Introduction to Selenium Concepts and Scripting

Student Manual

Version 1.0

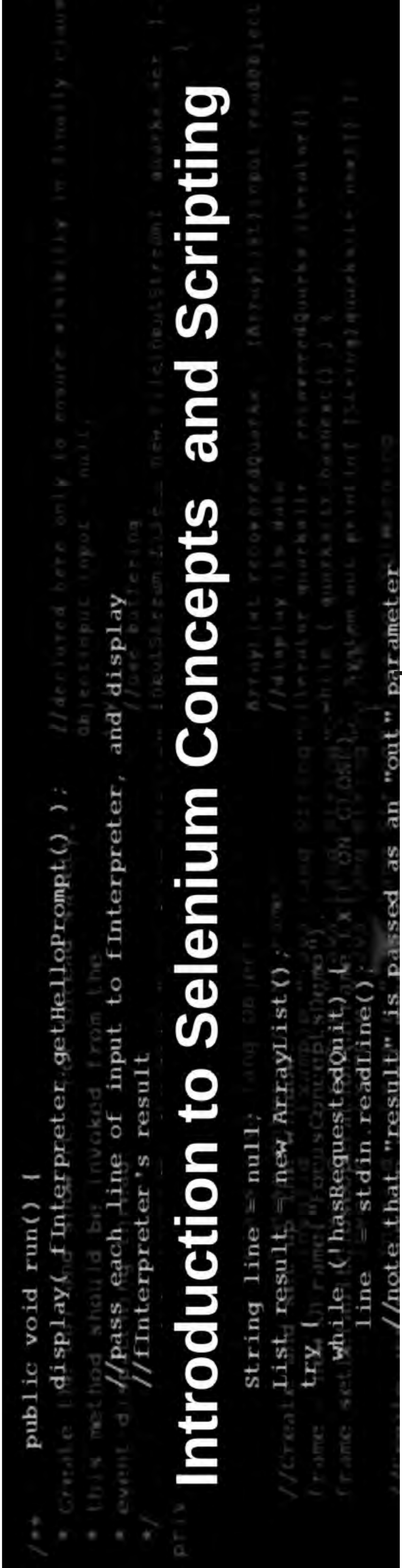
exgnosis

Document Information

Course Title:	Introduction to Selenium: Concepts and Scripting
Author:	Rod Davison
e-mail:	rod@exgnosis.ca
Version:	1.0
Release Date:	2018-08-01
ProTech ID:	PT20158
Doc Type:	Student Manual
Course Date:	September 17-21, 2018
Location:	Virtual

"The primary objective of copyright is not 'to reward the labour of authors, but to promote the Progress of Science and useful Arts.' To this end, copyright assures authors the right to their original expression, but encourages others to build freely upon the ideas and information conveyed by a work. This result is neither unfair nor unfortunate. It is the means by which copyright advances the progress of science and art."

Justice Sandra Day O'Connor



Introduction to Selenium Concepts and Scripting

Module One

Introduction to Selenium

Improving quality requires a culture change, not just a new diet.

Philip Crosby

Quality in a service or product is not what you put into it. It is what the customer gets out of it.

Peter Drucker

It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free.

Steve McConnell

It is not enough to do your best: you must know what to do, and THEN do your best.

W. Edwards Deming

Introduction to Selenium Concepts and Scripting

September 17-21, 2018

1. What is Selenium?

The very first paragraph on the Selenium website1 gives a wonderfully succinct and accurate description of Selenium:

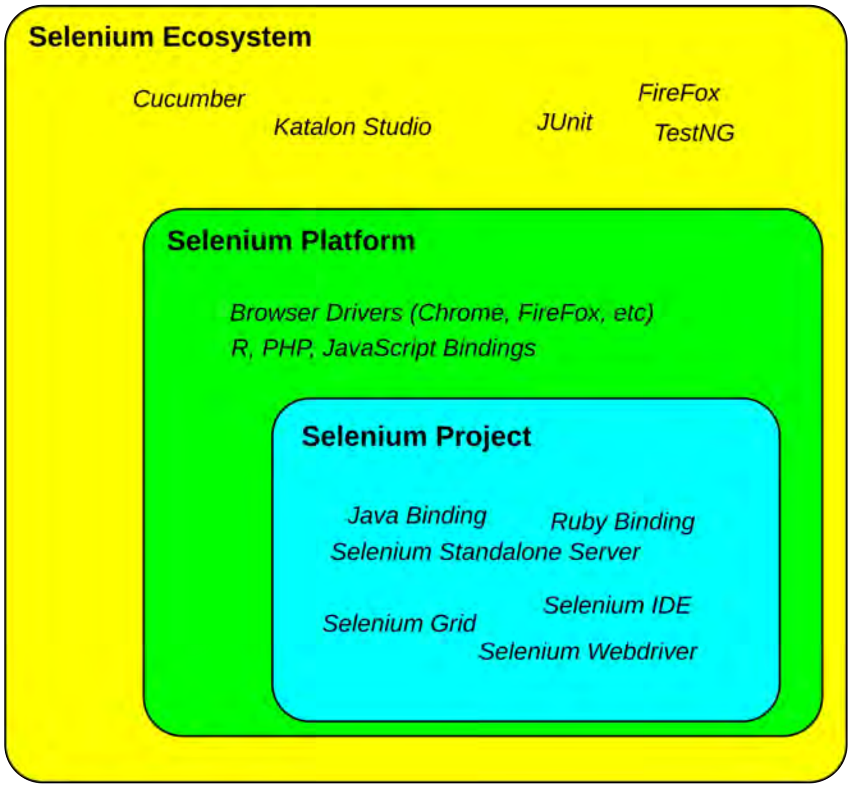
> *Selenium automates browsers. That's it! What you do with that power is entirely up to you. Primarily, it is for automating web applications for testing purposes, but is certainly not limited to just that. Boring web-based administration tasks can (and should!) be automated as well.*

Selenium is a set of tools that are used to implement and execute functional tests for web based applications. Selenium is not a stand alone application in the same sense that IBM Rational Functional Tester is, but is designed to be an open framework that relies on integration with other tools and software in order to function.

This first module is a high level overview of the Selenium ecosystem intended to provide a basic understanding of what all the different components are that make up Selenium and how they all work together to provide the Selenium functionality.

1.2 The Selenium Ecosystem

The Selenium Platform is a core set of tools that are maintained by the Selenium Project. In the diagram above, the platform is the blue inner circle. The code base for the Selenium platform tools is maintained in the repositories belonging to the Selenium Project[1].



There are also a number of Selenium plugins and add-ons such as other language bindings and browser drivers, which are developed and supported outside the Selenium project. These external projects are not part of the Selenium platform but extend its functionality, scope and usefulness. These tools are generally endorsed by the Selenium project but are maintained by third parties and belong to them although they have been made available to the Selenium project. The endorsed tools and plug-ins appear on the Selenium downloads page as "third party tools." A good rule of thumb is that you can trust third party tools downloaded from the Selenium website since their creators are considered reliable by the Selenium project. In the diagram, these plug-ins are represented by the green circle.

The external tools are products that are designed to use or interact with Selenium but are not part of the Selenium project. A good example of one of these tools is Cucumber, the Behaviour Driven Development test automation framework – when running acceptance tests through a web based interface, Cucumber step definitions utilize Selenium code for test automation. These are in the light red circle.

However, there is a caveat to this categorization – Selenium is an open source project which means that organization and responsibilities are fluid which makes this categorization fuzzy and changeable. For example, the FireFox web driver was part of the Selenium platform but because of the changes to FireFox after version 55, this driver had to be scrapped and was replaced by the gecko driver which is now a third party plugin.

I will generally use the term "Selenium ecosystem" to refer to the whole of the Selenium project, related projects and contributors. There is a list of organizations and individuals that make up the core of the ecosystem at https://www.seleniumhq.org/ecosystem/

1.3 The Selenium Platform

The core of the Selenium platform consists of Four main items.

1. **The Selenium web driver.** This is the main engine Selenium and the core of the project. The driver is now at version 3 and the Selenium API is now under consideration as a W3C standard which would be incorporated into future browsers directly.
2. **Core language bindings.** The Selenium project maintains several of the language bindings – which are language specific APIs for interacting with the Selenium driver in code. Java, Ruby, C# and Python are supported by the Selenium project although other language bindings like JavaScript and R are available as third party plug-ins.
3. **The Selenium IDE.** Originally this was a core part of Selenium project and available only in FireFox. However, after FireFox version 55 was released, the IDE became incompatible with the new FireFox extension architecture. Currently new IDEs are being developed via a third party organization – Katalon – for FireFox and Chrome. At the time this material was prepared, the new Selenium IDE is sort of on the boundary between being a third party tool and part of the platform.
4. **Selenium Grid.** This can be thought of as a server that coordinates the running of tests in parallel on multiple machines. This tool is used for a variety of test scenarios such as splitting up a large test suite into chunks that can be run simultaneously on different machines, or running the same tests on different browsers in different environments at the same time.

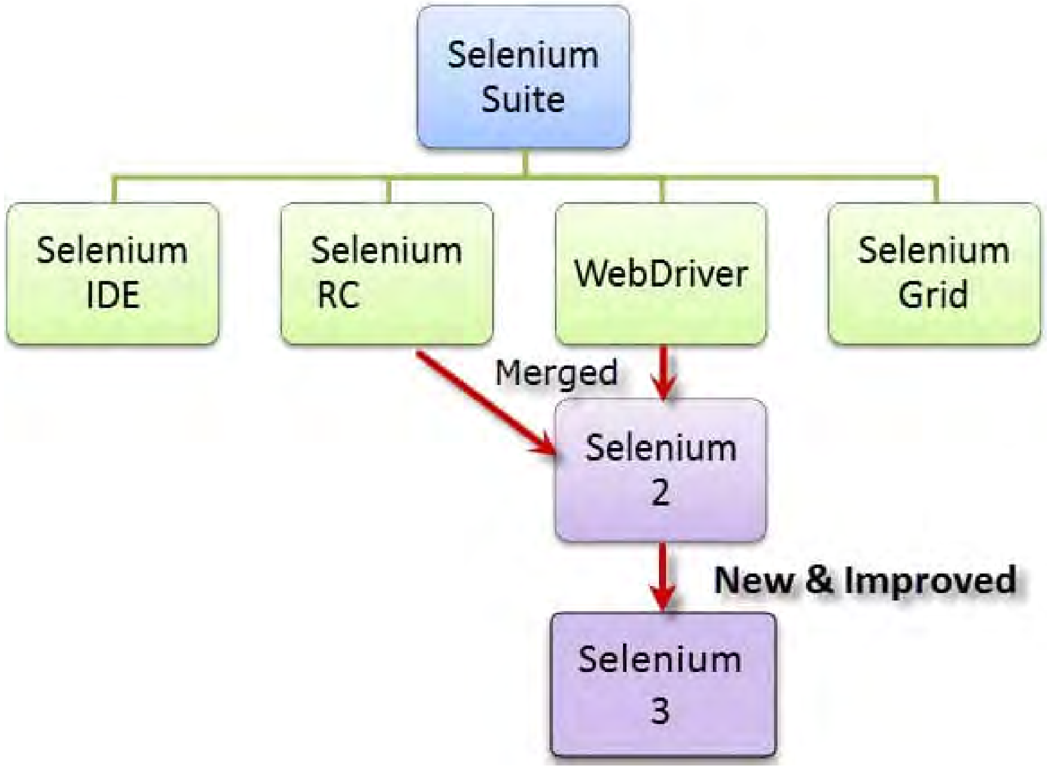
Each of these tools automates some aspect of the script design and execution process. They do not all have to be used together and, depending on our specific needs, we may find that we only use a subset of these tools in our day to day operations.

One important point to keep in mind during this course: Selenium does not “do” testing for us, rather it automates parts of the testing process that often are bottlenecks in terms of getting testing done, as well as automating tasks that tend to be the source of human error in manual testing. Since Selenium automates our existing testing process, if our testing is poorly designed, the using Selenium will only speed up the ultimate failure of our testing efforts.

1.4 The Evolution of Selenium

Selenium is one of those projects that evolved from a single tool into a suite of tools, and soon to a W3C standard. This section is a quick recap of how Selenium got to where it is today. The following is from the documentation on the Selenium project website[2].

2004: Selenium started as a JavaScript library by developed by Jason Huggins as an internal application at ThoughtWorks. The library that could drive interactions with a web page, allowing him to automatically rerun tests against multiple browsers. That library eventually became Selenium Core, which underlies all the functionality of Selenium Remote Control (RC), which is now referred to as Selenium 1.



2006: Simon Steward at Google started work on a project he called WebDriver. Google had long been a heavy user of Selenium, but testers had to work around the JavaScript limitations of the product. Simon wanted a testing tool that spoke directly to the browser using the 'native' method for the browser and operating system, thus avoiding the restrictions of a sand boxed JavaScript environment. The WebDriver project began with the aim to solve the Selenium' pain-points.

2008L The WebDriver Project is merged with Selenium to create the Selenium Web Driver which is called Selenium 2.

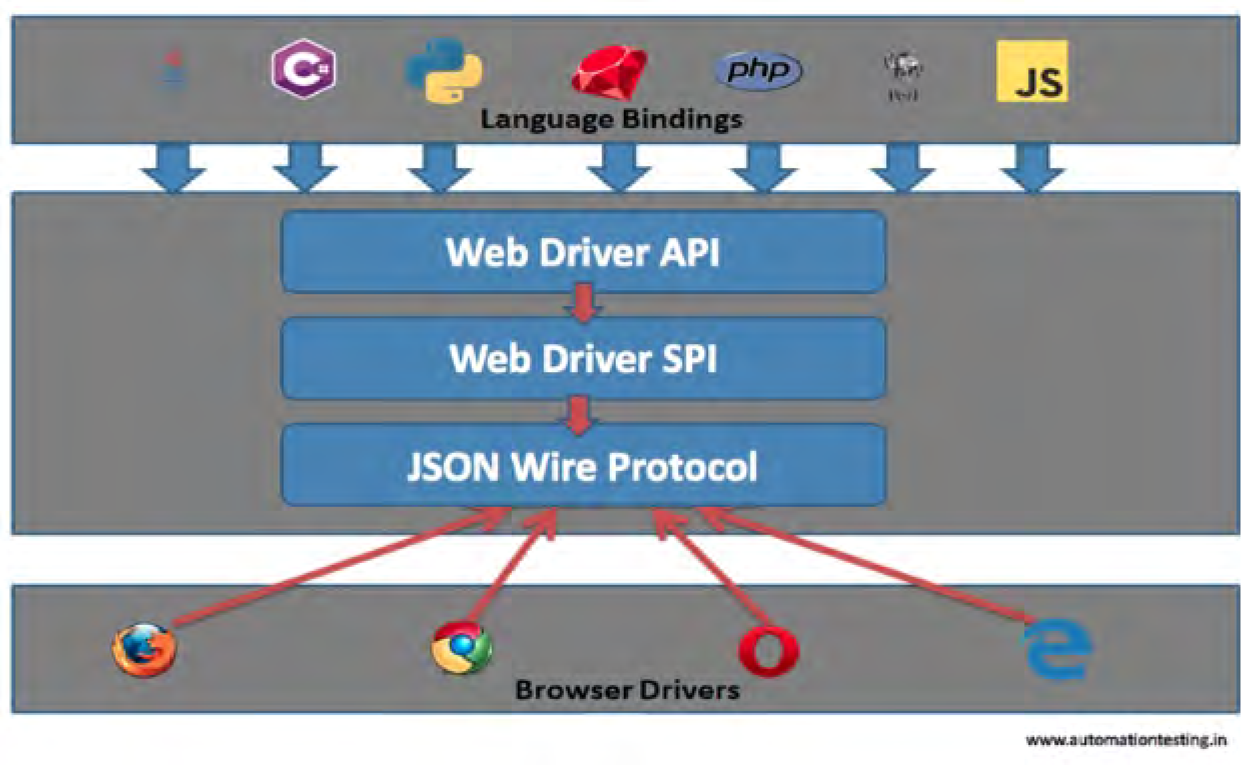
2016: Support for the Selenium Core (RC) is dropped and Selenium moves to a totally WebDriver base with enhanced functionality over version 2. Support for mobile platforms is spun off into the Appium project. Changes to browsers, especially FireFox require an overhaul of the browser drivers which are now supported by the browser vendors. As well, the Selenium 3 WebDriver API is now becoming a W3C standard. The previous IDE on FireFox is scrapped because of browser changes and a new IDE project launched for Chrome and FireFox.

One of the future trends we can expect to see the evolution of the Selenium projects and related products to evolve into other forms of http based interfaces on different platforms, especially mobile, but on any environment that uses http based Restful web services, one of the core technologies of the emerging Internet of Things (IoT).

1.4 How Selenium Works

The following diagram[3] gives a basic overview of how Selenium works.

1. **Language Binding:** This is a language specific API that offers WebDriver command in that language.
2. **Selenium API:** These is the interface to Selenium that the commands in the programming

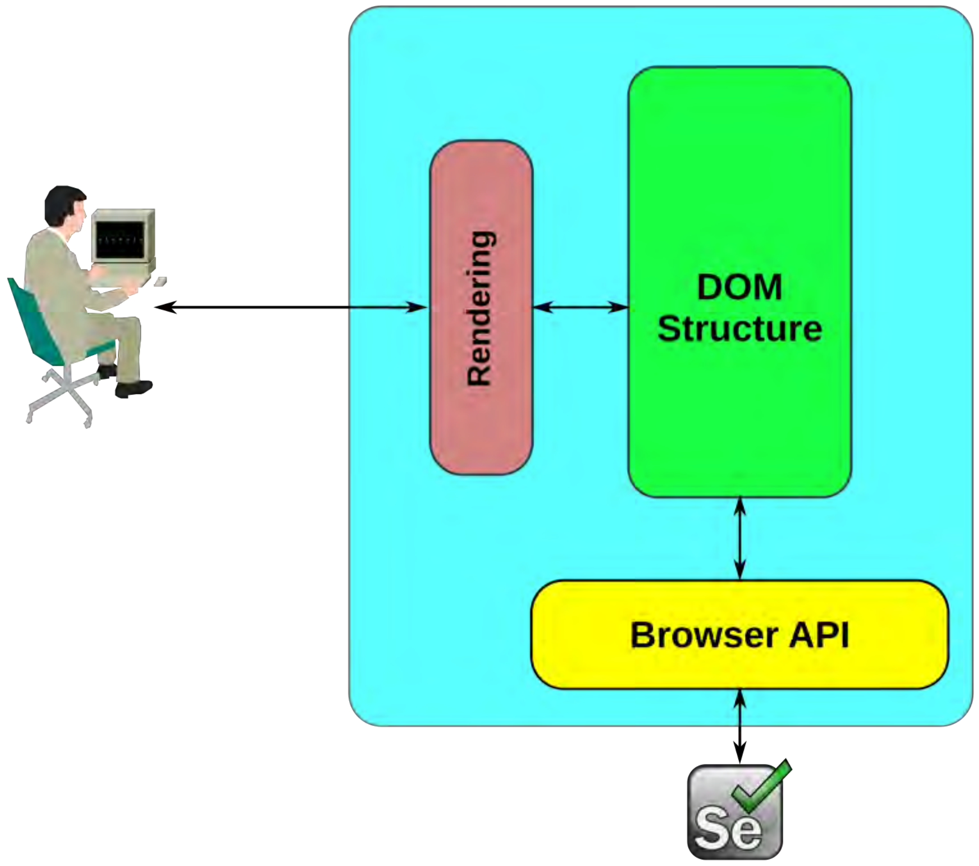


language communicate. The API receives the language specific requests from the language binding and passes them on to the SPI.
3. **Selenium SPI:** This is called the "stateless programming interface" and is were the API requests are translated into a common protocol and various other tasks are performed to get the request passed on to the browsers.
4. **Jason Wire Protocol:** Also called the Selenium wire protocol, this is a set of standard commands that communicate directives from the Selenium SPI to the various browser drivers using a Restful web service.
5. **Browser Drivers:** Each driver converts the Restful web service request into a request that is specific to that browser API. Since different browsers have different APIs. Selenium cannot issue commands directly to a browser. One of the goals of making the Selenium API a W3C standard is to eliminate the need for independent drivers for compliant browsers.

1.4.1 The Browser Interface

When most people think of a browser, what they are really thinking about is the the display of a web page in a browser. In reality, a lot happens behind the scenes which is where Selenium operates.

1. When a web page is loaded, the HTML is converted into what is called a DOM tree. This is a set of objects in memory that represent the "things" on the web page.
2. Each of the objects in the tree has a set of properties that tell the browser how to render or draw them on the scree. These properties include the kind of object to draw (button, text field, text, etc) and the styles of the object (font, colour, text to display,etc.) as well as its state (checked, visible, etc.). The CSS style sheets are used to set the properties and



JavaScript is used to change them dynamically.

3. The rendering engine draws the DOM tree on the screen using the properties of each of the objects. When the user interacts with the screen, for example clicking on a button, the rendering engine remembers what it drew there and converts the user action into a command to the underlying DOM object.
4. What the WebDriver does is bypass the rendering engine and interact directly with the underlying DOM object, analogous to the way JavaScript does. However, when the WebDriver affects the DOM object, then the rendering engine picks up the change, and since it doesn't know what made the change, it displays it on the screen.

Headless browsers are those that do not have a rendering engine are are common in testing. Both Chrome and FireFox can both be run in headless mode. There are a number of browsers like HtmlUnit and Splash that are headless browsers specifically intended for use in testing. Selenium can work with either standard browsers or headless browsers.

2. Using Selenium

There are five primary reasons to use Selenium or any automated testing tool:

1. Automating test execution eliminates many types of human error that cause erroneous test results.
2. The rate determining step in a manual testing activity is usually the number of testers available and the speed at which the testers can run tests and evaluate the results of the tests. Running through use cases on a browser manually is especially time consuming.
3. Tool assisted test case design can be used to identify errors in test case design before the test cases are deployed into the testing environment. We will look at this process in more detail in the next module.
4. It simplifies regression testing. We can rerun previous tests exactly as they were run before.
5. Automated test evaluation eliminates the errors made by testers who incorrectly evaluate test results using some sort of manual inspection process.

A primary source of errors in software testing are the mistakes made by the testers in the execution of the test cases. This is particularly true when running scenarios with a web application because the work is mind numbing as the tester clicks on a link, enters a value, clicks on another link... and so on ad infinitum. It is precisely because this sort of activity is so tedious and repetitive that causes testers to become bored and distracted; exactly the mindset were we tend to make errors.

Test automation eliminates this specific class of human based errors. Selenium never gets bored or distracted and always executes the test exactly the same way every time. Humans, on the other hand, have significant problems executing the same scenario in exactly the same way. Experience shows automated test execution produces more consistent test results by a significant measure.

Automated testing also eliminates the human bottleneck. In a manual environment, the rate at which we can test is dependent on how fast our testers can work and how many testers we have. With automated test execution, this resource dependency is no longer a factor which means that can scale up our testing activity by orders of magnitude without a corresponding increase in costs. This creates the capability to implement continuous testing in a seamless and cost effective manner.

When testers write bad tests or interpret the test results incorrectly, the whole testing process becomes suspect. Selenium IDE is a test script prototyping tool that can be used to ensure scripts are designed correctly, and also can be used to review and vet that the script does in fact do what we think it does. Similarly, using automatic test validation eliminates the interpretation errors of test results that humans are prone to make.

We can summarize some of the benefits of using Selenium as:

1. It ensures consistency in test execution in regression testing activities.
2. It ensure consistent test execution over the development cycle.
3. It allows closer integration between development activities and testing activities.
4. It supports the continuous testing philosophy of Agile and DevOps methodologies.
5. It reduces project resource needs for testing activities while not sacrificing completeness of test coverage.
6. It eliminates many of the common human errors made in the testing process.

In the next section, we will start looking at the use of Selenium in a testing environment. We will look at using Selenium in both a stand alone environment and integrated with the acceptance test driven development (ATDD) tool Cucumber.

2.1 Exploratory Testing

Exploratory testing might also be described as "what if.." testing. Typically, we would try the sort of edge cases, oddball scenarios and the other sorts of things that might arise to see how the application under test handles them.

The most common sort of case we look at in exploratory testing is the "outlier" which is a case that is valid but statistically unusual or rare. Consider this example of an outlier from CNN money:

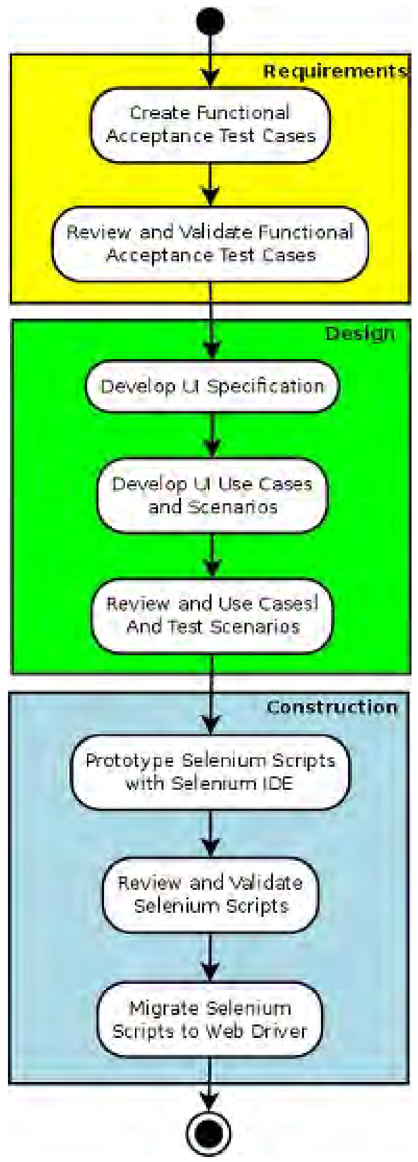
> *Steve Valdez knew he'd face scrutiny when he went to a Bank of America branch to cash a check written to him by his wife, since he doesn't have a personal account with the bank.*
>
> *Although the Tampa, Fla., resident brought two forms of picture identification, the teller said he needed to provide a thumbprint to cash the check.*
>
> *The problem: Valdez was born without arms and wears a prosthetic. He pleaded with the bank manager to use the photos, but to no avail.*
>
> *"It was really unfathomable," said Valdez, who was told that he could either bring in his wife or open an account of his own.*
>
> *A thumbprint is a longstanding requirement for any non-account holder to cash a check in order to prevent check-cashing fraud, according to a BofA spokesman, who admits that the bank should have offered alternatives to Valdez.*

Mr. Valdez is an outlier, a valid case that was not considered when setting up the bank policy. In exploratory testing, we would ask "What if someone shows up who is unable to provide a thumb print?"

When doing exploratory testing on a web application, using Selenium allows us to record the interaction so that the situation that might be problematic can be reproduced. One of the problems developers have in trying to debug an application is trying to reproduced the circumstances that led to the problem. Using Selenium to record the steps taken by the tester rectifies this.

2.2 Integration with Testing Protocols

Selenium automates our testing process. This cannot be emphasized enough – if we have no testing process in place, then the benefits of using Selenium are questionable. In this section, we will take a quick overview on when to use Selenium and what should be done before we start working with Selenium scripts.


Requirements
Create Functional Acceptance Test Cases
Review and Validate Functional Acceptance Test Cases
Design
Develop UI Specification
Develop UI Use Cases and Scenarios
Review and Use Casesl And Test Scenarios
Construction
Prototype Selenium Scripts with Selenium IDE
Review and Validate Selenium Scripts
Migrate Selenium Scripts to Web Driver

In the work flow diagram shown, we give a generic sense of the flow of what needs to be done before we develop a Selenium script. Suppose that we are developing a Capital One Mortgage Affordability Calculator. This already exists on the Capital One website so we will use it in the example, and for simplicity we will call it “The Calculator.”

If we were in a development process, we would normally start thinking about test cases in the requirements phase. I am using the term “requirements” here in a very generic term rather than as a reference to a specific formal phase in a development methodology. Requirements here could refer to getting a formal business process description or to an Agile collaborative face-to-face discussion between the developer and the product owner.

Requirements and Acceptance Tests

Usually in this phase of our development, we have not yet designed the features of the UI and tend to focus on the acceptance criteria for the application (i.e “What should it do for you to be considered to be working correctly?”) . In the Agile approaches, we would use these acceptance criteria to develop a set of acceptance test cases. These test cases would exercise the business logic of the calculator by specifying the output we should get from a given set of inputs. We use the product owner's acceptance criteria to verify a set of acceptance test cases.

The box on the next page describes a single acceptance test case.

Design and Use Cases

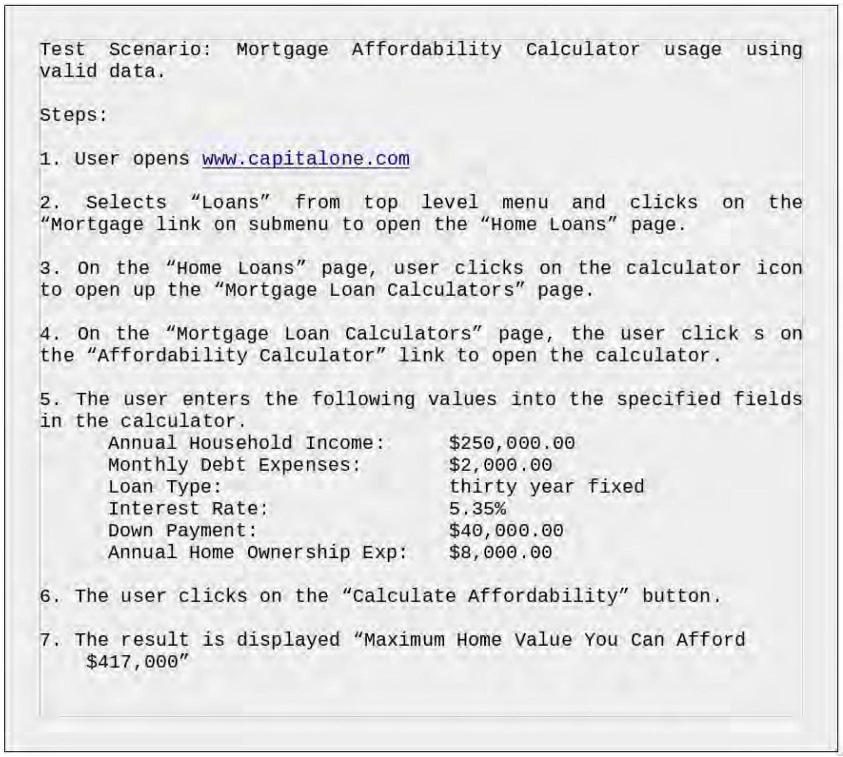
Once the UI is designed, we need to develop use cases to describe how to execute a particular piece of functionality. For example, to execute our test case above, we have to navigate to the calculator page at the Capital One web site and then enter the data and get a result in order to execute the test case.

There is a lot of hand waving and arguing among the development gurus about the difference between a user story and a use case. For the purposes of this course, we can consider a user story to be a general description of what a product owner or user would want to do at the web site to get the functionality they want. We will use the term use case to refer to the dialog between the user and the system described in terms of specific measurable actions between the UI elements and the user. The important feature of a use case is that ever interaction between the UI and the user is quantified and measurable. For example, users don't just provide data, they enter specific values into specified fields in the UI.

```
Summary:Test of the Capital One Mortgage Affordability
Calculator.

Input Test Data:

      Annual Household Income:      $250,000.00
      Monthly Debt Expenses:        $2,000.00
      Loan Type:                    thirty year fixed
      Interest Rate:                5.35%
      Down Payment:                 $40,000.00
      Annual Home Ownership Exp:    $8,000.00

Expected Result: $417,000.00
```

At this point there are three things that we need to deal with in a use case.

1. The navigation through the UI.
2. The correctness of the test case output.
3. The correctness of our scripting code.

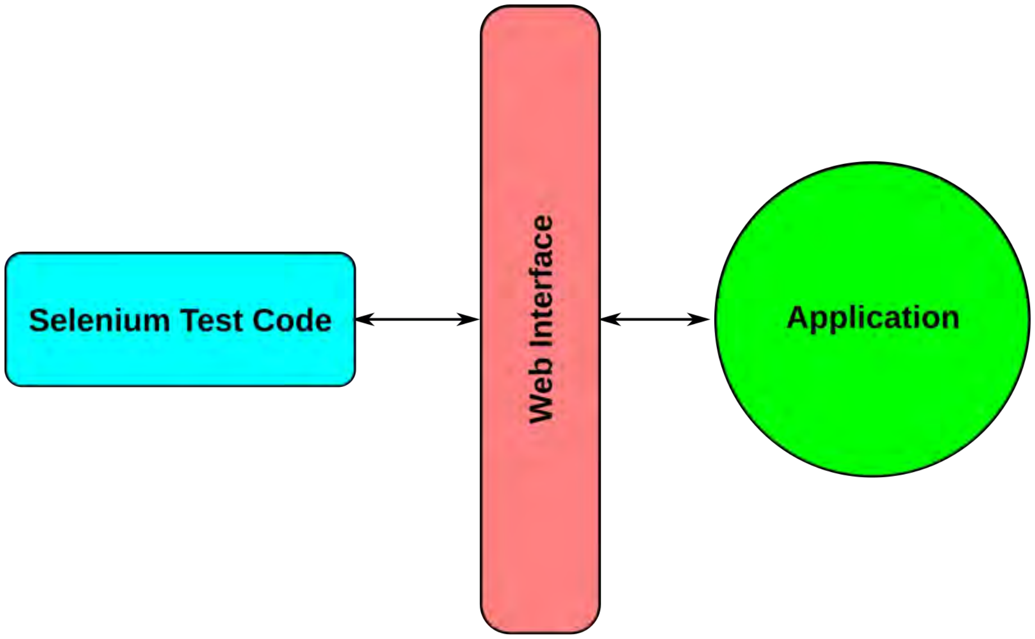


```
Test Scenario: Mortgage Affordability Calculator usage using
valid data.

Steps:

1. User opens www.capitalone.com

2. Selects “Loans” from top level menu and clicks on the
“Mortgage link on submenu to open the “Home Loans” page.

3. On the “Home Loans” page, user clicks on the calculator icon
to open up the “Mortgage Loan Calculators” page.

4. On the “Mortgage Loan Calculators” page, the user click s on
the “Affordability Calculator” link to open the calculator.

5. The user enters the following values into the specified fields
in the calculator.
      Annual Household Income:      $250,000.00
      Monthly Debt Expenses:        $2,000.00
      Loan Type:                    thirty year fixed
      Interest Rate:                5.35%
      Down Payment:                 $40,000.00
      Annual Home Ownership Exp:    $8,000.00

6. The user clicks on the “Calculate Affordability” button.

7. The result is displayed “Maximum Home Value You Can Afford
    $417,000”
```

The above use case describes the first two. From this use case, we develop test scenarios to improve the efficiency of out testing efforts. For example, the above use case might generate a number of different test scenarios, each differing only the input and expected outputs.

On the other hand we might split this up unto two scenarios, one to test the navigation to the right page, and another that starts on the calculator page and only is concerned with the inputs and and correctness of the output of the calculation.

In the diagram below, we see the problem. There are three places in our testing process that could affect the outcome of a test case.



1. There could be an error in the application itself.
2. There could be an error in the interface.
3. There could be an error in our Selenium code.

What we want to avoid is a situation like that described by Kent Beck:

> *I don't like interface based tests.*
>
> *In my experience, tests based on interface scripts are too brittle to be useful. When I was on a project where we used interface testing, it was common to arrive in the morning to a test report with twenty or thirty failed tests.*
>
> *A quick examination would show that most or all of the failures were the programming running as expected. Some cosmetic change in the interface had caused the actual output to no longer match the expected output.*
>
> *Our testers spent more time keeping the tests up to date and tracking down false failures and false successes then they did writing new tests.*

We can avoid this whole whack-a-mole situation by following a simple strategy.

1. Errors in scripting can often be identified by using Selenium IDE to prototype the script as baseline to compare our code to. If the Selenium IDE script works and our web driver code doesn't, we can generally assume our web driver code has an error.
2. When testing the application, we generally want to use a minimal test interface – this is a recommended best practice. Ideally we want to exercise the application directly without going through a specific interface since there may be multiple interfaces (for example, web, web service, mobile) and we want our testing of the application to be independent of those.
3. When testing the interface, we often want to use a dummy or mock application that always produces the right result for each test case. When we get an error in this case, it is probably located in the interface.

Once we have got tests passing for both the application and the interface separately, then we have to do a integration test to ensure the application and interface are working together properly.

Terminology Recap

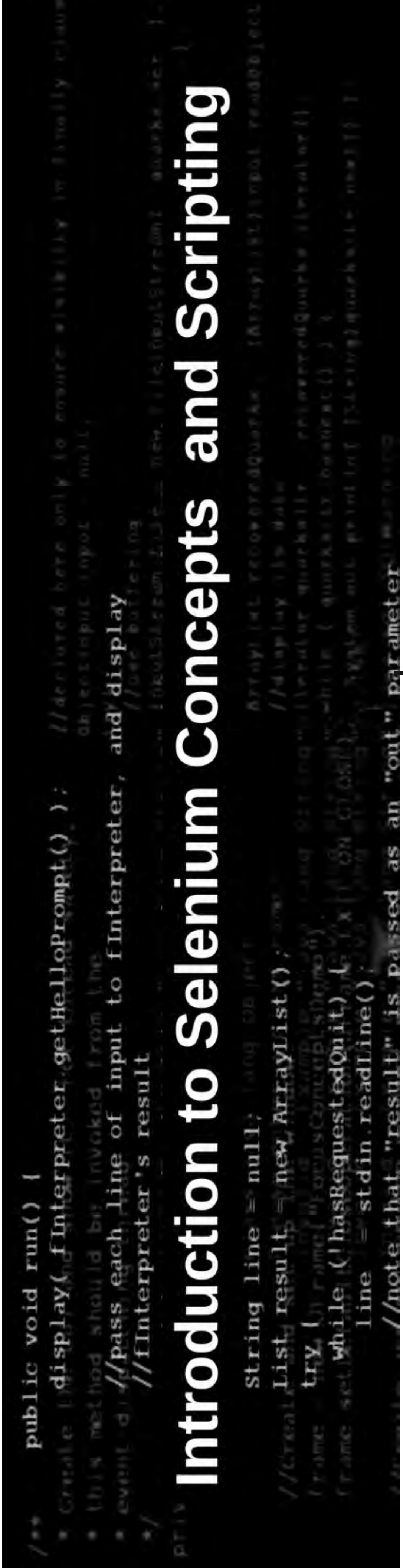
We cannot develop a Selenium Script until we have an actual UI to execute the script against. This is why I have placed the development of the Selenium scripts into the construction phase which generally means that we are writing code or building the web pages.

To recap terminology:

1. **Test Case:** An input (or set of inputs) and an expected output from the system when the system is in a given state. In a web based UI, the state often corresponds to a specific web page.
2. **Test Scenario:** A description of how a user would navigate through and interact with a UI to get a specific result. Before a scenario can be described, the UI must be specified but not necessarily implemented. Often story boards or mock ups can be used at this point rather than actual code.
3. **Selenium Script:** A set of Selenium commands that executes a test scenario on an existing and functional UI. The UI must be built before we can execute a Selenium script.

Notes and References

1. *The main Selenium page is located at: https://www.seleniumhq.org. However note that this page is not updated on a regular basis except for the downloads section.*
2. *Selenium History Diagram from: https://www.guru99.com/introduction-to-selenium.html*
3. *Selenium Architecture diagram from: http://automationtesting.in/selenium-webdriver-architecture*



Module Two

The Selenium IDE

A most important, but also most elusive, aspect of any tool is its influence on the habits of those who train themselves in its use. If the tool is a programming language this influence is, whether we like it or not, an influence on our thinking habits.

Edsger Dijkstra

More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded - indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest.

Boris Beizer

A fool with a tool is still a fool.

Martin Fowler

1. Authoring Selenium Scripts

There are two distinct activities that need to do to use Selenium effectively:

1. Developing Selenium scripts.
2. Automatically running Selenium scripts.

In this section, we are going to focus on the first activity; the developing of Selenium scripts, with a focus on prototyping scripts. The second activity is the realm of the Web Driver and will be dealt with later in the course.

The developers of Selenium are quite clear that the Selenium IDE is NOT intended to be a script automation tool but rather a script prototyping tool.

> *The Selenium-IDE (Integrated Development Environment) is the tool you use to develop your Selenium scripts. It's an easy-to-use Firefox plug-in and is generally the most efficient way to develop scripts. It also contains a context menu that allows you to first select a UI element from the browser's currently displayed page and then select from a list of Selenium commands with parameters pre-defined according to the context of the selected UI element. This is not only a time-saver, but also an excellent way of learning Selenium script syntax.*

We can define Selenium test prototyping as

The process of converting a test scenario into a Selenium script that executes correctly in the Selenium IDE on the FireFox or Chrome browser.

There are a number of reasons that we do not use the IDE as our test automation tool.

1. Since we are prototyping in FireFox or Chrome, we may need to make modification to the script so that it will execute correctly for IE, Safari or other browsers.
2. There are features in the Web Driver that are not available in the IDE so the full range of possible functionality is reduced in the IDE. For example, accessing data from a database for verification a lot easier to do in the Web Driver.
3. The IDE still is not real automation since it requires user interaction to execute.
4. The auto-generated script may require tweaking in various ways before it will execute in the manner specified by the test design.

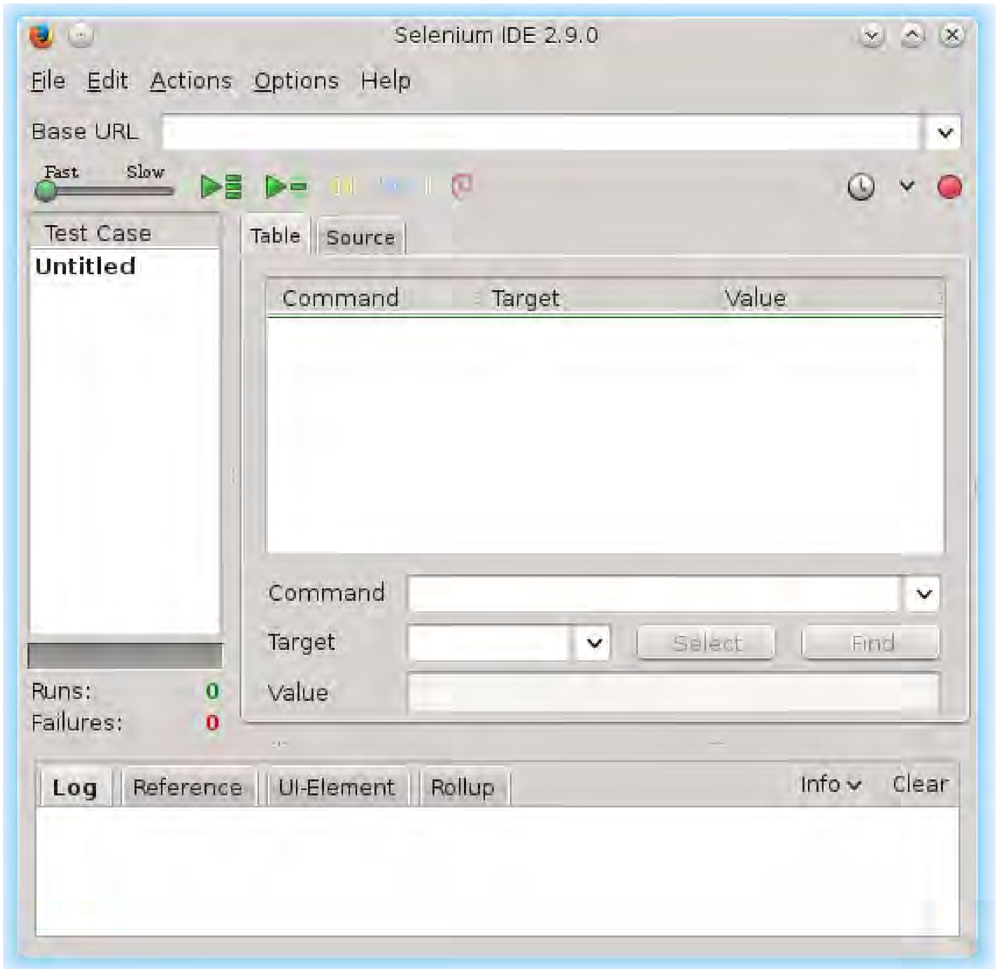
So why even prototype instead of just writing the scripts by hand? The only reason is that it is a lot easier and faster to let the Selenium IDE do the heavy lifting for you so that you can focus on the more critical aspects of getting the scripts right. The prototype of the script developed in the IDE serves as a template for the script we will then develop in Java or whatever other programming language we are using.

2 Selenium IDE Layout

In this section, we will develop a Selenium script for the Capital One Mortgage Calculator scenario we introduced in the last chapter.

The Selenium IDE

The Selenium IDE looks like this:



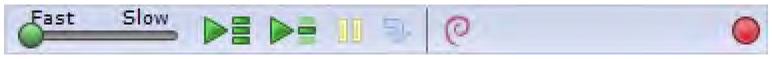
The basic components of the IDE are[1]:

The Main Menu

The File menu has options for Test Case and Test Suite (suite of Test Cases). Using these you can add a new Test Case, open a Test Case, save a Test Case, export Test Case in a language of your choice. You can also open the recent Test Case. All these options are also available for Test Suite.

The Edit menu allows copy, paste, delete, undo, and select all operations for editing the commands in your test case. The Options menu allows the changing of settings. You can set the timeout value for certain commands, add user-defined user extensions to the base set of Selenium commands, and specify the format (language) used when saving your test cases. The Help menu is the standard Firefox Help menu; only one item on this menu–UI-Element Documentation–pertains to Selenium-IDE. (From the Selenium Documentation)

Toolbar



The toolbar contains buttons for controlling the execution of your test cases, including a step feature for debugging your test cases. The right-most button, the one with the red-dot, is the record button.

Speed Control: controls how fast your test case runs.

Run All: Runs the entire test suite when a test suite with multiple test cases is loaded.

Run: Runs the currently selected test. When only a single test is loaded this button and the Run All button have the same effect.

Pause a test case.

Resume a paused test case.

Step: Allows you to “step” through a test case by running it one command at a time. Used for debugging test cases.

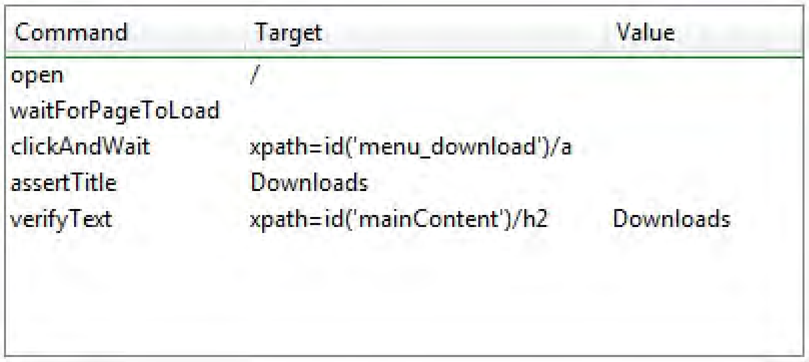
TestRunner Mode: Allows you to run the test case in a browser loaded with the Selenium-Core TestRunner. The TestRunner is not commonly used now and is likely to be deprecated. This button is for evaluating test cases for backwards compatibility with the TestRunner. Most users will probably not need this button

Apply Rollup Rules: This advanced feature allows repetitive sequences of Selenium commands to be grouped into a single action.

Record: Records the user’s browser actions.

Test Case Pane

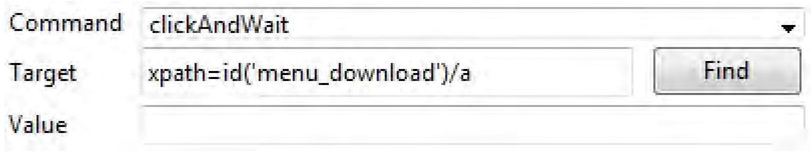
Your script is displayed in the test case pane. It has two tabs, one for displaying the command and their parameters in a readable “table” format.



Command	Target	Value
open	/	
waitForPageToLoad		
clickAndWait	xpath=id('menu_download')/a	
assertTitle	Downloads	
verifyText	xpath=id('mainContent')/h2	Downloads

The other tab - Source displays the test case in the native format in which the file will be stored. By default, this is HTML although it can be changed to a programming language such as Java or C#, or a *scripting language like Python. See the Options menu for details. The Source view also allows one to edit the test case in its raw form, including copy, cut and paste operations.*

The Command, Target, and Value entry fields display the currently selected command along with its parameters. These are entry fields where you can modify the currently selected command. The first parameter specified for a command in the Reference tab of the bottom pane always goes in the Target field. If a second parameter is specified by the Reference tab, it always goes in the Value field.



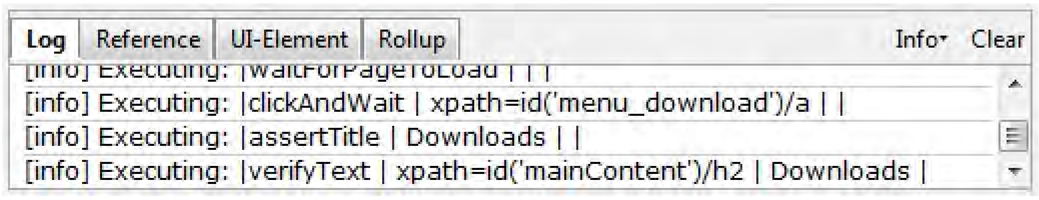
Command	clickAndWait	
Target	xpath=id('menu_download')/a	Find
Value		

If you start typing in the Command field, a drop-down list will be populated based on the first characters you type; you can then select your desired command from the drop-down.

Log/Reference/UI-Element/Rollup Pane

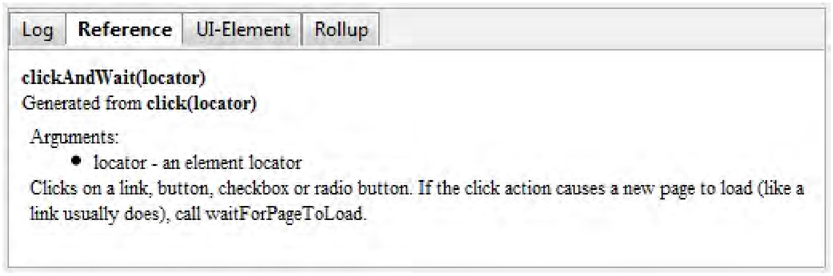
The bottom pane is used for four different functions–Log, Reference, UI-Element, and Rollup–depending on which tab is selected.

Log: When you run your test case, error messages and information messages showing the progress are displayed in this pane automatically, even if you do not first select the Log tab. These messages are often useful for test case debugging. Notice the Clear button for clearing the Log. Also notice the Info button is a drop-down allowing selection of different levels of information to log.



```
[info] Executing: |waitForPageToLoad | | |
[info] Executing: |clickAndWait | xpath=id('menu_download')/a | |
[info] Executing: |assertTitle | Downloads | |
[info] Executing: |verifyText | xpath=id('mainContent')/h2 | Downloads |
```

Reference: The Reference tab is the default selection whenever you are entering or modifying Selenese commands and parameters in Table mode. In Table mode, the Reference pane will display documentation on the current command. When entering or modifying commands, whether from Table or Source mode, it is critically important to ensure that the parameters specified in the Target and Value fields match those specified in the parameter list in the Reference pane. The number of parameters provided must match the number specified, the order of parameters provided must match the order specified, and the type of parameters provided must match the type specified. If there is a mismatch in any of these three areas, the command will not run correctly.



clickAndWait(locator)
Generated from **click(locator)**

Arguments:
- locator - an element locator

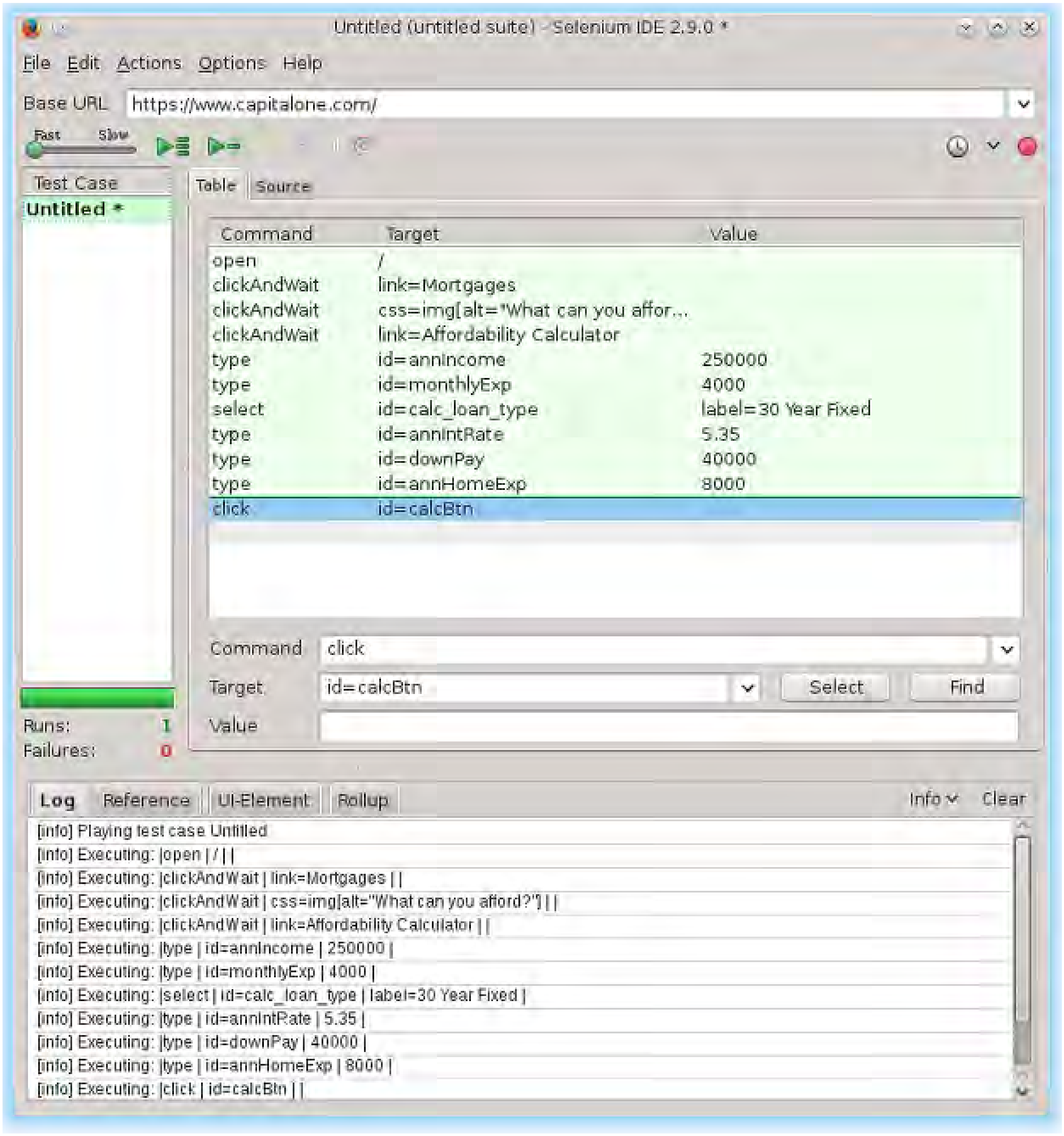
Clicks on a link, button, checkbox or radio button. If the click action causes a new page to load (like a link usually does), call waitForPageToLoad.

UI-Element and Rollup: Detailed information on these two panes (which cover advanced features) can be found in the UI-Element Documentation on the Help menu of Selenium-IDE.

3 Recording a Selenium Script

The instructor will walk through the process of recording a Selenium script – unfortunately it doesn't have the same effect seeing it on the printed page. For this script, we will just walk through the scenario presented in an earlier section with the Record button in the Selenium IDE set to "on."

If all goes well, then the result of this recording should look like the following in the Selenium IDE:


Untitled (untitled suite) - Selenium IDE 2.9.0 *
File Edit Actions Options Help
Fast Slow
Test Case
Untitled *
Table Source
Select Find
Log Reference UI-Element Rollup Info Clear

Base URL: https://www.capitalone.com/

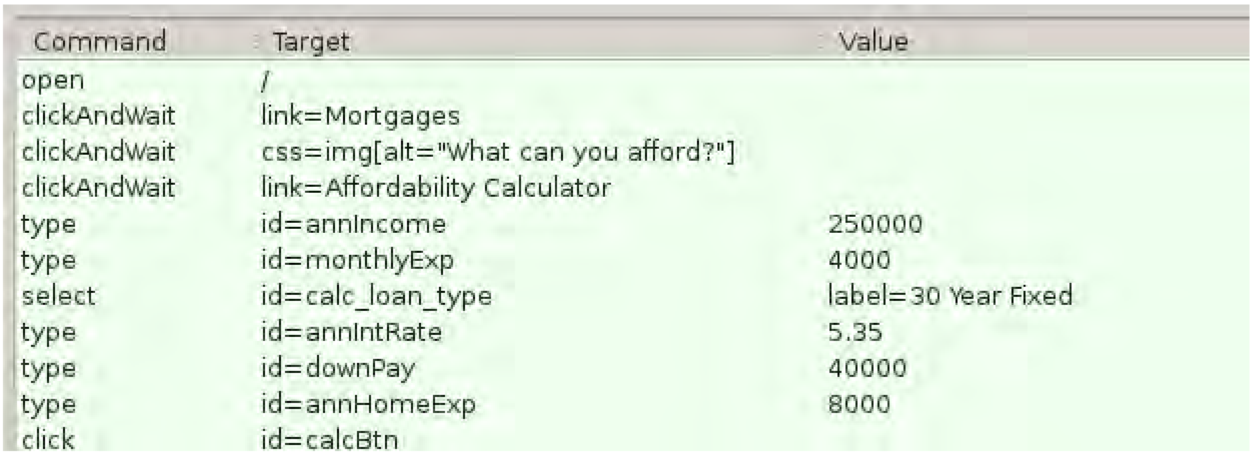
Command	Target	Value
open	/	
clickAndWait	link=Mortgages	
clickAndWait	css=img[alt="What can you affor...	
clickAndWait	link=Affordability Calculator	
type	id=annIncome	250000
type	id=monthlyExp	4000
select	id=calc_loan_type	label=30 Year Fixed
type	id=annIntRate	5.35
type	id=downPay	40000
type	id=annHomeExp	8000
click	id=calcBtn	

Command: click
Target: id=calcBtn
Value:

Runs: 1
Failures: 0

```
[info] Playing test case Untitled
[info] Executing: |open | / | |
[info] Executing: |clickAndWait | link=Mortgages | |
[info] Executing: |clickAndWait | css=img[alt="What can you afford?"] | |
[info] Executing: |clickAndWait | link=Affordability Calculator | |
[info] Executing: |type | id=annIncome | 250000 |
[info] Executing: |type | id=monthlyExp | 4000 |
[info] Executing: |select | id=calc_loan_type | label=30 Year Fixed |
[info] Executing: |type | id=annIntRate | 5.35 |
[info] Executing: |type | id=downPay | 40000 |
[info] Executing: |type | id=annHomeExp | 8000 |
[info] Executing: |click | id=calcBtn | |
```

In the table section we have our actions recorded in Selenese or the scripting language used in the Selenium IDE. Looking closer at the table window, it is relatively easy to follow the flow of the interaction between the user and the UI.



Command	Target	Value
open	/	
clickAndWait	link=Mortgages	
clickAndWait	css=img[alt="What can you afford?"]	
clickAndWait	link=Affordability Calculator	
type	id=annIncome	250000
type	id=monthlyExp	4000
select	id=calc_loan_type	label=30 Year Fixed
type	id=annIntRate	5.35
type	id=downPay	40000
type	id=annHomeExp	8000
click	id=calcBtn	

This demonstrates the actual structure of a Selenese command. An action (clicking, selecting and typing for example) is performed on a target, which is some UI widget, with an optional value involved.

What we have now is a prototype of a Selenium script that can be replayed to replicate the interaction we recorded with the UI. However now we have to tweak this prototype to make it in to an actual test scenario. For example, at this point we have no idea whether or not the right result, as specified in the test case criteria, actually occurred – we don't know if the test case passed or failed – all our script did was navigate us to the right page.

3.1 Some Selenese

There are three types of Selenese commands.

Actions

Actions are commands that modify the state of the UI in some manner. We can think of actions as the things that users do to widgets in the UI – click, select, enter text. The distinguishing feature of an action is that it changes the state of the widget or the UI.

Sometimes there is a time lag between an action and the effect of the action. For example, clicking on a link usually causes the UI to load a new page which takes a specific amount of time. If we try and click on a widget on the new page before it is loaded, we will get an error. What we can do instead is use the suffix "AndWait" to tell Selenium to wait until the action is completed before proceeding on to the next command. For example, the action "click" can be modified as "clickAndWait".

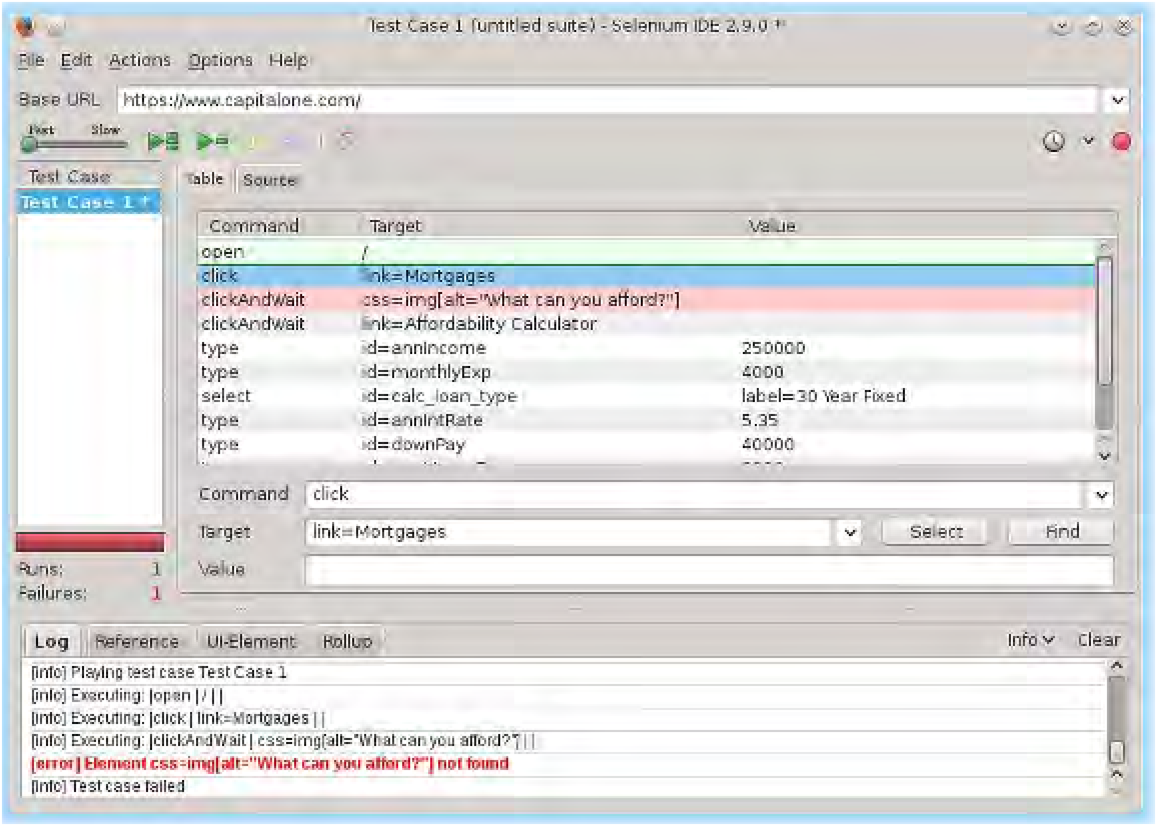
For example, if we edit the test case we just developed by changing the line

```
clickAndWait      link=Mortgages
```

to

```
click             link=Mortgages
```

then the test script will fail, as you can see from the log entry, because there was no icon to click on in the next line because the page hadn't loaded.


Test Case 1 (untitled suite) - Selenium IDE 2.9.0 *
File Edit Actions Options Help
Base URL https://www.capitalone.com/
Fast Slow
Test Case
Test Case 1 *
Table Source
Runs: 1
Failures: 1
Select Find
Log Reference UI-Element Rollup Info Clear

Command	Target	Value
open	/	
click	link=Mortgages	
clickAndWait	css=img[alt="What can you afford?"]	
clickAndWait	link=Affordability Calculator	
type	id=annIncome	250000
type	id=monthlyExp	4000
select	id=calc_loan_type	label=30 Year Fixed
type	id=annIntRate	5.35
type	id=downPay	40000

Command: click
Target: link=Mortgages
Value:

```
[info] Playing test case Test Case 1
[info] Executing: |open | / | |
[info] Executing: |click | link=Mortgages | |
[info] Executing: |clickAndWait | css=img[alt="What can you afford?"] | |
[error] Element css=img[alt="What can you afford?"] not found
[info] Test case failed
```

Accessors

Accessors report on the state of widgets is the UI and store the information in variables, for example "storeTitle". Accessors are the converse of actions – actions send data to the UI and accessors retrieve data from the UI.

Assertions

Assertions are like accessors in that they retrieve data from the UI, but they also verify that the data is correct or that it is what we expect it to be. Examples of assertions are include "make sure the page title is X" and "verify that this check-box is checked".

Assertions can be stated in three modes which differ in the effect they have after verifying data.

1. Assert: When this mode is used, such as in the command assertText, if the actual value of the text does not match the assertion expectation, then the script is aborted. We generally use assert mode when a failure at this point would make it meaningless to continue the script. For example, if we were looking for a confirmation that a log-in had succeeded in order to proceed and we did not get the log-in confirmation message, there probably would be not point in proceeding past that failure.
2. Verify: When this mode is used, such as in the command verifyText, if the actual and expected values do not match, the failure is logged but the script continues. We generally use verify when an error for a specific value is wrong, but does not impact the ability of the script to continue. For example, the wrong informational message might be displayed to the user.
3. WaitFor: This mode waits for some condition to become true. The command will proceed when the condition becomes true or will fail in the same manner as an assert if the condition fails to become true after a set timeout interval has expired.

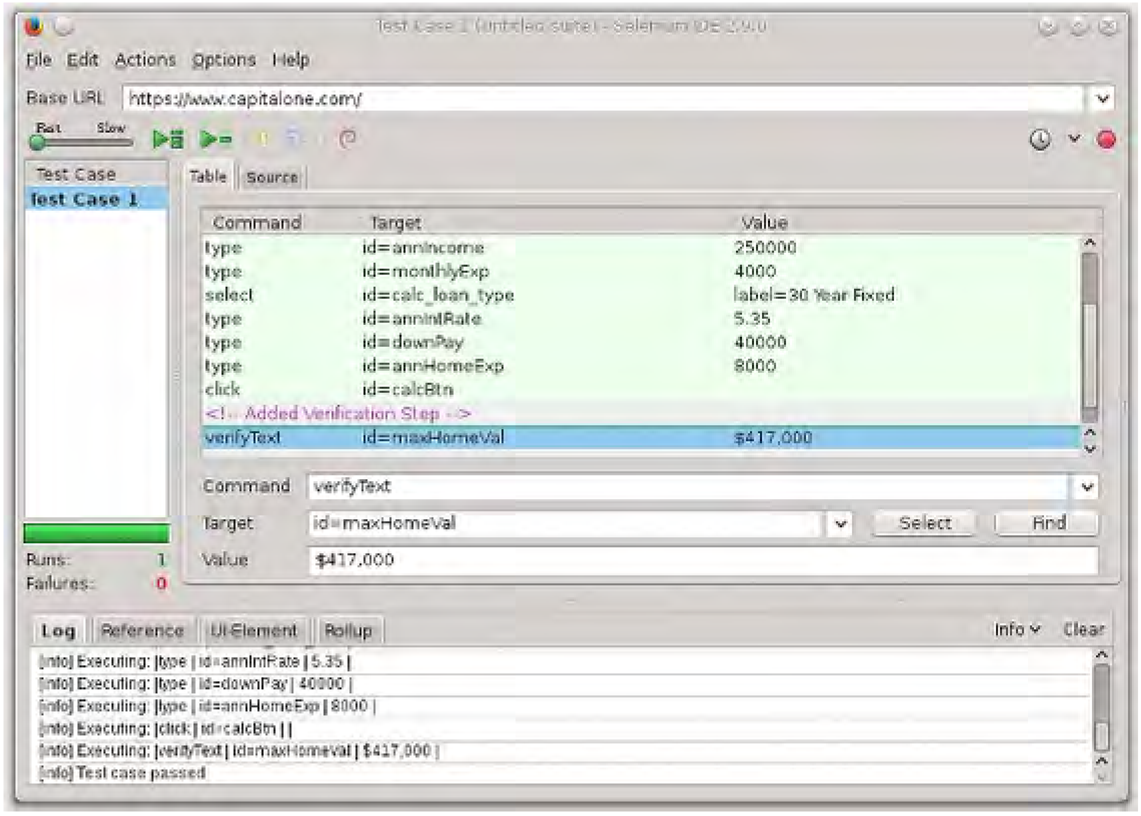
For example, in our Calculator example, we might use an assert to confirm we are on the right page. Clearly if we are not on the calculator page, there is no point to proceeding with entering the test inputs, so we should just abort the script that point and log this as a failed test.

However, once we have confirmed that we are in fact on the calculator page, then we can then use verify to check the values in the various fields (for example, confirming that the drop down list is on the "please select" option). We could then use the waitFor to ensure the result actually appears after submission.

3.2 Adding Verification to the Selenium Script

As mentioned before, the test case implemented by the script passes only if the actual result of running the script is the same as the expected value. In this case, examining the result screen, we would say that the test would pass if screen contained text corresponding to the expected result, or if some element on the page had that value. This is where we often have to be creative to ensure we are checking the right widget for the value we want to test.

In this case, inspecting the underlying HTML shows us that the value is displayed in an element with the id "maxHomeVal". The Selenium IDE provides convenient functionality through the FIND and SELECT buttons to make it easy to get an element locator. Using the editor, we can add a comment and verification step to get the following test result.



Base URL: https://www.capitalone.com/

Command	Target	Value
type	id=annIncome	250000
type	id=monthlyExp	4000
select	id=calc_loan_type	label=30 Year Fixed
type	id=annIntRate	5.35
type	id=downPay	40000
type	id=annHomeExp	8000
click	id=calcBtn	
<!-- Added Verification Step -->		
verifyText	id=maxHomeVal	$417,000

Command: verifyText
Target: id=maxHomeVal
Value: $417,000

Runs: 1
Failures: 0

```
[info] Executing: |type | id=annIntRate | 5.35 |
[info] Executing: |type | id=downPay | 40000 |
[info] Executing: |type | id=annHomeExp | 8000 |
[info] Executing: |click | id=calcBtn | |
[info] Executing: |verifyText | id=maxHomeVal | $417,000 |
[info] Test case passed
```

Notice the log entry for the verification step passing.

The other thing to notice is that the added comment is an HTML style comment. The reason for this is that the test case is actually stored as an HTML file. We can see this file by clicking the "Source" tab. In this case the first few entries of our test case would look like the HTML on the next page.

```
<tr>
      <td>open</td>
      <td>/</td>
      <td></td>
</tr>
<tr>
      <td>clickAndWait</td>
      <td>link=Mortgages</td>
      <td></td>
</tr>
```

Locating Web Elements

The Selenium commands require us to be able to specify an element on a page in order to perform an action, or to access or verify its contents or state. In our example, it was quite easy because each of the elements we wanted to access had an "id" attribute defined for it.

One of the fundamental truths of engineering is that the better something is designed, the easier it is to test, and good design always incorporates features that allow continuous and seamless testing. In terms of testing web applications, we can state a version of this design principle:

Any HTML element on a web page that will be manipulated by a user or will contain data or state information that will be accessed should have a unique ID associated with it so that it can be referred to unambiguously.

Of course this may not always be possible, consider the task of adding or changing functionality to a preexisting page for example. In this case, we may have elements without id attributes defined and so we need to reference the element by some other means.

Fortunately, Selenium allows us to use various schemes to locate elements on a page, some of which are more useful and easier to use than others, but given the range of options available to us, it is impossible to find some element on a page that we cannot write a locator expression for in some format. We will be using the same locator expressions in both the IDE and the Java Selenium scripts.

The following discussion assumes some familiarity with HTML concepts.

4.1 Locating by identifier

There are two types of attributes that can be defined in an HTML page:: id and name. The difference between the two is that an id attribute must be unique on a page and therefore uniquely identifies a specific UI element. The name attribute does not have to be unique which means that there may be more than one UI element with the same name. As well an element can have both a name and id attribute.

The relationship of name and id may seem a bit awkward, but it is an artifact of the evolution of the HTML standard. Early versions of HTML did not have the id attribute but did have the name attribute, without the restriction that a name had to be unique within a page.

Later versions of HTML introduced the id attribute to provide a unique identifier for elements but kept the name attribute as well. One reason for this, aside from backwards compatibility, is that the id attribute tended to be used to identify significant chunks of the HTML like forms, divs, lists and other container sort of tags. Names tended to be used for items within a container like input elements in a form or options in a drop down list. Inside an form, for example, we only needed to differentiate input elements locally from each other and not differentiate them from input items in other forms. To provide unique names for each submit button in multiple forms on a page is not only difficult but also tedious and unnecessary.

Using id as a locator

This is clearly the nicest and cleanest way to do it. Consider the follow HTML snippet.

```
<html>
  <body>
   <form id="loginForm">
    <input name="username" type="text" />
    <input name="password" type="password" />
    <input name="continue" type="submit" value="Login" />
    <input name="continue" type="button" value="Clear" />
   </form>
 </body>
 <html>
```

Using the locator

```
id=loginForm
```

finds

```
<form id="loginForm"> ... </form>
```

Using name as a locator

In the above snippet of code, using the locator

```
name=continue
```

matches two elements that have same name. By default, it finds the first element with that name, which in this case is:

```
<input name="continue" type="submit" value="Login" />
```

Unfortunately, this may not be the one we want. When we are locating by name we also have the option of providing filters, which are extra bits of information in other attributes that allow us to uniquely identify an element. If we used the locator:

```
name=continue value=clear
```

There is only one element that has that name and that value

```
<input name="continue" type="button" value="Clear" />
```

Using identifier as a locator

The locator identifier is the default in Selenium and is either an element's id or its name attribute depending on which ones are present on the element. If we do not specify what locator type we are using, the locator type defaults to identifier.

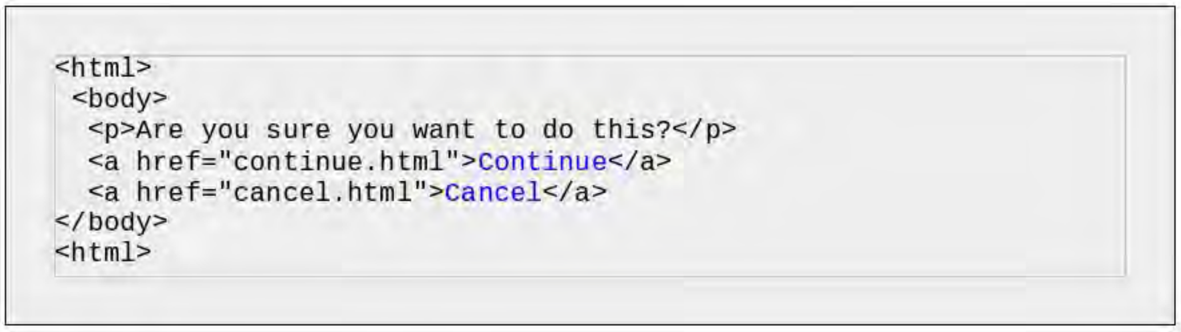
Using the locator

```
identifier=LoginForm    finds the form element
indentifier=continue    finds the input element value=”Login”
LoginForm               defaults to identifier=LoginForm
continue                defaults to identifier=continue
```

The default form in the last two lines above is not recommended and the use of just identifier is also discouraged since it is ambiguous to anyone reading the script.

4.2 Locating by Link Text

A common action we want to do in a script is to click on a link. Links typically do not have identifiers so we can look for links that have specific link text. For example.



```
<html>
 <body>
  <p>Are you sure you want to do this?</p>
  <a href="continue.html">Continue</a>
  <a href="cancel.html">Cancel</a>
</body>
<html>
```

In the above snippet, the link text is highlighted in blue. We could locate the second link element with the selector:

```
link=Cancel
```

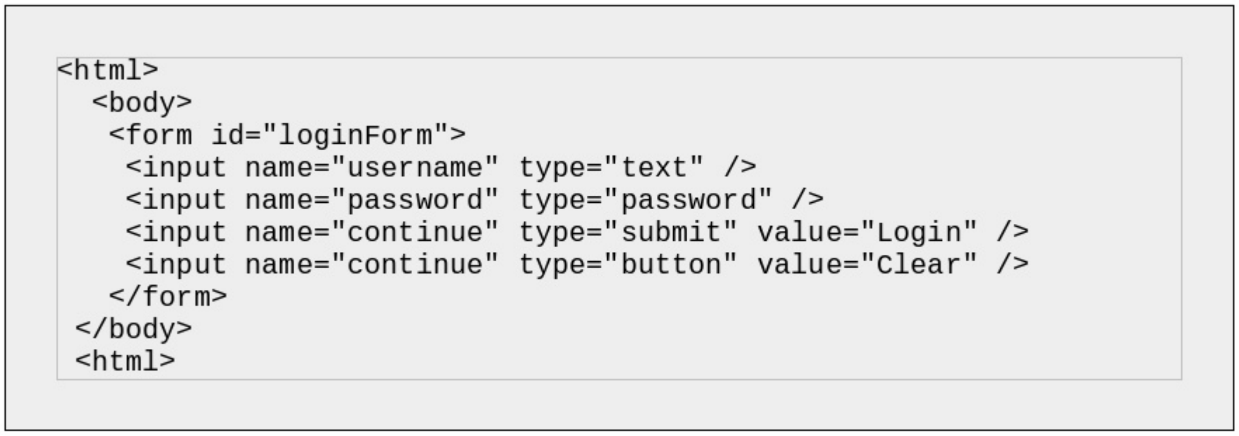
However this method is not recommended in a production environment for several reasons.

1. There is no assurance that the link text will be unique.
2. The link text may change at any time.
3. The link text may be dynamically generated.

4.3 Locating by DOM

The Document Object Model (DOM) is a representation of a web page so that the various elements can be accessed by JavaScript code. A web page is converted into a DOM tree in a browser which can then be traversed from the root to find any the DOM object corresponding to any element on the page.

The exploration of DOM is beyond the scope of this course, but an excellent description of DOM can be found at: http://www.w3schools.com/dom/



```
<html>
  <body>
   <form id="loginForm">
    <input name="username" type="text" />
    <input name="password" type="password" />
    <input name="continue" type="submit" value="Login" />
    <input name="continue" type="button" value="Clear" />
   </form>
 </body>
 <html>
```

In this form of locator, we essentially write the JavaScript expression that evaluates to the DOM node that corresponds to the element we want to locate.

All of the following locators are equivalent because they evaluate to the form element.

```
dom=document.getElementById('loginForm')
```

```
dom=document.forms['loginForm']
document.forms[0]
```

Notice that the last one does not have the “dom=” part – any locator that starts with “document.” is assumed to be DOM expressions.

As another example, all of the following locators evaluate to the username input element.

```
document.forms[0].username
document.forms[0].elements['username']
document.forms[0].elements[0]
```

Some authors have recommended that DOM not be used and replaced by CSS or XPath selectors for various reasons. One is that the DOM implementations are not uniform across browsers and another is that the DOM locators are too dependent on the actual page structure so that changes to structure of the page will cause the locator to fail – a concern when pages are dynamically generated.

A more practical concern for many testers is that it does require an understanding of how JavaScript and the DOM model work, which is a rather heavyweight technology.

4.4 Locating by XPath

Another scheme for locating elements by position in a page is XPath. The XPath standard provides a syntax for navigating through the structure of an XML or HTML document. It is part of the XSLT standard and is a component of both the XQuery and XPointer standards. A full discussion of XPath is beyond the scope of this course but this link http://www.w3schools.com/Xpath/ is a good starting point to explore the standard and how to work with XPath expressions.

In the same code snippet, we can use XPath expressions to locate the objects in the following ways. First to find the form element:

```
xpath=/html/body/form[1]
xpath=//form[@id='loginForm']
xpath=//form[input/@name='username']
//form[1]
```

In the last locator, we can leave out the xpath= part and implicitly tell Selenium this is an XPath by starting the locator with a double slash, similar to the dom case staring with a document.

XPath allows for very precise specification of elements, however there are a couple of downsides to using XPath:

1. Like DOM, it locates an element in terms of its location in the structure of the page. Changing the structure of the page can invalidate the expression.
2. Microsoft IE is known for an incomplete implementation of XPath which means that an expression may fail in that browser environment.
3. Compared to CSS, finding elements by XPath is very slow.

4.5 Locating by CSS

CSS is the W3C cascading style sheets standard. Since a styles sheet has to specify a particular HTML element in order to apply a style, CSS has what are called selectors to allow it to specify specific elements in and HTML page.

CSS uses both structural information in its selectors, like DOM and XPath, but also uses attribute and other information like the id and name locators do. As with the other standards, a discussion of CSS and CSS selectors is beyond the scope of this course. A good tutorial on CSS selectors can be found at: http://www.w3.org/TR/css3-selectors/

Using our standard code snipped, the following CSS locator find the form element.

```
css=form#loginForm
```

While the following CSS locators find the user name text field element:

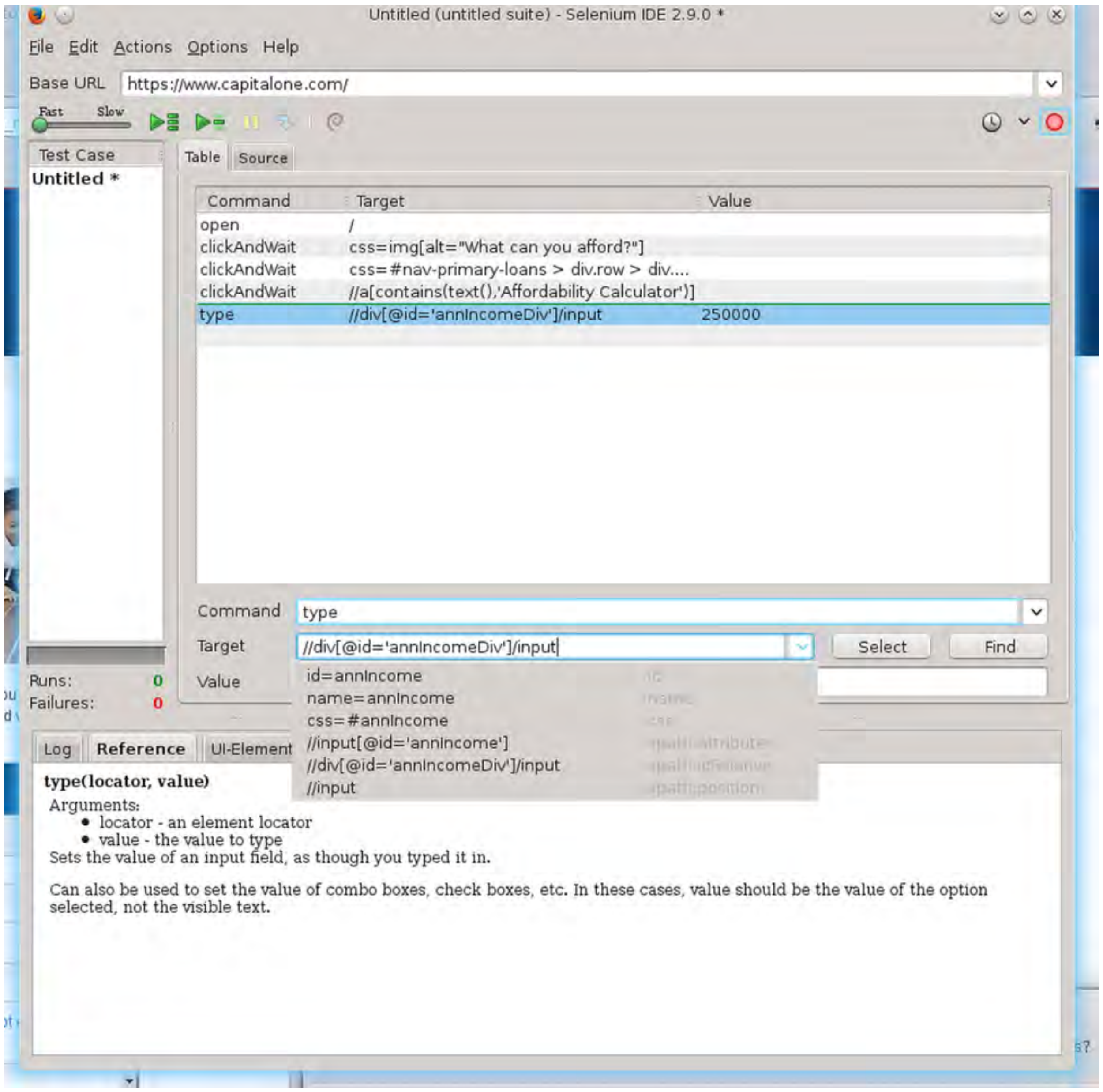
```
css=input[name="username"]
css=input.required[type="text"]
```

4.6 Locator Tools

Fortunately, we do not have to be masters of XPath or CSS in order use the locators. The Selenium IDE does provide a way for us to choose the type of locator we want to use while we are recording the script. This is one of the benefits of prototyping the script in the Selenium IDE.

As a line is added to the script, we can edit the locator by selecting the line and using the drop down list in the target field of the editor. In the screen shot below, I have added CSS and XPath locators to a re-recording of our original script using this method.

We can also go back and edit the selectors on our existing scripts but we cannot do that in the Selenium



Base URL https://www.capitalone.com/

Untitled *

Command	Target	Value
open	/	
clickAndWait	css=img[alt="What can you afford?"]	
clickAndWait	css=#nav-primary-loans > div.row > div....	
clickAndWait	//a[contains(text(),'Affordability Calculator')]	
type	//div[@id='annIncomeDiv']/input	250000

Command type

Target //div[@id='annIncomeDiv']/input

id=annIncome	id
name=annIncome	name
css=#annIncome	css
//input[@id='annIncome']	xpath:attributes
//div[@id='annIncomeDiv']/input	xpath:idRelative
//input	xpath:position

type(locator, value)

Arguments:

- locator - an element locator
- value - the value to type

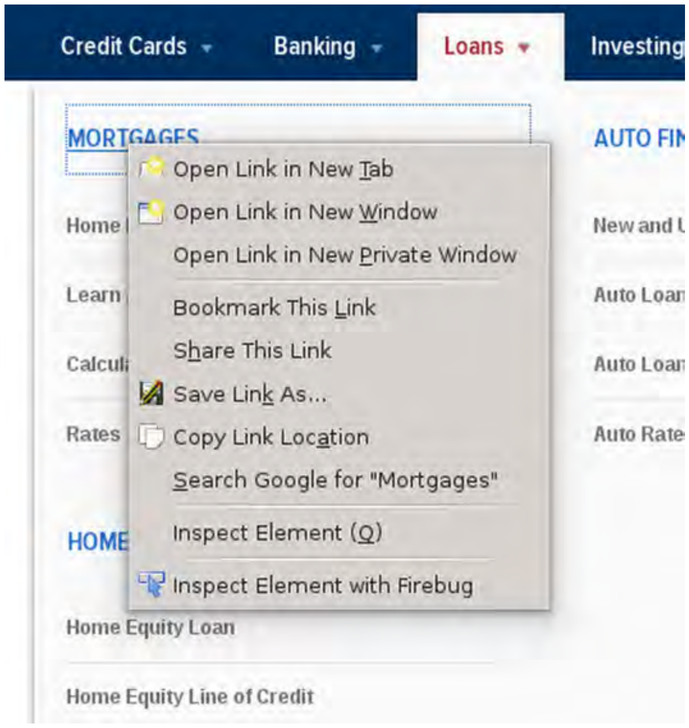
Sets the value of an input field, as though you typed it in.

Can also be used to set the value of combo boxes, check boxes, etc. In these cases, value should be the value of the option selected, not the visible text.

IDE since the IDE needs access to the underlying web page's HTML to provide the options in the drop down list.

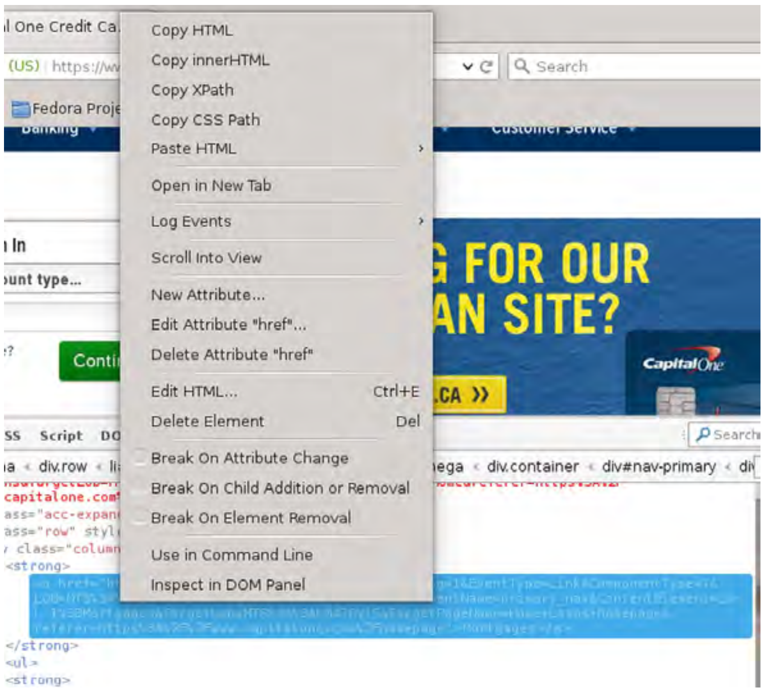
However, there is another way. If we open a page, we can use FireBug to examine an element and then copy the XPath or CSS for that element to the clipboard.

If you start Firebug and right mouse click on an element, the menu option inspect element with FireBug



appears.

Once we have the FireBug window open, we can look at the XPath and CSS paths to that element.



5 Working with Text

So far in this module, we have been using exact values for verification However this is very limiting because we may not be interested in an exact match with a string but whether or not a string matches a specific pattern For example, in verifying a phone number, we may not care about the specific phone number displayed but only that it looks like “(ddd) ddd-dddd” where “d” is a digit from 0-9.

As a reminder, all input we provide in a script to a UI element must be exact and not a pattern. Obviously.

There are three kinds of patterns that we can be used in commands like verifyText and click.

Globbing Patterns

Globbing is the system of using wild cards in standard shell programming. We are not going into the details of globbing. If you need to understand globbing, there are a large number of tutorials available on line.

However, while globbing normally supports the wild cards [ ] and * and ?, Selenium does NOT support the “?” globbing character.   As an examples, suppose we want to select a link on the Capital One web site that has the words “Local” and “Contact” in that order somewhere in the title.

```
            click       link=Glob:*[lL]ocal*[cC]ontact*
```

This would match the strings:

```
            Local Loan Manager Contact Information
            Agents who are local that you can contact
```

And so on...

Regular Expressions

Regular expressions are a pattern matching syntax that is more or less standardized. A discussion of regular expressions is beyond the scope of this course. Any one interested in learning more about regular expressions can find more information at either of these links as well as many other tutorials and sites on line.

The regular expressions used in Selenium are derived from the JavaScript usage because Selenium 1 was originally JavaScript based..

In the following verify, we want to ensure that the output is in the right form.

```
      verifyText   id=sunrise       regexp:Sunrise: *[0-9]{1,2}:[0-9]{2}
   [ap]m
```

This will match a string consisting of the string “Sunrise: ” followed by one or two digits, a colon, to other digits a space and then either am or pm.

Exact Expressions

Since globbing is the default, the command

```
            verifyText  id=SelChar        *error info*
```

is a problem because the asterisks would be interpreted as globbing characters. We can override this by telling Selenium that we want to match the exact string.

```
            verifyText  id=SelChar        Exact:*error info*
```

Now we consider everything in the string just a plain old character without any special meaning like in regular expressions or a globbed expression.

Notes and References

1. *All text taken from the Selenium documentation is italicized. This material has been reproduced here for convenience in classroom environment.. The text may be found at: http://docs.seleniumhq.org/docs/02_selenium_ide.jsp*

Module Three

Selenium WebDriver

The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency.

Bill Gates

A computer lets you make more mistakes faster than any other invention with the possible exceptions of handguns and Tequila.

Mitch Ratcliffe

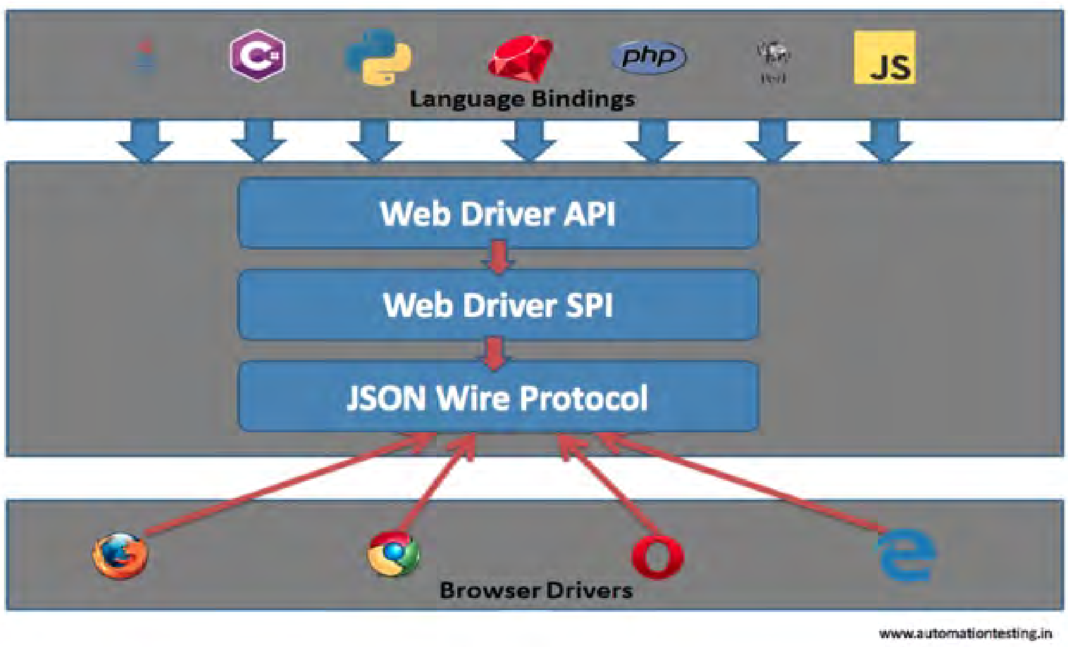
If you automate a mess, you get an automated mess.

Rod Michael

1. Web Driver Architecture

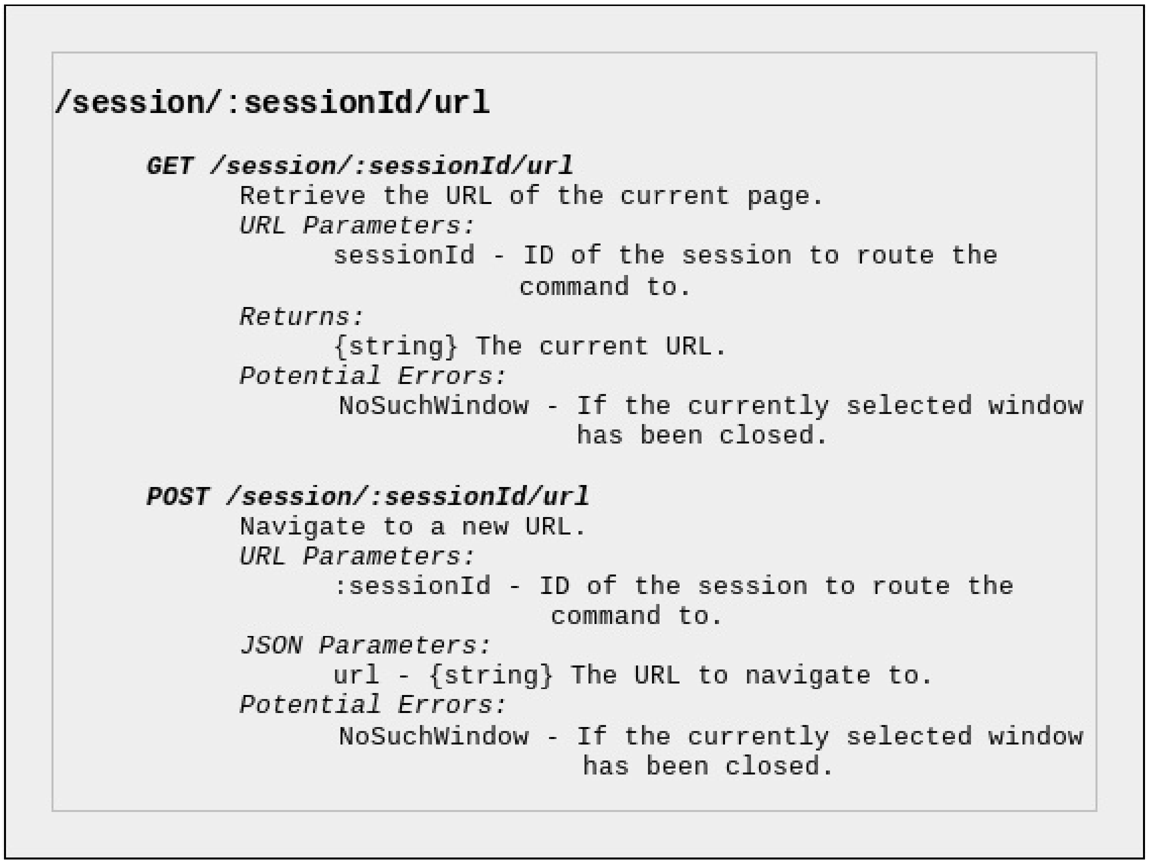
To recap what we covered in module one, the Web driver consists of three components.

1. **The Web Driver API:** This is the code that actually does the work of interpreting our commands and delivering them to the browser drivers.
2. **The Language Bindings:** These are implementations of the API in various programming languages that allow us to write tests in Java, C#, Python or whatever else is supported, then have them converted into commands that the Web Driver API understands.
3. **The Browser Drivers:** For each supported browser, there is a browser driver that converts the API commands into a form that can be understood by the specific browser.



The API converts the commands in a specific programming language into what is called the Selenium Web Driver Wire Protocol which is then passed to the browser driver. The wire protocol uses the Restful web service protocol and JSon messages.

We will not go into this protocol in this class, but just as an example, suppose that in Java we specify that we want to navigate to a new URL. The Java language binding allows the Selenium Java client to make a request to the WebDriver API that is converted to a specific Restful command. In the box on the next page, we can see the protocol command that would be sent to a specific browser driver.



```
/session/:sessionId/url

     GET /session/:sessionId/url
          Retrieve the URL of the current page.
          URL Parameters:
               sessionId - ID of the session to route the
                           command to.
          Returns:
               {string} The current URL.
          Potential Errors:
               NoSuchWindow - If the currently selected window
                              has been closed.

     POST /session/:sessionId/url
          Navigate to a new URL.
          URL Parameters:
               :sessionId - ID of the session to route the
                            command to.
          JSON Parameters:
               url - {string} The URL to navigate to.
          Potential Errors:
               NoSuchWindow - If the currently selected window
                              has been closed.
```

1.1 Project Supported Language Bindings

There are a number of officially supported language bindings for Selenium:

1. Java
2. Ruby
3. C#
4. js.Node
5. Python

By officially supported we mean that these are hosted in the official Selenium project repositories and under the direct control of the Selenium project.

In this class, we will only be looking at Java and Python. However, there are other language bindings available which are supported externally by third parties that are not part of the Selenium platform but are part of the Selenium ecosystem. Some of these languages are:

1. PERL
2. PHP
3. Haskell
4. Objective-C
5. JavaScript
6. R
7. Tcl

8. Dart
9. Haskell
10. Elixir

1.2 Supported Browsers

Drivers do exist for most of the browsers. Originally the FireFox driver was part of the core project and was built into the WebDriver. However, changes in the architecture of FireFox rendered this driver obsolete and it was replaced with the externally supported Gecko Driver which must be used for all version of FireFox after version 54.

In this class, we will only work with either the Chrome or FireFox browsers for simplicity since everything we will be doing is essentially the same no matter what browser we use.

Because of the popularity of Selenium and the looking W3C standard, the organizations that offer growers are also supplying browser drivers to work with their own product. Supporting any browser usually requires downloading the browser driver and making it available to the Web Driver.

Supporting browsers is more complex than supporting language bindings. The first reason is that browsers all interpret the various web page components differently. Anyone who has designed a production website knows that the designer always has to tweak some elements so that they don't break when certain browsers try to render or process them.

The second reason is that we also have to account for different versions of browsers and the underlying platform's operating system. However these issues are all beyond the scope of this course.

1.3 Set Up Requirements

The system set up requirements for starting off with Selenium are quite simple and won’t be covered in class beyond a simple overview. There are innumerable tutorials and guides on the Internet to walk you through the process.

For this course, the following steps were done to set up the environment:

1. The Java jdk 1.8 and Python 3.7 were installed.
2. The Eclipse Java IDE was installed and VSCode with Python extensions was installed for developing code in each language.
3. The Java Selenium jars were downloaded from the SeleniumHQ website.
4. Selenium for Python was installed using "pip".
5. The Chrome and Gecko drivers were downloaded and placed in the "C:\bin" directory that was then placed on the executable path.

No integrated development environments were installed, nor were any testing frameworks except those that came bundled with Python or Eclipse. If you want to replicate this set up, most of what was done here can be duplicated using tutorials available on the web.

2. Setting up Selenium

This first section will deal with setting up environment to run Selenium scripts. We will not go into the details of setting up Eclipse or Python because those details are available on many tutorial websites. For now, the instructor will walk you through the process and then you will try it on your own in Lab One.

For both of the setups below to work, the browser driver executables must be in a directory that is referenced in the "path" variable.

Java Setup

The only thing that we have to ensure is that Java has access to the Selenium jars. In the eclipse project, we do this by adding all the jars to the project build path as external archives. The instructor will walk through the process and then you will do the same.

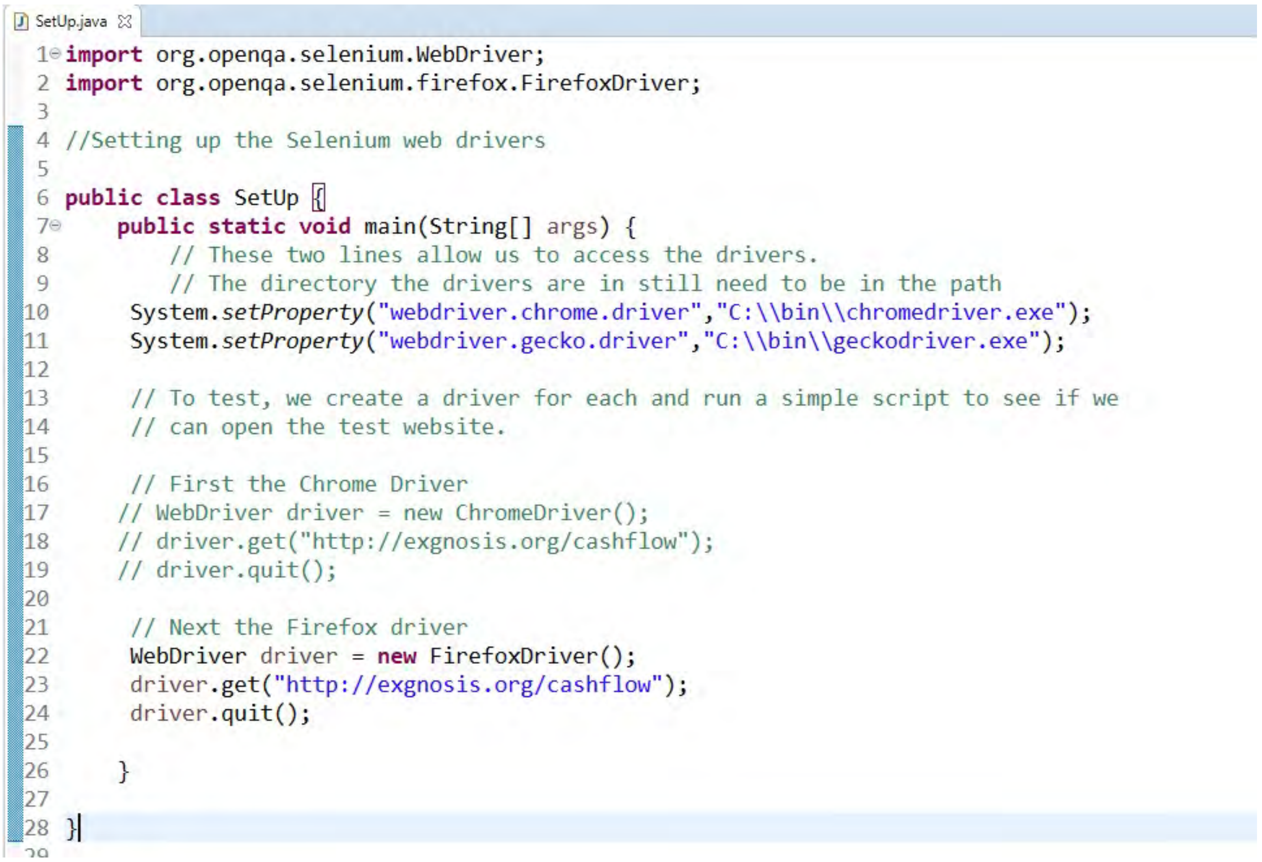
The other detail that needs to be included is the "System.setProperties()" line that registers where the browser executables are located.

Python Setup

Assuming that Selenium has been installed with "pip" then no further setup is required.

Verifying the Setup

The instructor will demo the setup using the test site "http://exgnosis.org/cashflow" for both Java and Python.



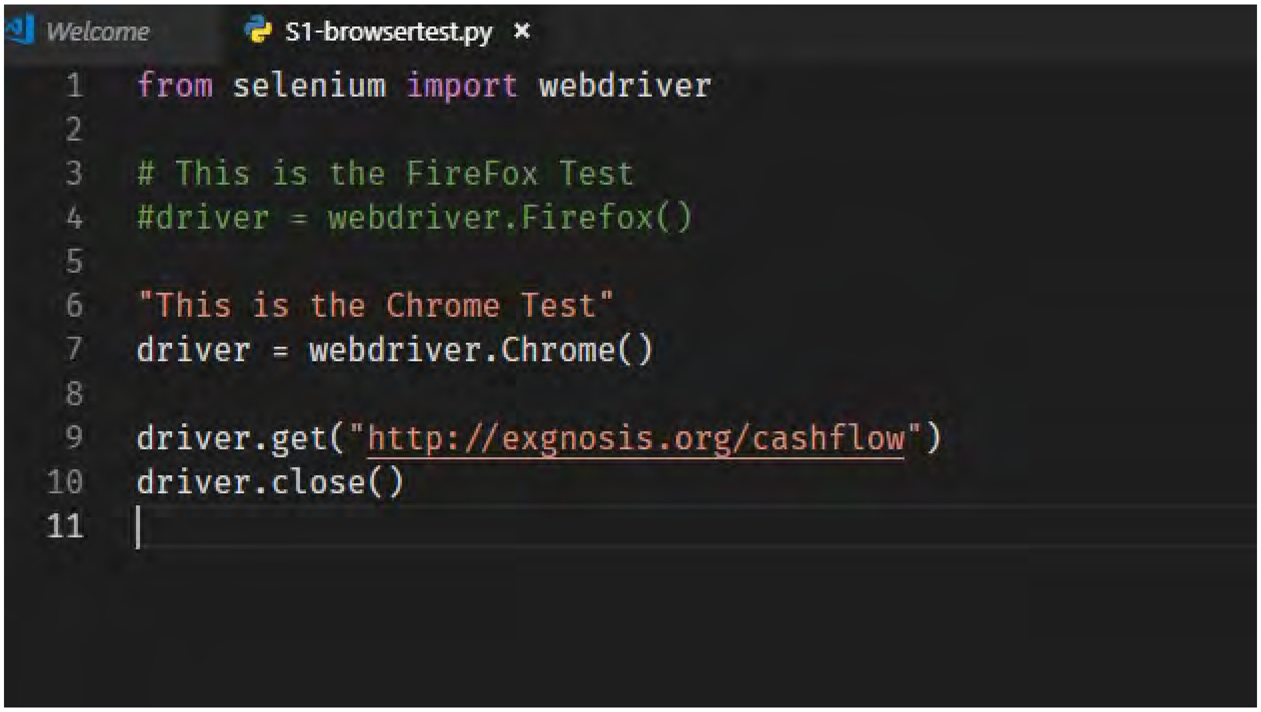
SetUp.java

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

//Setting up the Selenium web drivers

public class SetUp {
    public static void main(String[] args) {
        // These two lines allow us to access the drivers.
        // The directory the drivers are in still need to be in the path
     System.setProperty("webdriver.chrome.driver","C:\\bin\\chromedriver.exe");
     System.setProperty("webdriver.gecko.driver","C:\\bin\\geckodriver.exe");

     // To test, we create a driver for each and run a simple script to see if we
     // can open the test website.

     // First the Chrome Driver
    // WebDriver driver = new ChromeDriver();
    // driver.get("http://exgnosis.org/cashflow");
    // driver.quit();

     // Next the Firefox driver
     WebDriver driver = new FirefoxDriver();
     driver.get("http://exgnosis.org/cashflow");
     driver.quit();

    }

}
```

Switching Browsers

The actual code is available in the lab manual but switching which browser we use is simple a matter of changing one line in the code. For example, for Java is shown on the previous page.

And for Python:

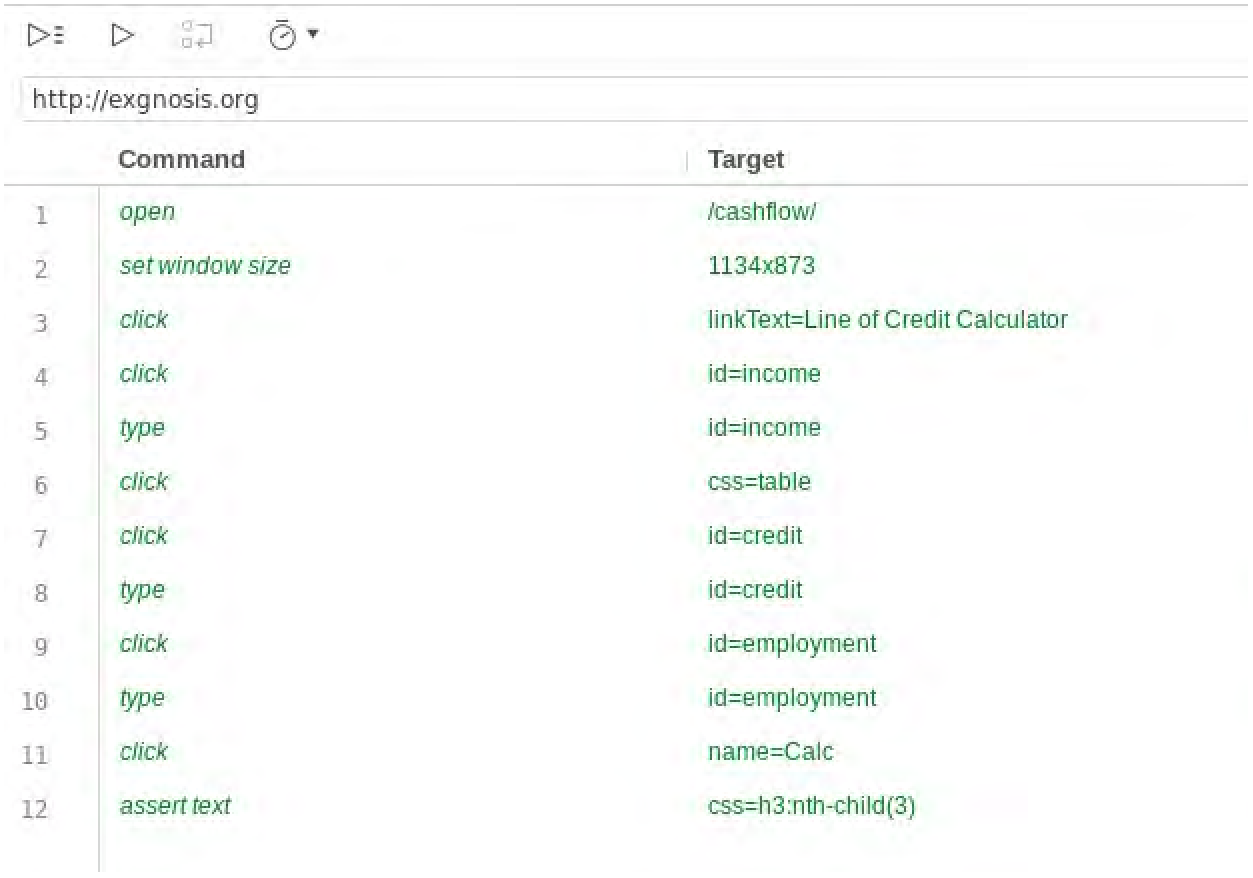


```
from selenium import webdriver

# This is the FireFox Test
#driver = webdriver.Firefox()

"This is the Chrome Test"
driver = webdriver.Chrome()

driver.get("http://exgnosis.org/cashflow")
driver.close()

```

The instructor will walk through the process of setting up the projects in both Java and Python and then you will replicate the process yourself in either language.

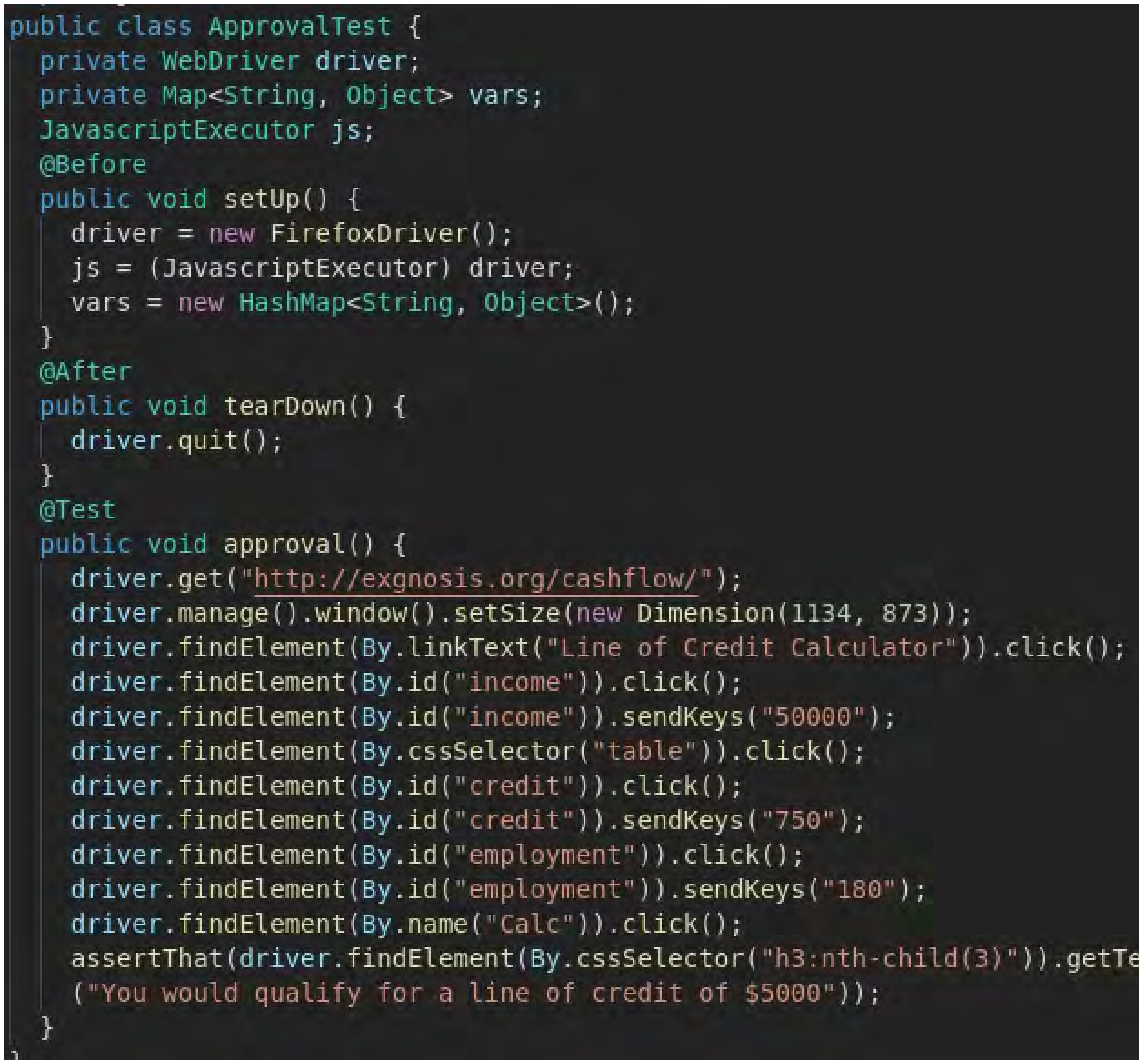
2.1 Generated Scripts

Once a script has been generated in the Selenium IDE, it can be exported as a Java JUnit test script. The previous IDE used to support a number of different language exports, but the current release only supports Java – although there are promises from the developers that the options for other languages will be restored.



http://exgnosis.org

	Command	Target
1	*open*	/cashflow/
2	*set window size*	1134x873
3	*click*	linkText=Line of Credit Calculator
4	*click*	id=income
5	*type*	id=income
6	*click*	css=table
7	*click*	id=credit
8	*type*	id=credit
9	*click*	id=employment
10	*type*	id=employment
11	*click*	name=Calc
12	*assert text*	css=h3:nth-child(3)



```
public class ApprovalTest {
  private WebDriver driver;
  private Map<String, Object> vars;
  JavascriptExecutor js;
  @Before
  public void setUp() {
    driver = new FirefoxDriver();
    js = (JavascriptExecutor) driver;
    vars = new HashMap<String, Object>();
  }
  @After
  public void tearDown() {
    driver.quit();
  }
  @Test
  public void approval() {
    driver.get("http://exgnosis.org/cashflow/");
    driver.manage().window().setSize(new Dimension(1134, 873));
    driver.findElement(By.linkText("Line of Credit Calculator")).click();
    driver.findElement(By.id("income")).click();
    driver.findElement(By.id("income")).sendKeys("50000");
    driver.findElement(By.cssSelector("table")).click();
    driver.findElement(By.id("credit")).click();
    driver.findElement(By.id("credit")).sendKeys("750");
    driver.findElement(By.id("employment")).click();
    driver.findElement(By.id("employment")).sendKeys("180");
    driver.findElement(By.name("Calc")).click();
    assertThat(driver.findElement(By.cssSelector("h3:nth-child(3)")).getTe
    ("You would qualify for a line of credit of $5000"));
  }
```

3. Page Navigation Commands

The first set set of commands are the basic browser navigation commands. In all of the commands, we are making a request to the “driver” which is a reference to the running Selenium Web Driver. Depending on the type of driver we have instantiated, it will relay our requests to a specific browser.

The line in the code

```
FirefoxDriver driver = new FirefoxDriver();
driver = webdriver.Firefox()
```

creates a driver object that we can send requests to that will be then sent to an instance of the FireFox browser. We can then issue commands to perform various browser navigation operations.

Some of these commands are listed below. Note that for Python, these are not all commands but some are properties of the driver.

```
driver.get(URL);
driver.get(URL)
```

Opens a new web page in the current browser.

```
driver.getTitle();
driver.title
```

This command returns the title of the current page.

```
driver.getCurrentUrl();
driver.current_url
```

Returns as a String the URL of the page currently loaded in the browser.

```
driver.getPageSource();
driver.page_source
```

Returns as a String, the HTML source of the last loaded page.

```
driver.close();
driver.close()
```

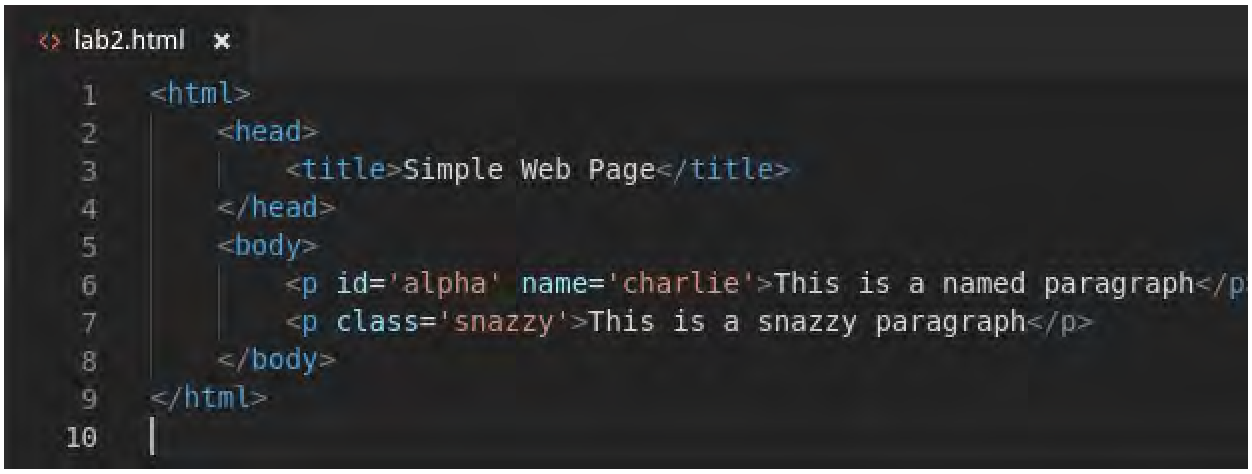
Closes the current window or quits the browser if there is only one window, if the current window is the only window, it will close the browser.

```
driver.quit();
driver.quit()
```

Quits the browser and closes all windows in the browser.

Example

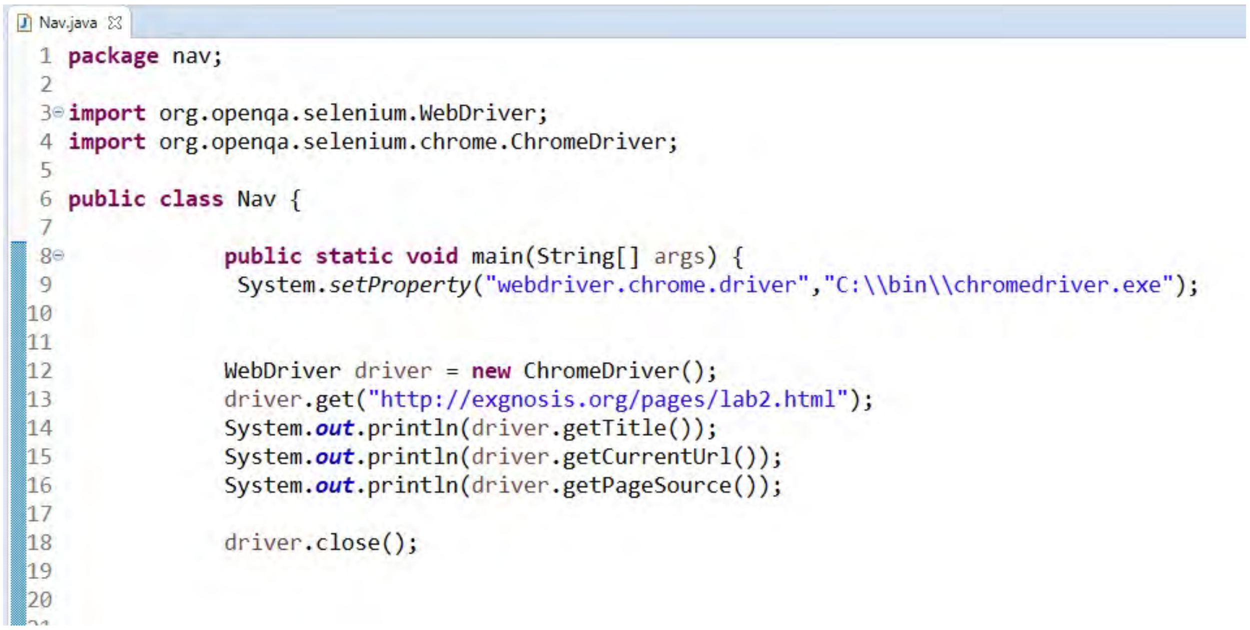
The screen shots show these commands in use using a sample webpage shown below:



lab2.html

```
<html>
    <head>
        <title>Simple Web Page</title>
    </head>
    <body>
        <p id='alpha' name='charlie'>This is a named paragraph</p
        <p class='snazzy'>This is a snazzy paragraph</p>
    </body>
</html>
```

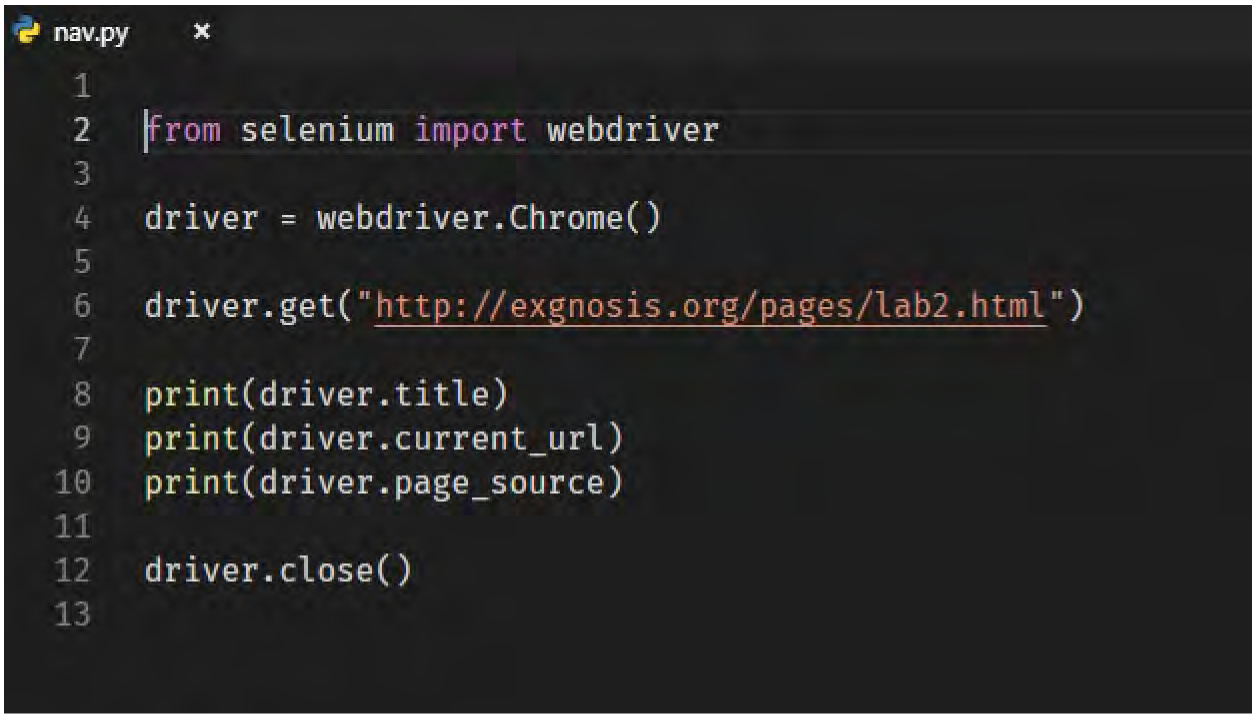
First the Java:



Nav.java

```
package nav;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class Nav {

            public static void main(String[] args) {
             System.setProperty("webdriver.chrome.driver","C:\\bin\\chromedriver.exe");


            WebDriver driver = new ChromeDriver();
            driver.get("http://exgnosis.org/pages/lab2.html");
            System.out.println(driver.getTitle());
            System.out.println(driver.getCurrentUrl());
            System.out.println(driver.getPageSource());

            driver.close();
```

Now the Python:



```python

from selenium import webdriver

driver = webdriver.Chrome()

driver.get("http://exgnosis.org/pages/lab2.html")

print(driver.title)
print(driver.current_url)
print(driver.page_source)

driver.close()

```

The instructor will walk you through performing the hands on for these.

The following commands are exposed through the driver.navigate() interface in Java but are accessed as just regular methods in Python.

```
driver.navigate().to(URL);
```

Navigates to a specific URL

Java only, Python uses get()

```
driver.navigate().forward();
driver.forward()
```

Emulates the effect of using the “next” button on a browser.

```
driver.navigate().back();
driver.back()
```

Emulates the effect of using the “next” button on a browser.

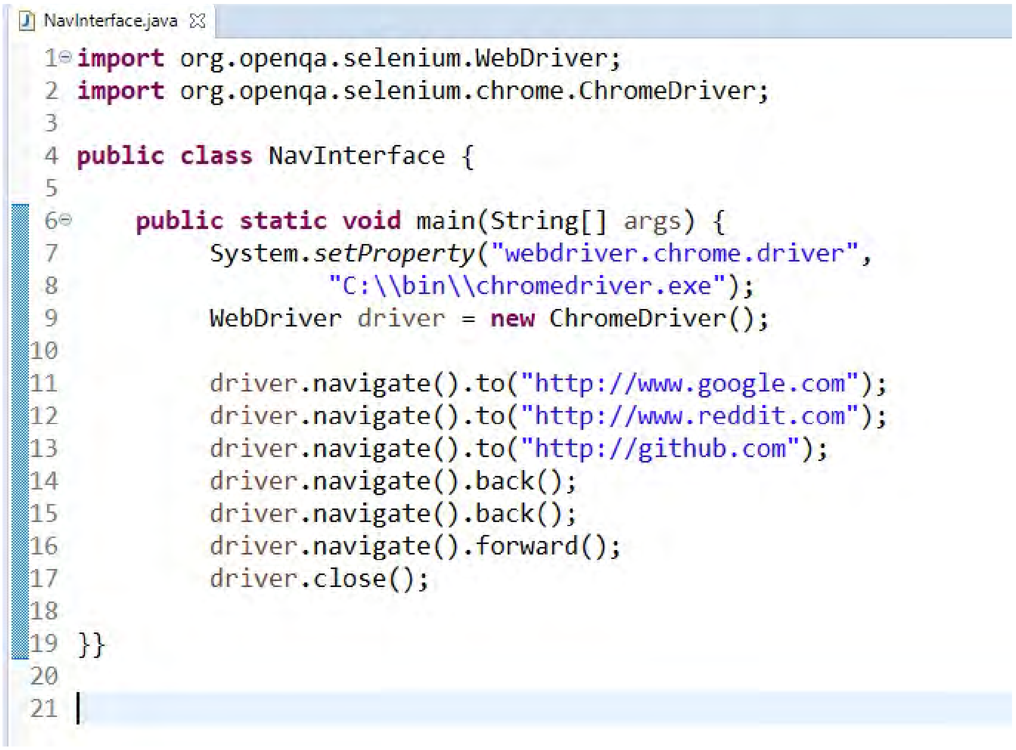
```
driver.navigate().refresh();
driver.refresh()
```

Refreshes the current browser page

Example 2

The screen shots show the navigation commands being used to move from page to page.

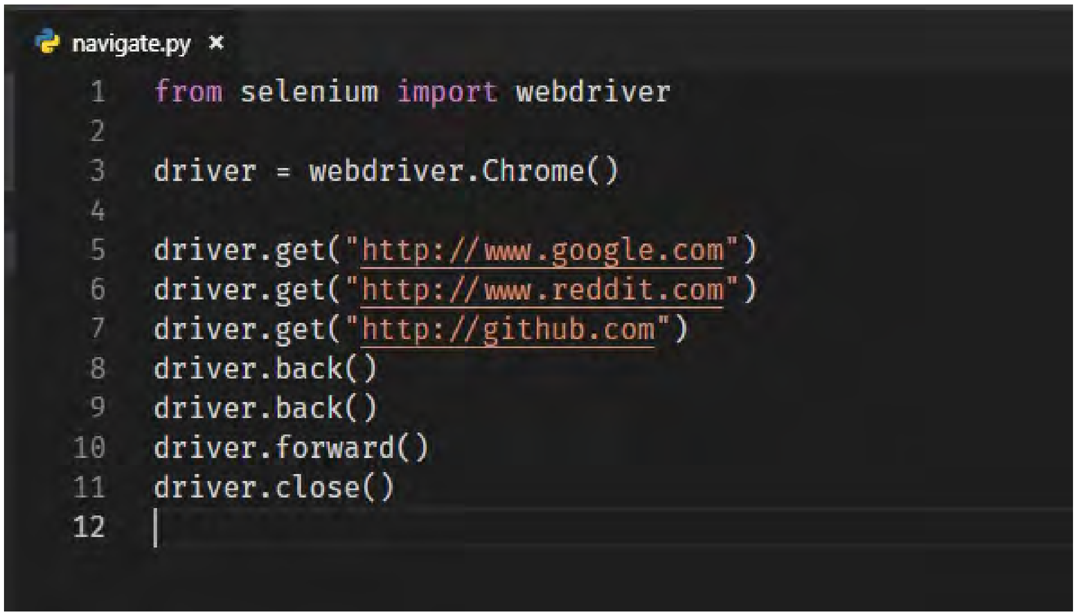
First the Java:



NavInterface.java

```java
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class NavInterface {

    public static void main(String[] args) {
        System.setProperty("webdriver.chrome.driver",
                "C:\\bin\\chromedriver.exe");
        WebDriver driver = new ChromeDriver();

        driver.navigate().to("http://www.google.com");
        driver.navigate().to("http://www.reddit.com");
        driver.navigate().to("http://github.com");
        driver.navigate().back();
        driver.navigate().back();
        driver.navigate().forward();
        driver.close();

}}

```

Now the Python:



navigate.py

```python
from selenium import webdriver

driver = webdriver.Chrome()

driver.get("http://www.google.com")
driver.get("http://www.reddit.com")
driver.get("http://github.com")
driver.back()
driver.back()
driver.forward()
driver.close()

```

The instructor will walk you through performing the hands on for these.

4. Finding elements

We saw in the test script we generated from the Selenium IDE file how the locator operations looked in the Java code.

In this section, we will look at and try out a few of the locator commands. The basic form is:

```
driver.findElement(By.xxx(arg));
diver.find_element_by_xxx(arg)
```

The argument arg is always a string and if no element is found, a NoSuchElementException is thrown.

Some of the ways these are used:

```
findElement(By.className(className));
find_element_by_class_name(class_name))
```

Finds elements based on the value of the "class" attribute.

```
findElement(By.cssSelector(selector));
find_element_by_css_selector(selector)
```

Finds elements using the CSS Selector.

```
FindElement(By.id(id));
find_element_by_id(id)
```

Finds elements using the unique id attribute value

```
FindElement(By.linkText(linkText))
find_element_by_link_text(link_text)
```

Finds <a> elements on an exact match of the link text.

```
FindElement(By.name(name));
find_element_by_name(name)
```

Finds the first element that matches the name attribute value.

```
FindElement(By.partialLinkText(linkText));
find_element_by_partial_link_text(linkText)
```

Finds <a> elements on an partial match of the link text.

```
FindElement(By.tagName(name));
find_element_by_tag_name(name)
```

Finds the first element whose HTML tag is <name>

```
FindElement(By.xpath(xpathExpression));
find_element_by_xpath(xpathExpression)
```

Finds elements using the XPath expression

One of the variations on the findElement(...) method is the findElements(...) method, which instead of returning a reference to a single element, returns a list of elements that match the criteria or an empty list if there is no match. One of little quirks we can exploit is that if we just want to see if an element is on a page, we can use findElements() and see if we get an empty list back.

4.1 Waiting for Things

One of the problems we tried to address in the Selenium IDE was the issue of making the script pause long enough to allow the page to load. We managed that with the “AndWait” suffix on the commands. In the Web Driver, we have to code the wait into the script.

Implicit Timeouts

When we create a driver object, it has two value associated with it. The first is a delay value which defaults to 250 milliseconds, and the second is a timeout, that defaults to 0.

When a page loads, any findElement(By,.xxx()) operations are immediately executed. If the element is not found, an exception is thrown. The default timeout value of 0 means the findElement() command does not wait, but looks immediately for the element.

If we set the timeout value to say 5 seconds, then the findElement() command will try to find the element, and if it does not find it, then it will wait for the amount of time specified by the delay (default of 250 milliseconds) and then try again. If it doesn't find it the second time, it will continue after the delay interval again repeatedly until either it finds the element or until the timeout limit of 5 seconds is reached, and only then will itl throw the exception.

We can set this timeout value with the following command.

```
driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
```

After this executes, the Web Driver has its timeout value set to 5 seconds. If we set the value to a negative number, the timeout is set to “forever”.

Page Load Timeouts

The page load timeout is the amount of time a driver will wail for a page to load during page navigation. When a Web Driver is created it has a default page load timeout value (which I have not been able to ascertain). Using this command

```
driver.manage().timeouts().pageLoadTimeout(5, TimeUnit.SECONDS);
```

The driver will now wait up to five seconds before declaring that a page failed to load. As in the other timeout, a negative value for the time out means essentially there is no timeout and the driver will wait forever for the page to load.

5. The Basic Command Set II

In this section we will look at working with a number of the types of HTML elements that are commonly used. This section does assume a fundamental knowledge of HTML.

5.1 Text Things

First we look at all of the elements we want to use text with. There are three basic operations that we want to do with any element that contains text: (think CRUD functionality)

1. Read any text that is in the element.
2. Write new text into the element.
3. Delete any text that is in the element.

There are two kinds of text widgets – editable ones and non-editable ones. We can only write and clear text for the editable one, ie the ones users could edit. However, the behaviour of these text widgets may not be as intuitive as one would think.

Consider the following part of sample webpage:

This renders in the browser as:

In our test script, we are going to enter a first name “Babloo”, clear out the last name and then examine what is there.

The important point to note here is that calling the getText() on these various elements produces different results. The reasons is that getText() does not necessarily return the visible text we see on the page but rather the text content of the tag. Looking at the HTML on the previous page, the visible text for the last name is not stored in the tag content but as the “value” attribute. We can see the value “Yogi” when we use the getAttribute(“value”) operation.

Notice that in Python, many of these operations are attributes as opposed to methods, to get the value of the text in a widget input_field, we would reference "input_field.text". The other thing we have to be careful of in Python is that sometimes the value "None" will be returned which is a token, not a string.

The other problem seems to be that while we added the text “Babaloo” to the first input field, we can't seem to be able to access it. The reason is that the text is not assigned to the value attribute until the form is submitted.

Generally pure HTML is static so if we want to do things like work with keystrokes as they are entered, we generally use JavaScript to implement that functionality. The other problem is that various browsers implement HTML parsing differently so we may have to test our script in different browsers to make sure it does what we want it to do.

The general rule of thumb here is that we should always validate our Selenium script to ensure that it is doing what we think it is doing. This is where we often have to have a good understanding of what is happening in the underlying HTML and related code to make sure we are getting the right result. Selenium scripts should always be code reviewed before they are made operational.

5.2 Returning Lists of Elements

We had noted before that if there were more than one element matching a selector, then the findElement() or find_element() operation would return the first match. However it may not be the first one that we are interested in bit rather some other one.

By using findElements() or the Python equivalent instead, we get back a list of all the elements that match the selector criteria which we can then work with using standard Java or Python list operations. Consider the following simple HTML

We can get a list of all the para tags by using findElements() as illustrated in the code on the next page.

Which produces the following results:

One little trick that can be used to avoid element not found exceptions. If you use the following:

```
findElements(By.id(“xxx”));
```

If the element does not exist on the page, then a list of length 0 is returned which can be tested using the size() operation instead of throwing an exception like the findElement(By.id(xxx)); would.

5.3 Input Events

Input events occur when the user performs some action that change the state of the page, such as:

1 Clicking on a button or link (a mouse event).
2. Entering some text into a field of some kind (a keystroke event).
3. Manipulating a drop down list or radio button sort of control (a select event).

We have a series of commands that allow us to do these various operations. The first two we will look at now in the cash flow example, and the third we will look at a bit later.

Generally, we emulate a mouse event by finding the "clickable" element and then sending it a "click()" method. We generally enter keystrokes by a similar process of finding the element and then using a sendKeys("string") sort of method.

Both of these will be demonstrated using the Cashflow example which you will then emulate in a lab.

5.4 Complex User Actions

So far we have not dealt with complex user interactions that might involve more than a single mouse click or set of characters entered at the keyboard. To allow us to incorporate these into our scripts, Selenium provides something called an action builder – modeled after the Builder design pattern.

What the action interface does is collect a series of user actions and then executes them on command. For example, consider this logical sequence of steps which is built of by a sequence of steps and then executed:

```
      driver.action.key_down(:shift). // holds the ‘shift’ key down
        click(element).   // clicks on the first element
        click(second_element). // clicks on the second to define a range.
        key_up(:shift).  // complete the selection
        drag_and_drop(element, third_element). // drag the selected items
   into a third
        perform  // now perform the actions described
```

This represents something like selecting multiple rows from different positions in a table. The user actions that we would we use manually are: select multiple rows by selecting the first row, then holding the Shift key, and then selecting another row and releasing the Shirt key. So how would we emulate this in code?

The interface to the Action builder is on the next page. Notice that we list the operations sequentially, call build() to create the complex action, then perform() to execute it.

click(element = nil)   Clicks in the middle of the given element.

click_and_hold(element)   Clicks (without releasing) in the middle of the given element.

context_click(element) Performs a context-click at middle of the given element.

double_click(element = nil) Performs a double-click at middle of the given element.

drag_and_drop(source, target) A convenience method that performs click-and-hold at the location of the source element, moves to the location of the target element, then releases the mouse.

drag_and_drop_by(source, right_by, down_by) A convenience method that performs click-and-hold at the location of the source element, moves by a given offset, then releases the mouse.

key_down(*args) Performs a modifier key press.

key_up(*args) Performs a modifier key release.

move_by(right_by, down_by)   Moves the mouse from its current position (or 0,0) by the given offset.

move_to(element, right_by = nil, down_by = nil) Moves the mouse to the middle of the given element.

perform   Executes the actions added to the builder.

release(element = nil) Releases the depressed left mouse button at the current mouse location.

send_keys(*args) Sends keys to the active element.

5.4 Executing JavaScript code

Given the amount of JavaScript in websites, we often need to execute JavaScript code to perform a test. Selenium provides a mechanism whereby client-side JavaScript code can tested in a Selenium script by using a call to the JavascriptExecutor interface. This interface is invoked by casting the driver to the interface and then invoking one of two methods on the interface. Then the executeScript() method can be used to execute the appropriate script in the context of the currently selected frame or window.

```
JavascriptExecutor js = (JavascriptExecutor) driver;
String title = (String) js.executeScript("return document.title");
```

There is also an executeAsyncScript() method that executes a script asynchronously.

5.5 Dropdowns and Lists

Selenium WebDriver uses a different mechanism for Dropdown and List controls by employing a special Select class instead of the WebElement class.

The interface methods for Select are:

```
void deselectAll()
```

Clear all selected entries.

```
void deselectByIndex(index)
```

Deselect the option at the given index.

```
void deselectByValue(text)
```

Deselect all options that have a value matching the argument.

`void deselectByVisibleText(text)`

Deselect all options that display text matching the argument.

`List<WebElement>    getAllSelectedOptions()`

`WebElement    getFirstSelectedOption()`

`List<WebElement>    getOptions()`

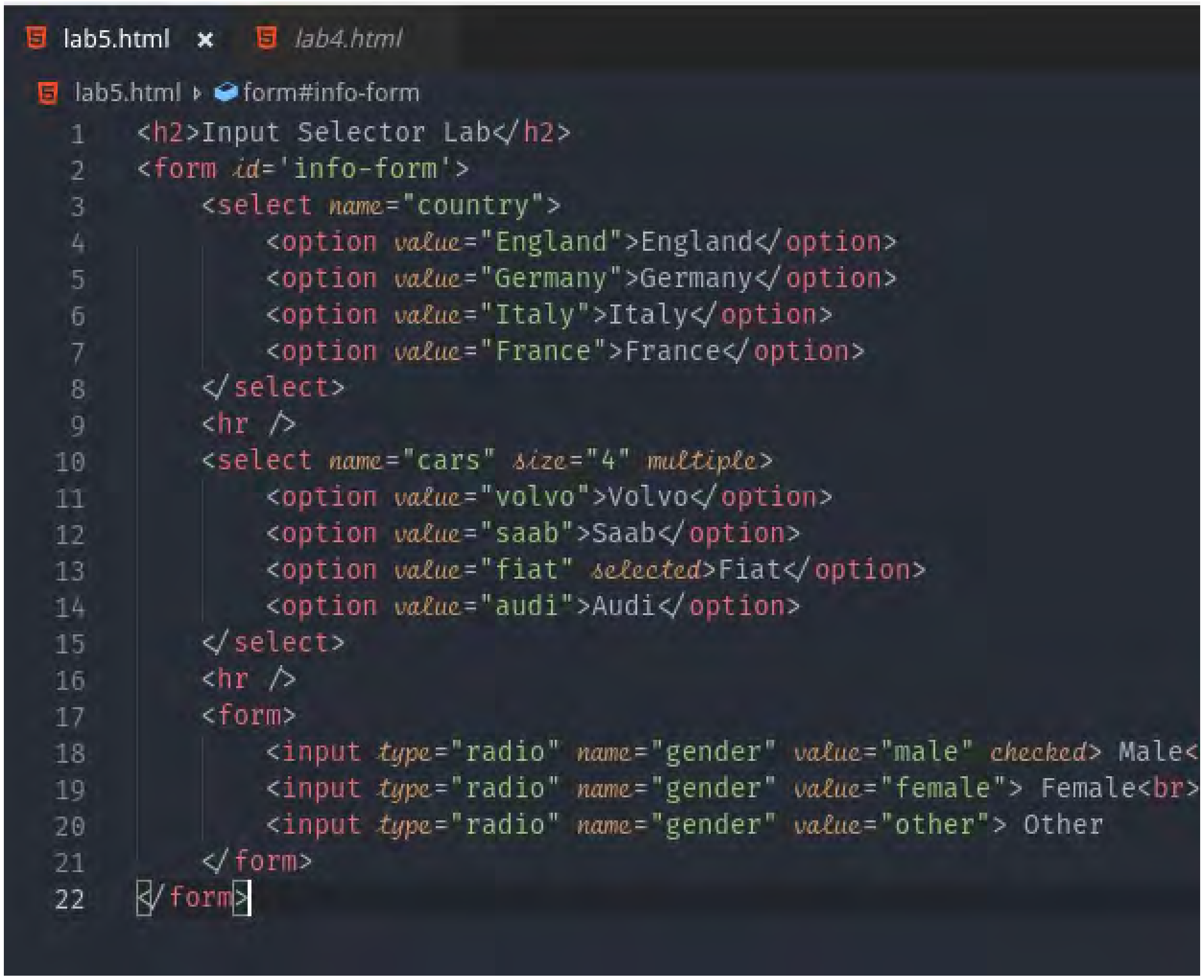
`boolean  isMultiple()`

`void selectByIndex(index) Select the option at the given index.`

`void selectByValue(text) Select all options that have a value matching the argument.`

`void selectByVisibleText(text) Select all options that display text matching the argument.`

The reason for this Select interface is that we extract the drop down list one we find it and we bundle it into this Select object. Then we can manipulate the various choices on the list through the appropriate commands.



```html
<h2>Input Selector Lab</h2>
<form id='info-form'>
    <select name="country">
        <option value="England">England</option>
        <option value="Germany">Germany</option>
        <option value="Italy">Italy</option>
        <option value="France">France</option>
    </select>
    <hr />
    <select name="cars" size="4" multiple>
        <option value="volvo">Volvo</option>
        <option value="saab">Saab</option>
        <option value="fiat" selected>Fiat</option>
        <option value="audi">Audi</option>
    </select>
    <hr />
    <form>
        <input type="radio" name="gender" value="male" checked> Male<
        <input type="radio" name="gender" value="female"> Female<br>
        <input type="radio" name="gender" value="other"> Other
    </form>
</form>
```

Let us suppose the select element has the name “county” with four options: England, Germany, Italy and France. First step is to get the element and make a Select object out of it.

```
      Select country = new Select(driver.findElement(By.name("country")));
```

We want to sure that it is a single valued select operation.

```
      assertFalse(country.isMultiple());
```

Verify that there are four options available

```
      assertEquals(4, country.getOptions().size());
```

We make an option selected by making if visible (think about doing the operation manually).and verify that the selection worked.

```
      country.selectByVisibleText("France");
      assertEquals("France", country.getFirstSelectedOption().getText());
```

Now make another selection by value

```
      country.selectByValue("Italy");
      assertEquals("Italy", country.getFirstSelectedOption().getText());
```

Make another selection by index

```
      country.selectByIndex(0);
      assertEquals("England", country.getFirstSelectedOption().getText());
```

The operations for a multiple choice list are very similar. Lets us suppose our example here has been changed to a multiple selection list.

```
      Select country = new Select(driver.findElement(By.name("country")));
```

We want to sure that it is a single valued select operation.

```
      assertTrue(country.isMultiple());
```

Verify that there are four options available

```
      assertEquals(4, country.getOptions().size());
```

We make an option selected by making if visible (think about doing the operation manually).and verify that the selection worked. But now we have multiple options for selection.

```
      country.selectByVisibleText("Italy");
      country.selectByVisibleText("France");
   country.selectByVisibleText("Germany");
```

5.6 Screenshots

The TakesScreenshot interface is used to capture a screen shot of a web page which helps in understanding exactly what happened at a particular point in the script. This is usually used when we want to see what was on the screen when an error occurs or just to verify that everything that should be on the screen is there and in the right state at a checkpoint in the script.

Here is a simple example.

```
      File scrFile = ((TakesScreenshot)driver).getScreenshotAs(OutputType.-
   FILE);
      FileUtils.copyFile(scrFile, newFile("/tmp/sshot.png"));
```

The screenshot is taken, then copied to a file as a png image. There are a number of different ways that we can generate output from this command For example.

```
      String base64 = ((TakesScreenshot)driver).getScreenshotAs(OutputType-
   .BASE64);
```

Now the screen shot is a string that can be stored, transmitted and manipulated in various ways.

5.7 Windows and Frames

Some web applications can utilize frames or multiple windows. The WebDriver assigns an alphanumeric id, called a handle, to each window as soon as the object is instantiated..

The two primary operations are to get the window handles and to switch between them.

```
   Set<String> handle= driver.getWindowHandles();
```

Once we have the set of Window handles then we can switch windows. For example, we could iterate over all the windows like this:

```
      for (String handle : driver.getWindowHandles()) {
            driver.switchTo().window(handle);
      }
```

Similarly, we can switch between named frames

```
      driver.switchTo().frame("frameName");
```

Or to a pop-up

```
      Alert alert = driver.switchTo().alert();
```

5.8 Windows and Cookies

Often we want to maximize a window. We can do this with this command

```
      driver.manage().window().maximize();
```

When running multiple test cases, we may want to unset cookies to restore the browser back to its original state. Since there is no reason to set cookies in Selenium, the only commands that would be useful are those to delete cookies. We can either delete a named cookie

```
      driver.manage().deleteCookieNamed("__mycookie");
```

Or we can delete all the cookies.

```
      driver.manage().deleteAllCookies()
```

5.9 Working with AJAX

AJAX stands for Asynchronous JavaScript and AJAX which is a scheme that allows the Web page to retrieve small amounts of data from the server without reloading the entire page. In essence, small bits of the page are updated with data fetched from the server without refreshing the page.

Because of this, the general strategies about waits and time outs are not adequate. When we perform any action on Ajax controls the best approach we would like to use is to wait for the required element in a dynamic period and then continue the test execution as soon as the element is found or is visible.

We do this using two commands to wait for an element dynamically, checking ever second or so until the condition is met whereupon we then go to the next command.

This can done achieved with WebDriverWait in combination with ExpectedCondition , the best way to wait for an element dynamically, checking for the condition every second and continuing to the next command in the script as soon as the condition is met.

The first thing we do is create a wait object:

```
WebDriverWait wait = new WebDriverWait(driver, waitTime)
```

Then we can send a variety of things to wait for to the wait object. For example, here are three:

First, we can wait for an element to become visible in the DOM

```
wait.until(ExpectedConditions.visibilityOfElementLocated(locator));
```

Or we can wait until until an object is invisible

```
wait.until(ExpectedConditions.invisibilityOfElementLocated(locator));
```

Or we can wait for it to become clickable.

```
wait.until(ExpectedConditions.elementToBeClickable(locator));
```