FUNCTIONS

STUDENT OUTLINE

Lesson 7 – More About Methods

INTRODUCTION: Programs of any significant size are broken down into logical pieces called methods. It was recognized long ago that programming is best done in small sections that are connected in very specific and formal ways. Java provides the construct of a function, allowing the programmer to develop new functions not provided in the original Java libraries. Breaking down a program into blocks or sections leads to another programming issue regarding identifier scope. Also, functions need to communicate with other parts of a program that requires the mechanics of parameter lists and a return value.

The key topics for this lesson are:

A. Writing Methods in Java
B. Value Parameters and Returning Values
C. The Signature of a Method
D. Lifetime, Initialization, and Scope of Variables

VOCABULARY:

METHOD DECLARATION
METHOD DEFINITION
PARAMETERS
ACTUAL PARAMETERS
VALUE PARAMETERS
FORMAL PARAMETERS
SIGNATURE
SCOPE
GLOBAL SCOPE
LOCAL SCOPE
BLOCK
FUNCTION
STATIC VARIABLE

DISCUSSION:

A. Writing Methods in Java

1. A method is like a box that takes data in, solves a problem, and usually returns a value. The standard math methods follow this pattern:

```
Math.sqrt (2)  -->  1.414
Math.sin (30)  -->  -0.988 (Note: computation is in radians!)
```

2. There are times when the built-in methods of Java will not get the job done. We will often need to write customized methods that solve a problem using the basic tools of a programming language.

3. For example, suppose we need a program that converts gallons into liters. We could solve the problem within the *main* method, as shown in Program 7-1.

AAAACK!

Program 7-1

```
import chn.util.*;

class GallonsToLiters
{
  public static void main (String[] args)
  {
    ConsoleIO console = new ConsoleIO();

    System.out.println("Enter an amount of gallons --> ");
    double gallons = console.readDouble();
    double liters = gallons * 3.785;
    System.out.println("Amount in liters = " + liters);
  }
}
```

This works fine, but the mathematics of the conversion is buried inside the `main` method. The conversion tool is not available for general use. We are not following the software engineering principle of writing code that can be recycled in other programs.

4. Here is the same routine coded as a reusable method, which would allow for conversion of gallons to units other than liters:

Program7-2

```
import chn.util.*;

class FluidConverter
{
  public double toLiters(double amount)
  {
    return amount * 3.785;
  }
}
public class TestConverter
{
  public static void main(String[] args)
  {
    ConsoleIO console = new ConsoleIO();
    FluidConverter convert = new FluidConverter();

    System.out.print("Enter an amount of gallons --> ");
    double gallons = console.readDouble();
    System.out.println("Amount in liters = " +
      convert.toLiters(gallons));
  }
}
```

ACCEPT THE ARGUMENT

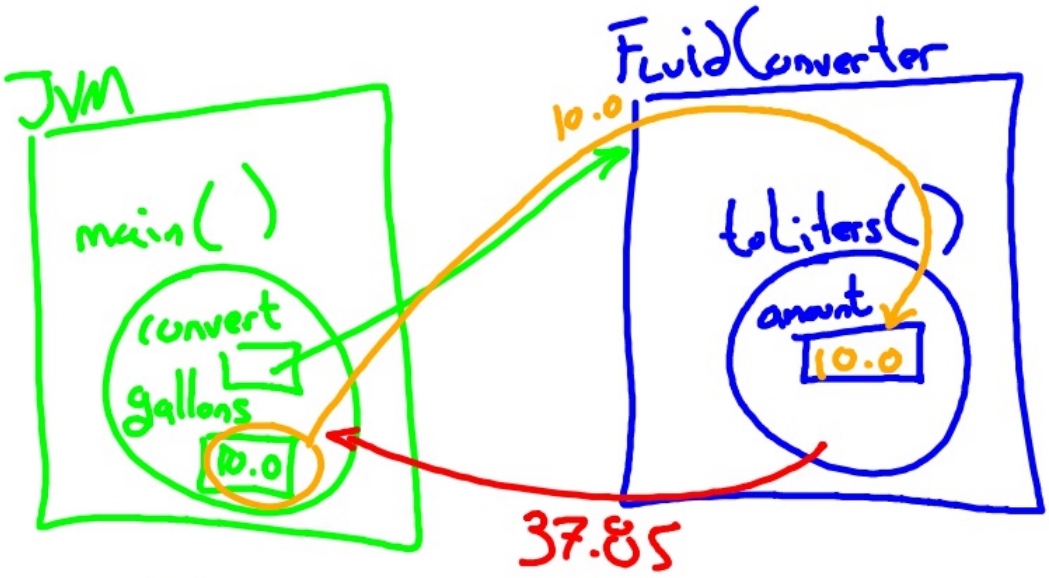
CAN ADD toGallons() toMiLi() toOz()

"DRIVER CLASS"

PASS THE PARAMETER

Sample run output:

```
Enter an amount of gallons --> 10
Amount in liters = 37.85
```



TIM THE ENCHANTER (CPU)

THIS IS THE STORY OF SCOPE ...

WHEN IS A VARIABLE VALID & ACCESSIBLE

5. Here is the sequence of events in Program 7-2.
   a. Execution begins in the method named `main` with the user prompt and the input of an amount of gallons.
   b. The `toLiters` method of the `convert` object is called and the number of gallons is passed as a parameter to the `toLiters` method.
   c. Program execution moves to `toLiters`, which does the computation and returns the answer to the calling statement.
   d. The answer is displayed.

6. The general syntax of a method declaration is

```
modifiers return_type method_name ( arguments )
{
    method_body
}
```

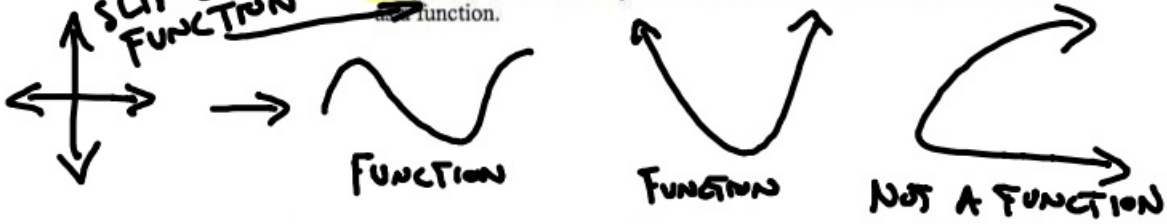
Example (from program 7-2):

```
public double toLiters(double gallons)
```

   a. The ***modifiers*** refers to a sequence of terms designating different kinds of methods. These will be discussed gradually in later lessons.
   b. The ***return_type*** refers to the type of data a method returns. The data type can be one of the predefined types (integer, double, char) or a user-defined type.
   c. *method_name* is the name of the method. It must be a valid identifier. In Program 7-2, the names of the methods are `main` and `toLiters`.
   d. The *arguments* list will allow us to accept values sent to a method. The argument list consists of one or more type-identifier pairs (example: `double amount`). The arguments in the method declaration are sometimes called the *formal parameters*. This is an older convention and can often lead to confusion. Use the word argument.
   e. The *method_body* contains statements to accomplish the work of the method. In the `toLiters` method there is one line in the body.

7. The last line of the main method contains the following reference to a method: `convert.toLiters(gallons)`. This line of code is known as a *method call* and causes the value represented by the variable `gallons` to be sent (passed) to `toLiters`, where a computation is done, and a value is returned. In the `toLiters` method definition the value that is passed is referred to as `amount`. In the next section we will talk in detail about passing values in this manner.

8. This idea of a method taking in a value, using the value in a computation, and returning a single result of a computation is similar to the mathematical idea of a *function*, which is why a method that returns a value is often referred to as a function.



FOUND AT METHOD CALL

B. Value Arguments and Returning Values

1. The word *parameter* is used to describe variables that pass information within a program. The simple process in Program 7-2 of passing a number (of gallons) to a method that will compute something (number of liters) is representative of a common occurrence in Java programming – passing values with parameters. Sometimes the word "argument" is used in place of parameter. This is NOT arbitrary!! It helps with location within code.

AT METHOD DEFINITION

2. A value argument has the following characteristics:

   a. This argument is a local variable. This means that it is valid only inside the block (method) in which it is declared.
   b. It receives a copy of the parameter that was passed to the method. The value of 10 stored in `gallons` (inside `main`) is passed to the argument `amount` (inside `toLiters`).
   c. The argument is the variable that can be modified within the method.
   d. In other words, you pass parameters and accept arguments.
   e. In this case where an argument accepts a copy of the value in a parameter which was passed we title the action "call by value".

3. In order for a method to return a value, there must be a `return` statement somewhere in the body of the method.

4. If a method returns no value the term `void` should be used. For example:

```
public void printHello( )
{
   System.out.println("Hello world");
}
```

5. A method can have multiple arguments in its argument list. For example:

```
public double doMath(int a, double x)
{
   ... code ...
   return doubleVal;
}
```

When this method is called, the parameters fed to the `doMath` method must be of an appropriate type. The first parameter must be an integer. The second parameter can be an integer because it will be promoted to a double. Parameters can't be demoted!!

```
double dbl = doMath(2, 3.5);      //  this is okay
double dbl = doMath(2, 3);        //  this is okay
double dbl = doMath(1.05, 6.37);  //  this will not compile
```

NO! A Double CANT BE DEMOTED TO AN int

6. (Call by) Value arguments are sometimes described as one-way parameters. The information flows into a method but no information is passed back through the value arguments. A single value can be passed back using the return statement, but the parameters in the method call remain unchanged.

7. The parameters used to supply values for the value arguments can be either literal values `(2, 3.5)` or variables `(a, x)`.

```
double dbl = doMath(a, x);     // example using variables
```

OR CONSTANTS, OR THE RESULT OF ANOTHER METHOD OK

C. The Signature of a Method

1. In order to call a method legally, you need to know its name, you need to know how many arguments it has, and you need to know the type of each argument. This information is called the method's *signature*. The signature of the method `doMath` can be expressed as: `doMath(int, double)`. Note that the signature does not include the names of the arguments; in fact, if you just want to use the method, you don't even need to know what the argument names are, so the names are not part of the signature.

2. Java allows two different methods in the same class to have the same name, provided that their **signatures** are different. We say that the name of the method is *overloaded* because it has several different meanings. The computer doesn't get the methods mixed up. It can tell which one you want to call by the number and types of the parameters that you pass in the subroutine call statement. You have already seen overloading used in the `System.out` class. This class includes many different methods named `println`, for example. These methods all have different signatures, such as:

```
println(int)      println(double)     println(String)
println(char)     println(boolean)    println()
```

UNIQUE SIGNATURES (REQUIRE INDIVIDUAL DEFINITIONS)

3. The signature does not include the method's return type. It is **illegal** to have two methods in the same class that have the same signature but that have different return types. For example, it would be a syntax error for a class to contain two methods defined as:

```
double dbl = doMath(int, double);
int dbl = doMath(int, double);
```

This breaks the idea of a mathematical function, no?

THE STORY OF TIM THE ENCHANTER

D. Lifetime, Initialization, and Scope of Variables

1. Three categories of Java variables have been explained thus far in this curriculum guide.
   - Instance variables
   - Local variables
   - Argument variables

PREVIOUS LESSONS

2. The lifetime of a variable defines the portion of run time during which the variable exists and is accessible.

JVM HAS AUTOMATIC GARBAGE COLLECT

   a. When an object is constructed, all its instance variables are created. As long as any part of the program can access the object, it stays alive.

BLOCK BY BLOCK

   b. A local variable is created when the program enters the statement that defines it. It stays alive until the block (of delimiters) that encloses the variable definition is exited.

TIM LEAVES THE ISLAND

   c. When a method is called, its argument variables are created. They stay alive until the method returns to the caller.

3. The initial state of a variable is also determined by its type.

   a. Instance variables (associated with a particular object) and static variables (associated with a particular class) are automatically initialized with a default value (0 for numbers, **`false`** for **`boolean`**, `null` for objects) unless you specify another parameter.
   b. Argument variables are initialized with copies of the parameters data.
   c. Local variables are not initialized by default. An initial value must be supplied. The compiler will generate an error if an attempt is made to use a local variable that has never been initialized.

4. Scope refers to the area of a program in which an identifier is valid and has meaning.

   a. Instance variables of a class are usually declared **`private`**, and have class scope. Class scope begins at the opening left delimiter, `{`, of the class definition and terminates at the closing delimiter, `}`, of the class definition. Class scope enables methods of a class to directly access all instance variables defined in the class.

CLASS SCOPE

DECLARE THESE AT THE TOP OF THE BLOCK

   b. The scope of a local variable extends from the point of its definition to the end of the enclosing block

BLOCK SCOPE

   c. The scope of an argument variable is the entire body of its method.

METHOD SCOPE

5. An example of the scope of a variable is given in Program 7-3. The class `ScopeTest` is created with four methods:

   - `printLocalTest`
   - `printInstanceTest`
   - `printParamTest`
   - `main`

6. The subclass `st` is created as "a kind of" `ScopeTest`, so it contains the same methods. Each of these methods contains a variable named `test`.

7. The statement `st.printLocalTest()` calls the method `printLocalTest`, and in a similar way each method is called.

8. The results show the following about the scope of the variable `test`:

   a. Within the scope of `main`, the value of test is 10, the value assigned within the `main` method.
   b. Within the scope of `printLocalTest`, the value of test is 20, the value assigned within the `printLocalTest` method
   c. Within the scope of `printInstanceTest`, the value of test is 30, the private value assigned within `ScopeTest`, because there is no value given to `test` within the `printInstanceTest` method
   d. Within the scope of `printParamTest`, the value of test is 40, the value sent to the `printParamTest` method

Program 7-3

```
public class ScopeTest
{
  private int test = 30;

  public void printLocalTest()
  {
    int test = 20;
    System.out.println("printLocalTest: test = " + test);
  }

  public void printInstanceTest()
  {
    System.out.println("printInstanceTest: test = " + test);
  }

  public void printParamTest(int test)
  {
    System.out.println("printParamTest: test = " + test);
  }

  public static void main (String[ ] args)
  {
    int test = 10;

    ScopeTest st = new ScopeTest();
    System.out.println("main: test = " + test);

    st.printLocalTest();
    st.printInstanceTest();
    st.printParamTest(40);
  }
}
```

Run output:

```
main: test = 10
printLocalTest: test = 20
printInstanceTest: test = 30
printParamTest: test = 40
```

SUMMARY/ REVIEW:

Your programs will grow in size and complexity. Initially you will not use all the tools presented in this lesson and other lessons regarding methods. However, you need to see and understand all the method-writing tools in Java since eventually you will need them in your own work and to help you read another programmer's code.

ASSIGNMENT:

Lab Exercise, L.A.7.2, *Polygon*
Lab Exercise, L.A.7.3, *RectangleMethods*