Chapter 5: Methods

Based on Introduction to Java Programming, Y. Daniel Liang, Brief Version, 9/E

You should be familiar with some of the material in this chapter (from CSCI-UA 004 or equivalent). We will cover the following concepts:

- defining methods
- calling methods
- passing parameters by value
- overloading methods
- scope of variables
- Math class methods

Motivation: why use methods?

- modularizing code
- reusing code
- easier debugging

Defining Methods

syntax:

```
modifiers returnType methodName ( list-of-parameters )
{
    method body
}
```

`returnType` can be any of the data types that we learned so far, or `void`; `void` methods do not return a value.

See Figure 1 for visual definitions of ”parts” of a method.
Calling Methods

Calling a method results in executing the code in that method.

- value returning methods are used like values, for example,

  ```java
  int smaller = min ( num1, num2 );
  ```

- void methods are use like statements (in fact, you used these already), for example,

  ```java
  System.out.println("Hello world");
  ```

Syntax:

```java
methodName( list-of-arguments );
```
Passing Parameters by Values

- When the function is called, the values/variables specified in the list-of-arguments must match the list-of-parameters from the function definition in
  - order,
  - number,
  - compatible type.

- In Java all parameters are passed to methods by value. This means that the content of the argument variables are copied to variables specific to the method.

What is the output of the following code?

```java
public class Increment {
    public static void main(String[] args) {
        int x = 1;
        System.out.println("Before the call, x is " + x);
        increment(x);
        System.out.println("after the call, x is " + x);
    }

    public static void increment(int n) {
        n++;
        System.out.println("n inside the method is " + n);
    }
}
```

Overloading Methods

**Overloading methods** - defining multiple methods with the same name, but different signatures. A signature of a method consists of its name and its parameter list.

```java
public class TestMethodOverloading {

    /** Main method */
    public static void main(String[] args) {
        // Invoke the max method with int parameters
        System.out.println("The maximum between 3 and 4 is "
                           + max(3, 4));
        // Invoke the max method with the double parameters
```
System.out.println("The maximum between 3.0 and 5.4 is "+max(3.0, 5.4));

// Invoke the max method with three double parameters
System.out.println("The maximum between 3.0, 5.4, and 10.14 is "+max(3.0, 5.4, 10.14));
}

/** Return the max between two int values */
public static int max(int num1, int num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}

/** Find the max between two double values */
public static double max(double num1, double num2) {
    if (num1 > num2)
        return num1;
    else
        return num2;
}

/** Return the max among three double values */
public static double max(double num1, double num2, double num3) {
    return max(max(num1, num2), num3);
}

Scope of Variables

**scope of variable** the part of the program in which the variable can be referenced

**local variable** = variable with local scope - variable declared inside a method; method parameters are also local variables

**block scope** - variable defined inside a block of code (code surrounded by curly braces { } ) can be references only within that block; nested blocks cannot have repeating variables

**Example 1:** When blocks are not nested, the variable name can be reused.
```java
public static void main(String[] args) {
    ...  
    int x = 5;
    System.out.printf("%d", x);
    
    int x = 7;
    System.out.printf("%d", x);
    ...  
}
```

**Example 2:** If blocks are nested, the re-declaring a variable with the same name will cause a compile-time error.

```java
public static void main(String[] args) {
    ...  
    int x = 5;
    System.out.printf("%d", x);
    
    int x = 7;
    System.out.printf("%d", x);
    ...  
}
```

**The Math Class**

The Math class provides methods that implement basic mathematical functions. You already used it for random number generation.

For a complete list of methods and constants provided by the Math class see:

http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

**Generating random things**

Math.rand() generates a random double value greater than or equal to 0.0 and less than 1.0:

```
0.0 <= Math.rand() < 1.0
```
To generate a random double value in the range dMin <= randomDoubleValue < dMax use:

\[ \text{Math.rand()} \times (dMax - dMin) + dMin \]

Example: Math.rand() * 10.0 - 5.0 generates random double values greater than or equal to 5.0 and less than 5.0.

To generate a random int value in the range iMin <= randomIntValue <= iMax use:

\[ (\text{int}) \left( \text{Math.rand()} \times (iMax - iMin + 1) \right) + iMin \]

Example: (int) (Math.rand() * 11) - 5 generates random int values greater than or equal to -5 and less than or equal to 5.

Note: the smallest possible integer represented by type int and the largest possible integer represented by type int can be obtained by using Integer.MIN_VALUE and Integer.MAX_VALUE, respectively.

To generate a random char value remember that each character has a unique Unicode between 0 and 65535. To generate a random character (any character) is to generate a random integer from 0 to 65535 and then cast it to a value of type char:

\[ (\text{char}) (\text{(int)} (\text{Math.rand()} \times (65535 + 1))) \]

To generate random character from a specific range of characters, you simply need to specify iMin and iMax in the formula for random integers by casting the starting and ending characters to a value of type int:

\[ (\text{char}) \left( (\text{int}) (\text{Math.rand()} \times ((\text{int})\text{charMax} - (\text{int})\text{charMin} + 1)) \right) + (\text{int})\text{charMin} \]

Example:

\[ (\text{char}) \left( (\text{int}) (\text{Math.rand()} \times ((\text{int})'z' - (\text{int})'a' + 1)) + (\text{int})'a' \right) \]
generates random char values in the range from 'a' to 'z' (i.e. any lower case letter).

**Method Abstraction and Stepwise Refinement**

These are probably the most important concepts regarding Java programming that you learned so far.

**Method abstraction** - a person (a client/programmer/user) can use a method without knowing its implementation. The only thing needed should be the method’s signature (method name, list of parameters), its return type and its documentation. The author of the method can change its implementation and, as long as the header of the method remains the same, the client code does not need to be modified.
**Stepwise refinement** - when writing a large program, one should use a divide and conquer approach to decompose the problem into subproblems, which then get decomposed into smaller subproblems and so on, until the final set of problems is of manageable size. What is manageable size? The "small problems" should be easily described as a single task and should be easily turned into simple methods or few lines of code.

**Example problem:** Simulate a single game of craps. Craps is a dice game. The rules of our version of the game follow.

**step1:** Roll two dice and check their sum. If the sum is 2, 3, or 12 (called *craps*), the player loses and the game ends. If the sum is 7 or 11 (called *natural*), the player wins and the game ends. If the sum is any other number (i.e., 4, 5, 6, 8, 9 or 10, called *point*) the dice are rolled again (see step 2).

**step 2:** Roll the dice and check their sum. If the sum is 7, the player loses and the game ends. If the sum is equal to *point*, the player wins and the game ends. If the sum is any other number, the step 2 is repeated.

**Stepwise refinement of the problem:**

- play the game of craps
  
  ─ play the game of craps
  ─ step1, as described above
  ─ step2, as described above

- play the game of craps
  
  – step1, as described above
  ─ compute the sum on two dice
  ─ determine the next step based on the sum
  – step2, as described above
  ─ compute the sum on two dice
  ─ determine the next step based on the sum
play the game of craps

  - step1, as described above
    * compute the sum on two dice
      - simulate rolling of die #1
      - simulate rolling of die #2
      - compute their sum
    * determine the next step based on the sum
      - if sum is 2, 3, or 12, player loses, game ends
      - if sum is 7 or 11, player wins, game ends
      - otherwise copy sum to point and proceed to step 2

  - step2, as described above
    * compute the sum on two dice
      - simulate rolling of die #1
      - simulate rolling of die #2
      - compute their sum
    * determine the next step based on the sum
      - if sum is 7, player loses, game ends
      - if sum is point, player wins, game ends
      - otherwise, repeat step2