

# Java

- Running Java Applications
- What the heck are @Annotations?
- Maven
  - Maven Intro
  - Parent and Child pom.xml
  - package, install, release, deploy phases
  - Lifecycle, Phases, goals, and Plugins?!
  - Compile time and runtime dependency
  - Jacoco Plugin
  - Lombok
  - Maven Tests
- Java Loggers
- System property value
- Lambda expression in Java
- Checked vs Unchecked Exception
- Mockito how does it work?
- Optional in Java
- Java Nested Class
- Stream map vs flatMap

# Running Java Applications

## Running Java App

In order to run your Java Applications you must first compile all of the source code into its `.class` byte code file respectively. Then in order to run your code you have to specify the fully qualified name for the class to the `java` command.

When you are running the code, the current working directory doesn't really matter as you can change the search path using the `-cp` or `--classpath` option

### Fully qualified name

The fully qualified name is the class under the corresponding package. A package is just a list of related files, think of it as a folder. When you write a Java class say under the `src/com/yy` directory call it `Circle.java`, then `Circle.class`'s fully qualified name is `com.yy.Circle` after you have compiled it into byte code.

`src` is not part of the package name, `src` itself is not a package, but inside the `src` directory all of it is considered to be Java packages, and they have their respective qualified name starting from inside `src`.

### How does `java` run the program?

`java` will take a fully qualified name, not a directory to the `.class` file! Keep in mind!

When you use `java` along with a fully qualified name it is going to do the following:

1. Search for the compiled version of the fully qualified named class
2. Load the class
3. Check that the class has a `main` method
4. Call that method and passing it the command line arguments if any

### Why Java cannot find the class

Remember that fully qualified name is not a file path! It is the name to refer to that class underneath `src`.

If your `.class` that you want to run is under the fully qualified name `com.yy.Circle` then you must specify that as the fully qualified name, not `Circle` or `circle`.

## Wrong classpath

Now, when you execute `java` and provide a fully qualified name, it is going to search for that `.class` in couple of places. If you set the `CLASSPATH` environment variable then it is going to search through that list of directory to find `.class`. If not then it is going to check if you have provided any `-cp/-classpath` argument to the `java` command and check through those. If you didn't specify it then the current directory is going to be used as the classpath.

- To make it clear say you are executing `com.acme.example.Foon`, the `Foon.class` class
- The full file path to the `Foon.class` is `/usr/local/acme/classes/com/acme/example/Foon.class`
- Your current working directory is `/usr/local/acme/classes/com/acme/example/`

Then in order to run `Foon.class` you have to run

```
java -cp /usr/local/acme/classes com.acme.example.Foon
```

Now Java is able to find the correct `Foon.class` file by following the classpath you have specified, and looking it under `/usr/local/acme/classes/com/acme/example` directory to find that `Foon.class` file correctly.

**Basically you have to provide the source directory path to the `.class` that you have compiled. That's all.** It will follow the fully qualified name underneath `classes` directory to find `com` folder, then follow `acme` folder then `example` folder and finally find the correctly file.

It is as if it `cd` to the classpath you have provided, then try to locate that file by following your qualified name like a folder traversal.

## How do I import classes I wrote in same package?

You do not need to import it, you can just use it directly because they exist in the same package.

## Wait I wrote a class directly under `src`, how do I import it?

You cannot directly import classes you have written directly under `src` folder because it technically does not have a package name. However, it is able to be exposed to outside.

# What the heck are @Annotations?

## Annotation

They are metadata that you can write into your Java program. They don't affect the execution of your code directly, but they can be processed by the compiler or at runtime to change the behavior of your code.

You should already have seen couple of common annotations such as `@SuppressWarnings(param)`, to get rid of those warning such as unused variables. You have to specify what kind of warnings you want to suppress, "unused" in this case to get rid of unused variable warnings.

Annotations can be added to classes, methods, variables, and even annotations themselves.

## Creating custom annotation

To create your own annotation you would do something similar to creating your own class:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface VeryImportant {

}
```

Declaring the annotation is really all you need to create your own annotation, but there are two annotation that you would want to use to custom it further. Those are `@Target`, and `@Retention`.

`@Target` allow you to specify what kind of Java element this annotation is valid to be used for. For example, make it so that you can only apply this annotation to a class, to a method, to a variable, or even to an interface. If you want to specify multiple target you would use an array of `ElementType.<Java element>`.

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface VeryImportant {
```

```
}
```

## Retention policy

Tells Java when to discard the annotations.

@Retention, most of the time you would use RetentionPolicy.RUNTIME which means to keep this annotation around through the running of your program so that other code can inspect this annotation and process it such.

The other two possible value is SOURCE and CLASS. SOURCE will get rid of annotation before the compilation of your code. These are annotation that matter before the program is being compiled such as @SuppressWarnings. CLASS will keep your annotation through the compilation process, but once your program starts those annotation will be discarded.

## Adding parameter to annotation

To add parameter to your annotation you have to declare them inside your annotation definition as a method, even though they are kind of accessed like a field:

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface VeryImportant {
    String why();
}
```

For example, in this annotation we specified for the @VeryImportant annotation it has a required String parameter that you must passed into the annotation when you use it.

You can set up default values for each of the parameter of annotation, but if they don't have default value they are required.

The type of parameter for annotation are limited, they can only be primitives such as String, int, float, boolean, or an array type by adding brackets.

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface VeryImportant {
    String why();
    int random() default 1;
    int[] nums;
}
```

## Accessing parameter of annotation

To access the parameter that you have passed to the annotation you have to first retrieve the annotation object by calling `getAnnotation` on the object, method, or variable the thing that you annotated on.

Then you can just access it as a getter method, by calling on the name you have set in the annotation.

```
VeryImportant annotation = myCat.getAnnotation(VeryImportant.class);  
  
annotation.why();
```

## Processing annotation

The annotation by themselves, with `@Target` and `@Retention` doesn't really do much on their own. It is the code that you write that actually inspects the annotation that gives them special effects.

You can inspect class instances to check whether they have a particular annotation by calling `isAnnotationPresent(Annotation.class)`. For example:

```
Cat myCat = new Cat("Stella");  
  
myCat.getClass().isAnnotationPresent(VeryImportant.class);
```

If the class `Cat` is annotated with the `@VeryImportant` annotation then this method will return `true`, otherwise if it is not annotated with `@VeryImportant` it will return false.

## Java reflection

Java reflection is an API that is provided by Java to get metadata about an Java object. Things like getting the list of methods is inspecting the metadata about the Java object.

`getDeclaredMethods()` returns you a list of `Method` objects which you can print out which tells you the method name that this particular object has. In addition, you can also invoke them by calling `.invoke` on each of the `Method` objects.

`getDeclaredFields()` return you a list of `Field` object which are fields that you have inside your object. This is how you would be accessing the annotation that you did on each of the field of your object.

# Maven

# Maven Intro

## What is Maven

Maven is a build automation tool. Think of it as a Makefile but instead of you writing the Makefile yourself, it writes that Makefile for you automatically to run and produce the final executable.

Maven also handle external third party dependency for you, meaning if you have another library that you would like to use and is available in the Maven repository, you can include it in the pom.xml file, and when you are writing your Maven project, you can just install those dependency and on your way to write your application with those dependency.

## Maven + Vscode

Keep in mind that Maven doesn't actually do auto compiling for you, you would have to run `mvn compile` everytime when you make changes to your code. Or any other way of compiling depending on other plugins you use.

Maven compiles the `.class` files under the `target` directory!

For Spring boot and maven-compiler-plugin you would have to compile it yourself everytime you make code changes.

However, if you use Vscode along with the Java language support then it will compile those classes for you. For projects without build tool the `.class` files are under `~/.vscode-server`, for those projects with build tool then it is under `bin` folder by default in your project root. You can change that by editing your pom.xml file.

Maven

# Parent and Child pom.xml

<https://howtodoinjava.com/maven/maven-parent-child-pom-example/>

# package, install, release, deploy phases

- `mvn package` will construct your artifact, i.e. a jar file and then place it in the current working directory
- `mvn install` will put your packaged maven project i.e. a jar file into the **local** repository, so that your other local application can use it as a project dependency. Just like dms-encryption-util, we have to install it into `~/.m2/repository` directory so that it can be found.
  - When your project does not find the dependency locally, it will fetch it from the remote repository and put it into `~/.m2/repository`
- `mvn release` will tag your current code in the specified SCM (version control system) and then change your version in your project. Basically it help you do automatic version tagging on GitHub
  - This is not the default life cycle phase, it is
- `mvn deploy` will put your packaged maven project into the remote central repository, maven or internal repository that you have specified for sharing with other developers

# Lifecycle, Phases, goals, and Plugins?!

## Maven Lifecycle

Maven is a build automation tool at heart. It has three built-in lifecycle that clearly define how the project will be build. There are three built-in build lifecycles:

1. default: handles the actual project compilation and deployment
2. clean: handles project cleaning, removing the target folder
3. site: handles the creation of project site documentation

You "can" add more lifecycle but then that go against the point of creating those three built-in build lifecycles. Those three built-in ones should be the standard of all the projects, you can create your own plugins and then hook it into the phases within the lifecycle.

Adding more lifecycle would take away the ease of reproducing the build and maintaining it.

Each of the lifecycle consists of phases. So you have life cycles like **default**, it consists of **phases**, and each **phases** consists of **goals**.

### Phases in default

Not going to list them all but here are some of the phases in order:

- validate
- compile
- test
- package
- verify
- install
- deploy

When you execute a phase all the goals tied to the goal will be executed. In addition, when you execute a phase all the previous steps will be executed. For example, executing `mvn install` will execute, validate, compile, test, package, verify, and install in order.

There is no way to execute the "lifecycle", to execute the lifecycle you would just specify the last phase to run through the entire phases. In this case running `mvn deploy` will in a sense run the entire lifecycle

## Maven phases

Let's clear something up, all the available goals are provided by plugins. The compile phase's goals are provided by the "maven-compiler-plugin", the package phase's goal are provided by the "maven-jar-plugin", you get the idea.

Official maven plugin like `compile`, `package` have standard name of "maven-`<name>`-plugin", whereas non-official maven plugin have "`<name>`-maven-plugin" naming convention.

The implication is that when you want to invoke the goal directly without invoking the phase if you are using official maven plugin is just "`<name>:<goal>`" for example, `mvn compiler:compile`. You don't need to do `mvn maven-compiler-plugin:compile`. However, unofficial maven plugins then you would need to provide the entire plugin name to invoke the goal.

A build phase doesn't necessarily need to have any goal bound to it. If it has no goal bound to it, it will simply not execute any goal.

However, the same goal can be bounded to multiple different build phases, so that goal will just be executed multiple times when the phase is executed.

Furthermore, not all the goals from a plugin are bind a phase. For example, the `maven-compiler-plugin` have two goals, `compile` / `testCompile`. However, only the `compile` goal is bound to the compile phase. You can add `testCompile` to the phase by adding an `<executions>` section in your plugin tag, will go more over that in the next section.

## Maven goals (Mojos)

Goals are the actual task that are executed, they help building and manage the project. Phases and lifecycle are essentially just abstraction of goals.

Goals may or may not not bound to a phase, those that are tied to a phase will be executed when you run the phase. Those are not bound to a phase, you can invoke them directly. By using the syntax discussed above.

To bound a plugin's goal to a phase if they are not bound by default, or you would like to bound to a different phase, i.e. adding jar plugin's jar goal to the compile phase. Whenever you run `mvn compile` it will also package it as a jar, as oppose to whenever you run `mvn package` if you don't want

to run test phase before it. What you would do is to add a `<executions>` section in your pom.xml file.

## Configuring plugin execution

If you would like to add a goal that isn't by default mapped to one of the default phases for your plugin, or you would like to add it to different phase then you would need to append a `<executions>` section for your plugin.

For example, the `jar:jar` goal which build a JAR from the current project is by default bind to the `package` but if you want to run the goal whenever `compile` plugin is run then you can add the following section to your `pom.xml`

```
<executions>
  <execution>
    <id>compile-jar</id>
    <phase>compile</phase>
    <goals>
      <goal>jar</goal>
    </goals>
  </execution>
</executions>
```

What this section does is that it will bind the goal `jar:jar` remember goals can be bind to multiple phases, to the `compile` phase, with the id `compile-jar`. The id is there so that when you invoke the `mvn` command, the execution output will be in the form of

```
<plugin-name>:<plugin-version>:<phase> (<execution-id>)
```

So in this case, as part of the `compile` phase, it will be running `maven-jar-plugin:2.2.2:jar (compile-jar)`, so right after you compile your code, you will pack it into a jar.

## Scenarios for executions

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-docck-plugin</artifactId>
  <version>1.0</version>
  <executions>
    <execution>
      <id>some-other-other-id</id> <!-- No goal for execution is defined -->
      <phase>pre-site</phase>
    </execution>
```

```

<execution>
  <phase>pre-site</phase>  <!-- No id for execution is defined -->
  <goals>
    <goal>check</goal>
  </goals>
</execution>
<execution>
  <id>some-id</id>      <!-- No phase for execution is defined -->
  <goals>
    <goal>check</goal>
  </goals>
</execution>
<execution>
  <id>some-other-id</id>  <!-- Both id and phase defined -->
  <phase>pre-site</phase>
  <goals>
    <goal>check</goal>
  </goals>
</execution>
</executions>
</plugin>

```

Results in sample output as following

```

[INFO] --- maven-docck-plugin:1.0:check (default) @ MavenJavaApplication ---
[INFO] Skipping unsupported project: MavenJavaApplication
[INFO] No documentation errors were found.
[INFO]
[INFO] --- maven-docck-plugin:1.0:check (some-other-id) @ MavenJavaApplication ---
[INFO] Skipping unsupported project: MavenJavaApplication
[INFO] No documentation errors were found.

```

## Execution 1: No goal for execution is defined

If no goal is specified then that execution is simply ignored. If a phase have no goal then it will just be ignored, because there is nothing to execute.

## Execution 2: No id for execution is defined

If no id is defined for a specified execution, but goal and phase is specified then it will simply execute with the id of `default`.

## Execution 3: No phase for execution is defined

If no phase is defined, then that execution of plugin goal is simply not run.

## Execution 4: id, phase, and goal is defined

Then it will get ran during the specified phase with the specified id

### default-cli vs default-<goalName>

The goals that are invoked directly from the command line for example `mvn jar:jar` (invoking the jar goal from the `maven-jar-plugin`) it will have the execution id of `default-cli` because you ran it in the CLI.

This ID is provided so that you can provide further configuration if needed for the plugin that's ran from CLI, by using the execution id of `default-cli` in your `pom.xml` file.

Plugin goals that are mapped to the default lifecycle like some of the official maven plugin such as `maven-jar-plugin` will have their execution id to be `default-<goalName>` for example:

```
<artifactId>maven-clean-plugin</artifactId>
<version>2.5</version>
<executions>
  <execution>
    <id>default-clean</id>
    <phase>clean</phase>
    <goals>
      <goal>clean</goal>
    </goals>
  </execution>
</executions>
```

The `clean` plugin's default goal mapping maps the `clean` goal to the `clean` phase with the execution id of `default-clean`.

Again this is there so that you can further configure these official built-in goals, it is a naming convention after all.

You can find out about this by looking at the effective pom via `mvn help:effective-pom`.



# Compile time and runtime dependency

## Problem

There was bunch of dependency that older version of spring-boot includes as compile time dependency, meaning in your project without you explicitly including those as a dependency in your `pom.xml` you can use them in your code.

However, if you upgrade spring-boot version to more recent versions those dependency became RUNTIME scope, meaning those dependencies will not be included as part of the compile classpath. You must explicitly declare it as a dependency in your project in order to use it.

## Why I bring this up

This is from my experience where I upgraded spring-boot to a higher version, in the older version it used a class from a transitive dependency and that compiled fine because that transitive dependency was included as a compile time dependency.

However, after upgrading it, the same transitive dependency became runtime only, thus it isn't available anymore, which lead to compilation error if you use the library without explicitly declaring it as a dependency in your `pom.xml`

# Jacoco Plugin

## What is Code Coverage

Code coverage is a quality measurement during development process to see how much of your code has been tested (or executed). Your unit test should cover most of business logic of your code, however, reaching 100% of code coverage does not mean that your code is bug-free. You can reach 100% code coverage while still having bugs.

## What is JaCoCo

Jacoco maven plugin is a free code coverage library for Java projects. It is used to generate code coverage reports.

How it works is that JaCoCo plugin attaches a runtime agent to the JVM when it starts. The JaCoCo agent will then instrument the class so that it can see when the class is called what lines of codes are called during the testing process. It then build up the code coverage statistics during the testing phase.

## How to setup JaCoCo

To set up Jacoco-maven-plugin you would need to include this following plugin in your pom.xml:

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.6</version>
</plugin>
```

The latest version you would need to determine based on what is released. Then you would also want to add the following executions tag in order to trigger Jacoco maven plugin whenever you run the unit test of your project.

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
```

```

<version>0.8.6</version>
<executions>
  <execution>
    <id>prepare-agent</id>
    <goals>
      <goal>prepare-agent</goal>
    </goals>
  </execution>
  <execution>
    <id>report</id>
    <phase>test</phase>
    <goals>
      <goal>report</goal>
    </goals>
  </execution>
</executions>
</plugin>

```

- **Prepare-agent goal:** This goal prepares the Jacoco runtime agent to record the execution data, record the number of lines that's executed, backtraced, etc. This must be attached otherwise, Jacoco will not have data to generate the code coverage
- **Report goal:** This goal will finally create the code coverage reports from the execution data recorded by the runtime agent. Since this is attached to the test phase, whenever the test phase finishes it will generate the report automatically without having you to invoke the goal manually.

## Add code coverage check

You can also enforce minimum code coverage check when you are executing the test

### Toggle me for full code

```

<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.10</version>
  <executions>
    <execution>
      <id>prepare-agent</id>
      <goals>

```

```

    <goal>prepare-agent</goal>
  </goals>
</execution>
<execution>
  <id>jacoco-report uwu</id>
  <phase>test</phase>
  <goals>
    <goal>report</goal>
  </goals>
</execution>
<execution>
  <phase>test</phase>
  <id>coverage-check</id>
  <goals>
    <goal>check</goal>
  </goals>
<configuration>
  <rules>
    <rule>
      <element>PACKAGE</element>
      <limits>
        <limit>
          <counter>LINE</counter>
          <value>COVEREDRATIO</value>
          <minimum>0.8</minimum>
        </limit>
      </limits>
    </rule>
  </rules>
</configuration>
</execution>
</executions>
</plugin>

```

The goal will fail if your code coverage doesn't meet the specified percentage.

## Ignore class files for coverage

You can add to the configuration the list of class files that you should ignore because it is difficult / unnecessary to test them for coverage.

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.10</version>
  <configuration>
    <excludes>
      <exclude>*/App.class</exclude>
    </excludes>
  </configuration>
</plugin>
```

# Lombok

## Project Lombok

Lombok is a Java library that can help you generate lots of the boiler plate code so that you don't have to end up writing them yourself. For example, getter, setter, toString, and equals method. You just have to slap an annotation onto the class and boom you will have the boiler plate stuff generated all for you.

### Jacoco code coverage

If you use Lombok annotation then you will be complained about a lot about code coverage because the method isn't being called in the unit test.

You can ignore Lombok generated method by adding a `lombok.config` file with the following configuration

```
# Adds the Lombok annotation to all of the code generated by
# Lombok so that it will be automatically excluded from coverage by jacoco.
lombok.addLombokGeneratedAnnotation = true
```

# Maven Tests

## Maven Test

When you execute `mvn test` it will compile and test the source code that's underneath the source code folder!

Maven Test allows integration with JUnit, TestNG, and POJO. We will go over JUnit because that's what we use most of the time.

## JUnit

The popular testing framework for Java. Most of the test are written in JUnit 4 but the latest version is JUnit 5.

Writing them are still the same, however, integrating it with behavioral driven testing framework like Cucumber will be different I will go over the differences for using cucumber with JUnit 4 and JUnit 5 (Mainly in the annotation that is used to specify the glue and feature files)

When you write your unit tests using JUnit you want to structure your file directory like the following:

```
src
├── main
│   ├── java
│   │   └── test
│   │       └── App.java
│   └── resources
│       └── application.properties
└── test
    ├── java
    │   └── test
    │       └── AppTest.java
    └── resources
        └── test.json
```

Your main source code directory will contain two different directories.

- The `main` directory contain the Java source code for your application as well as any resources that your application uses
- The `test` directory contain the Java source code used for testing. This is where you will be writing your unit test for your applications

When you run `mvn test` it will compile the source code from the `src` directory and execute the proper unit test using the JUnit framework.

## Dependency needed for JUnit 5

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.9.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

## Test Class

A test class is a class that will be used by Maven to carry out the testing.

A test class contains methods that will be tested by Maven. Maven use a pattern to check whether a class is a test class or name, the list of pattern used by Maven to determine if a class is a test class is of the following:

```
**/Test*.java
**/*Test.java
**/*Tests.java
**/*TestCase.java
```

## Test Runner Class

This applies to JUnit 4 test. JUnit 5 has a new API and `@RunWith` is no longer needed.

A test runner in JUnit is just a simple class that implements the `Describable` interface. The interface just have two abstract methods to implement:

```
public abstract class Runner implements Describable {  
    public abstract Description getDescription();  
    public abstract void run(RunNotifier notifier);  
}
```

1. getDescription: Is used to return a description containing the information about the test, this is used by different tools such as those from IDE
2. run: This is used to actually run the test class or test suite

You can implement your own test runner to do custom logic for running tests, for example you can print out silly messages just before each test is run.

A simple test runner class implementation can be found below: All it does is before it runs the test class is it prints the message "MyRunner <TestClass>"

### Example Test Runner

```
//src/main/java/MinimalRunner  
package com.codingninjas.testrunner;  
  
import org.junit.runner.Description;  
import org.junit.runner.Runner;  
import org.junit.runner.notification.RunNotifier;  
import org.junit.Test;  
import java.lang.reflect.Method;  
  
public class MinimalRunner extends Runner {  
  
    private Class testClass;  
    public MinimalRunner(Class testClass) {  
        super();  
        this.testClass = testClass;  
    }  
  
    //getDescription method inherited from Describable  
    @Override
```

```

public Description getDescription() {
    return Description
        .createTestDescription(testClass, "My runner description");
}

//our run implementation
@Override
public void run(RunNotifier notifier) {
    System.out.println("running the tests from MyRunner: " + testClass);
    try {
        Object testObject = testClass.newInstance();
        for (Method method : testClass.getMethods()) {
            if (method.isAnnotationPresent(Test.class)) {
                notifier.fireTestStarted(Description
                    .createTestDescription(testClass, method.getName()));
                method.invoke(testObject);
                notifier.fireTestFinished(Description
                    .createTestDescription(testClass, method.getName()));
            }
        }
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

To actually use this example Test Runner you would then just simply annotate your test class with `@RunWith(MinimalRunner.class)` which will delegate the responsibility of running this particular test class to that custom test runner.

```

//src/test/java/CalculatorTest.java
package com.codingninjas.testrunner;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;

```

```
@RunWith(MinimalRunner.class)
public class CalculatorTest {
    Calculator calculator = new Calculator();

    @Test
    public void testAddition() {
        System.out.println("in testAddition");
        assertEquals("addition", 8, calculator.add(5, 3));
    }
}
```

Note that this runner mechanism is no longer present in JUnit 5 and is replaced by Jupiter extension. <https://stackoverflow.com/a/59360733/7238625>

## Default Test Runner class

For JUnit 4 at least, if you don't specify a test runner, JUnit will default to the default `BlockJUnit4ClassRunner` which as you know it will just simply executes the test class.

## Specify one test class to execute

If you only want to run one of the test class then you can run maven test with the following option

```
mvn test -Dtest="TestFoo"
```

This will only execute the `TestFoo` class.

## Specify one method to execute from a test class

If you only want to execute one of the test then you can run maven test with the following option

```
mvn test -Dtest="TestFoo#testFoo"
```

The left of # denotes the test class, and the right of # denotes the method that you want to execute.

# Cucumber + JUnit 4

Note that the following code only works with JUnit 4

Cucumber is a test automation tool that follows BDD. You describe the things you want to test in something called feature files which uses Gherkin language specification. This is so that stakeholders who doesn't understand code can also look at the feature file and understand the behavior of your application without needing to learn the programming language.

Underneath cucumber will then translate the feature files into actual code.

Cucumber framework in Java works by implementing it's own test runner class with JUnit, this is how it makes it work, and why the test class using cucumber test doesn't have anything in them. In the CucumberOptions annotation you will be specifying the feature files you will be looking at, the glue it is using, the specific tags to run, and plugin options.

**The list of sample dependency to use Cucumber with JUnit will be the following:**

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>7.14.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>7.14.0</version>
  <scope>test</scope>
</dependency>
```

## Terminology

1. Glue: A glue is used to associate the Gherkin steps (i.e. what's written in feature files) to a step definition method (i.e. actual code that gets executed). The Gherkin step is "glued" to the step definition. If this is used in the context of the test class, then it is referring to the package that your step definition class lives under.
2. Features: A feature file contains the description of the test in Gherkin. This is what cucumbers look at to execute the actual test. It is meant to be easy to read for stakeholders in plain English so even if they don't understand programming language they

can interpret the test cases as well.

3. Plugin: Reporting plugins that can be used to output the result of the cucumber test into different formats. Json and html output, you can also use third party plugins as well!

## Writing cucumber test

Your file structure should look like this:

```
src
├── test
│   ├── java
│   │   ├── com
│   │   │   ├── lol
│   │   │   │   ├── StepDefinition.java
│   │   │   │   ├── OneMoreStep.java
│   │   │   └── TestingFoo.java
│   └── resources
│       ├── features
│       └── foo.feature
```

Your feature files will be living under the `resources` folder. Your step definition will be living under the Java folder. Honestly, whether or not it is in another package just depend on you, you will be specifying where the step definitions are as part of glue configuration anyway. If you don't specify glue it will look for it under the test class.

Your test class will need to be annotated with `@RunWith` to tell JUnit that it will be the responsibility of the Cucumber test runner to run the test.

```
package com;

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/resources/features",
    glue = {"com"},
    plugin = {"html:target/out.html"}
)
public class TestingFoo {
```

```
}
```

# Cucumber + JUnit 5

Now because JUnit 5 has changed a lot of stuff, what worked for JUnit 4 will no longer work with JUnit 5 if you want to integrate cucumber with it. Here are the steps on how to do it.

Include the following dependency:

```
<dependencies>
  <dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite-engine</artifactId>
    <version>1.10.0</version>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>7.14.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-junit-platform-engine</artifactId>
    <version>7.14.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

The test class will then have to have these following configurations

```
package com;

import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;

import static io.cucumber.junit.platform.engine.Constants.GLUE_PROPERTY_NAME;
```

```
@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features")
@ConfigurationParameter(
    key = GLUE_PROPERTY_NAME,
    value = "com"
)
public class TestingFoo {

}
```

As you can see although the annotation differs a bit, it is still relatively the same.

## Multiple Runners

If for some reason your project is configured to have multiple test classes, remember JUnit will scan all of the test classes underneath the test directory, and you only want to run say one of the test classes. Then you can use the option flag that was reference above for the JUnit section :smile:

# Java Loggers

## Java Logging

Logging in a program is important to keep track of a program's run-time behavior. Logs capture and persist the important data and make it available for analyst later on.

### Logging frameworks

In Java there are multiple frameworks that you can use for logging. The framework give you the objects, methods, and configuration necessary to create and send log messages.

The built-in framework is `java.util.logging` package. However, there are many third-party frameworks like `Log4j`, `Logback`, `tinylog`. There are also something called an abstraction layer like `SLF4J` and `Apache Commons Logging` that decouples your code from the underlying logging framework. This is so that you can switch between logging frameworks on the fly.

### More about abstraction layers

Each of the logging framework have different objects that is created and all have different method names that is used to do the logging. As a programmer you are not going to remember all of their names and such the abstraction layer is created to cope with that problem.

Abstraction layer like `SLF4J` decouple the logging framework from your application, so that as a programmer you won't have to worry about exactly what the method and what object to create when you want to use say `Log4j` to do your logging, or if you want to switch it after developing your application for some time to say `Logback` you can easily switch it without any pain. i.e. not having to change all the object and method name that was used with the previous logging framework.

### Logging components

There are three core components to Java logging

- **Loggers:** They are responsible for capturing the events that you are logging and they pass it to the appropriate appender. The logger can sent it to multiple appender
- **Appender (Also called handler):** Responsible for recording the log to the specified destination. Appender use layouts to format the log before sending them to the destination
- **Layouts (Also called formatter):** Responsible for converting and formatting the data in the log. They basically determine how your log will look like when they are sent to the destination

So when your application makes a logging call, the Logger will record the event to a LogRecord and then forwards it to the appropriate Appender. The Appender will format the record using the Layout before sending it to the destination like the Console, to a File, or to another Application as well.

Filters can be applied to specify which Appender should be used for what type of logs. You can send configuration logs to the console, but warning messages to a file.

## Configuration

Each framework will be configured through configuration files. The configuration files are bundled with your executable and then are loaded by the logging framework at runtime to determine the behavior of the logging framework. Although you can also configure it in code, doing it using a configuration file is more maintainable and easier.

# Loggers

The logger objects are used to trigger log events that then record the event to a LogRecord which then is forwarded to the appropriate Appender for destination. One class can have multiple independent Loggers responding to different events and you can also nest Loggers.

## Create new logger

To create a new Logger you would do:

```
Logger logger = Logger.getLogger(MyClass.class.getName());
```

## Logging event

Then to trigger a log event you would call methods on the Logger object that is created

```
logger.log(Level.WARNING, "This is a warning!");
```

```
// or a short hand
```

```
logger.warning("This is a warning!");
```

The logs have levels. The level determines the severity of the log and can be used to filter the event or send it to a different Appender.

You can also prevent a Logger from logging messages below a certain level. For example in `Log4j` the levels are ordered like such ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF.

This means that if you set the log level to WARN. Then it will disable all log levels below WARN, so it will show all messages except DEBUG and INFO.

## Default logger and handler

So if you are using the global logger initialized by `java.util.Logger`, it is defaulted to `Level.INFO`. So you want to change the root logger's level if you want finer logs if you are using the root logger.

In addition, each logger also have a default Console handler. It also have it's level set to `Level.INFO` by default as well.

So if you want your own logger to show finer logs then you want to disable the default console handler then add your own console handler with it's own custom levels.

```
log.setUseParentHandlers(false);
```

## Appender / Handler

Append formats the log then sent the log to the correct output destination.

A logger can have multiple Appender. The two common appender (handler) for `java.util.logging` are `ConsoleHandler` and `FileHandler`.

You can then format the

### Specifying the `java.util.logging.config.file`

You can specify where the `logging.properties` files are by passing it as a system property value

## Layout / Formatter

For each appender you can also specify a layout or formatter.

By default the Console handler have the `SimpleFormatter` which just outputs the log in simple text.

However, `FileHandler` outputs it in XML format.

# System property value

## System property value

They are contained only within the Java platform.

Environment variables are global at the operating system level.

You can set system property value via CLI using `-Dpropertyname=value`

You set environment variable using `export envName=value`

You can get system property value using `System.getProperty(String key)`

You can get environment variable in java using `System.getenv(String anem)`

# Lambda expression in Java

## How are lambda function done in Java

Because Java likes everything to be under a class or an interface, so the prerequisite of writing a lambda function is that it must be written according to an interface.

The specific interface must have one interface method in it, it cannot have other interface method! This will be called the **functional interface**. You are allowed to have other methods with implementation in the interface, however, it cannot have any other interface method.

The way you would write a lambda expression in Java is as follows

```
parameter -> expression
// or
(parameter1, parameter2) -> expression
// or
parameter -> {
    // multi-line code
    // with explicit return
}
```

Just like in JavaScript you would use the `=>` (in Java is `->`) to indicate the expression of the lambda function

## What lambda function solves

Let's go through an example of what Lambda function is used to solve and how things work without it.

Let's say we are writing an interface called **StringFunction.java** it contain one interface method called **returnString** which will take in one parameter of String. The specification of **returnString** basically allows the implementer of the StringFunction interface to define a class that contains the returnString method to the string with the implementer's modification. For example, append a String to the parameter string.

```
interface StringFunction {
    public String returnString(String s);
}
```

```

}

class LmaoStr implements StringFunction {
    public String returnString(String s) {
        return s + "lmao";
    }
}

public class Test {
    public static void main(String[] args) {
        LmaoStr ls = new LmaoStr();
        System.out.println(ls.returnString("ayy "));
    }
}

```

As you can see to use this interface, you would have to first define a class that implements it, then instantiate an object of type `LmaoStr` then finally call the method to use it. This is just so much work to just implement a simple interface no? Lambda function aim to solve that.

With lambda function, the above code can be shorten to just:

```

interface StringFunction {
    public String returnString(String s);
}

public class Test {
    public static void main(String[] args) {
        StringFunction sf = (s) -> s + "lmao";
        System.out.println(sf.returnString("ayy "));
    }
}

```

You no longer have to define a explicit class just to implement one simple method. You can just write the lambda function and it will be the implementation of `returnString`.

## Other example of functional interface

If you use the `List.forEach` method it uses the `Consumer` interface which allows you to write lambda function like such:

```

ArrayList<Integer> arr = new ArrayList<>();
arr.add(1);

```

```
arr.add(2);
arr.add(3);
arr.add(4);
arr.forEach((i) -> System.out.println(i));
```

```
// Consumer.class
public interface Consumer<T> {
    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

As you can see the Consumer class is an interface in order for you to define the lambda function. `andThen` is not an interface method because it has implementation so it is fine.

## Instance method method references

Method references can be a bit confusion because of the way it is used. So method reference are again also only be with used the functional interface context.

It is a reference to the body of the function and use it to implement the interface, so it has even a shorter syntax than a lambda function, since you are just borrowing a method to implement it instead of writing your own.

The one that I want to go over in particular is instance method method references because the rest of the ones the input type parameter matches and is self-explanatory.

```
Function<Cat, String> f1 = Cat::getName;
System.out.println(f1.apply(cat1));

Function<Cat, String> f1 = in -> in.getName();
System.out.println(f1.apply(cat1));
```

Here is an example. As you can see the first and second way of writing using lambda function are exactly the same. The output is the same. `getName` is an instance method, so how does instance method matches the interface of Function? Which is

```
public interface Function<T, R> {
    R apply(T t);
```

```
}
```

While the getName method is

```
class Cat {  
    public String getName() {  
        return name;  
    }  
}
```

When you make a reference to an instance method of a class, it will automatically assume that the interface method you are implementing for, have its first parameter as this class type. So in this example it will assume that for **apply** the first parameter is **Cat t**. Because you are referencing an instance method of a class. Then it will invoke the actual instance method on that first parameter, which is how it works despite not having the same headers with the interface.

To drive the point home:

You can see that **getName** doesn't take any parameter, but it can still be used as an method reference for Function, because the method will be called on the actual object passed into apply (the first parameter of apply), and not used **independently**.

# Checked vs Unchecked Exception

## Exceptions

In Java exceptions (which differs from the C exception) there are two types of exceptions.

### Checked exceptions

Checked exceptions are exceptions that are checked at compile time. If some code in a method throws a checked exception, then the method must either handle the exception by surrounding it with try-catch or has to throw the exception to make the caller of the method handle it.

Exception such as **FileNotFoundException** are checked exception that you must handle.

Checked exceptions are sub-class of Exception that aren't sub-class of RuntimeException

### Unchecked exceptions

Unchecked exceptions are the opposite that aren't checked by the compiler, therefore, you are not forced to handle it with try-catch. It is up to the programmer to handle it or throw it.

Exception such as **NullPointerException** are unchecked.

This is why you can use Integer.parseInt without surrounding it with try-catch clause.

Unchecked exceptions are sub-class of RuntimeException

Fun fact: RuntimeException is actually a sub-class of Exception, however, it is treated differently by the compiler to be unchecked exception.

# Mockito how does it work?

## Background

So what is Mockito? It is framework that is used on top of testing framework library like JUnit test to provide mocking and stubbing capability of objects for your unit test or integration tests.

"Mocking is the act of removing external dependencies from a unit test in order to create a controlled environment around it. Typically, we mock all other classes that interact with the class that we want to test". This is so that you can get a consistent behavior regardless how the external dependency actually react.

**So we are basically pulling out the external dependency and substitute in a fake object in it's place to use for testing purposes only.**

Why do you need to mock (fake) an object? Well, if the external dependency that your code base depends on is slow, and requires reading large files, or needs a database connection setup, you wouldn't want to set that up every time your test is run right?

That's where mocking comes into play, you assume that the external dependency you use are working, then you will fake those method calls that your code base is calling on those external dependency to return values that you expect it to return.

## Mock? Stub?

In mocking theory:

- A stub is a fake class that comes with preprogrammed return values. This gives you full control of the object, letting you control the return value when a certain method is called.
- A mock is a fake class as well, that can be examined after the test is finished for its interaction with the class under test. You can whether a particular method is invoked and also see how many times that method has been invoked.

So stubbing is the act of faking a method. You choose how the method behaves. This is done using `when().thenReturn()` calls.

Mocking on the other hand is done by using `verify()` from the Mockito library, to inspect the mocked object.

In Mockito, it uses the terminology "mock "for both stub and a mock.

# Basic stubbing with mockito

Stubbing is done via the `when().thenReturn()` method call on the mocked object. For example:

```
when(entityManager.find(2)).thenReturn(1);  
when(entityManager.find(anyString())).thenReturn(5);
```

When the `find` method is called on `entityManager` and passed in the int value 2, then it will return 1.

When the `find` method is called on `entityManager` and passed in any string value, then it will return 5.

Assuming that the `find` method is overloaded of course.

## What happens if you don't stub method and call it?

By default, for all methods that return values, if it returns an object it will return null, an empty collection, or an appropriate primitive default value if the method returns a primitive such as 0 for integers, false for booleans and so on.

**This means that a mocked object doesn't inherit any of the original method that the class actually has!**

You can change this behavior to have the real methods from the class to be called when not stubbed by adding

```
mock(Example.class, Mockito.CALLS_REAL_METHODS);
```

However, this is not recommended since they could use fields that doesn't exist in the mocked object :).

# Basic mocking with mockito

Now we discussed stubbing how about mocking? How do we inspect whether or not the method is called in an object and how many it has been called?

We can do that by calling the `verify` function.

```
verify(emailSender).sendEmail(sampleCustomer);  
verify(emailSender, times(0)).sendEmail(sampleCustomer);
```

The first `verify` will check whether or not the object `emailSender` called `sendEmail` with the specified parameter once. By default if you don't provide the number of invocation to check it will default to

1. i.e. it should only be invoked once.

The second verify will check that the object didn't call `sendEmail` at all.

# Optional in Java

## Optional

Optional is an object that can be think of as a container to hold other objects. It can contain null or an instance of the class.

Why is this object needed? Well there are many places in your code that can return the object or null, i.e. your custom find method for example. And if you call method on the null object this is where you will be seeing `NullPointerException`.

So the normal way of dealing with this type of project is just to add a if-statement check like such:

```
Cat c = Database.findCat("Lucy");
if (c != null) {
    c.meow();
} else {
    System.out.println("No cat found");
}
```

Optional is created to be more explicit telling the programmer that hey the method that returns the value you ask for might or might not return null, so you have to be prepared to deal with it. No worries, optional also provides some API that deals with null pretty easily.

Let's take a look at some of them.

### ofNullable and of

To actually create an optional object you wouldn't use the constructor but instead use the static method, `ofNullable` or `of`.

They accept in the object that you want to put into the Optional container.

Use `ofNullable` if the object can be null.

Use `of` if the object you are 100% will not be null. Otherwise it will be a `NullPointerException` :)

### public T get()

Now you created your Optional object how do you get the object out of it if it has any? These following API are use to do that.

Now this method shouldn't really be used since the creator of Optional regret adding this into the API because it is so tempting to use it like such.

```
Optional<Cat> catOpt = Optional.ofNullable(cat);  
catOpt.get().meow();
```

What if `cat` is null? You will again get `NullPointerException`

## public boolean isPresent()

You can use this method to tell whether or not the value is present, it will return `true` if it is present, otherwise `false`.

You can use `isPresent()` with `get()`

```
Optional<Cat> catOpt = Optional.ofNullable(cat);  
if (catOpt.isPresent()) {  
    catOpt.get().meow();  
}
```

But this is pretty much the same god damn code without Optional right? Right, but you make it clear with Optional that you the programmer acknowledged the fact that the object you get out from the `catOpt` could be null, as oppose to without Optional, you aren't sure if it can be null or not.

Rather than using `get()` and `isPresent()`, there are other more useful API methods.

## public T orElse(T other)

This method can be called on an Optional object to retrieve the value inside it, if it doesn't contain one, i.e. it is null, then the `other` is returned.

As an example:

```
Optional<Cat> catOpt = Optional.ofNullable(cat);  
Cat catOutTheBox = catOpt.orElse(new Cat("Default"));  
System.out.println(catOutTheBox.getName());
```

Now as you can see, if the cat exists in the Optional object, then it will return that and that cat's name from Optional will be printed. However, if `cat` is null, then a default cat is returned with name "Default".

## public Optional<U> map(mapper)

Now this method is super interesting because you can use it to map your Optional object into something else and get a Optional object of that type back.

In more formal terms, if the value exists in the called Optional object, it will apply the mapping function (could also be a lambda function, and if the result is non-null, it will return an Optional object of that particular type. Otherwise, if the value doesn't exist, return an empty Optional of that type.

Let's look at the example:

```
Optional<Cat> catOpt = Optional.ofNullable(catOpt);  
Optional<String> outOpt = catOpt.map(Cat::getName);
```

In this case, the map function calls getName on the Cat object inside the Optional if it exists, and return out an Optional of type String because getName returns a String. So ultimately you get back an Optional that may or may not contains a String depending on whether or not `catOpt` contains the Cat object.

# Java Nested Class

# Stream map vs flatMap

## Stream API

In Stream API there is both map and flatMap that's available for use. They are both used for transforming a stream into another, but does it a little bit differently.

## Map

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

Map has the above Java method header. It takes in a function in which it takes in the stream's type and then map it to another type. Basically, transforming each of the elements within the stream into a different element depending on the function that's provided.

Then it will return a new Stream which has the mapped element's type and each of the mapped element's value.

For example:

```
Stream.of(1, 2, 3).map(num -> num * 2).toList();
```

This will output a list of value `[2, 4, 6]`. Simple, each of the number is transformed by doubling it. A one to one mapping.

## FlatMap

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

FlatMap on the other hand as it's name suggests, does two operations, it first it does the mapping, in this case, the one element can be mapped to multiple elements (Another Stream if you will), then each of the Streams are flatten into a single stream.

Let's understand what a flattening operation does

## Flatten Example

Let's say we have the following array:

```
{{1,2,3,4},{5,6,7},{8,9}}
```

This is a 2D array, where each of the element in the 2D array is a 1D array. If we want to flatten it it means to converted the nested array into one dimension less, unwrapping the inner nested array if you will.

```
{1,2,3,4,5,6,7,8,9}
```

This is done typically to make the array easier to work with for analytical purposes.

## Back to flatMap

Now after understanding flatten, we can go back to understand how flatMap works.

It is the same idea, you have a function that takes in the input, but the function will map it to another Stream, meaning, one input can become 0 or more output, **whereas compared to map it is one to one**. If you have [1, 2, 3] the function will:

- Take the 1 and might produce Stream(1, 2)
- Take 2 and produce Stream() - Empty Stream
- Take 3 and produce Stream(3, 4)

Then at the end flatMap will take all three of the Stream. Combine it and then flatten it, so the process goes

- Stream(Stream(1, 2), Stream(), Stream(3, 4))
- Stream(1, 2, 3, 4)

That's it!

## Another way to think about it

You can think about the mapping as list outputs and the flattening as unwrapping the nested array.

If you have say:

```
const words = ["hello", "world"];  
const characters = words.flatMap(word => word.split(""));
```

- "hello" → split("") → ["h", "e", "l", "l", "o"].
- "world" → split("") → ["w", "o", "r", "l", "d"]
- flatMap flattens [{"h", "e", "l", "l", "o"}, {"w", "o", "r", "l", "d"}] into a single array

## 3D example

A more complex example

```
const array = [1, 2, 3];
const result = array.flatMap(x => [[x], [x * 2]]);
console.log(result); // [[1], [2], [2], [4], [3], [6]]
```

In this case we are taking an array of numbers, taking each number and mapping it into a 2D list.

- 1 → [[1], [2]]
- 2 → [[2], [4]]
- 3 → [[3], [6]]
- flatMap flattens one level: [[[1], [2]], [[2], [4]], [[3], [6]]] becomes [[1], [2], [2], [4], [3], [6]].

## flatMap implementation

```
function customFlatMap(array, callback) {
  // Map each element to an array using the callback
  const mapped = array.map(callback);
  // Flatten the resulting arrays by one level
  return mapped.reduce((flat, current) => flat.concat(current), []);
}
```

An example flatMap implementation can be done above. The callback is used to do the function, it maps the element to an array.

Then `reduce` is used to flatten the array into a list that is one nested level less.

```
const words = ["hello", "world"];
const characters = customFlatMap(words, word => word.split(""));
console.log(characters); // ["h", "e", "l", "l", "o", "w", "o", "r", "l", "d"]
```

- Callback: word => word.split("") returns ["h", "e", "l", "l", "o"] for "hello", ["w", "o", "r", "l", "d"] for "world"
- Map: Produces [{"h", "e", "l", "l", "o"}, {"w", "o", "r", "l", "d"}]
- Reducing: Merges into ["h", "e", "l", "l", "o", "w", "o", "r", "l", "d"] - You are just basically taking each of the array and concatenating the element into an aggregating list (which starts off

as empty)