

# ECS 160 – Discussion

## Setting Up Your Environment

Instructor: Tapti Palit

Teaching Assistant: **Gabe Bai**



# Agenda

- **What is JDK?**
- Maven Build System
- Demo: adding dependencies and plugins



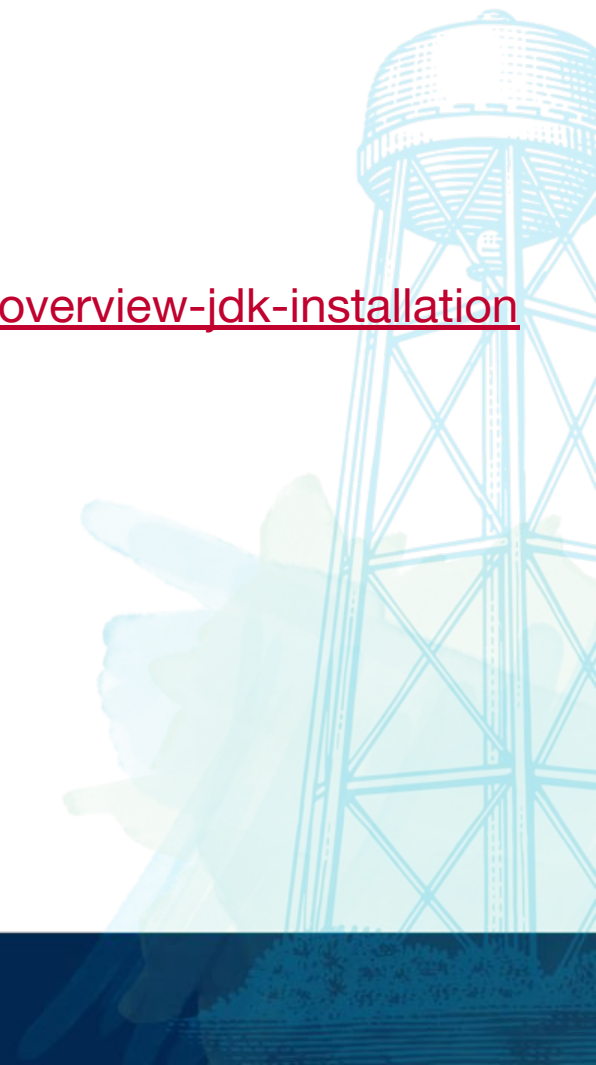
# JDK

- “Java Development Kit”
  - Provides compiler, *javac*, and runtime, *java*
- JRE
  - Runtime environment -> contains the *java* launcher
  - Contains JVM and core libraries needed to run Java
- JVM
  - Virtual machine that executes compiled files
  - This is what makes Java “write once, run anywhere”



# Installing JDK

- Differs by OS
- Please refer to the following documentation:
  - <https://docs.oracle.com/en/java/javase/11/install/overview-jdk-installation.html>



# Agenda

- What is JDK?
- **Maven Build System**
- Demo: adding dependencies and plugins



# Compiling a Java Program

- *javac* compiles source code to binaries the Java VM can run
- Suppose we want to compile

```
// src/HelloWorld.java
```

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

- `javac src/HelloWorld.java \newline java src/HelloWorld`

# Compiling a More Complicated Program

- Suppose

- src/
  - App.java
  - utils/
    - Helper.java

```
import utils.Helper;

public class App {
    public static void main(String[] args) {
        System.out.println(Helper.greet());
    }
}
```

- `javac src/utils/Helper.java src/App.java \newline java src/App`

# Adding External Libraries

- If we wanted to use external libraries:
  - Manually download .jar files
  - Download it to some *lib/* folder
  - Separately compile *those*
- If you forget a library...
- If a library has updated...





# Pain Points

- Extremely tedious
- Extremely fragile builds
- Constant recompiles
- Manual dependency identification
- Difficult onboarding
- Inconsistent builds *because of these reasons*
- Manual packing and distribution



# Maven

- Build automation and project management tool for Java
  - Amongst others, such as Gradle!
- Runs on top of JDK
- Key file: *pom.xml*
  - Handles dependency management
  - Provides Project Object Model (POM)
- NOTE: XML
  - Tool to store **metadata** – data about data
  - Analogous (and older) than JSON



# pom.xml

- Contains information about
  - Project – coordinates, metadata
  - Configuration details – build config, dependencies
- Contains default values;
  - We can add other values for dependencies, plugins, project version, description, developers, mailing lists [1]
- Pom.xml is a configuration file for your project
  - But also the blueprint that Maven uses to put together your project

[1] Source: <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>

# Coordinates

- Unique identifier for a dependency
  - Managed by Maven itself
- For example:
  - Spring Boot
    - `org.springframework.boot:spring-boot-starter-web:3.2.0`
- When you declare a line in `pom.xml`,
  - Maven organizes the relevant ***values inside the tags***
    - And combines them into coordinates to obtain `.jar` files

# Maven Build Cycle

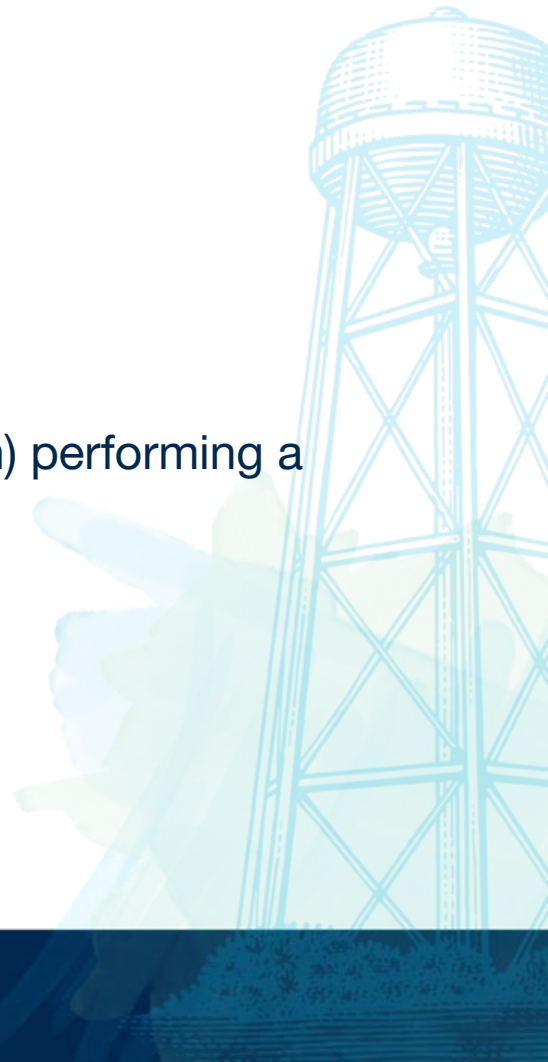
- Maven is built around the “build lifecycle”
  - Built-in: default, clean, site
- Default – Project deployment
- Clean – Project cleaning
- Site – Creation of website
- **Lifecycles** include **build phases**, which are built of **plugins**
  - When you run a phase, Maven executes all previous ones in that lifecycle automatically
  - *mvn help:describe -Dcmd=package*

# Phases in the Default Lifecycle

- **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 using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **package** - take the compiled code and package it in its distributable format, such as a JAR.
- **verify** - run any checks on results of integration tests to ensure quality criteria are met
- **install** - install the package into the local repository, for use as a dependency in other projects locally
- **deploy** - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

# Plugins

- Code modules
  - E.g. Clean, compiler, deploy, failsafe, install
- They define a goal
- Analogy:
  - Each step (phase) in a factory has a machine (plugin) performing a specific (goal)





# Examples of Plugins

- maven-compiler-plugin
  - Compiles .java files into .class files; compile phase
  - *mvn compile*
    - Invokes the compiler plugin, javac
- maven-surefire-plugin
  - Runs unit tests
  - *mvn test*
    - Invokes JUnit (for unit tests)
- Plugins are declared in pom.xml





# Agenda

- What is JDK?
- Maven Build System
- **Demo: adding dependencies and plugins**



# Demo Prereqs

- Make sure you have JDK installed, and on your path:
  - `java -version`
  - `javac -version`
- Maven as well
  - `mvn -v`
- `target/` – where everything generated is stored
- `mvn archetype:generate \`
- `-DgroupId=com.example \`
- `-DartifactId=hello-maven \`
- `-DarchetypeArtifactId=maven-archetype-quickstart \`
- `-DinteractiveMode=false`



# Commands

- `cd` into the relevant directory
- `mvn compile`
- `mvn test`
- `mvn package`
- `java -cp target/hello-maven-1.0-SNAPSHOT.jar com.example.App`

# Getting rid of that tag

```
<build>  
  <plugins>  
    <plugin>  
      <groupId>org.codehaus.mojo</groupId>  
      <artifactId>exec-maven-plugin</artifactId>  
      <version>3.1.0</version>  
      <configuration>  
        <mainClass>com.example.App</mainClass>  
      </configuration>  
    </plugin>  
  </plugins>  
</build>
```

