CS3231
Object Oriented Programming I

Name: _____ ( ) Date: _____

Chapter 3: Input, Output and Exception Handling

Lesson Objectives

After completing the lesson, the student will be able to

- *understand streams in Java*
- *display simple output to the screen using console output (System.out)*
- *obtain input from the console using the Scanner class with console input (System.in)*
- *format console output using format specifers in System.out.printf()*
- *get an overview of exceptions and exception handling*
- *write a try-catch block to handle exceptions*

3.1 Streams

In Java, a source of input data is called an **input stream** and the output data is called an **output stream.**

Figure 4.1 Input and Output Streams

In the picture, each "O" represents a piece of data waiting in line to be input or leaving as output. The input stream is like a pipeline which the program inputs one at a time, in order. The output stream is another pipeline to display program output. Often a program will read several data and then combine them somehow to produce one output value. For example, the input data might be a list of numbers, the output data might be their sum.

There are three standard I/O streams:

- `System.in` — the input stream.
- `System.out` — the output stream for normal results.
- `System.err` — the output stream for error messages.

Normally `System.in` is connected to the keyboard and the data are characters. `System.out` and `System.err` both are connected to the monitor, and also contain character data.

3.2 Console Output

In Chapter 2, `System.out.println()` was used to display simple console output. `System.out` is a standard output object that exists in the `java.lang` package which is implicitly imported in all Java program. `println` is a method invoked by that object to display information pass to it.

3.2.1 System.out.println vs System.out.print

`System.out.println()` instructs the computer to perform an action—namely, to display the characters contained between the double quotation marks (the quotation marks themselves are not displayed). Together, the quotation marks and the characters between them are a string also known as a character string or a string literal. White-space characters in strings are not ignored by the compiler.

You can output one line to the screen using `System.out.println`. **The items that are output can be quoted strings, variables, numbers, or almost any object you can define in Java**. **To output more than one item, place a plus sign (+ concatenation operator) between the items.**

```
System.out.println(Item_1 + Item_2 + ... + Last_Item);
```

A string cannot cross lines in the source code. Thus, the following statement would result in a compile error:

```
System.out.println("Introduction to Java Programming,
by NUS High School");
```

To fix the error, break the string into separate substrings, and use the concatenation operator (+) to combine them:

```
System.out.println("Introduction to Java Programming, " +
"by NUS High School");
```

Every invocation of `System.out.println` ends a line of output. For example,

```
System.out.println("A wet bird");
System.out.println("never flies at night.");
```

These two statements cause the following output to appear on the screen:

```
A wet bird
never flies at night.
```

If you want the output from two or more output statements to place all their output on a single line, then use **`print`** instead of `println`. See PROGRAM 3-1.

PROGRAM 3-1

```
1  public class ConsoleOutput {
2   public static void main(String[] args) {
3     System.out.print("A ");
4     System.out.print("wet ");
5     System.out.println("bird");
6     System.out.println("never flies at night.");
7  }
```

PROGRAM OUTPUT

```
A wet bird
never flies at night.
```

The only difference between `System.out.println` and `System.out.print` is that with `println`, the next output goes on a new line, whereas with `print`, the next output is placed on the same line.

Alternatively, you can also use the line feed escape sequence (`\n`) to print to a new line.
`System.out.println(SomeThing);` is equivalent to

```
System.out.print(SomeThing + "\n");
```

3.2.2 System.out.printf

If you have a variable of type double that stores some amount of money, you would like your programs to output the amount in a nice format. However, if you just use `System.out.println`, you are likely to get output that looks like the following:

```
Your cost, including tax, is $19.98327634144
```

If you would like the output to look like this:

```
Your cost, including tax, is $19.98
```

To obtain this nicer form of output, you need some formatting tools. **Java includes a method named `System.out.printf` that can be used to give output in a specific format.** This method is used the same way as the method `print` but allows you to add formatting instructions that specify such things as the number of digits to include after a decimal point.

PROGRAM 3-2

```
1  public class ConsoleOutput {
2   public static void main(String[] args) {
3     double price = 19.8;
4     System.out.print("$");
5     System.out.printf("%6.2f", price);
6     System.out.println(" each"); }
7  }
```

PROGRAM OUTPUT

```
$ 19.80 each
```

PROGRAM 3-2 at line 5 outputs the string `" 19.80"` (one blank followed by 19.80), which is the value of the variable price written in the format `%6.2f`. The first argument to `printf` is a string known as the **format specifier**, and the second argument is the number or other value to be output in that format.

The format specifier `%6.2f` outputs a floating-point number in a field (number of spaces) of width 6 (room for six characters) and a precision of two digits after the decimal point. So, 19.8 is expressed as “19.80” in a field of width 6. Because "19.80" has only five characters, a blank character is added to obtain the six character string " 19.80". Any extra blank space is added to the front (left-hand end) of the value output. That explains the 6.2 in the format specifier `%6.2f`. The `f` means the output is a floating-point number, that is, a number with a decimal point.

The first argument to `printf` can include text as well as a format specifier. Thus, Line 4 to 6 can be combined into a single statement as follows:

```
System.out.printf("$%6.2f each", price);
```

The text before and after the format specifier `%6.2f` is output along with the formatted number. The character % signals the end of text to output and the start of the format specifier. The end of a format specifier is indicated by a conversion character (`f` in our example). Table 4.1 shows other types of format specifiers.

Conversion Character	Type of Output	Example
`d`	Decimal (ordinary) integer.	`%5d, %d`
`f`	Fixed-point (everyday notation) floating-point	`%6.2f, %f`
`e`	E-notation floating point	`%8.3e, %e`
`g`	General floating point. (Java decides whether to use E-notation or not.)	`%8.3g, %g`
`s`	String	`%12s, %s`
`c`	Character	`%2c, %c`
`n`	Denotes a line break. This does not correspond to an output argument. It is approximately equivalent to \n.	`%n`

Table 3.1 Types of Format Specifiers

In summary, a number of the form **N.M** in a format specifier specifies a field width of **N spaces** with **M digits** after the decimal point**. If one number N is given only, it specifies a field width**;

if there is a decimal point in the output, then the number of digits after the decimal point is determined by Java. When the value output does not fill the field width specified, then blanks are added in front of the value output. The output is then said to be **right justified. If you add a hyphen (-) after the %, then any extra blank space is placed after the value output and the output is said to be left justified**. For example, `%8.2f` is right justified and `%-8.2f` is left justified.

`System.out.printf` can have any number of arguments. **The first argument is always a format string for the remaining arguments.** All the arguments except the first are values to be output to the screen and these values are output in the formats specified by the format string. The format string can contain text as well as format specifiers, and this text is output along with the values. See PROGRAM 3-3.

PROGRAM 3-3

```
1  public class ConsoleOutputMultipleArguments {
2   public static void main(String[] args) {
3     int age = 15;
4     double amount = 253.48;
5     String name = "Jack";
6     char letter = 'Z';
7     System.out.printf("%s is %d years old. He has $%6.2f in savings "
8     + "and he likes the letter %c.", name, age, amount, letter);
9    }
10 }
```

PROGRAM OUTPUT

```
Jack is 15 years old. He has $253.48 in savings and he likes the letter Z.
```

PROGRAM 3-4

```
1  public class ConsoleOutputFormatting{
2   public static void main(String[] args) {
3     String aString = "abc";
4     System.out.println("String output:");
5     System.out.println("*1234567890");
6     System.out.printf("*%s* %n", aString);
7     System.out.printf("*%4s* %n", aString);
8     System.out.printf("*%2s* %n", aString);
9     System.out.println();
10    char oneCharacter = 'Z';
11    System.out.println("Character output:");
12    System.out.println("*1234567890");
13    System.out.printf("*%c* %n", oneCharacter);
14    System.out.printf("*%4c* %n", oneCharacter);
15    System.out.println();
16    int number = 256;
17    System.out.println("Integer output:");
18    System.out.printf("*%d* %n", number);
19    System.out.printf("*%2d* %n", number);
20    System.out.printf("*%6d* %n", number);
21    System.out.printf("*%-6d* %n", number);
22    System.out.println();
23    double d = 12345.123456789;
24    System.out.println("Floating-point output:");
25    System.out.println("*1234567890");
26    System.out.printf("*%f* %n", d);
27    System.out.printf("*%.4f* %n", d);
28    System.out.printf("*%.2f* %n", d);
29    System.out.printf("*%12.4f* %n", d);
30    System.out.printf("*%e* %n", d);
31    System.out.printf("*%12.5e* %n", d);
32   }
33 }
```

PROGRAM OUTPUT

```
String output:
*1234567890
*abc*
* abc*
*abc*

Character output:
*1234567890
*Z*
*   Z*

Integer output:
*256*
*256*
*   256*
*256   *

Floating-point output:
*1234567890
*12345.123457*
*12345.1235*
*12345.12*
*  12345.1235*
*1.234512e+04*
* 1.23451e+04*
```

PROGRAM 3-4 illustrates the formatting capabilities of `printf` using format specifiers. The value is always output. If the specified field width is too small, extra space is taken as shown in line 8.

3.3 Console Input

You can make your programs more flexible if you ask the program user for inputs rather than using fixed values. Reading input from the console enables the program to accept input from the user. In PROGRAM 2-1, the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. However, this approach is neither convenient nor practical.

3.3.1 Scanner Class

The Scanner class is located in the `java.util` package. **A package is simply a library of classes**. You can use the Scanner class to create an object to read input from `System.in` (keyboard). **Creating a Scanner object is like setting up a pipeline for input to flow into your program.** To make the Scanner class available to your program you need to use the `import` declaration to help the compiler locate the class. **Import declarations must appear before the first class declaration in the file.** Placing an import declaration inside or after a class declaration is a syntax error.

The following statement creates an object which handles the reading of input.

```
Scanner input = new Scanner(System.in);  // create an input object
```

The syntax `Scanner input` declares that `input` is a variable whose type is Scanner. The syntax `new Scanner(System.in)` creates an object of the Scanner type. The whole statement creates a Scanner object and **assigns its reference (address)** to the variable `input`. An object may invoke its methods. **To invoke a method on an object is to ask the object to perform a task**. PROGRAM 3-5 shows how the program reads a number from the keyboard and assigns the number to `radius`.

PROGRAM 3-5

```
1  import java.util.Scanner;
2  public class ComputeAreaWithConsoleInput {
3   public static void main(String[] args) {
4     // Create a Scanner object
5     Scanner input = new Scanner(System.in);
6     // Prompt the user to enter a radius
7     System.out.print("Enter a number for radius: ");
8     double radius = input.nextDouble();
9     // Compute area
10    double area = radius * radius * 3.14159;
11    System.out.println("The area for the circle of radius " +
12    radius + " is " + area); // Display results
13  }
14 }
```

PROGRAM OUTPUT

```
Enter a number for radius: 2.5
The area for the circle of radius 2.5 is 19.6349375
```

Line 7 displays a string "`Enter a number for radius: `" to the console. This is known as a **prompt**, because it directs the user to enter an input. **Your program should always prompt the user when expecting input from the keyboard.** The statement in line 8 reads input from the keyboard.

```
double radius = input.nextDouble();
```

After the user enters a number and presses the Enter key, the program reads the number assign it to radius and perform the calculation.

You have seen how to use the `nextDouble()` method in the Scanner class to read a double value from the keyboard. You can also use the methods listed in Table 4.2 to read other types of input such as the `byte`, `short`, `int`, `long`, and `float` type or text string.

Method	Description
`next()`	Returns the String value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.
`nextLine()`	Reads the rest of the current keyboard input line and returns the characters read as a value of type String. Note that the line terminator '\n' is read and discarded; it is not included in the string returned
`nextBoolean()`	Returns the next value of type boolean that is typed on the keyboard. The values of true and false are entered as the strings "true" and "false" . Any combination of upper- and/or lowercase letters is allowed in spelling "true" and "false"
`nextByte()`	Returns the next value of type byte that is typed on the keyboard
`nextShort()`	Returns the next value of type short that is typed on the keyboard.
`nextInt()`	Returns the next value of type int that is typed on the keyboard.
`nextLong()`	Returns the next value of type long that is typed on the keyboard
`nextFloat()`	Returns the next value of type float that is typed on the keyboard.
`nextDouble()`	Returns the next value of type double that is typed on the keyboard

Table 3.2 Methods for Scanner objects

PROGRAM 3-6

```
1   import java.util.Scanner;
2   public class ScannerInput {
3    public static void main(String[] args) {
4      byte n1;
5      int n2;
6      Scanner scannerObject = new Scanner(System.in);
7      System.out.println("Enter two whole numbers");
8      System.out.print(" separated by one or more spaces:");
9      n1 = scannerObject.nextByte();
10     n2 = scannerObject.nextInt();
11     System.out.println("You entered " + n1 + " and " + n2);
12     System.out.println("Next enter two words:");
13     String word1 = scannerObject.next();
14     String word2 = scannerObject.next();
15     System.out.println("You entered \"" +
16     word1 + "\" and \"" + word2 + "\"");
17     String junk = scannerObject.nextLine(); //To get rid of ' \n'
18     System.out.println("Next enter a line of text:");
19     String line = scannerObject.nextLine();
20     System.out.println("You entered: \"" + line + "\""); }
21  }
```

PROGRAM OUTPUT

```
Enter two whole numbers separated by one or more spaces: 42 43
You entered 42 and 43
Next enter two words:
jelly beans
You entered "jelly" and "beans"
Next enter a line of text:
Java flavored jelly beans are my favorite.
You entered "Java flavored jelly beans are my favorite."
```

PROGRAM 3-6 also illustrates some of the other Scanner methods for reading values from the keyboard. The method `nextByte()` and `nextInt()` works in exactly the same way as `nextDouble()`, except that it reads a value of type `byte` and `int` respectively.

Line 13 and 14, the method `next()` reads in a word.

```
String word1 = scannerObject.next();
String word2 = scannerObject.next();
```

If the input line is `jelly beans` then this will assign `w1` the string `"jelly"` and `w2` the string `"beans"`. A word is any string of non-whitespace characters delimited by whitespace characters such as blanks or the beginning or ending of a line.

If you want to read in an entire line, you would use the method `nextLine()`. For example,

```
String line = scannerObject.nextLine();
```

reads in one line of input and places the string that is read into the variable `line`. **The end of an input line is indicated by the escape sequence `'\n'`**. This `'\n'` character is what you input when you press the Enter (Return) key on the keyboard. On the screen, it is indicated by the ending of one line and the beginning of the next line. When `nextLine()` reads a line of text, it reads this `'\n'` character, so the next reading of input begins on the next line. However**, the `'\n'` does not become part of the string value returned**. So, in the previous code, the string named by the variable `line` does not end with the `'\n'` character.

3.4 Exception Handling

Exception handling enables a program to deal with exceptional situations and continue its normal execution or gracefully exit. **Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out**. If you enter a double value when your program expects an integer, you will get a runtime error with an InputMismatchException.

In Java, runtime errors are triggered (thrown) as exceptions. An exception is an object that represents an error or a condition that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally. When an exception is thrown, it can be caught and handled in a `try-catch` block, as follows:

```
try {
 statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
 handler for exception1;
}
catch (Exception2 exVar2) {
 handler for exception2;
}
...
catch (ExceptionN exVarN) {
 handler for exceptionN;
}
```

PROGRAM 3-7 illustrates how you can use exception handling to inform a user of the errors encountered by the program and thereafter allow it to terminate gracefully.

PROGRAM 3-7

```
1   import java.util.*;
2   public class ScannerInput {
3    public static void main(String[] args) {
4      Scanner input = new Scanner(System.in);
5      int result = 0;
6      try{
7         System.out.print("Enter 2 numbers:");
8         byte num1 = input.nextByte();
9         int num2 = input.nextInt();
10        if (num2 == 0)
11          throw new ArithmeticException("Divisor cannot be zero");
12        result = num1/num2;
13     }
14     catch(InputMismatchException ex){  // handle type out of range
15         System.out.println("The number enter is out of range of a byte"
16  +
17         "type");
18         System.out.println(ex);
19         System.out.println("Program Exited!");
20         System.exit(1);
21     }
22     catch(ArithmeticException ex){  // handle division by zero
23         System.out.println("Cannot divide by zero");
24         System.out.println(ex);
25         System.out.println("Program exited!");
26         System.exit(1);
27     }
28     System.out.println("The result of the division is " + result);
29    }
    }
```

PROGRAM OUTPUT 1

```
Enter 2 numbers: 120 4
The result of the division is 30
```

PROGRAM OUTPUT 2

```
Enter 2 numbers: 135
The number enter is out of range of a byte type
java.util.InputMismatchException: Value out of range. Value:"135" Radix:10
Program Exited!
```

PROGRAM OUTPUT 3

```
Enter 2 numbers: 100 0
Cannot divide by zero
java.lang.ArithmeticException: Divisor cannot be zero
Program exited!
```

In PROGRAM 3-7, there are two possible runtime errors that may occur; (1) Input out of range and (2) Division by zero. When a value that is not within the range of a byte is entered, it will trigger an InputMismatchException. This exception will be handled by the `catch` block in line 14 because an exception handler is defined to handle the error.

If a zero is entered for the second number, the program will trigger line 11 to throw an ArithmeticException. The flow of execution will jump to line 21, the `catch` block that was defined to handle the division by zero error.

If no exceptions arise during the execution of the `try` block, the `catch` blocks are skipped. **If one of the statements inside the `try` block throws an exception, Java skips the remaining statements in the `try` block and starts the process of finding the code to handle the exception**. **The process of finding a handler is called catching an exception.**

The code that handles the exception is called the exception handler. Each **`catch`** block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the `catch` block.

If so, the exception object is assigned to the variable declared, and the code in the `catch` block is executed. If no handler is found, the program terminates and prints an error message on the console.

Note: Even if Line 10-11 is omitted, Java is still able to detect the type of exception and handle it accordingly. Other types of exceptions will be introduced in later chapters when appropriate.

3.5 Common Error: Dealing with the Line Terminator \n

The method `nextLine()` of the class Scanner reads the remainder of a line of text starting wherever the last keyboard reading left off. For example, suppose you create an object of the class Scanner as follows:

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

Suppose you continue with the following code:

```
int n = keyboard.nextInt();
String s1 = keyboard.nextLine();
```

```
String s2 = keyboard.nextLine();
```

Now, assume that the input typed on the keyboard is the following:

```
2 heads are
better than
1 head.
```

This sets the value of the variable `n` to 2, that of the variable `s1` to "heads are", and that of the variable `s2` to "better than". So far there are no problems, but suppose the inputs were instead as follows:

```
2
heads are better than
1 head.
```

You might expect the value of `n` to be set to 2, the value of the variable `s1` to "heads are better than", and that of the variable `s2` to "1 head". But that is not what happens. What actually happens is that the value of the variable `n` is set to 2, that of the variable `s1` is set to the empty string, and that of the variable `s2` to "heads are better than". The method `nextInt()` reads the 2 but does not read the end-of-line character '\n' . So the first `nextLine()` invocation reads the rest of the line that contains the 2.

There is nothing more on that line (except for '\n'), so nextLine() returns the empty string. The second invocation of `nextLine()` begins on the next line and reads "heads are better than". When combining different methods for reading from the keyboard, you sometimes have to include an extra invocation of `nextLine()` to get rid of the end of a line (to get rid of a '\n').

[Reference]

[1] Introduction to Java Programming Comprehensive Version 10th Ed, Daniel Liang, 2016.
[2] Java How To Program 10th Ed, Paul Deitel, Harvey Deitel, 2016.

[Self-Review Question]

1. Examine the following program, which reads in a temperature in Fahrenheit and converts it to Celsius. Fill in blanks.

```
import ____________________;

class FahrenheitToCelsius {

    public static void main(String[] args) {

    Scanner in = ____________________________;

    System.out.println("Enter temperature in Fahrenheit");

    double temp = ___________________;

    temp = ________________________________;

    System.out.println("Temperature in Celsius = " _______________);

    }
}
```