[Print](javascript:window.print())

**Course Transcript**

Selenium and Java

**Creating Projects and Locating Elements**

[1. Course Introduction](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t2)

[2. Selenium with Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t6)

[3. Exact Elements in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t10)

[4. Many Elements in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t14)

**Popups and Multiple Windows**

[1. Popups in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t19)

[2. Multiple Windows in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t23)

**Drag and Drop and Input Data**

[1. Drag and Drop in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t28)

[2. Input Data in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t32)

**Links and Navigation**

[1. Clicking Links in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t37)

[2. Navigation in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t41)

**Display Elements and Appearance**

[1. Display Items in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t46)

[2. Screenshots in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t50)

**JavaScript, Ajax and Cookies**

[1. JavaScript from Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t55)

[2. Ajax elements in Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t59)

[3. Cookies and Java Tests](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t63)

**Practice: Using Selenium and Java**

[1. Exercise: Use Selenium with Java](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#t68)

Course Introduction

Learning Objective

*After completing this topic, you should be able to*

* *start the course*

**1. Introduction to the course**

Selenium provides an easy to use, visual based, test creation tool through the IDE. But the real power of the framework is exposed by diving into a coding language. I'm Tony Lowe, a Java architect with over 15 years experience building and teaching IT solutions. If your preference is Java, Selenium provides a framework implementation that allows many of the same features as the IDE but with all the power of Java to back up your testing. You can drive a browser as if you were a live user but also reach into your system to grab resources as only a coded solution can. In this course, we'll see all of this in action.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Selenium with Java

Learning Objective

*After completing this topic, you should be able to*

* *create a Selenium Java project*

**1. Using Java and Selenium**

If your language of choice is Java, then you can easily create a Selenium project with the help of the Maven Framework. Here is an example of getting started using Eclipse and Maven. So we're starting in a brand new base Eclipse. There is only really one special thing that's being thrown on inside of here and that's the Maven Framework, because the plug-in has been added for the Maven Framework. And so when we go up to **File** and we say **New** and in **Project**, we can come down to the Maven options inside of here and be able to create a Maven project. So from the Maven project, there is not a lot of setup we have to do. We're going to actually create a simple project. We're not going to pick any archetype. We're going to just use the default workspace inside of there. And so our project name inside of here can be anything, the Group Id and the Artifact Id are basically ways for you to identify your piece of code. And for our purposes, we can just say myFirstProject and this could be named anything. We'll just call it myFirstProject again. They're identifiers – they're easy enough to change later on if you so choose.   
*In Eclipse, the presenter selects File - New - Project. The New Project Wizard includes a Wizards pane with a Maven node. The expanded node includes three subnodes – Maven Projects from SCM, Maven Module, and Maven Project. The presenter selects Maven Project and clicks Next.   
  
The New Maven project page of the wizard contains three checkboxes – Create a simple project (skip archetype selection), Use default Workspace location, and Add project(s) to working set. It also contains an Advanced option. The presenter selects Create a simple project (skip archetype select) and clicks Next.  
  
The next page of the wizard contains Artifact and Parent Project sections. The Artifact section contains Group Id, Artifact Id, Version, Packaging, Name, and Description options. The Parent Project section contains Group Id, Artifact Id, and Version options.  
  
The presenter enters myFirstProject in the Group Id and Artifact Id text boxes. He then clicks Finish.*  
  
So now our project is set up and you can see we have a general Maven setup. We have the Java code and the Java test codes inside of here and then we have our pom file. Now, where Maven is going to really help us quite a bit inside of here is managing dependencies. So we need a dependency inside of here to go to the Selenium framework. So the Selenium framework is identified by two key pieces, the Group Id being the package name that they have chosen inside of there. And then the Artifact Id being selenium-java version. And then the last key to getting this is your version number and you can use whatever the most recent version you want to choose inside of there. And so I'm using 2.41.0. So that's the first one we're going to put inside of here. We're going to add one more dependency just to make our lives easier. We're going to add in the JUnit and you can see for JUnit they are really simple. Their Group Id and Artifact Id are both JUnit, they are not getting too fancy inside of there. And then we're choosing 4.8.1 as a version and again almost any version will work inside of here. There is no strict dependency between versions of JUnit and Selenium for the most part here. So there are our two dependencies and that's really all we have to do. Let's say your choice is not Maven. Let's say you don't want to deal with Maven, then you just have to go off and find the appropriate jar file. And you have to find the appropriate JUnit file and Selenium jar file and that will download everything you want to. But by doing Selenium, this will download…I'm sorry... by using Maven, this will download both Selenium and JUnit and any dependencies that are needed along the way as well. So when I save that and I close that, then our project will be updated.   
*In Eclipse, a myFirstProject node is listed in the Package Explorer. The presenter expands the node, which includes the components are src/main/java, src/main/resources, src/test/java, src/test/resources, JRE System Library, src, target, and pom.xml. The presenter double-clicks pom.xml and the file opens in the code window. The heading for the page is Overview. It includes the sections Artifact, Parent, Properties, Modules, Project, Organization, SCM, Issue Management, and Continuous Integration.  
  
Five tabs are included at the bottom of the code window. These are Overview, Dependencies, Dependency Hierarchy, Effective POM, and pom.xml. The presenter clicks the Dependencies tab. This opens a page with a Dependencies pane, the buttons Add, Remove, Properties, and Manage, and a Dependency Management pane. The presenter clicks the Add button.  
  
The Select Dependency dialog box contains Group Id, Artifact Id, Version, and Enter groupId, artifactId, or sha1 prefix or pattern (\*) text boxes. The presenter enters org.seleniumhq.selenium in the Group Id text box, selenium-java in the Artifact Id text box, and 2.41.0 in the Version text box. The presenter closes the Select Dependency dialog box.  
  
The presenter again clicks Add and a new Select Dependency dialog box opens. The presenter enters Junit in the Group Id and Artifact Id text boxes, and 4.8.1 in the Version text box. He then closes the dialog box. Both dependencies are listed in the Dependencies pane. The presenter closes the Dependencies page.*   
  
And so if we look in the Build Path, we have the Maven dependencies and the things that are inside of there. And so it's actually got quite a bit of Selenium stuff if we look out of here, it's downloaded several files that it needs from Selenium. So it's not as simple as getting a single jar file. JUnit is pretty straightforward – there is a minimal number of jar files but Selenium has quite a few. And that's what makes Selenium so much easier to use within Maven as opposed to on its own, not impossible just easier. So let's get to our first test. So we create a new class inside of here and this can be our first test class. We don't need anything fancy, just a basic class and we can even start with a basic main program. Now, to jumpstart this, I'm going to take code that I've created ahead of time and we can investigate this code at some other time. But I'm going to throw this code out here. So just quickly, I'm going to fix my imports here and I've got everything imported. The driver there is what opens up Selenium. It gets it working and we're choosing to use Firefox as a driver. Then we have our test web site we're going to… We're going to find a text box on that web site and type in the username. We're going to find another text box, the Password text box inside and type a password. We are going to find the buttons for submitting and then we're going to click that button. And that's all the tests we're going to do here. But just, you can see on these quick steps, I have a test ready to go. And I can run this test and here it comes. I've got to give it a second.   
*The presenter accesses a dialog box and, in a pane on the left, selects Java Build Path. The Libraries tabbed page contains the options JRE System Library [J2SE-1.5] and Maven Dependencies. The presenter selects Maven Dependencies. A list of JAR files opens. The presenter scrolls through the list of files. The presenter then minimizes the list of files and closes the window.  
  
The presenter right-clicks src/test/java in the Package Explorer and selects New - Class. The Class dialog box contains options for configuring a class, such as Source folder, Package and Name text boxes, Modifiers options, and an Interfaces pane with Add and Remove buttons. The presenter enters FirstTest in the Name text box and returns to the code window. The following code is displayed:  
  
package myFirstProjectl  
  
public class FirstTest  
{  
  
}  
  
The presenter adds to the code, which in full is now  
  
package myFirstProjectl  
  
public class FirstTest  
{  
       public static void main(String[] args)  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/signon.vpd");  
  
               WebElement element = driver.findElement(By.id ("usernameInput"));  
               element.sendKeys("admin");  
  
               element = driver.findElement(By.id("passwordInput")):  
               element.sendKeys("test");  
  
               element = driver.findElement(By.name(submit"));  
               element.click();  
  
               Sytem.out.print1n("Test Completed");  
  
       }  
  
}  
  
The presenter changes the first line of code to import org.openqa.selenium.firefox.FirefoxDriver;.  
  
The presenter clicks Run.*  
  
And there is the test. It opened up Firefox. It ran through the whole test. And it, you know, put in the username and password – I don't know if you saw it real quick there. And then it went over and logged in for us. So we can close that and that's one key you need to close that Firefox window in between executions. And just to show you real quick, we can make this just as easily a JUnit test by making it the @Test symbol. And so we switched the main program in there to just a straight JUnit test and the @Test symbol makes that ready to go. And now when we run it, it's going to run it as a JUnit instead. So you can see here a JUnit window pops up. It's running the test and poof, there it goes. It brought up the browser, it ran the test, it put everything inside of there and our test passed. So we could add other JUnit things inside of the thing here and make it all work. So that's a quick look at getting started within Selenium and Java. Maven really simplifies the process, but again you can manage it all by yourself. There are not a lot of steps, just a few simple steps and you're ready to go.   
*A series of web pages opens and the steps in the test are performed. The test ends on a web page titled Administrator Home.  
  
The presenter closes the browser and returns to Eclipse.  
  
The presenter modifies the code, which is now  
  
public class FirstTest  
{  
       @Test  
       public void firstTest()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/signon.vpd");  
  
               WebElement element = driver.findElement(By.id ("usernameInput"));  
               element.sendKeys("admin");  
  
               element = driver.findElement(By.id("passwordInput")):  
               element.sendKeys("test");  
  
               element = driver.findElement(By.name(submit"));  
               element.click();  
  
               Sytem.out.print1n("Test Completed");  
  
       }  
  
The presenter clicks Run and the test runs again. It ends on the same Administrator Home web page, but the page no longer contains a set of navigational links on the left. The presenter closes the browser and returns to Eclipse.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Exact Elements in Java

Learning Objective

*After completing this topic, you should be able to*

* *find exact elements in Selenium Java projects*

**1. Finding individual elements**

One key in order to test a web page, is to locate the exact item on the page. Whether you're filling in a text box, pressing a button, or checking a text entry, you must find it in order to interact with it. Finding an exact element has many options; so let's take a look. So, for this test, we're going to start off with this real simple web page. This page has some elements on it. This one's actually in a table; we have some links, some page elements inside of here, a text box, just some random things we can search for – and they have many different items in there, and many similarities. Let's look at how we're going to do this. So, what we're going to look at here is the findElement() from the driver here. So we get our driver, and then we can have a findElement() call. And the way this call works, is it's going to be handed a helper here. And the helper's from the By class; there's going to be a lot of options – we can look down here. But the key of the findElement() is…we trust that this element must be out there. If it's not out there, this is actually going to give us some issues. So, the first one we'll look at is By.id; this is probably the simplest one.   
*The file FindingExactElements.java is open in Eclipse. The Junit tab is open in the Explorer pane on the left. On the right the file FindingExactElements.java is open in the code window. It contains the following code:  
  
package find;  
  
import org.junit.FixMethodOrder;  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class FindingExactElements extends BaseSearch  
{  
       @Test  
       public void a\_byID()  
       {  
               System.out.println("///////////////////////////////////////////////////////////////");  
               System.out.println("By ID\n");  
                 
               WebElement element = driver.findElement(By.id("greetings"));  
               System.out.println("Direct to ID " + element);  
               System.out.println(element.getText());  
       }  
  
       @Test  
       public void b\_byName()  
       {  
  
  
In Eclipse, the Console pane contains the following:  
  
//////////////////////////////////////////////////////////////  
By ID  
  
Direct to ID <p class="opening" id="greetings">  
Hello  
//////////////////////////////////////////////////////////////  
By Name  
  
Direct to name <input type="text" name="inputBoxTofind" value="Default input value" />  
  
//////////////////////////////////////////////////////////////  
  
The presenter switches to a browser window. A web page titled Searches is open. It contains a placeholder title called Centered Title, the links Question, Greeting, Test Link, and Test Multiple Links, a section titled Hellow with the text "I am here to say Hello", and a section titled Goodbye with the text "It is time to go" and a text box that currently contains the text "Default input value." It also contains a section titled Are you sure?, which contains the text "That was a question?"  
  
The presenter returns to the code in Eclipse.*   
  
If we go over to our page and look at our page source, we're looking for any of these tags in here that have an ID. In this case, you can see this has an ID of "greetings". And so, it's going to be very easy to locate this because, ideally, it's the only one – if the design is done well – that has this ID of "greetings". And we're going to go look for this ID element. We can see when we **Run** this, we go straight to that element and it returns to us the whole tag. So, the WebElement returns us the details of the whole tag. As we can see, this is the whole tag that gets returned to us and this what we see in code. And we can ask a lot of stuff from that element, but we're going to look for the text, and you can see the text inside of there says Hello, which is exactly what shows up on our web page. So, the first element is just a search for a page element. Now, we could've searched for anything that has an ID; it could be a div tag, it could be a text box – anything out there that has an ID, we can search by that. So, let's say we are looking for an input box specifically, and that input box doesn't have an ID, but it has a name. Easy enough; we put the By.name option out here.   
*The presenter switches back to the browser window and opens a second tab, which contains the source code. The code that's visible is  
  
<style>  
<!--  
       .opening  
       {  
               font-weight: bold;  
       }  
-->  
</style>  
  
<h1>Searches</h1>  
       <br/>  
       <br/>  
  
<table style="width :100%">  
       <tr>  
               <th align="center">  
                       <p>  
                               Centered Title  
                       </p>  
               </th>  
       </tr>  
</table>  
  
<ul>  
       <li> <a href ="">Question</a></li>  
       <li> <a href ="">Greeting</a></li>  
  
The presenter scrolls down. Additional code is visible, as follows:  
  
       <li> <a href="">Test Link</a></li>  
       <li> <a href="">Test Multiple Links</a></li>  
</ul>  
  
<p class="opening" id="greetings">  
Hello  
</p>  
  
<p = class="narrative">  
I am here to say Hello  
</p>  
  
<div>  
  
The presenter highlights id="greetings".  
  
The presenter switches back to the code in Eclipse and highlights the code <p class="opening" id="greetings"> in the Console pane. He then highlights By.id in the code window.  
  
The presenter switches back to the source code in the browser window. The presenter highlights the code  
  
<p class="opening" id="greetings">  
Hello  
  
The presenter briefly returns to Eclipse. He then switches back to the browser window and opens the first tab, containing the web page. He highlights the text Hello.  
  
The presenter switches back to Eclipse and scrolls through the code. The following code is visible:  
  
System.out.println(element.getText());  
       }  
  
       @Test  
       public void b\_byName()  
       {  
               System.out.println("///////////////////////////////////////////////////////////////");  
               System.out.println("By Name\n");  
                 
               WebElement element = driver.findElement(By.name("inputBoxTofind"));  
               System.out.println("Direct to name " + element);  
               System.out.println(element.getAttribute("value"));  
       }  
  
       @Test  
       public void c\_byLinkName()  
       {  
               System.out.println("///////////////////////////////////////////////////////////////");  
               System.out.println("By Link Name\n");  
                 
               WebElement element = driver.findElement(By.linkText("Question"));   
  
The presenter highlights the code (By.name("inputBoxTofind");.*   
  
And so the By.name option – we take the name of the text box. Again, if we go to the page source inside of here and we look at our input tag, our input tag has a name of the text we're going to find. And so, this text is going to be sent back to the web site. And so, it's a pretty good reliable indicator of what the field's going to be called, so it's not a bad thing for us to search on for our test case. And so, when we get that, again, we get the element – we get the whole text box inside of there. Now, there's no text; there's no text that comes back, because the input element didn't have text associated with it. So, there's other calls we can make to be able to look at the attributes, to look at the values – and so we could make another call inside of here. And so, instead of calling getText, we could say getAttribute. And let's say we ask for the attribute for "value" instead, here. And so we put the value there…Oop, made one more parenthesis…two more, just one more…and we can **Run** this again.   
*The presenter switches to the source code for the web page and scrolls down. The code that's visible is  
  
<p class="narrative">  
I am here to say Hello  
</p>  
  
<div>  
  
               <p class="opening">  
                       Goodbye  
               </p>  
  
               <p class="narrative">  
                       It is time to go  
               </p>  
               <input type="text" name="inputBoxTofind" value=default input value"/>  
</div>  
  
<p class="opening">  
Are you sure?  
</p>  
  
<p class="narrative">  
That was a question?  
</p>  
  
The presenter highlights the code name="inputBoxTofind".  
  
In Eclipse, the presenter highlights the code By.name("inputBoxTofind")).  
  
The presenter scrolls through the content in the Console pane. The following content is listed:  
  
Direct to ID <p class="opening" id="greetings">  
Hello  
//////////////////////////////////////////////////////////////////////////  
By Name  
  
Direct to name <input type="text" name="inputBoxTofind" value="Default input value" />  
  
/////////////////////////////////////////////////////////////////////////  
By Link Name  
  
Direct to name <a href="">  
  
The presenter highlights  
  
<input type="text" name="inputBoxTofind" value="Default input value" />.  
  
The presenter shows the tooltips for each of the components in the following code:  
  
element.getText());.   
  
The presenter then navigates to the line  
  
System.out.print1n(element.getAttribute(name)getText());  
  
In the line, the presenter removes (name)getText()); and replaces it with ("value")));.  
  
The presenter then runs the code.*  
  
And here, now we see we get the Default input value. So, just wanted to show you inside of there. We have some base tests inside of here that was working with a different type of element, but this search works on any type of element. We switched from searching text boxes…we switched to get back instead input element. And we just need to change how we interact with that element, but we find the elements the same way. Another common option is to link…click on a link. And so, if we know in our test what the name of the link is…so, let's say we're going to go click on this "Question" link out here. As we can see here, here's a link that's titled this **Question** right here. Well, we can go up and look up that link by the questions. We say By.Link, and this is just straight up the text you see on the page, which you would be clicking on if you were a user. And so we go straight to the name and so, for this one, you can see we get the <a href=""> out there. And the text inside of that…the link is Question – what we see inside of there. The key of this, though, as we've said, is it must have an element out there. If we go through – like we see in this last test – and we do a findItem and the ID, or the link name, or whatever it is, is missing –   
*The results are listed in the Console pane, as follows:  
  
//////////////////////////////////////////////////////////////  
By ID  
  
Direct to ID <p class="opening" id="greetings">  
Hello  
//////////////////////////////////////////////////////////////  
By Name  
  
Direct to name <input type="text" name="inputBoxTofind" value="Default input value" />  
Default input value  
//////////////////////////////////////////////////////////////  
  
The presenter highlights Default input value in the Console pane.  
  
In the code window, the presenter highlights  
  
WebElement element = driver.findElement(By.name("inputBoxTofind"));  
  
He then highlights the line  
  
System.out.println(element.getAttribute("value"));.  
  
The presenter scrolls down the code. The following code is now visible:  
  
public void c\_byLinkName()  
       {  
               System.out.println("///////////////////////////////////////////////////////////////");  
               System.out.println("By Link Name\n");  
                 
               WebElement element = driver.findElement(By.linkText("Question"));  
               System.out.println("Direct to name " + element);  
               System.out.println(element.getText());  
       }  
         
       @Test(expected=NoSuchElementException.class)  
       public void d\_missing()  
       {  
               System.out.println("///////////////////////////////////////////////////////////////");  
               System.out.println("Not there\n");  
               WebElement element = driver.findElement(By.id("missing"));  
               System.out.print1n("Direct to ID " + element);  
               System.out.print1n(element.getText());  
  
The presenter highlights By.linkText("Question").  
  
He switches to the web page in the browser and points out the link Question.  
  
The presenter switches back to Eclipse and highlights the code By.linkText("Question") again.  
  
He then highlights Direct to name.  
  
In the Console pane, he highlights <a href=""> and then Question.  
  
In the code window, the presenter highlights driver.findElement. He then highlights missing. The tooltip By org.openqa.selenium.By.id(String id) displays.*   
  
in this case, literally and figuratively; it's not out there, it's not in our web page – then what we'll get back is an exception. And the last test actually passes here, because we're looking for the NoSuchElementException – this is what is called. So, if I didn't have this exception catch inside of here, and I ran it without it and I was getting an item on here, it would say: Hey, NoSuchElementException. Now, the NoSuchElementException out here that we can see, if we look at the task that we…it tells us a lot of details here; which page we're looking at. It even tells us what host we're going to, the name of the operating system, all that sort of stuff. That's, you know, good stuff out there that helps us, because we need to figure out, Is our test bad or was our unit under test…the page under test bad? But, that is the key; is if I ask for a single element, I must get the single element. If I don't want a single element, I just use findElements with an 's' instead, and use that and check for a zero value if it's not in there. So, this is how we find the exact element in the page. You work with that separately but, if you need to find it – no matter what the element is – this is the way to do it.   
*The presenter selects the code =NoSuchElementException. The tooltip org.openqa.selenium.NoSuchElementException displays.  
  
Next the presenter changes the line @Test(expected=NoSuchElementException.class) to just @Test(). He runs the code and the Console pane includes the message "Not there."  
  
In Eclipse, the presenter expands the Failure Trace pane, which lists the entry org.opnqa.selenium.NoSuchElementExcepton: Unable to locate element with ID:, along with a URL to visit for more information and relevant build, system, and driver information.  
  
In the code window, the presenter highlights findElement in the line  
  
WebElement element = driver.findElement(By.id("missing"));*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Many Elements in Java

Learning Objective

*After completing this topic, you should be able to*

* *search for many page elements in Selenium Java projects*

**1. Finding multiple elements**

Some tests require you to search a page to find several elements at a time. Selenium has many options, using popular search technologies, to enable you to find the elements you need to complete your test. So, looking at this page we have, there's a lot of different items we might search for as a group. We might want to look for all the links; we might want to look for all the bolded text inside of here. There's a number of different ways we might want to settle down on specific elements. And, when we're not sure if there's just one or if we absolutely know there's more than one, we need to find all those elements – ideally, at once – with a search. So, the first thing we can do is say, Hey, let's search by a partial link. Now, we're still using the driver to findElement(), but now we're saying findElements() – we're looking for many. And so these searches can actually be used as part of finding a single element or finding many. If I do a single element, it just returns the first one; if I do find many, it's going to return all of them – or possibly none. So, when we say By.partialLink, we're searching all the links on the page – the <a href=""> sort of things – but we're looking for text included in the link that has the word ("Link") inside of there.   
*In a browser, a web page titled Searches is open. It contains a placeholder title called Centered Title, the links Question, Greeting, Test Link, and Test Multiple Links, a section titled Hellow with the text "I am here to say Hello", and a section titled Goodbye with the text "It is time to go" and a text box that currently contains the text "Default input value." It also contains a section titled Are you sure?, which contains the text "That was a question?"  
  
In Eclipse, the file FindingElements.java is open. It contains the code  
  
package find;  
  
import java.util.List;  
  
import org.junit.FixMethodOrder;  
import org.junit.Test;  
import org.junit.runners.MethodSorters;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebElement;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class FindingElements extends BaseSearch  
{  
       @Test  
       public void a\_byPartialLink()  
       {  
               System.out.println("///////////////////////////////////////////////////////////////");  
               System.out.println("By Partial Link\n");  
                 
               List<WebElement> elements = driver.findElements(By.partialLinkText("Link"));  
               for (WebElement e : elements)  
               {  
                       System.out.println(e.getText());  
               }  
       }  
  
The presenter focuses on the code  
  
               System.out.println("By Partial Link\n");  
                 
               List<WebElement> elements = driver.findElements(By.partialLinkText("Link"));*   
  
So, if we go back there, we can see I have the Test Link and Test Multiple Links. So, in theory, I should get two back; and if I look here, I actually do indeed get two elements back – and they each have the text as shown. I get the whole element back, but I'm just showing you the text as part of the test. But now, I can do whatever I want to. I can click on those links, I can analyze those links further, and do whatever behavior I need to… to test my page. So, very simple; I go from searching for a single link to, now, multiple links and now I can search on, specifically, partial text inside of that link. Now, past that, let's say I want to find all the <li> elements, or all the <a href=""> elements, or all the <p></p> page elements; whatever tag I want to search on, I can do that as an option – I can search by tag name. And so, if we go back to our page here, you can see there's a multiple number of the elements that have the <li> here, the line item. So, we have an unordered list and then a line-item list inside of here, and it shows up on our page as the different links that are on here. And, when we run this test, we find everything inside of there – and you can see, they have each one of the text that's inside of there; each one of those links that come up. And I find them all real quickly by searching on this. Now,   
*The presenter switches to the Searches web page and then returns to Eclipse. He highlights e.getText() in the code window and then, in the Console pane, highlights Test Link and Test Multiple.   
  
The presenter then highlights the code  
  
List<WebElement> elements = driver.findElements(By.partiallinkText("Link"));  
for (WebElement e : elements).  
  
The presenter scrolls down. The following code is now visible:  
  
@Test  
public void a\_byTagname()  
       {  
               System.out.println("///////////////////////////////////////////////////////////////");  
               System.out.println("By Tag Name\n");  
                 
               List<WebElement> elements = driver.findElements(By.tagName("li"));  
               for (WebElement e : elements)  
               {  
                       System.out.println(e.getText());  
               }  
       }  
         
       @Test  
       public void c byClass()  
  
He then highlights the code ("li"). Next he highlights the code By.tagName("li").  
  
The presenter switches to the browser and opens a second tab that contains the source code for the web page. He highlights the code li. He then switches back to the web page before returning to Eclipse.   
  
The presenter points out the links listed in the Console pane – Question, Greeting, Test Link, and Test Multiple Links. He then highlights the code ("li") in the code window. The By org.openqa.selenium.By.tagName(String name) tooltip opens.*   
  
it's going to literally return every single one of those line item in the entire page. But. that's only…well, that's what we asked for – return all the tags like that. So, if we know there's just one list, that works fine. If we need to be smarter, then we have to go a little bit deeper; use one of these stronger search tools. So, the next one we can talk about, is by the CSS class. And so we can say, find all the elements by the class. So, if I say by the class ("opening") – and I go back to my web page here real quick, I can see – in this source – I have a class of "opening" on several of these <p> elements inside of here. And so, I have a "opening" and "narrative"; another "opening" and "narrative", and I think I might even have a third one around here – yep, there it is, "opening" and "narrative" inside of there. And so there's three options that should be coming back and, if I look at my search, when I search for By.class, I'm actually doing the search just to show you that capability. I can ask for findElement() here, which just returns the first of them. So, it's returning back to me just that very first "opening" element – the one that has the ID of "greetings". It's, you know, the word Hello right here. Or I can search for "All of them"; and when I search for "All of them" right here, it returns back Hello, Goodbye, and Are you sure? And, just to equate that on our page here, you can see we have Hello, Goodbye, and Are you sure? Those are the elements that are bold because, in our style sheet, we defined the class opening and we made that font bold.   
*The presenter scrolls down the code window. The following code is shown:   
  
WebElement element = driver.findElement(By.className("opening"));  
               System.out.println("First element " + element);  
               System.out.println(element.getText());  
  
               System.out.println("All of them");  
               List<WebElement> elements = driver.findElements(By.className("opening"));  
               for (WebElement e : elements)  
               {  
                       System.out.println(e.getText());  
               }  
       }  
  
       @Test  
       public void d\_byCSS()  
  
The presenter highlights By.className("opening"), and then highlights ("opening").  
  
The presenter switches to the source code in the browser. The following source code is visible:  
  
</table>  
  
<ul>  
       <li> <a href="">Question</a></li>  
       <li> <a href="">Greeting</a></li>  
       <li> <a href="">Test Link</a></li>  
       <li> <a href="">Test Multiple Links</a></li>  
<ul>  
  
<p class="opening" id="greetings">  
Hello  
</p>  
  
<p class="narrative">  
I am here to say hello  
</p>  
  
<p class="narrative">  
I am here to say Hello  
</p>  
  
<div>  
               <p class="opening">  
                       Goodbye  
               </p>  
  
               <p class="narrative">  
                       It is time to go  
               </p>  
  
The presenter switches to Eclipse and scrolls through the results in the Console pane. He points out the following content:  
  
///////////////////////////////////////////////////////////////////////////  
By Class  
  
First element <p class="opening" id="greetings">  
Hello  
All of them  
Hello  
Goodbye  
Are you sure?  
  
The presenter switches to the Searches web page and then to its source code. The following code is shown:  
  
<style>  
<!--  
.opening  
{  
font-weight: bold;  
}  
-->  
</style>  
  
<h1>Searches</h1>  
<br/>  
<br/>  
  
<table style="width :100%">  
<tr>  
<th align="center">  
<p>  
Centered Title  
</p>  
</th>  
</tr>  
</table>  
  
<ul>  
<li> <a href="">Question</a></li>  
<li> <a href="">Greeting</a></li>  
  
The presenter highlights .opening, as well as font-weight: bold; in the source code.*   
  
So, again, if you want to validate…let's say, error messages, and the error messages have a tag that's associated with that, this is a great way to be able to find everything that's an error message – everything that's in that class, everything that's text to that class. It's a quick and easy way to find that. But, we can get even fancier. So, we can use – beyond that – CSS selectors. This is the same cssSelector that you would use to find your style sheet; to be able to narrow in. So, you can say (".opening") – this is just simply saying a class name of "opening" inside of here. And that's how we defined it right here in our cssSelector – ".opening". If you wanted to say, I want all p elements inside of a div tag, this is the cssSelector that would allow me to do that – just another example here to show you the nature of this language. If you want to know more about this language, there's lots of resources to go look that up. But, once you've figured out what you want to search for, you can do that inside of this Java search. And I am finding here, this <p> element…oops, I'm sorry, this <p> element down here that's located inside of a <div> tag – it's actually finding all of these. And so, it's not only…loading the rest of it; it's only going to load this one, inside of a <div> tag. And so, when I look at my CSS Selectors, it returns me all the <p> elements inside of a <div> tag; it says Goodbye, and It's time to go. And, as we can see; Goodbye, It's time to go – those are the only ones that are inside of the <div> tag.   
*The presenter switches back to Eclipse. He scrolls down to the following code:  
  
@Test  
       public void d\_byCSS()  
       {  
               System.out.println("\n///////////////////////////////////////////////////////////////");  
               System.out.println("By CSS selector\n");  
                 
               System.out.println("All elements with a class of opening");  
               List<WebElement> elements = driver.findElements(By.cssSelector(".opening"));  
               for (WebElement e : elements)  
               {  
                       System.out.println(e.getText());  
               }  
  
               System.out.println("\nAll <p> inside a <div>");  
               elements = driver.findElements(By.cssSelector("div > p"));  
               for (WebElement e : elements)  
               {  
                       System.out.println(e.getText());  
               }  
       }  
  
The presenter highlights the code "By CSS selector\n". He then highlights ".opening".  
  
The presenter switches to the source code for the web page and again highlights .opening.  
  
The presenter switches back to Eclipse. He then highlights the code <p> inside a <div> and "div > p".  
  
The presenter switches to the source code for the web page and scrolls down. The code now visible is  
  
Table style="width :100%">  
       <tr>  
               <th align="center">  
                       <p>  
                               Centered Title  
                       </p>  
               </th>  
       </tr>  
</table>  
  
<ul>  
       <li> <a href="">Question</a></li>  
       <li> <a href="">Greeting</a></li>  
       <li> <a href="">Test Link</a></li>  
       <li> <a href="">Test Multiple Links</a></li>  
</ul>  
  
<p class="narrative">  
I am here to say Hello  
</p>  
  
<div>  
  
The presenter switches to Eclipse. He scrolls through the results in the Console pane to the following results:  
  
Hello  
Goodbye  
Are you sure?  
  
All <p> inside a <div>  
Goodbye  
It is time to go  
  
///////////////////////////////////////////////////////////////////  
By xpath  
  
The presenter switches to the source code for the web page. The following part of the code is shown:  
  
</th>  
</tr>  
</table>  
  
<ul>  
<li> <a href="">Question</a></li>  
<li> <a href="">Greeting</a></li>  
<li> <a href="">Test Link</a></li>  
<li> <a href="">Test Multiple Links</a></li>  
</ul>  
  
<p class="narrative">  
I am here to say Hello  
</p>  
  
<div>  
<p class="opening">  
Goodbye  
</p>  
  
<p class="narrative">  
It is time to go  
</p>  
<input type="text" name="inputBoxToFind" value="default input value (remaining code cut off)  
</div>  
  
<p class="opening">  
  
The presenter highlights Goodbye.   
  
He then highlights:  
  
class="narrative">  
indent It is time to go*   
  
The last option we can look at here is an XPath search; an XPath search is an XML search. Again, a really powerful search technique; here we're saying show me all ("//p") elements anywhere in the document. We can see it returns them all inside of here – so, whether they're in <div> tags or whether they're native. I can say, give me only the <p> elements that have the class out there of 'opening'. You know, again, "\nOnly the headers" inside of there – it returns just those three. And then, I can even search to say; Hey, find me a //table and find me all //p elements inside of that table. And so that…actually, if I look back at my link inside of here, I only have one <table>. So, if I had many, it returned many. And I only have <p> element inside that table, so it tells me that Centered Title inside of there. And so it returns to me – here, you can see – the Centered Title. So, the XPath is another great search option. So, whichever one you're comfortable with, use that. Last note we'll here…we'll hear real quick before we…are finish up; again, I can search by something that doesn't exist – on the findElements(). And it doesn't fail; it just returns back zero elements. It doesn't fail at all, there's no exception, there's no problem. It's okay to not find something by using the multiple elements. So, these are all the great search options you can use inside of this to load everything. So, take advantage and use the one that really meets your needs the best.   
*The presenter switches to Eclipse. He scrolls down the code window. The following code is displayed  
  
@Test  
       public void e\_byXPath()  
       {  
               System.out.println("\n///////////////////////////////////////////////////////////////");  
               System.out.println("By xpath\n");  
                 
               System.out.println("All <p> elements");  
               List<WebElement> elements = driver.findElements(By.xpath("//p"));  
               for (WebElement e : elements)  
               {  
                       System.out.println(e.getText());  
               }  
  
               System.out.println("\nOnly the headers");  
               elements = driver.findElements(By.xpath("//p[@class='opening']"));  
               for (WebElement e : elements)  
               {  
                       System.out.println(e.getText());  
               }  
                 
               System.out.println("\nOnly the Title");  
               WebElement element = driver.findElement(By.xpath("//table//p"));  
               System.out.println(element.getText());  
                 
       }  
  
The presenter highlights "//p".  
  
The presenter scrolls through the Console pane. The following content is shown:  
  
All <p> elements  
Centered Title  
Hello  
I am here to say Hello  
Goodbye  
It is time to go  
Are you sure?  
That was a question?  
  
Only the headers  
Hello  
Goodbye  
Are you sure?  
  
Only the Title  
centered Title  
//////////////////////////////////////////////////////////////////////////////////  
Missing  
  
In the Console window, the presenter highlights   
  
Hello  
Goodbye  
Are you sure?  
  
In the code window the presenter highlights "//table//. He then highlights //p".  
  
The presenter switches to the source code for the web page. The following code is displayed:  
  
</style>  
  
<h1>Searches</h1>  
<br/>  
<br/>  
  
<table style="width :100%">  
<tr>  
<th align="center">  
<p>  
Centered Title  
</p>  
</th>  
</tr>  
</table>  
  
<ul>  
<li> <a href="">Question</a></li>  
<li> <a href="">Greetings</a></li>  
<li> <a href="">Test Link</a></li>  
<li> <a href="">Test Multiple Links</a></li>  
</ul>  
  
<p class="opening" id="greetings">  
Hello  
</p>  
  
<p class="narrative">  
I am here to say Hello  
</p>  
  
The presenter highlights <p>. He then highlights Centered Title.  
  
The presenter then switches to Eclipse. The presenter highlights Centered Title in the Console pane.  
  
The presenter scrolls through the code in the code window and focuses on the following code:  
  
@Test  
       public void f\_missing()  
       {  
               System.out.println("///////////////////////////////////////////////////////////////");  
               System.out.println("Missing\n");  
                 
               List<WebElement> elements = driver.findElements(By.partialLinkText("Nothing"));  
               for (WebElement e : elements)  
               {  
                       System.out.println(e.getText());  
               }  
       }  
         
}  
  
The presenter highlights By.partiallinkText(Nothing"). He then highlights WebElement e : elements)*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Popups in Java

Learning Objective

*After completing this topic, you should be able to*

* *check popups in Selenium Java tests*

**1. JavaScript popups**

If there are interactive JavaScript popups, our test case must not only expect them; but also may want to interact or validate their contents. Selenium provides Java with the tools it needs to detect and respond to popups, based on the needs of our test case. So, in this little simple page right here, if I was going to go through some testing and press these buttons, popups are going to happen. You can see an alert happens here, a confirmation dialog happens here, and a prompt happens here. So, as we're doing these, not only do we need to interact with them…and particularly the prompt, we want to add some text inside of here. So, you can see, if I enter a value inside of here, it's going to show up on my screen. And so, I want to validate this behavior. I might be doing something very useful here; I might even be logging into the site using a pop-p. That could be a basic authentication mechanism that's inside of there – I have to replicate that. So, in order to do that, I have to accommodate that for my Java application…my Java test. So, just to show you here; as an example, if I go through and I press that alert button, and I click it and I don't do anything else, I'll actually end up with an exception. Not only can I just not ignore it, I…you know, I have the desire to interact with it…I also can not ignore it.   
*A web page titled Alerts contains four buttons – Do Nothing, Show Alert, Show Confirm, and Show Prompt. The presenter clicks Show Alert. A popup with an OK button displays the message "popup happened." The presenter clicks OK.   
  
Next the presenter clicks Show Confirm. A popup with OK and Cancel buttons displays the message "confirm this", as well as a Prevent this page from creating additional dialogs checkbox. The presenter clicks Cancel.  
  
Next the presenter clicks Show Prompt. A popup with OK and Cancel buttons contains an Enter a value text box and a Prevent this page from creating additional dialogs checkbox. The presenter clicks Cancel.  
  
The presenter clicks Show Prompt again. He enters Test value in the Enter a value text box and presses Enter.  
  
On the web page, the text "Test value" appears above the four buttons.  
  
The presenter switches to Eclipse. In the code window, the following code is displayed:  
  
package alerts;  
  
import static junit.framework.Assert.assertEquals;  
  
import org.junit.Test;  
import org.openqa.selenium.Alert;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.firefox.FirefoxDriver;  
  
public class Popups  
{  
       @Test  
       public void ignoreAlert()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/alerts.html");  
                 
               WebElement button = driver.findElement(By.id("showAlert"));  
                 
               button.click();  
                 
               driver.close();  
       }  
         
       @Test  
       public void alerts()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/alerts.html");  
                 
               WebElement button = driver.findElement(By.id("showAlert"));  
                 
               button.click();  
                 
               Alert alert = driver.switchTo().alert();  
                 
               alert.dismiss();  
                 
               driver.close();  
       }  
  
       @Test  
       public void confirm()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/alerts.html");  
                 
               WebElement button = driver.findElement(By.id("showConfirm"));  
  
The presenter highlights   
  
WebDriver driver = new FirefoxDriver();  
driver.get("http://localhost:8080/website/pages/alerts.html");  
  
The presenter highlights WebElement button = driver.findElement(By.id("showAlert"));, and then highlights driver.close();.  
  
On the JUnit tab, the presenter scrolls through the results in the Failure Trace pane.*   
  
You can see, this just fails my test. It just absolutely says, UnhandledExceptions, Modal dialog present, popup happened. So, I have to accommodate for that in this test. So, the way I do that inside of my Java application is, from my driver, I say Hey, switchTo() me to an alert(). And that gives me the response to that alert that's going on there. And I can, very simply, go through and either confirm that alert, or dismiss that alert, or however I want to interact with that alert. We'll see some examples of that in a second so, for the basic alert, I can just say dismiss(), and my test case runs just fine – nothing fails. Now, in a confirm() dialog, I have that choice of hitting **OK** or hitting **Cancel**. I can do either one of them. I hit accept() or dismiss() inside of there. And so, in this case, I'm going to hit accept(). Now, that may or may not have some difference in behavior on the page. I might have one test case where I accept it and another one where I cancel it. To be able to see what's going on inside of there…but, again, Selenium's giving my JavaScript test…my Java code the test to test that JavaScript. However, it's going to be going either direction. I can control the flow of what's going on inside of there.   
*In the Failure Trace pane, the presenter highlights the entry org.openqa.selenium.UnhandledAlertException:.  
  
The presenter scrolls down the code window. The following code is displayed:  
  
WebElement button = driver.findElement(By.id("showAlert"));  
                 
               button.click();  
                 
               driver.close();  
       }  
         
       @Test  
       public void alerts()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/alerts.html");  
                 
               WebElement button = driver.findElement(By.id("showAlert"));  
                 
               button.click();  
                 
               Alert alert = driver.switchTo().alert();  
                 
               alert.dismiss();  
                 
               driver.close();  
       }  
  
       @Test  
       public void confirm()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/alerts.html");  
                 
               WebElement button = driver.findElement(By.id("showConfirm"));  
                 
               button.click();  
                 
               Alert alert = driver.switchTo().alert();  
                 
               alert.accept();  
                 
               driver.close();  
       }  
         
  
       @Test  
       public void prompt()  
       {  
  
The presenter highlights driver.switchTo().alert and then alert.dismiss();.  
  
The presenter scrolls down and the following code displays:  
  
alert.dismiss();  
  
driver.close();  
}  
  
@Test  
public void confirm()  
{  
WebDriver driver = new FirefoxDriver();  
driver.get("http://localhost:8080/website/pages/alerts.html");  
  
WebElement button = driver.findElement(By.id("showConfirm"));  
  
button.click();  
  
Alert alert = driver.switchTo().alert();  
  
alert.accept();  
  
driver.close();  
}  
  
  
@Test  
public void prompt()  
{  
  
The presenter highlights alert.accept() and then the line  
  
Alert alert = driver.switchTo().alert();  
  
The presenter scrolls down, revealing the code  
  
}  
         
  
       @Test  
       public void prompt()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/alerts.html");  
                 
               WebElement button = driver.findElement(By.id("showPrompt"));  
                 
               button.click();  
                 
               String promptText = "I am responding to your prompt";  
               Alert alert = driver.switchTo().alert();  
               alert.sendKeys(promptText);  
               alert.accept();  
  
               WebElement results = driver.findElement(By.id("answerText"));  
               assertEquals("The value sent is the same that is on the page",   
                                    promptText, results.getText());  
                 
               driver.close();  
       }  
         
}*   
  
The last type of popup we can talk about, is the prompt. And so the prompt out there is going to go and ask for text – the one we just saw inside of there. So, there, when I click the button, I can go through and I can switchTo() that alert(), and then I can interact with that. I can sendKeys over to that prompt. I can go and then accept() it – hit **OK**, and then go off and check; did the text I sent to on the page match the text that's updated on that page. I can go back and forth between those. So, in my Selenium, I'm given a lot of options to interact with this. Now, let's take a second before we leave this, and just look at all the different things that I can do when I have this alert box. So, you can see, I can actually accept() something, or I can authenticate – as we've said, in basic authentication, using credentials outside of there. The web page might ask me for a username/password and Selenium gives me an option to be able to do that – and allows me to protect those credentials along the way. I can cancel; I can check if the contents of the alert is equal to what I want and what I'm looking for. Did the text that gets put in the popup match what I'm expecting, because some of those popups might be dynamically generated? And I want to validate what's going on inside of there. Or I can just get the text myself. And then, I can do a sendKeys down here – so I can ship some text that I want to that prompt. So, again, my alert's capability, my confirmation and my prompt capability is not limited in my test suite. Not only do I want to interact with them – I have to interact with them in my test suite to acknowledge that it happened, and then be able to guide the behavior forward appropriately, according to my test cases.   
*The presenter highlights the line  
  
Alert alert = driver.switchTo().alert();  
  
Next the presenter highlights  
  
alert.sendKeys(promptText);  
  
The presenter also highlights the code alert.accept();.  
  
The presenter types alert. in the code and a drop-down list provides options such as  
  
accept():void - Alert  
authenticateUsing(Credentials credentials):void - Alert  
dismiss():void - Alert  
equals(Object arg0):boolean - Object  
getClass():Class<?> - Object  
getText():String - Alert  
hashCode():int - Object  
notify():void - Object  
notifyAll():void - Object  
sendKeys(String keysToSend):void - Alert  
toString():String - Object  
wait():void - Object*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Multiple Windows in Java

Learning Objective

*After completing this topic, you should be able to*

* *work with multiple windows in Selenium Java tests*

**1. Multiple frames and windows**

If a web site uses multiple frames or windows, I would say our Java-based solution needs to know. As it can't visually see the new window, it must be aware of the context shift as it goes between the pages. Thus, testing in sites with multiple windows requires special planning and consideration. As we look at the test here, we have a very simple web site. But, when I click on a button, it opens up a new window. And, as it goes through and clicks on this new window, it's going to pop up that pane. But, our code is not going to be aware of that pane directly. As you notice here, I find my button, I'm going to click() on that button three different times and…actually, naming the windows slightly different each different time. I'm going to click() on the button three different times, but I'm not actually switching to that window at any point. Our Java code's going to stay on the same window until we're told to change. So, we go ahead and just **Run** this. You can just see, as it goes through, it's going to open up the window – and it's going to open up a bunch of other windows. Now, when I want to switch to a specific window, I can go through and ask the driver to go give me all the handles of all the windows that are out there. There's no really easy, obvious way to tell which window's which. And so I can open up all the windows' handles and you can see from the output there, window handles is this big, long, encoded integer-string combination; it's hexadecimals inside of here.   
*The file Windows.java is open in Eclipse. It contains the code  
  
@Test  
public void checkHandles()  
{  
       WebDriver driver = new FirefoxDriver();  
       driver.get("http://localhost:8080/website/pages/windows.html");  
  
WebElement button = driver.findElement(By.id("makeVisible"));  
WebElement windowNameInput = driver.findElement(By.id("windowName"));  
  
windowNameInput.sendKeys("window 1");  
button.click();  
  
windowNameInput.sendKeys("window 2");  
button.click();  
  
windowNameInput.sendKeys("window 3");  
button.click();  
  
for (String handle : driver.getWindowHandles())  
{  
       System.out.println(handle);  
       driver.switchTo().window(handle);  
       driver.close();  
}  
  
driver.quit();  
}  
  
The presenter highlights windowNameInput.sendKeys("window 1");. The presenter highlights each instance of button.click(); while gesturing over it.  
  
The presenter clicks Run and the test automatically runs in a browser.  
  
In Eclipse, the presenter scrolls down. Additional code is visible:  
  
@Test  
@Ignore  
public void backAndForth()  
  
The presenter highlights driver.getWindowhandles(), and then highlights System.out.println(handle);.  
  
There are four sets of hexadecimals visible in the Console pane. These sets are:  
  
{f7421894-a1ce-4024-bd3f-2746b5c6e7a0}  
{8471a070-41ec-4466-956d-5f67d1f466d2}  
{1e506226-3f57-40b9-bb34-7e828a826f7d}  
{4716f552-0022-454b-8ecd-b0491baf62b5}*   
  
And so, I can switch to that window using the switchTo command and the (handle) – take that handle and say, Let's go switchTo() the window of that (handle). And I can even choose to close() a window inside of here. I just want to point out here; close() versus quit(). close() will close an individual window; quit() will stop the whole test – the driver, everything out there. It'll go off and close that and our Java application. So, just be aware of that difference inside of there. But, again, as I open it up, I stay on the same window. Once I've looked up that element, it's going to stay with that element along the way. That's just to show you how that works; I'm going to go open up our second test here. So, let me take the @Ignore off of this one…pop it back up here, so we can just run the second test. Now, the second test is very similar; it's the same web page actually, and what's going to happen is we're going to go through…we're going to open up a window. And then, it also has a functionality where I'm going to go to a field. I'm going to send a TextValue over that, and then I'm going to check that window to see if that TextValue was received. So, it's kind of like a text-messaging app between the windows inside of here. So, let me **Run** this second test; we can have that pop up. And we're actually going to see a problem inside of here; this is going to stop and let us see what's going on here. So, what we can see is, in the main window, when we hit **Open Window**, it's going to pop open a window. And then, in the second one; it says, I'm going to send some text here. And then, when I hit the Send button, it's going to send that text over to the other window. So, it's just to show you that feature here. I can put whatever garbage inside of there and, when I hit Send, you can see, it shows up on that second window.   
*In the code window, the presenter highlights driver.switchTo().window(handle);. He then highlights driver.close(); and driver.quit().  
  
The presenter then highlights the lines  
  
windowNameInput.sendKeys("window 1");  
button.click();  
  
windowNameInput.sendKeys("window 2");  
button.click();  
  
windowNameInput.sendKeys("window 3");  
button.click();  
  
The presenter scrolls up in the code and then scrolls further down in the file. The visible code is now as follows:  
  
for (String handle : driver.getWindowHandles())  
               {  
                       System.out.println(handle);  
                       driver.switchTo().window(handle);  
                       driver.close();  
               }  
                 
               driver.quit();  
       }  
  
       @Test  
       @Ignore  
       public void backAndForth()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/windows.html");  
                 
               WebElement newWindowButton = driver.findElement(By.id("makeVisible"));  
               newWindowButton.click();  
  
               String updatedHandle = driver.getWindowHandle();  
               System.out.println(updatedHandle);  
  
The presenter deletes @Ignore and scrolls back up. The following code is shown:  
  
import static junit.framework.Assert.assertEquals;  
  
public class Windows   
{  
       @Test  
       public void checkHandles()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/windows.html");  
                 
               WebElement button = driver.findElement(By.id("makeVisible"));  
               WebElement windowNameInput = driver.findElement(By.id("windowName"));  
                 
               windowNameInput.clear();  
               windowNameInput.sendKeys("window 1");  
               button.click();  
  
               windowNameInput.clear();  
               windowNameInput.sendKeys("window 2");  
               button.click();  
  
               windowNameInput.clear();  
               windowNameInput.sendKeys("window 3");  
               button.click();  
  
The presenter adds the line @Ignore immediately below the line @Test.   
  
The presenter scrolls down. The following code is displayed:  
  
@Test  
       public void backAndForth()  
       {  
               WebDriver driver = new firefoxDriver();  
               driver.get("http://localhost:8080/website/pages/windows.html");  
               newWindowButton.click();  
  
               String updatedHandle = driver.getWindowHandle();  
               System.out.println(updatedHandle);  
  
               WebElement windowNameInput = driver.findElement(By.id("windowName"));  
               windowNameInput.sendKeys("window 1");  
  
               WebElement sendText = driver.findElement(By.id("sendTextValue"));  
               String sendMessage = "I want to send this text";  
               sendText.clear();  
               sendText.sendKeys(sendMessage);  
  
The presenter highlights the line newWindowButton.click();.   
  
The presenter scrolls down. Additional code is displayed, as follows:  
  
public void backAndForth()  
       {  
               WebDriver driver = new firefoxDriver();  
               driver.get("http://localhost:8080/website/pages/windows.html");  
               newWindowButton.click();  
  
               String updatedHandle = driver.getWindowHandle();  
               System.out.println(updatedHandle);  
  
               WebElement windowNameInput = driver.findElement(By.id("windowName"));  
               windowNameInput.sendKeys("window 1");  
  
               WebElement sendText = driver.findElement(By.id("sendTextValue"));  
               String sendMessage = "I want to send this text";  
               sendText.clear();  
               sendText.sendKeys(sendMessage);  
  
               WebElement sendButton = driver.findElement(By.id("sendText"));  
               sendButton.click();  
  
The presenter highlights:  
  
WebElement sendText = driver.findElement(By.id("sendTextValue"));  
String sendMessage = "I want to send this text";  
  
The presenter clicks Run to runs the code. A browser window opens. The test starts to run but then encounters a problem.  
  
A web page titled Windows is open. It contains a text box with the entry Window1window1 and an Open Window button, and a text box with the entry "I want to send this text" and an associated Send Text button. The presenter drags the browser window slightly to the side to show a second browser window, which contains a web page titled Another Windows. The page includes the text "I want to send this text."  
  
The presenter switches back to the Windows web page. He enters some random letters in the Send Text text box and clicks Send Text.  
  
The presenter switches to the Another Windows page, which now contains the random letters he entered on the Windows page. The presenter closes the browser windows and returns to Eclipse.*   
  
Now, in our test here though, you can see I'm getting a NoSuchElementException when I'm going and looking up that textLocation. So, I send the text using the window. So, I get down here, and I fail down here – on this statement right down here. Just to show you, I can click here. And this statement right here; when I go and look up that textLocation on the second window, it fails because my context is still on the first window. Even though I clicked the button; even though I've gone to another window, the Java driver here doesn't go to that other window unless I tell it to. And so, let me go through and…oops, sorry, let me go through and add in a little bit of code here, just to show you what we'd need to do to actually fix this. So, if we come down here; before we go and look up that element, we would have to go through and switch context to this…just let me format this real quick so it looks a little bit better. There we go; and so, the first thing I want to do is…I want to get the handle for the window I'm currently on. So, if I say getWindowHandle() still while I'm focused on it, and I can store that String handle available to me. And then I'm going to loop through all the rest of the handles. So, as long as the handle is not the originalHandle, I'm going to skip over to it. I know at this point – because I know my test; I've planned out my test – this is going to be one additional (handle).   
*The code visible in Eclipse is  
  
public void backAndForth()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/windows.html");  
                 
               WebElement newWindowButton = driver.findElement(By.id("makeVisible"));  
               newWindowButton.click();  
  
               String updatedHandle = driver.getWindowHandle();  
               System.out.println(updatedHandle);  
                         
               WebElement sendText = driver.findElement(By.id("sendTextValue"));  
               String sendMessage = "I want to send this text";  
               sendText.clear();  
               sendText.sendKeys(sendMessage);  
  
               WebElement sendButton = driver.findElement(By.id("sendText"));  
               sendButton.click();  
  
The presenter scrolls down. The following code is displayed:  
newWindowButton.click();  
  
               String updatedHandle = driver.getWindowHandle();  
               System.out.println(updatedHandle);  
                         
               WebElement sendText = driver.findElement(By.id("sendTextValue"));  
               String sendMessage = "I want to send this text";  
               sendText.clear();  
               sendText.sendKeys(sendMessage);  
  
               WebElement sendButton = driver.findElement(By.id("sendText"));  
               sendButton.click();  
  
               String updatedText = driver.findElement(By.id("textLocation")).getText();  
               assertEquals("What we sent should match what is received", sendMessage updatedText);  
  
               driver.quit();  
       }  
}  
  
The presenter highlights driver.findElement(By.id("textLocation")).getText();.  
  
The Console pane lists the following hexadecimal set:  
  
{2a99f637-3560-467f-a544-2c9a1a2eca45}  
  
The presenter highlights various elements in the code and then scrolls up to show the following code:  
  
public void checkHandles()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/windows.html");  
                 
               WebElement button = driver.findElement(By.id("makeVisible"));  
               WebElement windowNameInput = driver.findElement(By.id("windowName"));  
                 
               windowNameInput.sendKeys("window 1");  
               button.click();  
  
               windowNameInput.sendKeys("window 2");  
               button.click();  
  
               windowNameInput.sendKeys("window 3");  
  
The presenter then scrolls down to the code  
  
System.out.println(handle);  
                       driver.switchTo().window(handle);  
                       driver.close();  
               }  
                 
               driver.quit();  
       }  
  
       @Test  
       @Ignore  
       public void backAndForth()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/windows.html");  
                 
               WebElement newWindowButton = driver.findElement(By.id("makeVisible"));  
  
The presenter highlights nameOrHandle. He deletes it and replaces it with handle.   
  
The presenter scrolls down. The code now visible is  
  
@Ignore  
public void checkHandles()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/windows.html");  
                 
               WebElement button = driver.findElement(By.id("makeVisible"));  
               WebElement windowNameInput = driver.findElement(By.id("windowName"));  
                 
               windowNameInput.sendKeys("window 1");  
               button.click();  
  
               windowNameInput.sendKeys("window 2");  
               button.click();  
  
               windowNameInput.sendKeys("window 3");  
  
The presenter highlights @Ignore.   
  
The presenter scrolls down. The following code is displayed:  
  
windowNameInput.sendKeys("window 1");  
  
               WebElement sendText = driver.findElement(By.id("SendTextValue"));  
               String sendMessage = "I want to send this text";  
               sendText.clear();  
               sendText.sendKeys(sendMessage);  
  
               WebElement sendButton = driver.findElement(By.id("sendText"));  
               sendButton.click();  
  
               String updatedText = driver.findElement(By.id("textLocation")).getText();  
               assertEquals("What we send should match what is received", sendMessage, updatedText);  
  
               driver.quit();  
       }  
}  
  
The presenter places the cursor below the line  
  
sendText.sendKeys(sendMessage);  
  
He then adds the code  
  
String.originalHandle = driver.getWindowHandle();  
               for (String handle ; driver.getWindowHandles())  
               {  
               if (handle.equals(originalHandle))  
               {  
               continue;  
               }  
               driver.switchTo().window(handle);  
               break;  
               }  
  
The presenter right-clicks to open a shortcut menu and selects Source - Format. The following code is displayed:  
  
WebElement sendText = driver.findElement(By.id("sendTextValue"));  
String sendMessage = "I want to send this text";  
sendText.clear();  
sendText.sendKeys(sendMessage);  
  
String.originalHandle = driver.getWindowHandle();  
               for (String handle ; driver.getWindowHandles())  
               {  
                       if (handle.equals(originalHandle))  
                       {  
                               continue;  
                       }  
                       driver.switchTo().window(handle);  
                       break;  
               }*   
  
And so let me go get that guy, and then I can go and say, Go find the element called "sendText". At that point, then I know I have it and then I can click()…I'm sorry…I actually did this one element below…I put this in the wrong spot…there you go. At that point, that I can go look for the textLocation and I can assert this. Let me just go ahead and **Run** this while we're…and we can see this working. The most important thing, though, is I need to go find and switchTo() the appropriate window. So, now that I've added this, I've gone from the original window – which was right here – and I've switched to the other window, which is down here. You can see, I now can go through and I can find the text and I can match; Hey, is the message I sent the same that…text that was updated on that window up above? This is the key part about the multiple windows; I have to find a way to isolate which window it is. Again, I have to be able to plan ahead. If, for some reason, three or four windows pop up, I might have to do some searching. Instead of saying findElement, I'd have to say findElements and see which window is which – I'd have to do that logic, because there's no easy and obvious way. I could look up the title of the page, I can look up an item on the page…there's ways I can do it – it's just not immediately given to me by the handle. The handle is just a text identifier. So, this is the way – in our coded solution using Java – we can switch between windows as we want to build tests that go about this.   
*The presenter scrolls down to show the code  
  
{  
                       continue;  
               }  
               driver.switchTo().window(handle);  
               break;  
       }  
  
  
       WebElement sendButton = driver.findElement(By.id("sendText"));  
       sendButton.click();  
  
       String updatedText = driver.findElement(By.id("textLocation")).getText();  
       assertEquals("What we sent should match what is received, sendMessage, updatedText);  
  
       driver.quit();  
}  
  
The presenter deletes the code  
  
WebElement sendButton = driver.findElement(By.id("sendText"));  
sendButton.click();  
  
The presenter deletes empty lines and then scrolls up. The code now displayed is  
  
sendText.clear();  
sentText.SendKeys(sendMessage);  
  
WebElement sendButton = driver.findElement(By.id("sendText"));  
sendButton.click();  
  
String originalHandle = driver.getWindowHandle();  
for (String handle ; driver.getWindowHandles())  
{  
       if (handle.equals(originalHandle))  
       {  
                               continue;  
                       }  
                       driver.switchTo().window(handle);  
                       break;  
               }                 
  
               String updatedText = driver.findElement(By.id("textLocation")).getText();  
               assertEquals("What we sent should match what is received, sendMessage, updatedText);  
  
               driver.quit();  
       }  
  
The presenter opens the Save and Launch dialog box and clicks OK. He then returns to the code and highlights various elements in it.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Drag and Drop in Java

Learning Objective

*After completing this topic, you should be able to*

* *use drag and drop in Selenium Java tests*

**1. Drag and drop screen elements**

Using advanced JavaScript on our pages allows designers to imagine screens where users can be very visual and hands-on to manipulate information. One key ability is the chance to drag-and-drop screen elements around the page. In order to test this, we must provide basic mouse actions as part of our Selenium testing. So looking at this page here, you can see we can move this box, this picture around the screen. It's just a basic picture that we have but we can move it wherever we want to on the screen. And JavaScript is enabling that behind the scenes. So to test this feature, we need to get into the actions of the driver that's in there for our Java solution. So we go and find that element called drag1. That's the ID given to that element. And we can actually target it to a very specific location. So we saw those target boxes around there. Well, we can name those targets, div1 and we'll see in a minute div2. And we can take the element from where it's at to that target.   
*In a browser window, a web page contains two empty panes and a block that consists of four differently colored squares.  
  
The presenter drags the block into the left pane and then into the right pane. He moves it between the panes again and then back to its starting point.  
  
The presenter switches to Eclipse. In a file named DragAndDrop.java, the following code is shown:  
  
import org.openqa.selenium.firefox.FirefoxDriver;  
import org.openqa.selenium.interactions.Actions;  
  
public class DragAndDrop  
{  
       @Test  
       public void dragAndDrop()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/drag5.html");  
                 
               WebElement dragger = driver.findElement(By.id("drag1"));  
               WebElement target = driver.findElement(By.id("div1"));  
               Actions worker = new Actions(driver);   
               worker.dragAndDrop(dragger, target);  
               worker.perform();  
                 
               pause(3);  
                 
               WebElement target2 = driver.findElement(By.id("div2"));  
               worker.dragAndDrop(dragger, target2);  
               worker.perform();  
  
The presenter scrolls down to show the additional lines  
  
pause(3);  
  
worker.clickAndHold(dragger);  
The presenter highlights the code WebElement dragger = driver.findElement(By.id("drag1"));. He then highlights drag1.  
  
The presenter highlights .findElement(By.id("div1"));. He then highlights By.id("div2"), and then highlights (By.id("div1")) again.  
  
The presenter highlights .findElement(By.id("drag1")); again. Next he highlights Actions worker.*   
  
And the way we do that was we get an action out there. And the action is built out of the driver, so we take the driver provided there is the constructor. And then we can call different actions on here. So you can see that one of the actions we can do is drag and drop. Now, the drag-and-drop action consists of a bunch of sub-actions. I'm going to move the mouse to a location. I'm going to hold the mouse down. I'm going to keep the mouse down and move it another location, release the mouse and that's going to be my drag and drop. If I wanted to, we can actually see inside of the worker, I have a bunch of those options available to me. So I can click; I can click and hold… Let's scroll this down. I can do key actions; I can move elements, to move to an element around there; I can move – there are a bunch of different options. We'll see a few of these as we go. And so drag and drop is the one we're looking at right now. And so once we've set up all those actions, then we will tell it to perform those actions. And then just visually speaking, we're going to pause for a few seconds. And then we're going to move to the second box inside of here to go down to drag2 and drag and drop around the next box. And then our last action here is going to just move by offset.   
*In the line worker.dragAndDrop(dragger, target);, the presenter right-clicks dragAndDrop. A drop-down list lists options such as build(), clickAndHold, contextClick(), and dragAndDropBy. The presenter selects clickAndHold(WebElement onElement):Actions.   
  
The presenter right-clicks again and selects contextClick():Actions.   
  
The following code is shown in the code window:   
driver.get("http://localhost:8080/website/pages/drag.html");  
         
       WebElement dragger = driver.findElement(By.id("drag1"));  
       WebElement target = driver.findElement(By.id("div1"));  
       Actions worker = new Actions(driver);  
       worker.dragAndDrop(dragger, target);  
       worker.perform();  
  
       pause(3);  
         
       WebElement target2 = driver.findElement(By.id("div2"));  
       worker.dragAndDrop(dragger, target2);  
       worker.perform();  
  
       pause(3)  
  
       worker.clickAndHold(dragger);  
       worker.moveByOffset(100, 100);  
       worker.release();  
       worker.perform();  
}  
  
The presenter scrolls up in the code window. The visible code is now  
  
public class DragAndDrop  
{  
       @Test  
       public void dragAndDrop()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/drag5.html");  
                 
               WebElement dragger = driver.findElement(By.id("drag1"));  
               WebElement target = driver.findElement(By.id("div1"));  
               Actions worker = new Actions(driver);   
               worker.dragAndDrop(dragger, target);  
               worker.perform();  
                 
               pause(3);  
                 
               WebElement target2 = driver.findElement(By.id("div2"));  
               worker.dragAndDrop(dragger, target2);  
               worker.perform();  
  
               pause(3);  
  
               worker.clickAndHold(dragger);  
  
The presenter highlights the lines  
  
worker = new Actions(driver);  
worker.dragAndDrop  
  
The presenter highlights worker.perform();. He then highlights pause(3);. Next he highlights the line  
  
target2 = driver.findElement(By.id("div2"));.  
  
The presenter then highlights the code .moveByOffset(100, 100);.*  
  
So we don't have to move to a location. We can just say move over a 100 pixels to the right and a 100 pixels down; or negative 100 pixels to the left, negative 100 pixels up. That's the relative motion inside of here. But you can see I can click and hold, move by an offset and release. Or I can do a drag-and-drop. The drag-and-drop only gives me the option to go to a target. I don't have a drag-and-drop next to my location. So if you want to do a random move offset I have to do that by hand: click and hold, move my offset and release. All right, and so let's take a look at this, let's take a look at running this guy and see how it goes. So once we execute this, it's going to pop up our screen and you can see it moves it up; pause for three seconds. It's going to move it over, pause for three seconds; and it's going to randomly shuffle it down a little bit. So when I move the first two, it's going to be putting it exactly where that location is. It's going to be putting it in the center of that div box that's outside of there. Now, our JavaScript here enables it to move wherever. So I can, you know, take this over. I can move this in between. It's not snapping to the center of this. There's no, okay, if I'm in the box I'm going to snap it too; unless I've coded that inside of my JavaScript. Now, this is just some basic JavaScript code, a library. It's been written and grabbed from somewhere. Know that HTML5 also has the ability to do a drag-and-drop.   
*The presenter highlights the code clickAndHold(dragger); and then worker.release();.  
  
The presenter right-clicks dragAndDrop and a drop-down list lists the options new, nls, runnable, and toarray.  
  
In the code, the presenter highlights the lines  
  
worker.clickAndHold(dragger);  
worker.moveByOffset(100, 100);  
worker.release();  
  
The presenter clicks Run to run the code.   
  
In a browser window, the web page that contains two panes and the multicolored block opens. The block automatically moves into the left pane, into the right pane, and then to a position overlapping the bottom of the right pane.   
  
The presenter then drags the block into various positions.  
  
The presenter changes the URL in the address bar to localhost:8080/website/pages/drag5.html and presses Enter. A similar web page opens but the two empty panels are outlined in red instead of gray. The presenter right-clicks and opens the source code for the page.  
  
The following code is shown:  
  
ev.preventDefault();  
       var data = ev.dataTransfer.getData("Text");  
       ev.target.apprentChild(document.getElementById(data));  
       document.getElementById('status').innerHTML = name;  
  
}  
</script>  
</head>  
<body>  
  
<div id="status"></div>  
<table>  
       <tr>  
               <td>  
                       <div id="div1" class="target" ondrop="drop(event, 'left')"  
                       </div>  
               </td>  
               <td>  
                       <div id="div2" class="target" ondrop="drop(event, 'right')  
                       </div>  
               </td>  
       </tr>  
</table>  
  
<img id="drag1" src="imagemap.png" draggable="true" ondragstart="drag (code cut off)  
  
The presenter highlights draggable="true".  
  
The presenter closes the source code and switches to Eclipse. The code that's currently visible in the DragAndDrop.java file is  
               driver.get("http://localhost:8080/website/pages/drag.html");  
                 
               WebElement dragger = driver.findElement(By.id("drag1"));  
               WebElement target = driver.findElement(By.id("div1"));  
               Actions worker = new Actions(driver);   
               worker.dragAndDrop(dragger, target);  
               worker.perform();  
                 
               pause(3);  
                 
               WebElement target2 = driver.findElement(By.id("div2"));  
               worker.dragAndDrop(dragger, target2);  
               worker.perform();  
  
               pause(3);  
  
               worker.clickAndHold(dragger);  
               worker.moveByOffset(100, 100);  
               worker.release();  
               worker.perform();  
       }  
  
The presenter highlights the URL in the line  
  
driver.get("http://localhost:8080/website/pages/drag.html");*   
  
We can actually look here. I have a second page that has enabled my HTML5 capabilities. And if I look at the page source inside of here, you can see there is… I'm using the draggable context inside of here for my picture. I have draggable instead. Now, if I target this at HTML5, unfortunately when I go off and run this solution and let me actually close this so it will run – it doesn't do anything. Because Selenium right now as of this recording is not enabled to make the drag-and-drop in HTML5 work. It just doesn't work that way. I can move it. You can see I can move it here and I can drop it wherever I want to. And it does actually, wherever I drop it into this box, as long as my mouse is in the box, it will actually center it into the item. The HTML5 drag-and-drop doesn't let you put it anywhere. It just let's put it into a specific location. And I have a little code here just to show that it's actually moving inside of there. So Selenium just does not enable that for HTML5 at this point. So you can use it, but you have to use it on old versions of drag-and-drop. But take good advantage in planning your test and even if you have this visual element drag-and-drop, you can test that from your Selenium Java tests.   
*The presenter changes the URL in the code to http://localhost:8080/website/pages/drag5.html. He then clicks Run.  
  
The second web page, with the panes outlined in red, opens but the test doesn't run.  
  
The presenter manually drags the multicolored block into different positions on the page. He demonstrates that the block automatically snaps into a centered position in a pane once it's dragged close to that pane.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Input Data in Java

Learning Objective

*After completing this topic, you should be able to*

* *provide input data in Selenium Java tests*

**1. Different types of input**

There are many types of inputs to a web page, and Selenium WebDriver gives us features to provide each HTML type from our Java tests. Let's take a look at some of the basic input types, as well as how we can submit the entire form when our input is completed. So here we see a text box, and a test for the text box. And when we run this, you can see we're going to find this text box element. We're finding it by name. And so the name element of this is text box. And then we're sending some keys over. We pause for a moment, we send some more text over, a very simple test. And just to show you here real quick, let me inspect this item. And you can see the name is text box here. That's how we're finding it. Now, there's…we can find this any way we want to. That's a whole different topic outside of here. But once we find the element, then you can see I can send keys over. Now, if I wanted to put all new text inside the box, I have to start with a clear. And so if I wanted to replace sending text with more text, I would have to send clear as opposed to what I did, which is send it two times. But past that, it works pretty simple; the text box is pretty easy.   
*In Eclipse, a file named HandlingInput.java is open. The following code is visible:  
  
import org.junit.Test;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.firefox.FirefoxDriver;  
import org.openqa.selenium.support.ui.Select;  
  
public class HandlingInput  
{  
       private WebDriver driver;   
         
       @Test  
       public void textbox()  
       {  
               WebElement text = driver.findElement(By.name("textbox"));  
               text.clear();  
               text.sendKeys("Sending Text");  
               pause(3);  
               text.sendKeys(" More Text");  
       }  
         
       @Test  
       public void checkbox()  
  
The presenter clicks Run to run the code.   
  
A web page titled Input Items opens. It contains a text box labeled Text input box, which currently contains the entry "Sending Text." It also contains a checkbox labeled Checkbox, radio buttons labeled one, two, and three, and a table that contains list boxes in columns titled Drop down, Basic Multiple List box, Multiple List Box with ID's, and Multiple List Box with Values. The page also contains Submit Query and Reset buttons.  
  
The presenter enters More Text in the text box. He then right-clicks the text box and selects Check Spelling from the shortcut menu. He right-clicks again and selects Inspect   
Element.  
  
The Inspector lists the HTML for the text box control. It includes the line  
  
<input type= "text" name="textbox"></input>  
  
The presenter switches to Eclipse and highlights the lines  
  
public void textbox()  
       {  
               WebElement text = driver.findElement(By.name("textbox"));  
               text.clear();  
               text.sendKeys("Sending Text");  
               pause(3);  
               text.sendKeys(" More Text");*   
  
Now, checkboxes, basically, you click them or not click them. Now, in this test, I'm going to go find my checkbox, and I'm going to randomly click it some number of times. I want you to see it pop, and on and off. It's going to click and pause, and click and pause, and that's the test. But after it clicks it, I want to make an assessment. I want to finish this up with being clicked. And so we see it's clicking and unclicking. And then it's going to pause for a second, and then finally make sure it ends clicked. And the way I can do that is I can go off and ask that checkbox is it currently selected. There's no way I can force a checkbox to be selected. But since I'm in source code here – since I'm writing java code – I can go through and ask "hey, is my checkbox selected? if it is, then leave it alone. If it's not, then click it". And that's what I'm doing here. So the next item we can look at here is radio buttons. Very much like checkboxes, except they tend to come in bunches. So instead of finding a single element, I am going to find all the radio buttons inside of here. They're all going to have the same name, which makes them a group of radio buttons. And so when I go through that, I can go click each one individually. So let me go ahead and run this. I'm going to click each one individually, and then I can see a walk down the line.   
*The presenter scrolls down in the HandlingInput.java file. The following code is now visible:  
  
text.clear();  
               text.sendKeys("Sending Text");  
               pause(3);  
               text.sendKeys(" More Text");  
       }  
         
       @Test  
       public void checkbox()  
       {  
         
               WebElement checkbox = driver.findElement(By.name("checkbox"));  
               int numberOfTimesToClick = new Random().nextInt(3) + 3;  
               for (int i = 0; i < numberOfTimesToClick; i++)  
               {  
                       checkbox.click();  
                       pause(1);  
               }  
               pause(2);  
                 
               // make sure it is clicked now  
               if (! checkbox.isSelected())  
               {  
                       checkbox.click();  
               }  
               pause(2);  
       }  
  
The presenter clicks Run to run the code.  
  
The Input Items web page opens in a browser. On the page, the checkbox is automatically cleared and then selected again twice.  
  
In Eclipse, the presenter highlights the code  
  
(! checkbox.isSelected())  
  
The presenter scrolls down. The following code is now shown:  
  
pause(2);  
       }  
         
       @Test  
       public void radio()  
       {  
               List<WebElement> radioButtons = driver.findElements(By.name("radios"));  
               for (WebElement radio : radioButtons)  
               {  
                       radio.click();  
                       pause(2);  
               }  
               WebElement specificRadio = driver.findElement(By.xpath("//input[@value='2']"));  
               specificRadio.click();  
               pause(2);  
       }                 
  
       @Test  
       public void select()  
       {  
               List<WebElement> allOptions = driver.findElements(By.tagName("option"));  
               for (WebElement option : allOptions)   
               {  
  
  
The presenter highlights radio. He then highlights findElements, and then highlights name("radios"));.  
  
The presenter clicks Run and the Input Items web page opens again.*   
  
Now, if I know which of these guys I want to go through, again, I don't have to load them all up and click them individually. I can actually do an XPath search, or other types of searches inside of there, and find exactly the one I want to click on and go and do that. In this case, I am doing an XPath search for the input element which has a value of 2. I'm mainly specifically looking for a radio button in this case, so I have to be careful of my XPath. I have to select it properly. But again, I can go select the specific item, and then click that. Now, once the radio buttons are clicked, they can't be unclicked. There is no option inside of there from our Java code. That's not something we can do. It works just like we do on the site. Now, the last type we are going to talk about here is the select boxes. And there's individual selects and multiple selects. So as I go and run this, it's going to go through and it's actually going to load up all the options. Now remember, the select is not an input tag, it's a separate tag. And the values of a select box is based off the option box. And so my first little run here actually goes through and selects every single one of them, and we're going to see that here in a second. So what it did here is it literally grabbed every single tag with an option, and on my page I have more than one drop-down list, I have more than one multiple List Box inside of here to choose from, so it literally clicked them all. That's not really what I wanted, so I can also get them all by the select itself. And within the select itself, I can take that web element and create a helper class, which is the select class. And then I can ask "hey, is this a multiple select?".   
*On the Input Items web page, radio buttons one, two, and three are automatically selected in turn.  
  
The presenter switches back to Eclipse.  
  
The presenter highlights the code "//input[@value='2']".  
  
The presenter then highlights specificRadio.click();.  
  
The presenter scrolls down. The following code is shown:  
  
}                 
  
       @Test  
       public void select()  
       {  
               List<WebElement> allOptions = driver.findElements(By.tagName("option"));  
               for (WebElement option : allOptions)   
               {  
                   System.out.println(String.format("Value is: %s", option.getAttribute("value")));  
                   option.click();  
               }  
                 
               List<WebElement> allSelects = driver.findElements(By.tagName("select"));  
               for (WebElement we : allSelects)   
               {  
                       Select select = new Select(we);  
                       if (select.isMultiple())  
                       {  
                               select.deselectAll();  
                       }  
                       select.selectByIndex(1);  
                       select.selectByVisibleText("Arizona");  
               }  
  
The presenter clicks Run to run the code.  
  
The Input Items web page opens and all the options in each of the list boxes are selected automatically.  
  
The presenter switches back to Eclipse and highlights various parts of the code. highlights the code  
  
tagName("option"));  
  
The presenter switches back to the browser and highlights various parts of the code. He highlights .tagName("select"), the line Select select = new Select(we); and the elements isMultiple and deselectAll.*  
  
If it is a multiple select, I can do options like deselecting all. Past that, if I have a select box, I can choose from the select box directly to pick which ones I want to. So I can, for instance, select it by index or select it by the text that you see. And so what we saw here is when it ran through, the index is one, and that's what it originally picked, Alaska, which is the second one – it starts with index 0. And then I picked some text outside there for Arizona. So I can do that either way; I can pick the index 0, 1, 2, 3, 4 up to the number that's in there, or I can say what text you've seen so…seen inside of here. Now, again, I don't want to select every single item that's out there from every single option box. And so inside of my test, I don't want to have to pick every one of those, so I can do a smarter selector. So in this case, I'm saying "select only the option boxes that live within this ID of statesWithValues". And in this item then, I can go through and I can actually – I mean, and clicking every one of the items in this case. Or I could just choose the select box by the item I wanted to. Now, if I did the select box by the item I wanted to, there's another option inside of here where I can say, we can see selectByValue. Now, I can't do selectByValue here, because if it doesn't have that value it would fail. And so in our one test here, we can see from down below, instead of the whole state name inside of here, the one by value gives us the actual short state name, the values inside of here.   
*In the code, the presenter highlights selectByIndex and then selectByVisibleText.  
  
The presenter switches to the web page and expands the drop-down list, which lists the names of American states in alphabetical order.  
  
The presenter switches back to Eclipse and scrolls to the following code:  
  
option.click();  
               }  
                 
               List<WebElement> allSelects = driver.findElements(By.tagName("select"));  
               for (WebElement we : allSelects)   
               {  
                       Select select = new Select(we);  
                       if (select.isMultiple())  
                       {  
                               select.deselectAll();  
                       }  
                       select.selectByIndex(1);  
                       select.selectByVisibleText("Arizona");  
               }  
  
               List<WebElement> allValueOptions = driver.findElements(By.cssSelector("#statesWithValues > option"));  
               for (WebElement option : allValueOptions)   
               {  
                   option.click();  
               }  
       }  
  
       @Test  
  
The presenter highlights cssSelector("#statesWithValues > option". The presenter then highlights statesWithValues, followed by option.click.  
  
The presenter selects selectByIndex and a drop-down list lists the options selectByIndex, selectByValue, and selectByVisibleText.  
  
In the Console pane, the following list displays:  
  
Value is: California  
Value is: Colorado  
Value is: AL  
Value is: AK  
Value is: AZ  
Value is: AR  
Value is: CA  
Value is: CO*   
  
All right, the last thing we're going to look at is submitting our form. And so when we go to submit our form, there's several options we can do inside of here. Now, the first option is we can go find the button. We can find a Reset button or a Submit button. So what our little test here is going to do…and actually, I need to come back here and close this to get our test to kick off. Our test is going to open this up, fill in some data, and hit the Reset button. It's going to clear out that item here. So you can see it's filling in some data, and it's going to hit the Reset button, and then it cleared it all out. And now it's filling in data again. It's going to hit the Submit button, and you can see it's submitted to a different location. And then it submits it one more time, and I want to show you that last submit. So I can actually go find the name of a Submit button, and again my Submit button was given the name submit – not very original, but that's what it was. And I can click on the Submit button, and that can engender that behavior. I can also, though, go to any input element. It could be a checkbox, it could be a text box, it could be any element that I find by name, and on that element I can hit Submit. And that's kind of like hitting the enter key, or something like that, that would kick off the browser. So these are all options for managing our input. You can use them to great effect to be able to create data, and submit forms as you need.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Clicking Links in Java

Learning Objective

*After completing this topic, you should be able to*

* *click links in Selenium Java tests*

**1. Testing links**

When creating our tests in Java, we still need to be able to find and click on links, even though we can't see the browser in order to execute and test those navigational flows. Finding links has several options and clicking them is pretty simple, once we've seen the basics. So, the most basic way to find a link is to search on all the elements By a tagName, because links generally come underneath an ("a") tag. And so, obviously, I can search on anything by ID, I can search on anything by any other criteria, you know, that I want to throw out there. But, if I'm looking specifically for links, an ("a") tag is a great way to see them. And so, once I've searched the ("a") tag – in this case, I'm getting all the links back. If I just did find a single element, just return the first link that it finds in the page regardless of order. Now, past that, there's other methods. I can go and search by link name or partialLinkText. So, when I search on a link name, that's saying I know exactly what's on the screen. I want to find the text that matches this on the screen, and then go and pull that element back – or a partialLinkText, as we can see right here, I'm doing my findElement call. Again, in this case, I'm looking for a single element; I'm looking for one element that has the word "contact" anywhere inside of that link text that one should see on the screen. Either one of these calls could also be done with the plural we see up here, by findElements. So, let me go ahead and execute this test and we can kind of see what's going on. So, the first thing we're doing here is we're printing out every link that's on the screen.   
*The file Clicking.java is open in Eclipse. In the file, the following code is visible:  
  
public class Clicking  
{  
       @Test  
       public void findingAndNavigating()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/content/imagemap.html");  
                 
               System.out.println("........... All the a tags ...................");  
               List<WebElement> links = driver.findElements(By.tagName("a"));  
               for (WebElement link : links)  
               {  
                       System.out.println(link.getText() + " " + link);  
               }  
  
               pause(15);  
                 
               System.out.println("........... Match the visible text exactly ...................");  
               WebElement link = driver.findElement(By.linkText("calendar"));  
               link.click();  
  
               pause(3);  
  
The presenter highlights By.tagName("a")). He then highlights <WebElement> links. Next he highlights find.Elements. The List<WebElement> org.openqa.selenium.WebDriver.findElements(By by) tooltip opens.  
  
The presenter scrolls down to the following code:  
  
driver.get("http://localhost:8080/website/content/imagemap.html");  
  
System.out.println("........... All the tags ..........\_\_;  
link.click();  
  
pause(3);  
  
System.out.println(".......... Match the visible text partially..........\*);  
WebElement partial = driver.findElement(by.partiallinkText("contact"));  
partial.clock();  
  
pause(3);  
  
The presenter highlights .linkText("calendar"));. He then highlights partiallinkText("contact"));. He then highlights findElement. Next he highlights ("contact"));. He then highlights each example of findElements that's displayed.  
  
The presenter clicks Run to run the code. The test runs in the background.  
  
The presenter highlights the line  
  
System.out.println(link.getText() + " " + link);  
  
The Console pane lists all a tags on the tested web page, as follows:  
  
home [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
news [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
calendar [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
contact us [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
login [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
no popups [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
[[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
 [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
 [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]  
 [[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]*   
  
So, now that we have this popped up, you can see we have home, news, calendar, contact us, login, popins, and then some other links down here which doesn't exactly make sense. So, if I look at the screen here, you can see **home**, **news**, **calendar**, **login**, **contact us**, **login**, popups. And so, now the test is continuing on and, you can see, it clicked on the **calendar** link, it clicked on the contact link, and now it's gone back to the main page. Because the last item inside of this test, was an image map. And so, inside of here, you can see we have an image map, and the test is going through – we'll look at the code here in a second – and clicking on each one of these colored areas. And, as it gets clicked on, behind the scenes there's some JavaScript that's telling us what we've just clicked on. So it's, you know, red, purple, green, yellow…it's clicking through these and cycling through them. Okay, so that's a quick run of this test. If we follow through what we just saw – again, we go get all the tags, all of them that are ("a") tags inside of there. And it's returning to us from that page; news, calendar, contact us, login, and no popups – which was the elements across the top. We can tell they're links, because we did a style sheet on those that made them change colors back and forth inside of there – you saw that.   
*The presenter switches to a web page titled Image Map. It contains a block made up of four differently colored squares. It also contains a menu with the links home, news, calendar, contact us, login, and no popups.  
  
As the test runs, the system clicks the calendar link, opening a page that displays a calendar. It then clicks the contact us link to open a page that provides contact details. The test then opens the Image Map page and each of the four squares is clicked. As a square is clicked, the name of its color displays above the block – red, green, purple, and yellow.  
  
The presenter switches back to Eclipse, where the following code is visible:  
  
WebDriver driver = new FirefoxDriver();  
driver.get("http://localhost:8080/website/contact/imagemap.html");  
  
System.out.println("........... All the a tags ..................."):  
List<WebElement> links = driver.findElements(By.tagName("a"));  
for (WebElement link : links)  
{  
    System.out.println(link.getText() + " " + link);  
}  
  
pause(15);  
  
System.out.println("............ Match the visible text exactly ..................");  
  
The presenter highlights the line  
  
driver.findElements(By.tagName("a"));  
  
The presenter then points out the links in the list of a tags in the Console pane – news, calendar, contact us, login, no popups.*   
  
And then, the rest of the links inside of there – there's four references inside of there, which have to do with the image map. The image map had those four items that allowed us to click red, green, purple, you know…and I think it was yellow was the fourth one inside of there – and go back and forth. And those are the four elements that came down here. Now notice, there was no text, so my printout here, as I'm printing the link text and then the link itself, the web element. And so, the first four of these guys…or five of these guys, had link text associated with it. The rest of these guys didn't, because they were part of the image map. There was no text associated with them, but you can still see all those details of those elements inside of there. All right, so that's finding the links inside of there. Clicking them, as we saw, is quite simple. Once we have the element, we just say click(). This is the same thing we did for a button. It's the same thing we'd do for any element on the screen that you can click on as to elicit the click response inside of there. It literally clicks as if a mouse was clicking on top of there. And so that's all we have to do to navigate. As far as the image map goes then; again, we can go and find it, but we can't find it by the same mechanism. So, if we looked it by the tag name…and you see there's very…it's not easy to distinguish between these four.   
*The presenter scrolls down in the Clicking.java file. The following code is shown:  
  
       pause(15);  
  
       System.out.println("............ Match the visible text exactly ..................");  
       WebElement link = driver.findElement(By.linktext("calendar"));  
       link.click();  
  
       pause(3);  
  
       System.out.println("............ Match the visible text partially ..................");  
       WebElement partial = driver.findElement(By.partiallinkText("contact"));  
       partial.click();  
  
The presenter highlights link.click();.   
  
The presenter scrolls down further to show the following code:  
  
       pause(3);  
  
       system.out.println("............ Image Maps ..................");  
       driver.get("http://localhost:8080/website/contact/imagemap.html");  
       WebElement imageMapElement;  
       String [] colors = {"red", "purple", "green", "yellow"};  
  
       for (int i = 0; i < 20; i++)  
       {  
            imageMapElement = driver.findElement(By.id(colors[i % 4]));  
            imageMapElement.click();  
            pause(1);  
       }  
       driver.quit();  
}  
  
private void pause (int seconds)  
  
In the Console pane, the presenter highlights the entry  
  
[[FirefoxDriver: firefox on XP (601ec733-149e-49d0-b822-5dba1fbc4dcd)] -> tag name: a]*   
  
So, the easiest way to deal with an image mapping as part of your testing, is when each element inside the image map has a separate ID associated with it. It's not necessarily required, but it's certainly helpful and something that you could probably ask for as part of the development, is to have each one of those to have an ID. So, I'm doing a findElement(By.id) in this case, and the id is this guy is "red", "green", "yellow", and "purple". And so, when I have the ID and I go search on that ID, I can pull that element back. And then you can see, I just do the click() command once again. So, whether you're dealing with an ("a") that's a link text, an ("a") that's an image, an ("a") that's an image map, anything that you can put an ("a") around, I can find that element and then I can click() on that element appropriately. But remember, in an image map, if I click in the whole thing, it's going to do the link for the whole thing – not the individual section. I actually have to go up and pick up that specific ID, and find the element by that ID in order to click on that link. So, there you go; now you can navigate and have fun testing.   
*The presenter highlights findElement in the code. He then highlights By.id. Next he highlights "red", "purple", "green", and "yellow".  
  
The presenter highlights the line  
  
imageMapElement.click();  
  
Finally the presenter highlights the code  
  
findElement(By.id(colors[i % 4]));  
imageMapElement.click();*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Navigation in Java

Learning Objective

*After completing this topic, you should be able to*

* *test navigation in Selenium Java projects*

**1. Navigating between pages**

A core part of browsing is the Navigation buttons for going back and forth between pages. With dynamic web sites, these buttons often cause as much trouble as they provide valuable user interaction. Thus, it becomes really important to test the site's behavior well, through the navigation buttons. Selenium gives us the ability to simulate these steps, and thus validating the behavior under the different circumstances that arise. So let me walk you through the flow we're going to test here. So we're going to log into the page. And then we're going to go over to a behavior. We're going to actually add an event into here. And then we're going to type in some stuff here – I'm just going to put garbage in for now. And we're going to make it recurring. And then, when we go through here, we're going to select it. And we're going to pick some recurrence here. And continue, come to this confirmation page, and then boom – we create the items. So generally speaking, this flow is what we're going to look at. Now what we want to see is, if I hit the **Back** button, you can see everything looks just fine. If I hit the **Forward** button, everything looks just fine. But how does the site behave within that? So it seems as though, based off of my first human run through this, everything behaves fine. But the question is, is will it behave fine every time? will it behave fine as functionality changes, and also will it behave fine as they go back and hit different buttons? will that behave the same way? Now, let me go ahead and delete these so I can go and start with a clean test. And everything is empty again.   
*In a browser window, a login page is open. It contains Username and Password text boxes, and a Login button. It also contains a menu with the links home, news, calendar, contact us, login, and no popups, and, to one side, the links Menu Bar Text, Another Menu Item, and Menu Bar option. The presenter clicks Login.  
  
The Administrator Home page opens. It includes the same menu options, except that login has changed to logout. It also contains a site navigation directory with the nodes Your Account, Admin Functions, Site Content, and Interactive Content.   
  
The presenter expands Site Content node, which contains the entries Manage News, Manage Calendar, Manage Bookmarks, Google Maps, Content Lists, Menus, and File Content. The presenter then expands the Manage Calendar node, which includes the entries View Events, Add Event, and Add Gathering.  
  
The presenter clicks Add Event. The Add a New Event page contains Title of Event, Date of Event, and Description text boxes. The presenter enters random text in the Title of Event and Description text boxes. He then accesses the Add Recurrance page, which contains six radio buttons – Multiple Day Event, Weekly Event, Monthly Event by Day of Month, Monthly Event on Relative Weeks, Yearly Event on Exact Day, and Yearly Event on Relative Day. The presenter selects Multiple Day Event and then accesses the Add Series Confirmation page, which lists three dates and contains two buttons – Edit Recurrance and Approve and Save.  
  
The presenter clicks Approve and Save, and the View Events for page opens. It lists details of three events, all with the event name that the presenter originally specified. It also contains Edit, Modify Schedule, and Delete buttons.  
  
The presenter navigates back and then forwards again, returning to the View Events for page. He selects the third listed event and clicks Delete.The Delete Event page lists all instances of the recurring event. It contains two radio buttons – Delete just this instance and Delete all Events – and Confirm Delete and Cancel buttons.  
  
The presenter selects the Delete all Events radio button and clicks Confirm Delete.  
  
On the View Events for page, no events are now listed.*   
  
So I'm going to show you this test now. The test is not terribly important, but you can see I go to the sign on page, I provide the username and password, and click that. And then I'm doing some navigation there to get to the add page. I'm going to go and add some information to the page, like you saw me type inside of there. And then, I get to my first point where I can use navigation. Now, within the driver, there's this navigate feature. So inside of the driver, I ask for navigate, and it gives me – inside of navigation – a bunch of options. So I can go back or forth through the buttons, or I can jump to a Refresh button, or just go straight to some other URL out there. So there's a lot of different options of what I can do, because, I mean, part of the navigation of your site – you have to understand is – you don't control it all. As a user, I can type in a URL at any given time. So all those different crazy combinations you might want to be able to try out, this navigate gives you the ability to try that. This, basically, works right around the browser capabilities you see, and goes to that top bar and the controls there. So inside of this, I'm going to go back and forth, okay. I'm just going to use the back button and the forward button.   
*The presenter switches to Eclipse, in which a file named BackAndForward.java is open. In the file, the following code is visible:  
  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;  
  
public class BackAndForward  
{  
       @Test  
       public void navigate()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/signon.vpd");  
                 
               // Login  
               driver.findElement(By.id("usernameInput")).clear();  
               driver.findElement(By.id("usernameInput")).sendKeys("admin");  
               driver.findElement(By.id("passwordInput")).clear();  
               driver.findElement(By.id("passwordInput")).sendKeys("test");  
               driver.findElement(By.name("submit")).click();  
                 
               // Go to the add page  
               driver.findElement(By.id("ygtvt23")).click();  
               driver.findElement(By.id("ygtvt28")).click();  
               driver.findElement(By.id("ygtvlabelel30")).click();  
  
               // Add the basic info  
               driver.findElement(By.id("titleInput")).clear();  
  
The presenter runs through various lines in the code and then scrolls down. The following code is shown: driver.findElement(By.id("passwordInput")).sendKeys("test");  
               driver.findElement(By.name("submit")).click();  
                 
               // Go to the add page  
               driver.findElement(By.id("ygtvt23")).click();  
               driver.findElement(By.id("ygtvt28")).click();  
               driver.findElement(By.id("ygtvlabelel30")).click();  
                 
               // Add the basic info  
               driver.findElement(By.id("titleInput")).clear();  
               driver.findElement(By.id("titleInput")).sendKeys("Back and Forth");  
               driver.findElement(By.name("description")).clear();  
               driver.findElement(By.name("description")).sendKeys("Test navigation");  
               driver.findElement(By.id("recurringInput")).click();  
               driver.findElement(By.name("buttonPressed")).click();  
  
               driver.navigate().back();  
               driver.navigate().forward();  
                 
               // Setup recurrence  
               driver.findElement(By.name("recurring.type")).click();  
               driver.findElement(By.name("recurring.endCriteria")).click();  
  
The presenter highlights the code driver.navigate().back and a drop-down list lists options such as back() : void - Navigation, forward(): void - Navigation, refresh(): void - Navigation, and to(URL url): void - Navigation.*   
  
After that, I'm going to go through and set the recurrence information. And then, again, I'm going to hit the **Submit** button. Then I'm going to go back, and I'm going to go forward again. And then I'm going to finally go to the confirmation page and get there, and then I'm going to click that **Confirmation** button. And here is the real test; I'm going to go back one more time, and then I'm going to click the **Confirmation** button again. And so then, I get to see what happens. Now the worst-case scenario would be it double adds those values, it's going to add them a second time, which it really shouldn't do. But if it doesn't do that, that's good. Hopefully it goes to a kind error page. But at least if it goes to an error page, it'll end up along those lines, and we can get some better feel for my test, from our test. So if I go and run this test then, it's going to pop this guy up. It takes a second for this to come over. And here comes our test, and again it's logging in, it's pressing the button, it's clicking it, it's going back and forward, clicking it. There, back and forth again, going back…there you go. And when it hit the button the last time, you get to the page "Oops! it looks like something broke". And so this is just a generic error page, you can see there's some error messages not entirely filled out inside of there. But that's okay. The important thing here is if I come back to my website, and I go to the main page here, and let me go to the admin home…my navigation is messed up.   
*The presenter highlights the following code:  
driver.findElement(By.name("recurring.type")).click();  
               driver.findElement(By.name("recurring.endCriteria")).click();  
               driver.findElement(By.name("recurring.numberInSeries")).clear();  
               driver.findElement(By.name("recurring.numberInSeries")).sendKeys("4");  
               driver.findElement(By.name("buttonPressed")).click();  
  
The presenter highlights driver.navigate().back();. He highlights buttonPressed. He then highlights driver.navigate().forward();.  
  
The presenter scrolls down and the following code is displayed: driver.findElement(By.name("buttonPressed")).click();  
  
               driver.navigate().back();  
               driver.navigate().forward();  
                 
               // Setup recurrence  
               driver.findElement(By.name("recurring.type")).click();  
               driver.findElement(By.name("recurring.endCriteria")).click();  
               driver.findElement(By.name("recurring.numberInSeries")).clear();  
               driver.findElement(By.name("recurring.numberInSeries")).sendKeys("4");  
               driver.findElement(By.name("buttonPressed")).click();  
  
               driver.navigate().back();  
               driver.navigate().forward();  
                 
               // Confirm  
               driver.findElement(By.xpath("//input[2]")).click();  
  
               // Go back and submit again see what happens  
               driver.navigate().back();  
               driver.findElement(By.xpath("//input[2]")).click();               
       }  
}  
  
The presenter highlights driver.findElement(By.xpath("//input[2]")).click();. The presenter highlights driver.navigate().back();. Then the presenter highlights the lines  
  
       driver.navigate().back();  
               driver.findElement(By.xpath("//input[2]")).click();  
  
The presenter runs the test, a browser opens, and the test encounters an error. A page displays the following text:  
  
Oops! It looks like something is broke. We are keeping track of it though, so we will try to fix the problem soon. If need more help immediately, use (cut off).  
  
Please correct the following errorson\_US-errors.prefixAn unknown error has occurred, you may try your task again now, but if the error (cut off).   
  
The presenter changes the URL in the address bar to localhost:8080/website/home.vpd and the Administrator Home page opens.*   
  
So actually, since that error page worked up I've another bug in the site, a good thing it's just a little test site inside of there. So I can come back here, and I'll go back through the sign on page. And then we can go to the page again, and we can go and double check and make sure. And again, we could do all those programmatically as well, if we took the time and viewed it all. But you can see, we only have one set. We added four items inside of there, which is what I expected inside of there. If I go through and I delete these, and I delete all the events, delete them all there, just added it one time. So now adding in the ability to use the navigate feature inside of the driver, I can test all sorts of crazy user interactions, both for positive interactions and also for negative interactions. Things users should be able to do and things we should be preventing them from doing, should all be part of my testing. And so I can add that to my test suite, and really ensure the behavior of my site is really exactly what I want it to be.   
*The presenter switches to Eclipse and then accesses the login page in the browser. The presenter enters a username and password, and clicks Login. The Administrator Home page opens. The presenter expands the Site Content node and below that, expands Manage Calendar, which contains the options View Events, Add Event, and Add Gathering.  
  
The presenter clicks View Events and the View Events for page lists the dates for four events. The presenter selects 05/22/2014 and clicks Delete. The Delete Event page opens. The presenter selects Delete all Events and clicks Confirm Delete. The View Events for page no longer lists any events.  
  
The presenter switches back to Eclipse and highlights the code  
  
driver.navigate().back();  
               driver.navigate().forward();*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Display Items in Java

Learning Objective

*After completing this topic, you should be able to*

* *validate display elements in Selenium Java projects*

**1. Confirming display setup**

Automated testing may mean moving our tests from eyes to electrons but it doesn't mean that we have to abandon the validation of visual elements. The Selenium framework for Java still allows us to inspect HTML elements that drive the display. So we can check that a page should look correct as it does in the browser. Let's see how. So our test here opens up a text box and we're going to send some text over to the text box and press a button. Now, as we run this test we're going to see after it sends the text and hits this button, that text box is going to disappear. It's going to come back after a second. So going through and running our test here, you can see our… It's disappeared and then it comes back and the text changes as it comes back. So when it does that, it's going to go through and it's going to – have the other test run, I'll get that out of the way. It's going to come back and we're going to assert in here that the text that we sent it's the text that's actually in the box. Now, the reason I do this assert right here is I hit the button and at this point, the display disappears. It's no longer visible, but we can still interact with it. We can still see it. We can still get elements from it and what not. So just because we can see text, doesn't mean that the user can see text. Because I can actually go out here as well and I can get the CSS value for either the display or the visibility. In this case, both of them are set to be not visible: none and hidden. And we saw that when the test ran, that box disappeared. But at this point, I still was out looking for it.   
*In Eclipse, a file named CheckingStyle.java is open. The following code is visible:  
driver.get("http://localhost:8080/website/pages/visible.html");  
                 
               WebElement hideMe = driver.findElement(By.id("hiddenToStart"));  
               hideMe.clear();  
               String testMessage = "Test visibility";  
               hideMe.sendKeys(testMessage);  
                 
               WebElement button = driver.findElement(By.id("makeInvisible"));  
               button.click();  
                 
               // I can get info no matter what!  
               assertEquals("This is the value I typed", testMessage, hideMe.getAttribute("value"));  
                 
               // Show the box went away  
               assertEquals("It is no longer displayed", "none", hideMe.getCssValue("display"));  
               assertEquals("It is no longer displayed", "hidden", hideMe.getCssValue("visibility"));  
                 
               WebDriverWait waiter = new WebDriverWait(driver, 3);   
               waiter.until(new ExpectedCondition<Boolean>()   
                       {  
                               public Boolean apply(WebDriver driver)   
                               {  
  
The presenter highlights the lines  
  
WebElement button = driver.findElement(By.id("makeInvisible"));  
button.click  
  
The presenter clicks Run to run the code.  
  
A web page titled Visibility opens. It contains Show Hidden and Hide buttons.  
  
As the test runs, a text box appears alongside the buttons, with the entry "I'm Back!".  
  
The presenter switches back to Eclipse. He highlights the code  
  
"This is the value I typed", testMessage, hideMe.getAttribute("value"));  
  
He then highlights the lines  
  
WebElement button = driver.findElement(By.id("makeInvisible"));  
button.click();  
  
Next the presenter highlights the code  
  
// I can get info no matter what!  
assertEquals("This is the value I typed", testMessage, hideMe.getAttribute("value"));*   
  
So we have to be aware in our testing, when things are no longer visible to the user. The test might report a success and it's not a successful test, in that the user couldn't see that information because of maybe something was rendered wrong on the page or something like that. So the next thing we can show inside of here is I can use my wait state to have a little test here, where my test is going to wait until the display value again. We're going back and getting that element. And the display value of that element is no longer going to be none. So right here, we can see that we validated that it is none and our test is going to delay and wait until it's no longer none. And at that point then I can go through and assert that the value that is coming back is the "I'm Back!" value – the last value we saw inside of there. And so this is just a really basic test but to show you that using the element to go off and either get an attribute; or a CssValue from the element that you went and found, went and searched on, allows you to really dive into the display elements of that item on the page. And in this case an input box but it could be a div. It could be anything. Because we use CSS for the most part to make things visible or invisible. There are some HTML attributes as well we can use and we can check those using the getAttribute. But we can combine those and check anything we want to on the page.   
*The presenter highlights hideMe.getCssValue("display"). Next he highlights hideMe.getCssValue("visibility").  
  
The presenter highlights "none", and then highlights "hidden".  
  
Next the presenter again highlights:   
  
// I can get info no matter what!  
assertEquals  
  
The presenter scrolls down to show the following code:  
       return ! driver.findElement(By.id("hiddenToStart")).getCssValue("display").equalsIgnoreCase("none");  
                       }  
               });  
                 
       assertEquals("This is the value after it reappears", "I'm Back!", hideMe.getAttribute("value"));  
}         
@Test  
public void beautify()  
  
The presenter highlights:  
  
WebDriverWait waiter = new WebDriverWait(driver, 3);   
waiter.until(new ExpectedCondition<Boolean>()    
  
The presenter then highlights   
  
return ! driver.findElement(By.id("hiddenToStart")).getCssValue("display").equalsIgnoreCase("none");  
  
The presenter highlights findElement, and then highlights getCssValue.  
  
The presenter highlights "non", hideMe.getCssValue("display"). He then highlights .getCssValue("display").equals.  
  
The presenter then highlights "This is the value after it reappears". Finally he highlights "I'm back!".*   
  
In this first test, we're looking for visibility. In the second test down here, we can actually just check straight look and feel on the page. So in this test, we go through and we are going to go back to our login page, we are going to put an invalid username and password inside of there and hit the button. Now, when the block comes back it's going to come back hopefully with the className("errors") on it. It's going to be whatever that style is inside of there, whatever that error style is going on inside of there. Now, in some cases, you might have a dynamically rendered stylesheet. For the most stylesheets are basically hardcoded and they are a static item. But sometimes you can randomly or not randomly, but dynamically, generate those or at least dynamically generate which style is being attributed to it. And so within that, I can go through and look at individual CssValues as part of the stylesheet. So when we run this test, it's going to go through, it's going to ask that errorBlock, the element we got inside of here. And here is the div tag in this case to go get certain CssValues outside of there – just to show you ones that we can look at. And there is a whole list of them right here that I'm going to look at off this block. So let's go and run this second test right here. So let me flick the test and run it. And as it runs, it's going to pop up the screen; it's going to have the errorBlock show up. See there it is, just for a second, popped and disappeared, but we can see the background color of that is this RGB value. It's going to be that pinkish red value. The color on the screen is basically black, the font weight is 700. So we can see it's a bolder font inside of here.   
*The presenter scrolls down to show the following code:  
  
@Test  
       public void beautify()  
       {  
               driver.get("http://localhost:8080/website/signon.vpd");  
  
               WebElement username = driver.findElement(By.id("usernameInput"));  
               username.sendKeys("admin");  
                 
               WebElement password = driver.findElement(By.id("passwordInput"));  
               password.sendKeys("junk");  
                 
               WebElement button = driver.findElement(By.name("submit"));  
               button.click();  
                 
               WebElement errorBlock = driver.findElement(By.className("errors"));  
               String[] cssValues = {"background-color", "color", "font-weight", "border", "border-top-color", border-bottom-color"};  
               for (String value : cssValues)  
               {  
                       System.out.println(value + " = " + errorBlock.getCssValue(value));  
               }  
                 
               driver.quit();  
       }  
  
The presenter highlights the lines  
  
driver.findElement(By.id("usernameInput"));  
username.sendKeys("admin");  
  
The presenter highlights the lines  
  
driver.findElement(By.id("passwordInput"));  
password.sendKeys("junk");  
  
The presenter then highlights .className("errors"), followed by the lines  
  
errorBlock = driver.findElement(By.className("errors")));  
  
Next he highlights getCassValue and the word beautify.   
  
The presenter clicks Run to run the code. The following results are listed in the Console pane:  
  
background-color = rgba(255, 175, 175, 1)  
color = rgba(0, 0, 0, 1)  
font-weight = 700  
border =  
border-top-color = rgba(255, 0, 0, 1)  
border-bottom-color = rgba(255, 0, 0, 1)*   
  
If we search for something like border, which is a compound item, it's not going to actually give us any details. It's not going to give us the summary item. We actually have to look at each individual item inside of there. And this is an important thing: if I wanted to validate that the border is applied, I've got to look at the top bottom left and right border individually. Because even though in CSS I can apply it to all four at the same time, inside of our Selenium tests, we look at them individually. So you can see each of these borders is straight red – 255 is just straight red. And 1 is the opacity value inside of there, really, really dark. So this little feature here, it's not really testing anything at this point, but it shows you what you could be testing. You could validate that, hey, this style has been applied. Hey, inside the style, it's a red color, it's a pink color, it's bolded, it's whatever you want to do. You can check those visual elements that are being rendered in CSS even through our testing in Selenium. And so this gives us another aspect, another wrinkle we can do within our testing to check the visibility elements that normally, you'd have to check with your eyes. But we know we can check because it should be applied within our CSS, within our HTML inside of our web pages.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Screenshots in Java

Learning Objective

*After completing this topic, you should be able to*

* *take a remote screenshot in a Selenium Java project*

**1. Using screenshots**

While we can do a lot to automate and reduce the amount of work for humans through Selenium, we still need sites to look good to people. To that end, Selenium gives us the ability to capture exactly what the screen looks like, during selected moments in our testing. We can take a screenshot, then Java can take over and save it, and do what we will with it. So as an example of that, we start off with just a little debug test case. So we have our test case that's going to go off, and this is going to be dragging an element around the screen. And so we have our worker that goes in, and it clicks on an item, and it moves it some distance. So we want to take a screenshot of what it looks like, because maybe it happens really quick, or maybe we have a series of moves or whatever. And so the code to do this is very, very simple. There is a separate interface that we can cast our driver to. Now, not every driver allows this to happen. The FirefoxDriver will let you take a screenshot, the HtmlUnitDriver won't – because it doesn't render anything. So some of the drivers will, and some of the drivers won't allow this. But if it does, we can go ahead and take that screenshot from it. And so from the screenshot driver – we're naming a camera here as a variable – we can ask it to say get a ScreenshotAs. And there's several formats we can grab it as; we can grab it as a file, or we can do it bytes, or 64 bit bytes inside of there.   
*A file named Screenshot.java is open in Eclipse. The following code is shown:  
  
{  
       @Test  
       public void positioning() throws IOException  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/drag.html");  
                 
               WebElement dragger = driver.findElement(By.id("drag1"));  
                 
               Actions worker = new Actions(driver);   
               worker.clickAndHold(dragger);  
               worker.moveByOffset(10, -150);  
               worker.release();  
               worker.perform();  
  
               TakesScreenshot camera = (TakesScreenshot) driver;  
               byte[] imageBytes = camera.getScreenshotAs(OutputType.BYTES);  
                 
               BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream("align.png"));  
               out.write(imageBytes);  
               out.close();  
                 
               driver.quit();  
  
The presenter highlights the line  
  
WebElement dragger = driver.findElement(By.id("drag1"));  
  
He then highlights clickAndHold(dragger);, followed by moveByOffset(10, -150);.  
  
The presenter highlights TakesScreenshot camera and the line  
  
get("http://localhost:8080/website/pages/drag.html")  
  
The presenter highlights  
  
WebDriver driver = new FirefoxDriver();  
  
He highlights  
  
camera = (TakesScreenshot) driver;  
  
Next the presenter highlights getsScreenshotAs.  
  
The presenter hovers over BYTES and a drop-down list lists the options BASE64, BYTES, class, and File.*   
  
And so for this little application, we're going to grab it as Bytes, because in Java we can deal with the bytes very easily. Instead of having to create a file, and move it and copy it, or whatever, we're going to just go ahead and grab the bytes. And then we can use our BufferedOutputStream to create a local file here and be able to store it. And I can create this file wherever I want to, that's the beauty of it here. If I create a file through the output type of file, it's going to put it in a temporary directory. Here I can put it wherever I want to, and save it as appropriately. So my bytes, I can just write it all at once, and then close it. Now, I'm not paying much attention to exception handling, or things like that, I'm just kind of throwing the error. But you get the picture of how easy it is to be able to capture, and then save a screenshot. So if I go and run this test real quick, I can go through, and let me select the specific test and run it. It's going to go through, it's going to pop up the screen, it's going to move stuff around, it's going to get the results there, and we don't even need to see it. I mean, it actually popped up, we just didn't even see it right here. It was just not something visible. Because we can come back and we can see, we set align.png, we have that file right here, and poof, there you go.   
*The presenter highlights the code  
  
new BufferedOutputStream(new FileOutputStream("align.png")  
  
He then highlights out.write(imageBytes);, followed by out.close();.  
  
The presenter scrolls up and right-clicks positioning(), and a menu lists options such as Open Declaration, Open Type Hierarchy, Quick Outline, and Cut and Copy. The presenter closes the menu.  
  
The presenter clicks Run to run the code.  
  
In a browser window, a web page contains two empty panes and a block that consists of four squares, each a different color. The block has been moved so that it overlaps the pane on the left.*   
  
Now, within this test, we can see is this where I wanted to drag it. The assumption is the mouse is probably up here in the corner. If I wanted it to be lower, or left or right, I can move it as appropriate. I can adjust it. I can use this to debug, and I can move this around. So instead of moving a 10 over, maybe I only wanted to move it 5 over, because I want it to be closer inside of there, and what not. So again, this gives me a visual feel. I don't have to worry about catching it really quick as a quick flash as it comes up. I don't have to worry about it step-by-step. I can actually catch that screenshot perfectly each time. And so now I can come back and look at this and say "oh, see that's perfect, right on the line. That's what I wanted it to be inside of there", if that's, perhaps, what I want in this test. Now, I can do different than that here, that's more of debug. Here is another chance here, where in this case, I've actually created a little helper method. It does all the same stuff – if I look at this helper method, it grabs images, it saves it as a file. I'm just allowing myself to be able to pass in the camera and the name of the file, just to make it nice and easy. So I just have a takeScreenshot helper method inside of here. So in this test, I want to go through and I want to try and log in. So at each step of the log in, I want to capture what the screen looks like. Because maybe in my testing, I really want a human to review this and make sure the screen looks right, for whatever my motivation would be during this test. So I can run this test, and it's going to open up, and it's going to pop through a series right here, and it's going to happen really quick. I may or may not watch the test running, this could be running on a server each day. And so maybe what's happening is on the server, it's going to go through. It's going to run, it's going to see what's happening, and I can see the net results.   
*The presenter returns to Eclipse and changes the value 10 to 5 in the line  
  
worker.moveByOffset(5, -150);  
  
The presenter then clicks Run to run the code.  
  
The presenter briefly switches to the browser, where part of the page is visible.  
  
The presenter switches back to Eclipse and scrolls down to show the code  
  
               driver.quit();  
       }  
  
       @Test  
       public void doubleCheck() throws IOException  
       {  
               WebDriver driver = new FirefoxDriver();  
               TakesScreenshot camera = (TakesScreenshot) driver;  
  
               driver.get("http://localhost:8080/website/signon.vpd");  
               WebElement submit = driver.findElement(By.name("submit"));  
               submit.click();  
  
               takeScreenshot("noEntry.png", camera);  
                 
               WebElement username = driver.findElement(By.id("usernameInput"));  
               username.sendKeys("admin");  
                 
               WebElement password = driver.findElement(By.id("passwordInput"));  
               password.sendKeys("junk");  
                 
               submit = driver.findElement(By.name("submit"));  
               submit.click();  
  
               takeScreenshot("invalid.png", camera);  
  
The presenter highlights doubleCheck.  
  
The presenter scrolls down and highlights the line  
  
takeScreenshot("noEntry.png", camera);.  
  
The presenter scrolls down to show the following code:  
  
submit.click();  
  
               WebElement header = driver.findElement(By.tagName("h2"));  
               if (! "Administrator Home".equals(header.getText()))  
               {  
                       takeScreenshot("failedLogin.png", camera);  
                       fail("Did not login");  
               }  
                 
               driver.quit();  
       }  
         
         
  
       private void takeScreenshot(String name, TakesScreenshot camera) throws IOException  
       {  
               byte[] imageBytes = camera.getScreenshotAs(OutputType.BYTES);  
                 
               BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(name));  
               out.write(imageBytes);  
               out.close();  
       }  
}  
  
The presenter highlights the line  
  
BufferedOutputStream out = new BufferedOutputStream  
  
The presenter then highlights String name.  
  
The presenter scrolls up to show the following code:  
  
driver.quit();  
       }  
  
       @Test  
       public void doubleCheck() throws IOException  
       {  
               WebDriver driver = new FirefoxDriver();  
               TakesScreenshot camera = (TakesScreenshot) driver;  
  
               driver.get("http://localhost:8080/website/signon.vpd");  
               WebElement submit = driver.findElement(By.name("submit"));  
               submit.click();  
  
               takeScreenshot("noEntry.png", camera);  
                 
               WebElement username = driver.findElement(By.id("usernameInput"));  
               username.sendKeys("admin");  
                 
               WebElement password = driver.findElement(By.id("passwordInput"));  
               password.sendKeys("junk");  
                 
               submit = driver.findElement(By.name("submit"));  
               submit.click();  
  
The presenter highlights the line  
  
takeScreenshot("noEntry.png", camera);  
  
He then highlights the line  
  
driver.get("http://localhost:8080/website/signon.vpd")  
  
The presenter highlights doubleCheck. He then scrolls down to the following code:  
  
WebElement password = driver.findElement(By.id("passwordInput"));  
               password.sendKeys("junk");  
                 
               submit = driver.findElement(By.name("submit"));  
               submit.click();  
  
               takeScreenshot("invalid.png", camera);  
  
               username = driver.findElement(By.id("usernameInput"));  
               username.sendKeys("admin");  
                 
               password = driver.findElement(By.id("passwordInput"));  
               password.sendKeys("test");  
                 
               submit = driver.findElement(By.name("submit"));  
               submit.click();  
  
               takeScreenshot("success.png", camera);  
                 
               driver.quit();  
       }  
         
       @Test*   
  
Now, in this case specifically if I go look at these tests, the first one outside of here is the noEntry. So you can see…beautiful. So I put noEntry in, I get a username password required, it worked perfectly. I can do a quick validation; I can check day by day, as I choose, to say if it's looking right. A quick look, I've executed it, and I can probably just scroll through these images. You can see, I can go through it here. You look at this one though, and when I put in an invalid username password, I have admin, admin, and then probably the password in there, twice. So in this case, by this quick look at the screenshot, I could see "gosh, there's something wrong on my test". I didn't mean to type this, right. I probably forgot to put a clear inside of there. So each one of these pictures will give us a different feel…and we're just scrolling through all of them now…of what's going on inside of here. And so, you know, in this…sorry, in this case here, I have invalid username, password, and so on and so forth. So I can use that. I can also use that specifically on a special purpose. So this last test case here, we can see I'm again logging in.   
*The presenter switches to the browser, in which a login page is open. On the page, the following error messages display in a red pane:  
  
Username is required.  
Password is required.  
  
The presenter enters the username adminadminand a password. The message "Invalid Username/Password, Try again..." appears.  
  
The presenter switches to another browser window, in which the web page with two panes and a multicolored block is open.  
  
The presenter switches to Eclipse and scrolls down to show the following code:  
  
{  
               WebDriver driver = new FirefoxDriver();  
               TakesScreenshot camera = (TakesScreenshot) driver;  
  
               driver.get("http://localhost:8080/website/signon.vpd");  
               WebElement submit = driver.findElement(By.name("submit"));  
               WebElement username = driver.findElement(By.id("usernameInput"));  
               username.sendKeys("admin");  
                 
               WebElement password = driver.findElement(By.id("passwordInput"));  
               password.sendKeys("junk");  
                 
               submit.click();  
  
               WebElement header = driver.findElement(By.tagName("h2"));  
               if (! "Administrator Home".equals(header.getText()))  
               {  
                       takeScreenshot("failedLogin.png", camera);  
                       fail("Did not login");  
               }  
                 
               driver.quit();  
       }*   
  
I'm going to go through and hit username and password, and my password is going to be wrong. It's going to be an invalid login. And so what I can do is, I can choose, I can do my check here, and if my check doesn't pass, at that point I can choose to take a screenshot. So maybe if my test runs fine, I have no screenshots that are…that show up for this. But if my test runs poorly, then I will take a screenshot. So again, I can go off and I can run this test. It's going to attempt to login. It's going to pop it up, it's going to go, it's going to do its thing. And then, since it's going to fail there, it can go off and choose to take a screenshot. So again, my test failed, so there should be a screenshot inside of here. Let me go ahead and refresh this. And there's a screenshot that would be generated from this. And actually, I'm not seeing the screenshot here, but you get the idea. The screenshot would be taken as a choice. I can choose to take a screenshot. So use this capability to really put that human element back into our automated testing. And even though we're driving this in order to be able to get it to go faster, and quicker, and automated, we can still have humans be able to see what's going on inside of our site.   
*In Eclipse, the presenter highlights "junk" in the line  
  
password.sendKeys("junk");  
  
He highlights submit.click() and then the line  
  
! "Administrator Home".equals(header.getText)  
  
The presenter then highlights the code  
  
takeScreenshot("failedLogin.npg", camera)  
  
The presenter scrolls up to the following code:  
  
@Test  
       public void onFail() throws IOException  
       {  
               WebDriver driver = new FirefoxDriver();  
               TakesScreenshot camera = (TakesScreenshot) driver;  
  
               driver.get("http://localhost:8080/website/signon.vpd");  
               WebElement submit = driver.findElement(By.name("submit"));  
               WebElement username = driver.findElement(By.id("usernameInput"));  
               username.sendKeys("admin");  
                 
               WebElement password = driver.findElement(By.id("passwordInput"));  
               password.sendKeys("junk");  
                 
               submit.click();  
  
               WebElement header = driver.findElement(By.tagName("h2"));  
               if (! "Administrator Home".equals(header.getText()))  
               {  
                       takeScreenshot("failedLogin.png", camera);  
                       fail("Did not login");  
               }  
  
The presenter highlights onFail and then clicks Run to run the code.  
  
Finally the presenter highlights the line  
  
takesScreenshot("failedLogin.png", camera);*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

JavaScript from Java

Learning Objective

*After completing this topic, you should be able to*

* *execute JavaScript from Selenium Java tests*

**1. Executing JavaScript from the browser**

Selenium gives us an even more powerful tool for completing test cases, by allowing us to execute JavaScript on the browser. Whether to gather data; manipulate the state of the page, this capability is available for enhancing our Java test cases with…inside our calls on that page. So, in this call here, we have a really basic little screen; it's going to allow us to go to an element and change the field value. So, we're going to, in this test, look at the field value of a input box – and the input box is given the ID "hiddenToStart". We're going to look that up and see the attribute of that. And then we're going to use the executor here, and it's just an object of the type JavaScriptExecutor. Now, it's different than WebDriver, because not all WebDrivers allow you to execute JavaScript. For instance, the htmlUnit WebDriver will not allow you to actually execute JavaScript. So, we're going to use the FirefoxDriver() here, and we're going to cast that over to this special interface. And, we have these two that are really the same object – but two looks at the object we're going to be working with. So, when we get to the executor, we're going to execute a script that allows us to take any JavaScript statement and execute it inside the browser. So, we're going to our document, getting – through the DOM – an element by its ID, and we're changing that value to this value; change to Java here. And so we're going to see the value; it's going to change from its original value to this new value as we execute this test. So, I'll let that go off and **Run**.   
*The file ScriptTest.java is open in Eclipse and the following code is shown:  
  
import org.junit.Test;  
import org.openqa.selenium.By;  
import org.openqa.selenium.JavascriptExecutor;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;  
  
public class ScriptTest  
{  
       @Test  
       public void callInternalJavascript()  
       {  
               WebDriver driver = new FirefoxDriver();  
               JavascriptExecutor executor = (JavascriptExecutor) driver;  
  
               driver.get("http://localhost:8080/website/pages/visible.html");  
                 
               System.out.println("Before internal call " + driver.findElement(By.id("hiddenToStart")).getAttribute("value"));  
                 
               executor.executeScript("document.getElementById('hiddenToStart').value = \"changed from Java\";");  
  
               System.out.println("After internal call " + driver.findElement(By.id("hiddenToStart")).getAttribute("value"));  
               driver.close();  
       }  
  
       @Test  
       public void callRemoteJavascript()  
       {*  
  
Now, this again allows me to execute absolutely any JavaScript out there. It's going to allow that to be changed and update that. And so this first test goes through; as you can see, it goes from a value of Show me and it changes the value to be changed from Java. And that's the internal call right here. Now, this is not limited to any JavaScript that I have to write inside of my Java app. My Java app can also choose to execute existing scripts on the other ones. If I bring up the page that we're looking at here, I can look at the Page Source inside of here. And, we can see, from the page source, that we actually have a number of JavaScript functions that are already defined in the page. This one we call the remoteCall. So, I can go inside of Java and have the same type of tests here but, when I execute the script, I can execute a remoteCall on there. Now, even better, I can execute that remoteCall with parameters. So, I can send in to this remoteCall – that we saw before…accepting a parameter here, which is some source of who's ever's making this change. So, it's going to print out the value "changed " plus whatever parameter gets sent in to it. And so I'm letting it know hey, I'm Java, I making the change for you.   
*In Eclipse, the Console pane lists the following results:  
  
Before internal call Show me  
After internal call changed from Java  
Before external call Show me  
After external call changed Java.  
  
In the code window, the presenter highlights the code  
  
"document.getElementById('hiddenToStart')  
  
The presenter then clicks Run to run the code.  
  
In a browser window, a web page titled Visibility opens. It contains a text box labeled changed Java and two buttons – Show Hidden and Hide.  
  
The presenter right-clicks the page and selects View Page Source to open the source code for the page. The following code is visible in the source file:  
  
function makeVisible()  
{   
       document.getElementById('hiddenToStart').style.visibility = 'visib (code cut off)  
       document.getElementById('hiddenToStart').style.display = '';  
       clearInterval(oneTime);  
}  
  
function makeInvisible()  
{  
       document.getElementById('hiddenToStart').style.display = 'none';  
       document.getElementById('hiddentToStart').style.visibility = 'hidde (code cut off)  
       oneTime = setinterval(function()(makeVisible();), 2000);  
)  
  
function remoteCall(source)  
(  
       document.getElementById('hiddenToStart').value = "changed " + sour (code cut off)  
}  
  
</script>  
  
The presenter highlights remoteCall(source).  
  
The presenter switches to Eclipse. The following code is shown:  
  
JavascriptExecutor executor = (JavascriptExecutor) driver;  
  
               driver.get("http://localhost:8080/website/pages/visible.html");  
                 
               System.out.println("Before internal call " + driver.findElement(By.id("hiddenToStart")).getAttribute("value"));  
                 
               executor.executeScript("document.getElementById('hiddenToStart').value = \"changed from Java\";");  
  
               System.out.println("After internal call " + driver.findElement(By.id("hiddenToStart")).getAttribute("value"));  
               driver.close();  
       }  
  
       @Test  
       public void callRemoteJavascript()  
       {  
               WebDriver driver = new FirefoxDriver();  
               JavascriptExecutor executor = (JavascriptExecutor) driver;  
  
               driver.get("http://localhost:8080/website/pages/visible.html");  
                 
               System.out.println("Before external call " + driver.findElement(By.id("hiddenToStart")).getAttribute("value"));  
               executor.executeScript("remoteCall(arguments[0])", "Java");  
                 
               System.out.println("After external call " + driver.findElement(By.id("hiddenToStart")).getAttribute("value"));  
               driver.close();  
       }  
}  
  
The presenter highlights remoteCall.  
  
The presenter switches to the source code for the web page and highlights the words source and changed.  
  
The presenter returns to Eclipse.*   
  
And so, if we look at this next call you can see, beforehand, it had the value of Show me and, then afterwards, it says to…changed Java. A little bit different from changed from Java – that was the first test case, where we had the code right inside of here – the JavaScript's code inside of our Java class or Java test here. The second one took a little bit of Java, mixed it with some JavaScript, executed that on the browser, and was able to get the test results. Now, in this case, we're definitely tweaking the state…we're changing the value out there. But there's nothing to say we couldn't use the DOM out there – or use a method that's being called out there – to be able to pull back the execution. So, maybe we have a browser that has a JavaScript method that needs to execute…that changes the display on the screen. Well, I could also call that method and get those results in my Java application and compare the two results. And so, I could go through and look at the results on the screen, compare the results from that JavaScript call, and then build my test cases dynamically. There's a lot of strategies for this. But, just know, you can actually get into that browser and execute tests on that browser from your Java application.   
*In Eclipse, the presenter highlights the code  
  
"Before external call " + driver.findElement(By.id("hiddenToStart")).getAttribute("value"));  
executor.executeScript("remoteCall(arguments[0])", "Java");  
  
Next the presenter highlights changed from Java.   
  
The presenter switches to the Console pane and highlights changed Java.  
  
In the code window, he highlights document.getElement.  
  
Finally the presenter highlights the code  
  
executeScript("document.getElementById('hiddenToStart').value = \"changed from Java\";");  
  
System.out.println("After*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Ajax elements in Java

Learning Objective

*After completing this topic, you should be able to*

* *work with Ajax elements in Selenium Java tests*

**1. Ajax elements and Selenium Java tests**

Ajax enables our sites to now load part of the page, while keeping the original page in place. This is great for humans, but requires our automated tests to be aware of these pauses. Selenium, though, provides us with tools to help our Java code look for when the pages are getting ready to continue on with its testing. To show you what we mean inside of here, we're going to run this first test. The test seems pretty straightforward; I'm going to go to a box, I'm going to click on an element inside of the box, I'm going to type some text inside of there, and hit the **Update** button. Except, instead of getting results right away, it doesn't come back for a little bit of time. So if I look at the test, go back to the results here, the test failed – because, again, it went and found an element. It went…that's a button that's inside of there. It went and found the entry it's going to put inside of there, add a message. It clears out the box and it sends the keys over there, and it clicks the button. Then it goes off and it finds the element for the results. And then it goes off and does a check there to say "hey, by the way, is the results what I thought it was going to be?" – does the results contain the message, basically, is what we are checking in this case.   
*In Eclipse, a file named Wait.java is open. In the file, the following code is visible:  
  
package ajax;  
  
import static junit.framework.accent(crossed out).assertTrue;  
public class Wait  
{  
       @Test  
       public void notThereYet()  
       {  
               webDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/wait.html");  
                 
               WebElement button = driver.findElement(By.id("go"));  
               WebElement entry = driver.findElement(By.id("entry"));  
               String message = "I am waiting!";  
               entry.clear();  
               entry.sendKeys(message);  
               button.click();  
  
               WebElement results = driver.findElement(By.id("results"));  
  
               try  
  
The presenter has highlighted the code notThereYet.  
  
The presenter clicks Run to run the code. In a browser window, a web page titled Wait for Response opens. It contains a text box with the entry "I am waiting!" and two buttons – Update and Update Fancy.   
  
The Update button is clicked and, above the buttons, the message "Results from site I am waiting!" appears.  
  
The presenter switches back to Eclipse.  
  
The JUnit tab specifies that the test failed and lists the line  
  
notThereYet[Runner: JUnit 4](6.002 s)  
  
In the Wait.java file, the presenter highlights the line  
  
WebElement button = driver.findElement(By.id("go"));  
  
He highlights WebElement entry = driver and then findElement(By.id("entry"));. Next he highlights String message = "I am waiting!";. The presenter then highlights entry.clear();, followed by entry.sendKeys(message);, and then button.click().  
  
The presenter scrolls down in the file to show the following code:  
  
assertTrue("The message sent contains the text sent", results.getText().contains(message));  
               } finally  
               {  
                   //driver.quit();  
               }  
       }  
  
The presenter highlights the line  
  
WebElement results = driver.findElement(By.id("results"));*   
  
And it doesn't. If we look at the error, it's the error specifically inside of here, says there's an assertion error, the message sent does not contain the text it was sent. Now, unfortunately, the way this is structured, since it's not looking for an exact match, we'll realize that that was actually blank at that point. It's just simply a timing error, because this looks immediately. And the second I click the button, it goes and looks immediately. Now, putting the click and wait here doesn't fix the situation, because click and wait would be waiting for the whole page to refresh, and it doesn't do it. We don't refresh the whole page. So what we have to do is, we have to make our tests smarter. So in this next test here, we can see I'm going to go through, and I'm going to wait. So after I click the button, I'm going to go through and find that element we are looking for, but I'm not going to do anything with it right away. So I'm going to create this waiter, and the waiter is going to wait up to 4 seconds. That's what this is going to have inside of here, this constructor. I pass in the driver, and I tell it how long I am willing to wait. In this case, the number four means seconds. And then I want to wait until some condition is out there. And we use this ExpectedCondition() class, we're using an anonymous inner class here, to fill in the apply method with whichever logic I want to.   
*The presenter highlights button.click() and then the line  
  
WebElement results = driver.findElement(By.id("results"))  
  
The presenter scrolls down to show the following code:  
  
       @Test  
       public void waitForIt()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/wait.html");  
  
               WebElement button = driver.findElement(By.id("go"));  
               WebElement entry = driver.findElement(By.id("entry"));  
               String message = "I am waiting!";  
               entry.clear();  
               entry.sendKeys(message);  
               button.click();  
  
               final WebElement results = driver.findElement(By.id("results"));  
               WebDriverWait waiter = new WebDriverWait(driver, 4);  
               waiter.until(new ExpectedCondition<Boolean>()  
                       {  
                               public Boolean apply(WebDriver driver)  
                               {  
  
The presenter highlights button.click();. He then highlights the line  
  
WebElement results = driver.findElement(By.id("results").  
  
The presenter scrolls down to show the following additional code:  
  
return ! results.getText().isEmpty();  
}  
});  
assertTrue("The message sent contains the text sent", results.getText().contains(message));.  
  
The presenter highlights various parts of the code, including the line  
  
WebDriverWait waiter = new WebDriverWait(driver, 4);*   
  
And what I am looking for is that results – the div actually, up here, the thing I grabbed up above – I'm looking for the text inside of there to no longer be updated. That's essentially what this check is saying inside of here. And when it's no longer empty, then the code will continue and then I can go off and check to see is my condition proper. So let me go and run this test, and I can show you, again, it's going to pop open the test. There it goes, it pops open the test, and it's waiting. It hasn't failed it yet, and the test is actually successful, because it doesn't continue running until this comes back true. Now, behind the scenes, it's calling this multiple times. It's calling "is it ready yet, is it ready yet, is it ready yet." But once this finally does say it's ready, then it goes off and runs. Now, we have to be careful, because it's not always blank like that. That same test might not just be looking for any text. I would have to go through and be a little bit smarter inside of here. So just to show you an alternative, I have pretty much the exact same code, except for instead of hitting the Go button, I have a fancy Go button out there. And so what's going to happen here is, it's going to go off and it's going to give a message, like Loading. And so there's several ways I could look for it here. But what my check is going to be in this case is I'm going to get the text, and as long as that text is equal to Loading, I am not going to continue. So once it's no longer equal to the word Loading,   
*The presenter highlights the code results.getText() and return | results.  
  
The presenter scrolls up in the code window. The following code is shown:  
               }  
       }  
       @Test  
       public void waitForIt()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/wait.html");  
  
               WebElement button = driver.findElement(By.id("go"));  
               WebElement entry = driver.findElement(By.id("entry"));  
               String message = "I am waiting!";  
               entry.clear();  
               entry.sendKeys(message);  
               button.click();  
  
               final WebElement results = driver.findElement(By.id("results"));  
               WebDriverWait waiter = new WebDriverWait(driver, 4);  
               waiter.until(new ExpectedCondition<Boolean>()  
                       {  
                               public Boolean apply(WebDriver driver)  
                               {  
  
The presenter highlights waitForIt. He then clicks Run to run the code.   
  
The presenter scrolls down in the code window to show the additional  code  
  
                                       return ! resulsts.getText()isEmpty();  
                               }  
                       });  
               assertTrue("The message sent contains the text sent", results.getText().contains(message));  
               driver.quit();  
       }  
  
The presenter highlights   
  
return ! results.getText().isEmpty();.  
  
The presenter scrolls down to show the additional code  
  
       @Test  
       public void fancyWait()  
       {  
               WebDriver driver = new FirefoxDriver();  
               driver.get("http://localhost:8080/website/pages/wait.html");  
  
               WebElement button = driver.findElement(By.id("goFancy"));  
               WebElement entry = driver.findElement(By.id("entry"));  
               String message = "I am waiting!":  
  
The presenter scrolls up in the code window and highlights results.getText().isEmpty();.  
  
He scrolls down again to the code  
  
               WebElement button = driver.findElement(By.id("goFancy"));  
               WebElement entry = driver.findElement(By.id("entry"));  
               String message = "I am waiting!";  
               entry.clear();  
               entry.sendKeys(message);  
               button.click();  
  
               final WebElement results = driver.findElement(By.id("results"));  
               WebDriverWait waiter = new WebDriverWait(driver, 4);  
               waiter.until(new ExpectedCondition<Boolean>()  
                       {  
                               public Boolean apply(WebDriver driver)  
                               {  
                                       return ! resulsts.getText().equalsIgnoreCase("Loading");  
                               }  
                       });  
  
               assertTrue("The message sent contains the text sent", results.getText().contains(message));  
  
               driver.quit();  
       }  
}  
  
The presenter highlights goFancy. He then highlights resulsts.getText(), followed by equalsIgnoreCase("Loading").*   
  
then it will continue on. So again, I'm not limited to just blank text, or no text, or whatever like that. So if I run this fancy last little test inside of here, we can see, when I click the other button inside of here, the other button says Loading. And it's doing some stuff in the back. And then it comes back, and the text comes back, and it's all fine. This is to reflect a more advanced site. It's going to give you some sort of visual indication. But the point here is, inside of my Java solution using the Selenium wait here, the WebDriverWait inside of here, I can wait for whatever condition I want to arrive at. It could be a combination of things, it could be for an element's status to change, it could be anything out there that I can see – as you can see, from a object that I'm already working with, or objects that I can find inside of here, because I have the full driver. Anything I can do from the driver, I can do to check my state. So now, even those tricky Ajax calls aren't going to be any problem for us, because we just wait for them to respond and continue on from there.   
*The presenter scrolls up in the code window and highlights fancyWait.  
  
The presenter switches briefly to the browser window and then back to Eclipse, where the following code is shown:  
  
               WebElement button = driver.findElement(By.id("goFancy"));  
               WebElement entry = driver.findElement(By.id("entry"));  
               String message = "I am waiting!";  
               entry.clear();  
               entry.sendKeys(message);  
               button.click();  
  
               final WebElement results = driver.findElement(By.id("results"));  
               WebDriverWait waiter = new WebDriverWait(driver, 4);  
               waiter.until(new ExpectedCondition<Boolean>()  
                       {  
                               public Boolean apply(WebDriver driver)  
                               {  
                                       return ! resulsts.getText().equalsIgnoreCase("Loading");  
                               }  
                       });  
  
               assertTrue("The message sent contains the text sent", results.getText().contains(message));  
  
               driver.quit();  
       }  
}  
  
The presenter highlights the code  
  
public Boolean apply(WebDriver driver)  
                               {  
                                       return ! results.getText().equalsIgnoreCase("Loading");  
                               }*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Cookies and Java Tests

Learning Objective

*After completing this topic, you should be able to*

* *use cookies in Selenium Java projects*

**1. Testing cookies using Selenium**

Cookies are a useful tool in designing pages, and thus testing cookies is needed to know that our pages are working well. Selenium provides simple methods for our Java-based testing to validate that our cookies are working properly on our page. So, here's a little test that we're using cookies as a key feature on a page. So, I'm going to run the test so we can see it in action. But as we do that, let me explain what's going on here. So, after we log into the page, we get this menu item over on this side. And as the user clicks these features, we're creating cookies. So that way as I move and navigate through the site, it will keep open these elements that I've clicked on previously. If I don't do that, then I either have to track it on the server side or they basically close every time I renavigate to a page. And so the designer here chose to use cookies as a way of being able to track these features. So, in our solution, we are going to create a login here. And we actually are using another login test to go through those steps. And once we've logged in, we're going to go out and ask for the very first item here. And so if we see our page from before,   
*In Eclipse, a file named CookieTest.java is open and the following code is shown:  
  
}  
  
@Test  
public void checkForCookie()  
{  
  
       // Login to the page  
       new LoginTest().loginSteps(drive);  
  
       WebElement navMenuItem = drive.findElement(By.id("ygtvlabel1"));  
       String linkName = navMenuItem.getText();  
       for (Cookie loadedCookie : allCookies)  
{  
               if (loadCookie.getName().equals(linkName))  
               {  
                       linkStatus = new Boolean(loadedCookie.getValue().equalsIgnoreCase("true"));  
               }  
       }  
  
       navMenuItem.click();  
  
       allCookies = drive.manage().getCookies();  
       for (cookie loadedCookie : allCookies)  
       {  
               if (loadedCookie.getName().equals(linkName))  
               {   
  
The Console pane includes the following details:  
  
<terminated> CookieTest.checkForCookie[JUNIT] C:\Program Files\Java\jre7\bin\javaw.exe (May 5, 2014, 8:28:45 PM)  
Your Account = true  
  
The presenter clicks Run to run the test and the Administrator Home page opens in a web browser. The page includes a site hierarchy with the nodes Your Account, Admin Functions, Site Content, and Interactive Content. The presenter expands Admin Functions to reveal the options Site Maintenance and Site Security. He expands Site Maintenance to reveal the options View Monthly Stats, View Logs, and Manage Site Text.  
  
Next the presenter expands the Site Content node. This reveals the Manage News, Manage Calendar, Manage Bookmarks, Google Maps, Content Lists, Menus, and File Content options. The presenter expands Manage News. This reveals the View Stories, Add Story, and Add Slideshow options.   
  
The presenter collapses the Site Content node and the Admin Functions node and switches back to Eclipse.  
  
The presenter highlights the code  
  
new LoginTest().loginSteps(driver);  
  
The presenter then highlights the line  
  
WebElement navMenuItem = driver.findElement(By.id("ygtvlabelel1"));  
  
The presenter briefly switches back to the test results.*   
  
you can see the very first item here is the ID of this nasty little text symbol inside of here that we see, but it loads up that first element. And then we ask it, "Hey what's the name of that element?", because the cookie name matches the name of that element. Now, this test is specifically designed this way, so I don't have to care what the text is on this page. So, if they change this from your account to owner account or just admin home or whatever it is going to be, we don't care about this text changing. We're just caring about the feature. Okay, so now we look to see if the cookies is already out there. And this is where we get into the behavior that we're trying to focus on here. So, from the driver we ask the manage. Now the driver is that prebuilt object that, in this case the Firefox driver we're using, this is the Selenium driver that we can go ask for information to the page. We hash the manage feature and then we say getCookies. And this returns to us a set of cookies. Now a cookie object is provided for us by Selenium. The cookie defines the name, the value and other details like the path, the domain and other items like the expiration time and stuff of the cookie. So, we can control every aspect of the cookie. We can see every aspect of the cookie as part of our tests. So, in this test, we're going to see if that cookie exists beforehand. We're looping through all the cookies to see if the cookie that matches the link name that's out there is already there. And if it is, we're going to store the status of it, because it's okay if somebody had clicked on that previously. We just want to know in this test, does it flip status appropriately?   
*The presenter switches back to Eclipse. He highlights the code ygtvlabelel1 and then the code  
  
linkName = navMenuItem.getText();  
  
The presenter switches back to the test results briefly and switches back to Eclipse. He highlights the code driver.manage(), the code new FirefoxDriver();. The presenter highlights the code driver.manage().getCookies(). He then highlights manage().getCookies();.  
  
In Eclipse, the presenter switches to a file named Cookie.class. It contains the code  
  
Copyright 2007-2009 Selenium committers  
  
package org.openqa.selenium;  
  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
public class Cookie {  
       private final String name;  
       private final String value;  
       private final String path;  
       private final String domain;  
       private final Date expiry;  
       private final boolean isSecure;  
       private final boolean isHttpOnly;  
  
       /\*\*  
               \* Creates an insecure non-httpOnly cookie with no domain specified.  
               \*   
               \* @param name The name of the cookie; may not be null or an empty string.  
               \* @param value The cookie value; may not be null.  
               \* @see #Cookie(String, String, String, String, Date)  
               \*/  
       public Cookie(String name, String value, String path, Date expiry) {  
               this(name, value, null, path, expiry);  
  
The presenter switches back to the CookieTest.java file. He highlights the code <Cookie> and then the code (loadedCookie.getName().equals(linkName). He also highlights the code  
  
linkStatus = new Boolean*   
  
So, in order to make sure, we have the status they had beforehand and then we click on it. Now, we stored the status as a Boolean, so if it's null, there was no cookie out there. The cookie didn't exist, our status is still null. If it was out there, then we have a true or false value that's associated with that. So, after we click on the link, we go and we ask for all the cookies again, and then we loop through the cookies and we make the decision. So, if the cookie was not out there beforehand and we see a status of false, then our functionality didn't work right, because it's supposed to be true. If the cookie wasn't there beforehand, it should be showing it open for the first it should be true. If the linkStatus out there is already out there and our status now isn't the opposite, then it's a problem. We can fail that test. Otherwise, if that status is false that was created out there, then that's a problem as well. It should be set to true. So, those are the types of tests we can do. I don't know if we had every one of the tests, but it seems like a pretty decent set of tests out there. And so now, within our Java testing solutions we can go and check cookies on a page. And we can make sure that any time we use cookies as a design element out there, we don't have to just trust the features look right on the screen we can know behind the scenes that the cookies are set up properly and they're doing the exact values and in such that we want behind the scenes. Which lets us do things of how the cookie data is stored, is encrypted, is it mean the right thing? All sorts of features that we wouldn't be able to do otherwise on a browser-based solution. So, using Java plus cookies to validate is a great way of really ensuring the quality of our website.   
*The presenter highlights the code navMenuItem.click(); and then the code Boolean linkStatus = null;.  
  
The presenter scrolls down and highlights the code  
  
allCookies = driver.manage().getCookies();  
  
He highlights the code (linkStatus == null, followed by the code && status == false).  
  
The presenter highlights the code  
  
fail("The cookie was added but false");  
  
He highlights the code  
  
linkStatus != null &&  
  
The presenter highlights the code linkStatus != ! status) and (status == false).*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

Exercise: Use Selenium with Java

Learning Objective

*After completing this topic, you should be able to*

* *use Selenium with Java*

**1. Exercise overview**

We've covered a lot of ground up to this point, so here's a chance for you to turn theory into practice.

You are working in the IT department for EasyNomad Travel Inc. Your company has requested your assistance with building a website for travel reservations.

**2. Using Selenium with Java**

Question

You are creating a new Maven project. Which fields are required when defining a project dependency?

**Options:**

1. Artifact ID
2. Group ID
3. Scope
4. Version

Answer

***Option 1:****Correct. The Artifact ID field cannot be left empty as it is required when defining a new dependency for a project.*

***Option 2:****Correct. The Group ID field is used as an identifier for a project and is a required field.*

***Option 3:****Incorrect. The scope field is an optional field which can include options such as; compile, provided, runtime, for example.*

***Option 4:****Incorrect. The Version field is an optional field used to identify the version of the dependency.*

**Correct answer(s):**

1. Artifact ID  
2. Group ID

Question

Complete the code required to retrieve the window handles.

**Code**  
for (String handle : driver.INSERT THE MISSING CODE

**Options:**

1. getWindowHandles
2. getWindow
3. getHandles

Answer

Each window is assigned an alphanumeric id called a Window Handle. The getWindowHandles string is used to retrieve the window handle of all windows, while the getWindowHandle string is used to retrieve only the handle of the current window.

***Option 1:****Correct. You use the code getWindowHandles to retrieve window handles.*

***Option 2:****Incorrect. You use the code getWindowHandles, rather than getWindow, to retrieve window handles.*

***Option 3:****Incorrect. You use the code getWindowHandles, rather than getHandles, to retrieve window handles.*

**Correct answer(s):**

1. getWindowHandles

Question

Complete the HTML5 code used to drag and drop an image.

**Code**  
<img id”drag1” src=”imagemap.png” draggable= INSERT THE MISSING CODE

**Options:**

1. “true”
2. "yes"
3. "1"

Answer

The draggable=”true” attribute can be used in HTML5 on any element you want to make moveable. This can also be accomplished using the draganddrop java action.

***Option 1:****Correct. You set the draggable property to "true" to enable an image to be dragged and dropped.*

***Option 2:****Incorrect. You set the draggable property to "true", rather than to "yes", to enable an image to be dragged and dropped.*

***Option 3:****Incorrect. You set the draggable property to "true", rather than to "1", to enable an image to be dragged and dropped.*

**Correct answer(s):**

1. “true”

Question

You want to search all elements for a link that contains the word "travel". Complete the code required to accomplish this.

**Code**  
WebElement partial = driver.findelementINSERT THE MISSING CODE

**Options:**

1. (By.partialLinkText(“travel”))
2. (By.partial(“travel”))
3. (By.partialLinkText)

Answer

The By.partialLinkText string is used to identify a specified string in any link text. In this scenario, by completing the code with (By.partialLinkText(“travel”)), you can search any link text containing the word “travel”.

***Option 1:****Correct. To complete the code, you specify (By.partialLinkText(“travel”)).*

***Option 2:****Incorrect. You need to use the code (By.partialLinkText(“travel”)).*

***Option 3:****Incorrect. You need to use the code (By.partialLinkText(“travel”)).*

**Correct answer(s):**

1. (By.partialLinkText(“travel”))

Question

Complete the code required to return all cookie objects.

**Code**  
allCookies = driver.manage().INSERT THE MISSING CODE()

**Options:**

1. getCookies
2. get
3. driver.cookie

Answer

The getCookies method is used to return cookie names and cookie objects in a request. If not cookies are available, the request returns null.

***Option 1:****Correct. You use the getCookies() method to return cookie objects.*

***Option 2:****Incorrect. You use the getCookies() method, rather than just get, to return cookie objects.*

***Option 3:****Incorrect. You use the getCookies() method, rather than driver.cookie, to return cookie objects.*

**Correct answer(s):**

1. getCookies

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/sd_slwt_a03_it_enus/output/html/course_transcript.html#top)

© 2017 Skillsoft Ireland Limited