SENG275 – Lab 5
Due Wednesday March 3, 11:55 pm

Technical Note: You have a new repository in gitlab named web_testing.git.

This lab will require a working web browser. It has been tested with Google Chrome and Mozilla Firefox, both of which are available for all common platforms. If multiple options are available to you, please install and use **Chrome**. We will attempt to support students using Safari to the extent possible, but configuring Safari to permit automated testing is non-trivial, and if you choose to use that browser you may encounter problems we cannot solve.

Welcome to Lab 5. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm. This lab is worth 4% of your final grade.

Quick summary of what you need to do:

1. Install the webdriver for the browser of your choice (Chrome preferred).
2. Run the tests written in web_testing.git to confirm that it works.
3. Write the tests for uvic.ca as directed in exercise 1. (3%)
4. Perform standards compliance testing as directed in exercise 2. Write your conclusions in the Labs repository, in Lab05/readme.txt. (1%)

You're done!

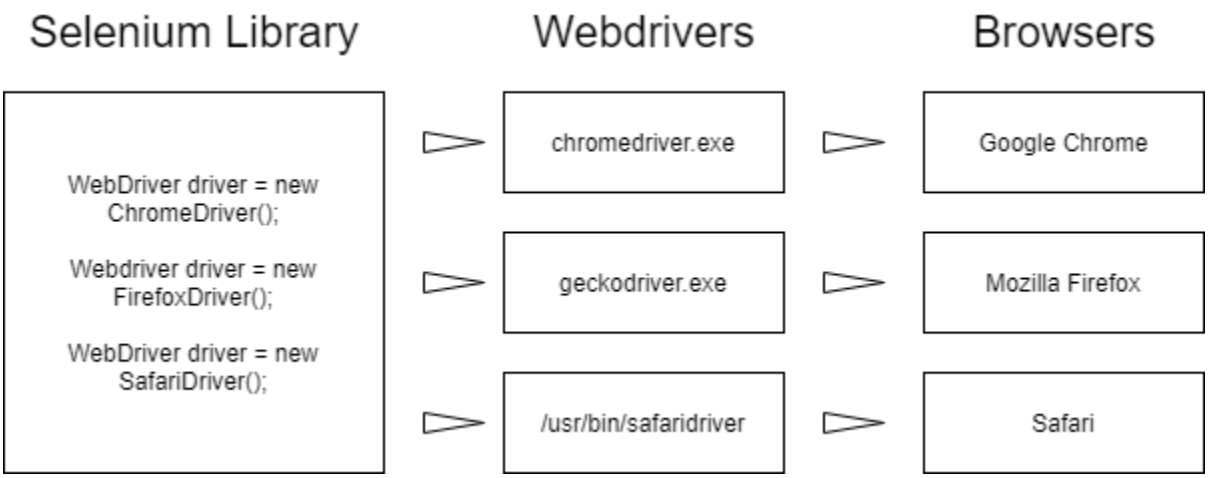
Overview

We would like to test web applications (apps and interactive webpages) the same way we test java applications – by isolating some portion of the program, giving it some known input, and comparing the result to what we expect.

Unlike the java projects we've been testing so far, however, we can't interact with a web-based program by calling its functions directly. Most web applications are written in Javascript, are distributed across numerous remote machines, and only permit access through web browsers that require synchronous interaction from the user and present their output visually, rather than through return values.

We could test these applications manually, by browsing to the page ourselves, and clicking on buttons, entering data into text fields, and looking at the result. However, we can instead automate much of this testing by combining Java code, the Selenium java library, and one or more 'Webdrivers' – programs that *emulate* a human interacting with a browser.

Webdrivers and their installation



Your web_testing repository already has a build.grade dependency specified for selenium:

```
testImplementation 'org.seleniumhq.selenium:selenium-java:3.+'
```

In addition, the classes we'll be using this lab (src/test/java/GoogleTest.java and UVicTest.java) have their imports set up correctly. However, the tests will not run, since we haven't

downloaded the particular Webdriver we need to allow our java program to communicate with our browser.

Identify the browser you'll be testing with. If you have several browsers available, choose Chrome. If you'd like to download a browser, the links are:

Chrome: https://www.google.com/intl/en_ca/chrome/

Firefox: https://www.mozilla.org/en-CA/firefox/new/

Safari: Should already be installed on your Mac

Next, if you're using Chrome, identify the particular version you're using (click the 'kebab', the three vertical dots on the right end of the address bar). Choose Help -> About Google Chrome.

Now download the Webdriver application corresponding to your browser:

Chrome: https://chromedriver.chromium.org/downloads

Firefox: https://github.com/mozilla/geckodriver/releases

Safari: Should already be installed at /usr/bin/safaridriver

Unarchive the downloaded file, and put the resulting executable (chromedriver.exe or geckodriver.exe) somewhere on your local storage. Remember where you placed it. There's no need to run it; selenium will run this executable itself when we run our test.

If you're using the Safari Webdriver, there are additional steps required to configure Safari to permit it to be used. Please see https://developer.apple.com/documentation/webkit/testing_with_webdriver_in_safari and follow the instructions in the section 'Configure Safari to Enable WebDriver Support'.

Selenium

Selenium tests follow the same basic Arrange / Act / Assert structure we've seen before.

- Arrange: specify the particular webdriver to use. Get (navigate to) the website.
- Act: find the element you're interested in, and interact with it if necessary.
- Assert: confirm that the element has the state you expect, or that the interaction had the expected outcome.

Example: GoogleTest.java

We've written a test class that tests the www.google.com website. The tests in GoogleTest.java perform the following:

- Navigate to www.google.com, and see if wwe can see the 'Google' page title.
- Check to see if the google search box is there.
- Check to see if the 'Search Google' button is there.
- Type 'uvic' into the search box and see if it appears
- Click the google search button, and see if we end up at the search results page

Each of the tests does as few of these actions as possible, and contains exactly one assertion – this way we'll know what went wrong if something fails.

```
@BeforeEach
public void setUp(){
    // Chrome
    System.setProperty("webdriver.chrome.driver", "*****LOCATION OF YOUR WEBDRIVER*****");
    browser = new ChromeDriver();

    // Firefox
    // System.setProperty("webdriver.gecko.driver", "*****LOCATION OF YOUR WEBDRIVER*****");
    // browser = new FirefoxDriver();

    // Safari
    // browser = new SafariDriver();

    browser.manage().window().maximize();
}
```

Before each test is run, we need to let Selenium know where we extracted our webdriver to on the previous page. For example, I'm using Chrome, so my setProperty line looks like:

```
System.setProperty("webdriver.chrome.driver",
"d:\\Work\\SENG275\\Chromedriver\\geckodriver.exe");
```

Note the escaped backslashes, which are necessary on windows filesystems.

Loading a webpage

```
@Test
public void googlePageLoads() {
    browser.get("https://www.google.com");
    assertEquals("Google", browser.getTitle());
}
```

Note that we're figuring out whether or not we successfully got to google.com by checking the page title. This is visible to a user as the text on the webpage tab, or in the title bar of the page. This points out an important limitation of web testing – since Selenium *emulates a human user*, it has no access to the inner workings of the web browser. Browsers certainly know when they're finished loading a page, but we have no access to that information.

The only way Selenium knows that the new page has appeared is the same way a human being would know – when the screen we're looking at changes. In this case we're asking Selenium to look at the title, but we could use any visible change to indicate success.

Finding the search box

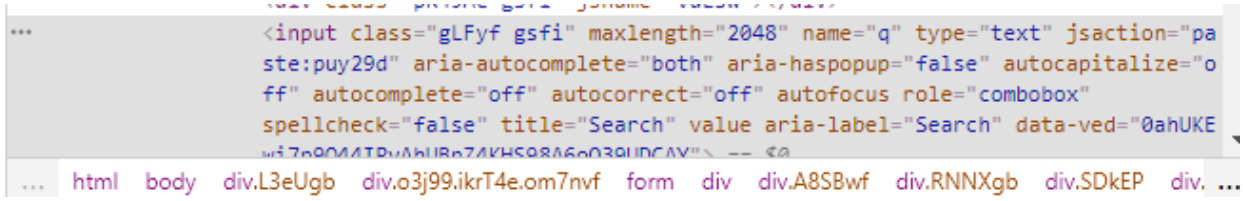
```
@Test
public void googleSearchBoxAppears() {
    browser.get("https://www.google.com");
    WebElement inputBox = browser.findElement(By.name("q"));
// by name - this works
    // WebElement inputBox = browser.findElement(By.className("gLFyf gsfi"));
// by className - this fails
    // WebElement inputBox = browser.findElement(By.cssSelector(".gLFyf"));
// by cssSelector (aka style) - this works
    // WebElement inputBox =
browser.findElement(By.xpath("/html/body/div[1]/div[3]/form/div[2]/div[1]/div
[1]/div/div[2]/input"));
    // by xpath - this works
    assertTrue(inputBox.isEnabled());
}
```

We want to figure out if the search box is 'Enabled', which means that it's on the page, visible and ready for interaction. To do this, we need to find it, using findElement(). This function takes the 'strategy' that we're going to use to find it as an argument. Choosing a strategy can be

frustrating – all of them sometimes work, for some elements, on some pages, but there's no 'one best choice' – often a strategy will fail simply because of arbitrary implementation decisions made years ago by some web developer.

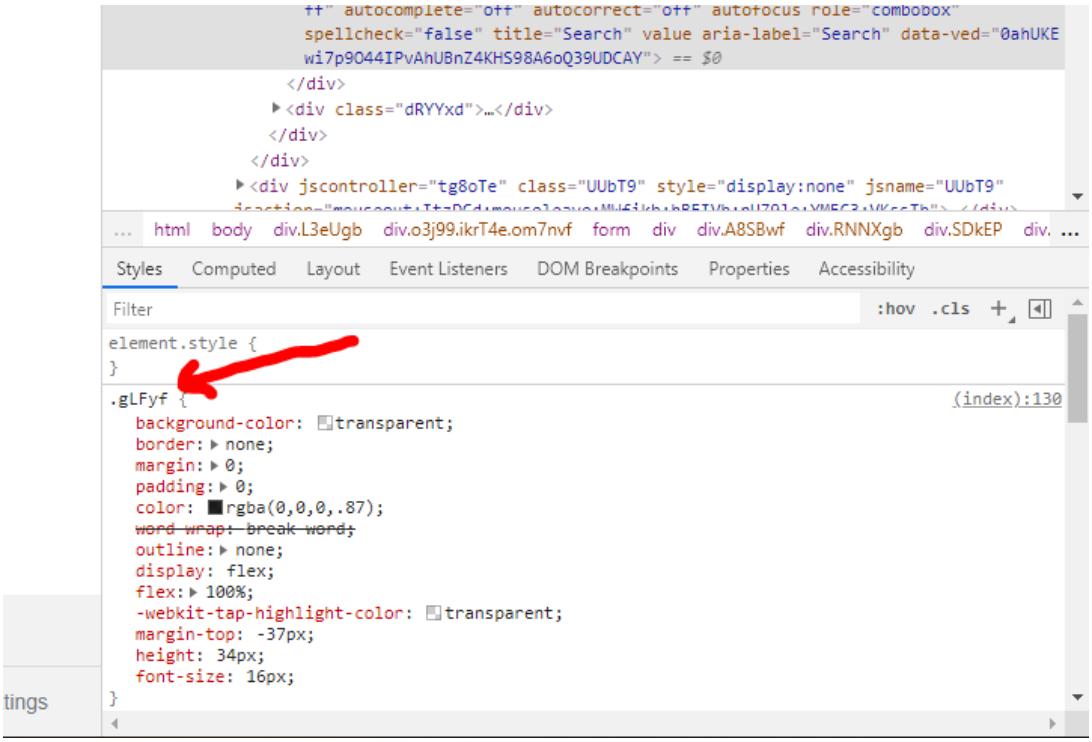
These strategies are: **By.name, By.className, By.id, By.xpath, By.cssSelector, By.tagName, By.linkText, By.partialLinkText** and **By.cssSelector**.

The best way to get started is to examine the HTML source for the element. In Chrome, right-click the element and choose 'Inspect'. ('Inspect Element' in Firefox or Safari). If I inspect the search bar in Chrome, I get the HTML source tree on the right, with the search bar highlighted.



This isn't a course on HTML, so all we can really do is scan the attributes of the element and see if there's anything we can use. This element doesn't have an id or a tag, but it does have a **name**, so I chose the By.name("q"); strategy. Commented-out are some of the other strategies I tried:

- className didn't work – "gLFyf gsfi". ClassNames often fail, since rushed web developers often use the same class name for multiple elements on their page.
- cssSelector worked – this is visible in the Styles section of the inspect page



- xpath worked. Xpath is the location of the element in the page. It's not the physical location (the x and y co-ordinates). Instead it's the *logical* location – every HTML document is structured like a tree, and every element on the page is a leaf. The XPATH is the list of nodes travelled from the root to the element in question. Xpath is a reliable way to find an element, but is fragile - if a developer changes the location of an element, the xpath will change completely, while the name or className will probably stay the same. You can copy the xpath by right-clicking on the highlighted section of the source display, and choosing Copy -> Copy xpath.

The important thing is that there was no way for me to know beforehand which of these strategies would work – I had to try them to find out.

The search button was found in the same way.

Typing into the search box

```
@Test
public void googleSearchTermAppears() {
    browser.get("https://www.google.com");
    WebElement inputBox =
browser.findElement(By.xpath("/html/body/div[1]/div[3]/form/div[2]/div[1]/div
[1]/div/div[2]/input"));
    inputBox.sendKeys("uvic");
    assertEquals("uvic", inputBox.getAttribute("value"));
}
```

Once I had a reference to the text box, I could use sendKeys to emulate typing **uvic** into the box. To check that the text appeared, inputBox.getText() seemed promising, but didn't work. I had to do some googling and find out that input boxes have a "value" attribute, and that worked.

Launching the search

```
@Test
public void googleSearchResultsAppear() {
    browser.get("https://www.google.com");
    WebElement inputBox =
browser.findElement(By.xpath("/html/body/div[1]/div[3]/form/div[2]/div[1]/div
[1]/div/div[2]/input"));
    WebElement searchButton =
browser.findElement(By.xpath("/html/body/div[1]/div[3]/form/div[2]/div[1]/div
[3]/center/input[1]"));
    inputBox.sendKeys("uvic");
    searchButton.click();
    new WebDriverWait(browser, 5).until(ExpectedConditions.titleIs("uvic -
Google Search"));
    assertEquals("uvic - Google Search", browser.getTitle());
}
```

Waiting

Web testing is vulnerable to race conditions, due to the high latency of some operations. For example, after clicking the search button, the search results appear quickly on a good internet connection, but may take several seconds to appear under other circumstances. We often want to wait for some remote operation to conclude before asserting some condition. We can do that either by an explicit pause, or by waiting for some condition.

If we just want to pause for a given length of time, we can pause the execution of our program:

```
try {
    Thread.sleep(5000);
} catch(InterruptedException ignored) {
}
```

This will pause for 5 seconds. If the web page is ready in half a second, this wastes 4.5 seconds every time we run the test. It's more efficient to wait until some expected condition becomes true:

```
new WebDriverWait(browser,5)
.until(ExpectedConditions.titleIs("uvic - Google Search"));
```

This will wait until the new title appears, or 5 seconds, whichever is quicker. Try typing ExpectedConditions. and see what options the autocorrect suggests.

Exercise 1 (3.0%):

Go to uvic.ca. In UVicTest.java, test the following:

- The webpage loads with **Home – University of Victoria** as its title.
- The webpage contains a search button (the magnifying glass in the upper right)
- When the button is pressed, the search bar appears.
- When the letters **csc** are typed in that search bar, they appear correctly.
- When the search for **csc** is launched (either by clicking the search button at the end of the search bar, or by sending Keys.ENTER, a new webpage loads with **Search – University of Victoria** as its title.
- When you load uvic.ca, somewhere on that page, the phone number **1-250-721-7211** appears. Note that it is not enough to find the link – your test must confirm that the phone number is **1-250-721-7211.**

Each of these six tasks will be worth 0.5%. Each of these tasks should involve one assertion – combining several tasks into one test will result in a lower grade.

Hints:

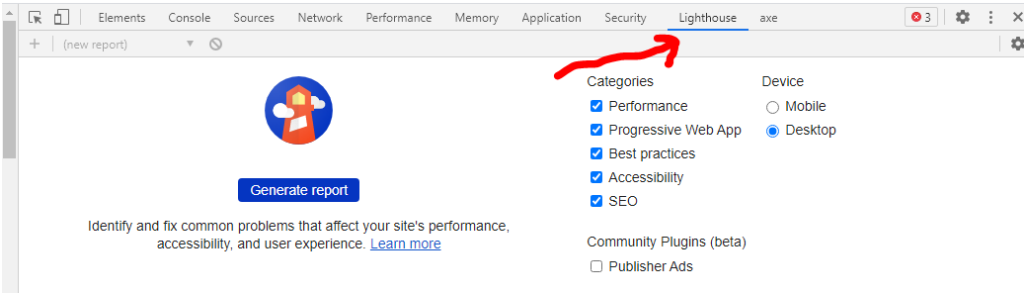
This is not a course on the web or HTML, so the following hints will be provided:

- Remember how to do string concatenation in Java, and how to deal with strings that have quotation marks in them.
- You can wait for something to become visisble with ExpectedConditions.visibilityOf().
- Finding the telephone number is intended to be difficult. You can't just use the xpath, because that finds the link – it doesn't specify whether the text displayed is **1-250-721-7211.** Take a look at the information in the inspect view and see if there's anything there you can use.

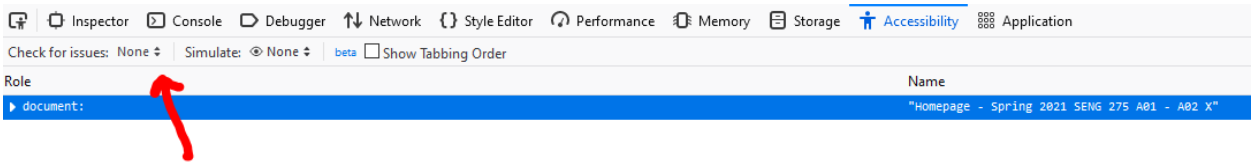
Standards Compliance Testing

In addition to checking the functionality of a web application, it's important to confirm that these sites comply with certain standards. Some of these standards encourage good practices in terms of cross-browser compatibility, network efficiency and performance, and others (such as accessiblity standards) carry the force of law.

Automated tools check to check websites for design, performance and accessibility issues exist. The easiest to use for our purposes is Google Lighthouse, which is shipped with the Chrome Browser. We can run it by clicking on the kebab at the top right of the window, and choosing More Tools -> Developer Tools. We can then generate a report on the current web page.



A less-powerful alternative for Firefox is the Accessibility tool (Web Developer -> Accessiblity).



Change the Check for Issues box to "All" and it will generate a report on the current web page.

Other standards testers are available as plug-ins for popular browsers.

Exercise 2 (1%):

Run a standards test against the SENG 275 Brightspace main page. In your labs repository, in Labs/Lab_5/readme.txt, answer the following:

- Which standards tool did you use?
- Name any test that passed, and describe what the test found in your own words (100 words max) (0.5%)
- Name any test that failed (yellow or red) and describe the problem in your own words (100 words max) (0.5)