **`PrintWriter`** takes the anonymous **`FileOutputStream`** object as its argument

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`**, **`printf`**, 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

Sample Code

```
public class TextFileOutputDemo
{
  public static void main(String[] args)
  {
    PrintWriter outStream = null;
    try {
       outStream =
              new PrintWriter(new FileOutputStream("stuff.txt"));
    }
    catch(FileNotFoundException e)
    {
       System.err.println("Error opening the file stuff.txt.");
       System.exit(0);
    }
    outStream.println("The quick brown fox");
    outStream.println("jumped over the lazy dog.");

    outStream.close( );
  }
}
```

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

IOException

- When performing file I/O there are many situations in which an exception, such as **`FileNotFoundException`**, may be thrown
- Many of these exception classes are subclasses of the class **`IOException`**
  - The class **`IOException`** is the root class for a variety of exception classes having to do with input and/or output
- These exception classes are all checked exceptions
  - Therefore, they must be caught or declared in a throws clause

Catching IOException

```
public class TextFileOutputDemo
{
  public static void main(String[] args)
  {
   PrintWriter outStream = null;

   // OPEN the file here as in previous code
    try {
        outStream =
                new PrintWriter(new FileOutputStream("stuff.txt"));
        outStream.println("The quick brown fox");
        outStream.println("jumped over the lazy dog.");
   }
   catch (IOException e) {
        System.err.println (e.getMessage());
   }

   finally  { // always close the file
       outputStream.close( );
   }
  }
}
```

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

Reading From a Text File Using **`Scanner`**

- The class **`Scanner`** can be used for reading from a text file as well as the keyboard
- 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 (**`nextInt`**, **`nextLine`**) behave the same whether reading from a text file or the keyboard

Input from a Text File Using **`Scanner`** (Part 1 of 4)

Display 10.3 Reading Input from a Text File Using Scanner

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4
5  public class TextFileScannerDemo
6  {
7      public static void main(String[] args)
8      {
9         System.out.println("I will read three numbers and a line");
10        System.out.println("of text from the file morestuff.txt.");
11
12        Scanner inputStream = null;
13
14        try
15        {
16            inputStream =
17               new Scanner(new FileInputStream("morestuff.txt"));
18        }
```

(continued)

Input from a Text File Using `Scanner` (Part 2 of 4)

Display 10.3 Reading Input from a Text File Using Scanner

```
19          catch(FileNotFoundException e)
20          {
21              System.out.println("File morestuff.txt was not found");
22              System.out.println("or could not be opened.");
23              System.exit(0);
24          }
25              int n1 = inputStream.nextInt( );
26              int n2 = inputStream.nextInt( );
27              int n3 = inputStream.nextInt( );
28
29              inputStream.nextLine(); //To go to the next line
30
31              String line = inputStream.nextLine( );
32
```

(continued)

Input from a Text File Using **`Scanner`** (Part 3 of 4)

Display 10.3 Reading Input from a Text File Using Scanner

```
33              System.out.println("The three numbers read from the file are:");
34              System.out.println(n1 + ", " + n2 + ", and " +  n3);
35
36              System.out.println("The line read from the file is:");
37              System.out.println(line);
38
39              inputStream.close( );
40          }
41      }
```

File morestuff.txt

```
1 2
3 4
He is a jolly good fellow
```

This file could have been made with a text editor or by another Java program.

(continued)

Input from a Text File Using **`Scanner`** (Part 4 of 4)

Display 10.3 **Reading Input from a Text File Using** Scanner

SCREEN OUTPUT

```
I will read three numbers and a line
of text from the file morestuff.txt.
The three numbers read from the file are:
1, 2, and 3
The line read from the file is:
He is a jolly good fellow
```

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

Checking for the End of a Text File with **`hasNextLine`** (Part 1 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.PrintWriter;
5  import java.io.FileOutputStream;
6
7  public class HasNextLineDemo
8  {
9      public static void main(String[] args)
10     {
11         Scanner inputStream = null;
12         PrintWriter outputStream = null;
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 2 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
13          try
14          {
15             inputStream =
16                new Scanner(new FileInputStream("original.txt"));
17             outputStream = new PrintWriter(
18                             new FileOutputStream("numbered.txt"));
19          }
20          catch(FileNotFoundException e)
21          {
22             System.out.println("Problem opening files.");
23             System.exit(0);
24          }

25          String line = null;
26          int count = 0;
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 3 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

```
27          while (inputStream.hasNextLine( ))
28          {
29              line = inputStream.nextLine( );
30              count++;
31              outputStream.println(count + " " + line);
32          }

33          inputStream.close( );
34          outputStream.close( );
35      }

36  }
```

(continued)

Checking for the End of a Text File with **`hasNextLine`** (Part 4 of 4)

Display 10.4 Checking for the End of a Text File with `hasNextLine`

File original.txt

```
Little Miss Muffet
sat on a tuffet
eating her curves away.
Along came a spider
who sat down beside her
and said "Will you marry me?"
```

File numbered.txt (after the program is run)

```
1 Little Miss Muffet
2 sat on a tuffet
3 eating her curves away.
4 Along came a spider
5 who sat down beside her
6 and said "Will you marry me?"
```

Checking for the End of a Text File with **`hasNextInt`** (Part 1 of 2)

Display 10.5 Checking for the End of a Text File with `hasNextInt`

```
1   import java.util.Scanner;
2   import java.io.FileInputStream;
3   import java.io.FileNotFoundException;

4   public class HasNextIntDemo
5   {
6       public static void main(String[] args)
7       {
8           Scanner inputStream = null;

9           try
10          {
11             inputStream =
12                new Scanner(new FileInputStream("data.txt"));
13          }
14          catch(FileNotFoundException e)
15          {
16             System.out.println("File data.txt was not found");
17             System.out.println("or could not be opened.");
18             System.exit(0);
19          }
```

(continued)

Checking for the End of a Text File with **`hasNextInt`** (Part 2 of 2)

Display 10.5 Checking for the End of a Text File with `hasNextInt`

```
20          int next, sum = 0;
21          while (inputStream.hasNextInt( ))
22          {
23              next = inputStream.nextInt( );
24              sum = sum + next;
25          }

26          inputStream.close( );

27          System.out.println("The sum of the numbers is " + sum);
28      }
29  }
```

File data.txt

```
1  2
3  4 hi 5
```

Reading ends when either the end of the file is reach or a token that is not an `int` is reached. So, the `5` is never read.

SCREEN OUTPUT

```
The sum of the numbers is 10
```

hasNext

- The scanner also provide a more general method named **`hasNext`** that returns false if there are no more tokens of any kind in the file.
- **`hasNext`** can be used when the file contains different kinds of data