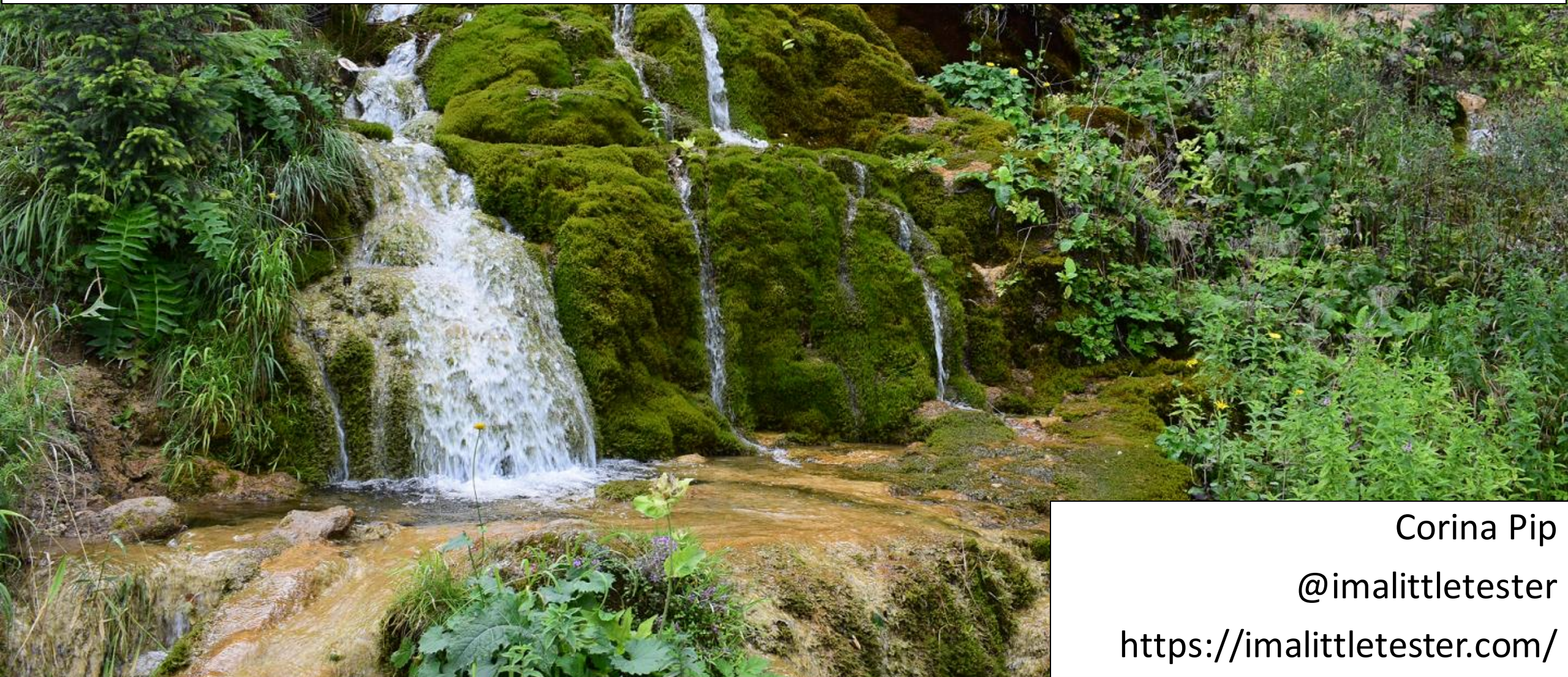
Designing Your Java and Maven-based Testing Framework

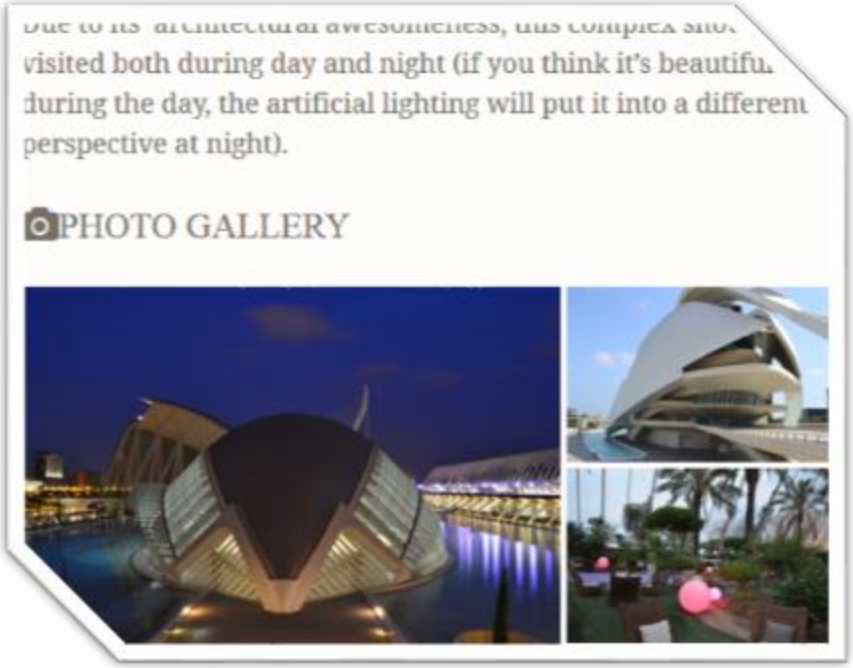


Corina Pip

@imalittletester

https://imalittletester.com/

- Twitter: **@imalittletester**
- Blog: https://imalittletester.com/
- Comics: https://imalittletester.com/category/comics/
- GitHub: https://github.com/iamalittletester



@imalittletester

AGENDA

1. CREATING THE FRAMEWORK
2. IMPORTING THE DEPENDENCIES
3. PROJECT STRUCTURE
4. SETTING UP THE BROWSER
5. TEST ENVIRONMENT SETUP
6. TESTING TRANSLATIONS
7. TEST DESIGN

What we will use

- Java
- Maven – for project setup, running groups of tests
- Selenium – to test browser interactions
- JUnit – to write the test and run them
- IntelliJ – to write the code

1. CREATING THE FRAMEWORK

Create framework (from IntelliJ)

1. Install IntelliJ (you can install the Community edition: https://www.jetbrains.com/idea/download).
2. Start IntelliJ.
3. Choose 'Create New Project'.
4. From the list of options you are presented: choose Maven.
5. Check the 'Create from archetype' checkbox.

Create framework (from IntelliJ)

6. Choose 'org.apache.maven.archetypes:maven-archetype-quickstart'. Click Next.

7. Type a new groupId: it must be your company url backwards. E.g. com.imalittletester.

8. Type a new artifactId: it will be the name of your project. All lowercase. - is allowed.

9. Leave version to 1.0-SNAPSHOT.

10. In the 'Maven home directory' screen leave all options as they are and click Next.

11. On the confirmation screen just click Finish.

Build project

- After the project is loaded in IntelliJ – click 'Enable Auto-Import' from the lower right side popup titled 'Maven projects need to be imported'.
- On the top-right hand side of the screen (or below left) click the ‘Maven Projects’ icon.
- Expand your project name. Expand Lifecycle.

Build project

- Hold your mouse and select both ‘clean’ and ‘install’ from the menu.
- Click the small green arrow at the top of the ‘Maven projects’ panel.
- In the lower part of IntelliJ a ‘Run’ tab opens. You will see a ‘BUILD SUCCESS’ message inside this tab.

2. IMPORTING THE DEPENDENCIES

Add dependencies – in pom.xml

- In <dependency> section define the
  - groupId
  - artifactId
  - version

Add dependencies – in pom.xml

- Find them: go to project site – find info there
- If not specified - go to the Github project, open pom.xml file and find the information in the top section of the pom file
- Once known, find the reference needed for your pom.xml <dependency> section in the Maven repository: https://mvnrepository.com/ and the dependency to your project

Why dependencies

- You need a library for a specific purpose: assertions, web testing, backend testing
- You need helper code that already exists: html client, json parsing, db connections
  - Don't rewrite something you can use directly – save time and write only the code you need to write

Add dependencies – in pom.xml

- Example: importing Selenium

```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
  </dependency>
</dependencies>
```

Add dependencies – Exercise

TODO: add dependencies for:

JUnit 5, latest

junit-jupiter-engine

junit-jupiter-params

Apache Commons lang, latest

+ remove classes App and AppTest from the project

Add dependencies – SOLUTION

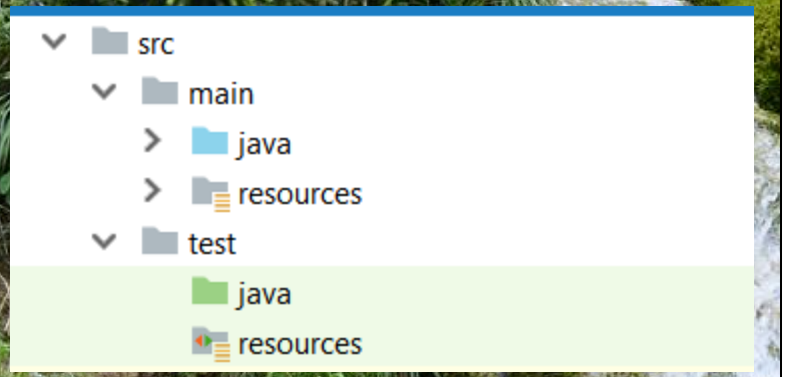
```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6.0-M1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.9</version>
</dependency>
```

Add dependencies – SOLUTION

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6.0-M1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <version>5.6.0-M1</version>
    <scope>test</scope>
</dependency>
```

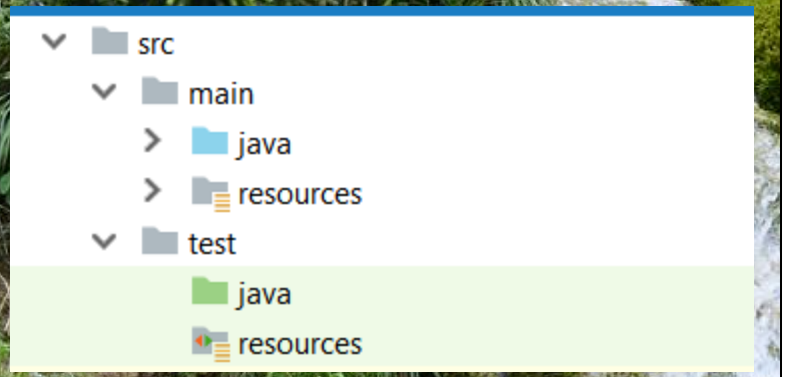
3. PROJECT STRUCTURE

Project structure

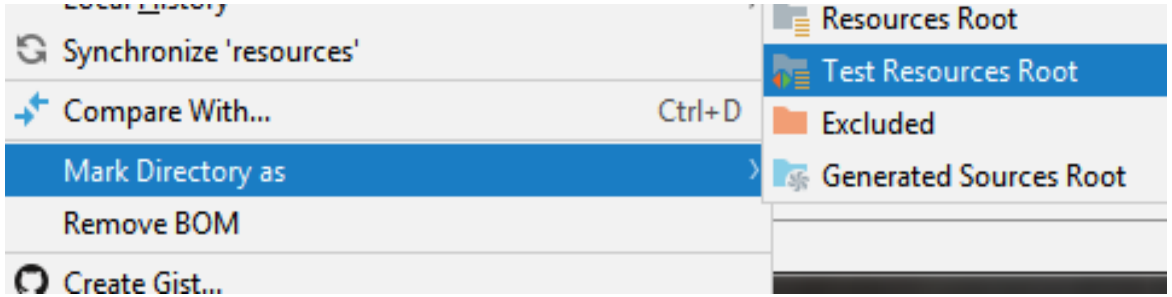


- main
  - java: for creating code needed across several tests. These are helper classes with code that is useful in tests, but is not actual tests
  - resources: folder with files (not .java, but instead: .xml, .txt, .properties, etc) used either for the project configuration or by the helper code (from main/java)
- test
  - java: this is where the tests will be created
  - resources: used to store files dedicated to tests, like: browser drivers, files containing test data
- The pom.xml file is at the root level of the project (not in any project folders)

Project structure



- If src/test/resources folder does not exist, create it
- Then, mark this folder as the test resources folder, by right clicking on it, choosing 'Mark Directory as', then select 'Test Resources Root':

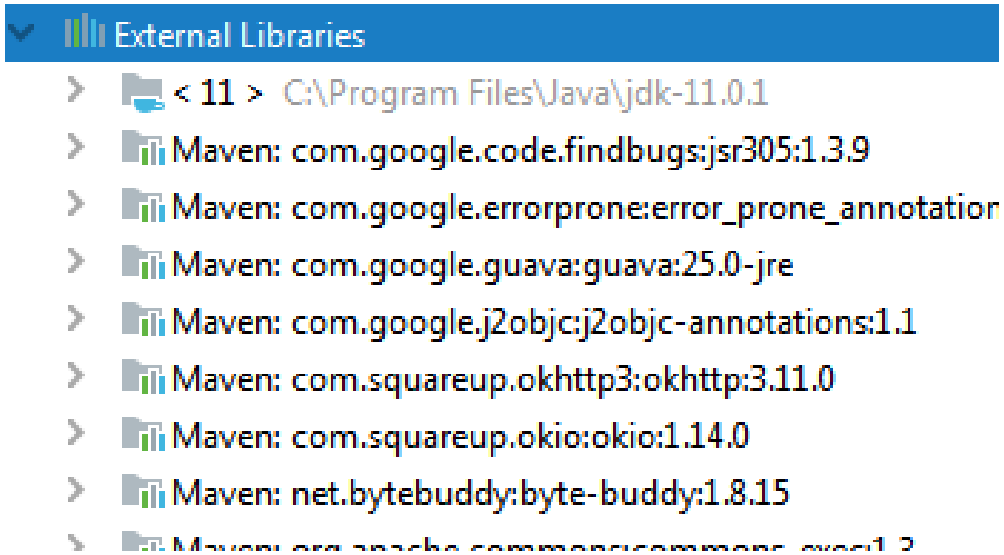


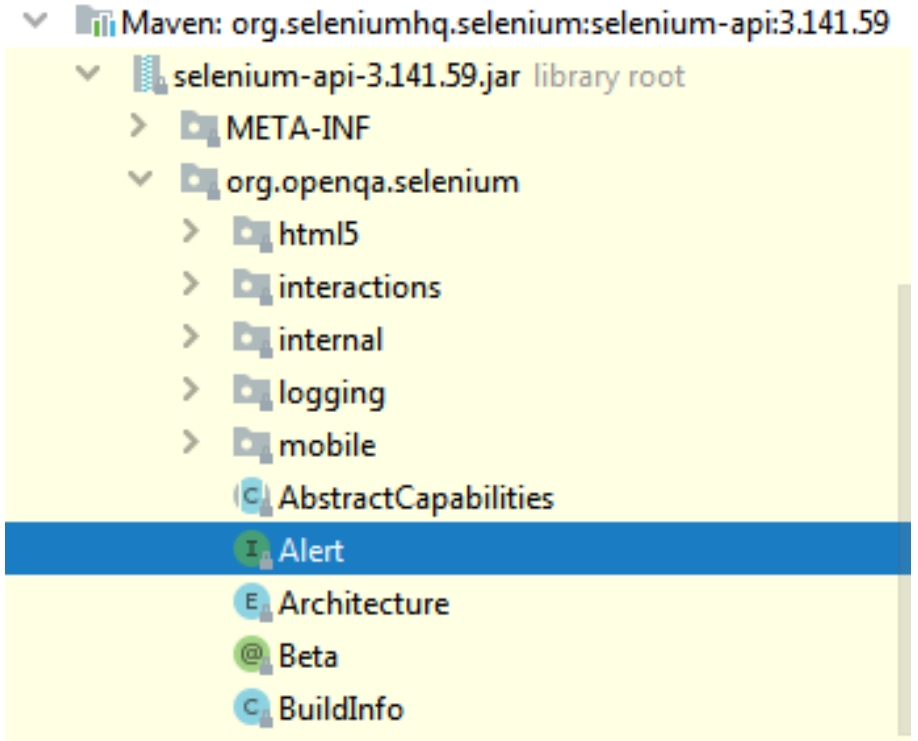
Project Structure example

- src\main\java\utils folder
    - BrowserUtil class - browser related code
    - BaseClass
        - to be extended by all tests
        - contains common code
        - contains class declarations/initializations
    - EnvironmentUtil class – code for selecting environment properties
    - DBUtil? APIUtil? StringProcessingUtil?
- src\test\resources:
    - browserBinaries folder: driver files for testing browsers
    - environments folder: environment specific files

What's inside the dependencies?

- Project view: External libraries section
- First time class opens: Download sources





4. SETTING UP THE BROWSER

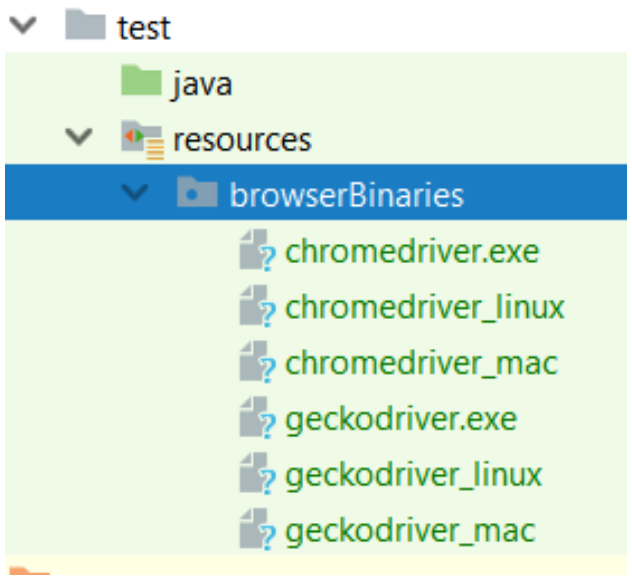
Downloading the driver binaries

- Firefox: https://github.com/mozilla/geckodriver/releases
- Chrome: https://chromedriver.storage.googleapis.com/index.html

- For Windows, Mac, Linux

- To: src\test\resources\browserBinaries

- In src\main\java\utils folder →
  create BrowserUtil class



Create start method for Chrome

```
public WebDriver startChrome() {
  if (SystemUtils.IS_OS_WINDOWS) { setProperty("webdriver.chrome.driver",
                    "src/test/resources/browserBinaries/chromedriver.exe");}
  if (SystemUtils.IS_OS_LINUX) { setProperty("webdriver.chrome.driver",
                    "src/test/resources/browserBinaries/chromedriver_linux");}
  if (SystemUtils.IS_OS_MAC) { setProperty("webdriver.chrome.driver",
                    "src/test/resources/browserBinaries/chromedriver_mac");}
  WebDriver driver = new ChromeDriver();
  driver.manage().window().maximize();
  System.out.println("\n--------------- CHROME has started! ------------------\n");
  return driver; }
```

Create start method for Firefox

```
if (SystemUtils.IS_OS_WINDOWS){  setProperty("webdriver.gecko.driver",
        “src/test/resources/browserBinaries/geckodriver.exe"); }

if (SystemUtils.IS_OS_LINUX){   setProperty("webdriver.gecko.driver",
        "src/test/resources/browserBinaries/geckodriver_linux");  }

if (SystemUtils.IS_OS_MAC){  setProperty("webdriver.gecko.driver",
        "src/test/resources/browserBinaries/geckodriver_mac"); }

FirefoxOptions capabilities = new FirefoxOptions();

capabilities.setCapability("marionette", true);

WebDriver driver = new FirefoxDriver();

 driver.manage().window().maximize();

System.out.println("\n---------------- FIREFOX has started! -------------------\n");

return driver;
```

Create method for switching browser

```
public WebDriver startBrowser() {
  switch (System.getProperty("browser").toLowerCase()) {
      case "chrome":
          return startChrome();
      case "firefox":
          return startFirefox();
      case "ie":
          return startInternetExplorer();
      default:
          System.out.println("THERE WAS AN ERROR READING THE BROWSER
CONFIGURATION! CHROME WILL START BY DEFAULT!");
          return startChrome();  }}
```

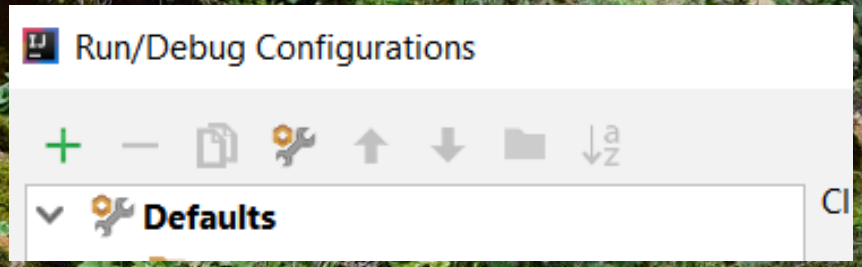
Exercise

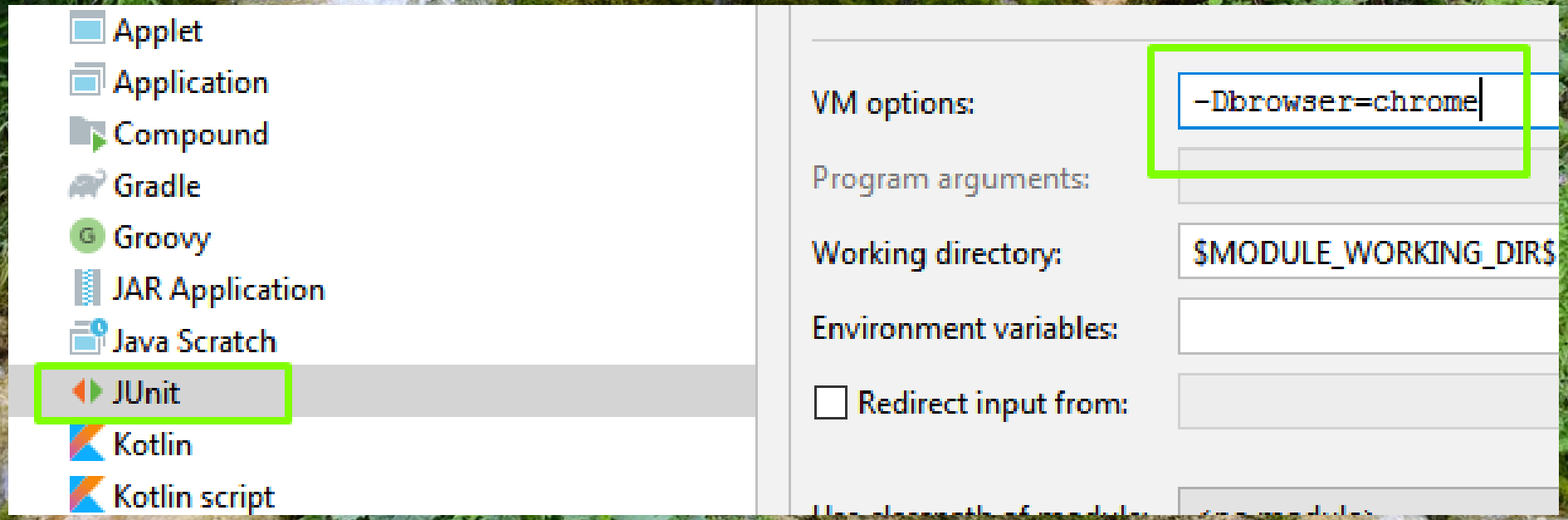
- Create a new package in src/test/java: "firstTests"
- Create a new test class in "firstTests": FirstTest
- Create a new test that opens a page. Hardcode the browser to “Chrome”.
- Create another test that opens a page. Use the “startBrowser” method to initialize the browser.

Exercise - solution

```
private WebDriver driver;
private BrowserUtil browserUtil = new BrowserUtil();

@AfterEach
void afterEach() {
    driver.quit();}

@Test
void startChromeDriver() {
    driver = browserUtil.startChrome();}

@Test
void startDriver() {
    driver = browserUtil.startBrowser();}
```

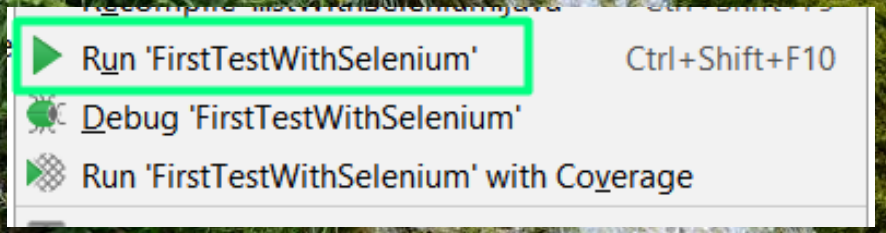
Running tests – from the IDE



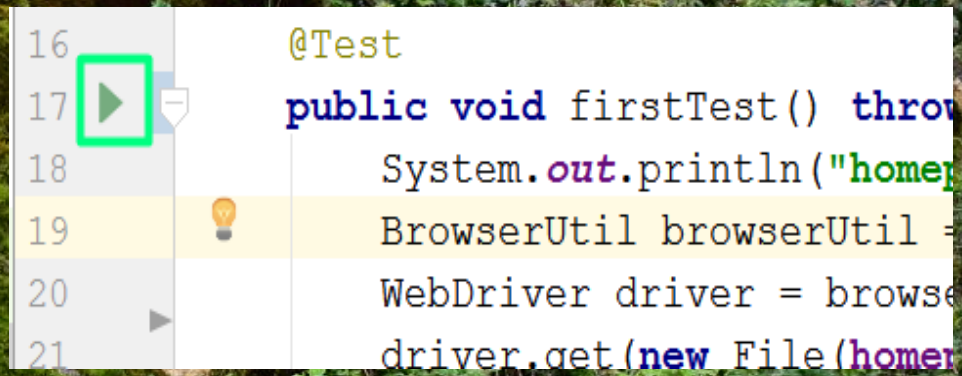


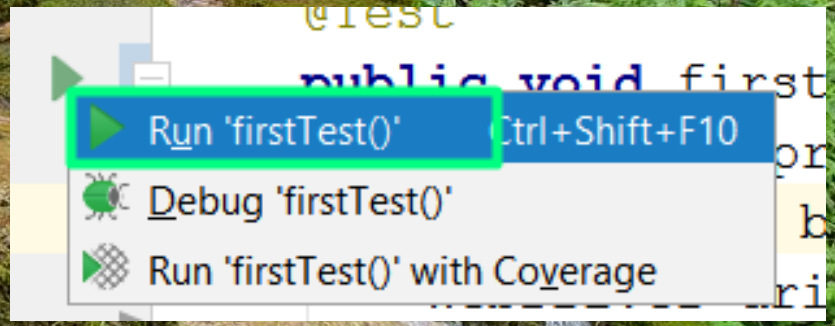
Running tests – from the IDE

On class, from left hand menu – right click and:



OR





Exercise

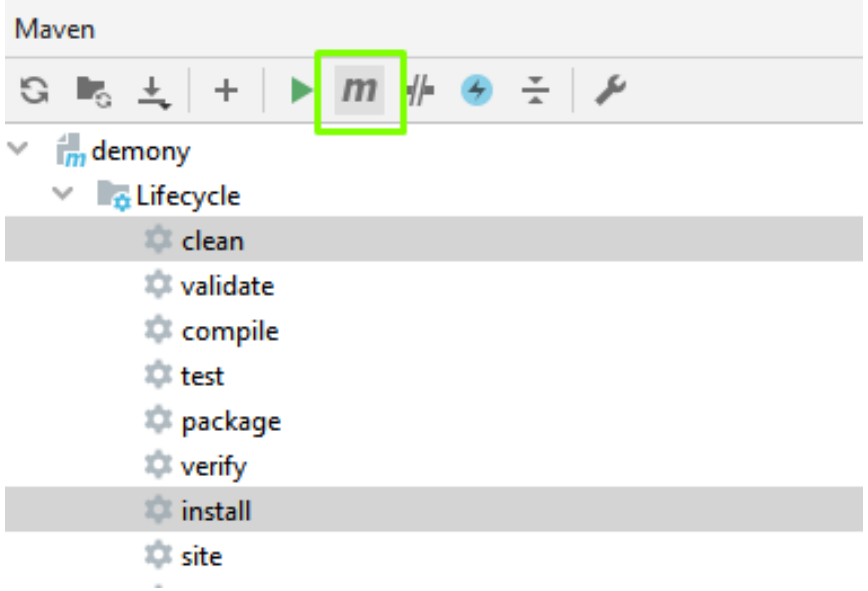
- Run the tests with Chrome
- Run the tests with Firefox
- Run the tests with “mobile”

Run one test class from command line

In command line,

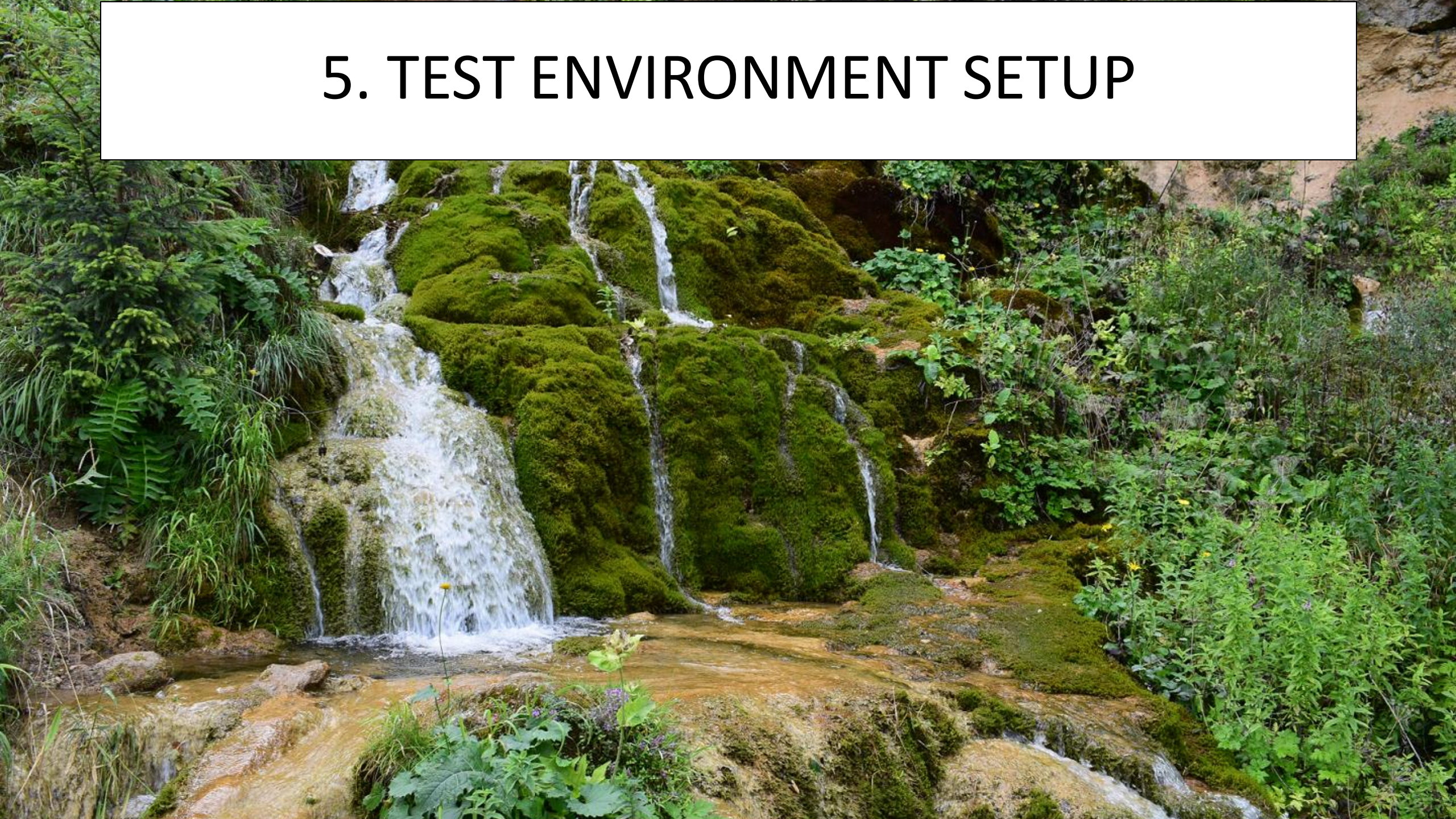
or Terminal from IntelliJ,

or Maven Goal:



mvn test -Dbrowser=chrome -Dtest=FirstTest

5. TEST ENVIRONMENT SETUP



Create the environment files

- In src\test\resources\environments
- In each file create environment specific properties by pattern:

propertyname=propertyvalue

Create the environment files

- In the .properties files, enter new property:

- qa: homepageurl=https://dummyqasite
- staging: homepageurl=https://dummystagingsite
- production: homepageurl=https://imalittletester.com/

Creating the EnvironmentUtil class

- In src\main\java\utils, create an EnvironmentUtil class
- Create the following method:

```
public String getEnvProperty(String propertyName)
throws IOException {
    Properties prop = new Properties();
    prop.load(new
FileInputStream("src/test/resources/environments/" +
        System.getProperty("environment") +
".properties"));
    return (prop.getProperty(propertyName));
}
```

Using the environment property files

- Initialize the EnvironmentUtil class in your test class
- But most tests will need to do this step
- Avoid repeating code by creating a BaseClass
  - It will be extended (inherited) by all test classes
  - It will hold the EnvironmentUtil initialization
  - All tests that inherit this class will have access to the code from the EnvironmentUtil class
  - Less code in tests

The BaseClass

```
public class BaseClass {
    protected EnvironmentUtil environmentUtil = new
EnvironmentUtil();     }
```

- Create a new test class: SecondTest
- Make it extend BaseClass:

```
public class SecondTest extends BaseClass {
```

Reading and using the properties from the environment files

- In a test, to refer a property from an environment file:

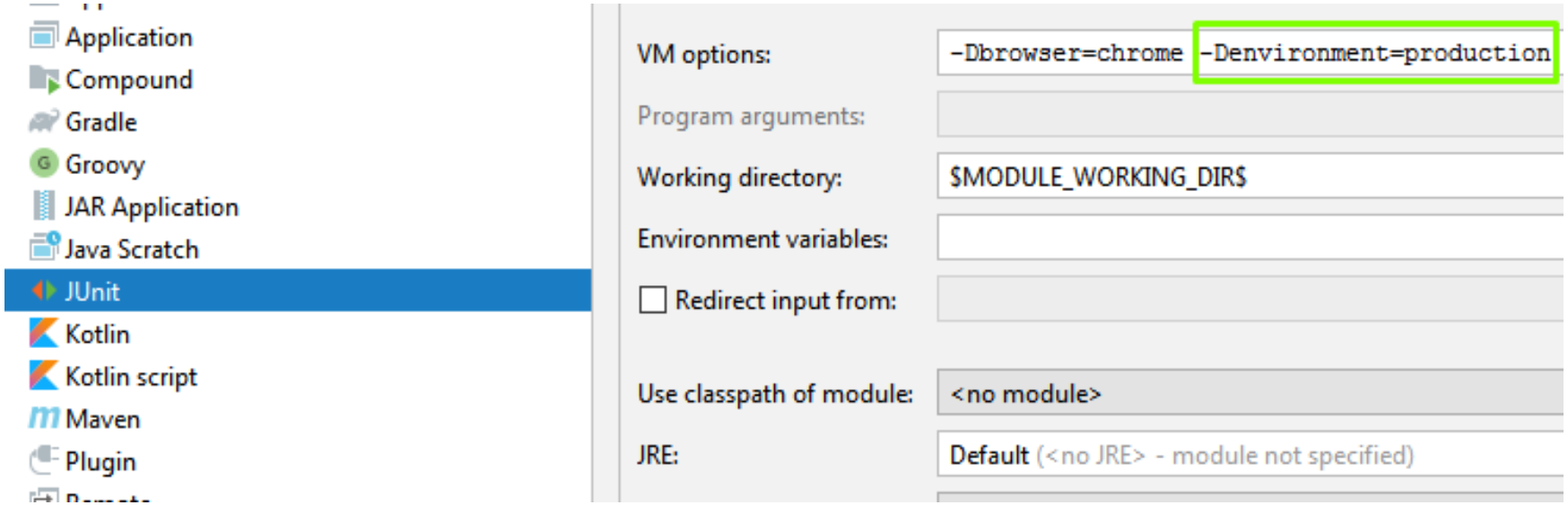
```
environmentUtil.getEnvProperty("nameOfProperty")
```

- Create a new test method that only prints the 'homepageurl' value to the console:

```
@Test
void secondTest() throws IOException {
    System.out.println(environmentUtil.getEnvPrope
rty("homepageurl"));}
```

Running the test

- From Intellij, similar to the browser:



- From command line or Maven Goal:

mvn test -Denvironment=production -Dtest=SecondTest

Exercise

- In the FirstTest class, after the browser opens, open the 'homepageurl'
- Solution:
  - Make FirstTest extend BaseClass

```
class FirstTest extends BaseClass {
```

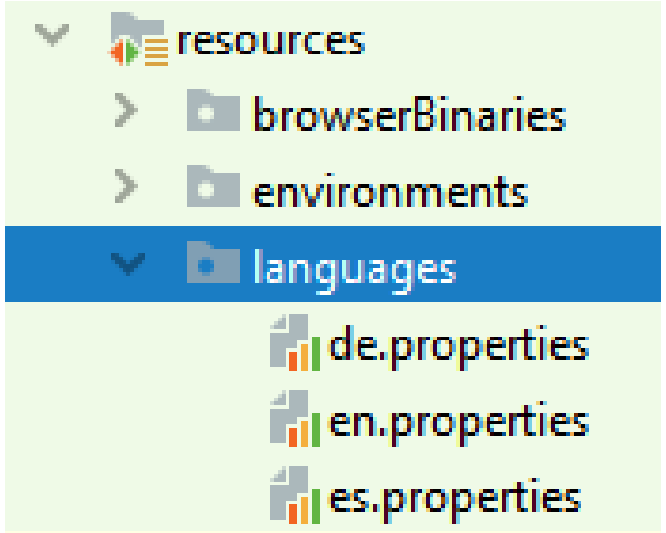
  - Add the code for opening the homepageurl, by reading its value from the property file

```
driver.get(environmentUtil.getEnvProperty("homepageurl"));
```

6. TESTING TRANSLATIONS

Creating the property files

- For each language in scope, create its own property file
- In src/test/resources/languages
  - English: en.properties
  - German: de.properties
  - Spanish: es.properties



- Naming convention for files:

http://www.oracle.com/technetwork/java/javase/java8locales-2095355.html

Adding the properties to the property files

- en.properties:

```
tomato=tomato
cucumber=cucumber
cabbage=cabbage
```

- it.properties:

```
tomato=pomodoro
cucumber=cetriolo
cabbage=cavolo
```

- es.properties:

```
tomato=tomate
cucumber=pepino
cabbage=repollo
```

Reading from the property files

- Create a class in src\main\java\utils: TranslationsUtil. Write this method:

```
public String getTranslation(String key, String language) throws IOException
{
	Properties prop = new Properties();
	FileInputStream input = new
FileInputStream("src/test/resources/languages/" + language +
".properties");
	prop.load(new InputStreamReader(input, Charset.forName("UTF-8")));
	input.close();
	return prop.getProperty(key);  }
```

Writing the test – Italian

- Create a new test class: ThirdTest
- Check that the translation for cucumber in the Italian file is correct

```
public class ThirdTest {
    private TranslationUtil translationUtil = new
TranslationUtil();
    @Test
    void checkItalianWordForCucumber() throws
IOException {
        assertEquals("cetriolo",
translationUtil.getTranslation("cucumber", "it"));
}}
```

Testing all languages in one test

- Parameterized test, checking all vegetables in Italian:

```
@ParameterizedTest
@CsvSource(value = {"cucumber,cetriolo",
"tomato,pomodoro", "cabbage,cavolo"})
void checkAllVegetablesInItalian(String
propertyName, String translation) throws IOException
{
    assertEquals(translation,
translationUtil.getTranslation(propertyName, "it"));
}
```

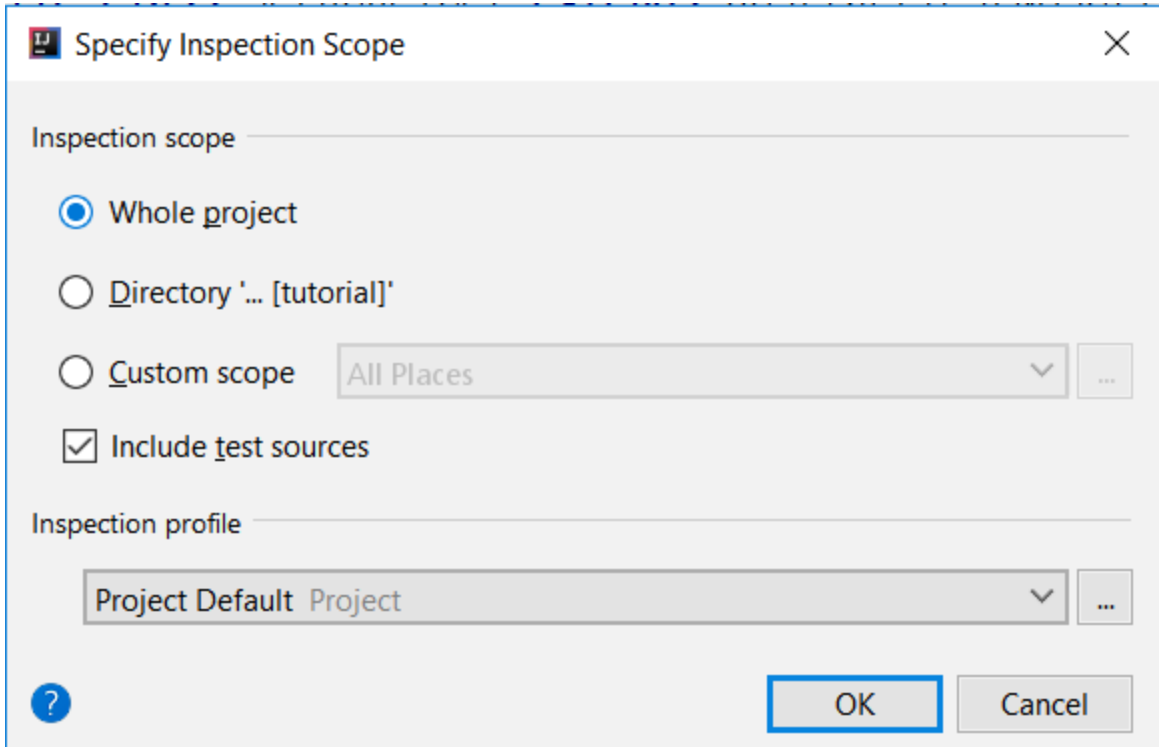
7. TEST DESIGN

Tools: IDE – Inspect code

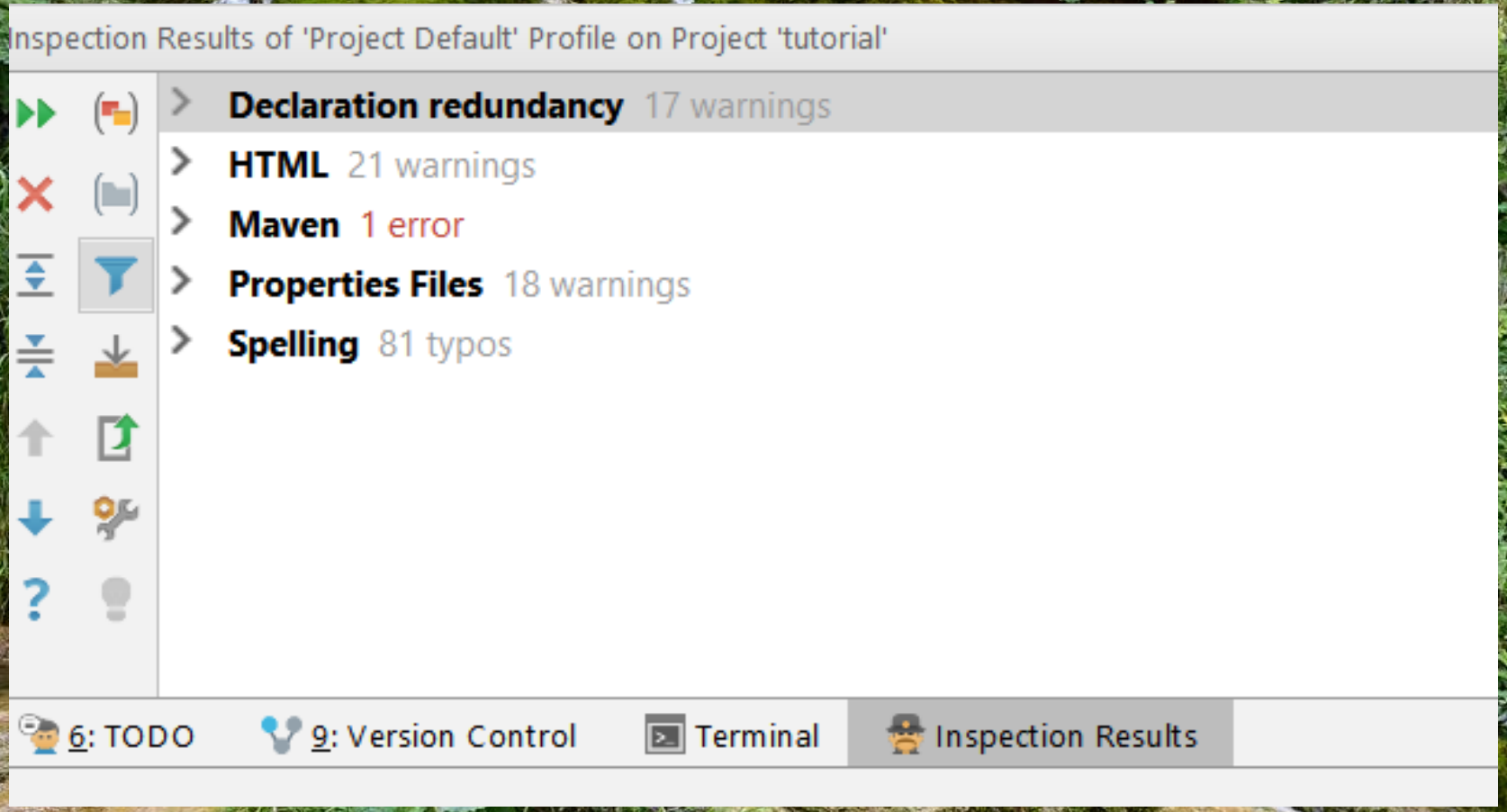
- To automatically check for code issues
- Right-click on a class, package or the project root
- Choose Analyze → Inspect Code

Tools: IDE – Inspect code

- Select scope. For example, “Whole project”
- “Include test sources”
- For the results
  - Check lower tab
  - “Inspection Results”



Tools: IDE – Inspect code

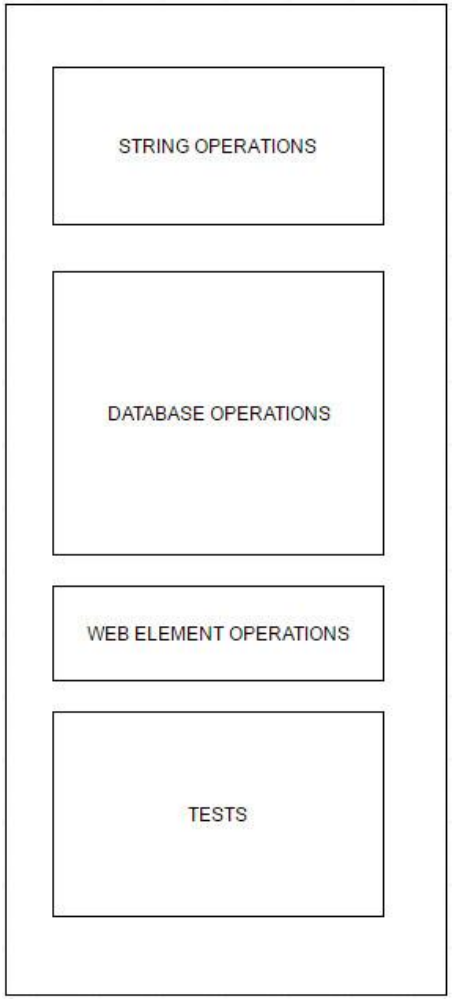


First step – Analysis

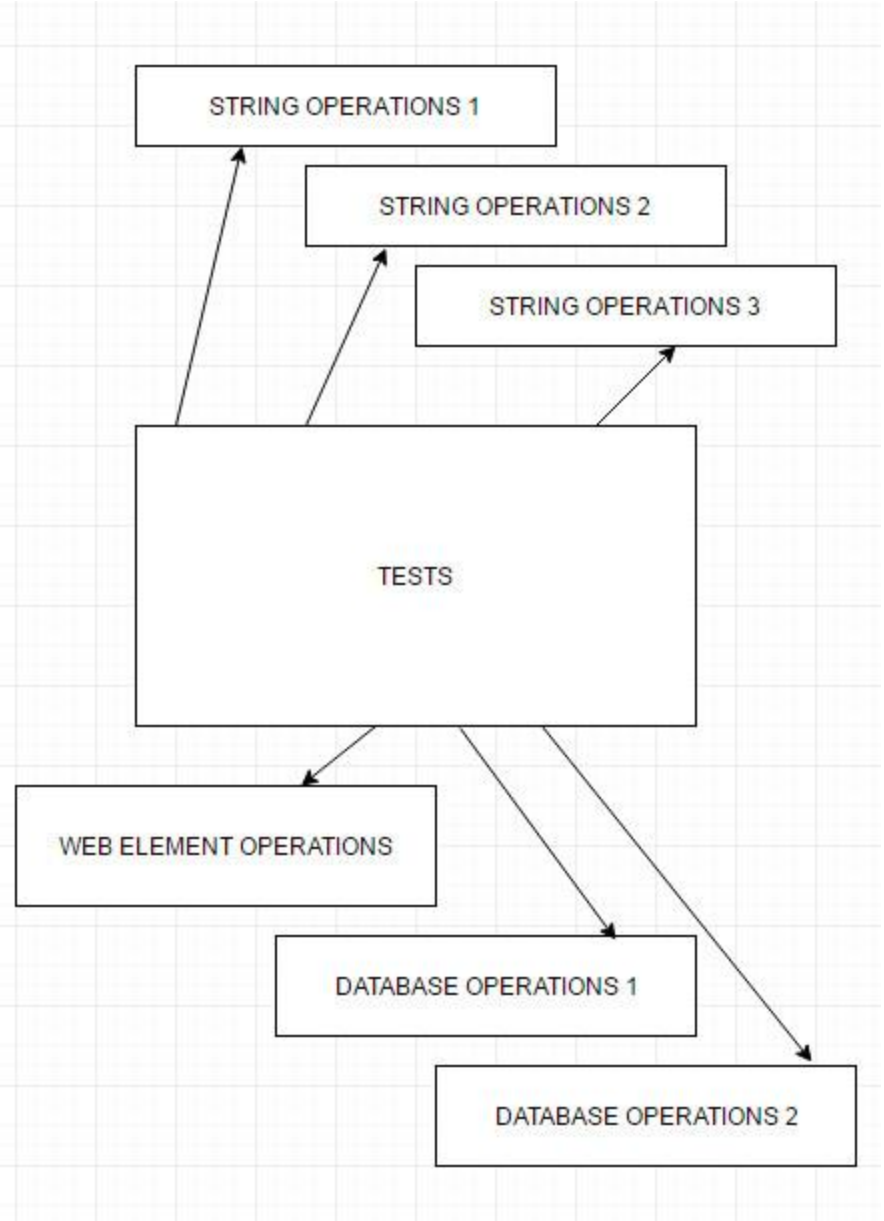
- Need clear requirements
- Spend time identifying best solution
- Write, draw, visualize, discuss

Test design: Separation of concerns principle

- Before:
  - Test class contains everything
    - Data processing
    - Processing that could be used by other tests



- After:
  - Test class contains test steps
  - Processing moved
  - To specific classes
  - Processing used across tests



Base class

- You can create a base class, extended by all test classes
- Can hold useful constants: phone numbers, error messages
- Can hold useful method: e.g. login, logout

Design advice

- Use proper naming for everything
- Put everything in its’ right place
- DO NOT COPY/PASTE code! Create methods for repeating code
- Don’t pass in too many parameters to a method
- Don’t pass constants to a method. Declare them locally instead.

Design advice

- Make sure your try/catches were written properly
  - You addressed the try
  - You addressed the catch
- Use variable scopes properly
- Define inline variables if you only use them once

Write good test output

- To identify what the test did
- When dealing with random data
- Helps when running tests remote

Also

- Do code reviews and refactor where needed
  - For example, add the WebDriver and BrowserUtil declaration to the BaseClass
- Make sure tests have repeatable results

THANKS

Twitter: **@imalittletester**

Blog: https://imalittletester.com/

Comics: https://imalittletester.com/category/comics/

GitHub: https://github.com/iamalittletester