

# Objects, Input & Output

(CISS-110 William Jojo)

## Objects

Let us continue our journey into OOP by describing what objects are and some terminology surrounding their use. Recall the use of the `String` class with the following:

```
String s;  
  
s = "My string value.";
```

The variable `s` is known as a *reference variable*. Remember that the type `String` is a class, not a primitive type like `int`. The reference variable `s` contains an address that references the string "My string value." That address is the *String object* or, more to the point, an *instance* of the `String` class.

Technically speaking, the above assignment statement is really shorthand for the following:

```
s = new String("My string value.");
```

The `new` reserved word indicates we want to create or *instantiate* a `String` class object. This also causes `String()` to be invoked which is known as a *constructor*. The constructor is responsible for the setup associated with a new instantiation of a class and, in this case, sets a specified value. Finally the address of the instance is stored in `s`.

We are able to use the previous version since the assignment operator acts as a `new` and instantiates the `String` object for us. You can think of the operator as being overloaded for assigning values for primitive types and for instantiating `String` objects.

Variables of our primitive data types store the data values directly in their named memory locations and do not store addresses of the data.

## Packages of predefined classes and methods.

The `String` class is defined in the package `java.lang`, which you should recall is the package that is automatically imported for our Java programs.

Java is loaded with predefined classes such as the `String` class. These classes also possess predefined methods. You can use any of these classes and their associated methods in your programs as long as you know what package to import and you know the syntax of the methods for a particular class.

```

import java.util.*;

public class methods
{
    static Scanner kb = new Scanner(System.in);

    public static void main(String[] args)
    {

        double x, y, square;

        System.out.println("\nThis program uses pow to demonstrate");
        System.out.println("static methods. It will also find");
        System.out.println("the min and max of the two values");
        System.out.println("you have entered.\n");

        System.out.print("Enter a value for x: ");
        x = kb.nextDouble();

        System.out.print("Enter a value for y: ");
        y = kb.nextDouble();

        System.out.println("The value of x raised to the y is " +
            Math.pow(x,y));

        System.out.println("The min value is " + Math.min(x,y));
        System.out.println("The max value is " + Math.max(x,y));
    }
}

```

*Example 1: Using static methods from the Math class.*

You also need to know if the methods are *static* or *non-static* methods. Simply put, static methods do not require an instance of an object in order for you to use them. You can simply call the method *on the class*.

Methods often require additional information. This additional information is called the method's *arguments* or *parameters*. These are *passed* to a method by simply placing the values in the parentheses of the method. We have already seen quite a bit of this with our output statements. Note, too, that the use of the `Scanner` class has shown us methods that require no arguments, like `nextInt()`. Even though the method required no input from us, we still provided the parentheses. This is essential to indicate to the compiler that we are invoking a method and not trying to access some identifier called `nextInt`.

The `String` class consists of a majority of non-static methods. The `Math` class is a class whose methods are all static. This means we can use all of these methods by simply using

the class name with the method.

The program in *Example 1* demonstrates the use of some static methods from the `Math` class. A sample run is shown below with user-provided values in bold.

```
$ javac methods.java
$ java methods
```

```
This program uses pow to demonstrate
static methods. It will also find
the min and max of the two values
you have entered.
```

```
Enter a value for x: 3.4
Enter a value for y: 2.7
The value of x raised to the y is 27.226579002152647
The min value is 2.7
The max value is 3.4
$
```

Some `Math` methods are shown in *Table 1*.

Method	Purpose
<code>static double <b>cbrt</b>(double a)</code>	Returns the cube root of the value in <i>a</i> .
<code>static double <b>pow</b>(double x, double y)</code>	Returns x raised to the y power.
<code>static double <b>exp</b>(double a)</code>	Return Euler's number e raised to the <i>a</i> power.
<code>static double <b>random</b>()</code>	Returns a positive double value $\geq 0.0$ and $< 1.0$

*Table 1: Some static methods available to the `Math` class.*

## String – Revisited

The `String` class really needs to have some of its more exciting methods discussed. It has already been stated how important the `String` class is to Java. Consider the next bit of code:

```
String s;

s = "My string value.";
/* several statements later */
s = "A new string value.";
```

This bit of code demonstrates a very important aspect of Java and object management. The reference variable `s` contains the instance of "My string value." initially. Then, in the not too distant future, possesses the instance of "A new string value."

Let us consider this very carefully. The assignment operator clearly has the power to instantiate a `String` object and assign the address to `s` which means that for each assignment made, a new object was instantiated. The address of the first was assigned to `s`, then the address of the second was assigned. So when the second address was assigned, what happened to the memory for the first instance and what of the knowledge of its whereabouts?

In the C programming language we would call this a *memory leak* – losing track of something as a result of carelessness and, once lost, unable to be reclaimed.

Not so with Java! Java keeps track of every instance of every class and a reference count of the number of reference variables that have knowledge of an object. When the reference count of an instance reaches zero, the instance is destroyed and the memory freed up to be used again. This is known as *garbage collection*. You can force garbage collection at any time during your program by simple calling the garbage collector with:

```
System.gc();
```

Now for some more fun with the `String` class. Table 2 shows some commonly used `String` methods.

String Method	Purpose
<code>char charAt(int index)</code>	Returns the character at <i>index</i> .
<code>int indexOf(char ch)</code>	Returns the index of the first occurrence of <i>ch</i> within the string or -1 if not found.
<code>String concat(String str)</code>	Returns the string that is this string concatenated with the string value of <i>str</i> .
<code>int length()</code>	Return the length of this string.
<code>String substring(int start, int end)</code>	Returns the string that is the substring of this string starting at <i>start</i> until <i>end-1</i> .
<code>String toLowerCase()</code>	Returns the string that is this string with all letters set to lower case.
<code>String toUpperCase()</code>	Returns the string that is this string with all letter set to upper case.

Table 2: Some common `String` methods.

```

public class stringEx
{
    public static void main(String[] args)
    {
        String s = "The quick brown fox jumps over the lazy dog.";
        String q, d, j;

        System.out.println("The length of s is " + s.length());
        System.out.println("s.toUpperCase() is " + s.toUpperCase());
        System.out.println("s.toLowerCase() is " + s.toLowerCase());

        q = s.substring(4, 9);
        d = s.substring(40, 43);
        j = s.substring(20, 25);

        System.out.println(q + " " + d + " " + j + ".");
    }
}

```

*Example 2: Program to demonstrate some String methods.*

*Example 2* showcases some of these string methods and the output of a sample run, with user-provided input in bold, is shown below.

```

$ javac stringEx.java
$ java stringEx
The length of s is 44
s.toUpperCase() is THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
s.toLowerCase() is the quick brown fox jumps over the lazy dog.
quick dog jumps.
$

```

## Output with the printf() method

New with JDK 5.0 is the `printf()` method of the standard output object `System.out`. Those familiar with the `printf()` function of the C programming language will have no problem adapting to the Java equivalent method. Consider the following example:

```

final float PI=3.14159;
String s = "My string.";

System.out.printf("Here is the output of our program.%n");
System.out.printf("The value of Pi is %.5f%n", PI);
System.out.printf("The value of s is \"%s\"%n", s);

```

The `printf()` method takes as its first argument a string. This string is also known as the *format string*. Zero or more additional arguments may follow the format string. The format string is essentially a combination of literal text and sequences of place holders, or *format specifiers*, that define *conversions*. The format specifiers are designed to hold a place in the format string which will be substituted at runtime with a value appropriate to the conversion type.

A format specifier is denoted by the percent (%) sign to indicate that the next sequence of characters describes how the value should be displayed. If a format string contains no format specifiers, then all of the text contained in the format string is processed as if it were a `print()` method, that is, the cursor remains on the current line and does not move to the beginning of the next line unless an explicit newline (`\n` or `%n`) is placed at the end of the format string. A format specifiers had the following form:

`%[arglist$][flags][width][.precision]conversion`

All of the bracketed values are optional and their values depend on the conversion selected. Some typical format specifiers are shown in *Table 3*. One of the really nice features of the `printf()` method is to chop a line into fields of specific sizes to align columnar data. Now, by default values are right justified. If the width is wider than the value to be printed, the value will be padded on the left with spaces. If the width is smaller than the value to be displayed, the space will be enlarged to fit the value.

Format Specifier	Resultant Output
<code>%-30s</code>	Left justified, space padded, string data that occupies 30 character columns.
<code>%c</code>	Single character data.
<code>%e</code>	Precision floating point data in scientific “e” notation.
<code>%.3f</code>	Precision floating point data as a decimal value printed to 3 decimal places.
<code>%10d</code>	Right justified, space padded, decimal integer that occupies 10 character columns. (No precision)
<code>%03d</code>	Zero padded, decimal integer that occupies 3 character columns. (No precision)
<code>%%</code>	Percent sign.
<code>%n</code>	Platform specific newline.

*Table 3: Examples of some simple printf() format specifiers.*

```

public class printf
{
    public static void main(String[] args)
    {
        String h1="Name", h2="Test 1", h3="Test 2",
            h4="Test 3", h5="Average";
        String u1="----", u2="-----", u3="-----";
        String name="Steve Jones";
        int test1=90, test2=85, test3=87;
        double average=(test1 + test2 + test3)/3.0;

        System.out.printf("%n%-20s%10s%10s%10s%10s%n",
            h1, h2, h3, h4, h5);
        System.out.printf("%-20s%10s%10s%10s%10s%n",
            u1, u2, u2, u2, u3);
        System.out.printf("%-20s%10d%10d%10d%10.2f%n%n",
            name, test1, test2, test3, average);
    }
}

```

*Example 3: Program demonstrating the use of the printf() method.*

The program in *Example 3* creates a formatted table of output which is shown below.

```

$ javac printf.java
$ java printf

```

Name	Test 1	Test 2	Test 3	Average
----	-----	-----	-----	-----
Steve Jones	90	85	87	87.33

```

$

```

## Wrapper Classes

It will soon become necessary to convert the string representation of numeric quantities, or *numeric strings*, into actual integers or doubles. When this happens you can use the *wrapper* classes `Integer`, `Float` and `Double`. These classes contain many static methods, one of which is use for *parsing*. Parsing is the term given to evaluating a sequence of characters and determining if they fit a particular expected pattern. Some examples of using the wrapper classes are shown in *Table 4*.

Wrapper Class Method	Purpose
<code>Integer.parseInt("12345");</code>	Parses the string "12345" and returns an integer 12345.
<code>Integer.parseInt(s);</code>	Parses the string s and returns the integer value represented by s.
<code>Float.parseFloat("3.14");</code>	Parses the string "3.14" and returns the float value 3.14.
<code>Double.parseDouble("3.1415926");</code>	Parses the string "3.1415926" and returns the double value 3.1415926.

Table 4: Examples using wrapper classes.

## GUI-based I/O

On to the fun stuff! No programming class is complete without delving into the Graphical User Interface (GUI). This section introduces two methods of the `JOptionPane` object. The `JOptionPane` class may be used in your program by importing the `javax.swing` package.

The first method of the `JOptionPane` is `showInputDialog()` and is intended for user input. This method comes in several flavors with several arguments that could be passed to each version. However, for the sake of introduction, we'll stick to the simplest form which is demonstrated in the following:

```
String name;
```

```
name = JOptionPane.showInputDialog("Please enter your name.");
```

This code declares a string called `input` and this string is used to collect the resulting `String` object return by the method. The method prompts the user with the string provided as an argument. The resultant dialog box is shown in *Illustration 1*.



Illustration 1: Result of invoking the `showInputDialog()` method.

It is important to remember that `showInputDialog()` returns a `String` object. This may be obvious since we are asking the user for their name, but keep in mind that any use of this method will always return a `String` object. This means if you were reading a numeric quantity from the user, the `String` object will need to be converted. This is demonstrated in the next piece of code.

```
String input;
int age;

input = JOptionPane.showInputDialog("What is your age" +
    " in years?");
age = Integer.parseInt(input);
```

Output is just as important as reading input from the user when it comes to the GUI. So another method is now introduced called the `showMessageDialog()` method. This method has several versions, similar to `showInputDialog()`, and again we will use only one here. The `showMessageDialog()` method shown here uses 4 arguments. The general layout is as follows:

```
JOptionPane.showMessageDialog(parentComponent,
    message, title,
    messageType);
```

The `parentComponent` is an object that represents the parent of the the dialog box. This will simply be the reserved word, `null`, for now indicating that the default component should be used that causes the dialog to appear in the center of the screen.

The `message` is the body of the dialog and is usually expressed in the form of strings used similar to the `println()` method.

The `title` is simply the top line title of the dialog box.

Lastly, the `messageType` is the category of message being delivered to the user. The options for `messageType` are shown in *Table 5*. These values are constants that are defined in the `JOptionPane` object.

Each of the dialog box examples in *Table 5* were created with the following statement changing only the value of `messageType`:

```
JOptionPane.showMessageDialog(null,  
    "Message",  
    "Title",  
    JOptionPane.ERROR_MESSAGE);
```

When you have output that needs specific precision in a dialog box, there is no direct way to process this within the `showMessageDialog()` method. There is a way, however, to prepare the formatted output so that it can be part of the message argument of the dialog box.

The `format()` static method of the `String` class offers a way to get the precision you need for your output using a format string similar to the `printf()` method. The program in *Example 4* demonstrates the use of `showInputDialog()`, `format()` and the `showMessageDialog()` methods.

```
import javax.swing.*;  
  
public class perimeter  
{  
    public static void main(String[] args)  
    {  
        String input, output;  
        double length, width, perimeter;  
  
        input = JOptionPane.showInputDialog("Length of box:");  
        length = Double.parseDouble(input);  
  
        input = JOptionPane.showInputDialog("Width of box");  
        width = Double.parseDouble(input);  
  
        perimeter = 2 * length + 2 * width;  
  
        output = String.format("For the box with length %.2f%n", length)  
            + String.format("and width %.2f%n", width)  
            + String.format("the perimeter is %.2f%n", perimeter);  
  
        JOptionPane.showMessageDialog(null,  
            output,  
            "Perimeter of the box.",  
            JOptionPane.INFORMATION_MESSAGE);  
  
        System.exit(0);  
    }  
}
```

*Example 4: Program to demonstrate methods for input, formatting and output in a GUI.*

Value of messageType	Example
JOptionPane.ERROR_MESSAGE	
JOptionPane.INFORMATION_MESSAGE	
JOptionPane.PLAIN_MESSAGE	
JOptionPane.QUESTION_MESSAGE	
JOptionPane.WARNING_MESSAGE	

Table 5: Example dialogs for the different messageType values.

Java programs that use the Swing or other GUI components must use the `System.exit()` method to assure proper termination. An argument value of zero, as shown in *Example 4*, indicates successful program termination.

## File Handling

Ultimately, you will need to handle data whose volume simply cannot be typed in or needs to be stored for a period longer than that of `showMessageDialog()`.

This is where *files* come into play. As a programmer, you should already have some idea of what files are and that their names and extensions mean a great many things about the type of data plus any special nature of the layout and encoding of its content.

Recall the `Scanner` class can take an argument like `System.in` to help associate a reference variable to the standard input object. We can also associate a filename to the `Scanner` class by using another class called `FileReader`. The `FileReader` class takes a filename as its argument and returns an object suitable for `Scanner`.

Using the example of calculating an average, if we know that a file contains a person's name and test scores, we can read them in and produce the same results. Let's modify *Example 3* to process the same data from the file `gradesFile.data`. A new version of this program is in *Example 5*.

The `gradesFile.data` file contains just one line:

```
Steve Jones 90 85 87
```

The first and last names will be read separately. Recall that the `next()` method of the `Scanner` class reads the next word and that the `Scanner` methods break on whitespace. We could have just as easily used a file in the form:

```
Steve
Jones
90
85
87
```

The `java.io` package will need to be imported to access the `FileReader` class, and, as shown in *Example 6*, the `PrintWriter` class. What is wonderful about the `PrintWriter` class is that our `outFile` reference variable inherits the methods that we previously used with `System.out`.

We also will continue to enjoy the input methods inherited by `inFile` as we previously did when simply reading from the keyboard since `inFile` is an instance of `Scanner`.

```

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

public class gradesFile
{
    public static void main(String[] args)
        throws FileNotFoundException
    {
        String h1="Name", h2="Test 1", h3="Test 2",
            h4="Test 3", h5="Average";
        String u1="----", u2="-----", u3="-----";
        String name;
        int test1, test2, test3;
        double average;
        Scanner inFile = new Scanner(
            new FileReader("gradesFile.data"));

        // Read first and last name into name.
        name = inFile.next() + " " + inFile.next();

        test1 = inFile.nextInt();
        test2 = inFile.nextInt();
        test3 = inFile.nextInt();

        // close the file
        inFile.close();

        average = (test1 + test2 + test3) / 3.0;

        System.out.printf("%n%-20s%10s%10s%10s%10s%n",
            h1, h2, h3, h4, h5);
        System.out.printf("%-20s%10s%10s%10s%10s%n",
            u1, u2, u2, u2, u3);
        System.out.printf("%-20s%10d%10d%10d%10.2f%n%n",
            name, test1, test2, test3, average);
    }
}

```

*Example 5: Program using data file and FileReader class.*

```

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

public class gradesReport
{
    public static void main(String[] args)
        throws FileNotFoundException
    {
        String h1="Name", h2="Test 1", h3="Test 2",
            h4="Test 3", h5="Average";
        String u1="----", u2="-----", u3="-----";
        String name;
        int test1, test2, test3;
        double average;
        Scanner inFile = new Scanner(
            new FileReader("gradesFile.data"));
        PrintWriter outFile =
            new PrintWriter("gradesFile.report");

        // Read first and last name into name.
        name = inFile.next() + " " + inFile.next();

        test1 = inFile.nextInt();
        test2 = inFile.nextInt();
        test3 = inFile.nextInt();

        average = (test1 + test2 + test3) / 3.0;

        outFile.printf("%n%-20s%10s%10s%10s%10s%n",
            h1, h2, h3, h4, h5);
        outFile.printf("%-20s%10s%10s%10s%10s%n",
            u1, u2, u2, u2, u3);
        outFile.printf("%-20s%10d%10d%10d%10.2f%n%n",
            name, test1, test2, test3, average);

        // close the files
        inFile.close();
        outFile.close();
    }
}

```

*Example 6: The gradesFile program with added PrintWriter class.*