Chapter 5 - Methods

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);
```

Classes vs. Objects

In lectures, I use the terms "Class" and "Object" interchangeably. I ask, "Can someone name a Java Object?" and say "Classes have methods", but I am referring to the same set of things.

Java Objects or Classes that we know include String, Scanner, File, PrintWriter, FileWriter, Random, Math, System, and (a bit of) the "wrapper" classes: Integer, Double, Boolean (which make Primitive Data Types behave like Java Objects).

In most cases it is okay that I use the terms interchangeably, but there is a technical difference between the two terms:

- **Class**: A blueprint or template for creating objects. It defines the properties (fields) and behaviors (methods) that the objects created from the class will have.
- **Object**: An instance of a class. It is created based on the class blueprint and has its own state and behavior as defined by the class.

The class is like a blueprint, and the object is like house we build from the blueprint. In Java, the blueprint is usually flexible (and we can write our own blueprints).

Let's say we write a Java class called Car and it had the following properties:

```
public class Car {
        String make = "Ford";
        String model = "Escape";
        int year = 2020;
        String color = "Blue";
        int numberOfDoors = 4;
}
```

If we want to use the Car object in another class, we'd import it (just like with Scanner, etc.) and then instantiate it:

```
import petes.cool.Car;
public class TestClass{
        Car petesCar = new Car();
        System.out.println("Prof's ex wife drives a " +
                                                petesCar.make + " " +
petesCar.model);
}
```

In the above example, we **instantiated** the Java class `Car`, which made a Java object that we called `petesCar`. This is our local copy of that Java Class.

In the `Car` class, we can do more work to write a class **constructor** that allows us to set the make, model, year, etc.

```
public class SampleClassesClass {
        public static void main(String[] args) {
                Car petesCar = new Car("Toyota", "Corolla");

                System.out.println("Prof drives a " +
                                                petesCar.make + " " +
petesCar.model);
```

```
                Car niasCar = new Car("Ford", "Escape");

                System.out.println("Nia drives a " +
                                                        niasCar.make + " " +
niasCar.model);
        }
}

class Car {
        String make;
        String model;
        public Car(String carObjectMake, String carObjectModel){
                make = carObjectMake;
                model = carObjectModel;
        }
}
```

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:

```
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.
- **Argument**: The actual value that is passed to the method when it is called.

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.

Exceptions vs. Errors

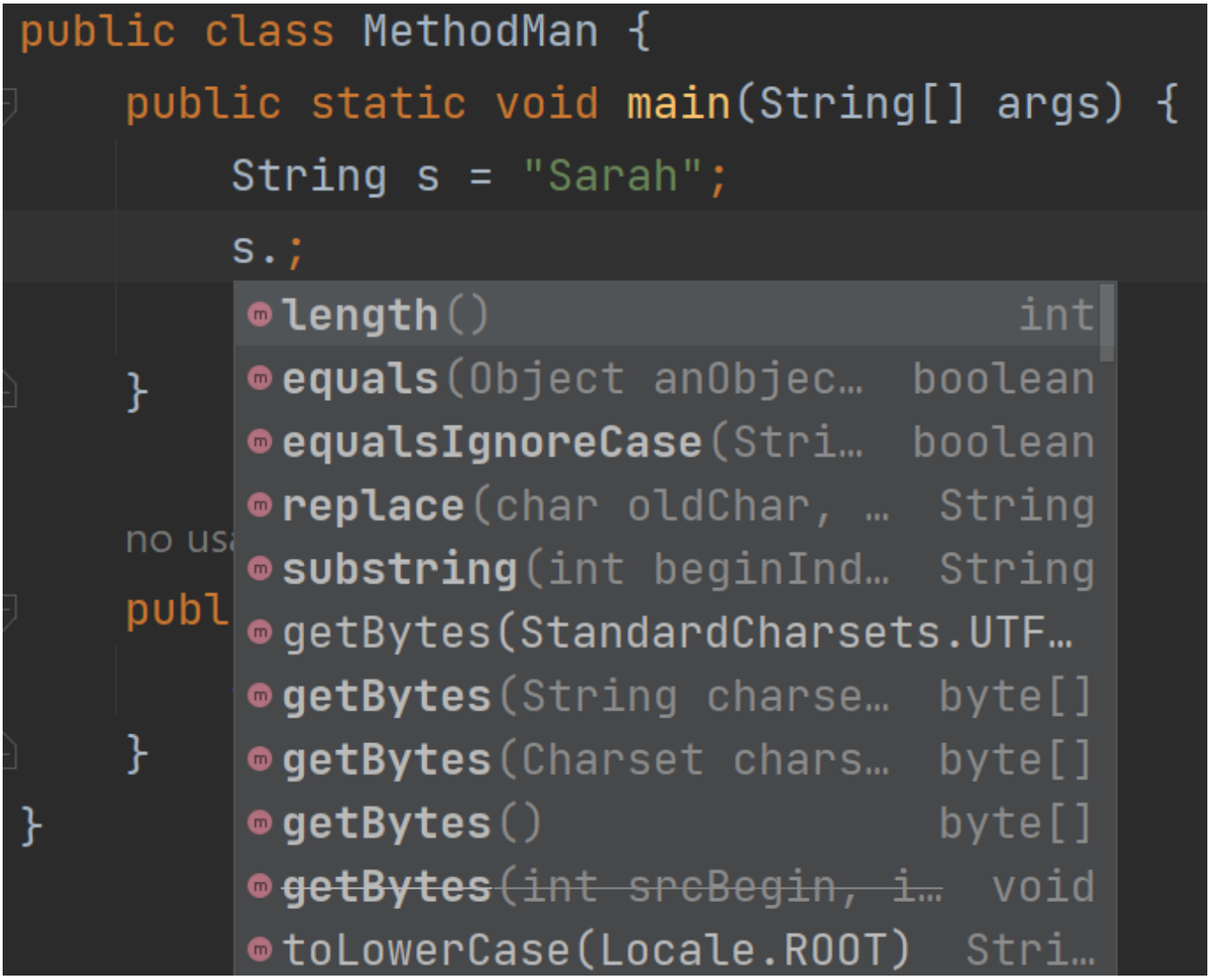
What are Methods?

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:



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:

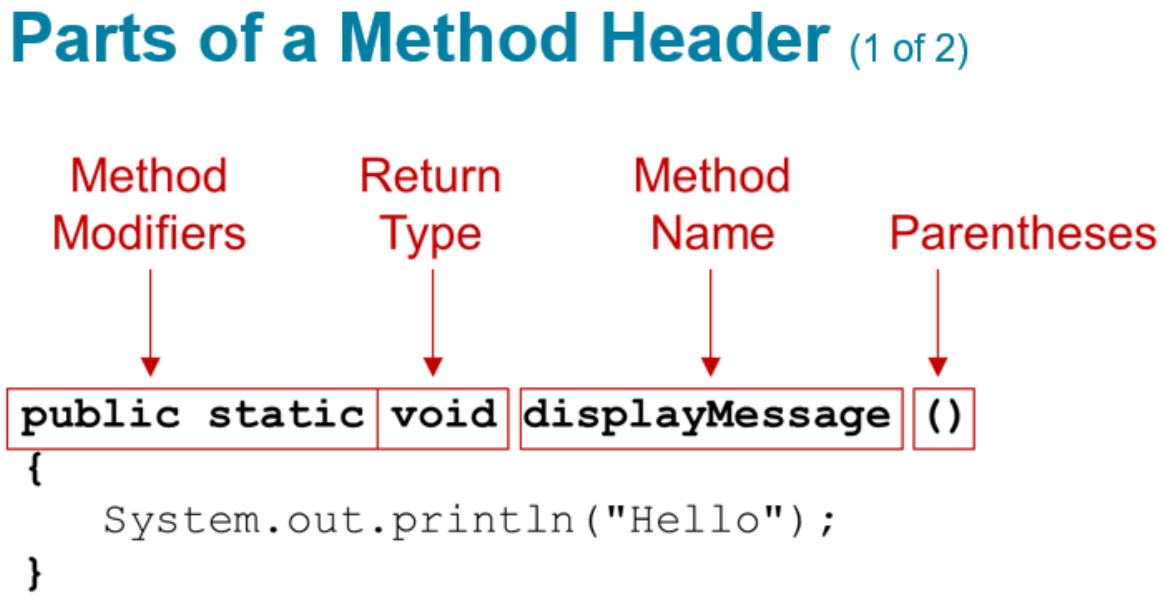
- **Static method call** -> called on the class itself without needing an instance.
  - Method defined with `static`.
  - Doesn't require local copy of object: `Math.random()`.
- **Instance method call** -> called on an instance of the class.
  - Method not defined with `static`.
  - Requires local copy 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**.



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.

Method 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).

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

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

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`.

```
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();
    }
}
```

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:

```
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.

```
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!).

Arguments are Passed by Value

(Slide 15 - 17)

Strings are immutable objects

(Slide 18)

More about Local Variables

(Slide 20)

Returning a value from a method

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

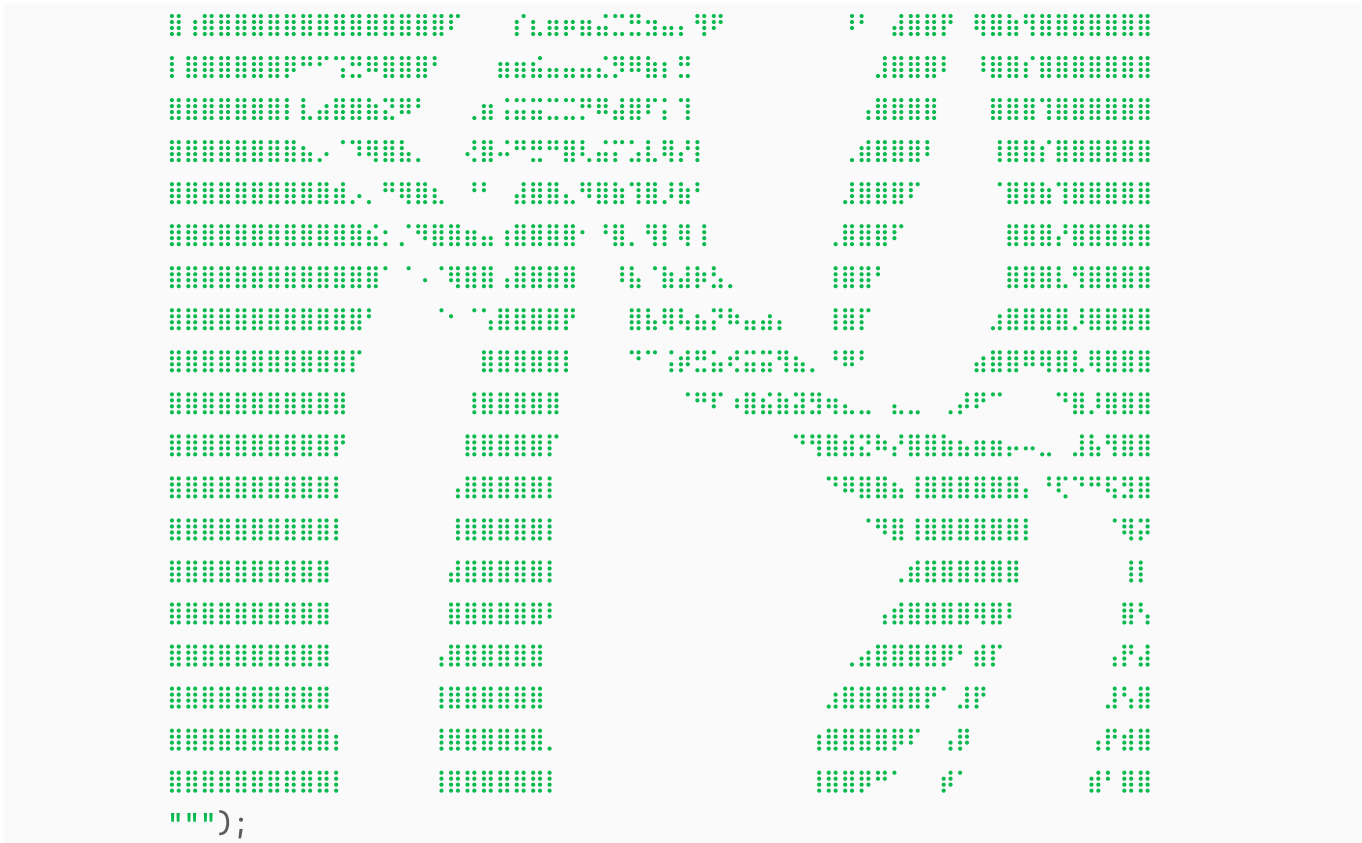
All methods that use a Scanner object to open a file must throw or handle `IOException`. Check out the notes in Chapter 4 - Files: Exceptions section for handling exceptions.

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("""
```



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