Module 11 - Output and Input Streams

General Notes

Module Content

Instructor Slides

> **Instructor Slides:** PDF | Slides

Instructor Video - Reading From a File

Instructor Video - Writing To a File

Instructor Video - Parse Data

ZyBooks

Output Streams

- Programs need a way to output data to a screen, file, or elsewhere.
- `OutputStream` is a class that supports output.
- Provides several overloaded methods for writing a sequence of bytes to a destination.

- The `print()` and `println()` methods are overloaded to support standard data types.
- `System.out` is a predefined `OutputStream` object reference associated with a system's standard output, usually a computer screen.
- The `PrintStream` class extends the base functionality of the `OutputStream` class and provides the `print()` and `println()` methods for converting different types of data into a sequence of characters.
- The `System.out` is essentially an object that has been associated with the screen.
  earlier section.

Input Streams

- Programs need to receive input data, whether from a keyboard, touchscreen, or elsewhere.
- `InputStream` is a class for achieving such input.
- `InputStream` provides several overloaded `read()` methods that allow a programmer to extract bytes from a particular source.
- `System.in` is a predefined input stream object reference that is associated with a system's standard input, which is usually a keyboard.
- The `System.in` input stream automatically reads the standard input from a memory region, known as a buffer, that the operating system fills with the input data.

Reading one byte at a time:

```
import java.io.IOException;

public class InputStreamReader {
   public static void main (String[] args)
   throws IOException {
      int usrInput;

      // Read 1st byte
      usrInput = System.in.read();
      // Read 2nd byte
      usrInput = System.in.read();
      // Read 3rd byte
      usrInput = System.in.read();
      // Read 4th byte
      usrInput = System.in.read();
      // Read 5th byte (empty buffer)
      usrInput = System.in.read();
   }
}
```

Byte stream, throws clause, and Scanner

- A byte stream is used by programs to input or output 8-bits (a byte).
- `System.in` is an input byte stream, and thus the `read()` method reads the first 8-bit ASCII value available from the operating system's buffer.
- When using an `InputStream`, a programmer must append the clause `throws IOException` to the definition of `main()`.
  - A throws clause tells the Java virtual machine that the corresponding method may exit unexpectedly due to an exception.
- Instead of directly reading bytes from `System.in`, a program typically uses the `Scanner` class as a wrapper that augments `System.in` by automatically scanning a sequence of bytes and converting those bytes to the desired data type.
- To initialize a `Scanner` object, a programmer can pass an `InputStream`, such as `System.in`, as an argument to the constructor.
- `System.in.read()` returns `-1` when data is no longer available.

Exploring Further

- Oracle's OutputStream class specification
- Oracle's PrintStream class specification
- Oracle's System class specification
- Oracle's Java tutorials on I/O Streams
- Oracle's InputStream class specification

Conclusion

- Output and input streams are essential for programs to output and receive data.
- `OutputStream` and `PrintStream` classes are used for output and `InputStream` and `Scanner` classes are used for input.
- The `System.out` is an `OutputStream` object reference that is associated with a system's standard output and `System.in` is an `InputStream` object reference associated with a system's standard input.
- The `Scanner` class is often used to extract strings or integers from an input stream.

Output formatting

`printf()` and `format()` methods

- Programmer can adjust the way that a program's output appears, a task known as **output formatting**
- `System.out` provides the methods `printf()` and `format()` for output formatting
- Both methods are equivalent, so this discussion only refers to `printf()`
- The first argument of the `printf()` method, the **format string**, specifies the format of the text to print along with any number of placeholders for printing numeric values
- Placeholders are known as **format specifiers** and specify the type of value to print in its place
- A format specifier begins with the `%` character followed by another character that indicates the value type to be printed. Ex: `%d` indicates an integer

type, and `%s` indicates a string type

Format specifiers for the `printf()` and `format()` methods

Format specifier	Data type(s)	Notes
%c	`char`	Prints a single Unicode character
%d	int, long, short	Prints a decimal integer value
%o	int, long, short	Prints an octal integer value
%h	int, char, long, short	Prints a hexadecimal integer value
%f	float, double	Prints a floating-point value
%e	float, double	Prints a floating-point value in scientific notation
%s	String	Prints the characters in a String variable or literal
%%		Prints the " % " character
%n		Prints the platform-specific new-line character

Floating-point Values & Formatting

- Formatting floating-point output is commonly done using **sub-specifiers**
- A sub-specifier provides formatting options for a format specifier and are included between the `%` and format specifier character
  - Ex: The `.1` sub-specifier in `printf("%.1f", myFloat);` causes the floating-point variable `myFloat` to be output with only 1 digit after the decimal point; if `myFloat` is **12.34**, the output would be **12.3**
- Format specifiers and sub-specifiers use the following form: `%(flags)(width)(.precision)specifier`

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to print. If the formatted value has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7.2f", 12.3456);` prints `Value: 12.35` (Note that the number is rounded up)
.precision	Specifies the number of digits to print following the decimal point. If the precision is not specified, a default precision of `6` is used.	`printf("%.4f", 12.3456);` prints `12.3456` (Note that the number is not rounded up)

flags	`-` (left aligns the output given the specified width, padding the output with spaces)
	`+` (prints a preceding `+` sign for positive values; negative numbers are always printed with the - sign)
	`0` (pads the output with 0's when the formatted value has fewer characters than the width)
	`space` (prints a preceding space for positive value)
		`printf("%+f", 12.3456);` prints `+12.345600`
		`printf("%-7.2f", 12.3456);` prints `12.35` (Note the two spaces after the number)
		`printf("%07.2f", 12.3456);` prints `0012.35`
		`printf("% f", 12.3456);` prints `12.345600` (Note the space before the number)

- Keep in mind that the `.` is counted as a character for the `width` sub-specifier.

Example Output Formatting for Floating-point Numbers

```
import java.util.Scanner;

public class FlyDrive {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      double miles;          // User defined distance
      double hoursFly;       // Time to fly distance
      double hoursDrive;     // Time to drive distance

      // Prompt user for distance
      System.out.print("Enter a distance in miles: ");
      miles = scnr.nextDouble();
      // Calculate the correspond time to fly/drive distance

      hoursFly = miles / 500.0;
      hoursDrive = miles / 60.0;

      // Output resulting values
      System.out.printf("%.2f miles would take:\n", miles);
      System.out.printf("%.2f hours to fly\n", hoursFly);
      System.out.printf("%.2f hours to drive\n", hoursDrive);
   }
}
```

Output:

```
Enter a distance in miles: 10.3
10.30 miles would take:
0.02 hours to fly
0.17 hours to drive
```

Integer Values and Formatting

- Formatting of integer values is also done using sub-specifiers
- The integer sub-specifiers are similar to the floating-point sub-specifiers except no .precision exists

Formatting of integer values is also done using sub-specifiers. The integer sub-specifiers are similar to the floating-point sub-specifiers except

no `.precision` exists. For the table below, assume `myInt` is `301`.

Sub-specifier	Description	Example
`width`	Specifies the minimum number of characters to print. If the formatted value has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or `0`'s if the `0` flag is specified).	`System.out.printf("Value: %7d", myInt);` `Value: 301`

flags	`-` : Left aligns the output given the specified width, padding the output with spaces.
	`+` : Print a preceding + sign for positive values. Negative numbers are always printed with the `-` sign.
	`0` : Pads the output with 0's when the formatted value has fewer characters than the width.
	`space` : Prints a preceding space for positive value.
		`System.out.printf("%+d", myInt);`
		`+301`
		`System.out.printf("%08d", myInt);`
		`00000301`
		`System.out.printf("%+08d", myInt);`
		`+0000301`

Example Output Formatting for Integers

```
public class CelestialBodyDist {
   public static void main(String[] args) {
      final long KM_EARTH_TO_SUN = 149598000;    // Dist from Earth to
sun
      final long KM_SATURN_TO_SUN = 1433449370;  // Dist from Saturn to
sun

      // Output distances with min number of characters
      System.out.printf("Earth is %12d", KM_EARTH_TO_SUN);
      System.out.printf(" kilometers from the sun.\n");
      System.out.printf("Saturn is %11d", KM_SATURN_TO_SUN);
      System.out.printf(" kilometers from the sun.\n");
   }
}
```

Output:

```
Earth is    149598000 kilometers from the sun.
Saturn is  1433449370 kilometers from the sun.
```

Strings

Strings may be formatted using sub-specifiers. For the table below, assume the `myString` variable is `"Formatting"`.

Sub-specifier	Description	Example
`width`	Specifies the minimum number of characters to print. If the string has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces.	`printf("%20s String", myString);`
`precision`	Specifies the maximum number of characters to print. If the string has more characters than the precision, the string will be truncated.	`printf("%.6s", myString);`
`flags`	`-`: Left aligns the output given the specified width, padding the output with spaces.	`printf("%-20s String", myString);`

Flushing output

Importance of Flushing output

- Printing characters from the buffer to the output device (e.g., screen) requires a time-consuming reservation of processor resources.
- Once the resources are reserved, moving characters is fast, whether there is 1 character or 50 characters to print.
- To preserve resources, the system may wait until the buffer is full, or at least has a certain number of characters before moving them to the output device.
- With fewer characters in the buffer, the system may wait until the resources are not busy.

Situations when flushing output may be necessary

- Sometimes a programmer does not want the system to wait.
- For example, in a very processor-intensive program, waiting could cause delayed and/or jittery output.

Flushing the buffer

- The PrintStream method `flush()` flushes the stream's buffer contents.
  - For example, the statement `System.out.flush();` writes the contents of the
    buffer for `System.out` to the computer screen.
- Most Java implementations make `System.out` flush when a newline character is output or `println()` method is called.

Additional Information

Flushing the output stream is an important concept in programming to ensure timely and accurate output to the user. It is also crucial in situations where the program needs to print output to an external device, such as a file or a network socket. The `flush()` method should be used judiciously to balance the need for immediate output with the need to preserve system resources.

Exploring Further

- Oracle's Java Formatter class specification

Streams using Strings

Scanner object with a String

- The Scanner object can accept different input sources including a String.
- A Scanner object associated with a String is often referred to as an **input string stream**.
- The Scanner object can use its methods like `nextInt()` and `next()` to break apart the String.
- A programmer has to know the String form, like "String String integer" or "integer String double", to parse individual items from

the String.

Reading From a String Using a Scanner Object

```
Scanner inSS = null;                // Input string stream
String userInfo;                    // Input string
String firstName;                   // First name
String lastName;                    // Last name
int userAge;                        // Age

userInfo = "Amy Smith 19";

// Init scanner object with string
inSS = new Scanner(userInfo);

// Parse name and age values from string
firstName = inSS.next();
lastName = inSS.next();
userAge = inSS.nextInt();

// Output parsed values
System.out.println("First name: " + firstName);
System.out.println("Last name: " + lastName);
System.out.println("Age: " + userAge);
```

Output:

```
First name: Amy
Last name: Smith
Age: 19
```

Using String streams to process input text

- String streams are useful to process user input line-by-line.
- The program reads the input line as a String, then extracts individual data items from that String.
- The program can extract input from the stream using the `next()` methods.

Using a String Stream to Process a Line of Input Text

```
Scanner scnr = new Scanner(System.in); // Input stream for standard input
Scanner inSS = null;                   // Input string stream
String lineString;                     // Holds line of text
String firstName;                      // First name
String lastName;                       // Last name
int userAge;                           // Age
boolean inputDone;                     // Flag to indicate next iteration

inputDone = false;

// Prompt user for input
System.out.println("Enter \"firstname lastname age\" on each line");
System.out.println("(\"Exit\" as firstname exits).\n");

// Grab data as long as "Exit" is not entered
while (!inputDone) {

   // Entire line into lineString
   lineString = scnr.nextLine();

   // Create new input string stream
   inSS = new Scanner(lineString);

   // Now process the line
   firstName = inSS.next();

   // Output parsed values
   if (firstName.equals("Exit")) {
      System.out.println("   Exiting.");

      inputDone = true;
   }
   else {
      lastName = inSS.next();
      userAge = inSS.nextInt();

      System.out.println("   First name: " + firstName);
      System.out.println("   Last name: " + lastName);
      System.out.println("   Age: " + userAge);
      System.out.println();
```

```
    }
}
```

Output:

```
Enter "firstname lastname age" on each line
("Exit" as firstname exits).

Mary Jones 22
   First name: Mary
   Last name: Jones
   Age: 22

Mike Smith 24
   First name: Mike
   Last name: Smith
   Age: 24

Exit
   Exiting.
```

Using `StringWriter` and `PrintWriter` to create String streams

- An **output string stream** is created using `StringWriter` and `PrintWriter` classes.
  - To use these classes, import `java.io.StringWriter` and `java.io.PrintWriter`.
- `StringWriter` provides a character stream that allows a programmer to output characters.
- `PrintWriter` is a wrapper class that augments character streams, such as `StringWriter`, with `print()` and `println()` methods that allow a programmer to output various data types (e.g., `int`, `double`, `String`, etc.) to the underlying character stream in a similar manner to System.out.
- The StringWriter's `toString()` method can be used to copy the buffer to a String.

Creating a String Using Streams

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

// Basic character stream for fullname
StringWriter fullnameCharStream = new StringWriter();
// Augments character stream (fullname) with print()
PrintWriter fullnameOSS = new PrintWriter(fullnameCharStream);
// Basic character stream for age
StringWriter ageCharStream = new StringWriter();
// Augments character stream (age) with print()
PrintWriter ageOSS = new PrintWriter(ageCharStream);

String firstName;      // First name
String lastName;       // Last name
String fullName;       // Full name (first and last)
String ageStr;         // Age (string)
int userAge;           // Age

// Prompt user for input
System.out.print("Enter \"firstname lastname age\": \n   ");
firstName = scnr.next();
lastName = scnr.next();
userAge = scnr.nextInt();

// Writes formatted string to buffer, copies from underlying char buffer
fullnameOSS.print(lastName + ", " + firstName);
fullName = fullnameCharStream.toString();

// Output parsed input
System.out.println("\n   Full name: " + fullName);

// Writes int age as characters to buffer
ageOSS.print(userAge);

// Appends (minor) to object if less than 21, then
// copies buffer into string
if (userAge < 21) {
   ageOSS.print(" (minor)");
}

ageStr = ageCharStream.toString();
```

```
// Output string
System.out.println("   Age: " + ageStr);
```

Output:

```
Enter "firstname lastname age":
  Mary Jones 22

  Full name: Jones, Mary
  Age: 22

...

Enter "firstname lastname age":
  Sally Smith 14

  Full name: Smith, Sally
  Age: 14 (minor)
```

Conclusion

- The Scanner class can accept different input sources, including Strings and standard input.
- String streams can process user input line-by-line and are useful when a programmer wishes to read input data from a string rather than from the keyboard.
- StringWriter and `PrintWriter` classes can be used to create output string streams to output various data types to the underlying character stream in a similar manner to `System.out`.

File Input

Opening and Reading from a File

- To read file input, a programmer can create a new input stream that comes from a file.
- `FileInputStream` is used to create a file input stream that opens the file denoted by a String variable, str, for reading.

- A common error is a mismatch between the variable data type and the file data.
- To read varying amounts of data in a file, a program can use a loop that reads until valid data is unavailable or the end of the file has been reached.
- The `hasNextInt()` method returns true if an integer is available for reading.

Example Input From a File

File Contents:

```
5
10
```

Program:

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadNums {
   public static void main (String[] args) throws IOException {
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      int fileNum1;                        // Data value from file
      int fileNum2;                        // Data value from file

      // Try to open file
      System.out.println("Opening file numFile.txt.");
      fileByteStream = new FileInputStream("numFile.txt");
      inFS = new Scanner(fileByteStream);

      // File is open and valid if we got this far
      // (otherwise exception thrown)
      // numFile.txt should contain two integers, else problems
      System.out.println("Reading two integers.");
      fileNum1 = inFS.nextInt();
      fileNum2 = inFS.nextInt();

      // Output values read from file
      System.out.println("num1: " + fileNum1);
      System.out.println("num2: " + fileNum2);
      System.out.println("num1+num2: " + (fileNum1 + fileNum2));

      // Done with file, so try to close it
      System.out.println("Closing file numFile.txt.");
      // close() may throw IOException if fails
      fileByteStream.close();
   }
}
```

Failure to open file:

```
Opening file numFile.txt.
Exception in thread "main"
java.io.FileNotFoundException:
numFile.txt
(No such file or directory)
...
```

Successfully open file:

```
Opening file numFile.txt.
Reading two integers.
num1: 5
num2: 10
num1 + num2: 15
Closing file numFile.txt.
```

Reading Until the End of the File

- A program can read varying amounts of data in a file by using a loop that reads until valid data is unavailable or the end of the file has been reached.
- The Scanner class offers multiple `hasNext()` methods for various data types such as `int`, `double`, `String`, etc.

Example: Reading a Varying Amount of Data from a File

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;

public class FileReadVaryingAmount {
   public static void main(String[] args) throws IOException {
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      int fileNum;                           // Data value from file

      // Try to open file
      System.out.println("Opening file myfile.txt.");
      fileByteStream = new FileInputStream("myfile.txt");
      inFS = new Scanner(fileByteStream);

      // File is open and valid if we got this far (otherwise exception
thrown)
      System.out.println("Reading and printing numbers.");

      while (inFS.hasNextInt()) {
         fileNum = inFS.nextInt();
         System.out.println("num: " + fileNum);
      }

      // Done with file, so try to close it
      System.out.println("Closing file myfile.txt.");
      fileByteStream.close(); // close() may throw IOException if fails
   }
}
```

Example: Counting Instances of a Specific Word

- The following program uses both the `hasNext()` and `next()` methods to determine how many times a user entered word (type String) appears in a file.
- The number of words in the file is unknown, so the program extracts words until no more words exist in the file.

```java
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.IOException;

public class CountingWords {
   public static void main(String[] args) throws IOException {
      Scanner scnr = new Scanner(System.in);
      FileInputStream fileByteStream = null; // File input stream
      Scanner inFS = null;                   // Scanner object
      String userWord;
      int wordFreq = 0;
      String currWord;

      // Try to open file
      System.out.println("Opening file wordFile.txt.");
      fileByteStream = new FileInputStream("wordFile.txt");
      inFS = new Scanner(fileByteStream);

      // Word to be found
      System.out.print("Enter a word: ");
      userWord = scnr.next();

      while (inFS.hasNext()) {
         currWord = inFS.next();
         if(currWord.equals(userWord)) {
            ++wordFreq;
         }
      }

      System.out.println(userWord + " appears in the file " +
                         wordFreq + " times.");

      // Done with file, so try to close it
      fileByteStream.close(); // close() may throw IOException if fails
   }
}
```

Input file:

```
twenty
associable
twenty
unredacted
associable
folksay
twenty
```

Output:

```
Opening file wordFile.txt.
Enter a word: twenty
twenty appears in the file 3 times.
```

Exploring Further

- Oracle's Java FileInputStream class specification
- Oracle's Java IOException class specification

Conclusion

- File input can be useful when a program should get input from a file rather than from a user typing on a keyboard.
- A programmer can create a new input stream that comes from a file, rather than the predefined input stream System.in that comes from the standard input (keyboard).
- The FileInputStream class is used to create a file input stream and opens the file denoted by a String variable or String literal for reading.
- A program can read varying amounts of data in a file by using a loop that reads until valid data is unavailable or the end of the file has been reached.
- The `hasNextInt()` method returns true if an integer is available for reading, and the Scanner class offers multiple `hasNext()` methods for various data types such as int, double, String, etc.
- While using the `hasNext()` method, the program extracts data from the file until no more data exists in the file.

Overall, file input is a useful tool for programs that need to read data from a file. By utilizing the FileInputStream and Scanner classes, a program can read

data in various data types, including integers, doubles, and Strings. Using loops and `hasNext()` methods, a program can read varying amounts of data from a file and extract data from the file until there is no more data available.

File Output

Introduction

- `FileOutputStream` is a class that allows writing to a file, and it inherits from `OutputStream`.
- `PrintWriter` is commonly used to write strings and other data types to a file.

Basic steps for opening and writing a file

Action	Sample code
Open the file helloWorld.txt for writing	`FileOutputStream fileStream = new FileOutputStream("helloWorld.txt");`
Create a PrintWriter to write to the file	`PrintWriter outFS = new PrintWriter(fileStream);`
Write the string "Hello World!" to the file	`outFS.println("Hello World!");`
Close the file after writing all desired data	`outFS.close();`

Note: The above table is written in Markdown format. Markdown is a lightweight markup language that allows you to format text using plain text syntax. Markdown is commonly used in documentation, README files, and online forums.

Example: Writing a text file

- `FileOutputStream` constructor throws an exception if the file cannot be opened for writing.
- If no exception is thrown, the file is created and initially empty.
- Data can be written to the file, then the file is closed.

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

public class TextFileWriteSample {
  public static void main(String[] args) throws IOException {
     FileOutputStream fileStream = null;
     PrintWriter outFS = null;

     // Try to open file
     fileStream = new FileOutputStream("myoutfile.txt");
     outFS = new PrintWriter(fileStream);

     // Arriving here implies that the file can now be written
     // to, otherwise an exception would have been thrown.
     outFS.println("Hello");
     outFS.println("1 2 3");

     // Done with file, so try to close
     // Note that close() may throw an IOException on failure
     outFS.close();
  }
}

/* Contents of
 * myoutfile.txt:
 * Hello
 * 1 2 3
 */
```

Example: Writing a simple HTML file

- An HTML file can be written similar to a text file using `PrintWriter`.

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

public class HTMLFileWriteSample {
  static void writeHTMLFile(PrintWriter printer, String innerHTML) {
     printer.println("<!DOCTYPE html>");
     printer.println("<html>");
     printer.println("  <body>");
     printer.println("    <p>" + innerHTML + "</p>");
     printer.println("  </body>");
     printer.println("</html>");
  }
  public static void main(String[] args) throws IOException {
     // Open an output file stream and create a PrintWriter
     FileOutputStream fileStream = new FileOutputStream("simple.html");
     PrintWriter filePrinter = new PrintWriter(fileStream);

     // Write the HTML file, then close filePrinter
     writeHTMLFile(filePrinter, "Hello <b>HTML</b> world!");
     filePrinter.close();

     // Use the same function, writeHTMLFile, to write to the console
     PrintWriter systemOutPrinter = new PrintWriter(System.out);
     writeHTMLFile(systemOutPrinter, "Hello <b>HTML</b> world!");
     systemOutPrinter.close();
  }
}

/* simple.html file contents:
 * <!DOCTYPE html>
 * <html>
 * <body>
 * <p>Hello <b>HTML</b> world!</p>
 * </body>
 * </html> */

/* Console:
 * <!DOCTYPE html>
 * <html>
 * <body>
 * <p>Hello <b>HTML</b> world!</p>
```

```
 * </body>
 * </html> */
```

Conclusion

- `FileOutputStream` and `PrintWriter` classes are used to write to a file in Java.
- The `FileOutputStream` constructor opens a file for writing, while `PrintWriter` is commonly used to write strings and other data types to a file.
- Writing a text file and an HTML file examples are provided.