The second assignment

NOTE: This assignment is going to continue where we left off in the first one. It is highly recommended that the first assignment is completed prior to working on this one.

This assignment will focus on introducing Selenium Webdriver in our already laid-out structure.

Converting the project to Maven and installing dependencies

1. Convert our existing project to Maven project
    a. Resources - LINK
    b. Add the proper groupId when prompted (code.auto.qa, for example)
2. Add Selenium Java 3.141.59 as a dependency to pom.xml file
    a. Resources - LINK
    b. Selenium Java - LINK
3. Add WebDriverManager 3.7.1 as a dependency to pom.xml file
    a. WebDriverManager - LINK
4. Import changes by clicking on the prompt in the lower right corner of IntelliJ

Creating a class to hold webdriver

1. Create a new folder under src.main.java and name it **browser**
2. Create a new class under src.main.java.browser and name it **Browser**
3. Add a field do that class that is private, **static**, has a return type of **Chromedriver** and is named **browser**.
4. Add a method to that class that is public, **static**, has a return type of **Chromedriver** and is named **getBrowser();**
5. Add the following logic to the body of getBrowser() method
    a. Using the installed WebDriverManager library, perform the setup for Chromedriver
        i. Resources - LINK, LINK2
    b. If the value of **browser** field is equal to **null**, assign it a **new instance** of ChromeDriver class
    c. **Return** the browser field
6. This class could look something like this:

```
import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium.chrome.ChromeDriver;

public class Browser {
    private static ChromeDriver browser;

    public static ChromeDriver getBrowser(){
        WebDriverManager.chromedriver().setup();

        if (browser == null){
            browser = new ChromeDriver();
        }

        return browser;
    }
}
```

Introducing the browser to the BasePage

1. Navigate to the **BasePage** class
2. Create a new method that is **public**, has a return type of **WebElement**, is named **findElement** and takes a String named **xpath** as the only argument.
3. Add the following logic to the body of the findElement() method:
    a. Call the getBrowser() **static** method from our **Browser** class
    b. Call the findElementByXpath method from the result of getBrowser() method and pass it the xpath argument that we added to our method
    c. Return the result of these calls
    d. Your method should look like something along these lines:

```
package pages;

import browser.Browser;
import org.openqa.selenium.WebElement;

public class BasePage {

    public WebElement findElement(String xpath){
        return Browser.getBrowser().findElementByXPath(xpath);
    }
}
```

Adding locators to the elements within our Page classes

We are now going to replace the printMessage logic from our methods within Page classes with actual Selenium locators. We will focus on Xpath in order to give ourselves plenty of opportunities to practice writing robust locators.

• **Analysis phase:**

1. For each method representing an element in our Page classes do the following:
    a. Revisit the element that method represents on IMDB website
    b. Try to find a unique xpath locator that will highlight exactly that one element
        i. Writing Xpath - LINK
        ii. Verifying Xpath in browsers - LINK
    c. Note the locator down

• **Implementation phase:**

1. For each method representing an element in our Page classes do the following:
    a. Replace the current **printMessage** body with the following:
        i. **Return** the value of the call to iherited **findElement** method, while passing the **xpath locator** we previously noted down as the argument
    b. Update their return type from **void** to **WebElement**
    c. An example of a mapped search field could look like this:

```
package pages;

import org.openqa.selenium.WebElement;

public class HomePage extends BasePage{

    public WebElement searchField(){
        return findElement("//*[@id='navbar-query']");
    }
}
```

Adding automation steps to our main method

1. Remove or comment out the code currently available in the main() method of our Main class
2. Add the following logic to main() method body:
    a. Create a **new instance of Imdb class** and place it in a local variable named **imdb**
    b. Call the getBrowser() **static** method from our Browser class
    c. Call the get() method from the result of getBrowser() method and pass https://www.imdb.com/ as the argument
    d. Write the desired steps that you would like to automate
        i. The simplest two steps could be:
            1. From imdb variable, call the homePage() method, call the searchField() method from it and then call the clear() method from it
            2. From imdb variable, call the homePage() method, call the searchField() method from it and then call the sendKeys() method from it and pass „12 Angry Men" as the argument
    e. Call the getBrowser() **static** method from our Browser class again
    f. Call the quit() method from the result if getBrowser() method
1. The simplest steps could look something like this:

```
import browser.Browser;
import pages.Imdb;

public class Main {

    public static void main(String[] args) {
        Imdb imdb = new Imdb();

        Browser.getBrowser().get("https://www.imdb.com/");

        imdb.homePage().searchField().clear();
        imdb.homePage().searchField().sendKeys("12 Angry Men");

        Browser.getBrowser().quit();
    }
}
```

2. Run or debug the main() method
3. Enjoy your first browser automated actions!
4. Continue adding even more meaningful steps in order to practice