# Tutorial: Selenium Webdriver

Automated Testing of a Web App

## Introduction

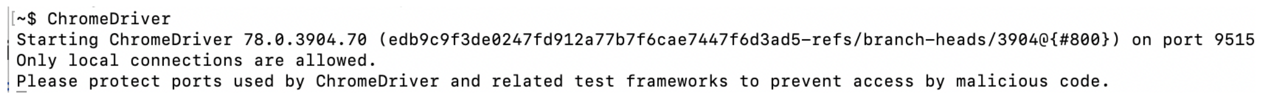
Automated testing is critical in any software engineering project. Web apps are notoriously difficult to test with automated tests. In this tutorial, you will learn to use Selenium Webdriver--an automated testing framework for testing web applications. The tests you create with Selenium Webdriver are typically end-to-end tests. Unlike unit tests that test the individual units that make up an application, automated end-to-end tests simulate a user using an application and are intended to test whole applications.

The web application we will be testing is modified from Dr. Zappala's CS 260 todo-graph example (found here: https://github.com/zappala/cs260-examples). This web application allows a user to enter, complete, delete, and display tasks (todos). We have provided the web application for you. In this tutorial, you will create tests for the application.

## Setup

There are a few things you need to set up before starting this tutorial. Follow these steps to confirm that you have Selenium Webdriver installed and ready and to create and set up a project that will give you a starting point for the tutorial.

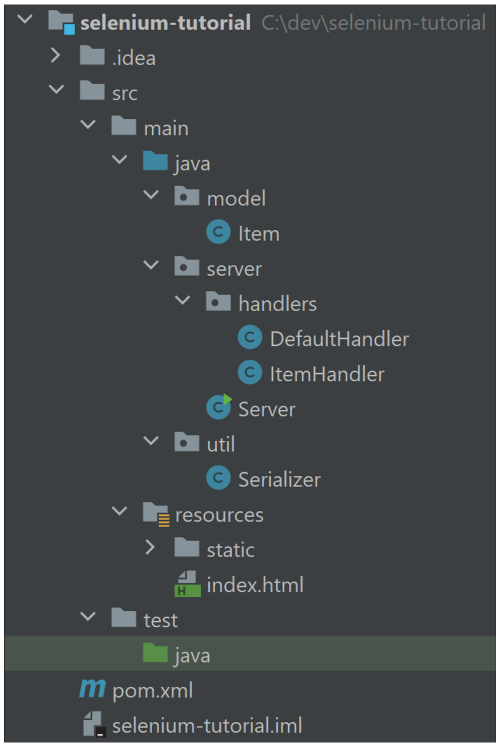
1. Make sure you have installed the Chrome web browser and chromedriver (from the topic preparation assignment).
2. Confirm that you have successfully installed ChromeDriver by opening a terminal or command prompt and typing: ChromeDriver. You should see output that looks something like this:



```
~$ ChromeDriver
Starting ChromeDriver 78.0.3904.70 (edb9c9f3de0247fd912a77b7f6cae7447f6d3ad5-refs/branch-heads/3904@{#800}) on port 9515
Only local connections are allowed.
Please protect ports used by ChromeDriver and related test frameworks to prevent access by malicious code.
```

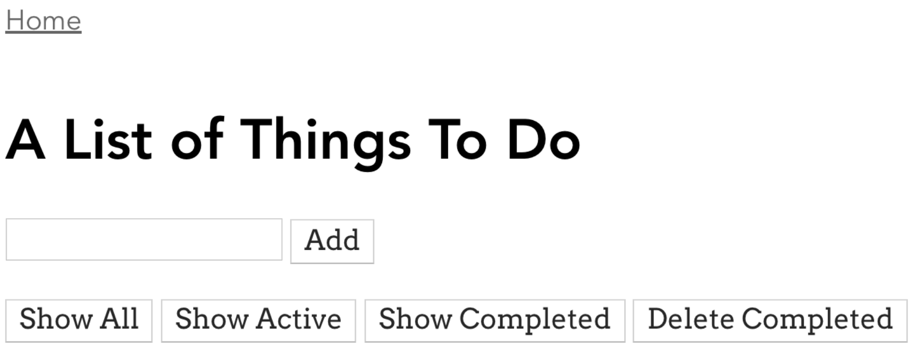
3. Once you confirm that ChromeDriver is properly installed, you can close the terminal window to shut down ChromeDriver. The tests we write will start ChromeDriver as needed.
4. Download the provided selenium-tutorial.zip file and unzip it to where you want IntelliJ to open the project.
5. In IntelliJ, choose the option to create a new project from existing sources.

6. A window will pop up asking you where the new project is located. Find the directory where you unzipped the provided files, select the pom.xml file, then click OK.
7. IntelliJ will then create the files and configurations necessary to run the project. Confirm that you have a similar directory structure as below:



```
selenium-tutorial C:\dev\selenium-tutorial
  .idea
  src
    main
      java
        model
          Item
        server
          handlers
            DefaultHandler
            ItemHandler
          Server
        util
          Serializer
      resources
        static
        index.html
    test
      java
  pom.xml
  selenium-tutorial.iml
```

8. Run the main method in the server.Server class. You should see "Server started on port 8080" in the console.
9. Open a Chrome browser[1] and go to the following URL: http://localhost:8080. You should see a page that looks like this:

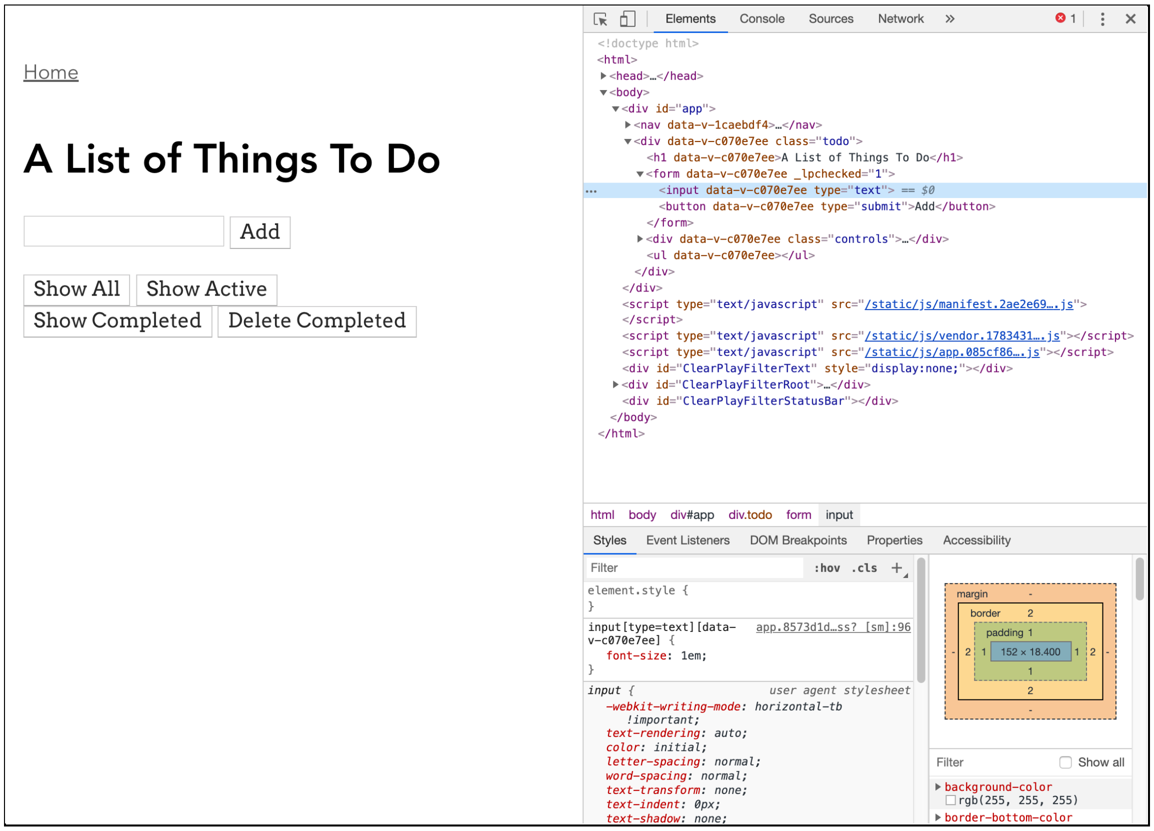


If you don't see the web application in your browser, get help from other students or a TA before continuing.

[1] Other browsers could be used but we will be using the developer tools included in Chrome to view details about the application.

# Tutorial

1. With Chrome open and displaying the TODO application, right-click on the input element (the text field to the left of the 'Add' button) and select 'Inspect'. You should see something like this:



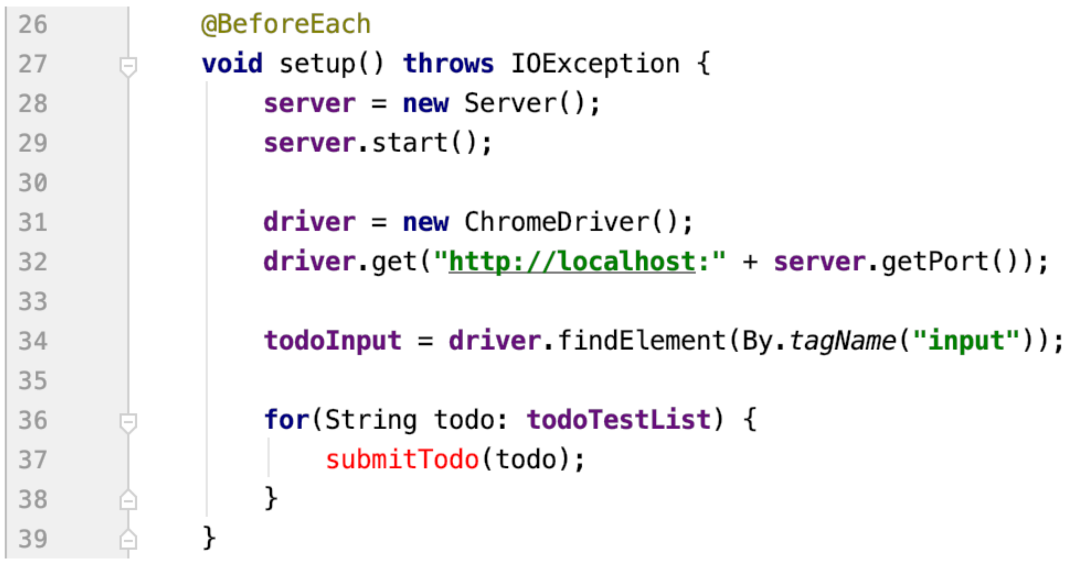
```
<!doctype html>
<html>
 ▸<head>…</head>
 ▾<body>
   ▾<div id="app">
     ▸<nav data-v-1caebdf4>…</nav>
     ▾<div data-v-c070e7ee class="todo">
        <h1 data-v-c070e7ee>A List of Things To Do</h1>
       ▾<form data-v-c070e7ee _lpchecked="1">
          <input data-v-c070e7ee type="text"> == $0
          <button data-v-c070e7ee type="submit">Add</button>
        </form>
       ▸<div data-v-c070e7ee class="controls">…</div>
        <ul data-v-c070e7ee></ul>
      </div>
    </div>
    <script type="text/javascript" src="/static/js/manifest.2ae2e69….js">
    </script>
    <script type="text/javascript" src="/static/js/vendor.1783431….js"></script>
    <script type="text/javascript" src="/static/js/app.085cf86….js"></script>
    <div id="ClearPlayFilterText" style="display:none;"></div>
   ▸<div id="ClearPlayFilterRoot">…</div>
    <div id="ClearPlayFilterStatusBar"></div>
  </body>
</html>
```

```
element.style {
}
input[type=text][data-      app.8573d1d…ss? [sm]:96
v-c070e7ee] {
  font-size: 1em;
}
input {                     user agent stylesheet
  -webkit-writing-mode: horizontal-tb
     !important;
  text-rendering: auto;
  color: initial;
  letter-spacing: normal;
  word-spacing: normal;
  text-transform: none;
  text-indent: 0px;
  text-shadow: none;
```

```
▸ background-color
  rgb(255, 255, 255)
▸ border-bottom-color
```

This shows you the content and structure of the HTML making up the web page and the styling applied to the page. It also highlights the HTML for the element we said we wanted to inspect. We'll make use of this ability to inspect the HTML of elements on a page later when writing our tests.

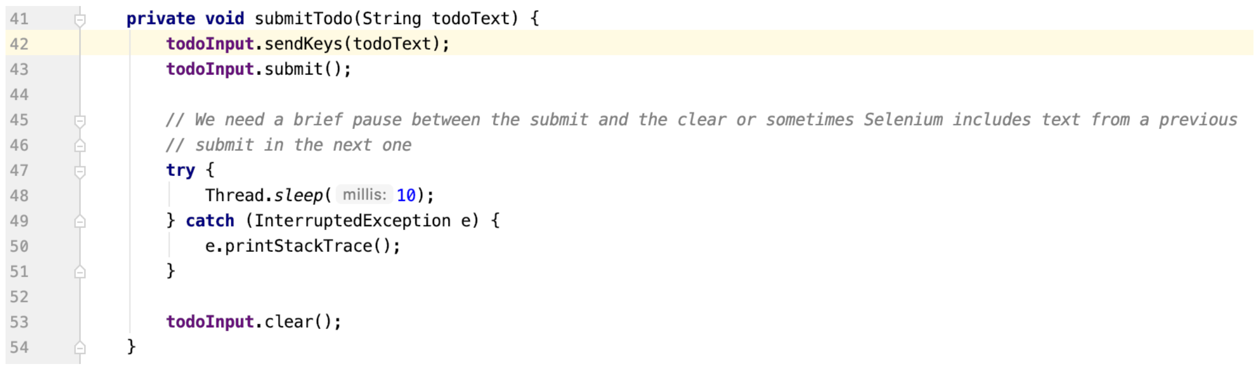
2. In this lab, we will test that the web application served up by the `Server` class behaves appropriately. The starter to the tests is in src/test/java/server/ServerTest.java. Open the file and understand what is going on. Below is the explanation of the `setUp()` method.
   a. `@BeforeEach` setup method does the following:
      i. Start the Server class
      ii. Start a Selenium WebDriver attached to our web application and open the application in a browser
      iii. Get a reference to the input element
      iv. Use the input element to load a list of todos into our application



```java
26      @BeforeEach
27      void setup() throws IOException {
28          server = new Server();
29          server.start();
30
31          driver = new ChromeDriver();
32          driver.get("http://localhost:" + server.getPort());
33
34          todoInput = driver.findElement(By.tagName("input"));
35
36          for(String todo: todoTestList) {
37              submitTodo(todo);
38          }
39      }
```

Lines 28 and 29 start the web server that serves the web application we intend to test.
Lines 31 and 32 create the ChromeDriver instance and use it to open a Chrome browser page that accesses the web application.
Line 34 gets a reference to the input field of the displayed web application.
Lines 36 - 38 submit a list of todos to the application.
Line 37 calls `submitTodo(...)` method, which you would start implementing next.

3. Remove the “TODO” comment in the `submitTodo(...)` method that is below the `teardown()` method, and fill it out with the code shown below:

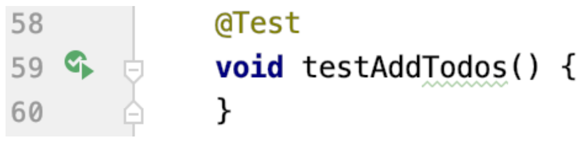


```java
    private void submitTodo(String todoText) {
        todoInput.sendKeys(todoText);
        todoInput.submit();

        // We need a brief pause between the submit and the clear or sometimes Selenium includes text from a previous
        // submit in the next one
        try {
            Thread.sleep( millis: 10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        todoInput.clear();
    }
```

This method uses the `todoInput` element we retrieved in the `setup()` method to “type” the specified string into the input field, submit the string to the application, and then clear the field in preparation for the next time it’s called.

4. We are now ready to see Selenium in action. Run the empty testAddTodos() test method. Doing so will execute the code in the setUp() method. All we need now is a test that will cause the `setup()` method to execute before the test method is executed. We

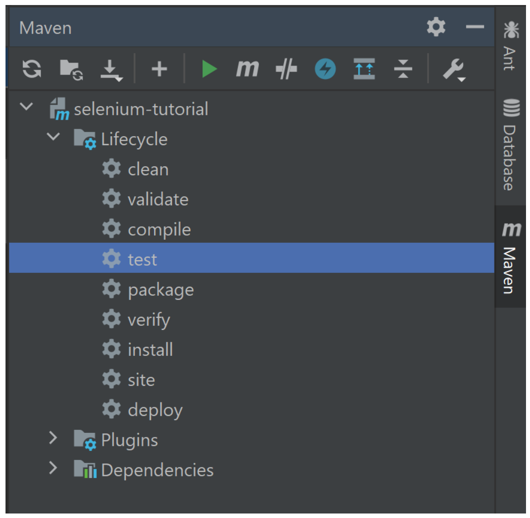
will start by just writing an empty test method so we can confirm that Selenium opens and interacts with the web application correctly. Write the following empty test method below the `submitTodo(...)` method:



```
@Test
void testAddTodos() {
}
```

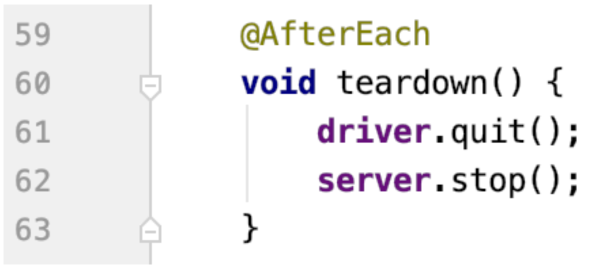
5. Run the test. You should see a browser window open and display the TODO web application. You should also see 6 todos added to the web application as if a user were typing them in the input field and pressing enter. If you look at the test results, you should see that your test passed. Your test may have opened more than one browser window. This is ok and is part of the normal operation of Selenium.

   If your test did not pass, or you did not see the application with the todos being entered, get help from other students or a TA before continuing. To help the TAs better understand the problem, there is an option to have ChromeDriver log the configuration and any errors it may encounter. To do that, run your tests using Maven by selecting the Maven tab on the right in IntelliJ (You can also find it under View > Tool Windows > Maven) and double clicking on "test" shown below:

   Maven will run the tests in ServerTest.java and configure ChromeDriver to output a chromedriver.log file in the root directory of your project.

6. Notice that the browser windows the `setUp()` method opened are still open even though the test has finished. Close the windows, remove the "TODO" comment in the `teardown()` method, and then add the following to it:

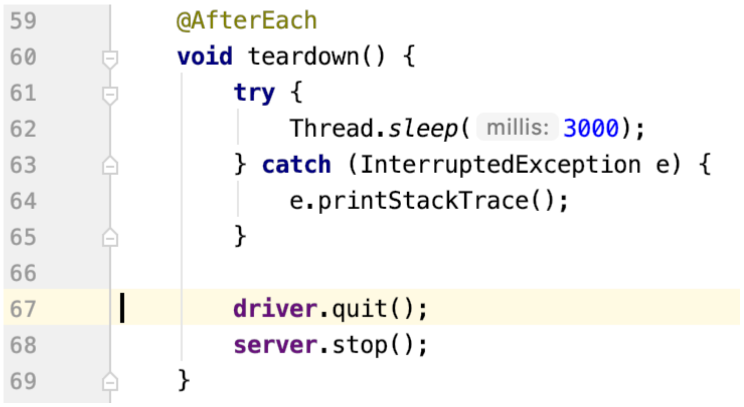


```
59      @AfterEach
60      void teardown() {
61          driver.quit();
62          server.stop();
63      }
```

Line 61 causes your WebDriver instance to close any browser windows it opened. Line 62 stops the server, effectively shutting down the web application. The web application doesn't have a persistent storage mechanism, so shutting the server down after each test ensures that each test starts without any data left in the application from a previous test.

Before re-running your test, manually close the browser windows left open by the previous run and then re-run your test.

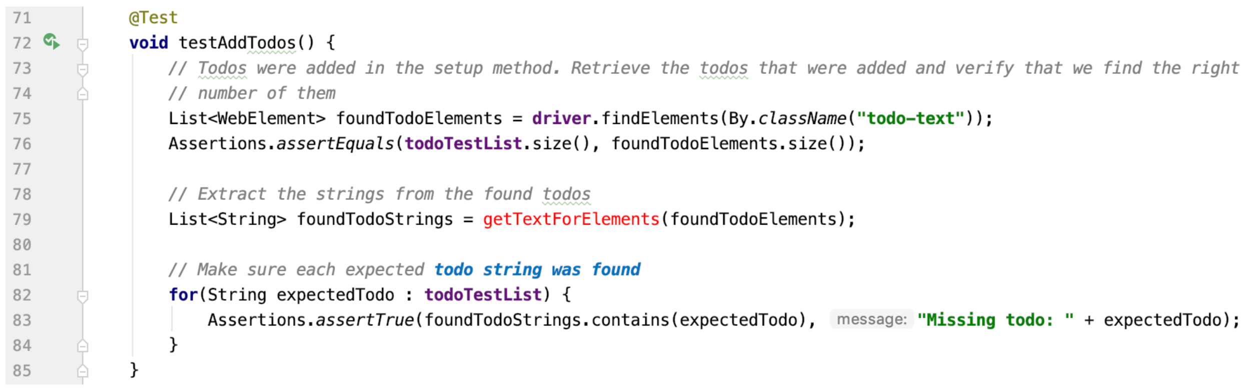
7. It's nice that the tests will clean up after themselves now, but that happened so fast, we couldn't really see what the browser window looked like at the end of the test. Normally this is what you want. In a production testing environment, these tests would all be executed as part of an automated build process with no one watching. We may have thousands of tests, and we would want them all to run as quickly as possible. We would rely on assertions in the tests telling us if they passed instead of relying on manual inspection of the pages. However, while you're learning Selenium in this tutorial, it will be useful to be able to see what the pages look like at the end of each test before the browser windows are closed. We'll accomplish that by adding a 3-second pause at the beginning of the teardown method as follows:



```
@AfterEach
void teardown() {
    try {
        Thread.sleep( millis: 3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    driver.quit();
    server.stop();
}
```

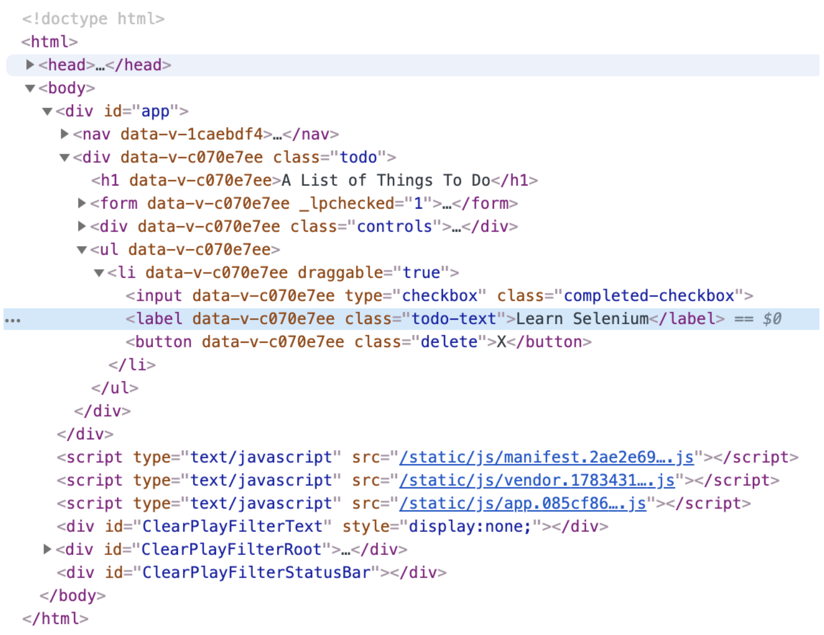
8. Now run the test again. It will still clean up after itself, but it will now give you time to see the end result of the test before closing the browser window.
9. It's nice to see that Selenium works, but so far, our test doesn't actually test any behavior. The `testAddTodos()` test should programmatically confirm that the

elements the setup method added are actually on the page. Do that by first removing the "TODO" comment in the `testAddTodos()` method, then insert the following code:



```
71      @Test
72      void testAddTodos() {
73          // Todos were added in the setup method. Retrieve the todos that were added and verify that we find the right
74          // number of them
75          List<WebElement> foundTodoElements = driver.findElements(By.className("todo-text"));
76          Assertions.assertEquals(todoTestList.size(), foundTodoElements.size());
77
78          // Extract the strings from the found todos
79          List<String> foundTodoStrings = getTextForElements(foundTodoElements);
80
81          // Make sure each expected todo string was found
82          for(String expectedTodo : todoTestList) {
83              Assertions.assertTrue(foundTodoStrings.contains(expectedTodo), message: "Missing todo: " + expectedTodo);
84          }
85      }
```

Line 75 uses the WebDriver to inspect the page and find all elements matching the parameter of the `findElements(...)` method call. This is called a selector. When we want to receive a list of matching elements we call `findElements(...)`. When we want to receive only a single element, we call `findElement(...)`.
The `findElement(...)` and `findElements(...)` methods take a parameter we generate by calling static methods of the `By` class. There are several options. For these elements, we used a selector of "class name", where the class name is equal to "todo-text". This returns a list of all HTML elements on the page that have a class of "todo-text".
As shown in the following screenshot of the HTML of the web application with a single todo added, the label elements of all todos are created with a class of "todo-text" so this selector will return the label elements for all todos displayed on the page.

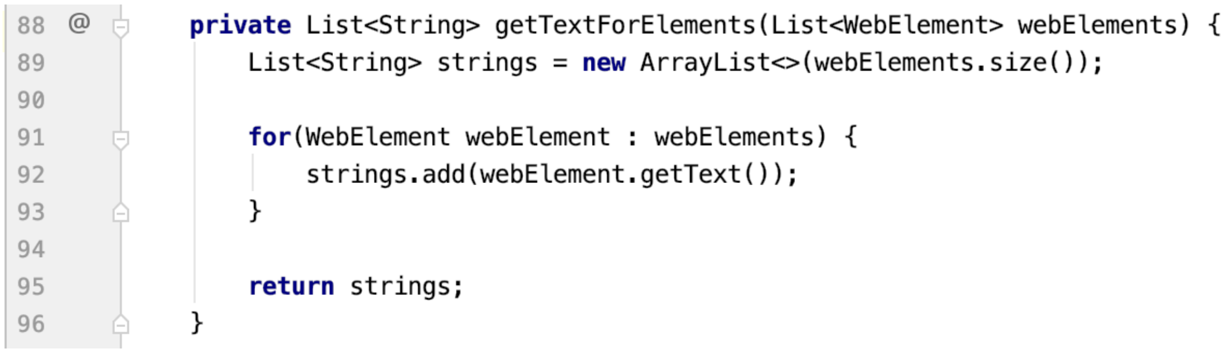


```
<!doctype html>
<html>
▶<head>…</head>
▼<body>
  ▼<div id="app">
    ▶<nav data-v-1caebdf4>…</nav>
    ▼<div data-v-c070e7ee class="todo">
        <h1 data-v-c070e7ee>A List of Things To Do</h1>
      ▶<form data-v-c070e7ee _lpchecked="1">…</form>
      ▶<div data-v-c070e7ee class="controls">…</div>
      ▼<ul data-v-c070e7ee>
        ▼<li data-v-c070e7ee draggable="true">
            <input data-v-c070e7ee type="checkbox" class="completed-checkbox">
...         <label data-v-c070e7ee class="todo-text">Learn Selenium</label> == $0
            <button data-v-c070e7ee class="delete">X</button>
          </li>
        </ul>
      </div>
    </div>
    <script type="text/javascript" src="/static/js/manifest.2ae2e69….js"></script>
    <script type="text/javascript" src="/static/js/vendor.1783431….js"></script>
    <script type="text/javascript" src="/static/js/app.085cf86….js"></script>
    <div id="ClearPlayFilterText" style="display:none;"></div>
  ▶<div id="ClearPlayFilterRoot">…</div>
    <div id="ClearPlayFilterStatusBar"></div>
  </body>
</html>
```

Line 76 asserts that we found the same number of elements as the number that should have been added by the `setup()` method.

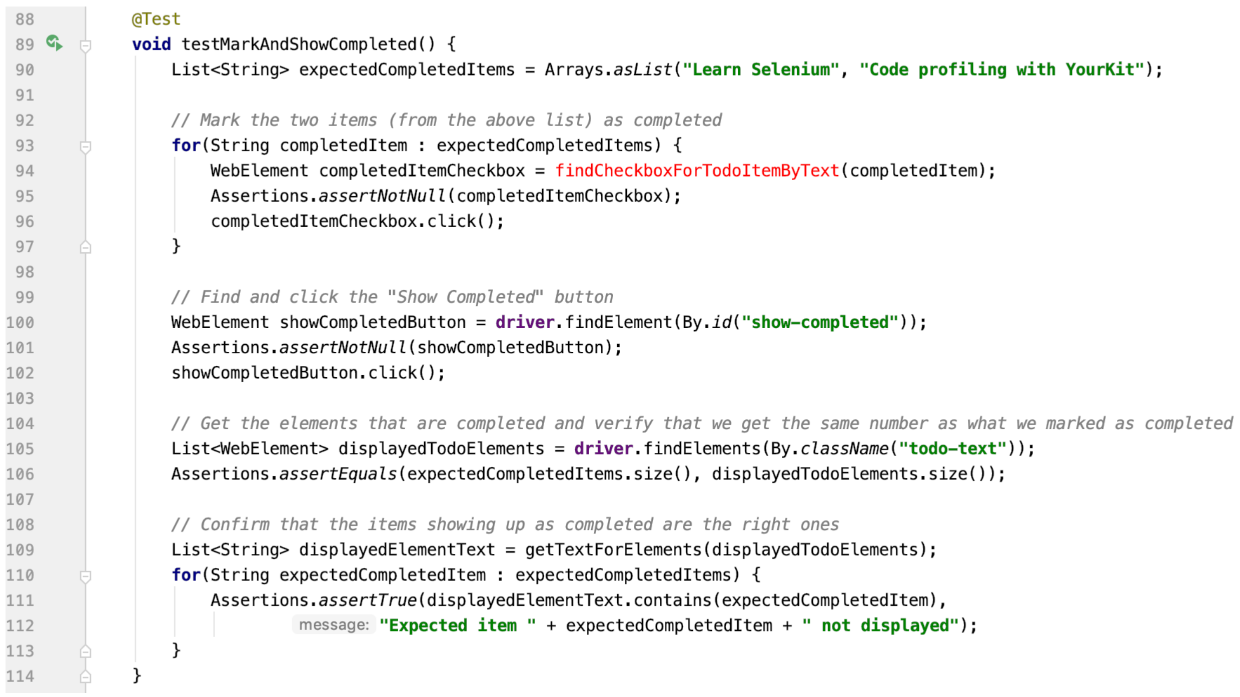
Now that we know the right number of elements that were added, we want to confirm that the text displayed for the elements matches the strings we intended to add. Line 79 extracts the text from each of the web elements we retrieved into a list of strings we can use to confirm that all expected strings were found. Multiple tests will need this functionality, so we are handling the extraction of the text from the strings in a separate method--which we will write in the next step. Once we have the strings, lines 82 - 84 iterate the list of todos our `setup()` method should have added and assert that each string was found in the Web Elements extracted from the page. Since we already confirmed the count, we will know we have exactly the right todos on the page if all strings from the expected list of strings exist in the list extracted from the retrieved web elements.

10. Now we'll write the method that was called on line 79 above. This is a simple helper method that will be used by multiple tests. It receives a list of web elements and iterates the list. For each web element in the list, it extracts the text of the element by calling `getText()` and adds the extracted text to a list of strings. Write the following method at the end of your `TestServer` class:



```java
    private List<String> getTextForElements(List<WebElement> webElements) {
        List<String> strings = new ArrayList<>(webElements.size());

        for(WebElement webElement : webElements) {
            strings.add(webElement.getText());
        }

        return strings;
    }
```

11. Now run the test. It should pass, confirming that the web application successfully adds and displays todos.
12. Now we want to test that we can mark some todos as completed, then press the "Show Completed" button and have only the completed todos displayed on the page. Start by writing the following `testMarkAndShowCompleted()` test method between the `testAddTodos()` and `getTextForElements(...)` methods:



```
88       @Test
89       void testMarkAndShowCompleted() {
90           List<String> expectedCompletedItems = Arrays.asList("Learn Selenium", "Code profiling with YourKit");
91
92           // Mark the two items (from the above list) as completed
93           for(String completedItem : expectedCompletedItems) {
94               WebElement completedItemCheckbox = findCheckboxForTodoItemByText(completedItem);
95               Assertions.assertNotNull(completedItemCheckbox);
96               completedItemCheckbox.click();
97           }
98
99           // Find and click the "Show Completed" button
100          WebElement showCompletedButton = driver.findElement(By.id("show-completed"));
101          Assertions.assertNotNull(showCompletedButton);
102          showCompletedButton.click();
103
104          // Get the elements that are completed and verify that we get the same number as what we marked as completed
105          List<WebElement> displayedTodoElements = driver.findElements(By.className("todo-text"));
106          Assertions.assertEquals(expectedCompletedItems.size(), displayedTodoElements.size());
107
108          // Confirm that the items showing up as completed are the right ones
109          List<String> displayedElementText = getTextForElements(displayedTodoElements);
110          for(String expectedCompletedItem : expectedCompletedItems) {
111              Assertions.assertTrue(displayedElementText.contains(expectedCompletedItem),
112                       message: "Expected item " + expectedCompletedItem + " not displayed");
113          }
114      }
```

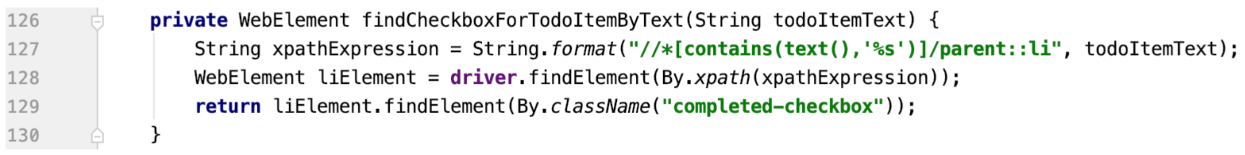
This method starts by creating a list of strings of the todos we want to mark as completed. Then lines 93 - 97 iterates this list of strings, finds the checkbox web element associated with each todo from the list, and clicks it. This marks each of the desired elements as being completed. Finding the checkbox associated with a todo is more complicated than other selectors you have seen so far, and this is another task that will need to be done from multiple tests, so this is handled by a `findCheckboxForTodoItemByText(...)` method you will write in the next step.

Once the desired items are checked (marked as completed), we need to find and click the “Show Completed” button so the web app will update the page to show only the completed todos. This is done by lines 100 - 102. Line 100 finds the button by using an id selector. Each button element on the page has a unique id attribute, making it easy to select the desired button. If this fails for some reason, we don’t want to try to click a null button so we assert on line 101 that the button is not null before clicking it on line 102.

Finally, lines 105 - 113 confirm that the page is displaying the same number of todos as our test marked as completed and that the text of the items displayed matches the text of the items we intended to mark as completed. This code is almost identical to the code in the `testAddTodos()` test.

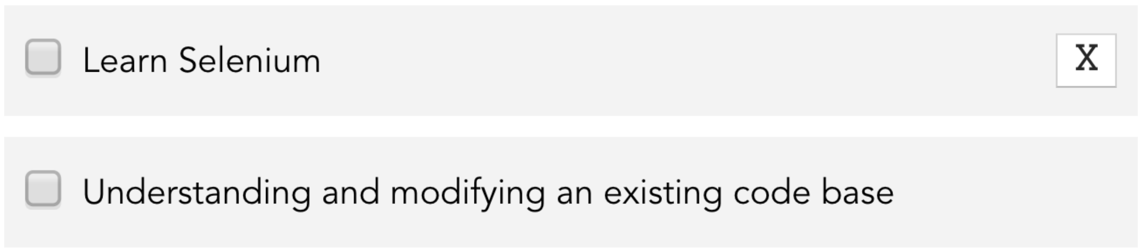
**Note:** If this were a real software engineering project, we would want to factor as much of the code into callable methods as possible to eliminate duplicate code between the tests. However, that would have the effect of obscuring the code somewhat, so we have chosen not to do that in some cases in this tutorial to keep the tests as understandable as possible.

13. Now we need to write the `findCheckboxForTodoItemByText(...)` method called by the test we wrote in the previous step. This method uses an Xpath selector and other logic to find the desired checkbox. Write the method first (at the bottom of your `TestServer` class) and then review the following explanation and make sure you understand what it's doing.

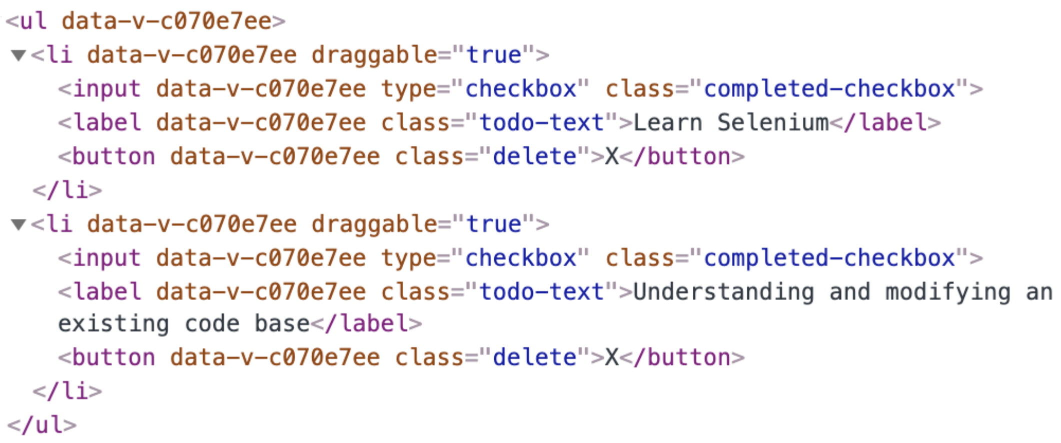


```java
private WebElement findCheckboxForTodoItemByText(String todoItemText) {
    String xpathExpression = String.format("//*[contains(text(),'%s')]/parent::li", todoItemText);
    WebElement liElement = driver.findElement(By.xpath(xpathExpression));
    return liElement.findElement(By.className("completed-checkbox"));
}
```

Before explaining the code, we need to understand the html used to represent todo items on the web page. The following image shows two todo items as they appear on the page.



Notice the X button on the right side of the "Learn Selenium" todo. This button is invisible until the mouse hovers over a todo item. This is the html for the above todo items:



```html
<ul data-v-c070e7ee>
 ▼<li data-v-c070e7ee draggable="true">
    <input data-v-c070e7ee type="checkbox" class="completed-checkbox">
    <label data-v-c070e7ee class="todo-text">Learn Selenium</label>
    <button data-v-c070e7ee class="delete">X</button>
  </li>
 ▼<li data-v-c070e7ee draggable="true">
    <input data-v-c070e7ee type="checkbox" class="completed-checkbox">
    <label data-v-c070e7ee class="todo-text">Understanding and modifying an
    existing code base</label>
    <button data-v-c070e7ee class="delete">X</button>
  </li>
</ul>
```

We can see from this that todo items are represented as an unordered list <ul> of list items `<li>`, where each `<li>` consists of three elements: an input checkbox, a label, and a button (which is invisible unless we hover over it's `<li>` element).

The `findCheckboxForTodoItemByText(...)` method needs to find the checkbox in the same `<li>` element as the label whose text matches the text specified as a parameter to the method. We start by finding the `<li>` element containing a label whose text matches the method parameter. This is easy if you know about XPath expressions. Xpath expressions are similar to regular expressions. They provide a concise syntax for specifying matching criteria for the element(s) we want to retrieve. Any elements matching the specified criteria are returned.

Line 127 specifies the XPath expression we need to retrieve the correct `<li>` element. Xpath expressions can be complex and it is not our intent to teach everything about them in this lab, but we will give an overview of the XPath expression used in this method. Here is the code on line 127 of the method:

```
String xpathExpression = String.format("//*[contains(text(),'%s')]/parent::li", todoItemText);
```

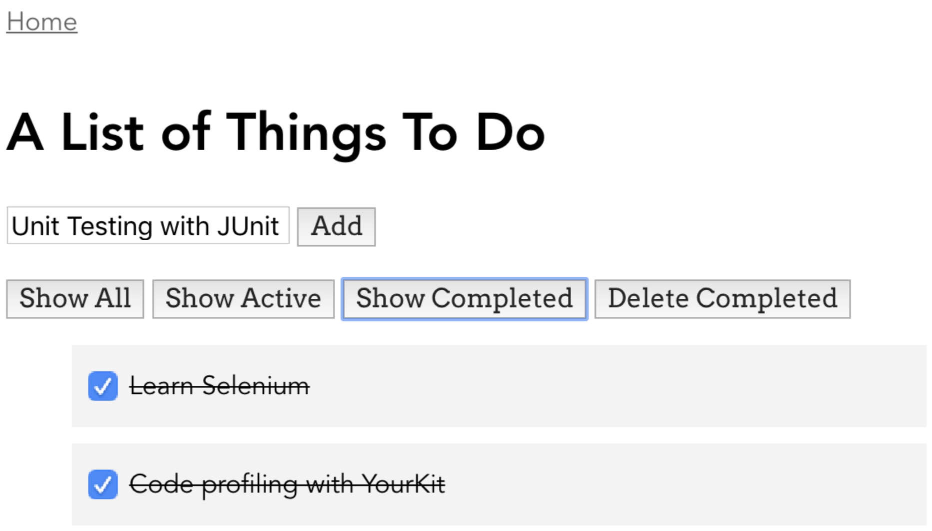
The first thing to notice is that we're generating the string by calling `String.format(...)`. This provides a convenient way to generate a string by using parameter substitution instead of string concatenation, which makes the string easier to read in the code. The `%s` in the middle of the string gets replaced by the `todoItemText` parameter, which is the parameter passed into the method. If we call this method with a parameter of "Learn Selenium", the `String.format(...)` method generates the following xpathExpression string:

```
//*[contains(text(),'Learn Selenium')]/parent::li
```

We'll use this string to describe what the expression does. Reading it from left to write, the part up to but not including **/parent** says, find all the elements containing the text "Learn Selenium". Then the **/parent::li** part says, for each element found by the first part of the expression, find it's closest ancestor element that is an `<li>` element.

Line 128 executes this XPath expression and returns the `<li>` element containing a child with the text "Learn Selenium". Now we have the `<li>` element we need. All we have to do now is get that element's checkbox child element. Line 129 does that with a class name selector of "completed-checkbox". This is the element this method returns, which gets clicked by the test we are now ready to run.

14. Run the `testMarkAndShowCompleted()` method you wrote in the previous two steps. You should see the following for a few seconds before the `teardown()` method removes the browser pages it creates:


Home

A List of Things To Do

Unit Testing with JUnit

~~Learn Selenium~~

~~Code profiling with YourKit~~

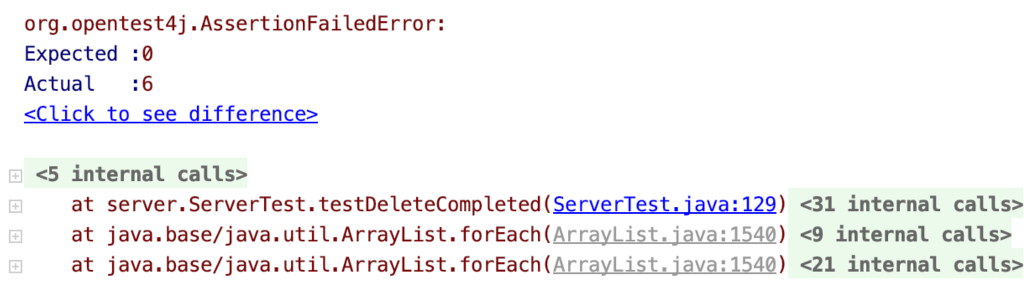
17. We still have three buttons we haven't tested: "Show All", "Show Active" and "Delete Completed". The code would be very similar to the `testMarkAndShowCompleted()` test. We will have you write a test for "Delete Completed" mostly on your own as a mini-lab before submitting your code.

## Mini-Lab: Writing a "Delete Completed" Test

Now that we have walked you through writing two Selenium tests, we will have you write another one mostly on your own. We will explain what you need to do, but we will not provide code for this test (although most of it can be found in the tests you've already written).

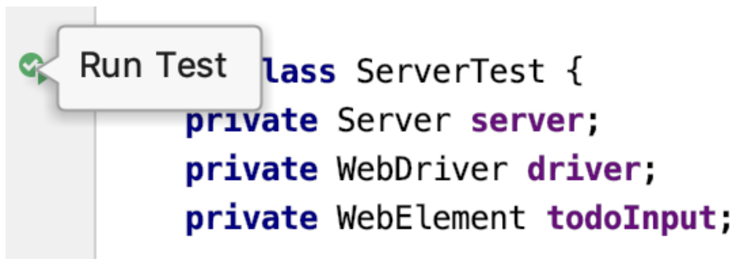
Start by writing a `testDeleteCompleted()` method and annotating it as a test. Place it after your other two tests but before the helper methods at the end of your test class. Now write your test code by following these steps.

1. Write code at the top of your new test method to mark all todos as completed.
2. Write code to retrieve the todo items from the page and write an assertion that asserts that 0 items were retrieved.
3. Run your test. You should see a browser window appear and you should see all six todos added to the page and then marked as completed, but the test should fail because we haven't deleted the completed items yet. You should see the following message in your test result output:


```
org.opentest4j.AssertionFailedError: 
Expected :0
Actual   :6
<Click to see difference>

<5 internal calls>
   at server.ServerTest.testDeleteCompleted(ServerTest.java:129) <31 internal calls>
   at java.base/java.util.ArrayList.forEach(ArrayList.java:1540) <9 internal calls>
   at java.base/java.util.ArrayList.forEach(ArrayList.java:1540) <21 internal calls>
```

Now let's think about why we wanted to see this test fail before clicking the "Delete Completed" button. It was important because there are more reasons than one that we could have our assertion of finding 0 todos pass. It could pass because there really are no todos on the page after the button is clicked. This is the reason we want it to pass. However, if we have a typo in the selector we use to select the items remaining on the page before the assertion, that would also cause our assertion to pass, even though there could still be todos on the page. To ensure that this doesn't happen and that our selector really is selecting the todos that are on the page if there are any, we want to see our test fail when we haven't clicked the button and then start passing after we write the code that does click the button. Then we will know it's passing because the test and the behavior we're testing is correct.

4. Now that you've seen the test fail, go ahead and write the code to select and click the "Delete Completed" button. This should be inserted right after the code that marked all of the todos as completed. To find the button, you will need to select it by id. To find its id, you will need to start your `Server` class manually, open a web browser and navigate to the web app's page, and then inspect the "Delete Completed" button (like you did in step 1 of the tutorial).
5. Run your test and make sure it passes.
6. Now run all of your tests in the same test run by clicking the run icon next to the `ServerTest` class name as shown below:



```
class ServerTest {
    private Server server;
    private WebDriver driver;
    private WebElement todoInput;
```

You should see a browser window appear for the first test and then disappear after a few seconds, followed by another browser window appearing and disappearing after the second test. This will occur for all three of your tests. The test results should show that all tests passed.

## Submission

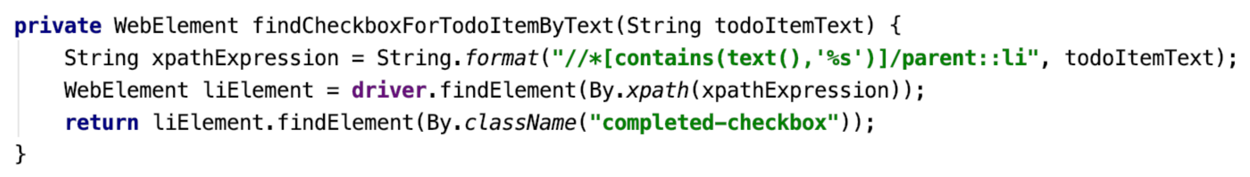
To receive credit for this tutorial, take a screenshot of your test results window showing all tests passing. Submit the screenshot and your `ServerTest.java` file to Canvas.

## Bonus Material

This section will show you how to test the “X” delete button that only appears when you hover over a todo item. This section is for the curious who want to learn more about Selenium. It is not a graded part of the assignment, so doing it is completely optional.

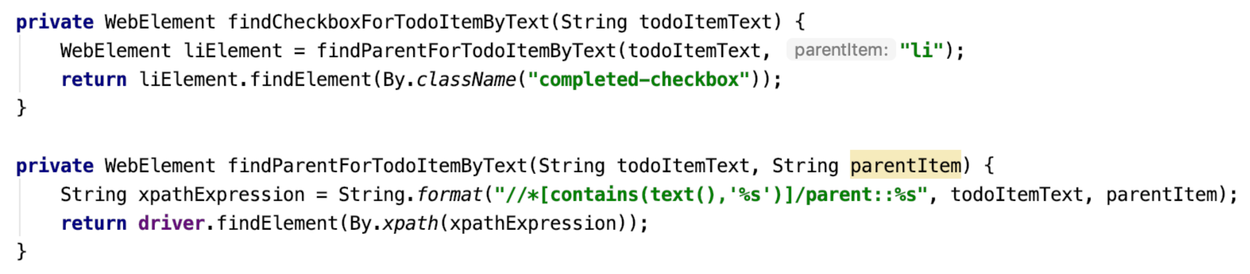
At first glance, you might think you can just get a reference to the “X” button for the item you want to delete in the same way you got the checkbox for the items you wanted to check in the `testMarkAndShowCompleted()` test. However, there is an important difference that does not allow this solution to work. The “X” button is not visible until the user hovers over the corresponding `<li>` tag with the mouse. As far as Selenium is concerned, this means the button is not clickable.

We have to write a test that simulates the user hovering over (or clicking) the `<li>` element of the todo they want to delete and then finding and clicking the corresponding “X” button. This all has to happen as part of the same user action. If you look at the code for the `findCheckboxForTodoItemByText(...)` helper method you will see that the first two lines of code found the `<li>` item we need for our new test, but the next line returns the checkbox child, not the `<li>` item:



```java
private WebElement findCheckboxForTodoItemByText(String todoItemText) {
    String xpathExpression = String.format("//*[contains(text(),'%s')]/parent::li", todoItemText);
    WebElement liElement = driver.findElement(By.xpath(xpathExpression));
    return liElement.findElement(By.className("completed-checkbox"));
}
```

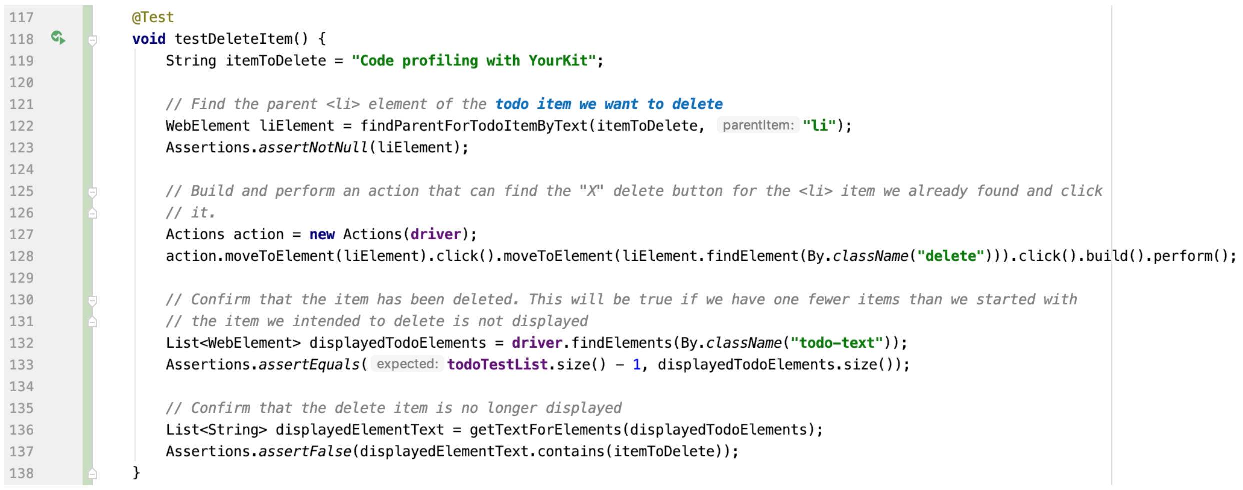
A little refactoring will keep this method working while extracting those first lines of code into a separate method we can use to find and return the `<li>` tag we need:



```java
private WebElement findCheckboxForTodoItemByText(String todoItemText) {
    WebElement liElement = findParentForTodoItemByText(todoItemText,  parentItem: "li");
    return liElement.findElement(By.className("completed-checkbox"));
}

private WebElement findParentForTodoItemByText(String todoItemText, String parentItem) {
    String xpathExpression = String.format("//*[contains(text(),'%s')]/parent::%s", todoItemText, parentItem);
    return driver.findElement(By.xpath(xpathExpression));
}
```

Notice that `findCheckboxForTodoItemByText(...)` now calls a new `findParentForTodoItemByText(...)` method that we can call to find any parent for any todo item. We can use this new method in our new `testDeleteItem()` test to find the `<li>` element we need.

As mentioned above, this new test will need to find the desired `<li>` item, click it, and then find it's child "X" button and click it, all in one action. This can be done by creating a Selenium `Actions` object. Within an `Actions` object, we can specify a chain of actions to perform. The chain ends with a call to `build()` to generate the action, followed by a call to `perform()` to cause the whole thing to happen as one action. Now write the new test as follows:



```
117    @Test
118    void testDeleteItem() {
119        String itemToDelete = "Code profiling with YourKit";
120
121        // Find the parent <li> element of the todo item we want to delete
122        WebElement liElement = findParentForTodoItemByText(itemToDelete,  parentItem: "li");
123        Assertions.assertNotNull(liElement);
124
125        // Build and perform an action that can find the "X" delete button for the <li> item we already found and click
126        // it.
127        Actions action = new Actions(driver);
128        action.moveToElement(liElement).click().moveToElement(liElement.findElement(By.className("delete"))).click().build().perform();
129
130        // Confirm that the item has been deleted. This will be true if we have one fewer items than we started with
131        // the item we intended to delete is not displayed
132        List<WebElement> displayedTodoElements = driver.findElements(By.className("todo-text"));
133        Assertions.assertEquals( expected: todoTestList.size() - 1, displayedTodoElements.size());
134
135        // Confirm that the delete item is no longer displayed
136        List<String> displayedElementText = getTextForElements(displayedTodoElements);
137        Assertions.assertFalse(displayedElementText.contains(itemToDelete));
138    }
```

Lines 127 and 128 create and perform the action we need. Then lines 132 - 137 confirm that the delete actually worked.