Methods in Java – Basics

As the complexity of a task increases, it becomes more and more difficult to see the entire solution at once. It is often useful to break the problem into manageable pieces. After solving each subproblem, the solutions can be combined into a solution to the entire problem.

This is known as *modular programming*, where the program is broken into separate parts, or *modules*, which are then assembled into an overall solution. In Java, the primary structure for modularity is the *method*.

Modular programming offers some distinct advantages:

- Break the problem into smaller pieces, making them easier to understand and solve.
- Multiple programmers can work on the same overall problem by solving its subproblems.
- Modules can be reused in other parts of the program.
- Some modules can be reused in other programs.
- A module is easier to test in detail than a larger program solving a complicated problem.

Methods as Modular Programming

Methods should already be familiar. Many first programs use the `println` method to produce output. The String class has built-in methods such as equals and `compareTo`, and the Math class have numerous useful methods for mathematical operations. So far, the only method we have written is the main method, required in all programs, but now we will begin writing our own methods.

Example 1 – Convenient Output of a Message

Suppose we are writing a program that requires the identical address information to be output at regular intervals.

Ahmed's Video Wonderland
123 Roehampton Avenue
613-555-1212

The following code can be used to output this information whenever we need it by calling the associated method. This is the *method definition*.

```
public static void printHeading ()
{
  System.out.println();
  System.out.println("Ahmed's Video Wonderland");
  System.out.println("   123 Roehampton Avenue");
  System.out.println("      613-555-1212");
}
```

This code very closely resembles out previous programs, except that instead of a main method, this

method is called `printHeading`. Also notice that the parentheses following `printHeading` are empty.

To use this method, the simplest option is to call, or *invoke*, the `printHeading` method from within the main method of the same class. When the main method executes, it finds the reference to the `printHeading` method, and executes the code from `printHeading`.

```
class Sample
{
  public static void printHeading ()
  {
      System.out.println();
      System.out.println("Ahmed's Video Wonderland");
      System.out.println("   123 Roehampton Avenue");
      System.out.println("      613-555-1212");
  }

  public static void main (String[] args)
  {
      printHeading();
  }
}
```

Note that we need to include the parentheses and semicolon with the call to printHeading in the main method. This lets Java know that printHeading is a method, and not simply a variable.

Passing Parameters to Methods

To make methods more flexible, we often want to give them different values to use in their processing and calculations. Consider the following method, which outputs any character a specified number of times on the same line.

```
public static void printRow (char c, int n)
{
  for (int i = 1; i <= n; i++)
  {
    System.out.print(c);
  }
}
```

In the definition of the method printRow, the char c and int n are called the *parameter list*. They identify the variables c and n as *parameters* of the method. This method might be called by

```
printRow('*', 10);
```

which will result in ten asterisk characters output on the same line. The character '*' and the number 10 are the arguments of the method, and their values are *passed* to the parameters of the method.

In this case, the arguments of the method are *constants*, but it is also valid to use *variables* or

expressions (mathematical or logical) as arguments for a method.

When the argument to a method is a *variable*, it is important to understand that a copy of the value is passed to the method. Thus the original value contained in the variable does not change, regardless of what happens to the parameter inside the method.

Example 2 – Passing a Variable as a Parameter to a Method

Suppose we have a method that, among its other statements, increments the input parameter by one. It might look like the following:

```
public static void sample (int n)
{
  ...          // other useful code
  n++;
  ...          // other useful code
}
```

Suppose we were to call this method by writing

```
int n = 3;
sample(n);
```

The argument of the method call is the variable, `n`, containing the value of 3. As part of the *parameter passing* process, a copy is made of the value of `n` (3), and it is this copy that is received by the actual method. So even though the method changes the value of `n` to 4 inside the method, the actual value of the variable `n`, in the calling code, is unchanged.

Methods in Java – Basics

Questions:

1. Explain the difference between a method *definition* and a method *invokation*.

2. Write a method that will simulate the results of rolling two fair (balanced) dice by printing two random integer values in the range 1 to 6 along with their total. Sample output from a call to this method could be: `4 and 3 - a total of 7`

3. A student wrote the following method to exchange the values of two integer variables.

```
public static void swap (int m, int n)
{
 int temp = m;
 m = n;
 n = temp;
}
```

   He then tested the method with the following:

```
i = 7;
j = 3;
swap(i, j);
System.out.println("i = " + i + " and j = " + j);
```

   What did the fragment output? Explain.

4. Complete the definition of the following method, `printRect`, so that it prints a filled rectangular pattern, using the character `c`, that is `width` characters wide and `height` characters high.

```
public static void printRect (char c, int width, int height)
```