Chapter 5: Methods

October 29, 2019

Introduction to Methods

Why Write Methods?

- Methods are commonly used to break a problem down into small manageable pieces.
- Methods simplify programs. If a specific task is performed in several places in the program, a method can be written once to perform that task, and then be executed anytime it is needed. This is known as code reuse.

Example

Imagine that we have a method named `printMoe()` that simply displays “Moe” to the screen. We could then do this:

```
System.out.println("Larry");
printMoe();
System.out.println("Curly");
```

This would print to the screen the names Larry, Moe, and Curly in that order. If we never called `printMoe()`, then the code would never be used.
A **method** is a segment of code that is called as needed.

Static Methods and Non-static methods

Static and Non-Static

- In Java, you have the choice to not associate a method with an object. This is called “static”.
  - The keyword for a static method is **static**.
  - The main method in Java is `static`. `main` is not associated with any object.
- By default, all methods are associated with an object. This is called “non-static”.
  - There is no keyword to designate a method as “non-static”. Non-static is the default.

Public, Private, and Protected Methods

- Each method should be individually labeled as `public` or `private`.
- You can order the methods however you like.
- Meanings:
  - **public** means that this method can be called from anywhere.
  - **private** means that this method can be called only from within the same class.

void Methods and Value-Returning Methods

A **void** method is one that simply performs a task and then terminates. Remember the term **void**.

```
System.out.println("Hi!");
```

A value-returning method not only performs a task, but also sends a value back to the code that called it.

```
int number = Integer.parseInt("700");
```

Defining a void Method

Void methods are easier to write, so we'll start with them.

- To create a **void** method, you must write a definition, which consists of a header and a body.
- The method header, which appears at the beginning of a method definition, lists several important things about the method, including the method's name.
- The method body is a collection of statements that are performed when the method is executed.

Two Parts of Method Declaration

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

Parts of a Method Header

- Method modifiers
  - public: method is publicly available to code outside the class
  - static: method belongs to a class, not a specific object.
- Return type: void or the data type from a value-returning method
- Method name: name that is descriptive of what the method does
- Parentheses: contain nothing or a list of one or more variable declarations if the method is capable of receiving arguments.

Calling a Method

- A method executes when it is called.
- The main method is automatically called when a program starts, but other methods are executed by method call statements.
- There is no “starting object” in Java. Thus, the main method must be **static**.
  - **static** means that a method isn’t associated with an object.

Code.

```
displayMessage();
```

Notice that the method modifiers and the void return type are not written in the method call statement. Those are only written in the method header.

Static and Non-Static

The terms “static” and “non-static” can be confusing. If you are still stuck on this, think of a car. Now think of your car.

- All cars have four wheels. (static)
- My car is blue. (non-static)
- All cars have a front windshield. (static)
- My car seats five people. (non-static)
- All cars have side-view mirrors. (static)
- My car gets 31 miles to the gallon. (non-static)

Static and Non-Static

- The difference between static and non-static is the same as all cars and my car.
  - All cars have four wheels is an example of something being “static”.
  - My car is blue is an example of something being “non-static”.
- If a method is static, that means it can be understood without an object.
- If a method is non-static, that means you have to have an object.
- Asking the question, “What is the color of all cars?” is nonsensical. There are many different color cars.
- For the same reasons, calling a non-static method without an object will generate the error message “Cannot call non-static method from a static context.”
- Java treats all methods as non-static by default. For now, we need to explicitly state that methods are static.

Documenting Methods

Documenting Methods

- A method should always be documented by writing comments that appear before the method's definition.
- The comments should provide a brief explanation of the method's purpose.
- The documentation comments begin with /** and end with */.
  - This is known as the **javadoc** standard.

Documenting Methods

There are five parts of a method's documentation. Depending on the features of a method, not all parts will be used.

- Summary: This is a description of the method.
- Precondition: This is what is required for this method to execute without throwing any errors. For example, "The file must exist" in a method which reads a file.
- Postcondition: This is what changes about the arguments or object when a method is called.
- Parameter description: A description of each parameter. If a method has no parameters, then you don't need this.
- Return description: A description of what is returned by the method. If this is a `void` method, then you don't need this.

Documentation Example

```
/**
 * Displays the word "Hello"
 * @precondition None.
 * @postcondition "Hello" is written to the screen.
 */
public static void displayMesssage() {
    System.out.println("Hello");
}
```

Since there are no parameters and return, we only need three parts of the documentation.

Passing Arguments to a Method

Passing Arguments to a Method

Values that are sent into a method are called arguments.

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

number = Integer.parseInt(str);
```

- The data type of an argument in a method call must correspond to the variable declaration in the parentheses of the method declaration. The parameter is the variable that holds the value being passed into a method.
- By using parameter variables in your method declarations, you can design your own methods that accept data this way.

Passing 5 to the displayValue Method

```
displayValue(5);

public static void displayValue(int num) {
    System.out.println("The value is " + num);
}
```

Argument and Parameter Data Type Compatibility

- When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type.
- Java will automatically perform widening conversions, but narrowing conversions will cause a compiler error.

Code.

```
double d = 1.0;
displayValue(d);
```

Passing Multiple Arguments

```
showSum(5, 10);

public static void showSum(double num1, double num2) {
    double sum;        //to hold the sum
    sum = num1 + num2;
    System.out.println("The sum is " + sum);
}
```

Arguments are Passed depending on type.

- In Java, all primitive data types are passed by value.
  - This means that the data is copied into new memory value location upon calling a method.
  - A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.
  - If a parameter variable is changed inside a method, it has no affect on the original argument.

Passing Object References to a Method

- In Java, all objects are passed by reference.
  - Recall that a class type variable does not hold the actual data item that is associated with it, but holds the memory address of the object. A variable associated with an object is called a reference variable.
  - Passing by reference means that data is **not** copied. If you change the data of an object inside of a method, then that change remains after the object is finished.
  - When an object such as a String is passed as an argument, it is a **reference** to the object that is passed.

Java’s approach is much better.

- Primitive variables are all small and will pass-by-value. This is what you want.
    - “Pass-by-value” means that the value is passed directly to the method. This is good for small things but bad for large things.
- Objects are always declared dynamically and will always pass-by-reference, which is faster for larger variables. This is what you want.
    - “Pass-by-reference” means that values are passed by a memory address. This is bad for small things but good for large things.
- There is no special syntax to remember. The default behavior of Java is almost always the best behavior. This is what you want.

Passing a Reference as an Argument

```
showLength(name);

public static void showLength(String str) {
    System.out.println(str + " is " + str.length()
             + " characters long.");
    str = "Joe"; // see next slide
}
```

Strings are Immutable Objects

Strings are immutable objects, which means that they cannot be changed. When the line

```
str = "Joe";
```

is executed, it cannot change an immutable object, so creates a new object.

@param Tag in Documentation Comments

- You can provide a description of each parameter in your documentation comments by using the @param tag.

General format

```
@param parameterName Description
```

All @param tags in a method's documentation comment must appear after the general description.The description can span several lines.

Documentation Example

```
/**
 * Displays the length of a String to the screen.
 * @precondition none
 * @postcondition Screen output.
 * @param str the String which will be studied.
 */
public static void showLength(String str) {
    System.out.println(str + " is " + str.length()
             + " characters long.");
    str = "Joe";
}
```

More About Local Variables

More About Local Variables

- A local variable is declared inside a method and is not accessible to statements outside the method.
- Different methods can have local variables with the same names because the methods cannot see each other's local variables.
- A method's local variables exist only while the method is executing. When the method ends, the local variables and parameter variables are destroyed and any values stored are lost.
- Local variables are not automatically initialized with a default value and must be given a value before they can be used.

Returning a Value from a Method

Returning a Value from a Method

Data can be passed into a method by way of the parameter variables. Data may also be returned from a method, back to the statement that called it.

```
int num = Integer.parseInt("700");
```

- The string “700” is passed into the parseInt method.
- The int value 700 is returned from the method and assigned to the num variable.

Defining a Value-Returning Method

```
public static int sum(int num1, int num2) {
    int result;
    result = num1 + num2;
    return result;
}
```

Calling a Value-Returning Method

```
total = sum(value1, value2);

public static int sum(int num1, int num2) {
    int result;
    result = num1 + num2;
    return result;
}
```

The Return Tag

You can provide a description of the return value in your documentation comments by using the @return tag.

```
@return Description
```

The @return tag in a method's documentation comment must appear after the general description. The description can span several lines.

Returning a booleanValue

Sometimes we need to write methods to test arguments for validity and return true or false

```
public static boolean isValid(int number) {
   boolean status;
   if(number >= 1 && number <= 100) {
      status = true;
   }
   else {
      status = false;
   }
   return status;
}
```

Calling code

```
int value = 20;
if(isValid(value)) {
    System.out.println("The value is within range");
}
else {
    System.out.println("The value is out of range");
}
```

Returning a Reference to a String Object

```
customerName = fullName("John", "Martin");
public static String fullName(String first, String last) {
        String name;
        name = first + " " + last;
        return name;
}
```

Problem Solving with Methods

Problem Solving with Methods

- A large, complex problem can be solved a piece at a time by methods.
- The process of breaking a problem down into smaller pieces is called functional decomposition.
- If a method calls another method that has a throws clause in its header, then the calling method should have the same throws clause.

Calling Methods that Throw Exceptions

- Note that the main and getTotalSales methods in SalesReport.java have a throws IOException clause.
- All methods that use a Scanner object to open a file must throw or handle IOException.
- You will learn how to handle exceptions in Chapter 12.
- For now, understand that Java required any method that interacts with an external entity, such as the file system to either throw an exception to be handles elsewhere in your application or to handle the exception locally.