Topic 1: The Java Programming Language

Part 3: Files, Exceptions

Files

File Input and Output

- Reentering data all the time could get tedious for the user.
- The data can be saved to a file.
  - Files can be *input files* or *output files*.
- Files:
  - Files have to be opened.
  - Data is then written to the file.
  - The file must be closed prior to program termination.
- In general, there are two types of files:
  - binary
  - text

The `PrintWriter` Class (1)

- The `PrintWriter` class allows you to write data to a file using the `print` and `println` methods, as you have been using to display data on the screen.
- Just as with the `System.out` object, the `println` method of the `PrintWriter` class will place a newline character after the written data.
- The `print` method writes data without writing the newline character.

The `PrintWriter` Class (2)

To use the `PrintWriter` class, put the following `import` statement at the top of the source file:

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

```
PrintWriter outputFile = new PrintWriter("Names.txt");
outputFile.println("Chris");
outputFile.println("Katie");
outputFile.println("Jean");
outputFile.close();
```

Exceptions (1)

- When something unexpected happens in a Java program, an *exception* is thrown.
- The method that is executing when the exception is thrown must either handle the exception or pass it up the line.
- Handling the exception will be discussed later.
- To pass it up the line, the method needs a `throws` clause in the method header.

Exceptions (2)

- To insert a `throws` clause in a method header, simply add the word *throws* and the name of the expected exception.
- `PrintWriter` objects can throw an `IOException`, so we write the `throws` clause like this:

```
public static void main(String[] args) throws
   IOException
```

Appending Text to a File

- To avoid erasing a file that already exists, create a `FileWriter` object in this manner:

```
FileWriter fw =
new FileWriter("names.txt", true);
```

- Then, create a `PrintWriter` object in this manner:

```
PrintWriter fw = new PrintWriter(fw);
```

Specifying a File Location

- On a Windows computer, paths contain backslash (\) characters.
- Remember, if the backslash is used in a string literal, it is the escape character so you must use two of them:

```
PrintWriter outFile = new
    PrintWriter("A:\\PriceList.txt");
```

- This is only necessary if the backslash is in a string literal.
- If the backslash is in a `String` object then it will be handled properly.
- Fortunately, Java allows Unix style filenames using the forward slash (/) to separate directories:

```
PrintWriter outFile = new
  PrintWriter("/home/rharrison/names.txt");
```

Reading Data From a File (1)

- You use the `File` class and the `Scanner` class to read data from a file:

```
File myFile = new File("Customers.txt");
Scanner inputFile = new Scanner(myFile);
```

Reading Data From a File (2)

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter the filename: ");
String filename = keyboard.nextLine();
File file = new File(filename);
Scanner inputFile = new Scanner(file);
```

- The lines above:
  - Creates an instance of the `Scanner` class to read from the keyboard
  - Prompt the user for a filename
  - Get the filename from the user
  - Create an instance of the `File` class to represent the file
  - Create an instance of the `Scanner` class that reads from the file

Reading Data From a File (3)

- Once an instance of `Scanner` is created, data can be read using the same methods that you have used to read keyboard input (`nextLine`, `nextInt`, `nextDouble`, etc).
- The `Scanner` class can throw an `IOException` when a `File` object is passed to its constructor.

Detecting The End of a File

- The `Scanner` class's `hasNext()` method will return true if another item can be read from the file.

```
// Open the file.
File file = new File(filename);
Scanner inputFile = new Scanner(file);
// Read until the end of the file.
while (inputFile.hasNext())
{
   String str = inputFile.nextLine();
   System.out.println(str);
}
inputFile.close();// close the file when done.
```

(DemoFileIO.java)
(DemoFileIO2.java)

Exceptions

Handling Exceptions (1)

- To handle an exception, you use a *try* statement.

```
try
   {
      (try block statements...)
   }
   catch (ExceptionType ParameterName)
   {
      (catch block statements...)
   }
```

- First the keyword `try` indicates a block of code will be attempted (the curly braces are required).
- This block of code is known as a *try block*.

Handling Exceptions (2)

- A *try block* is:
  - one or more statements that are executed, and
  - can potentially throw an exception.
- The application will not halt if the try block throws an exception.
- After the try block, a `catch` clause appears.

Handling Exceptions (3)

- A catch clause begins with the key word `catch`:

```
catch (ExceptionType ParameterName)
```

  - *`ExceptionType`* is the name of an exception class and
  - *`ParameterName`* is a variable name which will reference the exception object if the code in the try block throws an exception.
- The code that immediately follows the catch clause is known as a *catch block* (the curly braces are required).
- The code in the catch block is executed if the try block throws an exception.

Handling Exceptions (4)

- This code is designed to handle a `FileNotFoundException` if it is thrown.

```
try
  {
    File file = new File ("MyFile.txt");
    Scanner inputFile = new Scanner(file);
  }
  catch (FileNotFoundException e)
  {
    System.out.println("File not found.");
  }
```

- The Java Virtual Machine searches for a `catch` clause that can deal with the exception.

Handling Exceptions (5)

- The parameter must be of a type that is compatible with the thrown exception's type.
- After an exception, the program will continue execution at the point just past the catch block.

- Each exception object has a method named `getMessage` that can be used to retrieve the default error message for the exception.

(ExceptionMessage.java)
(ParseIntError.java)

The `finally` Clause (1)

- The try statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all of the `catch` clauses.

```
try
{
   (try block statements...)
}
catch (ExceptionType ParameterName)
{
   (catch block statements...)
}
finally
{
   (finally block statements...)
}
```

The `finally` Clause (2)

- The *finally block* is one or more statements,
  - that are always executed after the try block has executed and
  - after any catch blocks have executed if an exception was thrown.
- The statements in the finally block execute whether an exception occurs or not.

Uncaught Exceptions

- When an exception is thrown, it cannot be ignored.
- It must be handled by the program, or by the default exception handler.
- When the code in a method throws an exception:
  - normal execution of that method stops, and
  - the JVM searches for a compatible exception handler inside the method.
- If there is no exception handler inside the method:
  - control of the program is passed to the previous method in the call stack.
  - If that method has no exception handler, then control is passed again, up the call stack, to the previous method.
- If control reaches the `main` method:
  - the main method must either handle the exception, or
  - the program is halted and the default exception handler handles the exception.

Checked and Unchecked Exceptions (1)

- There are two categories of exceptions:
  - unchecked
  - checked.
- *Unchecked exceptions* are those that are derived from the `Error` class or the `RuntimeException` class.
- Exceptions derived from `Error` are thrown when a critical error occurs, and should not be handled.
- `RuntimeException` serves as a superclass for exceptions that result from programming errors.

Checked and Unchecked Exceptions (2)

- These exceptions can be avoided with properly written code.
- Unchecked exceptions, in most cases, should not be handled.
- All exceptions that are *not* derived from `Error` or `RuntimeException` are *checked exceptions*.
- If the code in a method can throw a checked exception, the method:
  - must handle the exception, or
  - it must have a `throws` clause listed in the method header.
- The `throws` clause informs the compiler what exceptions can be thrown from a method.

(NumberInput.java)

Throwing Exceptions (1)

- You can write code that:
  - throws one of the standard Java exceptions, or
  - an instance of a custom exception class that you have designed.
- The `throw` statement is used to manually throw an exception.

```
throw new ExceptionType(MessageString);
```

- The `throw` statement causes an exception object to be created and thrown.

Throwing Exceptions (2)

- The *MessageString* argument contains a custom error message that can be retrieved from the exception object's `getMessage` method.
- If you do not pass a message to the constructor, the exception will have a null message.

```
throw new Exception("Out of fuel");
```

  - *Note: Don't confuse the `throw` statement with the `throws` clause.*

(DieExceptionDemo.java)