Supplement: Text IO Using Reader and Writer

For Introduction to Java Programming
By Y. Daniel Liang

0 Introduction

The text introduced text I/O using the **Scanner** class and **PrintWriter** class, which greatly simplified Text I/O. Prior to JDK 1.5, you have to use several classes to perform text I/O. If you are interested to know how to perform Text I/O in the old way, this supplement is for you.

1 The Reader and Writer Classes

Figure 1 lists the classes for performing text I/O.

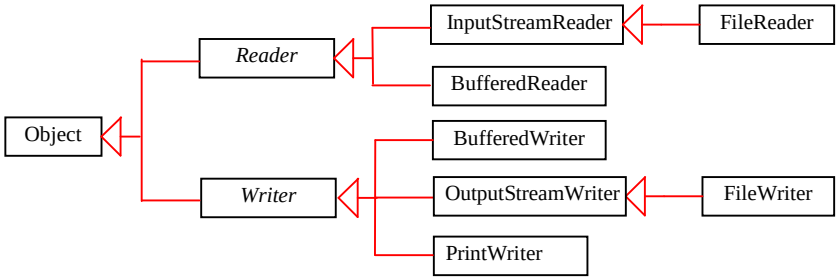

Figure 1
***Reader**, **Writer**, and their subclasses are for text I/O.*

Reader is the root for text input classes, and **Writer** is the root for text output classes. Figures 16.6 and 16.7 list all the methods in **Reader** and **Writer**.

java.io.Reader	
+read(): int	Reads the next character from the input stream. The value returned is an int in the range from 0 to 65535, which represents a Unicode character. Returns -1 at the end of the stream.
+read(cbuf: char[]): int	Reads characters from the input stream into an array. Returns the actual number of characters read. Returns -1 at the end of the stream.
+read(cbuf: char[], off: int, len: int): int	Reads characters from the input stream and stores them in cbuf[off], cbuf[off+1], …, cbuf[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+close(): void	Closes this input stream and releases any system resources associated with it.
+skip(n: long): long	Skips over and discards n characters of data from this input stream. The actual number of characters skipped is returned.
+markSupported(): boolean	Tests whether this input stream supports the mark and reset methods.
+mark(readAheadLimit: int): void	Marks the current position in this input stream.
+ready(): boolean	Returns true if this input stream is ready to be read.
+reset(): void	Repositions this stream to the position at the time the mark method was last called on this input stream.

Figure 2

*The abstract **Reader** class defines the methods for reading a stream of characters.*

> **NOTE**
> The **read()** method reads a character. If no data are available, it blocks the thread from executing the next statement. The thread that invokes the **read()** method is suspended until the data become available.

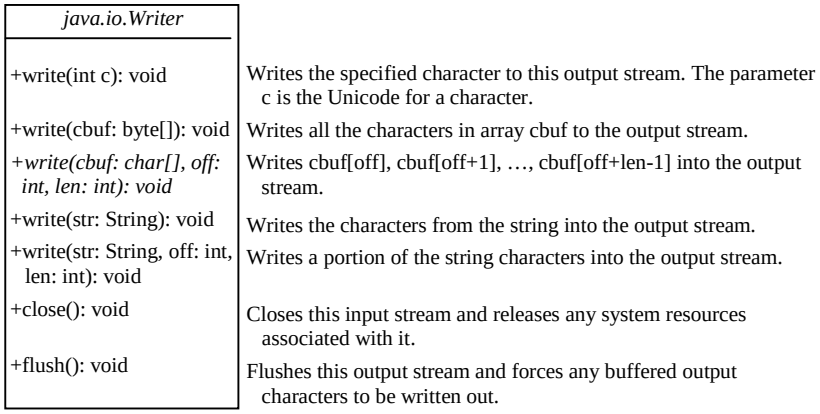

Figure 3
*The abstract **Writer** class defines the methods for writing a stream of characters.*

> **NOTE**
> All the methods in the text I/O classes except **PrintWriter** are declared to throw **java.io.IOException**.

2 FileReader/FileWriter

FileReader/FileWriter are convenience classes for reading/writing characters from/to files using the default character encoding on the host computer. **FileReader/FileWriter** associates an input/output stream with an external file.

All the methods in **FileReader**/**FileWriter** are inherited from their superclasses. To construct a **FileReader**, use the following constructors:

```
public FileReader(String filename)
public FileReader(File file)
```

A **java.io.FileNotFoundException** would occur if you attempt to create a **FileReader** with a nonexistent file. For

example, Listing 1 reads and displays all the characters from the file temp.txt.

Listing 1 TestFileReader.java (Input Using FileReader)

```
import java.io.*;

public class TestFileReader {
  public static void main(String[] args) {
    FileReader input = null;
    try {
      // Create an input stream
      input = new FileReader("temp.txt");

      int code;
      // Repeatedly read a character and display it on the console
      while ((code = input.read()) != -1)
        System.out.print((char)code);
    }
    catch (FileNotFoundException ex) {
      System.out.println("File temp.txt does not exist");
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    finally {
      try {
        input.close(); // Close the stream
      }
      catch (IOException ex) {
        ex.printStackTrace();
      }
    }
  }
}
```

The constructors and methods in **FileReader** may throw **IOException**, so they are invoked from a try-catch block. Since **java.io.FileNotFoundException** is a subclass of **IOException**, the catch clause for **FileNotFoundException** (Line 15) is put before the **catch** clause for **IOException** (Line 18). Closing files in the **finally** class ensures that the file are always closed in any case (Line 23).

Recall that Java allows the assignment operator in an expression (see page 37). The expression **((code = input.read()) != -1)** (Line 12) reads a byte from

input.read(), assigns it to **code**, and checks whether it is –1. The input value of –1 signifies the end of a file.

NOTE

Attempting to read data after the end of a file is reached would cause **java.io.EOFException**.

To construct a **FileWriter**, use the following constructors:

```
public FileWriter(String filename)
public FileWriter(File file)
public FileWriter(String filename, boolean append)
public FileWriter(File file, boolean append)
```

If the file does not exist, a new file will be created. If the file already exists, the first two constructors will delete the current contents of the file. To retain the current content and append new data into the file, use the last two constructors by passing true to the **append** parameter. For example, suppose the file temp.txt exists, Listing 2 appends a new string, "Introduction to Java," to the file.

Listing 2 TestFileWriterer.java (Output Using FileWriter)

```
import java.io.*;

public class TestFileWriter {
  public static void main(String[] args) throws IOException {
    // Create an output stream to the file
    FileWriter output = new FileWriter("temp.txt", true);

    // Output a string to the file
    output.write("Introduction to Java");

    // Close the stream
    output.close();
  }
}
```

If you replace Line 6 with the following statement,

```
FileWriter output = new FileWriter("temp.txt");
```

the current contents of the file are lost. To avoid this, use the **File** class's **exists()** method to check whether a file exists before creating it, as follows:

```
File file = new File("temp.txt");
if (!file.exists()) {
  FileWriter output = new FileWriter(file);
}
```

3 InputStreamReader/OutputStreamWriter (Optional)

InputStreamReader/**OutputStreamWriter** are used to convert between bytes and characters. Characters written to an **OutputStreamWriter** are encoded into bytes using a specified encoding scheme. Bytes read from an **InputStreamReader** are decoded into characters using a specified encoding scheme. You can specify an encoding scheme using a constructor of **InputStreamReader**/**OutputStreamWriter**. If no encoding scheme is specified, the system's default encoding scheme is used. All the methods in **InputStreamReader**/**OutputStreamWriter** are inherited from **Reader**/**Writer** except **getEncoding()**, which returns the name of the encoding being used by this stream. Since **FileReader** is a subclass of **InputStreamReader**, you can also invoke **getEncoding()** from a **FileReader** object. For example, the following code

```
public static void main(String[] args) throws IOException {
  FileReader input = new FileReader("temp.txt");
  System.out.println("Default encoding is " +
input.getEncoding());
}
```

displays the default encoding for the host machine. For a list of encoding schemes supported in Java, please see **http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html** and **http://mindprod.com/jgloss/encoding.html**.

> NOTE: Java programs use Unicode. When you read a character from a **FileReader** stream, the Unicode code of the character is returned. The encoding of the character in the file may be different from the Unicode encoding. Java automatically converts it to the Unicode. When you write a character to a **FileWriter** stream, Java automatically converts the Unicode of the

character to the encoding specified for the file. This is pictured in Figure 4.

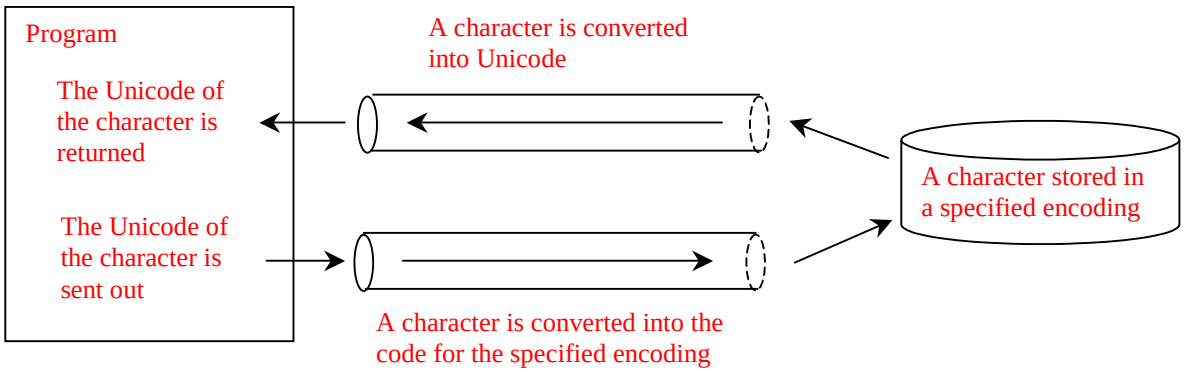

Figure 4
The encoding of the file may be different from the encoding used in the program.

4 BufferedReader/BufferedWriter

BufferedReader/**BufferedWriter** can be used to speed up input and output by reducing the number of reads and writes. Buffered streams employ a buffered array of characters that acts as a cache. In the case of input, the array reads a chunk of characters into the buffer before the individual characters are read. In the case of output, the array accumulates a block of characters before writing the entire block to the output stream.

The use of buffered streams enables you to read and write a chunk of characters at once instead of reading or writing the bytes one at a time. You can wrap a **BufferedReader**/**BufferedWriter** around any **Reader**/**Writer** streams. The following constructors are used to create a **BufferedReader**/**BufferedWriter**:

```
// Create a BufferedReader
public BufferedReader(Reader in)
public BufferedReader(Reader in, int bufferSize)

// Create a BufferedWriter
public BufferedWriter(Writer out)
public BufferedWriter(Writer out, int bufferSize)
```

If no buffer size is specified, the default size is 8192 characters. A buffered input stream reads as many data as

possible into its buffer in a single read call. By contrast, a buffered output stream calls the write method only when its buffer fills up or when the **flush()** method is called.

The buffered stream classes inherit methods from their superclasses. In addition to using the methods from their superclasses, **BufferedReader** has a **readLine()** method to read a line, and **BufferedWriter** has a **newLine()** method to write a line separator. If the end of stream is reached, **readLine()** returns **null**.

> **NOTE**
> The **readLine()** method return a line without the line separator. The **newLine()** method writes a system-dependent line separator to a file. The line separator string is defined by the system and is not necessarily a single ('\n') character. To get the system line separator, use

```
static String lineSeparator = (String)java.security.
  AccessController.doPrivileged(
    new sun.security.action.GetPropertyAction("line.separator"));
```

Listing 3 uses **BufferedReader** to read text from the file "Welcome.java", displays the text on the console, and copies the text to a file named "Welcome.java~".

Listing 3 TestBufferedReaderWriter.java (Using Buffers)

```
import java.io.*;

public class TestBufferedReaderWriter {
  public static void main(String[] args) throws IOException {
    // Create an input stream
    BufferedReader input =
      new BufferedReader(new FileReader("Welcome.java"));

    // Create an output stream
    BufferedWriter output =
      new BufferedWriter(new FileWriter("Welcome.java~"));

    // Repeatedly read a line and display it on the console
    String line;
    while ((line = input.readLine()) != null) {
```

```
        System.out.println(line);
        output.write(line);
        output.newLine(); // Write a line separator
      }

      // Close the stream
      input.close();
      output.close();
    }
  }
```

A **BufferedReader** is created on top of a **FileReader** (Lines 6-7), and a **BufferedWriter** on top of a **FileWriter** (Lines 10-11). Data from the file are read repeatedly, one line at a time (Line 15). Each line is displayed in Line 16, and output to a new file (Line 17) with a line separator (Line 18).

You are encouraged to rewrite the program without using buffers and then compare the performance of the two programs. This will show you the improvement in performance obtained by using buffers when reading from a large file.

> **TIP**
> Since physical input and output involving I/O devices are typically very slow compared with CPU processing speeds, you should use buffered input/output streams to improve performance.

5 PrintWriter and PrintStream

BufferedWriter is used to output characters and strings. **PrintWriter** and **PrintStream** can be used to output objects, strings, and numeric values as text. **PrintWriter** was introduced in JDK 1.2 to replace **PrintStream**. Both classes are almost identical in the sense that they provide the same function and the same methods for outputting strings and numeric values as text. **PrintWriter** is more efficient than **PrintStream** and therefore is the class you ought to use.

PrintWriter and **PrintStream** contain the following overloaded **print** and **println** methods:

```
public void print(Object o)
public void print(String s)
public void print(char c)
public void print(char[] cArray)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(boolean b)
```

```
public void println(Object o)
public void println(String s)
public void println(char c)
public void println(char[] cArray)
public void println(int i)
public void println(long l)
public void println(float f)
public void println(double d)
public void println(boolean b)
```

A numeric value, character, or boolean value is converted into a string and printed to the output stream. To print an object is to print the object's string representation returned from the **toString()** method. **PrintWriter** and **PrintStream** also contain the **printf** method for printing formatted output, which was introduced in Section 2.17, "Formatting Output."

You have already used these methods in **System.out**. **out** is declared as a static variable of the **PrintStream** type in the **System** class. By default, **out** refers to the standard output device, that is, the screen console. You can use the **System.setOut(PrintStream)** to set a new **out**. Since **System** was introduced before **PrintWriter**, **out** is declared **PrintStream** and not **PrintWriter**.

This section introduces **PrintWriter**, but **PrintStream** can be used in the same way. To construct a **PrintWriter**, use the following constructors:

```
public PrintWriter(Writer out)
public PrintWriter(Writer out, boolean autoFlush)
```

If **autoFlush** is **true**, the **println** methods will cause the buffer to be flushed.

> **NOTE**
> The constructors and methods in **PrintWriter** and **PrintStream** do not throw an **IOException**. So you don't need to invoke them from a **try-catch** block.

Listing 4 is an example of generating ten integers and storing them in a text file using **PrintWriter**. The example later writes the data back from the file and computes the total.

Listing 4 TestPrintWriter.java (Output Data and Strings)

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

public class TestPrintWriter {
  public static void main(String[] args) throws IOException {
    // Check if file temp.txt already exists
    File file = new File("temp.txt");
    if (file.exists()) {
      System.out.println("File temp.txt already exists");
      System.exit(0);
    }

    // Create an output stream
    PrintWriter output = new PrintWriter(new FileWriter(file));

    // Generate ten integers and write them to a file
    for (int i = 0; i < 10; i++) {
      output.print((int)(Math.random() * 100) + " ");
    }

    // Close the output stream
    output.close();

    // Open an input stream
    BufferedReader input =
      new BufferedReader(new FileReader("temp.txt"));

    int total = 0;  // Store total
    String line;
    while ((line = input.readLine()) != null) {
      // Extract numbers using string tokenizer
      StringTokenizer tokens = new StringTokenizer(line);
      while (tokens.hasMoreTokens())
        total += Integer.parseInt(tokens.nextToken());
    }

    System.out.println("Total is " + total);

    // Close input stream
    input.close();
  }
}
```

Lines 7-11 check whether the file exists. If so, exit the program. Line 14 creates a **PrintWriter** stream for the file. Lines 17-19 generate ten random integers and output the integers separated by a space.

Line 25 opens the file for input. Lines 30-35 read all the lines from the file and use **StringTokenizer** to extract tokens from the line. Each token is converted to a number and added to the total (Line 34).

6 Case Study: Text Viewer

This case study writes a program that views a text file in a text area. The user enters a filename in a text field and clicks the View button; the file is then displayed in a text area, as shown in Figure 5.

Figure 5
The program displays the specified file in the text area.

Clearly, you need to use text input to read a text file. Normally, you should use **BufferedReader** wrapped on a **FileReader** to improve performance. When the View button is pressed, the program gets the input filename from the text field; it then creates a text input stream. The data are read one line at a time and appended to the text area for display. Listing 5 displays the source code for the program.

Listing 5 FileViewer.java (View Text Files)

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

public class FileViewer extends JFrame implements
ActionListener {
  // Button to view a file
  private JButton jbtView = new JButton("View");

  // Text field to the receive file name
  private JTextField jtfFilename = new JTextField(12);
```

```
// Text area to display the file
private JTextArea jtaFile = new JTextArea();

public FileViewer() {
  // Panel p to hold a label, a text field, and a button
  Panel p = new Panel();
  p.setLayout(new BorderLayout());
  p.add(new Label("Filename"), BorderLayout.WEST);
  p.add(jtfFilename, BorderLayout.CENTER);
  p.add(jbtView, BorderLayout.EAST);

  // Add jtaFile to a scroll pane
  JScrollPane jsp = new JScrollPane(jtaFile);

  // Add jsp and p to the frame
  getContentPane().add(jsp, BorderLayout.CENTER);
  getContentPane().add(p, BorderLayout.SOUTH);

  // Register listener
  jbtView.addActionListener(this);
}

/** Handle the View button */
public void actionPerformed(ActionEvent e) {
  if (e.getSource() == jbtView)
    showFile();
}

/** Display the file in the text area */
private void showFile() {
  // Use a BufferedReader to read text from the file
  BufferedReader input = null;

  // Get file name from the text field
  String filename = jtfFilename.getText().trim();

  String inLine;

  try {
    // Create a buffered stream
    input = new BufferedReader(new FileReader(filename));

    // Read a line and append the line to the text area
    while ((inLine = input.readLine()) != null) {
      jtaFile.append(inLine + '\n');
    }
  }
  catch (FileNotFoundException ex) {
    System.out.println("File not found: " + filename);
  }
  catch (IOException ex) {
```

```
      System.out.println(ex.getMessage());
    }
    finally {
      try {
        if (input != null) input.close();
      }
      catch (IOException ex) {
        System.out.println(ex.getMessage());
      }
    }
  }

  public static void main(String[] args) {
    FileViewer frame = new FileViewer();
    frame.setTitle("FileViewer");
    frame.setSize(400, 300);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

A **BufferedReader** is created on top of a **FileReader** (Line 53). Data from the file are read repeatedly, one line at a time, and appended to the text area (Line 57). If the file does not exist, the catch clause in Lines 60-62 catches and processes it. All other I/O errors are caught and processed in Lines 63-65. Whether the program runs with or without errors, the input stream is closed in Lines 66-73.