Bootcamp de Automation 2016

1. Environment Setup
2. Java - Selenium
3. Elementos dinámicos / Iframes / Waits
4. Page Object pattern
5. testNG - Data provider
6. Jmeter Appendix

1. Installation

1.1. Install Java Platform (JDK)

http://www.oracle.com/technetwork/es/java/javase/downloads/index.html

1.2. Install eclipse (Eclipse IDE for Java EE Developers)

http://www.eclipse.org/downloads/

1.3. Install TestNG plugin for eclipse.

http://testng.org/doc/download.html

See:
http://stackoverflow.com/questions/5230702/how-to-install-testng-plug-in-for-eclipse
(Way 1 works)

1.4. Install Apache Maven.

https://maven.apache.org/download.cgi
https://maven.apache.org/install.html

(Add paths to PATH and JAVA_HOME as suggested by Maven)

1.5. Install Git For Windows.

https://git-scm.com/download/win

(All default settings will do)

1.6. (Optional) No credits for this step. Create a Maven Project in eclipse

Archetype: maven-archetype-quickstart

group id: com.globant.bootcamp
artifact: tae

1.7. Create a github account.

https://github.com/

Fork the bootcamp repository to your own account from the following url:

https://github.com/JoaquinLosada/GlobantBootcampTAE

You will work using your copy of the repository, not ours. We will not accept pull requests to our repository.

1.8. Git Console

Open the git console. You achieve this by doing these:

- Go to eclipse's workspace directory using Windows Explorer,
- right click inside of the folder and select "Git Bash Here"

1.8.1. Configure personal information.

Execute:

```
git config --global user.name "FirstName LastName"
git config --global user.email "Your Email"
git config --global push.default simple
```

1.8.2. Clone your repo into your local machine.

```
git clone https://github.com/yourAccount/GlobantBootcampTAE.git
```

1.9. Eclipse

Import cloned project into eclipse.
(General > Existing Projects into Workspace)

If not already in place, add the following variable to eclipse's classpath:
`M2_REPO = C:\Users\UserName\.m2`

For instructions, see:
http://www.mkyong.com/maven/how-to-configure-m2_repo-variable-in-eclipse-ide/

1.10. Git Console

Open git console at the project directory. You can use the `pwd`, `ch` and `dir` commands. For each exercise do the following:

```
git checkout master
```

You will see the word (master) in cyan at the top right

```
git checkout -b ex-01-fulanito
```

You need to see (ex-01-fulanito) instead of (master).
ex-01-fulanito is the name of the branch. Create a new one for each exercise.

<< Now you solve an exercise using eclipse >>

When you finish

```
git status
git add .
git status
git commit -m "Completed Exercise 01"

git push -u origin ex-01-fulanito
```
(enter email and password of your github account)

For the next exercise you repeat all these steps. Starting from git checkout master.

You will need to study these commands to understand what's going on.

Start here:
http://rogerdudler.github.io/git-guide/

Tutors will verify each exercise separately. All exercises need to be visible in its corresponding branch uploaded to github, they should not be merged to master or to any other branches. This means, Master branch will remain unmodified.

1.11. Install Firefox and extensions.

https://www.mozilla.org/en-US/firefox/new/

1.11.1. Install Firebug

1.11.2. Install Firefinder for Firebug

1.12. Add Selenium and TestNG dependencies to `pom.xml`

See:
http://www.seleniumhq.org/download/maven.jsp

http://testng.org/doc/download.html

1.13. Maven

Each time you add or remove a dependency to *pom.xml*, execute the following commands in the Windows console while located at the project directory:

```
mvn -U dependency:resolve
mvn clean install
mvn eclipse:eclipse
```

1.14. Create an xml suite

1.14.1. Create the folder `./src/main/resources/suite/`

1.14.2. Create the file `./src/main/resources/suite/suite.xml`

1.14.3. Add the following contents

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Bootcamp" verbose="2">
	<test name="Bootcamp">
		<classes>
			<class name="com.globant.bootcamp.tae.App">
				<methods>
					<include name="exercise01" />
				</methods>
			</class>
		</classes>
	</test>
</suite>
```

1.15. Create the test app

```
package com.globant.bootcamp.tae;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.testng.ITestContext;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class App
{
    private WebDriver driver;

    @BeforeMethod(alwaysRun = true)
    public void beforeMethod(ITestContext context){
        try {
            driver = new FirefoxDriver();
        } catch (Exception e) {
        }
    }

    @AfterMethod(alwaysRun = true)
    public void afterMethod(ITestContext context){
        try {
            driver.close();
            driver.quit();
        } catch (Exception e) {
        }
    }


    @Test(
        description = "Exercise"
    )
    public void exercise01() {
       driver.get("http://www.google.com");
    }
}
```

2. a. Locators

Los comandos de Selenium necesitan como parámetro un **locator** del elemento sobre el cual realizar la acción. Es muy importante que dicho **locator** se resuelva de forma única, para que la prueba sea correcta, y la acción no se realice sobre un elemento indeseado.

Por otra parte, también debe tenerse en cuenta que el identificador elegido sea reutilizable en el futuro: por ejemplo debe evitarse la elección de un identificador que cambie con cada nueva versión de la aplicación.

Selenium cuenta con diferentes métodos para encontrar los distintos elementos de la página.
Existen 8 estrategias para encontrar locators en Selenium

- ID
- XPATH
- CSS
- Name
- Link
- Identifier
- DOM
- UI-element

En este curso, nos vamos a centrar en los 3 primeros, que son los más utilizados.

ID:
La manera más eficiente de encontrar un elemento en una página web es mediante el ID. Esta estrategia busca un elemento en una página mediante el atributo id de la forma

<label id=”my_id”>

Buscar un elemento por id permite encontrarlo rapido y facilmente, ya que este debería ser único, solo relacionado al elemento requerido. Es comparable a un “número de libreta” o un “número de cuenta”, los cuales son únicos y de fácil identificación.

Desafortunadamente hay muchos casos donde un elemento no tiene un id, no es único o es generado dinámicamente. Dada esta situación, deberíamos buscar una estrategia alternativa para ubicar el elemento deseado.

Pros:

- Cada id debería ser único, lo que facilitaría su identificación

Contras:

- Funciona solo con id fijos, no con id dinámicos
- No siempre los elementos poseen ids

2. a. i. XPath:

Está diseñado para navegar documentos XML, con el propósito de seleccionar elementos individuales, atributos, o alguna parte del document para un uso específico.
Hay 2 tipos de Xpath:

- Xpath nativo: es una dirección específica y completa.

  html/head/body/table/tr/td

  La ventaja del Xpath nativo es que es muy fácil de encontrar un elemento ya que estamos mencionando la ruta completa del elemento desde la raíz. El problema radica en que si hay algún cambio en cualquier parte del path, este dejará de funcionar como se espera.

- Xpath relativo: se indica una dirección relativa, a partir de cierto elemento hasta el requerido.

  //table/tr/td

  La ventaja aquí, es que si hay algún cambio en el html no especificado en la ruta, este seguirá funcionando. Esto no sucederá si alguna parte del mismo cambia. Es importante determinar en forma correcta qué parte de la dirección es necesaria y cuál no.

Pros:

- Permite determinar locators muy precisos y complejos

Contras:

- Es más lento que otros métodos
- En ciertas ocasiones se encuentran diferencias entre diferentes browsers, por lo que no es recomendable para hacer pruebas cross-browser.

Ejemplos:

- xpath=//table[@id='table1']//tr[4]/td[2]
- xpath=(//table[@class='nice'])//th[text()='headertext']/
- xpath=//a[contains(@href,'href goes here')]
- xpath=//a[contains(@href,'#id1')]/@class
- xpath=//input[@name='name2' and @value='yes']

2. a. i. CSS Selector:

CSS es principalmente usado para proveer reglas de estilo a las páginas web, y también lo podemos utilizar para identificar uno o más elementos en la página que estemos trabajando. La utilización de un locator identificado mediante CSS, será mucho más rápido comparado con el método Xpath. Además, es mucho más seguro, ya que es mejor para realizar pruebas cross-browser.
Es por esto que, en caso de ser posible Css, es la mejor opción a utilizar, para identificar un locator debido a su gran flexibilidad.

Pros:

- Mucho más rápido que Xpath
- Provee un buen balance entre la utilización de estructuras y atributos

Contras:

- Tienden a ser complejos si la estructura es compleja

Algunos ejemplos de CSS:
http://software-testing-tutorials-automation.blogspot.com.ar/2013/06/selenium-css-locators-tutorial-with.html

Repaso general

Para cada uno de estos locators, indicar el método empleado y el objetivo:

- ol[id="list"]//li
- ol[id="list"]/*/li
- document.forms[0]
- p.this_class
- div span#thisid
- div.sidenav li > a [name=this_name]
- /html/body/span[@id="thatspan"]
- //span[@id="thatspan"]

2. b. Funciones principales de selenium

get: accede a una página desde el navegador.

getTitle: retorna el título de la página actual.

quit: cierra todas las ventanas del navegador asociadas al test

findElement: busca y devuelve el primer elemento especificado en la página. Caso contrario, devuelve una excepción.

click: ejecuta un click en el elemento indicado

getText: devuelve el texto de un elemento indicado

sendKeys: introduce texto en un campo de texto

submit: envía la información de un formulario especificado

getValue: obtiene el valor de un valor requerido de un elemento determinado

isEnabled: devuelve un valor booleano dependiendo si el elemento está habilitado o no

isSelected: devuelve un valor booleano dependiendo si el elemento está seleccionado o no

2. c. Asserts

Asserts

Un **assert** es una instrucción que contiene una expresión **booleana** que en un momento dado de la ejecución del programa se debe evaluar a verdadero. Generalmente se conforman de un parámetro booleano que es el que se usa para evaluar el assert y otro opcional que es un parámetro String usado para comunicar algo cuando el assert no es verdadero
Generalmente se evalúa usando assertTrue que evalúa como verdadero cuando el parámetro da True. De igual forma existen métodos para hacer lo contrario (assertFalse) que toma como verdadero el assert si el parámetro es False. A su vez, existen otros que comparan igualdades como assertEquals.
Por convención se suele usar no más de un único Assert por test.

Ejemplo

```
Assert.assertTrue(signInLoginPass(), "The full sign in could not be done");
```

En este caso el test da como válido si el método signInLoginPass retorna un valor True. En caso contrario el test falla y retorna el String que figura en el segundo parámetro del assert.

Uso de Assert.fail para terminar un test

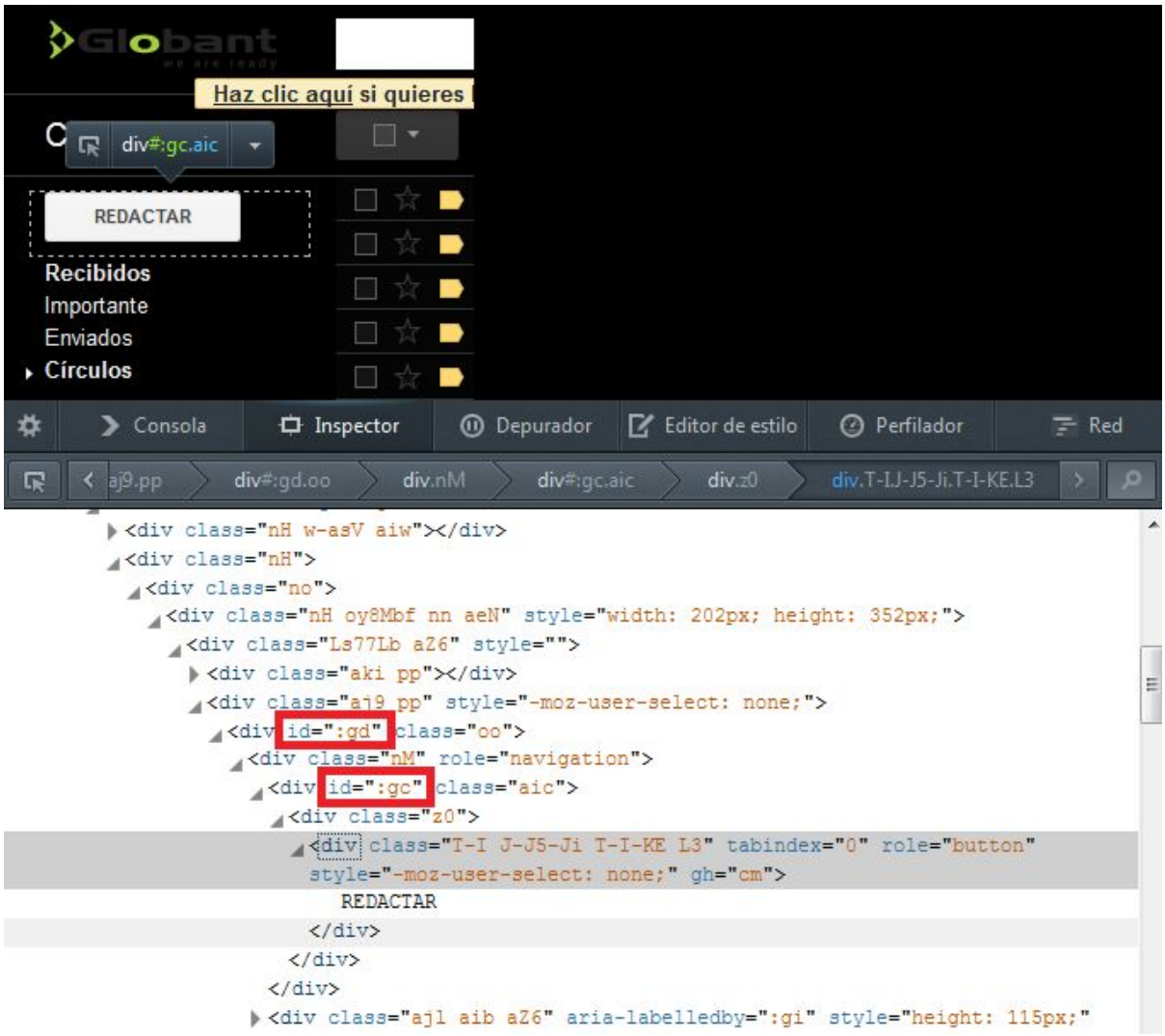
A veces, si una condición interna de un test no se cumple, deseamos que el test finalice sin intentar proseguir con los demás pasos para que no se incurra en esperas innecesarias que igualmente darán un resultado de fail al test en concreto. En el caso anterior supongamos que si un elemento particular no se encuentra presente, el test que llame a este método no debería proseguir su ejecución ya que el resto de pasos fallarían. En este caso se puede incorporar una llamada a Assert.fail al método para que termine la ejecución del test si el elemento no está.

```
public boolean waitForElement(WebDriver driver, By locator, int seconds) {
	if (seconds > 0){
		System.out.println("Waiting for element "+ locator +" for " + seconds + "
seconds.");
	} else {
		System.out.println("Checking if element "+ locator +" is present.");
	}
	try {
		seconds = (seconds == 0 ? 1 : seconds);
		driver.manage().timeouts().implicitlyWait(seconds, TimeUnit.SECONDS);
		driver.findElement(locator);
		return true;
	} catch (NoSuchElementException e) {
		return false;
	}
}


public static void isDisplayedElementByXpath(String elementName) {

	boolean result = waitForElement(driver, By.xpath(elementName), nullWait);
	if (!result){
		Assert.fail("The element " + elementName + " could not be found");
	}
}
```

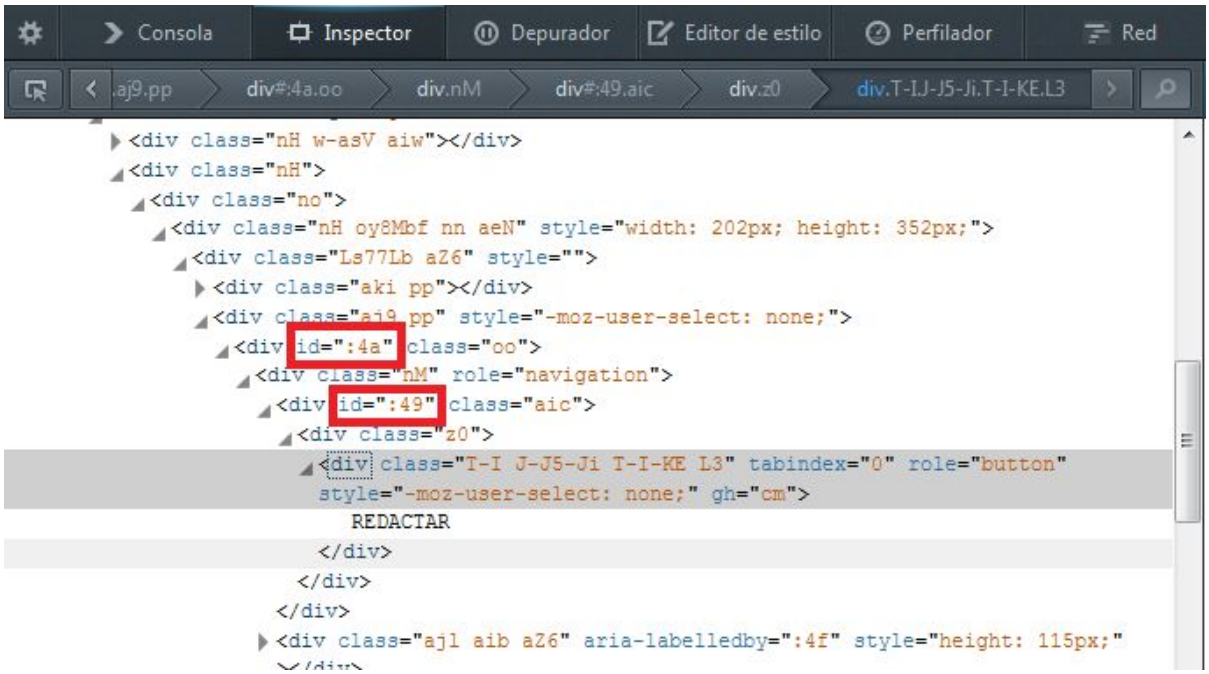
3. Elementos dinámicos

Locators dinámicos

Ciertas páginas utilizan locators que cambian constantemente (generalmente cuando se refresca la página). Esto dificulta crear los locators necesarios para identificar los elementos de esta página y se necesita evitarlos para así poder lograr locators genéricos que funcionen en cada corrida de un test.
Por ejemplo, la página de gmail identifica varios de sus elementos con ids que cambian dinámicamente.



Si esta página se refresca, los ids marcados cambian a unos nuevos.



Si no supiéramos esto, normalmente para identificar el botón de “redactar” se tendería a utilizar alguno de estos ids. Un ejemplo de un locator incorrecto sería:

xpath = “//div[@id = ‘:gc’]//div[@role = ‘button’]”
Por lo dicho anteriormente, este locator podría llegar a funcionar solo algunas veces, ya que la parte del id cambia constantemente cada vez que se carga la página y por ende los tests que llamen a este locator fallarían.
Para evitar este problema, hay que identificar las partes de los elementos que no se modifican y crear nuestro locator basados en esto. Una forma de modificar el locator para hacerlo genérico y que no falle podría ser:
xpath = “//div[@class = ‘z0’]//div[@role = ‘button’]”
Con esta modificación, no importa que el test corra muchas veces refrescando la página, el elemento seguirá siendo identificado sin dar error.

iFrames

Un iframe es un internal frame, una sección dentro de un documento HTML en el cual puedes referenciar otro documento HTML.
Estos elementos dentro de selenium se consideran como páginas aparte a la hora de tratar de interactuar con sus elementos. Por ejemplo, si lanzamos una buscada de elemento y este se encuentra dentro de un iFrame, el método no logrará encontrarlo ya que para hacerlo, primero se debe setear a selenium en el iFrame apropiado.
De la misma forma, cuando terminemos de interactuar con elementos dentro de un iFrame, deberemos salirnos del mismo, si no no podremos usar elementos que estén fuera de este.
Si al tratar de interactuar con un elemento o buscarlo, no logramos hacerlo (y comprobamos que las demás cosas están bien, como por ejemplo el locator usado) tenemos que buscar si el elemento en sí, no se encuentra dentro de un iframe

```
<body>
<div>
      <iframe id="frame1">
          <iframe id="frame2">
            <body>
             <input type="text" id="input2">UserName</input>
           </body >
          </iframe>

          <body>
             <input type="text" id="input1">Password</input>
          </body >
      </iframe>

          <body>
             <button name="btnG">OK<button>
          </body >
</div>
```

En este ejemplo, para poder acceder al input = input2 dentro del iframe = frame2, primero tenemos que setear nuestro driver a frame1 y luego a frame2. Solo así se podrá interactuar con el input = input2
La forma de hacer el switch con selenium es:

```
driver.switchTo().frame("frame1");
driver.switchTo().frame("frame2");
```

Finalmente, al terminar de interaccionar con los iframes, devolvemos el driver a su estado inicial.

```
driver.switchTo().defaultContent();
```

Esperas dinámicas

Generalmente hay páginas donde se realizan cargas o se debe esperar hasta que cierto elemento esté presente o cambie su estado. Normalmente se tiene a llamar a la función waitForElement de selenium que recibe un locator y un tiempo de espera.
Pero una mejor práctica sería esperar dinámicamente mientras la carga de la página siga en efecto (por ejemplo, esperar mientras el elemento de la página de carga sea visible)

```
public WebElement findElement(WebDriver driver, By locator, int seconds) {

        try {
                seconds = (seconds == 0 ? 1 : seconds);
                driver.manage().timeouts().implicitlyWait(seconds, TimeUnit.SECONDS);
                return driver.findElement(locator);
```

```
        } catch (NoSuchElementException e) {
                return null;
        }
}
```

Este método anterior espera implícitamente cierta cantidad de segundos por un elemento a que aparezca.

```
public static boolean waitWhileElementPresentXpath(String locator, int maxSeconds) {

        long initialMillis = System.currentTimeMillis();
        boolean loading = true;
        boolean timeout = false;
        do {
                try{
                        findElement(By.xpath(locator));
                }catch(Exception e){
                        loading = false;
                }
                long currentMillis = System.currentTimeMillis();
                if ((currentMillis - initialMillis) / 1000 >= maxSeconds){
                        loading = false;
                        timeout = true;
                }
        } while (loading);
        if(timeout){
                Reporter.log ("Timeout: "+ maxSeconds + " seconds have passed waiting for
        page");
                return false;
        } else {
                return true;
        }
}
```

Este método por ejemplo recibe un locator de xpath y un tiempo de espera máximo y su función es la de mantenerse esperando mientras el elemento esté presente y no se haya excedido el tiempo de espera máximo dictado. Si el elemento ya no se encuentra más y el tiempo de espera no se terminó, el método retorna True. Caso contrario, retorna False.
De igual forma se pueden hacer variantes para esperar mientras un elemento no esté presente o no cambie su estado.

Esperas de tiempo fijas vs esperas nulas

A veces, luego de esperar cierta carga de una página, queremos comprobar si un elemento está visible o no. Lo primero que uno pensaría es usar la función findElement para encontrar un elemento y si el elemento se encuentra entonces cumplimos nuestra condición. El problema de usar la función findElement para esta lógica es que aun cuando el método no recibe un tiempo de espera como entrada, tiene un tiempo de espera implícito por lo que si nuestro elemento a controlar por algún error no se encontrara en la página, findElement continuará esperándolo un tiempo implícito hasta fallar.
Una mejor solución para este problema es utilizar la función waitForElement pero en el dato de entrada de tiempo de espera se le pasa un tiempo nulo que corresponde a -1. Luego podemos crear una función boolean que pregunte si el elemento está presente pero no genere ninguna espera si este no se encuentra presente.

```
public boolean waitForElement(WebDriver driver, By locator, int seconds) {
	if (seconds > 0){
		System.out.println("Waiting for element "+ locator +" for " + seconds + "
seconds.");
	} else {
		System.out.println("Checking if element "+ locator +" is present.");
	}
	try {
		seconds = (seconds == 0 ? 1 : seconds);
		driver.manage().timeouts().implicitlyWait(seconds, TimeUnit.SECONDS);
		driver.findElement(locator);
		return true;
	} catch (NoSuchElementException e) {
		return false;
	}
}

public static boolean isDisplayedElementByXpath(String elementName) {
	return waitForElement(driver, By.xpath(elementName), nullWait);
}
```

Ejercicios

1) Ingresar a la página www.labrujula24.com, cargar una noticia que posea comentarios, esperar por los mismos y copiar el texto de algún usuario. Hacer un assert si el contenido del mismo posee el texto “Noticia”
2) Ingresar a www.hotmail.com con una cuenta y abrir un mensaje. Realizar un assert si el remitente es support.com
3) Ingresar a www.google.com, realizar una búsqueda, esperar a que carguen todos los resultados, clickear en el tercero, aguardar a que esa página cargue y controlar la existencia de algún elemento
4) Ingresar a www.tn.com.ar, cargar una noticia, apretar en el botón de cargar comentarios y controlar si la cantidad es mayor a 1

4. Page Object pattern

El patrón Page Object Pattern básicamente consiste en:

Modelar áreas de una página con los cuales los tests interactuarán
Métodos públicos representan los servicios que la página provee. Por ejemplo: writeLoginName(username)
No se exponen detalles estructurales de la página.
Generalmente no se utilizan assertions
Algunos métodos devuelven otro PageObject
No es necesario representar la página en su totalidad.

Lecturas recomendadas:
http://martinfowler.com/bliki/PageObject.html
http://www.seleniumhq.org/docs/06_test_design_considerations.jsp (Solo sección Page Object Design Pattern)

Ejemplo:

```
public class LoginPage extends BaseWebDriverTestCase {
    // Assume WebDriver is inherited from here
    public LoginPage() {
        // Check that we're on the right page.
        if (!"Login".equals(driver.getTitle())) {
            // Alternatively, we could navigate to the login page,
            // perhaps logging out first
            throw new IllegalStateException("This is not the login page");
        }
    }

    // Conceptually, the login page offers the user the service of
    // being able to "log into" the application using a user name and password.
    public HomePage loginAs(String username, String password) {
        // This is the only place in the test code that "knows" how to
        // enter these details
        driver.findElement(By.id("username")).sendKeys(username);
        driver.findElement(By.id("passwd")).sendKeys(password);
        driver.findElement(By.id("login")).submit();

        // Return a new page object representing the destination.
        return new HomePage();
    }
}
```

Todos los PageObjects se almacenan en la carpeta src/java y los tests en la carpeta test

PageFactory & selenium

La clase PageFactory es utilizada para inicializar los elementos del PageObject.

@FindBy
La annotation FindBy permite mapear un webElement con lo que indiquemos dentro de la annotation. Al utilizar @FindBy selenium automáticamente mapeara el webElement sin necesidad de ejecutar FindElement

@FindBy soporta varias estrategias para obtener un webElement:
id, name, className, css, tagName, linkText, partialLinkText, xpath
En general los más usados son id, css y xpath

Ejemplo básico:

```
public class Tests {

	WebDriver driver = null;

	@BeforeMethod
	public void before() {
		driver = new FirefoxDriver();
	}

	@AfterMethod
	public void after() {
		driver.quit();
	}

	@Test
	public void test1() {
		HomePage homePage = PageFactory.initElements(driver, HomePage.class);
		homePage.go();
		homePage.search(“Testing”);
		assertTrue(homePage.getTitle().contains("Automation Bootcamp"));
	}
}


public class HomePage {

	final String baseUrl = "www.foobar.com";
	WebDriver driver;

	@FindBy(id = "s")
	private WebElement searchBox;

	public HomePage(WebDriver driver) {
		this.driver = driver;
	}

	public void go() {
		driver.get(baseUrl);
	}

	public String getTitle() {
		return driver.getTitle();
	}

	public boolean search(String query) {
		searchBox.sendKeys(query);
```

```
                searchBox.sendKeys(Keys.RETURN);
        }
}
```

Ejercicios:

Ejercicio 1:

1. Ingresar a www.amazon.com
2. Ingresar en el campo de búsqueda “kindle”
3. Clickear en la lupa
4. Clickear en el 5to resultado
5. Controlar que el titulo del articulo contenga la palabra “kindle”
6. Volver a la página anterior
7. Seleccionar “Movies & TV en el menú izquierdo del campo de búsqueda
8. Ingresar en el campo de búsqueda “superman”
9. Clickear en la lupa
10. Controlar que al menos se muestran 4 resultados
11. Clickear en el logo de Amazon en la parte superior para volver a la home page
12. Controlar que el titulo sea igual al string “**Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more**”

Ejercicio 2:

Realizar el ejercicio 4 del topic 3 utilizando PageObject pattern

5. TestNG

TestNG es un framework para pruebas y testing que trabaja con Java y está basado en JUnit (para Java) y NUnit (para .NET), pero introduciendo nuevas funcionalidades que los hacen más poderosos y fáciles de usar, tales como:

- Anotaciones JDK 5 (Annotations) (JDK 1.4 también es soportado con JavaDoc annotations).
- Configuración flexible de pruebas.
- Soporte para pruebas para data-driven testing (with @DataProvider).
- Soporte de pasaje de parámetros.
- Permite distribución de las pruebas en máquinas esclavas.
- Modelo de ejecución poderoso (TestSuite nunca más).
- Soportado por herramientas y plugins importantes y variados como: (Eclipse, IDEA, Maven, etc.).
- Permite embeber BeanShell para una flexibilidad más amplia.
- Funciones JDK por defecto de runtime y logging. (sin dependencias)
- Métodos dependientes para pruebas sobre servidores de aplicación.

Grupos y prioridades

Con TestNG se les pueden asignar grupos y prioridades a los tests que luego se pueden utilizar en las suites para hacer conjuntos específicos de tests y setear órdenes de importancia de los tests.
El grupo se agrega entre paréntesis seguido de la anotación @Test. Este nombre puede ser cualquiera y se le puede asignar más de uno a un test o repetirlos.

```
public class Test1 {

        @Test(groups = {"functest", "checkintest"} )
        public void testMethod1() {
        }

        @Test(groups = {"functest", "checkintest"} )
        public void testMethod2() {
        }

        @Test(groups = { "functest" }
        public void testMethod3() {
        }

}
```

Una forma de utilizar estos grupos es creando un archivo xml que se usará para invocar y agrupar los tests que queramos ejecutar

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Test" verbose="1">

	<test name="Test1">
		<groups>
			<run>
				<include name="functest"/>
			</run>
		</groups>
		<classes>
			<class name="example1.Test1"/>
		</classes>
	</test>
</suite>
```

También se pueden excluir grupos usando el tag “exclude”. Esto es útil cuando se quiere crear un grupo de tests específicos pero en ciertas suites no se lo desea incluir

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Test" verbose="1">

	<test name="Test1">
		<groups>
			<run>
				<include name="functest"/>
				<exclude name="broken"/>
			</run>
		</groups>
		<classes>
			<class name="example1.Test1"/>
		</classes>
	</test>
</suite>
```

Los grupos también se pueden anidar a nivel de clase y luego subanidar grupos a los tests.

```
@Test(groups = { "checkin-test" })
public class All {

	@Test(groups = { "func-test" })
	public void method1() {
	}

	public void method2() {
	}
}
```

Si deseamos que ciertos tests se ejecuten primero antes que otros porque posean un orden de prioridad más alto, podemos agregar el parámetro “priority” entre paréntesis de la anotación @Test. Si el número asignado a priority es menor, se le da un importancia mayor

```
	@Test(priority = 1)
	public void method1() {
	}

	@Test(priority = 2)
	public void method2() {
```

```
        }

        @Test(priority = 3)
        public void method3() {
        }
```

En el ejemplo anterior, testMethod1 se ejecutará primero ya que su valor de prioridad es el menor y luego le seguirán testMethod2 y testMethod3

Dataproviders y parametrización

A los tests que creamos se les pueden pasar valores de entrada para poder hacerlos más genéricos y dinámicos. Estos valores pueden ser fijos o pueden ser externalizados en un archivo que pueda ser modificado por otras personas. Este manejo de datos permite que los tests puedan ser más versátiles y mantenibles. Por ejemplo si se crea un test que controle el logueo válido de un usuario, se podría hacer un test estático que solo pruebe con datos hardcodeados al mismo test pero también se podrían pasar estos datos como datos de entrada al test, permitiendo que una lista de usuarios pueda ser ingresada.
Ejemplo estático

```
@Test
public void testValidDate() {
        selenium.open("/tested-webapp/index.jsp");
        selenium.type("username", "user");
        selenium.type("password", "secret");
        selenium.click("//input[@value='Login']");
        selenium.waitForPageToLoad("30000");
        verifyTrue(selenium.isTextPresent("Login succedd"));
}
```

Este test utiliza los valores “user” y “secret” de forma harcodeada. Como único test este es válido, pero qué pasaría si muchos de nuestros tests necesitaran loguearse o lo usuarios a usar cambiaran o se necesitara testear más de uno? Para ello podemos parametrizar el test y darle como dato de entrada los valores de username y password. Para esto utilizamos los dataProviders que aportan los inputs a un test.

```
@DataProvider(name ="LoginData")
public Object[][] getLoginData() {
        return new Object[][] {{ “Usuario1”, “Password1”},{ “Usuario2”, “Password2”}};

@Test(dataProvider = "LoginData")
public void testValidDate(String userData, String passData) {
        selenium.open("/tested-webapp/index.jsp");
        selenium.type("username", userData);
        selenium.type("password", passData);
        selenium.click("//input[@value='Login']");
        selenium.waitForPageToLoad("30000");
        verifyTrue(selenium.isTextPresent("Login succeed"));
}
```

Ahora nuestro test se volvió más genérico y dinámico. El mismo test se ejecuta 2 veces (una vez para los datos de Usuario1 y otro para Usuario2). La cantidad de datos parametrizados es ilimitada.

A su vez los datos que el dataprovider obtiene pueden provenir de cualquier medio como un archivo externo XML o un excel.

Se pueden crear varios dataproviders dentro de una clase y usarlos dependiendo del test que los necesite.
Cuando un test parametrizado se corre con TestNG, los resultados se muestran discriminando cada ejecución múltiple de un test y mostrando sus valores de entrada.