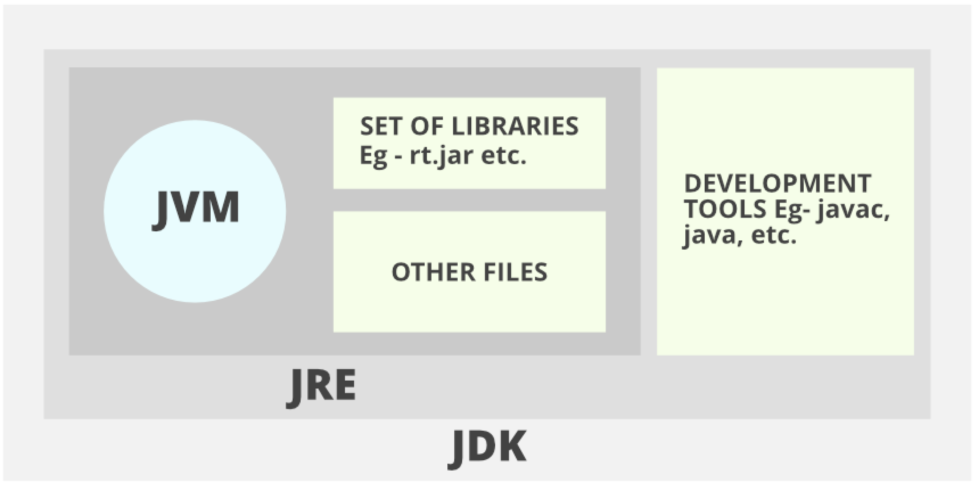
Java Project Management

First, let's clear some common terms in the Java development ecosystem.

First, you must understand the differences between **JDK**, **JRE**, and **JVM**. We'll explain it from bottom up:


JVM
SET OF LIBRARIES Eg - rt.jar etc.
OTHER FILES
DEVELOPMENT TOOLS Eg- javac, java, etc.
JRE
JDK

1. The **Java Virtual Machine (JVM)** is the component which simply executes your compiled Java bytecode. But this is actually not enough to run Java programs because it lacks essential libraries and support files needed to make Java applications fully functional.
2. The **Java Runtime Environment (JRE)** is a package that includes not only the JVM but also the core libraries and additional resources that Java applications rely on to operate. The JRE offers a complete environment that enables Java programs to run smoothly on your system.
3. The **Java Development Kit (JDK)** is a package that includes additional tools needed to write, compile, and debug Java code.

Open the SpotifyCatalogAPI project in IntelliJ

1. Enter the project page in GitHub: https://github.com/alonitac/SpotifyCatalogAPI. This is a Java project that we'll be working on during the course.
2. Fork this repo by clicking **Fork** in the top-right corner of the page.

Because this project repository is not owned by you, you don't have permission to directly make changes. Thus, we want you to fork this repository. When you fork a repository, you create a copy of the original repository under your own GitHub account. This copy is completely separate from the original repository, so you can make changes to it without altering the original project.

3. Launch IntelliJ IDEA.

If the Welcome screen opens, click **Clone Repository**. Otherwise, go to **File | New | Project from version control** in the main menu.

4. Enter **your** fork URL (change YOUR_GITHUB_ACCOINT accordingly): https://github.com/YOUR_GITHUB_ACCOINT/SpotifyCatalogAPI.git.
5. You might be prompted to configure your project structure. If you don't see such screen, click on the settings wheel in the top right corner, and then choose **Project Structure**.
6. From the JDK list, select the **Oracle OpenJDK 23**:
    - If the JDK is installed on your computer, but not defined in the IDE, select **Add JDK** and specify the path to the JDK home directory.
    - If you don't have the necessary JDK on your computer, select **Download JDK**.

Maven intro

Maven is a tool helping you to manage your project throughout the build lifecycle. **Build Lifecycle**, simply, is the process of taking your .java files and turning them into an executable file that can be run in production systems, this file is called an **Artifact**. The build lifecycle includes all the different phases you should go through during the process - install dependencies, compile the code, test it, etc...

Download and install

1. For Windows users, follow this guide.
2. Ubuntu users:

```
sudo apt update

sudo apt install maven
```

3. Make sure Maven is available by open up a Terminal and type:

```
$ mvn -v

Apache Maven 3.9.5 (57804ffe001d7215b5e7bcb531cf83df38f93546)

Maven home: /mvn

...
```

The project groupId and artifactId properties

In each Maven project you must define two properties called **groupId** and **artifactId**.

Taking a look on a file called pom.xml in our **SpotifyCatalogAPI** project files, you'll notice that our project groupId is com.example and the artifactId is catalog.

A groupId is the project's group or organization name, typically following a reverse domain name convention (e.g., com.example, edu.mit, org.who, bla.bla). An artifactId is the name of the specific module within the group name.

Note

You don't need to be the actual owner of the example.com domain to create such groupId. It's simply a convention used to avoid naming conflicts, and you can use any domain name that you own or don't own.

The file system of the project is structured in corresponding to the groupId and artifactId:

```
.
├── pom.xml
└── src
  ├── main
  │   ├── java
  │   │   └── com          <──┌─── groupId
  │   │     └── example     <──┘
  │   │       └── catalog  <──────── artifactId
  │   └── resources
  │     ├── application.properties
  │     └── data
  └── test
    └── java
      └── com
        └── example
          └── catalog
```

- The default source directory for the .java files forming you app is src/main/java
- Test files are under src/test/java
- App resources (config files, data, any file that is not a Java file) under src/main/resources

Keep in mind that the above structure is the standard (and recommended) convention in projects managed by Maven, but not a mandatory.

Maven build lifecycle and phases

Many programming projects share common phases in order to make your code runnable, as follows: First, you **validate** that everything is set up correctly, like the list of dependencies, some configurations. Next, you **compile** the code into executable files, **test** it to catch bugs, then **package** the application into a single .jar or .war file. Lastly, you **verify** its end-to-end functionality, **install** or **deploy** it for production use.

Similarly, Maven defines these **phases** (only a partial list):

- Default:
    - validate - validate the project is correct and all necessary information is available
    - compile - compile the source code of the project
    - test - test the compiled source code (unittests)
    - package - take the compiled code and package it, e.g. a .jar file
    - verify - run any checks on results of integration tests to ensure quality criteria are met
    - install or deploy - install the package as a dependency of another project, or upload it remotely, for using by others
- Clean:
    - clean - clean old build artifacts from previous runs
- Site:
    - site - generate a project documentation "website" based of the documentation found in your code

These are the main and most commonly used phases, but there are many more available in Maven.

As can be seen, the above list of phases is divided into groups. Why is that? Because the phases in each group depend on each other. You can’t perform the test phase if the project hasn’t been compiled beforehand.

A group of phases that depend on one another and must be executed **sequentially** define a Build lifecycle.

Maven defines only three build lifecycles: default, clean, and site.

Let's see it in action by executing the following commands in the terminal (Alt + F12 in IntelliJ to open up a terminal session):

1. Validate the project (checks if the project setup is correct and all necessary information is available):

```
$ mvn validate

[INFO] Scanning for projects...

[INFO]

[INFO] -----------------------< com.example:catalog >------------------------

[INFO] Building catalog 0.0.1-SNAPSHOT

[INFO] -------------------------------[ jar ]--------------------------------

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------
```

2. Compile the source code (compiles the project's source files into bytecode):

```
$ mvn compile

[INFO] Scanning for projects...

[INFO]

[INFO] -----------------------< com.example:catalog >------------------------

[INFO] Building catalog 0.0.1-SNAPSHOT

[INFO] -------------------------------[ jar ]--------------------------------

[INFO]

[INFO] --- maven-resources-plugin:3.3.1:resources (default-resources) @ catalog ---

[INFO] Copying 1 resource from src/main/resources to target/classes

[INFO] Copying 3 resources from src/main/resources to target/classes

[INFO]

[INFO] --- maven-compiler-plugin:3.13.0:compile (default-compile) @ catalog ---

[INFO] Nothing to compile - all classes are up to date.

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------
```

After running mvn compile, the target/ directory is created (or updated) and contains the compiled classes and resources for your project. In this case, you will find the compiled .class files from the src/main/java directory inside target/classes.

Additionally, any resources (like configuration files) from the src/main/resources directory will be copied into target/classes as well.

3. Run unit tests and package the project into a JAR file (runs tests on the compiled code and then packages the project into an artifact):

```
$ mvn package

....
```

As mentioned, the mvn package command internally completes the validate, compile, test and package phases.

Exploring the output, you'll see the compilation process, as well as one executed test (successfully) before Maven packed the project into an executable file under target/catalog-0.0.1-SNAPSHOT.jar. This .jar file contains the compiled code, resources, and dependencies required to run or distribute the application.

As said, this packaged output is also known as an **Artifact**.

You can run the app by java -jar <path-to-your-jar-file>, for example:

```
java -jar target/catalog-0.0.1-SNAPSHOT.jar
```

4. Finally, clean the project (removes previously generated build artifacts):

```
$ mvn clean

[INFO] Scanning for projects...

[INFO]

[INFO] -----------------------< com.example:catalog >------------------------

[INFO] Building catalog 0.0.1-SNAPSHOT

[INFO] --------------------------------[ jar ]---------------------------------

[INFO]

[INFO] --- maven-clean-plugin:3.3.2:clean (default-clean) @ catalog ---

[INFO] Deleting /home/alon/Documents/SpotifyCatalogAPI/target

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------
```

The mvn clean command deletes the target/ directory, where Maven stores all the compiled classes, JAR files, and other build artifacts.

It's very common to clean previous artifacts before starting a new build to ensure that no outdated files are left from previous builds:

mvn clean package

Maven first run the clean phase, then the package phase (which internally completes the validate, compile, test and package phases). Notice that the clean phase is not part of the **default** lifecycle (the lifecycle that all the above phases belong to), but it belongs to a different build lifecycle, called the **clean** lifecycle (yes... confusing - a **clean** lifecycle which has a phase called clean).

Note

Here is a table summarizing the three Maven lifecycles and the main phases under each lifecycle.

default	**clean**	**site**
validate	pre-clean	pre-site
compile	clean	site
test	post-clean	post-site
package		
verify		
install		
deploy		

The POM file

So far, we've talked about the general build lifecycle for a Java project. But how does it work for a specific project? How to configure maven to pack my project with the list of my project dependencies? How to configure the test phase to execute different kind of test?

The **Project Object Model (POM)** file, named pom.xml, is an XML file that contains information used by Maven to build and manage the project.

When executing a phase, Maven looks for the POM in the current directory and follow the configurations there.

Tip

Take some time to learn the XML format if you're not already familiar with it.

Here's a general structure of a pom.xml:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">

        <modelVersion>4.0.0</modelVersion>

        <groupId>com.example</groupId>

        <artifactId>demo</artifactId>

        <version>0.0.1</version>


        <properties>

                ...

  </properties>



        <dependencies>

    ...

        </dependencies>



        <build>

    ...

        </build>


</project>
```

1. Every POM requires <groupId>, <artifactId>, and <version> to be configured. These three values form the project's identifier, in the form of <groupId>:<artifactId>:<version>.

As for the example above, its fully qualified name is com.example:demo:0.0.1.

2. The <properties> entry allows for defining variables to be used within the POM.
3. The <dependencies> entry specifies external libraries required by the project.
4. The <build> entry allows customization of the build phases themselves.

Versioning your app

While you are totally free to choose the way to denote the version of your app in the <version> tag, many organizations follow specific tagging convention called **semantic version**, or **SemVer**.

Take a look on the <version> specified in the POM: 0.0.1-SNAPSHOT.

- The SNAPSHOT value refers to the 'latest' code development version. It provides no guarantee the code is stable. Conversely, the code in a 'release' version (any version value without the suffix SNAPSHOT) should be stable an unchanging. In other words, a SNAPSHOT version is the 'development' version before the final 'release' version.
- As for 0.0.1, each component has a specific meaning. Given a version number MAJOR.MINOR.PATCH:
    i. Increment the MAJOR version when you make **incompatible** API changes.
    ii. Increment the MINOR version when you **add functionality** in a backward compatible manner.
    iii. Increment the PATCH version when you make backward compatible **bug fixes**.

Plugin goals

What actually happens when you execute a Maven phase, e.g. mvn test?

When you run a Maven command like mvn test, Maven executes a series of tasks associated with that phase. These tasks are called **plugin-goals**, which are specific actions provided by Maven plugins.

To see what tasks (plugin-goals) a phase performs, you can use:

```
$ mvn help:describe -Dcmd=test

[INFO] 'test' is a phase corresponding to this plugin:

org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
```

```
It is a part of the lifecycle for the POM packaging 'jar'. This lifecycle includes the following
phases:

* validate: Not defined

* initialize: Not defined

* generate-sources: Not defined

* process-sources: Not defined

* generate-resources: Not defined

* process-resources: org.apache.maven.plugins:maven-resources-plugin:2.6:resources

* compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:compile

* process-classes: Not defined

* generate-test-sources: Not defined

* process-test-sources: Not defined

* generate-test-resources: Not defined

* process-test-resources: org.apache.maven.plugins:maven-resources-
plugin:2.6:testResources

* test-compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile

* process-test-classes: Not defined

* test: org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test

* prepare-package: Not defined

* package: org.apache.maven.plugins:maven-jar-plugin:2.4:jar

* pre-integration-test: Not defined

* integration-test: Not defined

* post-integration-test: Not defined

* verify: Not defined

* install: org.apache.maven.plugins:maven-install-plugin:2.4:install

* deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy


[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------
```

This output shows that the test phase relies on the maven-surefire-plugin, which runs unit tests in your project.

Keep in mind when you execute mvn test, Maven runs the test phase **and all preceding phases in the lifecycle**. Based on the provided list, the following plugin-goals that will be executed:

```
org.apache.maven.plugins:maven-resources-plugin:2.6:resources

org.apache.maven.plugins:maven-compiler-plugin:3.1:compile

org.apache.maven.plugins:maven-resources-plugin:2.6:testResources

org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile

org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
```

If you want to modify how the test phase behaves, you can customize the Surefire Plugin (or replace it with another plugin) in the POM file:

```
<build>

  <plugins>

    <plugin>

      <groupId>org.apache.maven.plugins</groupId>

      <artifactId>maven-surefire-plugin</artifactId>

      <version>2.22.2</version>

      <configuration>

        <includes>

          <include>**/*Test.java</include>

        </includes>

      </configuration>

    </plugin>

  </plugins>

</build>
```

This above configures the plugin to include only test files in the form **/*Test.java.

The Java classpath

The **classpath** in Java refers to the location (or set of locations) where the Java Virtual Machine (JVM) looks for class files, resources, and libraries needed to run a Java application. It essentially tells the JVM where to find the compiled .class files, libraries (JARs), and resources like configuration files, properties, or YAML files.

In a Maven project, the classpath is typically constructed by Maven itself: Maven uses the target/classes directory to store the compiled classes and resources. It also includes any dependencies listed in your pom.xml file (JAR files located in the target/dependency folder or downloaded from Maven Central).

Maven in IntelliJ

IntelliJ IDEA supports a fully-functional integration with Maven, it should automatically read the pom.xml file from your project files and relate maven as the build tool (there are other build tool used in Java, like Grade, Ant, etc...).

Every time you manually change the pom.xml file in the editor, you need to synchronize the changes. IntelliJ IDEA either displays a notification, or you can do it manually from the Maven menu (click on the "M" button at the right side menu).

Exercises

✏️ Using Maven

Copy the following controller code into src/main/java/com/example/catalog/controller/PracticeController.java. Don't worry if you don't understand every piece of code, soon we'll cover this content.

```
package com.example.catalog.controller;


import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.core.io.ClassPathResource;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.apache.commons.lang3.StringUtils;


import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import java.util.Map;


@RestController
public class PracticeController {


  private final ObjectMapper objectMapper = new ObjectMapper();


  @GetMapping("/capitalize")
  public String capitalizeName(String name) {
    return StringUtils.capitalize(name);
  }


  /**
   * @return sample sorted songs list
   */
  @GetMapping("/sampleSongsNames")
  public List<String> getSampleSongs() {
    List<String> songNames = Arrays.asList(
            "Kill Bill", "Daydreaming", "Havana (feat. Young Thug)"
    );

    // TODO sort songs by names

    return songNames;
  }

  /**
   * @return the song object with the highest popularity
   * @throws IOException
   */
  @GetMapping("/mostPopularSongs")
  public Map<String, Object> getMostPopularSongs() throws IOException {
    ClassPathResource resource = new ClassPathResource("data/popular_songs.json");
    JsonNode songsNode = objectMapper.readTree(resource.getFile());
    List<Map<String, Object>> songsList = objectMapper.convertValue(songsNode, List.class);

    return songsList.get(0);  // TODO return the song with the highest popularity
  }

}
```

Your goal is to use mvn command to compile, pack and run the app.

- The import org.apache.commons.lang3.StringUtils; statement depends on the org.apache.commons:commons-lang3:3.12.0 package, make sure you define it as a dependency in the POM file.
- You should run the app by the java -jar ... command.
- When the app is running, open up your web browser and visit http://localhost:8080/. Make sure you get a response.
- When done, you should **commit your changes** and **push them to GitHub**.

✏️ Implement very easy features

In the PracticeController, implement the following two functions:

- getSampleSongs()
- getMostPopularSongs()

Again, don't worry if you don't understand every piece of code. Try to follow you common sense and implement the TODOs.

- Check your solution by running the app the visit the http://localhost:8080/sampleSongsNames and http://localhost:8080/mostPopularSongs endpoints to see the expected output.
- After the changes you've made, what should be the app version in the <version> entry in the POM? Change the version accordingly.
- Commit and push your changes to GitHub.

✏️ Bind a plugin to a phase

In your pom.xml file, add the maven-clean-plugin to the <build> section, and bind the plugin's execution to the validate phase.

The effective result would be that running mvn validate effectively behaves like running mvn clean validate.