STUDENT OUTLINE

Lesson 4 - Simple I/O

INTRODUCTION: Input and output of program data are usually referred to as I/O. There are many different ways that a Java program can perform I/O (input and output). In this lesson, we present some very simple ways to handle simple text input typed in at the keyboard.

The key topics for this lesson are:

A. Reading Input with the `ConsoleIO` Class
B. Multiple Line Stream Output Expressions
C. Formatting Output
D. `String` Objects
E. `String` Input

VOCABULARY:

`ConsoleIO`	`Format`
`String`	CONCATENATE

DISCUSSION:

A. Reading Input with the `ConsoleIO` Class

1. Some of the programs from the preceding lessons have not been written in the most efficient manner. To change any of the data values in the programs, it would be necessary to change the variable initializations, recompile the program, and run it again. It would be more practical if the program could ask for new values for each type of data and then compute the desired output.

2. However, accepting user input in Java has some technical complexities. Throughout this curriculum guide, we will use a special class, called `ConsoleIO`, to make processing input easier and less tedious.

3. Just as the System class provides `System.out` for output, there is an object for input, `System.in`. Unfortunately, Java's `System.in` object does not directly support convenient methods for reading numbers and strings.

4. To use the `ConsoleIO` class in a program, you first need to import from the `chn.util` package with the statement

```
import chn.util.*;
```

5. To use a `ConsoleIO` method, you need to construct a `ConsoleIO` object.

```
ConsoleIO console = new ConsoleIO();
```

Next, call one of `ConsoleIO` methods

See Handout H.A.4.1, *ConsoleIO Class* for a complete list of input features.

```
int n  = console.readInt();
double d  = console.readDouble();
boolean done  = console.readBoolean();
String token = console.readToken();
String line = console.readLine();
```

6. Here are some example statements:

```
int num1;
double bigNum;
String line;

num1 = console.readInt( );
bigNum = console.readDouble( );
line = console.readLine( );
```

When the statement `num1 = console.readInt()` is encountered, execution of the program is suspended until an appropriate value is entered on the keyboard.

7. Any whitespace (spaces, tabs, newline) will separate input values. When reading values, white space keystrokes are ignored. If it is desirable to input both whitespace and non-whitespace characters, the method `readLine()` is required.

8. The `readToken()` method reads and returns the next token from the current line. A token is a `String` of characters separated by the specified delimiters (whitespace). For example when the following code fragment is executed

```
String input = console.readToken();
System.out.print(input);
```

when given an input line of

```
twenty-three is my favorite prime number
```

would output

```
twenty-three
```

since this is the first string of characters read before a whitespace value (space) is encountered.

9. When requesting data from the user via the keyboard, it is good programming practice to provide a prompt. An unintroduced input statement leaves the user hanging without a clue of what the program wants. For example:

```
System.out.print("Enter an integer -->  ");
number = console.readInt();
```

B. Multiple Line Output Expressions

1. We have already used examples of multiple output statements such as:

```
System.out.println("The value of sum = " + sum);
```

2. There will be occasions when the length of an output statement exceeds one line of code. This can be broken up several different ways.

```
System.out.println("The sum of " + num1 + " and " + num2 +
                   " = " + (num1 + num2));
```

or

```
System.out.print("The sum of " + num1 + " and " + num2);
System.out.println( " = " + (num1 + num2));
```

3. You cannot break up a `String` constant and wrap it around a line. This is not valid:

```
System.out.print("A long string constant must be broken
up into two separate quotes.  This will NOT work.");
```

However, this will work:

```
System.out.print("A long string constant must be broken up"
               + "into two separate quotes. This will work.");
```

C. Formatting Output

1. Formatting output in Java has some technical complexities. Throughout this curriculum guide, we will use a special class, called `Format`, to format numerical and textual values for a properly aligned output display.

2. The `Format` methods are available by including the import directive,

```
import apcslib.*;
```

at the top of the source code.

3. The basic idea of formatted output is to allocate the same amount of space for the output values and align the values within the allocated space. The

space occupied by an output value is referred to as the *field* and the number of character allocated to a field is its *field width*.

4. To format an integer you would use the following expressions:

See Handout H.A.4.2, *Format Class* for a complete list of formatting features.

```
Format.left(int_expression, fieldWidth);
Format.center(int_expression, fieldWidth);
Format.right(int_expression, fieldWidth);
```

where *int_expression* is an arithmetic expression whose value is an **int** and *fieldWidth* designates the field width. Example:

```
int i1 = 1234;
int i2 = 567;
int i3 = 891011;

System.out.println(Format.left(i1, 15) + "left");
System.out.println(Format.center(i2, 15) + "center");
System.out.println(Format.right(i3, 15) + "right");
```

Run output:

```
1234           left
      567      center
         891011right
```

5. The same type of format statements can be used for a `String` value.

```
Format.left(String_expression, fieldWidth);
Format.center(String_expression, fieldWidth);
Format.right(String_expression, fieldWidth);
```

where *String_expression* is an expression that evaluates to a `String` and *fieldWidth* designates the field width. Example:

```
String s1 = "left";
String s2 = "center";
String s3 = "right";

System.out.println(Format.left(s1, 15) + "String");
System.out.println(Format.center(s2, 15) + "String");
System.out.println(Format.right(s3, 15) + "String");
```

Run output:

```
left           String
     center    String
          rightString
```

6. To format a real number (**float** or **double**) we need additional arguments to specify the decimal places:

```
Format.left(real_expression, fieldWidth, decimalPlaces);
Format.center(real_expression, fieldWidth, decimalPlaces);
Format.right(real_expression, fieldWidth, decimalPlaces);
```

where *real_expression* is an arithmetic expression whose value is either a **float** or a **double** and *decimalPlaces* designates the number of digits shown to the right of the decimal point. The value for *fieldWidth* must be at least as large as the value of *decimalPlaces* plus two. Example:

```
double d1 = -123.4e-5;
double d2 = 678.9;
double d3 = 12345.6789;

System.out.println(Format.left(d1, 15, 6) + "left");
System.out.println(Format.center(d2, 15, 3) + "center");
System.out.println(Format.right(d3, 15, 2) + "right");
```

Run output:

```
-0.001234      left
    678.900    center
       12345.68right
```

D. String Objects

1. Next to numbers, *strings* are the most important data type that most programs use. A string is a sequence of characters such as "Hello". In Java, strings are enclosed in quotation marks, which are not themselves part of the string.

2. `String` objects can be constructed in two ways:

```
String name = "Bob Binary";
String anotherName = new String("Betty Binary");
```

Due to the usefulness and frequency of use of strings, a shorter version without the keyword **new**, was developed as a short-cut way of creating a `String` object. This creates a `String` object containing the characters between quote marks, just as before. A `String` created in this method is called a *`String` literal*. Only `String`s have a short-cut like this. All other objects are constructed by using the **new** operator.

3. Assignment can be used to place a different `string` into the variable.

```
name = "Boris";
anotherName = new String("Bessy");
```

4. The number of characters in a string is called the *length* of the string. For example, the length of "Hello World!" is 12. You can compute the length of a `String` with the `length` method.

```
int n = name.length();
```

Unlike numbers, strings are objects. Therefore, you can call methods on strings. In the above example, the `length` of the `String` object `name` is computed with the method call `name.length()`.

5. A `String` of length zero, containing no characters, is called the *empty string* and is written as `""`. For example:

```
String empty = "";
```

6. Programmers often make one `String` object from two strings with the `+` operator, which *concatenates* (connects) two or more strings into one string. Concatenation and these string messages are illustrated below.

```
public class DemoStringMethods
{
  public static void main(String[] args)
  {
    String a = new String("Any old");
    String b = " String";
    String aString = a + b; // aString is "Any old String"

    // Show string a
    System.out.println("a: " + a);

    // Show string b
    System.out.println("b: " + b);

    // Show the result of concatenating a and b
    System.out.println("a + b: " + aString);

    // Show the number of characters in the string
    System.out.println("length: " + aString.length());
  }
}
```

Run output:

```
a: Any old
b:  String
a + b: Any old String
length: 14
```

7. Notice that using strings in the context of input and output statements is identical to using other data types.

8. The `String` class will be covered in depth in a later lesson. For now you will use strings for simple input and output in programs.

E. `String` Input

1. The `ConsoleIO` class has two methods for reading textual input from the keyboard.

2. The `readToken` message returns a reference to a `String` object that has from zero to many characters typed by the user at the keyboard. A *token* is a sequence of printable characters separated from the next word by *white space*. White space is defined as blank spaces, tabs, or newline characters in the input stream. White space separates **`int`**s and **`double`**s on input. White space also separates words on input. When input from the keyboard, `readToken` stops adding text to the `String` object when the first white space is encountered on the input stream from the user.

3. A `readLine` message returns a reference to a `String` object that contains from zero to many characters entered by the user. With `readLine`, the `String` object may contain blank spaces and tabs. The newline marker is not included. It is discarded from the input stream.

4. Input from these string messages is illustrated below.

```
import chn.util.*;

public class DemoStringInput
{
  public static void main(String[] args)
  {
    ConsoleIO keyboard = new ConsoleIO( );
    String word1, word2, anotherLine;

    // ask for input from the keyboard
    System.out.print("Enter a line: ");

    // grab the first "word"
    word1 = keyboard.readToken();

    // grab the second "word"
    word2 = keyboard.readToken();
```

```
    // ask for input from the keyboard
    System.out.print("Enter another line: ");

    // discard any remaining input from previous line
    // and read the next line of input
    anotherLine = keyboard.readLine();

    // output the strings
    System.out.println("word1 = " + word1);
    System.out.println("word2 = " + word2);
    System.out.println("anotherLine = " + anotherLine);
  }
}
```

Run output:

```
Enter a line: Hello World! This will be discarded.
Enter another line: This line includes whitespace.
word1 = Hello
word2 = World!
anotherLine = This line includes whitespace.
```

SUMMARY/ REVIEW: These two classes, `ConsoleIO` and `Format`, will be used in many programs. The labs in this lesson will provide an opportunity to practice using simple I/O and formatting.

ASSIGNMENT: Lab Exercise, L.A.4.1, *Change*
Lab Exercise, L.A.4.2, *CarRental*

LAB EXERCISE

Change

Background:

Some cash register systems use change machines that automatically dispense coins. This lab will investigate the problem solving and programming behind such machinery. You should use integer mathematics to solve this problem.

You will need to extract the amount of cents from dollar amounts expressed in real numbers. This will require using the type cast operator and dealing with the approximate nature of real number storage. Here is an important example:

```
double purchaseAmount, cashPaid, temp;
int change;

... data input stuff                                Example Values:

    temp = cashPaid - purchaseAmount;               8.06 = 30.00 - 21.94
    temp = temp - (int)temp;                        0.06 = 8.06 - 8
    change = (int) (temp * 100);                    6 = (int)(0.06 * 100)
```

However, when the above example was run on a computer, the answer of `5` was given. Because real numbers are stored as approximations, the value of 0.06 was actually something like `0.05999998`. Because the type conversion of `(int)(0.05999998 * 100)` will truncate the fractional part, the result is erroneously `5`. We need to make a minor adjustment to the second line:

```
double purchaseAmount, cashPaid, temp;
int change;

... data input stuff                                Example Values:

    temp = cashPaid - purchaseAmount;               8.06 = 30.00 - 21.94
    temp = temp - (int)temp + 0.00001;              0.06000998 = 8.05999998 - 8 + 0.00001
    change = (int)(temp * 100);                     6 = (int)(0.06000998 * 100)
```

Assignment:

1. Write a program that does the following:

    a. Prompts the user for the following information.

        Amount of purchase
        Amount of cash tendered

b. Calculates and prints out the coinage necessary to make correct change. Do not solve the amount of bills required in the change amount.

2. Sample run output:

```
Amount of purchase = 23.06

Cash tendered = 30.00

Amount of coins needed:

   94 cents =

     3 quarters
     1 dime
     1 nickel
     4 pennies
```

3. Include appropriate documentation in your program.

4. You are encouraged, but not required, to use the formatting tools in `apcslib`.

5. Do not worry about singular versus plural endings, i.e. quarter/quarters.

Instructions:

1. Complete and run the program and verify the calculations. Use the values given on page one.

2. When it comes time to save the run output, cut and paste the run output to your source code, but do not include user prompts. Only copy the relevant answer section of the run output window.

LAB EXERCISE

Car Rental

Background:

When you rent a car from an agency, the key ring has several pieces of information: license plate, make and year of car, and usually a special code. This code could be used for some data processing within the company's computers. This lab will practice determining that special car rental code from the license plate.

Assignment:

1. The following sequence of steps will be used to convert a license plate into a car rental code.

   a. A license plate consists of 3 letters followed by a 3 digit integer value.

   b. Type in the license plate information as 3 characters followed by a single integer value. For example, CPR 607.

   c. Add up the ASCII values of the 3 letters, `67 + 80 + 82 = 229`.

   d. Add the sum of the letters to the single integer value. For example, `229 + 607 = 836`.

   e. Take this sum (`836`) and determine the integer remainder after dividing by `26`:
   `836 % 26 = 4`.

   f. Determine the 4th letter in the alphabet after the letter '`A`': 4th letter after '`A`' = E.

   g. Combine the letter and the sum, the car id number for license plate
   `CPR 607 = E836`.

2. You may assume that all sample data will be in the format of 3 alphabet characters, then a space, followed by a 3 digit integer.

Instructions:

1. Prompt the user for the make and model of the car. Use strings to create this part.

2. Prompt the user for the license plate.

3. Print the run output in this format.

```
Make = Chevrolet
Model = Suburban
```

```
CPR 607 = E836
```

4. Solve the following run outputs:

   RJK 492
   SPT 309

   The input values for the make and model strings are your choice.

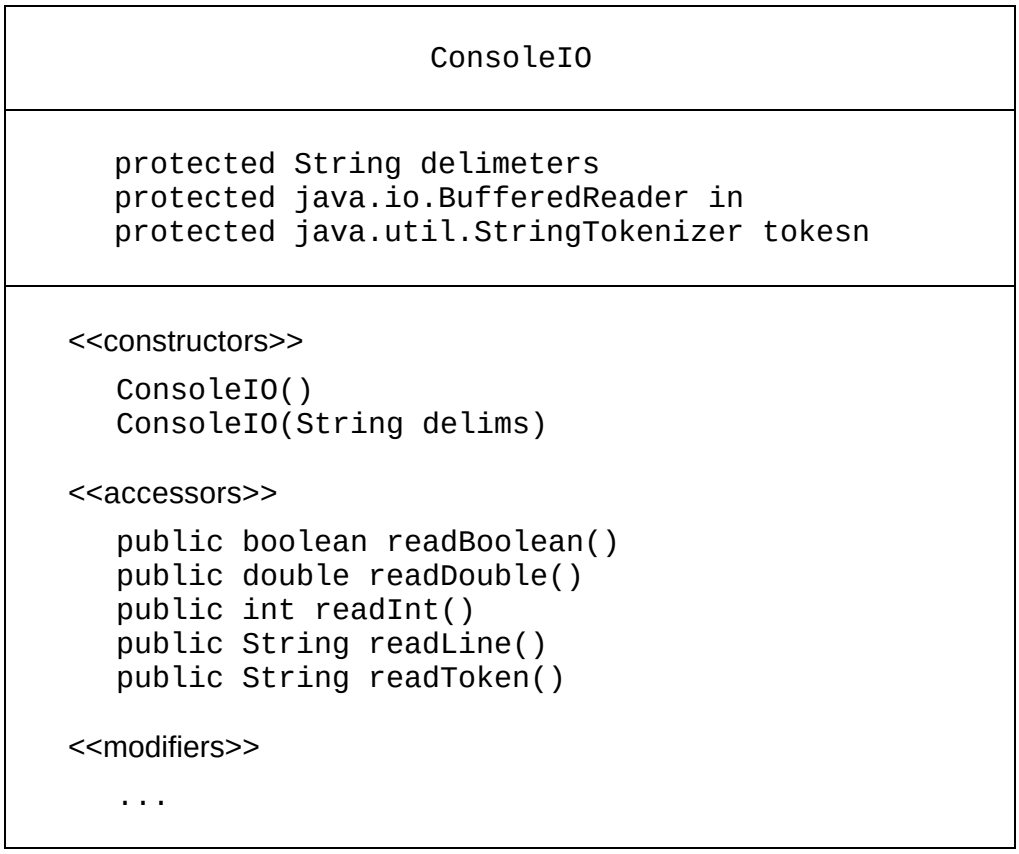
5. Turn in your source code and two run outputs.

ConsoleIO CLASS SPECIFICATIONS

These classes are not part of Java but are available through the library named `chn.util`. You must have the file `chnutil.jar` in the appropriate directory where Java can access it. To have these classes available in your program, use this command:

```
import chn.util.*;
```

Other features of `chn.util` will be covered in later lessons.



Constructor Methods

public ConsoleIO()

postcondition

Constructs the `ConsoleIO` object with default delimiters, space, tab, formfeed.

public ConsoleIO (String delims)

postcondition

Constructs the `ConsoleIO` object with given delimiters.

ConsoleIO CLASS SPECIFICATIONS

Accessor Methods

`public boolean readBoolean()`

Reads and returns the next **`boolean`** from the current line. If the current input line has no more unread tokens, reads the first token from the first non-empty line entered. A valid **`boolean`** is any token. The value true is returned if the token is "true", with upper or lower case letters, otherwise false is returned.

postcondition
returns the next **`boolean`** from input.

`public double readDouble()`

Reads and returns the next double from the current line. If the current input line has no more unread tokens, reads the first token from the first non-empty line entered. If the next token is not the correct format for a double, then a error message is printed and the program terminates.

postcondition
returns the next **`double`** from input.

`public int readInt()`

Reads and returns the next integer from the current line. If the current input line has no more unread tokens, reads the first token from the first non-empty line entered. If the next token is not the correct format for an integer, then a error message is printed and the program terminates.

postcondition
returns the next **`integer`** from input.

`public String readLine()`

Flushes any remaining tokens on the current input line, then reads the next full line input as a `String` and returns it.

postcondition
returns the next line read as a `String`.

`public String readToken()`

Reads and returns the next token from the current line. A token is a `String` of characters separated by the specified delimiters. If the current input line has no more unread tokens, reads the first token from the first non-empty line entered.

postcondition
returns the next `String` token from input, as determined by delimiters

ConsoleIO CLASS SPECIFICATIONS

.

Format CLASS SPECIFICATIONS

These classes are not part of Java but are available through the library named `apcslib`. You must have the file `apcslib.jar` in the appropriate directory where Java can access it. To have these classes available in your program, use this command:

```
import apcslib.*;
```

Other features of `apcslib` will be covered in later lessons.

Format
<<constructors>> ... <<accessors>> public static String center(double d, int fWidth, int dPlaces) public static String center(long l, int fWidth) public static String center(String s, int fWidth) public static String left(double d, int fWidth, int dPlaces) public static String left(long l, int fWidth) public static String left(String s, int fWidth) public static String right(double d, int fWidth, int dPlaces) public static String right(long l, int fWidth) public static String right(String s, int fWidth) <<modifiers>> ...

Accessor Methods

`public static String center(double d, int fWidth, int dPlaces)`

postcondition

returns a String representing the **`double`** `d` centered in a field of `fWidth` rounded to `dPlaces`.

`public static String center(long l, int fWidth)`

postcondition

returns a String representing the **`long`** `l` centered in a field of `fWidth`.

```
public static String center(String s, int fWidth)
```

postcondition
returns a String representing the `String s` centered in a field of `fWidth`.

```
public static String left(double d, int fWidth, int dPlaces)
```

postcondition
returns a String representing the **`double`** `d` adjusted to the left in a field of `fWidth` rounded to `dPlaces`.

```
public static String left long l, int fWidth)
```

postcondition
returns a String representing the **`long`** `l` adjusted to the left in a field of `fWidth`.

```
public static String left(String s, int fWidth)
```

postcondition
returns a String representing the `String s` adjusted to the left in a field of `fWidth`.

```
public static String right(double d, int fWidth, int dPlaces)
```

postcondition
returns a String representing the **`double`** `d` adjusted to the right in a field of `fWidth` rounded to `dPlaces`.

```
public static String right(long l, int fWidth)
```

postcondition
returns a String representing the **`long`** `l` adjusted to the right in a field of `fWidth`.

```
public static String right(String s, int fWidth)
```

postcondition
returns a String representing the `String s` adjusted to the right in a field of `fWidth`.