Topics covered: Version control, git commands and github

PROGRAM-1:

a)Create a git repository and push the it to the GitHub account.

b)) Illustrate the working of following commands

i)clone ii)diff iii)branch, checkout and merge iv)pull

a) Create a java project in intellij IDEA and push the code to the GitHub account.

Step-1: Create new folder named Test on desktop

Step-2: Navigate into the test folder and right click and choose open git bash here option .This will open the git bash

Step-3: Initialize empty repository using

git init command

Step-4: Add all files of current project into staging area using

git add .

Step-5: Commit the changes using

git commit -m "Commit Message"

Step-6: Add the remote repo link to local repo using

git remote add origin "github-url"

Step-7: Push your local changes to remote using

git push -u origin master

b) Illutrate the working of following commands

i)clone ii)diff iii)branch, checkout and merge iv)pull

i)clone is used to to clone a gitHub Repo

git clone "github url"

ii)diff is used to see the changes done to file.

git diff

iii)branch is used to create a new branch to add new feature. Below are the Steps to follow while working with branches in Git

Step 1: Create branch using

git branch <branch_name>

Step 2: Checkout branch

git checkout <branch_name>

Step 3: Make some changes to your project. Add files and commit,

Step 4: On local repo checkout to master branch

git checkout master

Step 5: Merge new branch in master branch

git merge <branch_name>

Step 6: Push all your changes

git push -u origin master

Step 7: Delete a particular branch

git branch -d <branch_name >// Will delete from local repository

git push origin --delete <branch_name>// Will delete from remote repository

Topics covered: Overview of Build Automation Tools, Key Differences Between Maven and Gradle, Install and set up the maven and gradle

Program-2:

a)Overview of Build Automation Tools, Key Differences Between Maven and Gradle
b)Install and set up the maven and gradle in your system .

a)Overview of Build Automation Tools, Key Differences Between Maven and Gradle

Build automation tools help developers streamline the process of building, testing, and deploying software projects. They take care of repetitive tasks like compiling code, managing dependencies, and packaging applications, which makes development more efficient and error-free.

Two popular tools in the Java ecosystem are **Maven** and **Gradle**. Both are great for managing project builds and dependencies, but they have some key differences.

Maven

- **What is Maven?** Maven is a build automation tool primarily used for Java projects. It uses an XML configuration file called pom.xml (Project Object Model) to define project settings, dependencies, and build steps.
- **Main Features:**
  - Predefined project structure and lifecycle phases.
  - Automatic dependency management through Maven Central.
  - Wide range of plugins for things like testing and deployment.
  - Supports complex projects with multiple modules.

Gradle

- **What is Gradle?** Gradle is a more modern and versatile build tool that supports multiple programming languages, including Java, Groovy, and Kotlin. It uses a domain-specific language (DSL) for build scripts, written in Groovy or Kotlin.
- **Main Features:**
  - Faster builds thanks to task caching and incremental builds.
  - Flexible and customizable build scripts.
  - Works with Maven repositories for dependency management.
  - Excellent support for multi-module and cross-language projects.
  - Integrates easily with CI/CD pipelines.

Key Differences Between Maven and Gradle

Aspect	**Maven**	**Gradle**
Configuration	XML (pom.xml)	Groovy or Kotlin DSL
Performance	Slower	Faster due to caching
Flexibility	Less flexible	Highly customizable
Learning Curve	Easier to pick up	Slightly steeper
Script Size	Verbose	More concise
Dependency Management	Uses Maven Central	Compatible with Maven too
Plugin Support	Large ecosystem	Extensible and versatile

b)Install and set up the maven and gradle in your system .

Installing Maven

Step 1: Install Java (JDK 17 Recommended)

Check if Java is installed:

java -version

javac -version

Step 2: Download and Install Maven

- Download Apache Maven – Maven
- Extract it to a folder (e.g., C:\Maven).

Step 3: Configure Environment Variables

- Add MAVEN_HOME → C:\Maven\apache-maven-<version>
- Add M2_HOME → C:\Maven\apache-maven-<version>
- Update Path → C:\Maven\apache-maven-<version>\bin

Step 4: Verify Installation

- Run in command prompt
  mvn --version
- Expected Output:
  Apache Maven 3.x.x
  Maven home: C:\Maven\apache-maven-<version>
  Java version: 17.0.4

b)Installing Gradle

Step 1: Install Java (JDK 17 Recommended)

Check if Java is installed:

java -version

javac -vesrion

Step 2:

- Download gradle from Gradle | Releases
- Extract it to a folder in any drive (e.g., C:\Gradle).

Step 3: Configure Environment Variables

Update Path → C:\Gradle\gradle-<version>\bin

Step 4: Verify Installation

- Run in command prompt
  Gradle -v
- Expected Output:
  Gradle 8.12.1
  Build time: 2025-01-24 12:55:12 UTC
  Revision: 0b1ee1ff81d1f4a26574ff4a362ac9180852b140
  Kotlin: 2.0.21
  Groovy: 3.0.22
  Ant: Apache Ant(TM) version 1.10.15 compiled on August 25 2024
  Launcher JVM: 17.0.9 (Oracle Corporation 17.0.9+11-LTS-201)
  Daemon JVM: C:\Program Files\Java\jdk-17
  (no JDK specified, using current Java home)
  OS: Windows 11 10.0 amd64

Topics covered: Creating a Maven Project, Understanding the POM File, Dependency Management

Program-3:

Create a Java program to create a class called Person with properties name and age. Display the person details in json format using Gson libraray. Show how to configure the pom file in the Maven project to automatically download the Gson dependencies .

Step 1: Create a New Maven Project:

- Open IntelliJ IDEA.
- Go to File > New > Project .
- Select Maven from the build types.
- Set the project name , then click Finish .

Step 3: Open the Main.java file under the directory PROJECT_NAME\src\main\java\com.example and write the following code

/* Main.java */

```
package com.example;
import com.google.gson.Gson;
public class Main
{
    public static void main(String args[])
    {
        Gson gson= new Gson();
        String json= gson.toJson(new Person("john",30) );
        System.out.println(json);
    }
}
class Person
{
    private String name;
    private int age;
    public Person(String name, int age)
    {
        this.name=name;
        this.age=age;
    }
}
```

Step-2: Set Up the pom.xml File:

The pom.xml file is where you define dependencies, plugins, and other configurations for your Maven project.

Example of a basic 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      ht
tp://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.example</groupId>
     <artifactId>simple-project</artifactId>
     <version>1.0-SNAPSHOT</version>
     <dependencies>
        <!-- Add your dependencies here -->
     </dependencies>
 </project>
```

Step-3: Add Dependencies for Gson:

Add Gson dependency to the section of the pom.xml file. To accomplish this, go to www.mvnrepositorry.com and look up "Gson." The code should then be copied and pasted into the dependencies section.

```
  <dependencies>
    <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
     <dependency>
         <groupId>com.google.code.gson</groupId>
         <artifactId>gson</artifactId>
         <version>2.8.9</version>
     </dependency>
</dependencies>
```

Step-4: Run the Main.main to see the output as follows

{"name":john, "age":30}

Topics covered: Plugin management in maven

Program-4:

a) Illustrate the use of maven-jar-plugin to package the project into a jar file. Showing how to run a main class and show simple output.

b) Develop a basic webpage and demonstrate how to use the relevant plug-in to deploy it on GitHub pages.

a) Illustrate the use of maven-jar-plugin to package the project into a jar file. Showing how to run a main class and show simple output.

Steps to Package the Project as a JAR and Run a Main Class

Step 1: Add maven-jar-plugin to pom.xml

To package your Maven project as a JAR file and specify the main class, we need to configure the **maven-jar-plugin** in the pom.xml . Add the following configuration to your pom.xml :

```
 <build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <!-- Specify the main class to be executed -->
                <archive>
                    <manifestEntries>
                        <Main-Class>com.example.Main</Main-Class>
                    </manifestEntries>
                </archive>
            </configuration>
        </plugin>
    </build>
</plugins>
```

This will tell Maven to include the **Main-Class** in the JAR manifest and specify the main class that should be executed when the JAR is run.

Step-2: The following default code is already available in Main.java in the directory PROJECT_NAME\src\main\java\com\example\Main.java

```
package com.example;
public class Main
{
    public static void main(String[] args)
    {
        System.out.printf("Hello and welcome!");
```

```
        for (int i = 1; i <= 5; i++)
        {
            System.out.println("i = " + i);
        }
    }
}
```

This class contains a simple main method that prints output when run.

Step 3: Package the Project into a JAR:

Run the following Maven command to build the project and package it into a JAR file:

```
mvn clean package
```

This will clean any previous builds, compile the source code, and package it into a JAR file located in the target directory (e.g., target/your-project-name.jar).

Step 4: Run the JAR File:

Once the JAR is created, you can run it with the following command:

```
java -jar target/your-project-name.jar
```

This will execute the main method from your Main.java and print the message:

```
Hello and welcome
i=1
i=2
i=3
i=4
i=5
```

Or

Copy the jar file and paste it on desktop. Open the command prompt from start menu and execute the following command to see the output of Main.java

```
cd desktop
java -jar target/your-project-name.jar
```

b) Develop a basic webpage and demonstrate how to use the relevant plug-in to deploy it on GitHub pages.

- **Create a Simple Website**

In the src/main/resources folder, create an index.html file

/* index.html */

```
<html>
    <head>
        <title>My Simple Website</title>
    </head>
    <body>
        <h1>Welcome to My Simple Website</h1>
    </body>
 </html>
```

Step 2: Deployment

To deploy your Maven project to **GitHub Pages** using the /docs folder (** **having all files inside root folder/dir not recommended**), you can follow these simple steps. This method is easy and doesn't require switching branches—just use the /docs folder of your main branch.

- **Modify Maven Configuration to Copy Static Files to** /docs **Folder**: First, you need to ensure that Maven places your static files (index.html , style.css , logo.png) into the /docs folder instead of the target directory.To do this, configure the Maven Resources Plugin in your pom.xml to copy the files directly into /docs :

```
<build>
 <plugins>
   <plugin>
     <artifactId>maven-resources-plugin</artifactId>
     <version>3.3.1</version>
     <executions>
       <execution>
         <id>copy-resources</id>
         <!-- here the phase you need -->
         <phase>validate</phase>
         <goals>
           <goal>copy-resources</goal>
         </goals>
         <configuration>
           <outputDirectory>${project.basedir}/docs</outputDirectory>
           <resources>
             <resource>
               <directory>src/main/resources</directory>
               <filtering>true</filtering>
             </resource>
           </resources>
         </configuration>
       </execution>
     </executions>
   </plugin>
 </plugins>
</build>
```

In this configuration:

The maven-resources-plugin copies all files from src/main/resources to the /docs folder in the root of your project(not the target directory).This is done during the prepare-package phase, just before Maven prepares the package.

- **Build the Project**: Run the following Maven command to build your project and copy the resources to the /docs folder:

```
mvn clean install
```

After this, your index.html , should now be inside the docs folder in the root of your project.

- **Upload the Website to GitHub**: Now that the files are in the /docs folder, they are ready to be served by GitHub Pages.

  Follow these steps:

  - Initialize a Git repository in your project folder:
    - `git init`
  - Add your files and commit them:
    - `git add .`
    - `git commit -m "Initial commit"`

- Create a GitHub repository and push the local project to GitHub:

```
git remote add origin <your-repository-url>
git push -u origin master
```

- **Enable GitHub Pages**: After pushing to the main branch, follow these steps to enable GitHub Pages:
  - Go to your GitHub repository.
  - Navigate to **Settings** > **Pages** (on the left sidebar).
  - Under the **Source** section, select the main branch and /docs folder as the source.
  - Click **Save**.
- **Access Your Website**: Your static website is now hosted on GitHub Pages! You can access it at

```
https://<your-github-username>.github.io/<your-repository-name>/
```

Topics covered: Configuration of pom.xml of maven project with selenium and TestNG for automation testing , **Maven** site **&** deploy **Commands for Documentation**

Program-5:

a)Write the java class for **Testing** the title of your website using **Selenium**, **Java**, and **TestNG**

b) Illustrate the maven site and deploy commands for the project created from previous programs

a)Write the java class **Testing** the title of your website using **Selenium**, **Java**, and **TestNG**

To test the title of your website using **Selenium**, **Java**, and **TestNG**, follow all the steps of program 4b to publish ur website on Github pages. Once your website is ready, now follow the given steps to test the title of your website.

Step 1: Set Up Selenium and TestNG Dependencies

In your Maven project created in program 4b, add the necessary dependencies for **Selenium WebDriver** and **TestNG** to the pom.xml file:

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
  <dependency>
     <groupId>org.seleniumhq.selenium</groupId>
     <artifactId>selenium-java</artifactId>
     <version>4.28.1</version>
  </dependency>
     <!-- https://mvnrepository.com/artifact/org.testng/testng -->
     <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.11.0</version>
        <scope>test</scope>
     </dependency>
 </dependencies>
```

- **Selenium WebDriver**: This is used for browser automation.
- **TestNG**: This is a testing framework used to run Selenium tests.

Step 2: Create Selenium Test Class Using TestNG

Next, create a test class in your src/test/java directory. You can name WebsiteTitleTest.java.

/* WebsPageTest.java */

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;
import static org.testng.Assert.assertTrue;
public class WebPageTest
{
    private static WebDriver driver;
```

```
    @BeforeTest
    public void openBrowser() throws InterruptedException
    {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        Thread.sleep(2000);
        driver.get("https://sarvarbegum-coder.github.io/LAB_1/");
    }
    @Test
    public void titleValidationTest()
    {
        String actualTitle = driver.getTitle();
        String expectedTitle = "My simple website";
        Assert.assertEquals(actualTitle, expectedTitle);
        assertTrue(true, "Title should contain 'simple'");
    }
    @AfterTest
    public void closeBrowser() throws InterruptedException
    {
        Thread.sleep(1000);
        driver.quit();
    }
}
```

Step 3: Run the Test Using TestNG

Option 1: Run TestNG from IntelliJ IDEA

1. Right-click the WebpageTest.java file.
2. Select **Run WebpageTest.**

IntelliJ IDEA will execute the TestNG test and show the results in the output console.

Option 2: Run TestNG via Command Line

If you want to run the tests from the command line, use the following Maven command:

```
mvn test
```

b) Illustrate the maven site and deploy commands.

mvn site command :

The mvn site command is used to **generate a project website** containing reports like dependencies, build details, test results, and more

Step 1: Create a New Maven Project:

- Open IntelliJ IDEA.
- Go to File > New > Project .
- Select Maven from the project types.
- Set the project name and location, then click Finish .

Step 2: Add Site Plugin in pom.xml

Before running the site command, you need to add the **Maven Site Plugin** inside the <build> section of your pom.xml :

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-site-plugin</artifactId>
            <version>3.12.1</version> <!-- Use latest version -->
        </plugin>
    </plugins>
</build>
```

Step 3: Run the Site Command

Once the plugin is added, execute:

```
mvn site
```

- Maven scans your project for available reports.
- Generates an **HTML-based website** inside target/site/ .
- Includes various reports like dependencies, plugin management, and test results.

Step 4: Open the Generated Site

After successful execution, open the following file in a browser:

```
D:\Idea Projects\CA-MVN\target\site\index.html
```

mvn deploy command:

The mvn deploy command is used to **upload the built artifact (JAR, POM, etc.)** to a repository for distribution and sharing.

Create a Local Repository: Let us create the local repository for the project created for site command.

Step 1: Open terminal and create a new directory using the following command

```
mkdir D:\my-local-maven-repo
```

Step 2: Configure pom.xml for Local Deployment

Add the following inside <project> in pom.xml :

```
<distributionManagement>
    <repository>
        <id>local-repo</id>
        <url>file:///D:/my-local-maven-repo</url>
    </repository>
</distributionManagement>
```

Step 3: Run the Deploy Command

```
mvn deploy
```

Maven **builds** the project and stores the artifact (JAR, POM, etc.) in D:/my-local-maven-repo.

Step 4: Verify Deployment

- Navigate to D:/my-local-maven-repo/ and check if the project is stored correctly

D:/my-local-maven-repo/
├— org/example/CA-MVN/1.0-SNAPSHOT/
│ ├— CA-MVN-1.0-SNAPSHOT.jar
│ ├— CA-MVN-1.0-SNAPSHOT.pom

Topics covered: Working with Gradle: Setting Up a Gradle Project, Understanding Build Scripts (Groovy and Kotlin DSL), Dependency Management

Program-6

Create a Java program to create a class called Person with properties name and age. Display the person details in json format using Gson libraray. Show how to configure the build.gradle in the gradle project to automatically download the Gson dependencies .

Step-1: Open IntelliJ IDEA and Create a New Project

- Click on **"New Project"**.
- Select **"Gradle"** (under Java/Kotlin).
- Choose **Groovy (Domain Specific Language)** for the build script.
- Set the **Group ID** (e.g., com.example).
- Click **Finish**.
- Once u click on finish, u will see the following gradle project structure

my-gradle-project

```
| ——— build.gradle (Groovy Build Script)
| ——— settings.gradle
| ——— src
|   ├— main
|   |   ├— java
|   |   └——— resources
|   ├— test
|   |   ├— java
|   |   └——— resources
```

Step-2: Open the Main.java file under the directory PROJECT_NAME\src\main\java\com.example and write the following code

/* Main.java */

```
package com.example;
import com.google.gson.Gson;
public class Main
{
    public static void main(String args[])
    {
        Gson gson= new Gson();
        String json= gson.toJson(new Person("john",30) );
        System.out.println(json);
    }
}
class Person
{
    private String name;
    private int age;
```

```
    public Person(String name, int age)
    {
        this.name=name;
        this.age=age;
    }
}
```

Step-2: Set Up the build.gradle File:
The build.gradle file is where you define dependencies, plugins, and other configurations for your gradle project.
Example of a basic build.gradle:

```
plugins {
    id 'java'
}

group = 'com.example'
version = '1.0-SNAPSHOT'

repositories {
    mavenCentral()
}

dependencies {
    testImplementation platform('org.junit:junit-bom:5.10.0')
    testImplementation 'org.junit.jupiter:junit-jupiter'
    }

test {
    useJUnitPlatform()
}
```

Step-3: Add Dependencies for Gson:
Add Gson dependency to the section of the pom.xml file. To accomplish this, go to www.mvnrepositorry.com and look up "Gson." Select gradle code, then code should be copied and pasted into the dependencies section.

```
dependencies {
    testImplementation platform('org.junit:junit-bom:5.10.0')
    testImplementation 'org.junit.jupiter:junit-jupiter'
```

```
    // https://mvnrepository.com/artifact/com.google.code.gson/gson
    implementation 'com.google.code.gson:gson:2.8.9'
    }
```

Step-4: Run the Main.main to see the output as follows

{"name":john, "age":30}

Topics covered: Migrate the maven Application to Gradle

Program-7:

Build and Run a Java Application with Maven, Migrate the Same Application to Gradle

Step 1: Create a New Maven Project:

- Open IntelliJ IDEA.
- Go to File > New > Project .
- Select Maven from the project types.
- Set the project name and location, then click Finish .

Step-2: Update pom.xml to Add jar Plugin

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>3.2.0</version>
            <configuration>
                <!-- Specify the main class to be executed -->
                <archive>
                    <manifestEntries>
                        <Main-Class>com.example.Main</Main-Class>
                    </manifestEntries>
                </archive>
            </configuration>
        </plugin>
    </build>
</plugins>
```

Step-3: Build and Run the Maven Project

- Open the terminal and Run the following commands to build the project
  `mvn clean package`
- Locate the JAR File
  `D:\Idea Projects\MVNGRDLDEMO\target\MVNGRDLDEMO-1.0-SNAPSHOT.jar`
- Execute the jar file either by right click and select run `MVNGRDLDEMO-1.0-SNAPSHOT.jar or` by executing the following command
  `java -jar target\MVNGRDLDEMO-1.0-SNAPSHOT.jar`

Step-4: Migrate Maven Project to Gradle

- Execute the following command to migrate your Maven project to Gradle. This command will convert your Maven pom.xml into a Gradle build.gradle file.
  `gradle init --type pom`
- `Review and Update build.gradle. Ensure the following configurations are correct`

```
plugins
{
    id 'java'
}

group = 'com.example'
```

```
version = '1.0-SNAPSHOT'
repositories
{
    mavenCentral()
}

dependencies
 {
    testImplementation 'junit:junit:4.13.2'
 }
jar
{
    manifest
    {
        attributes
        (
        'Main-Class': 'com.example.Main'
        )
    }
}
```

Build and Run the gradle Project

- Open the terminal and Run the following commands to build the project
  gradle `clean build`
- Locate the JAR File
  `D:\Idea Projects\MVNGRDLDEMO\build\libs\MVNGRDLDEMO-1.0-SNAPSHOT.jar`
- Execute the jar file either by right click and select run `MVNGRDLDEMO-1.0-SNAPSHOT.jar or` by executing the following command
  `java -jar MVNGRDLDEMO-1.0-SNAPSHOT.jar`

Topics covered: Introduction to Jenkins: What is Jenkins?, Installing Jenkins on Local or Cloud Environment, Configuring Jenkins for First Use

Program-8:
Install Jenkins on Local or Cloud Environment, Configuring Jenkins for First Use

CI/CD is a method to frequently deliver apps to customers by introducing automation into the stages of app development. The main concepts attributed to CI/CD are continuous integration, continuous delivery, and continuous deployment.

Installing Jenkins Using WAR File:

Step 1: Download the Jenkins WAR File
Download from: https://www.jenkins.io/download/
Choose Generic Java Package (.war).
Step 2: Run Jenkins Using Java
Navigate to the folder where the .war file is downloaded and run:

```
java -jar jenkins.war --httpPort=8181
```

This will start Jenkins on port **8181**.
Step 3: Open Jenkins in Browser
Go to:
http://localhost:8181

Step 4: Unlock Jenkins & Setup
Follow the **same steps** as the Windows/Linux installation:
✓ Find the initial password
✓ Install suggested plugins
✓ Create an admin user
Jenkins is now running without installation!
To stop Jenkins, press **CTRL + C** in the terminal.

Configuring Jenkins for First Use:

Understanding the Jenkins Dashboard:
After logging in, you will see:

- **New Item** → Create Jobs/Pipelines
- **Manage Jenkins** → Configure System, Users, and Plugins
- **Build History** → View previous builds
- **Credentials** → Store secure authentication details

Installing Additional Plugins:
Jenkins supports **plugins** for various tools like Maven, Gradle, Docker, and Azure DevOps.
To install a plugin:

- Go to **Manage Jenkins** → **Manage Plugins**
- Search for the required plugin

- Click **Install without Restart**

Setting Up Global Tool Configuration:
Configure Java, Maven, and Gradle in Jenkins:

- Go to **Manage Jenkins → Global Tool Configuration**
- Add paths for:
  Git(C:\Program Files\Git\bin\git.exe)
  JDK (C:\Program Files\Java\jdk-17)
  Maven (C:\apache-maven-<version>)
  Gradle (C:\gradle-<version>)
- Click **Save**

Topics covered: Continuous Integration with Jenkins: Setting Up a CI Pipeline, Integrating Jenkins with Maven/Gradle, Running Automated Builds and Tests

Program-9:

Create a maven project to test the title of the webpage and publish the webpage in GitHub pages. Illustrate how to run the automated builds and tests from Jenkins.

To test the title of your website using **Selenium**, **Java**, and **TestNG**, follow all the steps of program 4b to publish ur website on Github pages. Once your website is ready, now follow the given steps to test the title of your website.

Step 1: Set Up Selenium and TestNG Dependencies

In your Maven project created in program 4b, add the necessary dependencies for **Selenium WebDriver** and **TestNG** to the pom.xml file:

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
  <dependency>
     <groupId>org.seleniumhq.selenium</groupId>
     <artifactId>selenium-java</artifactId>
     <version>4.28.1</version>
  </dependency>
     <!-- https://mvnrepository.com/artifact/org.testng/testng -->
     <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.11.0</version>
        <scope>test</scope>
     </dependency>
 </dependencies>
```

- **Selenium WebDriver**: This is used for browser automation.
- **TestNG**: This is a testing framework used to run Selenium tests.

Step 2: Create Selenium Test Class Using TestNG

Next, create a test class in your src/test/java directory. You can name WebsiteTitleTest.java.

/* WebsPageTest.java */

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;
import static org.testng.Assert.assertTrue;
public class WebPageTest
{
    private static WebDriver driver;
    @BeforeTest
    public void openBrowser() throws InterruptedException
    {
        driver = new ChromeDriver();
```

```
        driver.manage().window().maximize();
        Thread.sleep(2000);
        driver.get("https://sarvarbegum-coder.github.io/LAB_1/");
    }
    @Test
    public void titleValidationTest()
    {
        String actualTitle = driver.getTitle();
        String expectedTitle = "My simple website";
        Assert.assertEquals(actualTitle, expectedTitle);
        assertTrue(true, "Title should contain 'simple'");
    }
    @AfterTest
    public void closeBrowser() throws InterruptedException
    {
        Thread.sleep(1000);
        driver.quit();
    }
}
```

Step 3: Add your files, commit and push the changes to gitHub:

- git add .
- git commit -m "Initial commit"
- git push -u origin master

Step 4: Running a Selenium Java Test from a Local Maven Project

- Create a New Jenkins Job
  - Go to Jenkins Dashboard → Click New Item.
  - Enter a project name → Select Freestyle Project.
  - Click OK.
- Configure the Build Step
  - Scroll to Build → Click Add build step → Execute Windows Batch Command.
  - Enter the following commands (ensure correct navigation to project directory):

```
cd <Absolute path of project>
mvn test
```

  - Click Save → Click Build Now to execute the test

Step 5: Running Selenium Tests from a GitHub Repository via Jenkins

- Set Up a New Jenkins Job for GitHub Project
  - Go to Jenkins Dashboard → Click New Item.
  - Enter a project name → Select Freestyle Project.
  - Click OK.
- Configure Git Repository in Jenkins

- Under Source Code Management, select Git.
- Enter your GitHub repository URL (e.g., https://github.com/your-repo-name.git).

- Add Build Step for Maven
  - Scroll to Build → Click Add build step → Invoke top-level Maven targets
  - Select maven version as MAVEN
  - Add Goals as test
  - Click **Save**.
- **Trigger the Build**
  - Click Build Now to fetch the code from GitHub and execute the Selenium tests.
  - Check the **Console Output** to verify test execution

Step 6: Running Selenium Tests whenever we push the new code into github repository

- Set Up a New Jenkins Job for GitHub Project
  - Go to Jenkins Dashboard → Click New Item.
  - Enter a project name → Select Freestyle Project.
  - Click OK.
- Configure Git Repository in Jenkins
  - Under Source Code Management, select Git.
  - Enter your GitHub repository URL (e.g., https://github.com/your-repo-name.git).
- Configure triggers in Jenkins
  - Under triggers section, select poll SCM.
  - Schedule the trigger every minute by writing *** * * * *** in schedule. This will make Jenkins to check for change in source code every minute.If there is a change, then trigger the build
- Add Build Step for Maven
  - Scroll to Build → Click Add build step → Invoke top-level Maven targets
  - Select maven version as MAVEN
  - Add Goals as test
  - Click **Save**.
- **Trigger the Build**
  - Go to Intellij Idea , do some changes in index.html.
  - Add the files using git add .
  - Commit the changes using git commit -m “message”
  - Push the modified code to git hub using git push -u origin master
  - Go to Jenkins, and check to see that the build is automatically started when we pushed the changes to git hub