COURSEWARE


Professional Skills

Agile Fundamentals

Jira

Git

Databases Introduction

Java Beginner

- What is Java?
- Installation
- Hello World Example
- Data Types
- Packages
- Naming Conventions Cheat Sheet
- Flow of Control
- Class Members
- Operators
- Conditionals
- Iteration
- Arrays
- ArrayList
- Enhanced For Loops
- String Manipulation
- Class Constructors
- Access Modifiers
- Installing Java & Maven To PATH
- Object-Oriented Programming Principles
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction
- Interfaces
- Type Casting
- Static
- Final
- Garbage Collection
- Input With Scanner
- Pass by Value/Reference
- JUnit


# Flow of Control

## Contents


- Overview
  - Flow of Control
  - Java Call Stack
  - Scope
- Tutorial
- Exercises


## Overview

The `flow of control` defines the order in which our code is executed by Java. This module will explore how Java controls the order our code will be executed in, as well as *scope* and how it differs from classes, methods, and blocks of code.

### Flow of Control

In Java, code execution will begin in the main method, this method can call other methods, create variables, etc.
If you try to run your code and get the error "main method not found" it is because you have either not declared the main method, or not declared it correctly.
The main method should be declared like this:

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

}
```

Code execution starts in the main method and the first line is called followed by the second, then the third, and so on.
So if we had the following code:

```
public static void method1() {
    System.out.println("Hello");
}

public static void method2() {
    System.out.println("World");
}

public static String method3() {
    return "!";
}

public static void main(String[] args) {
    method1();
    method2();
    System.out.println(method3());
}
```

- It would start in the main method, execute the first line, which is a method call to the method called method1().
- The code would then go into method1() and execute the first line of that method, which is to print "Hello" to the console.
- After that line is executed, the method has no more lines to execute; so Java goes back to the main method and executes the next line of code, which in this case is a method call to method2().

- Test Driven Development
- UML Basics
- JavaDoc
- Peer Programming
- Code Reviews

Maven

Testing (Foundation)

Java Intermediate

HTML

CSS

Javascript

Spring Boot

Selenium

Sonarqube

Advanced Testing (Theory)

Cucumber

MongoDB

Express

NodeJS

React

Express-Testing

Networking

Security

Cloud Fundamentals

AWS Foundations

AWS Intermediate

Linux

DevOps

Jenkins Introduction

Jenkins Pipeline

Markdown

IDE Cheatsheet


- Java will now go into method2() and execute the first line of that code, which is to print "World" to the console.
  Once again, after that code of line has been executed the method has no more code to execute, so Java goes back to the main method.
- Java will now execute the next line of code in the main method, which in this case is to print whatever is returned by method3() to the console.
  So Java will first go into method3() to get the value to be printed, and the first line of code in method3() will be executed which is to return the value "!".
  Whenever the return keyword is used, no more code within that method will be executed, anything after the return statement will be unreachable.
- Now that method3() has returned the value "!" for us, Java can print it to the console.
- This gives us the following output:

```
Hello
World
!
```

Each word is printed on a new line because we called the System.out.println() function to print to console instead of the System.out.print() function.

## Java Call Stack

The `Call Stack` is what Java uses to keep track of method calls. The call stack is made up of `Stack Frames` - for each new method call, one is generated and added to the call stack in a *Last-In, First-Out (LIFO)* fashion.

```
import java.util.Random;

public class Dice {

    public static void main(String[] args) {
        System.out.println(rollTwoDTwenty());
    }

    public static int rollTwoDTwenty() {
        int total = 0;
        total += rollDTwenty();
        total += rollDTwenty();
        return total;
    }

    public static int rollDTwenty() {
        return new Random().nextInt(21);
    }

}
```

In the above example. First, our program calls `rollTwoDTwenty()` and adds it to the call stack:

| rollTwoDTwenty() |
| --- |

Then, that method calls `rollDTwenty()`, adding it to the top of the stack:

| rollDTwenty() |
| --- |
| rollTwoDTwenty() |

Next, `Random().nextint()` is called and added:

| new Random().nextint() |
| --- |
| rollDTwenty() |
| rollTwoDTwenty() |

| new Random().nextint() |
| --- |
| rollTwoDTwenty() |

When `Random().nextint()` returns a value, it is `popped` off the stack and we return to `rollDTwenty()` with the value from `Random().nextint()`:

| rollDTwenty() |
| --- |
| rollTwoDTwenty() |

`rollDTwenty()` now returns the value to `rollTwoDTwenty()` and pops off the stack:

| rollTwoDTwenty() |
| --- |
| new Random().nextint() |
| rollDTwenty() |
| rollTwoDTwenty() |

`rollDTwenty()` now calls `rollDTwenty()` again, and we end up with a familiar-looking stack:

| new Random().nextint() |
| --- |
| rollDTwenty() |
| rollTwoDTwenty() |

We keep returning the values down through the stack until we end up with this:

| rollTwoDTwenty() |
| --- |

Now `rollTwoDTwenty()` can return the value of `total`, and the main method prints the result to the console.

Through our understanding of the stack and flow of control, we can move through our programs much more efficiently.

## Scope

When working with Java understanding scope is very important. Understanding scope is important because it helps to understand when a variable will be accessible to a certain line of code and when it will be inaccessible, the smaller the scope the less accessible the variable is. There are three main levels of scope, class level/instance scope, method/local scope, and loop scope.

Class Level/Instance Scope : Referenceable throughout the entire class, these variables are inside the class but outside of methods. Generally these variables will be defined at the top of the class.

```
public class HelloWorld {

    public static String message = "Hello World!";

    public static void method1() {
        System.out.println(message);
    }

    public static void main(String[] args) {
        method1();
    }

}
```

In the above example, message is defined and initialised as a String variable within the class scope, but outside of any method scope.
Therefore the variable can be referenced and used throughout the class freely.

Method/Local Scope : Variables that are temporary and generally only used in the method that they are declared in.
As soon as the method ends all variables declared inside that method are no longer referenced and cannot be accessed anymore.

```
public class HelloWorld {

    public static void method1() {
        String message = "Hello World!";
        System.out.println(message);
    }

    public static void main(String[] args) {
        method1();
    }

}
```

In the above example message is defined and initialised as a String variable inside a method's scope, in this case inside the scope of method1().
The message variable will no longer be accessible once method1() has finished executing all of the lines of code within its scope.

Loop Scope : Variables that are declared inside a loop declaration and are only accessible inside the loop and are lost once the loop has ended.

```
public class HelloWorld {

    public static void method1() {
        for(int i = 0; i < 10; i++) {
            int number = 20;
            number += i;
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        method1();
    }

}
```

In the above example, `number` is defined and initialised within the for loop's scope.
The number variable will be accessible whilst Java is executing the code within the loop, but once the loop has finished it will no longer be accessible.

## Tutorial

There is no tutorial for this module.

## Exercises

There are no exercises for this module.