# Lesson 3 Section 1 Understand Methods and Encapsulation

# Subsection 1: Work with methods, arguments, return values and overloading

Throughout this course you have been using the **main()** method to run code examples. The main method is a special type of method that provides the entrance to an application. As you learnt in the first section, when an application is launched the **main()** method is the first method to be executed and from this method all other methods are called.

In this section we will look deeper into the structure of methods and examine their constituent parts.

## Method signature

If you remember from the **main()** method has a special signature that identifies it as a **main()** method and looks like this:

**public void static main(String… args){ … }**

The **main()** method must be **public**, so that it can be accessed by the JVM from outside the package structure, **static** so that it can be executed by using the class name as a reference rather than an object instance (more on this concept later), it must be called **main** and must declare an array of arguments: **main(String… args).** Any deviation from this format will create a method that is not a main method and the JVM will not call it and your application will not launch.

The main method will call other methods to provide more functionality to the application. These methods follow a similar structure to the main method but with far more options to choose from when designed the method signature.

## Dissecting the Java Method

The each element of a method declaration gives it certain characteristics that we can use to help our program behaviour in certain ways. Let’s look at the entire structure and examine each element in turn.

Here are examples of valid and invalid method declarations.

**Valid Method declarations**

public String calculate(){ … }  
public static final String calculate (){ … }  
public final static String calculate (){ … }  
final public String calculate (){ … }   
static String calculate (){ … }  
abstract void calculate();

**Invalid Method declarations**

public String final calculate (){ … } // the optional specifier is in the wrong place  
public static String final calculate (){ … } // the optional specifier is in the wrong place  
public Final Static String final calculate (){ … } // the optional specifier is in the case   
abstract void calculate(){ … } // abstract methods don’t have method bodies  
public String final calculate (){ … } // the optional specifier is in the wrong place

Note: **native** and **strictfp** cannot be used together, nor can **abstract** and **final**, and **strictfp** and **synchronized**. All other optional specifiers can be combined like so:

private static final strictfp synchronized void calculate6() { … }

### Return types

The return type identifies the data type of the value the method can return. Here are some examples of valid and invalid methods with return defined.

**Valid return types**

public void calculate(){ … }  
public integer calculate(){ return 100; }  
public void calculate(){ return;}   
public String calculate(){ return getString(); }

**Invalid return types**

public void calculate(){  
 return 100;  
}  
This method should not return a value.

public void calculate(){  
 return Void;  
}  
Void is not a return type.

public integer calculate(){ return "Five"; }  
The value returned must be compatible with the return type

public integer calculate(){ return 100 }  
The semi-colon is missing from the return statement.

public String message (){ … }  
The method must have a return statement that returns a string

public message (){return "Hello" }  
All methods must have a return type defined.

public String message(int score){ if( score >= 90) return "You’ve won";}  
This will not compile because there is no return statement for the case where score is less than 90. The compiler will complain of a missing return statement

### Method name

The naming of a method follows the same rules as the naming of a variable. We covered this topic in the very first section of this course but let’s recap.

These rules are:

* It MUST start with a letter, currency ($,€,£ etc) or a connecting character (underscore \_ )
* It CANNOT start with a number, but can include a number
* It CANNOT include a space
* It CANNOT be a Java keyword (e.g. if, while, switch etc. See full keyword list below)
* It CANNOT be a literal (true, false, null)
* It CAN be of any length
* It CAN include a unicode characters, although note exception mentioned above
* It is case sensitive (Cat is not the same as cat)

It is conventional, but not legally required, to start the method name with a lowercase letter. Here is a list of perfectly legal, although not good, method names:

public void $greenback() {};  
public void \_$() {};  
public void \_\_\_\_\_\_2\_w() {};  
public void a\_true\_thing() {};  
public void £2000f() {};  
public void \_Ö\_() {};  
public void שלום() {};  
public void হ্যালো() {};  
public void €\_forever() {};

### List of parameters

The parameter list can contain zero to many parameters and is required. There are several ways to construct a parameter list.

Here are some invalid method declarations with var-args:

public void calculate(int x; int y String message){}   
You can only use commons to separate parameters, not semi-colons or spaces

public void calculate(int, String){}   
Don’t forget the variable name

public void calculate(int[5] numbers){}   
Ensure the you declare the array variable correctly

### Exceptions

We dedicated an entire chapter to exceptions so you should know all that’s required to know about exceptions. Here are some legal exception declarations

public void calculateScore() throws Exception {}

public void calculateScore() throws Exception, RuntimeException {}  
You can throw a RuntimeException if you want

public void calculateScore() throws Exception, Exception {}  
Legal but meaningless

public void calculateScore() throws ConnectionException, IllegalArgumentException {}  
You can declare as many exceptions as you wish

### Method body

The method body contains the code block to execute when the method is called, unless it is an abstract method or a method in an interface. You will meet these two concepts later in the course.

## Variable arguments (Var-args)

Var-args were designed to allow methods to have an indeterminable number of parameters, so that one method could be called by methods with a variable number of arguments.

# Subsection 2: Learn to use the static keyword on methods and fields

Methods and fields can be declared as static. This means that only one copy of the method or field exists and it is shared among all instances of the class.

# Subsection 3: Understand constructors; overloaded, default and user defined

At the beginning of this course you learnt a little about constructers and that when a class is created with the **new** keyword the class’s constructor is called. In this section I will expand on this topic and cover constructor in more detail.

## Constructors in detail

All classes have constructors, except interfaces. If a class does not explicitly define a constructor the compile inserts a no argument constructor. The constructor has the same name as the class (and the same case). They appear like methods but don’t have a return type. A **void** return is not valid. A constructer can have zero to many parameters and is invoked when the **new** keyword is used to create a new object instance. The constructor can have one of the following access modifiers: **public**, **protected**, **default access** or **private**, but they cannot be **final**, **static** or **abstract**. Constructors must declare all the checked exceptions declared in the base constructor (or the super classes of the checked exceptions). They may add other exception. Constructors marked **private** can only be called from within their class and by using a call to this.

#### Here are some valid constructors

public class Number {  
   
}

This class has a no-argument constructor inserted by the compiler. If you could see it, it would look like the constructor in the example below.

public class Number {  
 public Number(){}  
}

public class Number {  
 private int x;  
 public Number(int y){  
 x = y;  
 }  
}

Constructors can have parameters

public class Number {  
 private Number(){}  
}

Constructors can be public

public class Number {  
 private int x;  
 public Number(int... y){  
 x = y[0];  
 }  
}

Constructors can have var-args

#### Here are some invalid constructors

public class Number {  
 private Number(){}  
 public Number(){}  
}

The constructor signature must be unique.

public class Number {  
 public number(){}  
}

The constructor must be the same name and case as the class name.

public class Number {  
 public static Number(){}  
}

Constructors cannot be static

## Default Constructor

A class without a constructor means that the compiler will insert a constructor that has no arguments and no body.

## Initialisation blocks and constructors

The order in which constructors, static and instance initialisers are called is important. You do not have control over the order in which they are set so you must learn the rules. The simplest way to show the flow of execution is with an ordered list. Below is a list of the order from when the class is loaded to when the class is ready for use.

The order of execution:

1. Static blocks and static members are executed in the order they appear in the class starting with the class at the top of the class hierarchy tree down to the current class.
2. Instance variables are given their values (including default values) in the parent class
3. Instance blocks are executed in the parent in the order they appear in the class.
4. The code in the constructors that were called on the way up the class hierarchy is executed from the parent class down to the current class.
5. The class have been created and is ready for use.

This might be a little confusing as we are talking about class hierarchy. This is a topic that we’ll cover in a later section. I suggest that you return to this subsection once you are comfortable with how a class hierarchy works

# Subsection 4: Use access modifiers correctly

Let’s have a closer look at access modifiers. You already know that they advise the access level of the method, but how does this affect method calls? I will discuss each access modifier in turn from the most restricted to the least restrictive.

# Subsection 5: Create classes with encapsulation

Encapsulation is about ensuring that data in a class is only accessed via a method and not directly by referencing the field.

# Subsection 6: Understand the behaviour of object references and primitive values

When a method is called with arguments, the argument values passed to the method are copies of the original values.