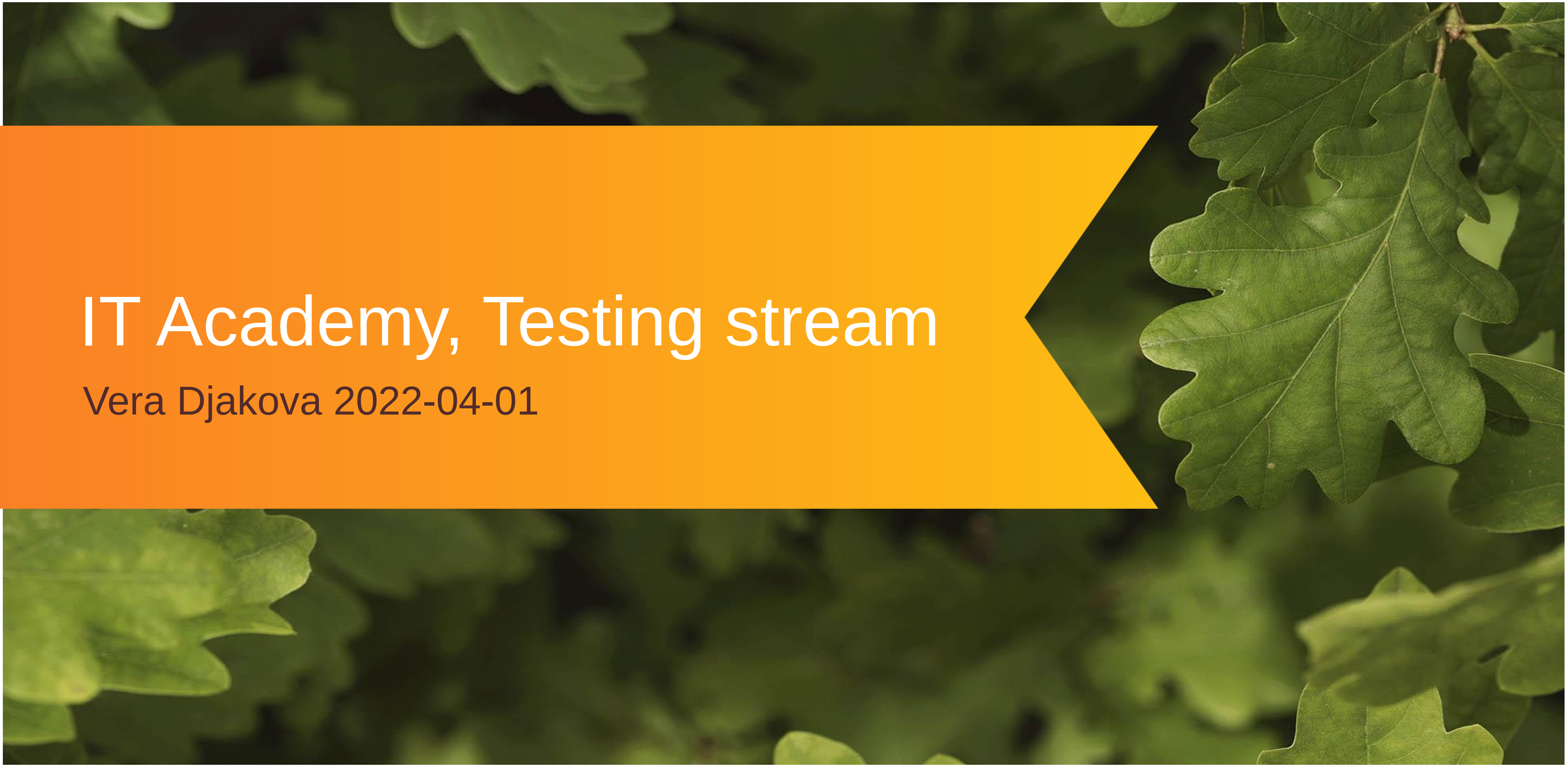
Swedbank



IT Academy, Testing stream

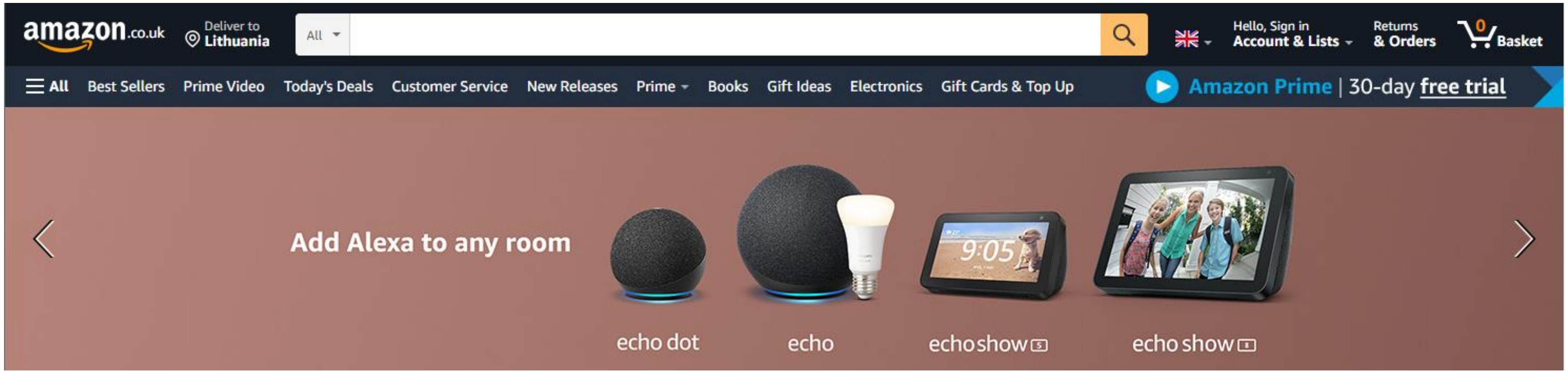
Vera Djakova 2022-04-01

UI test automation

Simulates what user does using UI interface.

It can be used to test separate UI components or full user workflow scenarios.

Can be used to test application end-to-end.



Pros and cons

Pros

- Faster than executing tests manually;
- Can simulate customer behaviour;
- Easy to understand;
- Can do fast cross browser/device testing.

Cons

- High cost (especially initially);
- Usually unstable results;
- Slowness.

Skills needed

HTML & CSS

Skills needed

Dev tools (e.g., Chrome)

Skills needed

- Basics of a programming language (e. g. Javascript, Java, Python…);
- Object-oriented programming/functional programming.

Good to know:

- Git;
- Jenkins (or any other CI/CD platform).

Checking prerequisites

1

Check if you have Google Chrome browser installed

2

Check if you already have **IntelliJ and Java installed**

3

Check if you already have **Git** client by typing **git --version** in your terminal

Lets do something together

Please download chrome webdriver from here:

https://chromedriver.chromium.org/downloads

Meet Selenium

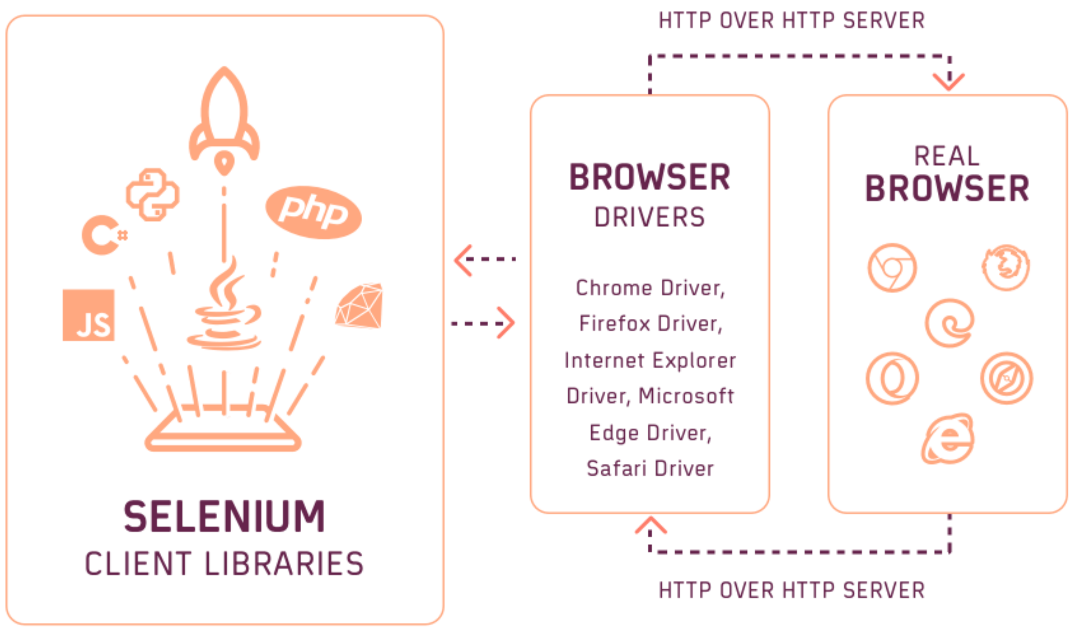
Browser automation test framework

- Through a simple setup, WebDriver can be used with all major browsers. Automate real user interactions in Firefox, Safari, Edge, Chrome, Internet Explorer and more
- Tests could be written in Java, C# or Python

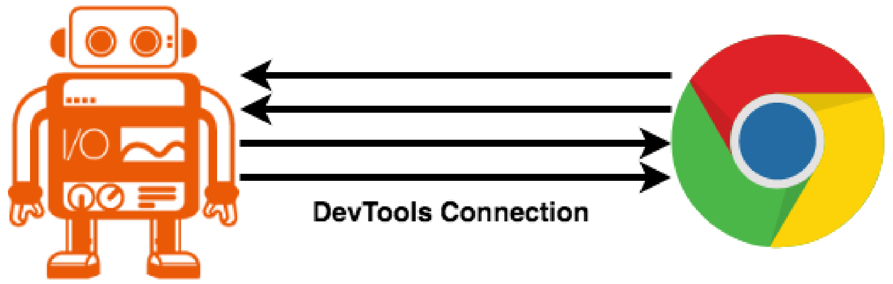
https://www.selenium.dev/

Automation protocols

WebDriver Protocol


HTTP OVER HTTP SERVER
BROWSER DRIVERS
Chrome Driver, Firefox Driver, Internet Explorer Driver, Microsoft Edge Driver, Safari Driver
REAL BROWSER
SELENIUM CLIENT LIBRARIES
HTTP OVER HTTP SERVER

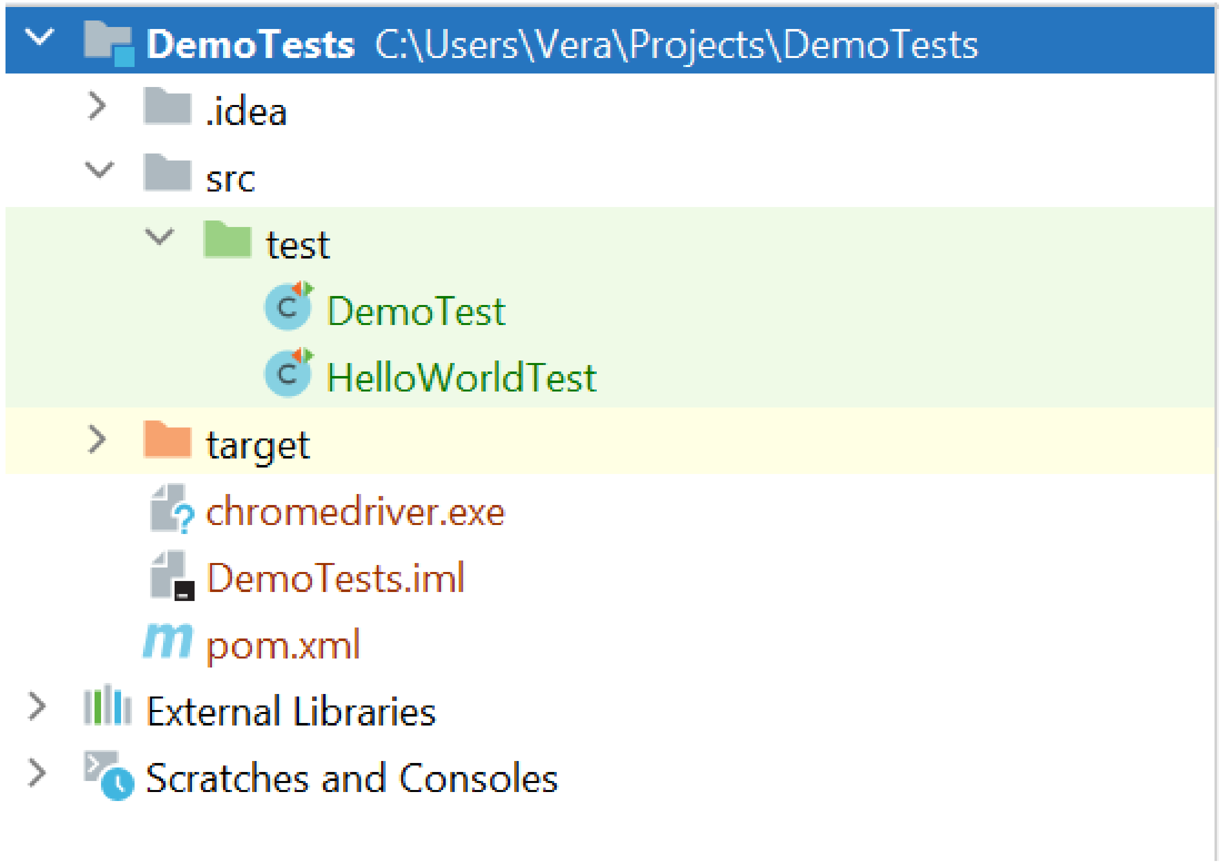
DevTools Protocol



Cloning a project

- Create a **projects** folder somewhere on your computer (e.g C:\\ or /home/<user-name>)
- Open a terminal window (e. g. Git Bash) in the created folder
- Run **git clone https://github.com/dreadonmyhead/automation.git**
- After cloning is finished, lets run maven script

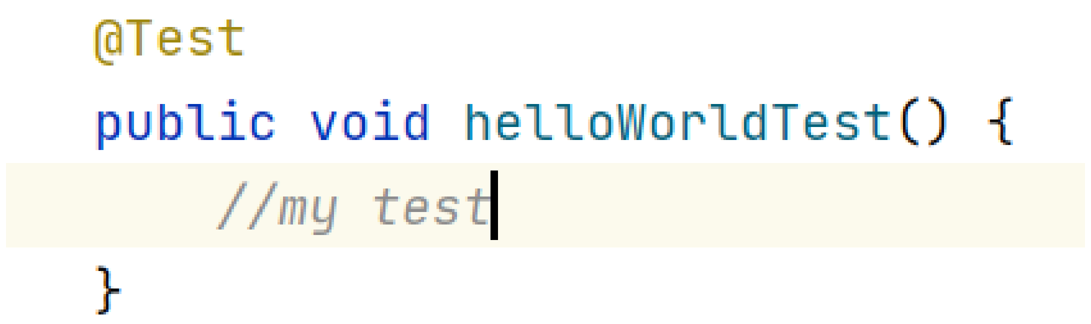
Opening the project in IDE and exploring its structure



```
DemoTests C:\Users\Vera\Projects\DemoTests
  .idea
  src
    test
      DemoTest
      HelloWorldTest
  target
  chromedriver.exe
  DemoTests.iml
  pom.xml
External Libraries
Scratches and Consoles
```

Organizing your tests

- We will use JUnit to write tests



```
@Test
public void helloWorldTest() {
    //my test
}
```

1st example. Hello World

Open **HelloWorldTest.java** file.

Write the first test.

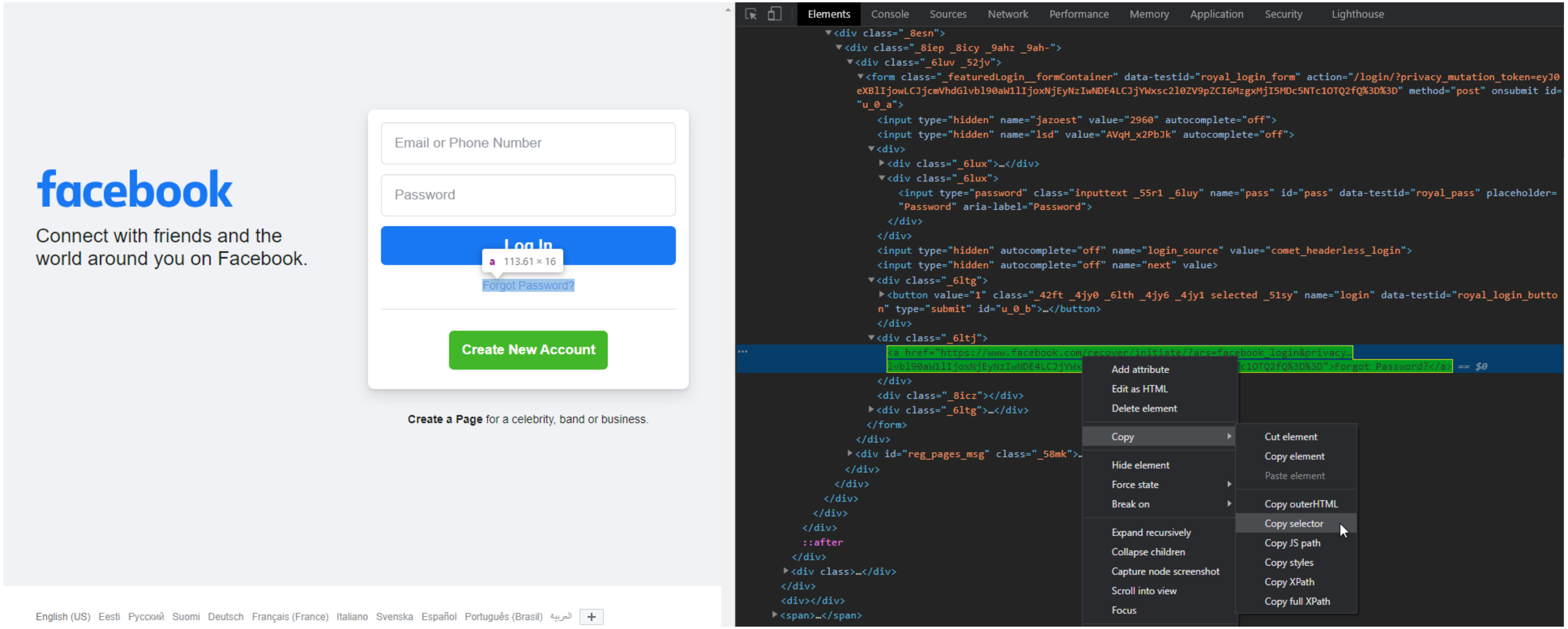
Typical sequence of logic in UI tests

Locating HTML elements

Performing actions on HTML elements

Making assertions

Locating elements



facebook

Connect with friends and the world around you on Facebook.

Create a Page for a celebrity, band or business.

English (US) Eesti Русский Suomi Deutsch Français (France) Italiano Svenska Español Português (Brasil) العربية +
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse

```
▼<div class="_8esn">
  ▼<div class="_8iep _8icy _9ahz _9ah-">
    ▼<div class="_6luv _52jv">
      ▼<form class="_featuredLogin__formContainer" data-testid="royal_login_form" action="/login/?privacy_mutation_token=eyJ0
      eXBlIjowLCJjcmVhdGlvbl90aW1lIjoxNjEyNzIwNDE4LCJjYWxsc2l0ZV9pZCI6MzgxMjI5MDc5NTc1OTQ2fQ%3D%3D" method="post" onsubmit id=
      "u_0_a">
          <input type="hidden" name="jazoest" value="2960" autocomplete="off">
          <input type="hidden" name="lsd" value="AVqH_x2PbJk" autocomplete="off">
        ▼<div>
          ▶<div class="_6lux">…</div>
          ▼<div class="_6lux">
              <input type="password" class="inputtext _55r1 _6luy" name="pass" id="pass" data-testid="royal_pass" placeholder=
              "Password" aria-label="Password">
            </div>
          </div>
          <input type="hidden" autocomplete="off" name="login_source" value="comet_headerless_login">
          <input type="hidden" autocomplete="off" name="next" value>
        ▼<div class="_6ltg">
          ▶<button value="1" class="_42ft _4jy0 _6lth _4jy6 _4jy1 selected _51sy" name="login" data-testid="royal_login_butto
          n" type="submit" id="u_0_b">…</button>
          </div>
        ▼<div class="_6ltj">
...          <a href="https://www.facebook.com/recover/initiate/?ars=facebook_login&privacy…
            lvbl90aW1lIjoxNjEyNzIwNDE4LCJjYWx                        c1OTQ2fQ%3D%3D">Forgot Password?</a> == $0
          </div>
          <div class="_8icz"></div>
        ▶<div class="_6ltg">…</div>
        </form>
      </div>
    ▶<div id="reg_pages_msg" class="_58mk">…
    </div>
  </div>
</div>
</div>
</div>
::after
</div>
▶<div class>…</div>
</div>
<div></div>
▶<span>…</span>
```

Copied selector: `#u_0_a > div._6ltj > a`

Main selectors supported by Selenium

CSS Query Selector

```
<html>
  <body>
    <p class="main">barfoo foobar <a href="https://baz.net">baz</a>
    </p>
  </body>
</html>
```

Link element can be located by using CSS selector:

```
WebElement linkElement = driver.findElement(By.cssSelector("p.main a"));
linkElement.click();
```

Main selectors supported by Selenium (cont.)

xPath

An xPath selector has a format like `//body/div[6]/div[1]/span[1]`

```
<html>
  <body>
    <p>foobar</p>
    <p>barfoo</p>
  </body>
</html>
```

The second paragraph by can be located by calling:

```
WebElement paragraph = driver.findElement(By.xpath("//body/p[2]"));

System.out.print(paragraph.getText()) // outputs: "barfoo"
```

Main selectors supported by Selenium (cont.)

Link text and partial link text

```
<a href="https://www.selenium.dev/">Selenium is cool</a>
```

This link can be located by exact text:

```
  WebElement link = driver.findElement(By.
  linkText(“Selenium is cool"));
System.out.print(link.getText()) //
outputs:“Selenium is cool"
```

Or by partial text:

```
  WebElement link = driver.findElement(By.
  partialLinkText(“Selenium"));
System.out.print(link.getText()) // outputs:
“Selenium is cool"
```

//div[.//a[text()='SELENIUM']]

Main selectors supported by Selenium (cont.)

Element with certain text

```
<h1>Welcome to my
Page</h1>
```

h1 element can be located by partial text:

```
WebElement header =
driver.findElement(By.
xpath("//h1[contains(text(),'Welcome')]"));

System.out.print(header.getText())
 // outputs: "Welcome to my Page"
```

```
<p class="content" id="info1">Selenium rocks</p>
```

p element can be found by calling:

```
WebElement classNameAndPartialText =
driver.findElement(By.xpath("//p[contains(text(),'Welcome')
and @className='content']"));
System.out.print(classNameAndPartialText.getText())
// outputs: "Selenium rocks"
WebElement idAndPartialText = driver.findElement(By.
xpath("//p[contains(text(),'Welcome') and @id='info1']"));
System.out.print(idAndPartialText.getText())
// outputs: "Selenium rocks"
```

Finding multiple elements

findElements method is used to find multiple elements

```
<ul id="menu">
    <li class=“list">Home</li>
    <li class=“list">Developer Guide</li>
    <li class=“list">API</li>
    <li class=“list">Contribute</li>
</ul>
```

```
List<WebElement> elements =
driver.findElements(By.cssSelector(“.list"));
System.out.print(elements.get(2).getText());  // outputs: "API"
```

Question time

Question:

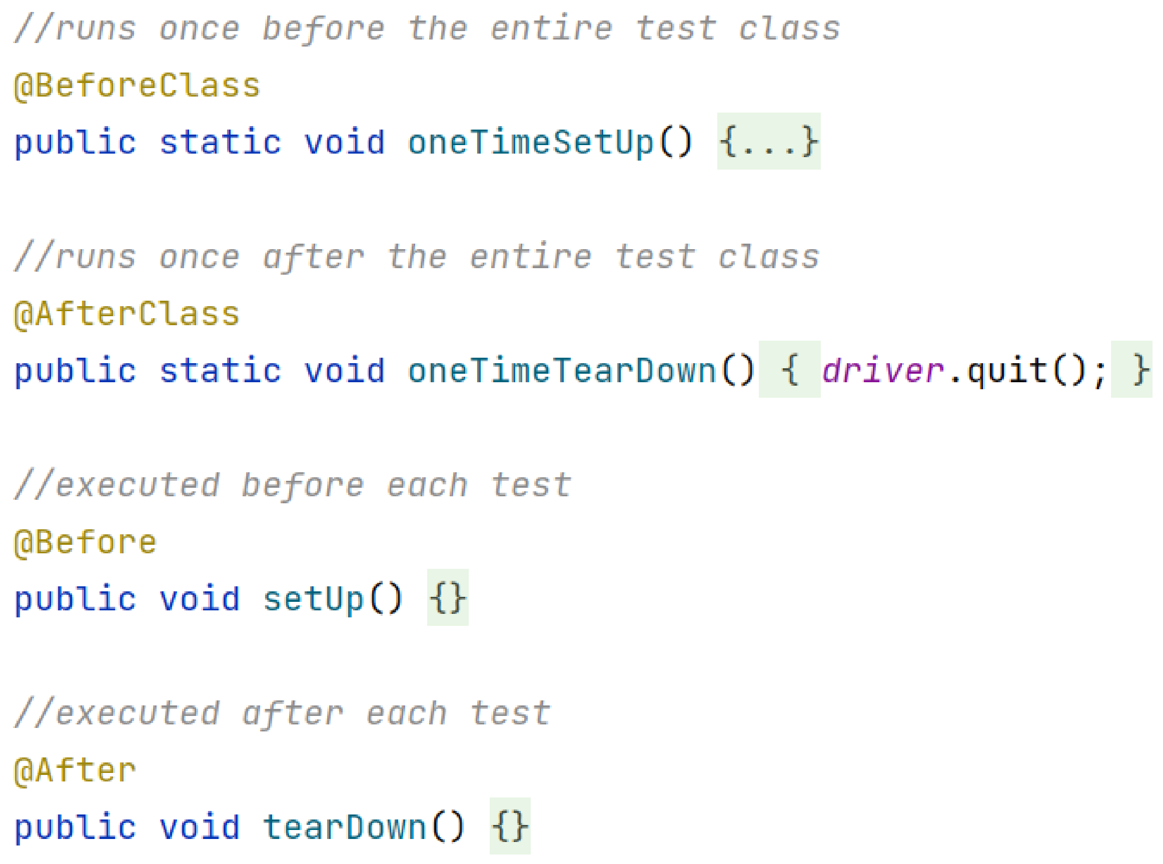
What are the best selectors to use when selecting an element?

Answer:

The ones that are least likely to change.

1. Specific attributes (make separate ones for automation);
2. IDs;
3. Text.

JUnit annotations



```
//runs once before the entire test class
@BeforeClass
public static void oneTimeSetUp() {...}

//runs once after the entire test class
@AfterClass
public static void oneTimeTearDown() { driver.quit(); }

//executed before each test
@Before
public void setUp() {}

//executed after each test
@After
public void tearDown() {}
```

JUnit provides annotationa **@BeforeClass, @AfterClass**, **@Before**, and **@After**.

These should be used to set up preconditions and clean up after your tests.

2nd example. Selectors

Open **SelectorTest.java** file.

Write several tests that find elements on the page using selectors supported by Selenium:

1. Find “Sign in” link in top navigation bar and verify its text
2. Find phone number in top navigation bar and verify it
3. Find last product (Printed Chiffon Dress) and verify its price without discount
4. Find phone number in footer ((347) 466-7432) and verify its value

browser object

- browser.get(url) – loads a page at given url
- browser.getCurrentUrl() – gets current url
- browser.getTitle() – returns current page title
- browser.navigate().refresh() – reloads current page
- browser.manage().getCookies(names) – retrieves cookies visible to the current page
- browser.manage().window().getSize() – returns browser window size

Abstraction used for interacting with the web browser

3rd example. Basic actions

Go to **Contact us** page from homepage and check if we are on the relevant page.

Use **BasicActionTest.java** file.

Interacting with elements

```
WebElement element = driver.findElement(By.cssSelector(".title"));
System.out.print(titleElem.getText());
```

When we have a reference to an element, we can perform different actions on an element

For all elements

- getText()
- getAttribute(attributeName)
- isDisplayed()
- isExisting()
- isEnabled()
- …

For input fields

- clearValue()
- setValue()
- getValue()
- …

For links and buttons

- click()
- …

4th example. Form test

Send a customer service message in **Contact us** page . Fill in all the input fields.

Use **FormTest.java** file.

Question time

Question:

How could we improve the test?

Answer:

Move page specific code such as selectors to a separate class

Page object design pattern

Design pattern which has become popular in test automation

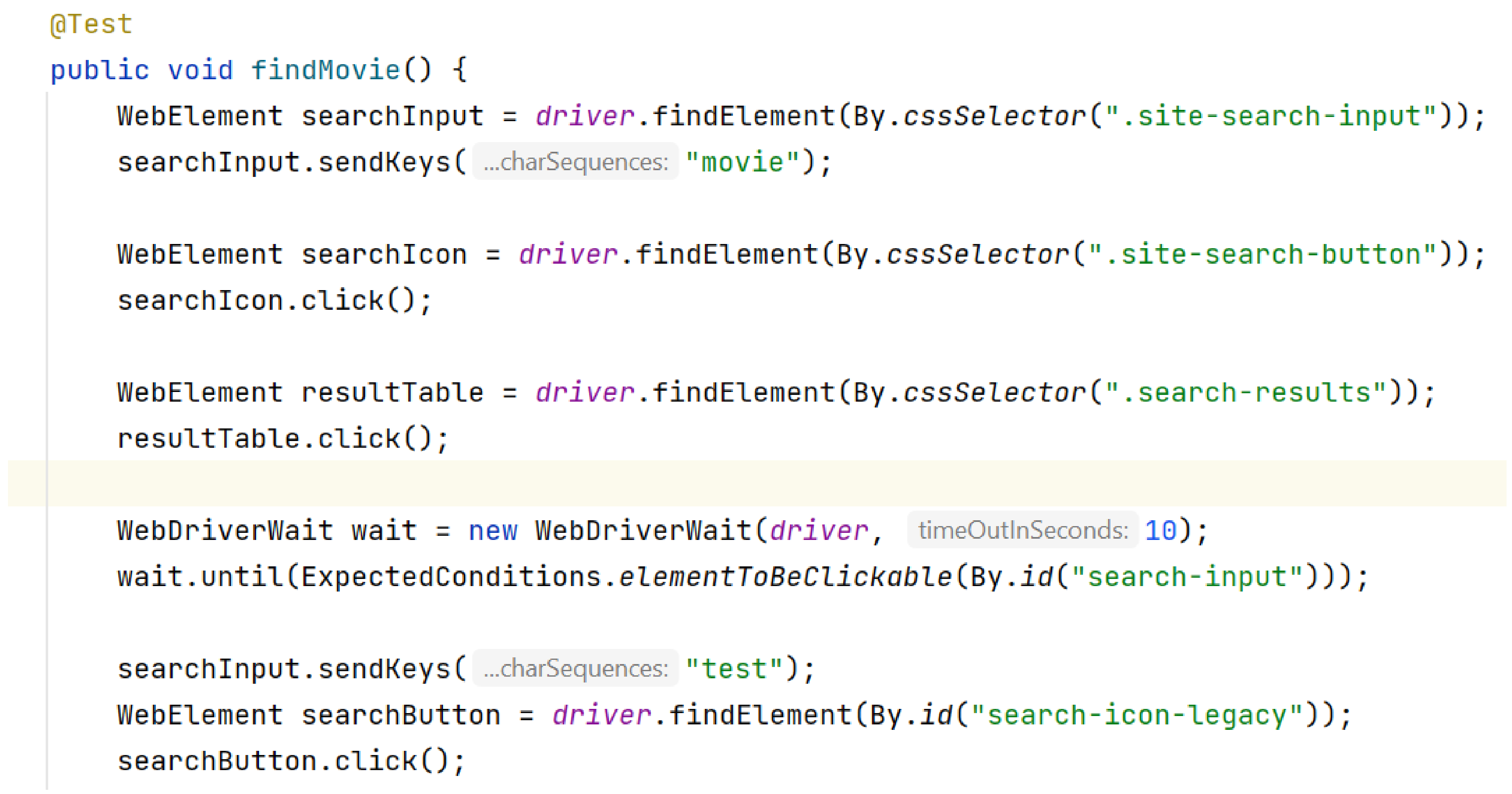
Keeps code clean and easy to maintain

Ideally: page redesign should only affect changes in page object and should keep test unchanged

A page object contains:

- Locators
- Methods that abstract away workflows for that specific element or page

Test case without page object

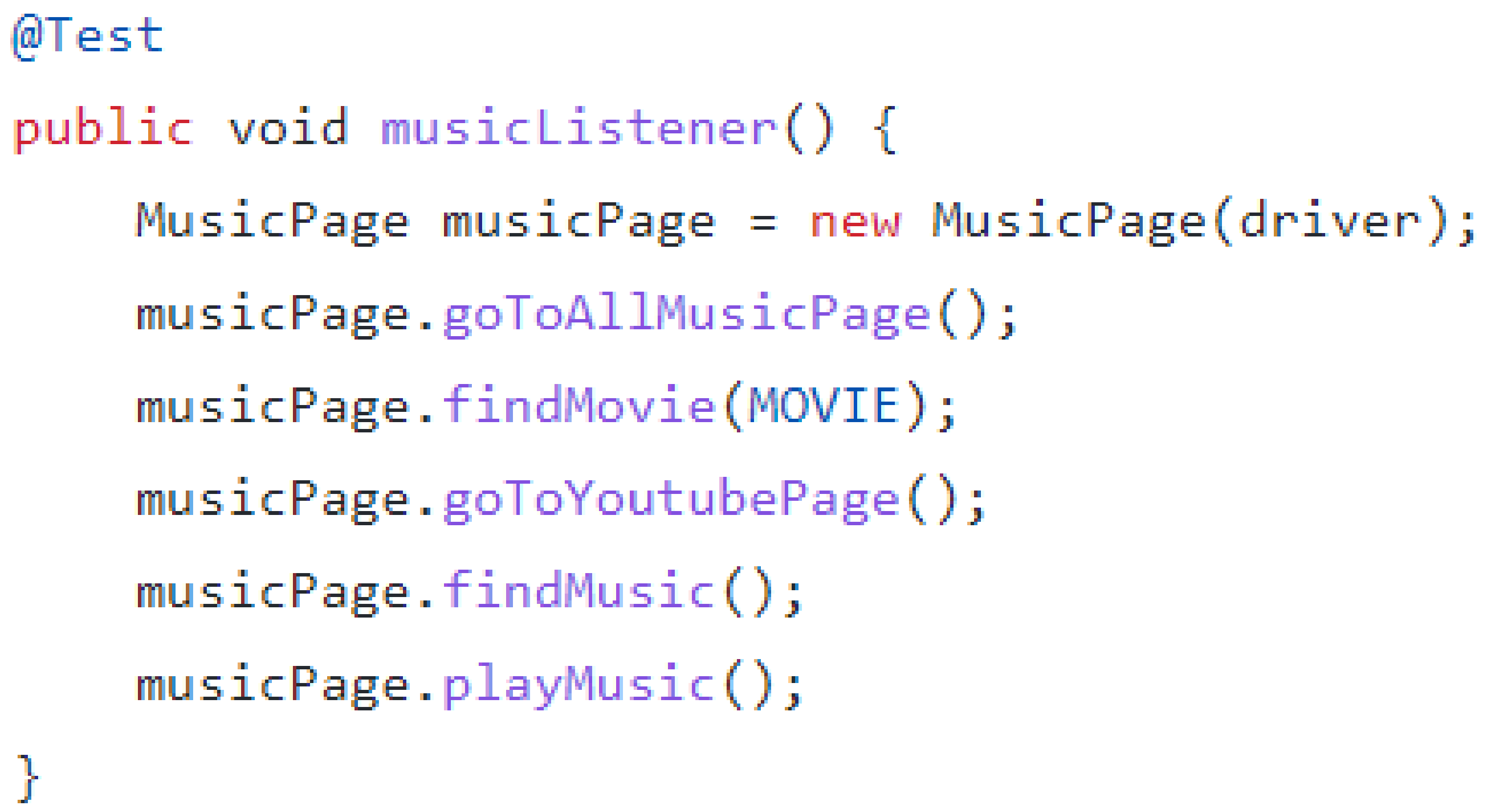


```
@Test
public void findMovie() {
    WebElement searchInput = driver.findElement(By.cssSelector(".site-search-input"));
    searchInput.sendKeys( ...charSequences: "movie");

    WebElement searchIcon = driver.findElement(By.cssSelector(".site-search-button"));
    searchIcon.click();

    WebElement resultTable = driver.findElement(By.cssSelector(".search-results"));
    resultTable.click();

    WebDriverWait wait = new WebDriverWait(driver,  timeOutInSeconds: 10);
    wait.until(ExpectedConditions.elementToBeClickable(By.id("search-input")));

    searchInput.sendKeys( ...charSequences: "test");
    WebElement searchButton = driver.findElement(By.id("search-icon-legacy"));
    searchButton.click();
```

Page object class

```
public void findMovie(String movie) {

    WebElement searchInput = driver.findElement(By.cssSelector(".site-search-input"));
    searchInput.sendKeys(movie);

    WebElement searchIcon = driver.findElement(By.cssSelector(".site-search-button"));
    searchIcon.click();
}

private void findArtist() {
    WebElement resultTable = driver.findElement(By.cssSelector(".search-results"));
    List<WebElement> allResults = resultTable.findElements(By.cssSelector(".album"));
    artistName = allResults.get(0).findElement(By.cssSelector(".artist")).getText();
}

public void goToYoutubePage() {
    driver.get("https://www.youtube.com/");
}
```

Refactored test case



```
@Test
public void musicListener() {
    MusicPage musicPage = new MusicPage(driver);
    musicPage.goToAllMusicPage();
    musicPage.findMovie(MOVIE);
    musicPage.goToYoutubePage();
    musicPage.findMusic();
    musicPage.playMusic();
}
```

5th example. Refactoring time!

Let’s use 4th example and make it more robust.

Use **PageObjectPatternTest.java** file.

Use page object model, test data separation to make your code cleaner.

Assertions

- JUnit comes with a built-in assertion library that allows to make powerful assertions on various aspects of the browser or elements within a web application

```
Assert.assertEquals(0, 4 % 2);
```

```
Assert.assertTrue("Valio".equals(element.getText()));
```

```
Assert.assertFalse(element.isDisplayed());
```

Waiting with Waits

We can try different ways to wait for something:

```
Thread.sleep(1000);

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.elementToBeClickable(By.id("search-input")));
```

6th example. Waits

Add the first product to shopping cart by using quick view function

Use **TestWait.java** file.

Note that product’s quick view opens in an iframe therefore browser.switchToFrame() method will be needed.

7th task. Create login test

Create a new user in http://automationpractice.com/ eshop.

You can use random information to create an account.

Use **TestSignIn.java** file.

Create a test case using page objects, where user would sign in. Use the user that you created.

8th task. Add to cart through search

Use **TestAddToCard.java** file.

Create a test case, where user would add an item to cart through search:

1. User enters search term in the search bar on home page and selects the first search result
2. When product page is opened user adds product to cart
3. Verify the success message

9th task. Buying an item

Use **TestBuyAnItem.java** file.

Create a test case, where user would add an item to cart through homepage and go through the process of buying it.

Things to keep in mind about UI tests

- It’s easy to write a brittle UI test but writing reliable test is not so simple
- Less is more
- Dedicated test environment is vital