Java Basics Unit

Lesson 7: Methods

The Learning House

427 S 4th Street #300

Louisville KY 40202

Lesson 07: Methods

Overview

Do you like to repeat yourself unnecessarily? Do you like to leverage technology and/or techniques to reduce the amount of work you need to do? Have you ever been called lazy? …as a complement?

In this lesson, we'll look at methods. Methods allow us to better organize our code so that it can be used in multiple places. This, of course, means less work! Good coders are lazy coders — well, smart and lazy...

Why Methods?

One of the very important concepts in writing good code is the DRY principle — Don't Repeat Yourself. When given a task, we want to do it well and we only want to do it once. Methods allow us to write a bit of code, give it a name, and then use that code over and over again anywhere in our program. Methods give us two new tools for designing and writing solid code:

1. We can reuse code that we've written.
2. We can use methods to break down large, complex tasks into simpler steps.

Methods don't add any features to the language; we can't do anything that we couldn't before. Methods are an organizational tool. Remember, a large component of writing good software consists of good code organization.

Defining Methods

A method is defined by **method declaration** which is made up of the following parts:

<access modifier> <return type> <method name> (<parameter list>) <exception list> {

<method body>

}

1. Access modifier: for now, we'll use **public**. Other values are possible and we'll cover these later.
2. Return type: this indicates the data type of the value returned by the method; use **void** if the method does not return a value.
3. Method name: the rules for identifiers apply here but there are additional conventions (see below).
4. Parameter list: comma-delimited list of input parameters. Each parameter consists of its data type followed by its identifier. The parameter list must be enclosed in parentheses; if the method has no parameters, you must use empty parentheses.
5. Exception list: we'll cover this later.
6. Method body: the code block for this method.

Method Signature

The method signature (this is an important definition - **memorize it!**) consists of:

1. The method name
2. The parameter list

We'll talk more about method signatures when we get to the object-oriented features of Java in the next unit.

Method Naming Conventions

Technically, a method name can be any legal identifier (see Programs, Statements, and Variables). However, there are additional conventions that further restrict method names in practice:

1. Begins with lowercase letter
2. Is a verb (if a single word name)
3. Begins with a verb (if a multi-word name) followed by other words
4. Should be in camel case (i.e. the first letter of the second and following words should be capitalized)

Here are some examples:

1. calculateTotal
2. processOrders
3. storeUserData
4. checkCardValidity

Return Values

Methods can return values to the caller, but they don't have to. We have seen examples of both:

- System.out.println("Hello") does not return a value to the caller; it simply prints a value to the Console.
- The Scanner method nextLine() does return a value to the caller; it returns the string of characters that the user typed into the console. We put that value into a variable (see WindowMaster).
- Float.parseFloat("5.32") does return a value to the caller; it converts the string data type input parameter into a float data type and returns it. We put that value into a variable and then use it to perform math calculations (see WindowMaster).

Whether or not your method returns a value, you must indicate the **return type** of your method when declaring the method. If your method does not return a value, the return type is **void**; otherwise, it is the data type of whatever value the method returns:

```
public void printHiYa() {
    System.out.println("HiYa");
}

public double calculatePi() {
    return 3.14159;
}
```

Input Parameters

Methods can have zero or more input parameters. Input parameters are simply placeholders for values that will be "passed in" as data for the method to work on. Methods without parameters are useful because we can reuse the code contained in the method in many places. However, methods get really powerful when we can pass parameter values into them.

Here's a somewhat contrived example: say we wanted to write some reusable code that added two numbers together. Further, let's assume that we don't know how to use parameters. If this were the case, we'd have to write a separate method for each number pair we wanted to add:

```
public int add1And1() {
    return 1 + 1;
}

public int add1And2() {
    return 1 + 2;
}

public int add1And3() {
    return 1 + 3;
}

And so on...
```

This is not a good situation — in order to be able to add any two numbers, we would have to write an infinite number of these methods.

Each equation applies to only one situation; while the equation is true, it is not general:

1 + 1 = 2

1 + 2 = 3

1 + 3 = 4

And so on…

What we really want is the general equation for adding two numbers together:

a + b = y

We want to be able to supply the values for a and b and have the method calculate y for us. We'll use input parameters to help us here:

```
public int add(int a, int b) {
    return a + b;
}
```

There — now we have a method that will take any two numbers and will calculate the sum!

Method Forms

Given the rules discussed above, there are four forms that a method can take:

1. No return value, no parameters
2. Return value, no parameters
3. No return value, one or more parameters
4. Return value, one or more parameters

We've seen examples of most of these but we'll go over examples of each here:

No Return Value, No Parameters:

```
public static void doit() {
    System.out.println("Hello");
}
```

Return Value, No Parameters:

```
public static int get5() {
    return 5;
}
```

No Return Value, One or More Parameters:

```
public static void silly(int i) {
    System.out.println("My parameter is: " + i );
}
```

Return Value, One or More Parameters:

```
public int add(int a, int b) {
    return a + b;
}
```

System.out.println(...) IS NOT A RETURN VALUE

One fairly common point of confusion for beginning programmers is the difference between returning a value from a method and printing something to the Console. Printing to the Console IS NOT the same as returning a value from a method. Returning a value from a method requires the return keyword, as shown in the examples above.

The Static Keyword

I'm sure you have noticed that we are using the **static** keyword for our method definitions. Don't worry about the meaning right now — we'll cover it in detail when we talk about the object oriented features of Java in the next unit. For now, just remember to include it with your method definition — if you forget, your code will not compile.

Something We Can Use

Let's put all of this newfound knowledge about methods to work for us by creating a method that we can use in one of our programs.

So far, WindowMaster is looking pretty good but we are repeating ourselves in at least one place. We're going to clean that up by creating a reusable method. Look at the following code from WindowMaster:

```
36          // Get input from user
37          System.out.println("Please enter window height:");
38          stringHeight = sc.nextLine();
39          System.out.println("Please enter window width:");
40          stringWidth = sc.nextLine();
41
42          // Convert String values of height and width to floats
43          height = Float.parseFloat(stringHeight);
44          width = Float.parseFloat(stringWidth);
```

Figure 1: Redundant Code

Lines 37 and 38 look *very* similar to lines 39 and 40. Lines 43 and 44 are almost identical as well. This presents a great opportunity for **refactoring** this code into a reusable method. Refactoring is simply the process reorganizing and/or cleaning up your code *without adding, subtracting, or changing functionality*. After refactoring, the code still does exactly the same thing it did before, but it is now more readable and maintainable.

The first step in refactoring code into a method is to divide out the similarities and the differences of the repeated code. The similarities of the code will represent the main body of the new method. The differences can be factored out into input parameters and/or return types for the method. We'll follow a process similar to the one we followed with the **add** method in the Parameters section above.

Similarities

So, what are the similarities in our code example? For each input value that we request from the user we do the following:

1. Print a message to the Console to let the user know what type of value we're asking for.
2. Wait for the user to provide the value, read the value, and store it in a variable.
3. Convert the string value read from the Console into a float value that we can use for mathematical operations later in the program.

Differences

Now, what are the differences in the repeated code?

- The message printed to the Console is different for each value requested.
- We store the user-provided values in different variables.

The Method

Armed with this knowledge, we can now create our new method. We'll need to do the following:

1. Decide on a name.
2. Decide if the method needs input parameters. If so, how many and of what type?
3. Decide if the method needs to return a value. If so, what type?
4. Write the code for the body of the method.

Name:

For our name, let's go with **readValue**.

Parameters:

For the decision regarding input parameters, we need to look at our list of differences. What needs to vary each time we run the method? According to our list, the message changes and the variable in which we store the user value changes. So, it looks like we'll need to provide a message each time — this sounds like an input parameter. Putting the user value into a variable sounds like a return type, so we'll handle that next. That means we'll have one input parameter and the input parameter will represent the message that we want to print to the Console to prompt the user for a value. What type should this be? To answer this, we look at the existing code. In the existing code, the user prompt message is represented by a string literal that is passed to System.out.println(), which means that our input parameter should be a string as well.

Return Type:

Now for the return type... In the existing code, we store both string and float representations of the values entered by the user. If you look closely, however, you will notice that we really only care about the float representation of these values — the string representations are only needed because the Console doesn't

understand numbers. So what we really want is to prompt the user to input a float value and then get the float representation of the value — we don't really want to deal with Strings. This indicates that the return type of the method should be **float**.

Basic Definition:

This is what we have so far:

```
public static float readValue (String prompt) {
    // method body TBD
}
```

Method Body:

Now we need to make our method do something useful. Our method needs to do the following:

1. Print the provided prompt to the Console
2. Wait for the user to input a value
3. Read the value
4. Convert the value into a float data type
5. Return the converted value

We need a Scanner variable in order to read the value in from Console. You may ask: why can't we use the one in main? This has to do with **scoping rule,** which we'll cover in the next section. For now, just go with it.

```
    public static float readValue(String prompt) {
        // declare and initialize a Scanner variable
        Scanner sc = new Scanner(System.in);
        // print prompt to Console
        System.out.println(prompt);
        // read value into String data type
        String input = sc.nextLine();
        // convert the String to a float
        float floatVal = Float.parseFloat(input);
        // return the float value
        return floatVal;
    }
```

Using the Method

The final step in our refactoring adventure is to replace the repeated code in our example with calls to our new bright, shiny method. We will replace lines 37 to 44 with just two lines (each line will be a call to our new method). You'll notice also that we no longer need the variables **stringHeight** or **stringWidth,** which means we can remove the lines on which they are declared. You'll notice that removing these variables causes a compilation problem — why? Well, we are still printing out the value of stringHeight and stringWidth to the Console. Now that they are gone, we have a problem. To fix this, we'll simply print out the float values (height, width) instead. After making all of these changes, WindowMaster looks like this:

```
package windowmaster;

import java.util.Scanner;

public class WindowMaster {

    public static void main(String[] args) {
        // declare variables for height and width
        float height;
        float width;

        // declare other variables
        float areaOfWindow;
        float cost;
        float perimeterOfWindow;

        // Declare and initialize our Scanner
        Scanner sc = new Scanner(System.in);

        // Get input from user
        height = readValue("Please enter window height:");
        width = readValue("Please enter window width:");

        // calculate area of window
        areaOfWindow = height * width;

        // calculate the perimeter of the window
        perimeterOfWindow = 2 * (height + width);

        // calculate total cost - use hard coded for material cost
        cost = ((3.50f * areaOfWindow) + (2.25f * perimeterOfWindow));

        System.out.println("Window height = " + height);
        System.out.println("Window width = " + width);
        System.out.println("Window area = " + areaOfWindow);
        System.out.println("Window perimeter = " + perimeterOfWindow);
        System.out.println("Total Cost = " + cost);
    }

    public static float readValue(String prompt) {
        // declare and initialize a Scanner variable
        Scanner sc = new Scanner(System.in);
        // print prompt to Console
        System.out.println(prompt);
        // read value into String data type
        String input = sc.nextLine();
        // convert the String to a float
        float floatVal = Float.parseFloat(input);
        // return the float value
        return floatVal;
    }
}
```

Scope

Now, back to the questions regarding scope… Looking at our code, we have a variable called sc of type Scanner that is declared and initialized in the main method. We also have a variable called sc of type Scanner that is declared and initialed in the readValue method. This raises a couple of questions:

1. Why do we need both? Do we need both?
2. How can we have two different variables with the same name? How can the compiler tell them apart?

The answers to these questions have to do with the concept of scope. As we've seen, a Java program consists of blocks of code. These blocks are marked by curly braces { }. We have also seen that these blocks can be nested; for example, the WindowMaster class block contains both the main method block and the readValue method block. Method blocks can contain other blocks of code such as if statement blocks, while loop blocks, etc.

The Java language allows us to define variables inside a block of code. These variables are said to be **local** to that block of code. Variables declared in outer blocks can be accessed by code inside nested code blocks but the reverse is not true. Code in outer blocks cannot "see into" nested blocks. Here are a couple of examples:

```
public static void main(String[] args) {
    int age = 42;

    for (int i = 0; i < 5; i++) {
        // this is ok - the nested block can access the variables
        // in the outer block
        System.out.println(age);
    }

    if (age < 18) {
        // this is ok - the nested block can access the variables
        // in the outer block
        int yearsToWait = 18 - age;
    }

    // NOT ok - outer block cannot access variables declared inside
    // inner blocks
    System.out.println(yearsToWait);
}
```

Wrap-up

In this lesson, we learned all about methods. Here's what we covered:

1. Why methods are important
2. How to define methods
3. What a **method signature** is
4. The different forms a method can take
5. How to create a useful method by refactoring code
6. Java scoping rules

In the next lesson, we'll cover arrays.