香 港 中 文 大 學（深 圳）
The Chinese University of Hong Kong, Shenzhen

CSC1004 Computational laboratory using JAVA

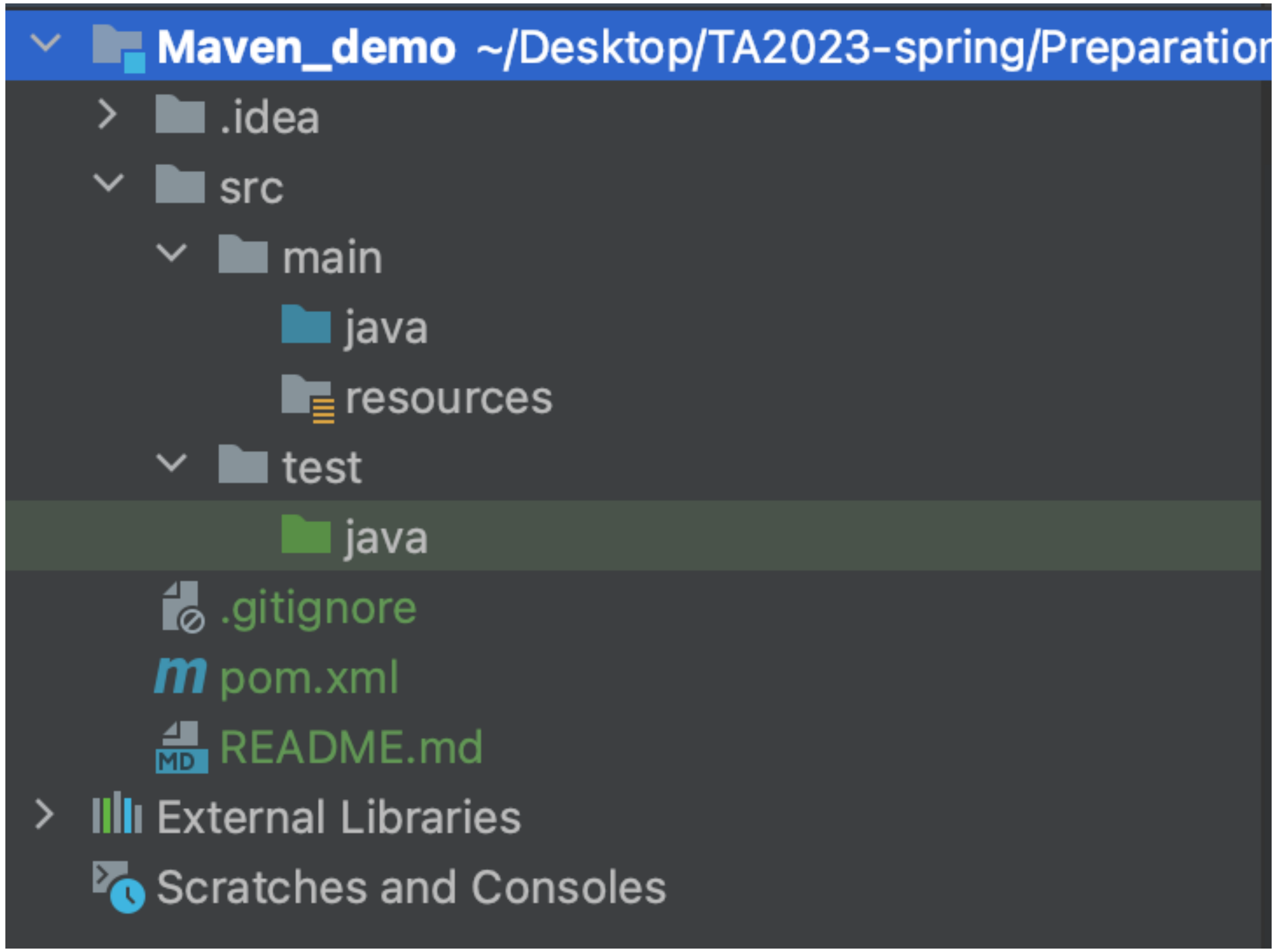
Tutorial 2

Contents

What is MAVEN?

- Project Management Tool for JVM languages.
- Major utilities:
  - Building source codes
  - Testing
  - Packing into .jar, .war or .ear
  - Generate Java Docs
  - Manage Dependencies
- Also called Build Tool or Dependency Management Tool

Project Structure



```
Maven_demo ~/Desktop/TA2023-spring/Preparatior
  .idea
  src
    main
      java
      resources
    test
      java
  .gitignore
  pom.xml
  README.md
External Libraries
Scratches and Consoles
```

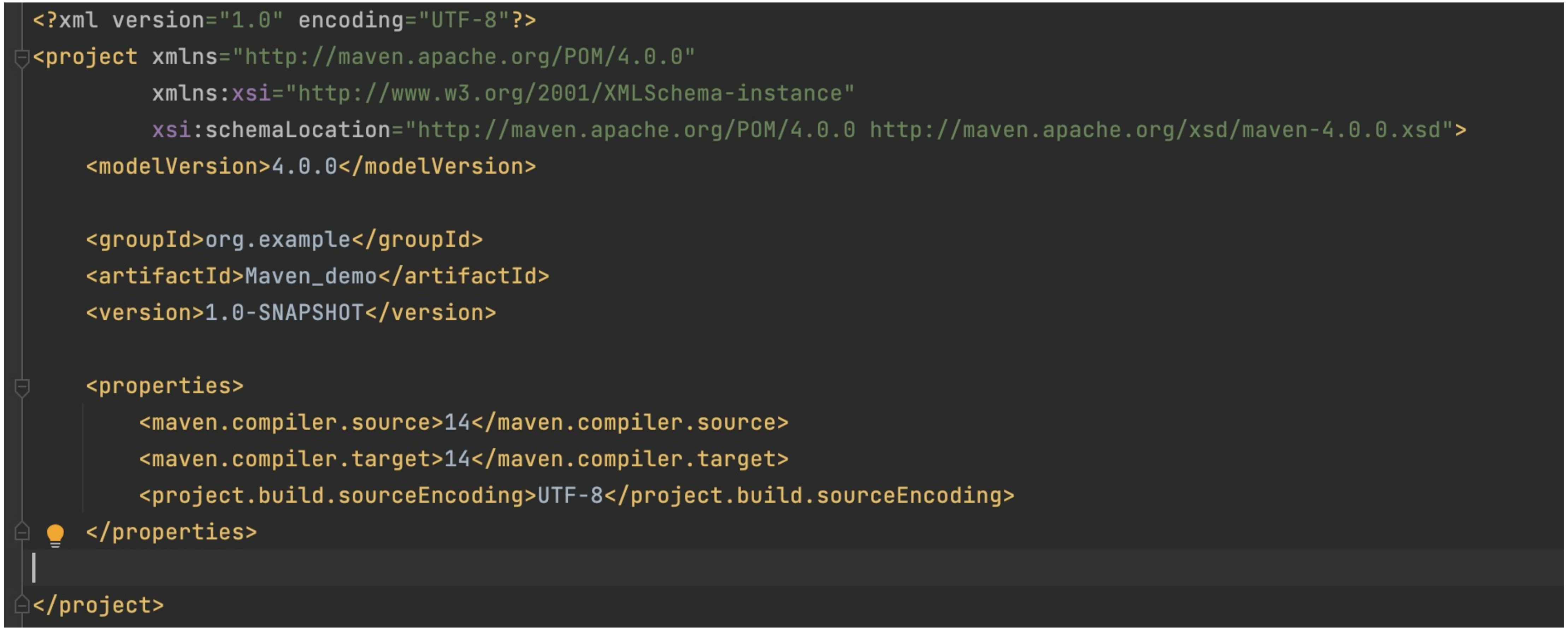
pom.xml

- POM stands for **Project Object Model**, and it is the core of a project's configuration in Maven. It is a single configuration XML file called pom.xml that contains most of the information required to build a project.
- The role of a POM file is to describe the project, manage dependencies, and declare configuration details that help Maven to build the project.
- There are three kinds of pom file:
    - Supper pom
    - Simplest pom
    - Effective pom

pom.xml

- **Super pom**: The super POM file defines all the default configurations. Hence, even the simplest form of a POM file will inherit all the configurations defined in the super POM file.
- **Simplest pom**: The simplest POM is the POM that you declare in your Maven project. In order to declare a POM, you will need to specify at least these four elements: *modelVersion*, *groupId*, *artifactId*, and *version*. The simplest POM will inherit all the configurations from the super POM.
- **Effective pom**: Effective POM combines all the default settings from the super POM file and the configuration defined in our application POM.

pom.xml



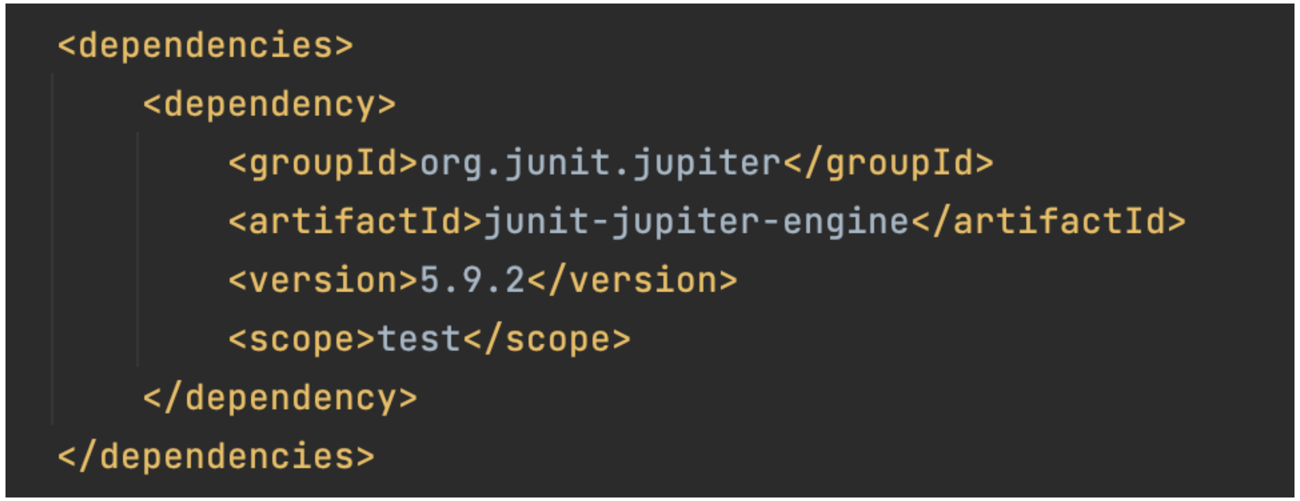
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Maven_demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>14</maven.compiler.source>
        <maven.compiler.target>14</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

pom.xml

- **modelVersion** modelVersion specifies which version of the descriptor the pom.xml conforms to. Maven 2 and 3 only have 4.0.0.
- **groupID**: the id for the project group.
- **artifactID** : the id for the artifact (project).
- **Version**： SNAPSHOT， RELEASE

Dependency

- You need to specify the dependency you use in the pom.xml
  - If it is not in your local repository, maven will connect to the remote repository and download it to your local repository.
- You need to take care about the scope of the dependency.
- You could download the dependencies from https://mvnrepository.com

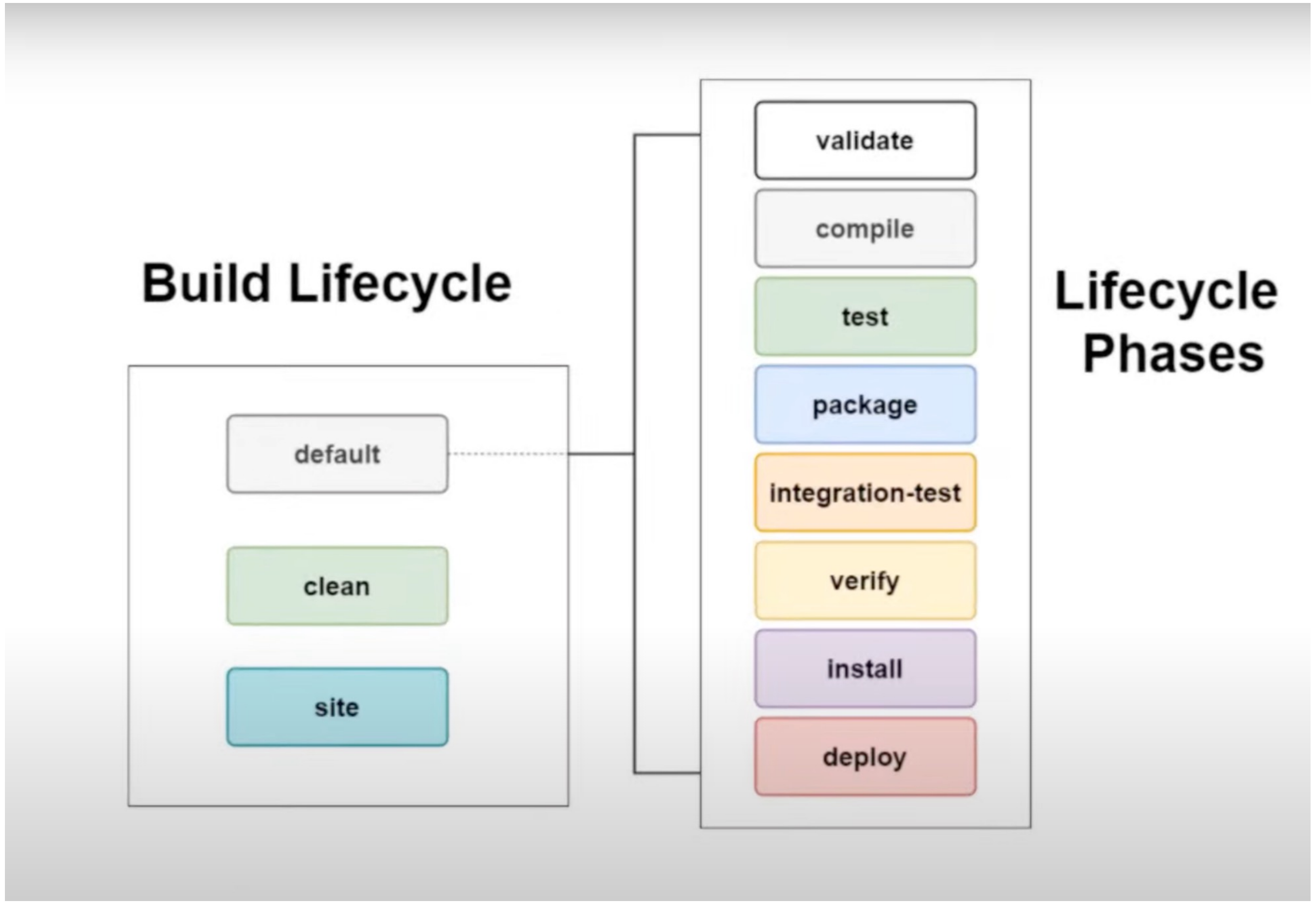


```xml
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.9.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Dependency -- Scope

- **Compile**: available at compile, test, and run(deploy).
- **Provided**: available at compile, test.
- **Runtime**: available test, and run(deploy).
- **Test**: available at test phase
- **System**: similar to Provided, path to JAR should be provided manually using <systemPath>

Build lifecycle and plugins

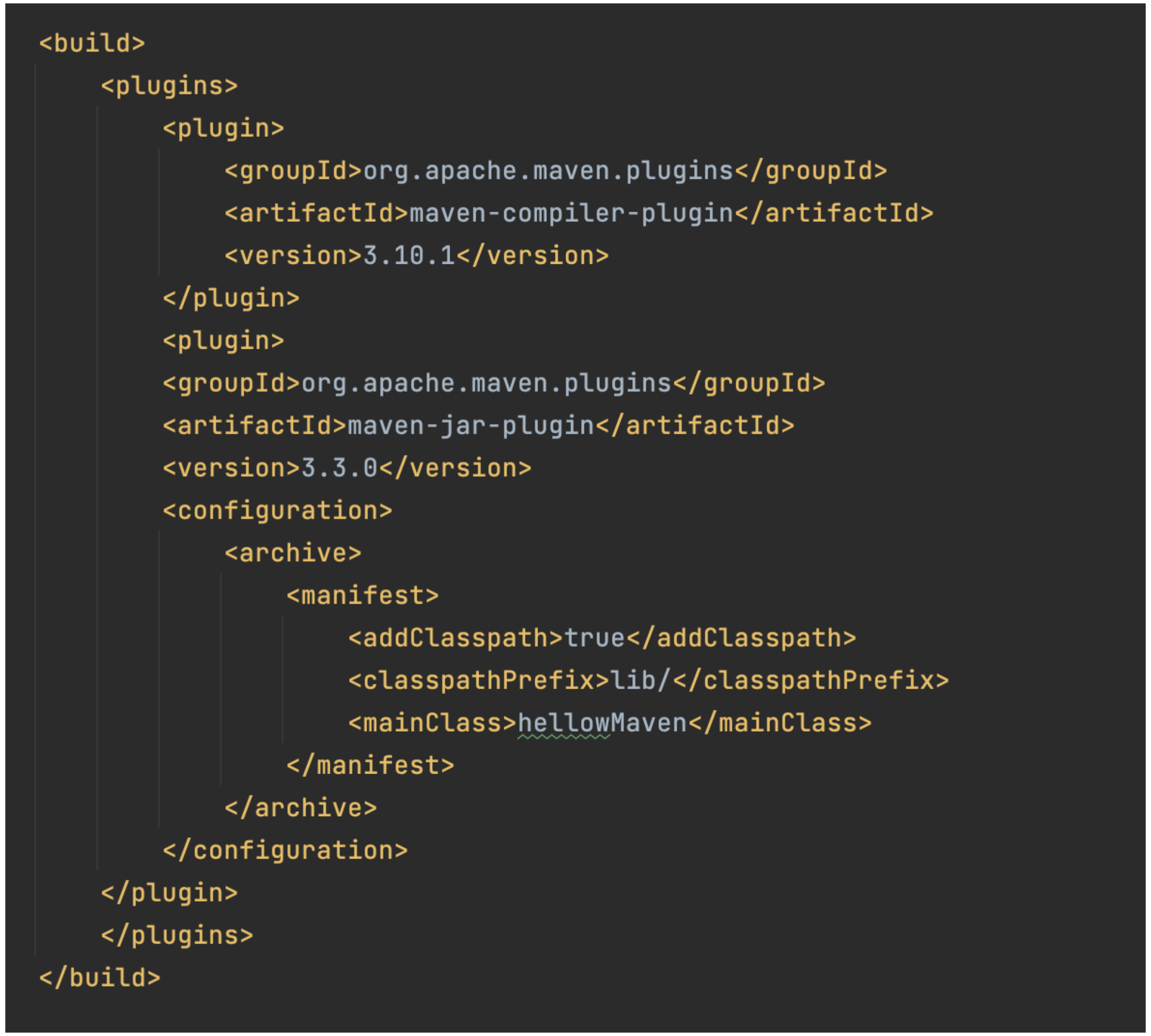


- Plugins enable us to run the lifecycle phases in our maven project.
- Each plugin is associated to a certain phase

Build lifecycle and plugins

Lifecycle Phase	Task
validate	Verify that the project is correct and that all the necessary information is available.
compile	Compile the code and output to the target file
package	Packing the source code
test	Run the test cases, generate test reports
verify	Run any checks to verify that the packaging is valid and meets quality standards.
install	Deploy the project to the local repo
deploy	Deploy the project to the remote repo

Build lifecycle and plugins



```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.10.1</version>
        </plugin>
        <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>3.3.0</version>
        <configuration>
            <archive>
                <manifest>
                    <addClasspath>true</addClasspath>
                    <classpathPrefix>lib/</classpathPrefix>
                    <mainClass>hellowMaven</mainClass>
                </manifest>
            </archive>
        </configuration>
    </plugin>
    </plugins>
</build>
```

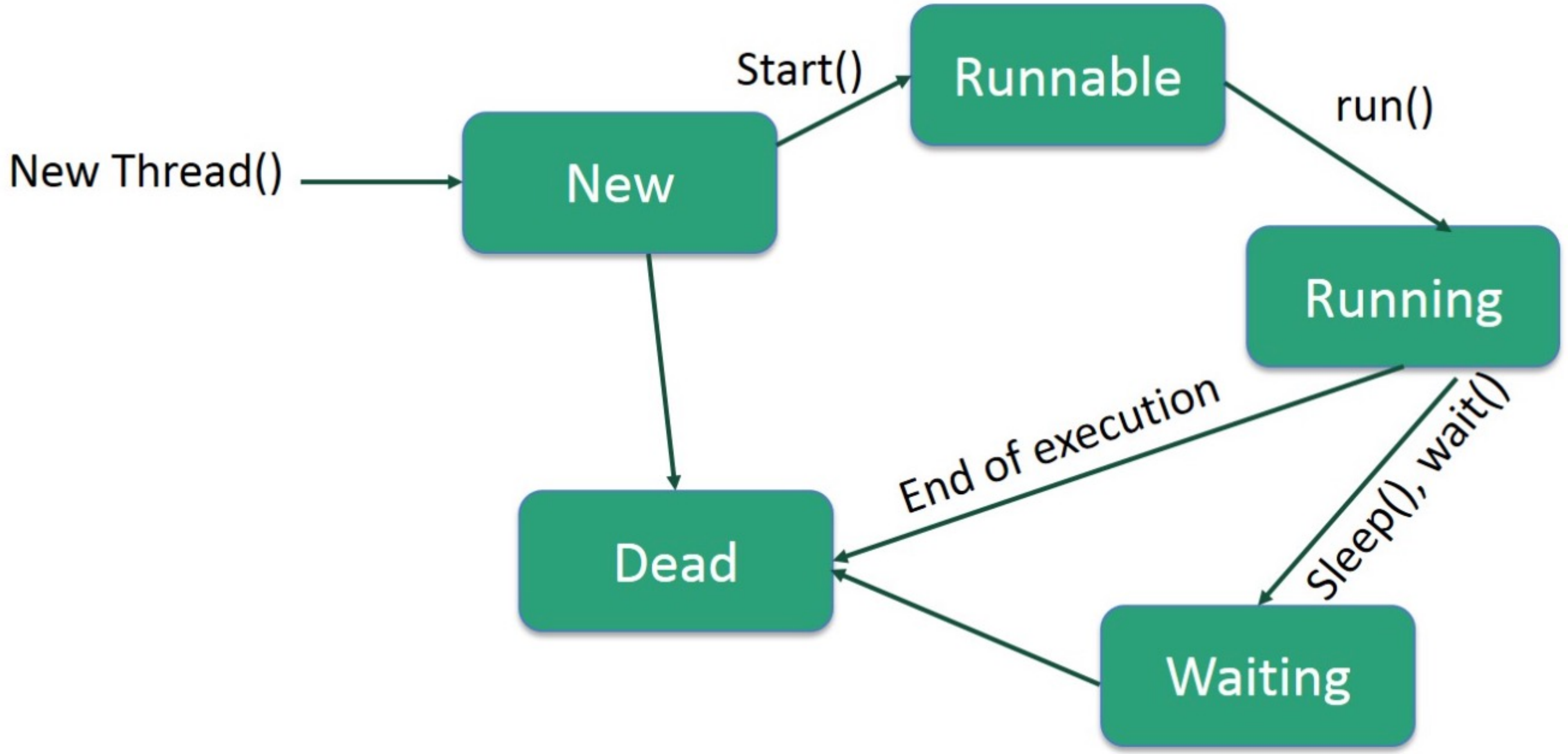
Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

- Extending the Thread class
- Implementing the Runnable Interface

Multithreading in Java



Multithreading in Java

- **Deadlock**: Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.
- Solution :
  - use synchronized
  - use Lock.

Multithreading & Multiprocessing

- Multithreading shares the memory heap within the process; any thread can access or modify the objects in the same memory heap. Therefore, thread is rather lightweight because it contains less resources, and doesn't carry the memory space.
- Multiprocessing spawns new processes instead of thread. Each process generally has a complete, private set of basic run-time resources including its own memory heap; therefore, all objects in the memory have to be copied when spawning new sub-processes, which increases the overhead of multiprocessing.