Chapter 5 - Methods

Last updated: Wednesday November 27, 2024

Chapter 5 discusses the following main topics:

- Introduction to Methods
- Passing Arguments to a Method
- More About Local Variables
- Returning a Value from a Method
- Problem Solving with Methods

Notes and Clarifications

Sometimes I don't get things right on the first go-around. I'm only human!

Chapter 5 Source Code files in IntelliJ

There are some errors in the Chapter 5 Source Code project. We might fix them as part of the lessons, but for now:

1. Delete `TwoArgs2.java` and `ValueReturn.java`.
2. Comment out the following lines in AreaRectangle.java.

```
//length = getLength();
//width = getWidth();
//area = getArea(length, width);
//displayData(length, width, area);
```

Parameters vs. Arguments

Classes have methods. Methods can take input parameters, but they are not required! They are written in the parentheses after the method name. Multiple parameters are separated by a comma.

> Parameters go in parentheses!

Here are some examples:

```java
String s = "Sam is cool";
/*
The String.equals() method takes one input parameter:
a String "Sarah"
It returns True or False.
*/
s.equals("Sarah");

/*
The String.replace() method takes two input parameters:
A string to find ("Sam"), and the string to replace it with ("Sarah").
It returns "Sarah is cool"
*/
s.replace("Sam", "Sarah");

/*
The String.length() method takes zero input parameters.
It returns 11.
*/
s.length();

/*
The System.out.println() method takes one input parameter.
The System.out.printf() method takes additional arguments,
depending on how many placeholders are included in the string.
In the following example, there are 2 placeholders,
so it requires two additional parameters.
*/
System.out.println("This is an input parameter");
System.out.printf("This %s string is number %d %n", "formatted", 1);


Random randomNumber = new Random();
/*
Random.nextInt() without parameters returns
a random number in between -2,147,483,648 to +2,147,483,648.
*/
randomNumber.nextInt();

/*
Random.nextInt() with one parameter returns
a random number between 0 and the number.
*/
randomNumber.nextInt(100);

/*
```

```
Random.nextInt() with two parameters returns
a random number between the first and the second number.
*/
randomNumber.nextInt(1, 6);
```

Parameters make methods more flexible and reusable, so you can adapt the method to various situations (I'm not stuck driving a Ford Escape!).

Similar to the difference between Class and Object (Blueprint vs. Actual Thing), a Parameter is part of the blueprint while the Argument is the actual value:

- **Parameter**: A variable in the method definition that accepts the value passed to the method. It’s like a placeholder.
  `Random.nextInt(int start, int boundary)`
- **Argument**: The actual value that is passed to the method when it is called.
  `randomNumber.nextInt(1, 6)`

In the above example, `Random.nextInt(int bound)` takes one parameter to determine a boundary for the random integer.

The argument I gave `randomNumber.nextInt(100)` says that my Random object should produce a number with a boundary of 0 - 100.

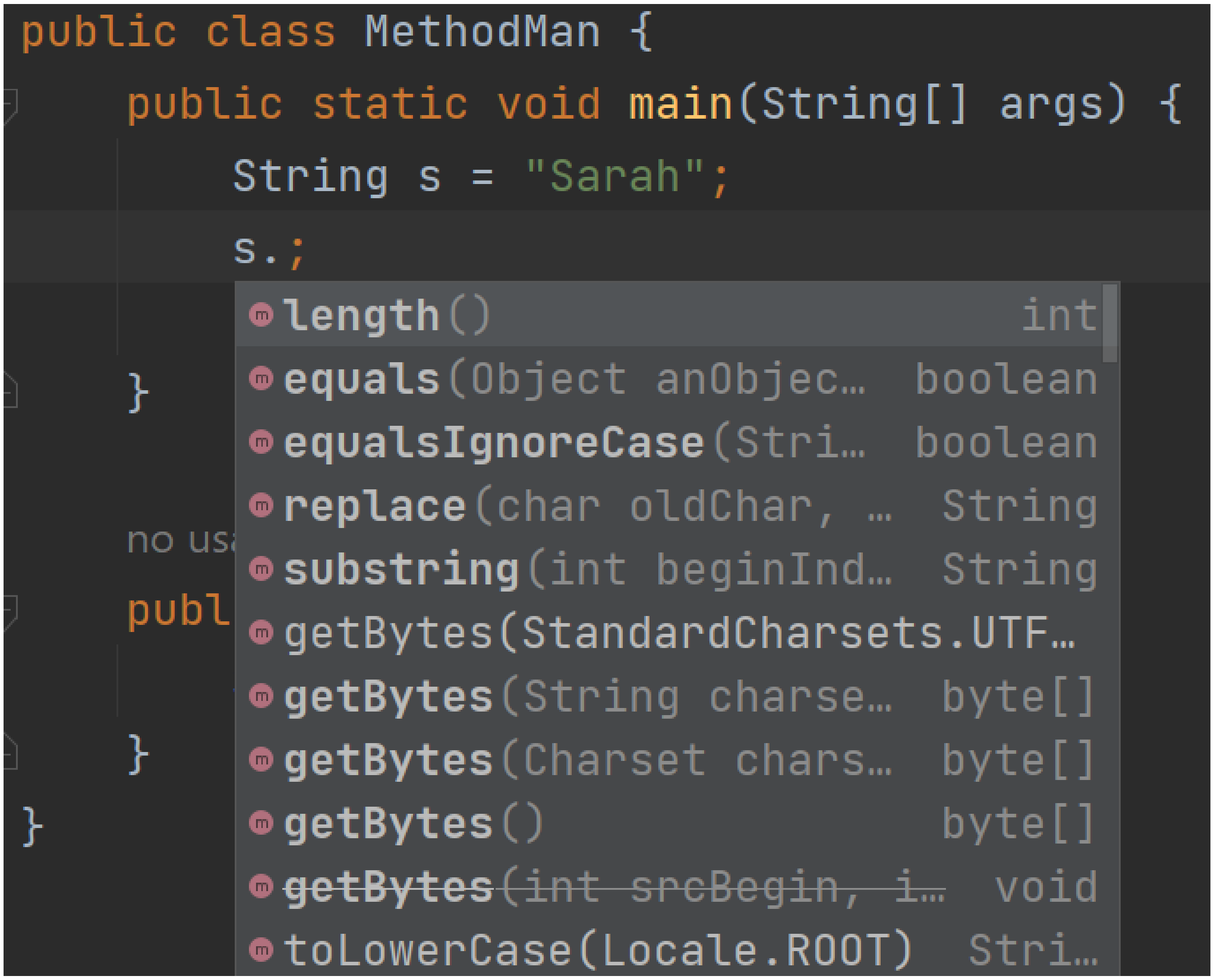
What are Methods?

Methods are reusable pieces of code that allow you to perform a task on an object (a class). In other languages, `methods` are called `functions`.

IntelliJ brings up a list of available methods when you type a period after an object. For example:

```
String s = "Sarah";
s.
```

Typing the period lists the String methods available to this variable:



```
public class MethodMan {
    public static void main(String[] args) {
        String s = "Sarah";
        s.;
    }

    no us
    publ

    }
}
```

```
length()                          int
equals(Object anObjec…        boolean
equalsIgnoreCase(Stri…        boolean
replace(char oldChar, …        String
substring(int beginInd…        String
getBytes(StandardCharsets.UTF…
getBytes(String charse…        byte[]
getBytes(Charset chars…        byte[]
getBytes()                     byte[]
~~getBytes(int srcBegin, i…~~    void
toLowerCase(Locale.ROOT)        Stri…
```

An important thing to understand is that the right side of that list contains the **RETURN TYPE**. The return type can be a primitive (boolean), an object (String), a custom object (a class that YOU write yourself), or void (nothing).

So those are Java methods given to us by Java. But we can write our OWN methods (we can also write our own classes, but that comes in the next chapter).

We all remember that **CLASSES HAVE METHODS**. String has a `length()` method, and a comparison method called `equals(String comparisonString)`.

At this point, we're not worried about defining our own classes. In Chapter 5, we are writing methods inside the same class.

There two ways of accessing a method in the same class:

- **Static method call** -> called on the class itself without needing an instance.
    - Method defined with `static`.

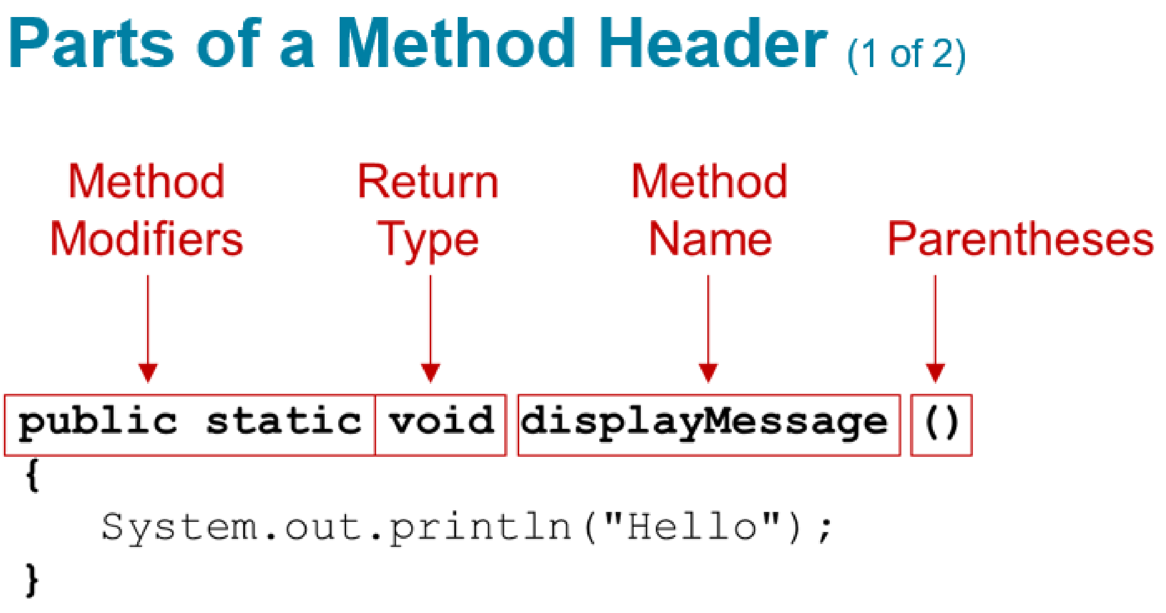
    - Doesn't require local copy of object: `Math.random()`.
- **Instance method call** -> called on an instance of the class.
    - Method not defined with `static`.
    - Requires local copy of object: `Random myRandom = new Random()`.

Method Declaration

Methods have two parts: Header and Body.

```
// Method header
public static void displayMessage(){
        // Method body
        System.out.println("Hello");
}
```

All the parts of the header make up the **Method Signature**.


Parts of a Method Header (1 of 2)

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

Method Modifiers

(slides 7 and 8)
Method modifiers are the most mysterious members of the method header. I feel they are poorly understood because they are poorly explained!

There are two types of modifiers: Access and Non-Access.

Access Modifiers

- **public**: The method is accessible from any other class.
- **protected**: The method is accessible within its own package and by subclasses.
- **default** (no modifier): The method is accessible only within its own package.
- **private**: The method is accessible only within its own class.

Non-Access Modifiers

- **static**: The method belongs to the class rather than any instance. It can be called without creating an instance of the class.
- **final**: The method cannot be overridden by subclasses.
- **abstract**: The method does not have a body and must be implemented by subclasses. This is used in abstract classes.
- **synchronized**: The method can be accessed by only one thread at a time.
- **native**: The method is implemented in native code using JNI (Java Native Interface).
- **strictfp**: The method adheres to strict floating-point calculations.

Obviously, you shouldn't worry about most of those. In fact, the most important modifier to learn is `static`, and the access modifier are also important to know.

Static vs. Instance methods

Take note of the difference between the Static method `staticMethod()` and the `instanceMethod()`.

1. In the method signature, only `staticMethod()` uses the `static` modifier
2. When calling the `staticMethod()`, we just call it.
3. When calling the `instanceMethod()`, we instantiate the object with this line: `MyClass myObject = new MyClass()`, then we call the method on `myObject`.

```java
public class MyClass {
    // Static method
    public static void staticMethod() {
        System.out.println("Static method called");
    }

    // Instance method
    public void instanceMethod() {
        System.out.println("Instance method called");
    }

    public static void main(String[] args) {
        // Calling static method within the same class
        staticMethod();

        // Creating an instance of MyClass
        MyClass myObject = new MyClass();
        // Calling instance method on the created object
        myObject.instanceMethod();
    }
}
```

Differences between `static` and instance From the Java Language Specification:

> A method that is declared `static` is called a *class method*.

> A class method is **always invoked without reference to a particular object**. The declaration of a class method introduces a static context), which limits the use of constructs that refer to the current object. Notably, the keywords `this` and `super` are prohibited in a static context, as are unqualified references to instance variables, instance methods, and type parameters of lexically enclosing declarations.

> A method that is not declared `static` is called an *instance method*, and sometimes called a non-`static` method.

> An instance method is **always invoked with respect to an object**, which becomes the current object to which the keywords `this` and `super` refer during execution of the method body.

In Java, a commonly used static method is:

`Math.max(double a, double b)`

This static method has no owning object and does not run on an instance. It receives all information from its arguments

Why 'static'?

Static methods are called "static" because they are resolved at compile time based on the class they are called on and not dynamically as in the case with instance methods, which are resolved based on the runtime type of the object.

Method Return Type

The return type can be:

- a primitive (boolean)

- an object (String)
- a custom object (a class that YOU write yourself.. example: Car)
- void (nothing).

To declare the type, write it after any Method Modifier.

There can only be one return type. But you can write a custom class that combined different primitives and classes.

Calling a Method

(Slide 9)
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.

```
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.

Examples: SimpleMethod.java, LoopCall.java, CreditCard.java, DeepAndDeeper.java

No Return (Void) Method Examples

Simplest Method Example

The simplest method would be one that doesn't take any parameters nor return anything. One use case for this would be to simply print something to the console, like a program header or maybe even some cool ASCII art.

A method is `void` if it doesn't return anything. Void methods are usually used for displaying information. Methods that also don't take any parameters are standalone methods. Nothing goes into them, so data isn't changed. Here's an example:

```
public class MethodMan {
        public static void main(String[] args) {
                printHello();
        }

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

This method is not flexible at all. It only does one thing, and it's not that interesting (unless of course you're printing the grim reaper).

Next Simplest Method Example (One input parameter)

(Slide 11-12)
If we add one element to our simple method -- an **input parameter** -- we gain flexibility. The method can do different things depending on what we provide for input. Here's an example:

```java
public class MethodMan {
        public static void main(String[] args) {
                printHello("Mike");
                printHello("Barry");
                printHello("Lisa");
        }

        public static void printHello(String name){
                System.out.println("Hello " + name);
        }
}
```

We have redefined our method to take one input parameter -- a String that we call `name`. Parameter names are like variables. We come up with our own descriptive name for them. We define the parameter name in the method header, and we use it in the method body.

(Slide 13)
What's also important about input parameters is the data type. In the above example, we say that `name` is a String. When we call the method from main, we provide a String. You must provide data of the correct type when calling a method. Here's another example with a different data type.

```java
public class MethodMan {
        public static void main(String[] args) {
                printHello(1);
                printHello(6);
                printHello(99);
        }

        public static void printHello(int number){
                System.out.println("Hello, you are my #" + number + " favorite
student");
        }
}
```

Finally, our method is still void. It still doesn't return any data, it merely prints data to the console.

Methods with multiple input parameters

(Slide 14)
The final thing to mention about void methods is that they can take multiple input parameters.

```
public class MethodMan {
        public static void main(String[] args) {
                printStudentInfo("Sally Smith", 3.75);
                printStudentInfo("Richard Wright", 4.0);
                printStudentInfo("Michael Mantia", 2.4);
        }

        public static void printStudentInfo(String name, double gpa){
                System.out.printf("%20s %d %n", name, gpa);
        }
}
```

You simply private the parameter types and names in a comma-separated list in the method header. Then when you call the method, provide your data in the correct format (and correct order!).

Return a Value from a Method

(Slide 21-27)
See example: ReturnString.java

```
public static void main(String[] args) {
        for (int i = 0; i < 20; i++){
        int result = squareNumber(i);
        System.out.println(i + " squared is " + result);
        }
}

public static int squareNumber(int number){
        return number * number;
}
```

Passing Arguments by Value vs. by Reference

(Slide 15 - 17)
In Java, all arguments are passed by value, but this can be a bit confusing because of how Java handles objects and primitives.

1. **Primitives**: When you pass a primitive type (like `int`, `double`, etc.), Java passes a **copy of the value**. Changes to the parameter inside the method do not affect the original value.

   ```java
   public static void main(String[] args) {
       int num = 5;
       modifyPrimitive(num);
       System.out.println(num); // Output: 5
   }

   static void modifyPrimitive(int n) {
       n = 10;
   }
   ```

2. **Objects**: When you pass an object, Java passes a copy of the reference to the object. This means that while the reference itself is passed by value, the object it points to can be modified within the method.

   ```java
   public static void main(String[] args) {
       MyObject obj = new MyObject();
       obj.value = 5;
       modifyObject(obj);
       System.out.println(obj.value); // Output: 10
   }

   static void modifyObject(MyObject o) {
       o.value = 10;
   }

   static class MyObject {
       int value;
   }
   ```

In summary, Java always passes arguments by value, but for objects, the value is the reference to the object, allowing the method to modify the object's fields.

```java
public static void main(String[] args) {
int myNum = 5;
myNum = modifyPrimitive(myNum);
System.out.println(myNum);
}
static int modifyPrimitive(int n) {
n = 10;
return n;
}
```

Passing Argument by Value

In Java, all arguments of the primitive data types are passed by value, which means that o**nly a copy of an argument's value** is passed into a parameter variable.

A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.

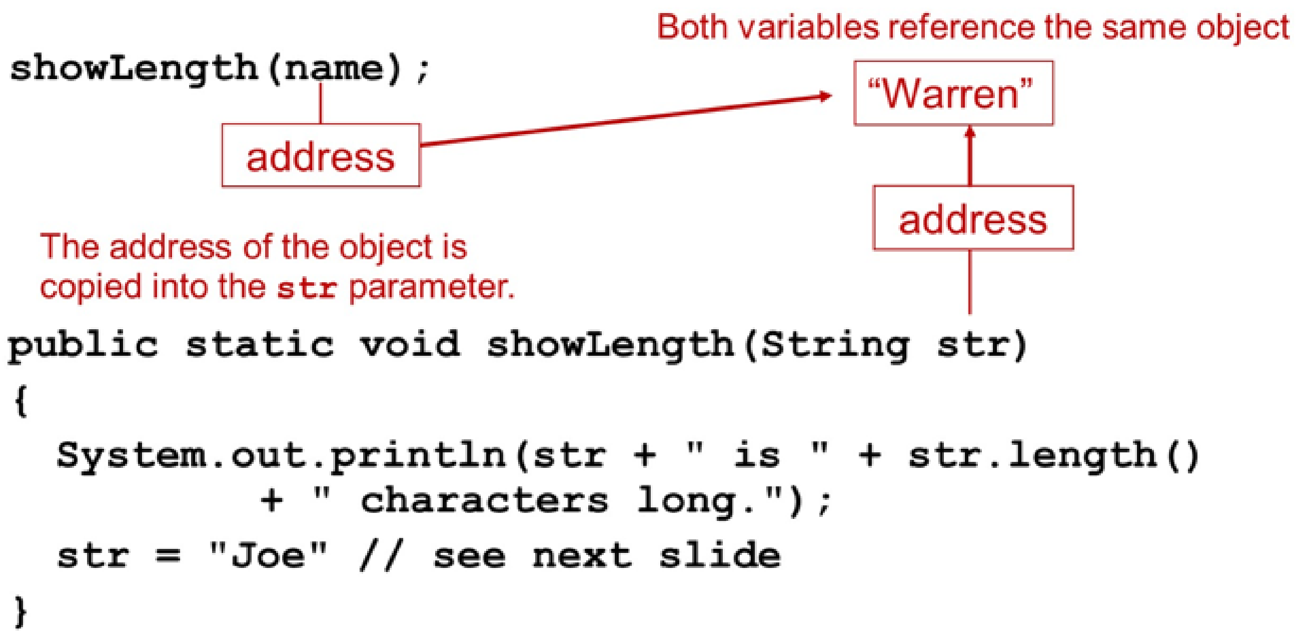
If a parameter variable is changed inside a method, it has no affect on the original argument.

See example: PassByValue.java

Passing Object References to a Method

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.

When an object such as a String is passed as an argument, it is actually a reference to the object that is passed.


Both variables reference the same object
"Warren"
address
address
The address of the object is copied into the str parameter.

```
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

(Slide 18)
When we say "Strings are immutable objects" in Java, it means that once a `String` object is created, its value cannot be changed. Any operation that seems to modify a `String` actually creates a new `String` object with the modified value, leaving the original `String` unchanged.

```java
public class Main {
    public static void main(String[] args) {
        String str = "Hello";
        str = str.concat(" World");
        System.out.println(str); // Output: Hello World
    }
}
```

In this example, the `concat` method does not change the original `String` "Hello". Instead, it creates a new `String` "Hello World" and assigns it to `str`.

You can reassign a `String` variable to a new `String` as long as the variable is not declared as `final`. However, this reassignment does not change the original `String` object; it simply points the variable to a new `String` object.

More about Local Variables

(Slide 20)
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.

See example: LocalVars.java

Problem Solving with Methods

(Slide 28)
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.
See example: SalesReport.java

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` throw `IOException`s.

All methods that use a Scanner object to open a file must throw or handle `IOException`. Check out the notes in [Chapter 4 - Files: Exceptions] section for handling exceptions.

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.

BONUS! Print Grim Reaper ASCII Art Method

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

public static void printGrimReaper(){
        System.out.println("""
```

```
        """);
}
```