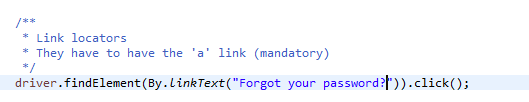
LOCATORS

1. Link text

Se puede coger el valor del link text para poder utilizarlo

Siempre tiene que tener el tag ‘a’.

Se le ha de pasar el texto, no lo del href **OJO**



**Para cerrar el browser, se apreta driver.close() y se cierra únicamente la ventana que se está manejando**

**Para cerrar todas las ventanas, se apreta driver.quit()**

**Se usa uno u otro indistintamente**

OJO

Las nuevas versiones de Selenium pueden prescindir del setProperty()

Se puede proveer en el nuevo SeleniumManager, quien indica en un JAR lo que meter y lo que no.

Descarga la última versión de ChromeDriver.

SELECTORES XPATH

Es parecido al css selector.

Se construye con una sintaxis tal que

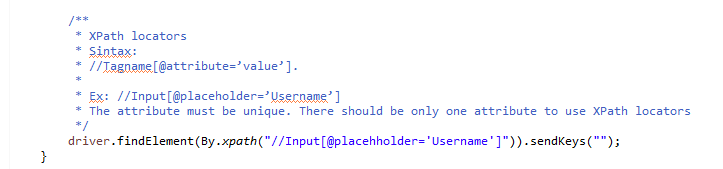
//Tagname[@attribute=’value’]

Ejemplo

//Input[@placeholder=’Username’]

Para encontrar un elemento en la consola de DEV Tools, usamos

**$x()**



**SYNTAX CSS SELECTOR VS XPATH**

\* Para borrar un texto, está el método clear();

Por otro lado, si no tenemos el valor, se puede indexar el elemento, como

//input[@type=’text’][2]

La sintaxis

//Tagname[@attribute=’value’][index]

**Tener en cuenta que puede haber elementos que ignora Xpath y otros que incluya CSS Selector y viceversa.**

Para acceder a los hijos en **CSS, se usa ‘:’**.

La sintaxis para acceder a los hijos:

Tagname[attribute=’value’]:nth-child(index)

**Xpath childs**

En este caso, para poder acceder a los hijos hay que poner **un** slash, tal que:

//form/h2 (por ejemplo)

La **sintaxis** será:

//parentTagname/childTagname[index]

**CSS Childs**

Aquí no se van a poner slashes como en Xpath

Se ponen espacios vacíos, tal que “ “

Por ejemplo:

**parentTagname childTagname**

**CSS Selectors based on RegEx**

Si queremos pausar la ejecución Java durante un tiempo determinado, se ha de usar el método:

*Thread.sleep(number\_of\_milliseconds*)

A veces cuando se utiliza Selenium, su ejecución es más rápida que los elementos que aparecen en pantalla. **A esto se le llama InterceptionError** y es muy importante de cara a una entrevista.

En un futuro se utlizarán **Explicit Waiters**, que es lo que de verdad se debe utilizar en vez de utilizar un Thread.sleep()

Recuerda:

* Implicit waiters se utilizan para eventos que esperamos que aparezcan en la pantalla.
* Threads.sleep() se utiliza para que algo esté en una versión estable para ser utilizado. No es aconsejable utilizarse ya que no hay ninguna condición de espera.

**Ahora con el RegEX**

Se utiliza para casos en los que el tipo cambia, etc.

Para ello se pone una estrella delante del = tal que

Tagname[attribute\*=’value’]

Por ejemplo:

Input[type\*=’pass’] – Sería de password

**Finalmente se puede utilizar una RegEX para hacer lo mismo con Xpath, como:**

//Tagname[contains(@attribute, ‘splitted-value’)]

Esto se utiliza porque el contenido puede cambiar

**SELENIUM TEXT PARTIAL**

Se puede coger también el tag name como locator. No se suele utilizar mucho ya que suele haber muchísimos.

By.tagName()

Para coger en función del texto, hay que llamarlo como (en Xpath)

**Tagname[text()=’value’]**

**Por ejemplo**

**//button[text()=’Log Out’]**

OJO

Si no importa el Tagname, se puede usar un asterisco

//\*[text()=’Log Out’]

**PARA CERRAR EL NAVEGADOR**

**driver.close();**

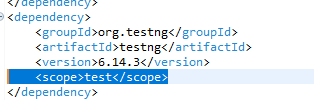
**PARA ASERCIONES**

Utilizamos el framework TestNG.

Recordar meterlo en las dependencias de Maven.

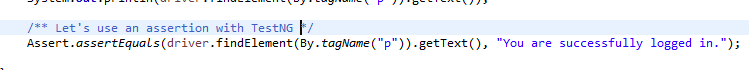
**Importante**

Quitar el scope:



Usamos el método assertEquals de la clase **Assert** de TestNG

Este espera comparar el valor esperado con el real.

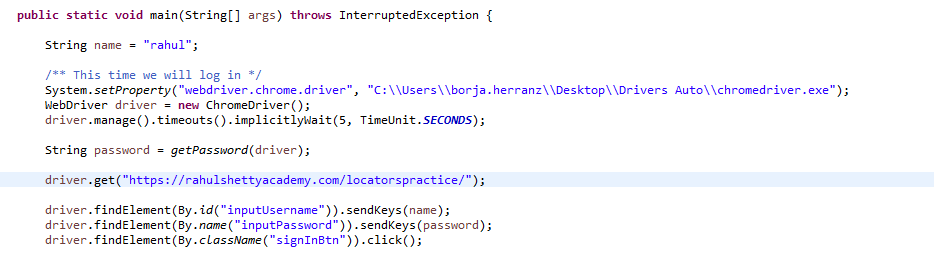


**PARSE STRINGS**

Vamos a parsear los string para poder utilizarlos.

Vamos a usar métodos como split, etc, al igual que getText() de selenium





**RUNNING THE TESTS WITH CROSSBROWSER**

Vamos a realizar crosswbrowser testing

Es para verlo con diferentes browser

SIBLING – PARENTS TRAVERSE

Recordar que para llegar al elemento que queremos utilizar, se pueden usar:

* Rutas absolutas: se llegan desde el parent /html/...
* Rutas relativas: se llegan desde uno de los hijos: //div...

Por ejemplo, una ruta relativa sería:

//header/div/button[1]

Los siblings o primos estarían al mismo nivel. En este caso, todos los botones que aparecen

Para llegar al siguiente sibling, se puede usar en xpath *following-sibling*.

Con eso llegaremos al elemento que queremos

Por ejemplo:

//header/div/button[1]/following-sibling::button[1]

Para ir al padre, hay que escribir */parent::tagname/parent::tagname*

Por ejemplo, para ir de padres a hijo:

//header/div/button[1]/following-sibling::button[1]/parent::div/parent::header/div

**ESTO NO SE PUEDE HACER EN CSS, ES MEJOR HACERLO EN XPATH**

**AUTOMATE BROWSER NAVIGATIONS**

Vamos a usar técnicas para ampliar el tamaño de la página

Para ello: *driver.manage().window().maximize();*

La base es:

*driver.manage().window().method();*

Por otro lado, se pueden utilizar técnicas de navegación como

driver.navigate().to(String url);

La principal diferencia entre get() y navigate().to():

* Get espera a que la página haya cargado
* Navigate no espera. Si queremos probar otra de las páginas, nos vale para no tener que esperar

Si queremos volver a la página anterior

driver.navigate().back()

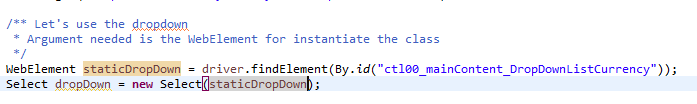
<https://rahulshettyacademy.com/dropdownsPractise/>

SELENIUM ADVANCED TECHNIQUES

**1. Handling Static Dropdowns with Select webdriver API**

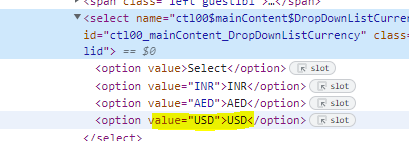
Hay una clase especial para recoger los dropdowns en Selenium.

En este caso, nos centraremos en los elementos estáticos: aquellos desplegables cuyos elementos no varían en función del número de elmentos que añadas.

La clase que debemos utilizar es **Select(WebElement)**. Vamos a instanciar la clase. 

Hay un mogollón de métodos que soportan esta clase, como:

* Seleccionar por index: selectByIndex(index)
* Coger el primer item seleccionado: getFirstSelectedOption()
* Seleccionar por texto visible: selectByVisibleText()
* Seleccionar por value (atributo): selectByValue()

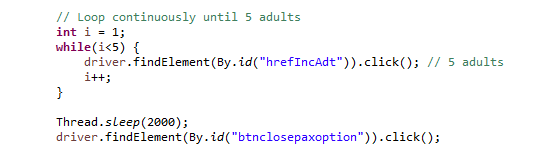


Para otros seleccionables, si no tiene el TagName, se verá en siguientes lecciones

**Looping**

Se puede hacer looping de los clickables.

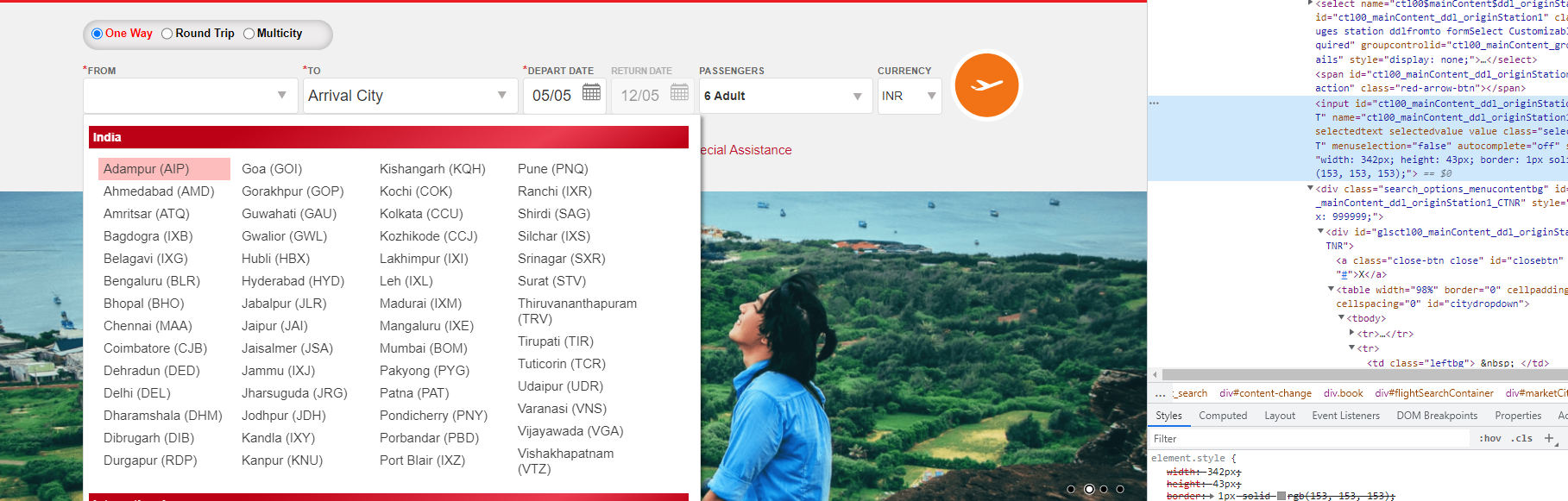
While y for loop



**2. Handling Dynamic Dropdowns with WebDriver API**

Son seleccionables que se cargan cuando se seleccionan una opción anterior.

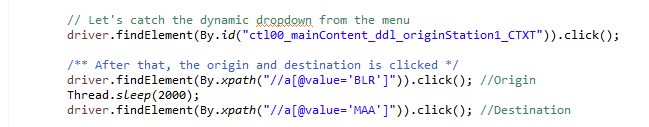
Por ejemplo:



En este caso, se cambia el destino en función del origen.

Para ello, se deben coger las opciones en función de lo que seleccionamos.

Si lo hacemos asi



Debería fallar.

Hay dos elementos MAA en la página, uno en el origen y otro en el destino.

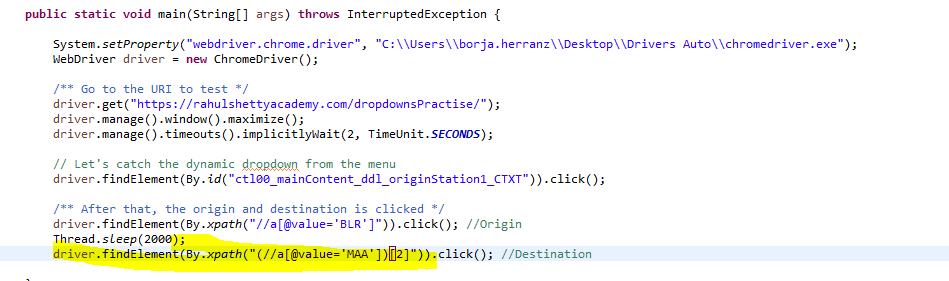
En ese caso, debemos saber cual utilizar.

Para coger un elemento, debemos poner entre paréntesis todo el Xpath y poner el elemento a seleccionar.

Por ejemplo:

driver.findElement(By.xpath((//TagName[@attribute=’value’])[index])

driver.findElement(By.xpath((//a[@value=’MAA’])[2])



**3. Parent-child relationship locators**

Puede ser que a un cliente no le gusten los indexes (por poder manipularse)

En ese caso, se ha de buscar una alternativa para acceder a los diferentes elementos del web.

Se pueden usar parent-child para acotar el scope

Para ello, se ha de poner un **espacio** entre padre e hijo.

Por ejemplo

//ParentName[@attribute=’value’] //ChildElement[@attribute=’value’]

//div[@id=’ccc’] //a[@value=’BLR’]

Va a mirar solo en el padre, en nada más.

Ese es el scope.

Esto se le llama **parent-child** travel mecanism para Xpath.

**4. Auto Suggestive**

Para aquellos campos que se autocompletan cuando se les pasa un valor, se puede utilizar un bucle for loop para ver si contiene lo que queremos.

**5. Handling checkboxes and getting the size**

Vamos a aprender a recoger los checkboxes, a marcarlos o ver su tamaño con Selenium.

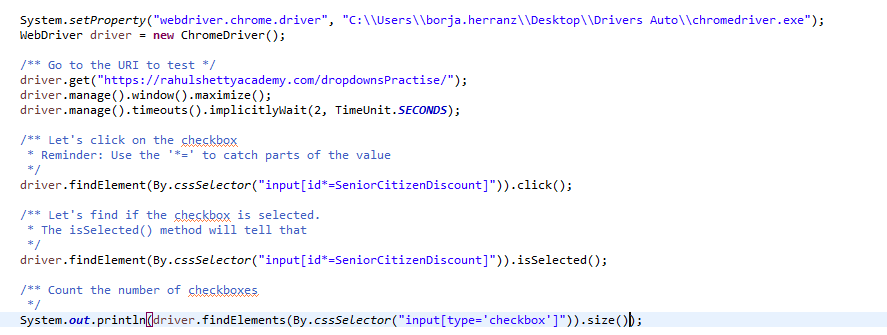
Para indicar parte del texto, se puede poner, en CSS Selector:

Tagname[attribute\*=’value\_partly’]

**Para saber si el checkbox está seleccionado, hay un método en Selenium que te lo indica**.

El atributo es **isSelected()**.

Esto devuelve un booleano True o False



**6. Importancia de los assertions**

Para usar aserciones, se deben usar frameworks como:

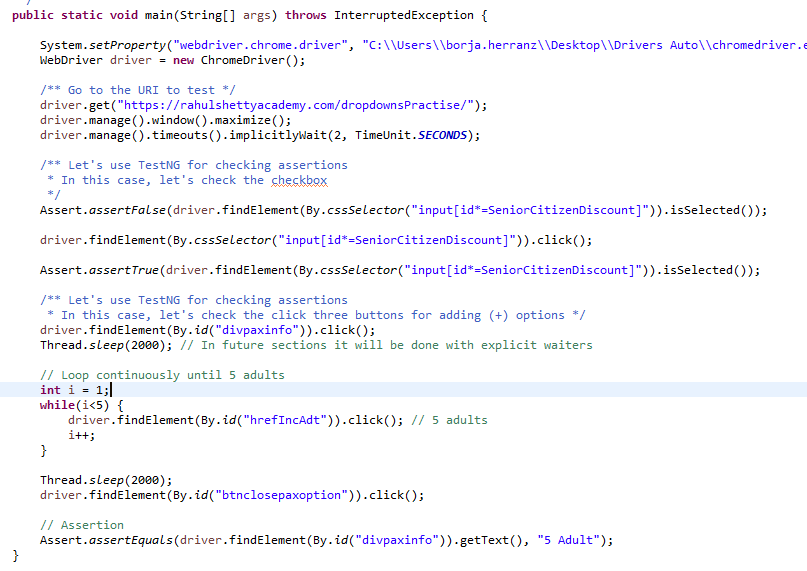
* TestNG
* Junit

Se estudiarán más adelante en detalle

TestNG es uno de los frameworks utilizados por Selenium para utilizar Robot Utilities.

Nos van a dar una primera idea de si lo que se valida cumple con lo que se espera

De esta forma, nos da una idea aproximada si el código funciona o no.



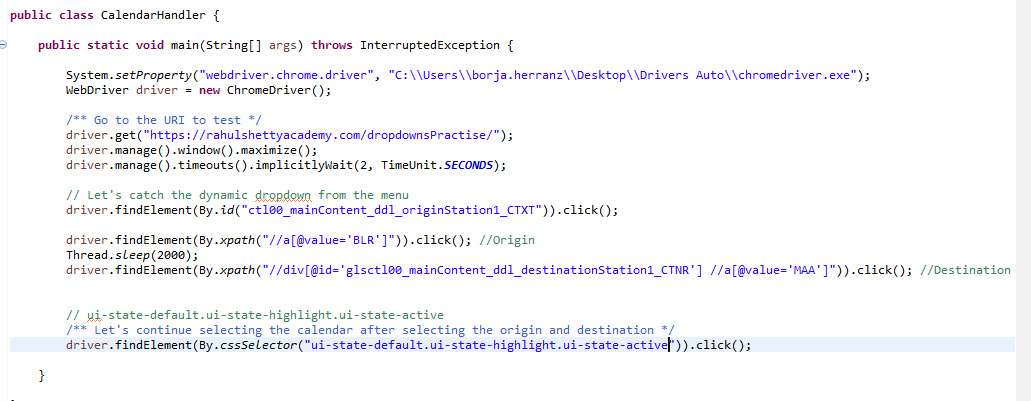
Para ello usamos la clase Assert y los métodos como:

* assertTrue()
* assertFalse()
* assertEquals()
* assertNull()
* ...

**7. Handling Calendar**

Tener en cuenta que para utilizar CSS Selector y la parte de los hijos, necesitamos no tener **espacios vacíos**, y tener en contraparte puntos.

Por ejemplo



Para este caso, hemos cogido el elemento activo, que es el mismo para cada uno de los elementos del calendario.

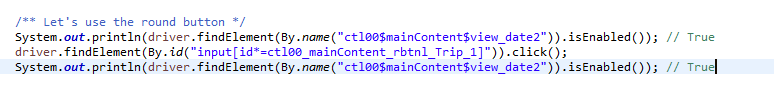
**8. Validación de elementos UI**

Hay elementos, como los round connectors, que no son fáciles de pillar en Selenium

Hay inconsistencias entre lo que espera Selenium con lo que debe ser.

Para ello, hay métodos como de isEnabled() que cambian entre lo que espera el HTML con lo que hace Selenium.

Por ello, no hace nada. Eso es porque en las últimas versiones de UI, no está deshabilitado per se, lo parece, pero no lo es. Puedes pincharlo y hacer cosas.



Sólo cambia la opacidad.



¿Cómo solventarlo?

Vamos a usar el método **getAttribute()**, que nos permite coger el atributo.

Si queremos coger el atributo style



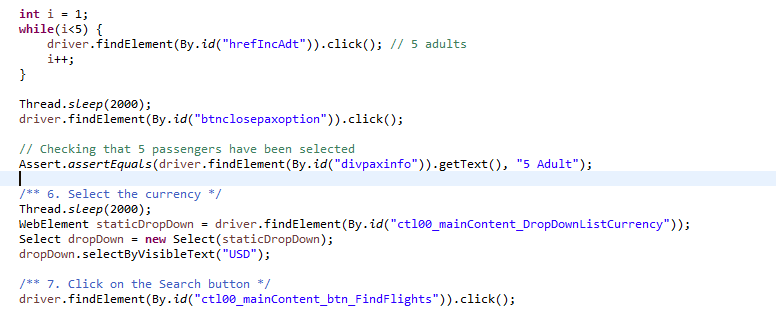
**9. E2E testing automation**

En este punto, vamos a hace un ejercicicio e2e para ver qué todas las piezas funcionan.

Vamos a hacer el siguiente circuito:

1. Elegir un origen
2. Elegir un destino
3. Seleccionar la fecha actual
4. La vuelta por defecto
5. Elegir número de pasajaros
6. Dar al botón de buscar





**10. Handling alerts in Java**

Vamos a gestionar las alertas en web.

No son HTML, no están en la web, por lo que no hay nada para coger

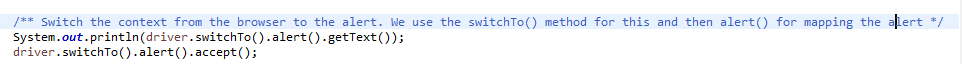
Es JavaScript

Hay un mecanismo para poder coger estas alertas.

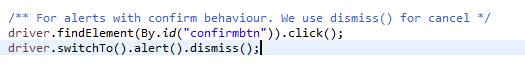
El método que se va a usar cambia el contexto de la alerta. Es switchTo(). Se pasa del browser a la alerta, para poder testearla.

Aquí:

* alert(): acepta alertas
* cancel(); cancela alertas
* getText() para coger el texto de la alerta



Para las alertas con el botón de cancelar... (alertas de confirmación). Con dismiss() quitamos alertas



Siguiente leccion

**1. Coding formatting and debugging**

Es necesario formatear el correo y debugearlo para que funcione correctamente.

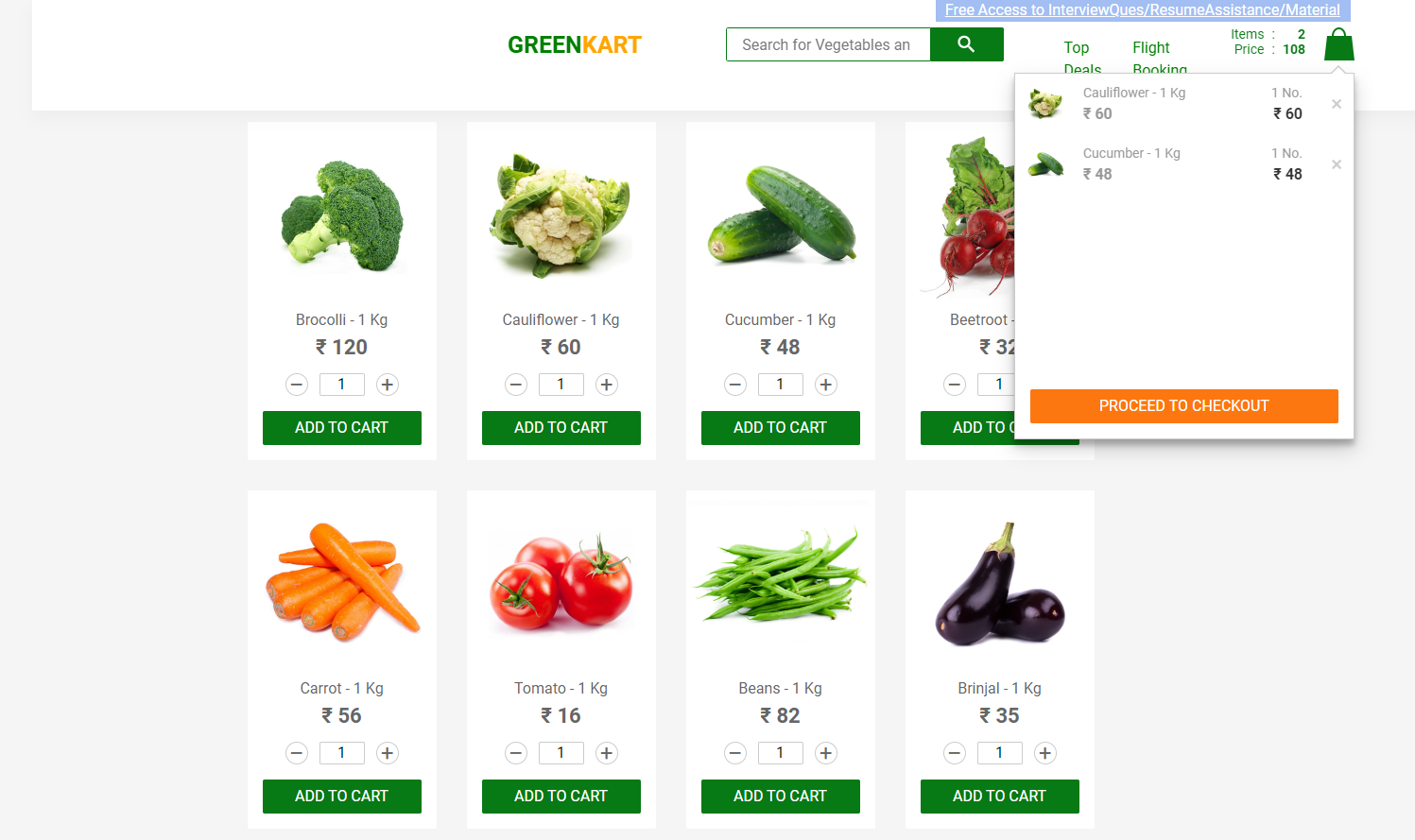
Para eclipse: Ctrl + Shift + F

Las convenciones para escribir las clases, métodos,...

* Clases: empieza por mayúscula
* Métodos en camellcase
* Variables: no empiezan con números y en misnúscula

**2.** Añadir items a una lista – Ecommerce

**P**regunta de Amazon muy utilizada



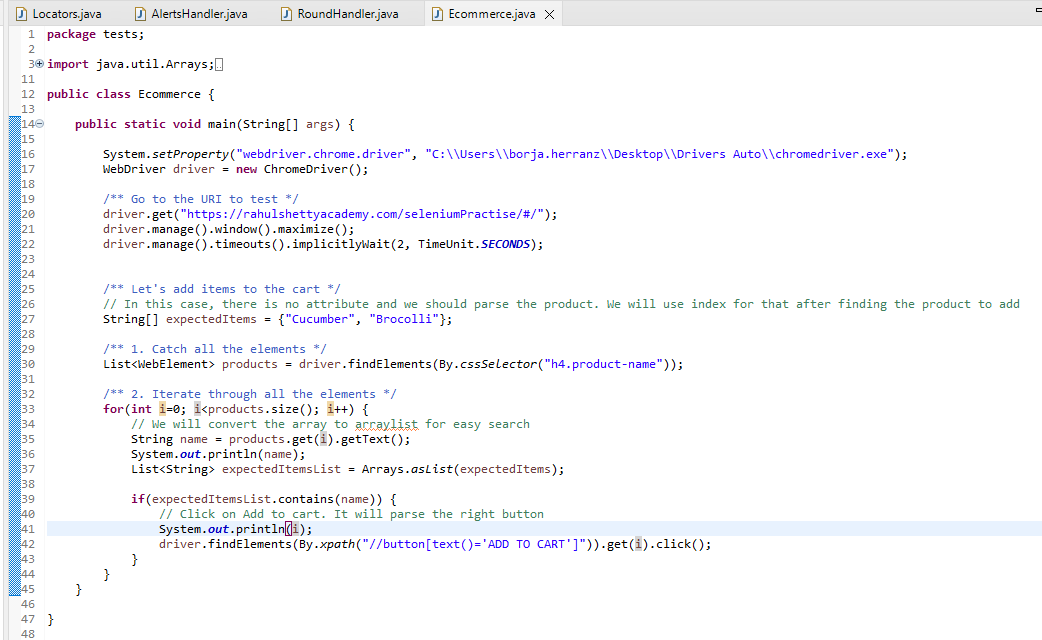
Vamos a utilizar Automation para esto

Para esta práctica, debemos parsear el elemento a seleccionar antes de dar al botón ‘Add to Cart’.

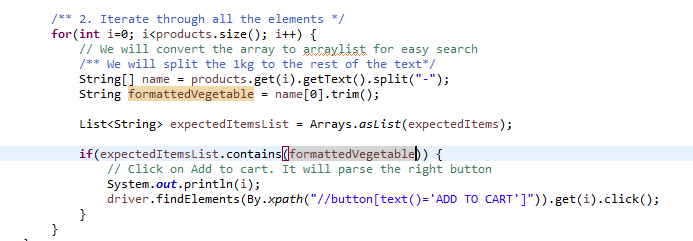
No hay ids ni nada que nos ayude a identificarlos.

Vamos a iterar a lo largo de los productos existentes, y cuando lo tengamos, seleccionar el index y darle al botón añadir.

Vamos a añadir un array de items: por ejemplo pepino y coliflor. Vamos a utilizar ArrayList ya que contienen métodos de ‘contains’.



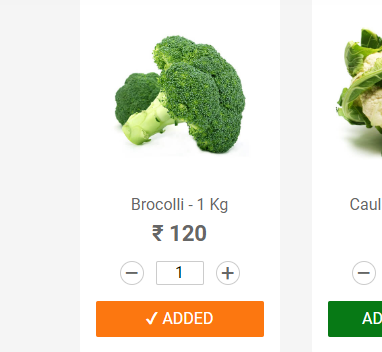
Por ahora, debemos separar el texto de 1kg para poder trabajar



Sin embargo, hay un problema, que es la condición de carrera

Hay un tiempo desde el que se clicka el botón hasta que se añade a la cesta. En ese tiempo, se puede comprobar que no añade el item adecuado (en vez de meter nabos, mete tomates).

De esta forma, el texto no se queda estático, sino que llegado un punto vuelve a su valor original, rompiendo el esquema



Por ello, vamos a utilizar otro locator, ya que este no nos vale de forma dinámica

**OJO--- ES UNA BUENA PRÁCTICA NO UTILIZAR NUNCA TEXT LOCATORS**

Vamos a quitar el de ADD TO CART



SYNCHRONIZATION IN SELENIUM

Hay varios tipos de espera en Selenium.

Estos son:

* Implicit Waiters
* Explicit Waiters
* Thread.sleep
* Fluent Wait

**Sincronizacion**

Por ejemplo, imaginar que estamos en un formulario y después de meter los campos, damos enter. A partir de unos segundos, se mostraría el PayLoad.

En esos segundos que está cargando la página, nuestros test automáticos de Selenium son más rápidos. No tiene espera. Buscará antes de que pasen los 3 segundos.

Hay que indicar al WebDriver que tiene que esperar un tiempo o que espere a un evento antes de revisar la página, para que esté cargada.

**Implicit Wait**

Se define un tiempo específico de espera para el WebDriver antes de que empiece a seguir con la automatización. **Es una espera de X segundos, antes de que lance un exception.**

* La primera ventaja es que se define de forma global, ya que todos los steps tienen que esperar (es secuencial).
* No espera todo el tiempo que se le haya indicado. Es decir, si se define un tiempo de espera de 5 segundos y la página carga en 3, continuará los tests desde ahí. De esta forma no limita al resto de los test cases.
* Va a escuchar al DOM, una vez cargado, continuará

Las principales desventajas:

* El principal problema es que no sabemos de antemano el tiempo que tardará en cargar un elemento. Puede cargar antes de X segundos debido a un glitch, puede no aparecer si el intervalo que se ha dado es menor,...

**Explicit Wait**

En este caso, las esperas no son globales, sino que están localizadas en el método/elemento que testean. Si se le indica un tiempo, es para un locator específico.

**Diferencias, ¿cual utilizar?**

Nos decantaremos por los casos implícitos cuando apliquen a todos los casos de prueba.

Por el contrario, los casos explícitos nos decantaremos por aquellos que sólo apliquen a un elemento específico.

**Estrategia**

Es necesario utilizar esperas, ya sea implícitas o explícitas. No hacer casos a aquellos que dicen que nos quitamos los implicit.

La combinación es buena.

Cuando hay miles de test cases, es mejor usar implicit waits ya que pueden aplicar a varios y permite quitar gran parte del código repetido.

**Thread.sleep()**

No es parte de Selenium, sino de Java.

En este caso, se le indica cuanto debe esperar, **pero no escucha el DOM, por lo que solamente esperará el tiempo que se le indique, y nada más.**

Es diferente al implícito.

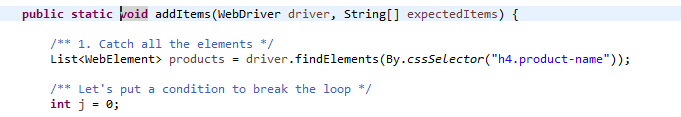
Es mejor no usarlo en real time.

EJERCICIO CESTA ELECTRÓNICA – CONTINUACIÓN

Vamos a continuar el ejercicio de la cesta electrónica usando para ello las esperas que estudiamos en el apartado anterior.

Lo primero es separar la funcionalidad en métodos

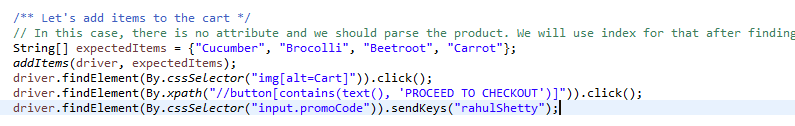
Hay que declarar el método como estático, porque no se está creando un objeto a partir de ese método (no devuelve nada)



Se puede prescindir de la palabra ‘static’ al llamar a la clase base y de ahí pasando los parámetros.

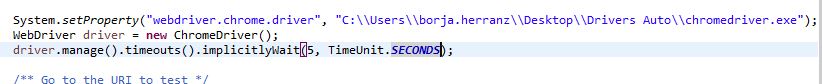
Ahora vamos a apretar al botón de aceptar, ver la lista, y aplicar un promo code.

En el botón, vamos a coger el texto a pesar de lo que dijimos antes, ya que se presupone que el texto es estático. Finalmente, meteremos el promo code ‘rahulshettyacademy’ y cerraremos las pruebas.

Ahora tenemos que meter los waiters para que pueda encontrar todos los elementos.

Vamos a resolverlo con **implicit o explicit waits**

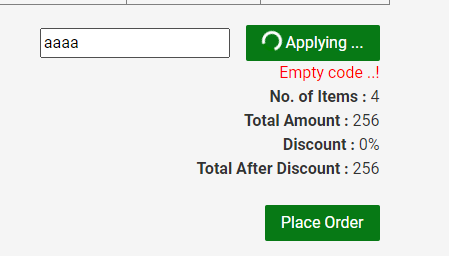
IMPLICIT WAITS



En este caso, le estamos diciendo que espere 5 segundos, de forma global, antes de decirle a nuestro test que haya fallado. De esta forma permitimos que el flujo total del programa se ejecute.

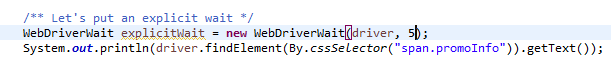
De esta forma, y como dijimos, se aplica de forma global.

Con esto cumpliría, salvo la parte de aplicar el promo code, que se tira más segundos de los necesarios.



Además, que el mensaje aparece después de un tiempo, es dinámico. Por ello se utilizan **explicitWait**.

Para ello, se instancia primero la clase WebDriverWait



Luego utilizaremos los métodos presentes en esta clase.

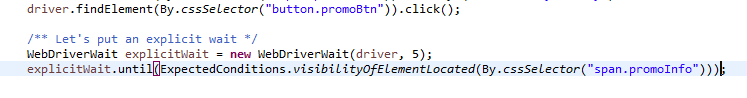
Se pueden esperar:

* Tiempos: con el método wait().
* Eventos: con el método until(ExpectedConditions).

Dentro de los expected conditions habrá una serie de elementos (verlos).

Por lo general, se utilizarán aquellos como:

* visibilityOfElementLocated(By)
* invisibilityOfElementLocated(By)
* textMatches()
* urlContains()
* ...



¿POR QUÉ UTILIZAMOS EXPLICIT E IMPLICIT WAIT?

Implicit:

* Ventajas
  + El código es legible
  + Código más limpio y optimizado
  + Si se está utilizando por varios clientes contra un mismo servicio.
* Desventajas
  + Esconden problemas de rendimiento, ya que no tiene en cuenta estos errores.  
    No fallará la prueba.
  + Nunca **dar más de 10 segundos de implicit wait.**

Explicit:

* Ventajas
  + Afecta únicamente al elemento que queremos que espere.
  + Los errores de rendimiento (performance) desaparecen al visualizar elementos localizados.
  + No se tocan otras partes del código al ser localizado.
  + Utilizarlo si sabemos que hay errores de rendimiento.
* Desventajas
  + Es necesario añadir más líneas de código, haciéndolo menos legible.

FLUENT WAITERS

Es un tipo de espera referente a los explicit wait.

Hay dos tipos de explicit wait:

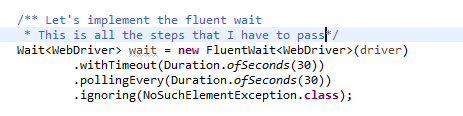
* WebDriver wait: este elemento escucha el DOM
* Fluent wait: monitorea por intervalos de tiempo. Es repetitivo.

En este caso, la principal diferencia radica en que estos últimos encuentran el elemento en intervalos regulares de tiempo hasta que se produce **un timeout** o **se encuentra el objeto**.

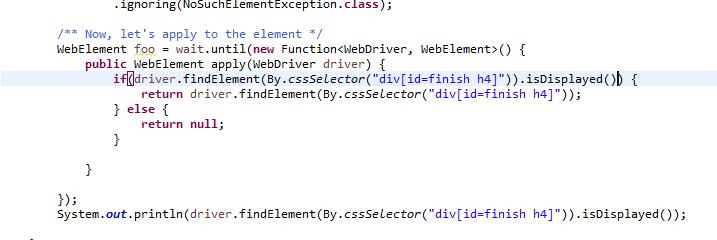
Hay que construir métodos **customizados** en este caso para poder trabajar

Ambos tipos de espera implementan **WaitInterface**.

La mayor parte del tiempo usaremos WebDriver wait, pero el Fluent se usa para intervalos.



Ahora lo aplicamos al objeto



Hay que crear un método para ello.

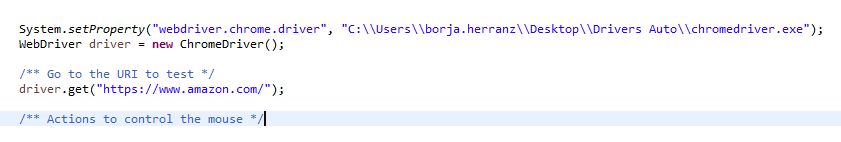
**AJAX, CHILDS AND WINDOWS IFRAMES AUTOMATION**

Vamos a usar eventos y como poder manejarlos.

*Actions*

Vamos usar lo siguiente:

* Mousover
* Interacciones con el teclado
* Clickar en un elemento
* Arrastrar un elemento,...



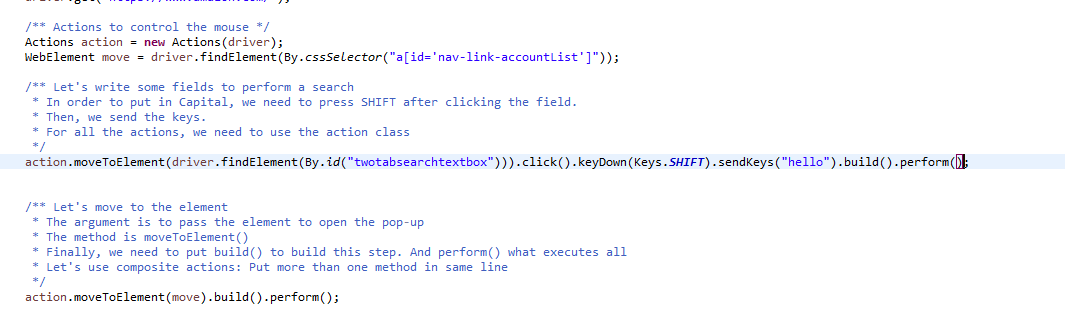


En este caso, vamos a generar una acción de mouseover. Para ello, se utiliza el método moveToElement().

Finalmente, se pasa el método **build()** para poder realizar la acción y se apreta **perform().**

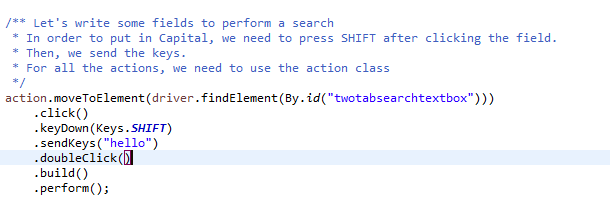
**RECORDAR: Se pueden pasar composite actions.**

Ahora vamos a escribir textos en mayúscula. Para ello se ha de apretar al botón SHIFT para poder hacerlo. Se controlará con los Actions.



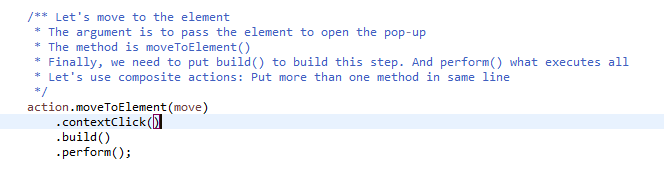
A continuación vamos a seleccionar todo el texto. Actions nos permite utilizar estas cosas.

Para ello utilizamos el doble click, con **doubleClick()**



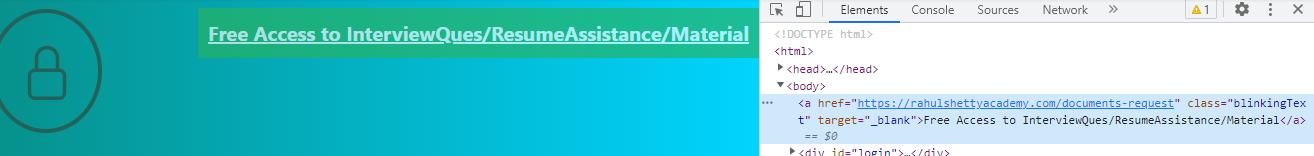
Ahora vamos a dar botón derecho

Es el método **contextClick()**



**Window Handle**

Otro tema importante es la gestión de ventanas que se abren después de pinchar en un enlace. En este caso, se abre una nueva URL, y se debe gestionar con el comando **Actions**.



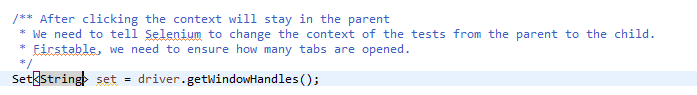
Vamos a apretar a ese enlace.

En este caso, el scope del test está en el padre. El hijo será el enlace que se abra.

Hay que darle el conocimiento a selenium para que cambie el contexto del padre al hijo.

Lo primero es saber cuántas ventanas están abiertas.

El método para esto es **getWindowHandles()** para poder revisar esto. Devolverá un **set de strings.**

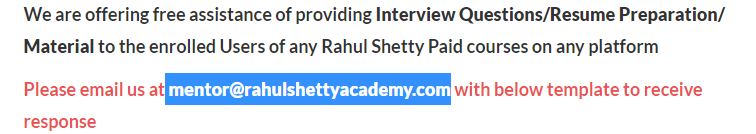


Se guarda como un set [parentId, childId]

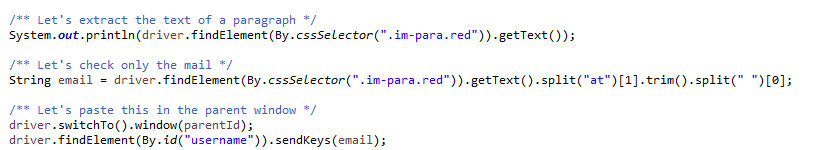
Después, cambiamos el contexto del padre al hijo con el método **switchTo().window(window).**

**Habrá que iterar entre las distintas ventanas con la clase Iterator().**

Una vez en la otra ventana, vamos a coger el mail que se utilizará de nuevo en el formulario para el padre.



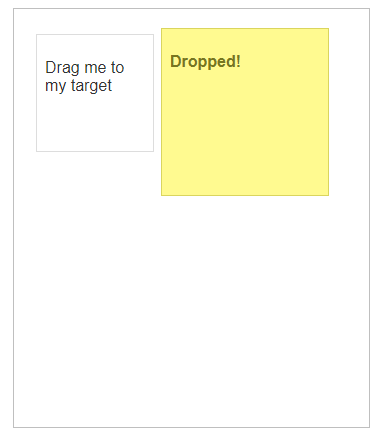
Lo pegamos en el padre, para ello hay que cambiar de nuevo el contexto



**HOW TO HANDLE FRAMES?**

Por último, vamos a aprender como poder gestionar frames (ventanas)

Para ello, debemos tener en cuenta que Selenium no es capaz de identificar los frames. Es por ello, que no se pueden realizar acciones como clickar,...



Se le debe decir a Selenium como poder realizarlo correctamente.

Luego se identificará el elemento.

Un **frame** es una parte de la página web donde se muestra contenido independientemente de su contenedor, con la capacidad de cargar contenido de forma independiente a la página web.

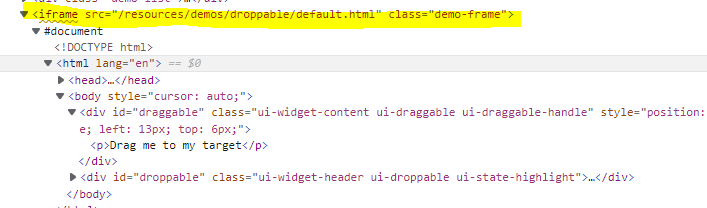
Este contenido puede o no venir de la misma página web.

Debemos cambiar el **contexto,** como hizimos con los windows. Para ello se utiliza el método **driver.switchTo().frame().**

**Hay tres métodos para cambiar el contexto:**

* frame(int arg0): se le pasa el índice del frame. No se recomienda.
* frame(String arg0):
* frame(WebElement arg0): se le pasa el elemento web con el driver.

En el caso que se nos presenta, el ID no aparece



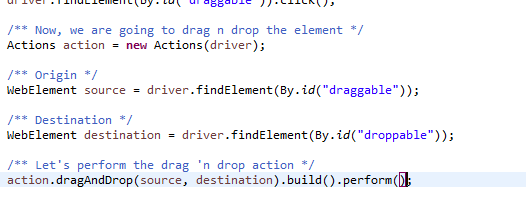
Por ello, debemos encontrar una forma de poder cambiar.

En este caso, vamos a utilizarlo como WebElement.

**DRAG AND DROP**

Esto es una acción muy utilizada en el desarrollo web. Es mover un elemento a otro punto dentro de la página web.

Para ello se debe utilizar la clase **Actions** para mover un elemento del punto A al punto B. Se debe buscar el ID del origen y del destino.



Para volver a poner el estado actual de la página web, se usa el método **defaultContent()**.



**REAL TIME EXERCISES**

Los siguientes capítulos serán referentes a meter ejercicios para probar que sabemos.

1. Vamos a contar todos los links de una página

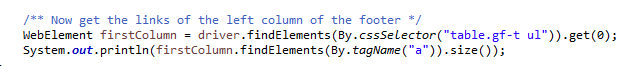
Para ello se van a utilizar los findElements y buscar por tagname.



2. Ahora limitamos el scope al footer

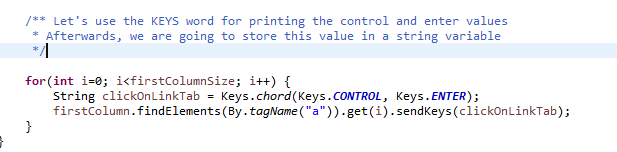


3. Finalmente, le decimos los primeros elementos de la primera columna.



4. Vamos a ver si pinchamos en los links y si aparecen.

Para apretar los diferentes enlaces, se puede apretar el siguiente enlace



EJERCICIO – PASOS

1. Selecciona un checkbox de los tres presentes

2. Selecciona el texto del checkbox (label)

3. Selecciona del dropdown aquel valor seleccionado del checkbox

4. Introduce el valor seleccionado en el EditBox

5. Clica en la alerta y verifica que el texto presente aparece el label

**Manejo de calendarios**

Vamos a aprender como manejar calendarios con Selenium.

Tener en cuenta que los calendarios tienen un carácter dinámico y pueden variar.

El problema está en el Xpath.

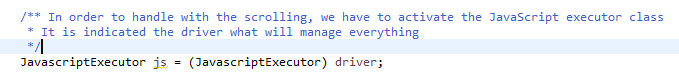
Tenemos que generar while y for loop para seleccionar primero la fecha, el mes, el año,...

**Practical Problems and methods to handle**

1. El primer ejemplo es hacer scroll en la página. Para ello, se ha de utilizar las propiedades de JavaScript.

Se utiliza para identificar objetos que tienen parámetros ocultos

Se ha de iniciar la clase JavaScriptExecutor class (instanciarla) y darle la instancia al driver (se castea)



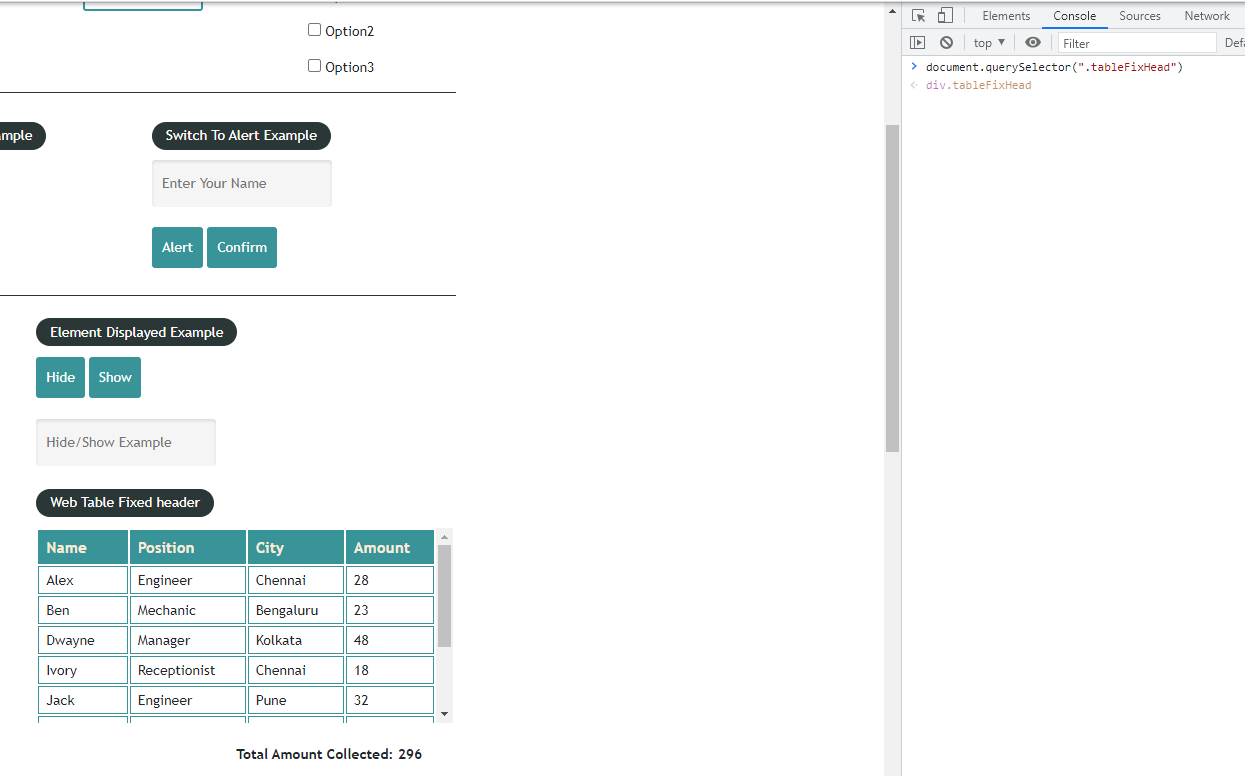
Para manejar el scroll lateral se encuentra el método window.scrollBy()



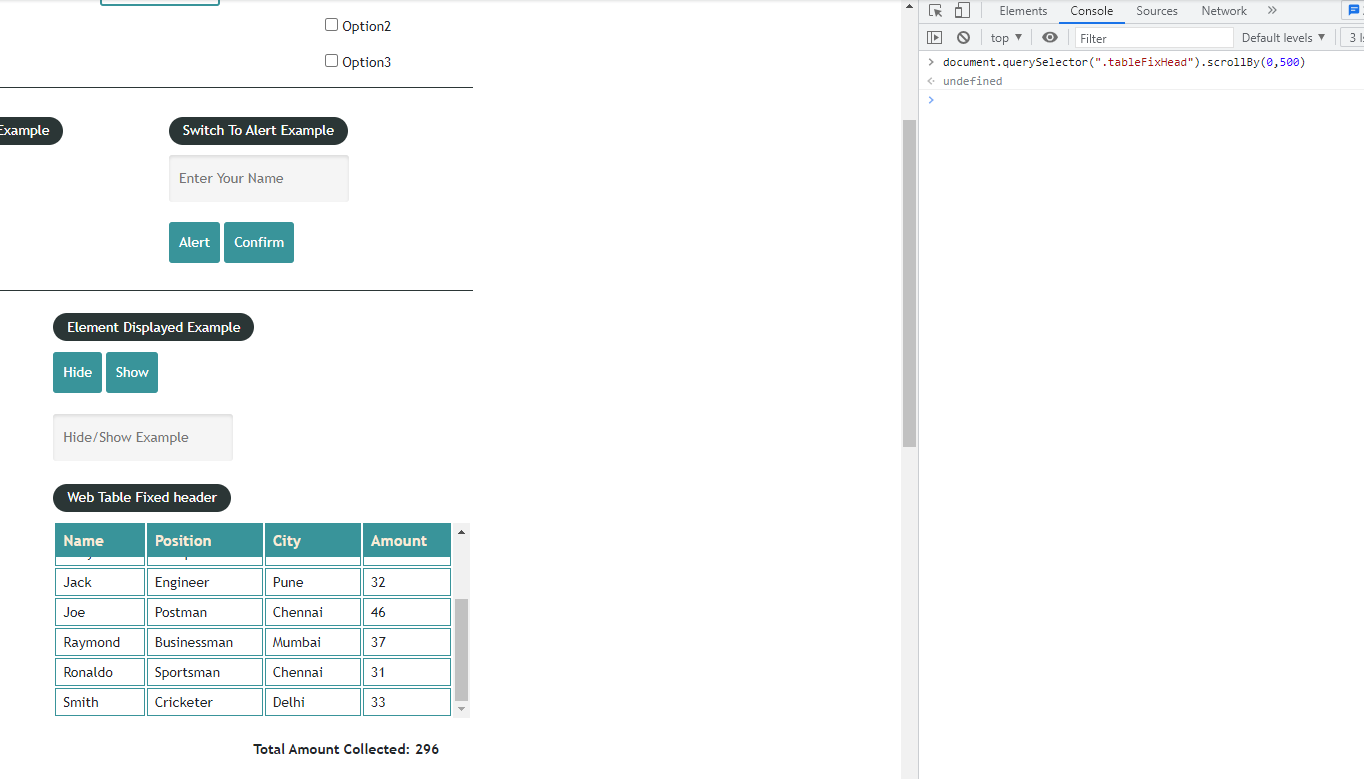
Hay otros métodos que manejan scroll arriba y scroll abajo.

Si por ejemplo queremos hacer scroll en un elemento en específico, se ha de llamar en el DOM al document e indicar con el método querySelector() que es lo que queremos manejar.

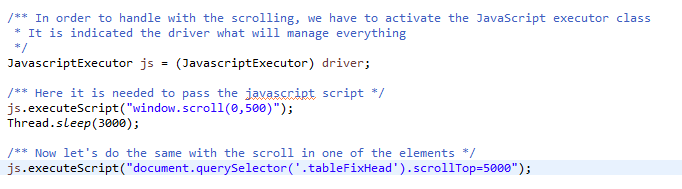
Aceptará css



Se le puede pasar el método al que hacer scroll



Finalmente, se pone en código todo esto:



2. Grid Handle

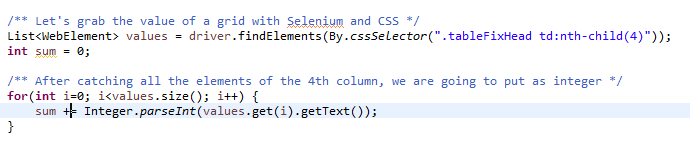
Vamos a revisar como se pueden coger elementos e iterar en una tabla en web.

Para coger el 4 elemento hijo de una tabla en CSS se escribe como:

*td:nth-child(4)*

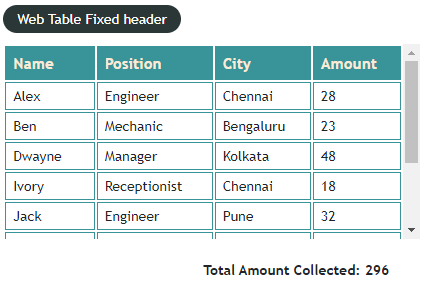
Sin embargo, si queremos que el elemento sea único en el caso de que haya varias tablas, se ha de ir de recorrido de padre a hijo.

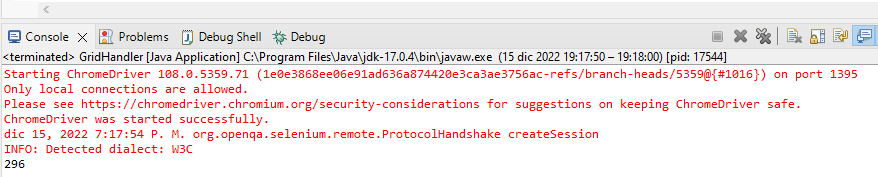
Vamos a coger todos los elementos de la columna 4 y vamos a sumarlos.



3. Parsing Strings to compare with values

Finalmente, vamos a convertir el valor que nos da de 296 en la página y comparar la suma



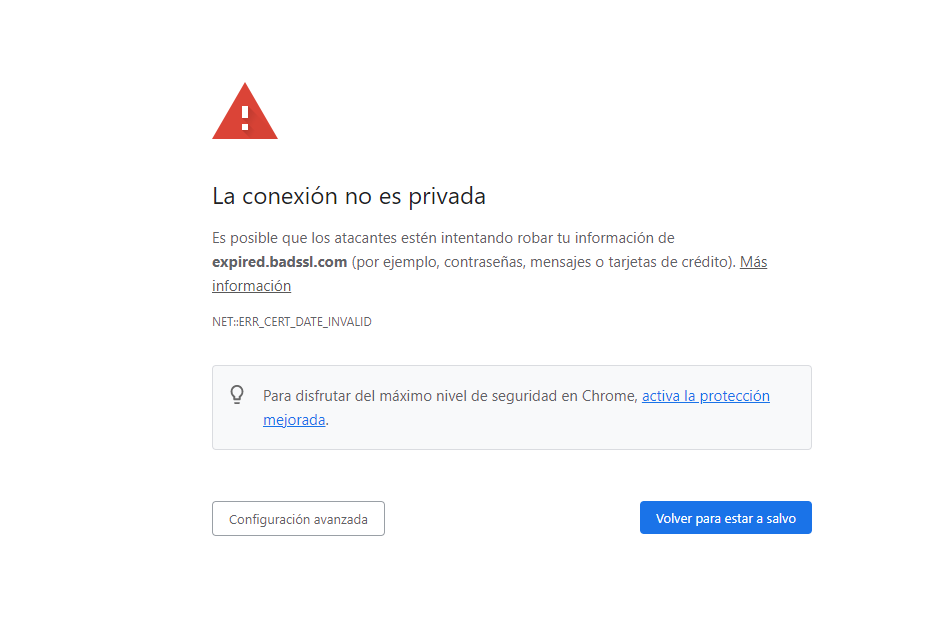


**LESSON 13: MISCELLANEOUS TOPICS IN SELENIUM**

Ahora vamos a ver otros topics igual de importantes

1. Handling HTTPS certifications with Java automation

Vamos a by-passear el certificado HTTPS para que no nos salte el error.



Para ello, se ha de usar el ChromeOptions y pasarlo al controlador del navegador para poder correr y saltarte cualquier error por certificado.

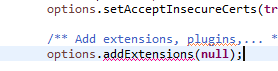
Vale para cualquier navegador.

2. Set proxies, plugins & paths on Chrome Browser

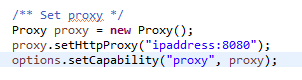
Se pueden añadir nuevas extensions con ChromeOptions.

Lo vamos a utilizar para añadir plugins, add-ons,..., ya que cuando el driver toma el control del browser, no hay ninguna pestaña para ello.

Para añadir extensiones, se utiliza el método *ChromeOptions.addExtensions(String path)*-

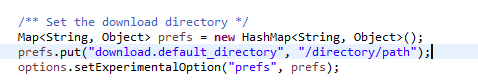


Por otro lado, se puede controlar el proxy en el caso que la web tenga un cortafuegos.



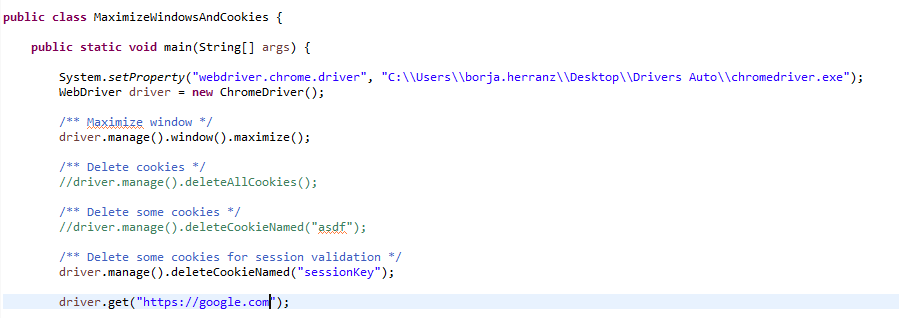
Para más información, visita: <https://chromedriver.chromium.org/capabilities>

Por ejemplo, para configurar la carpeta para descargar los archivos.



3. Maximazing and deleting cookies

Se hace con el método manage()



4. How to take screenshots with Selenium

Hay que convertir el driver en el screenshot object.

Para ello, se puede usar el método *TakeScreenshot* class. Hay que castearla.



Es la única forma.

Luego hay que utilizar el método *getScreenshotAs(FILE)*.

5. Automate broken links

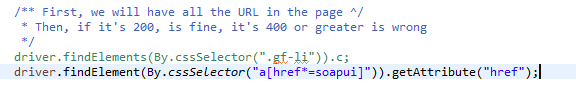
Vamos a probar links que no funcionen y lancen un error.

¿Se mirará con la respuesta de las cabeceras?

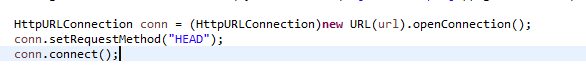
Se pueden ver de forma manual con el DEV Tools – Network – XHR

En el caso de que queramos automatizar, hay métodos que llaman a la URI y recibe el estado de la respuesta.

Para empezar, vamos a coger uno de los links



Ahora, vamos a utilizar el método *openConnections()* para poder utilizar el link roto



Ahora, probando esta conexión nos dará un estado y nos dirá si la respuesta es correcta o no después de conectar.

Vamos a probar con todos los links.



Sin embargo esto no soluciona de todo el problema, ya que cuando falle la primera vez, se rompe el flujo. Vamos a usar **Soft Assertions** para no parar la ejecución.

Hay dos tipos de aserciones:

* Hard Assertions: Las típicas que creamos con la clase Assert
* Soft Assertions

Para este último caso, debemos instanciar la clase



De esta forma quedaría



Se le pasa como una aserción normal.

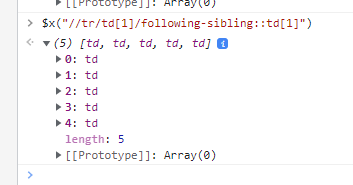
JAVA STREAMS

Voy a verlo un poco por encima para utilizarlo en Selenium.

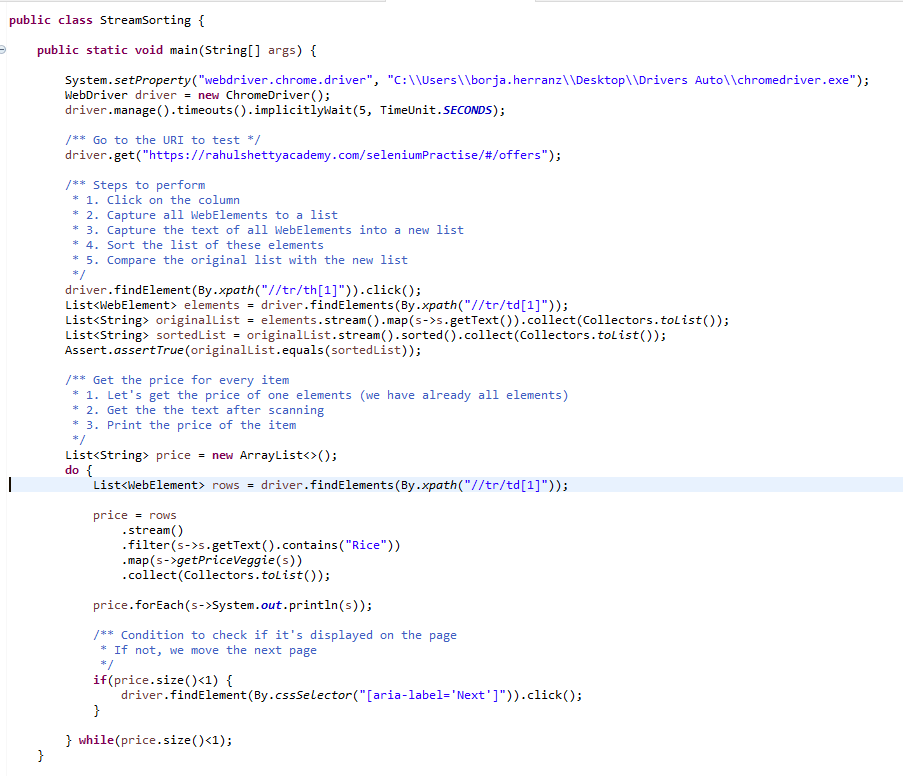
Lo principal a ver:

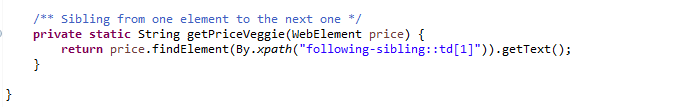
* Se utiliza la clase Stream API 8 para trabajar con colecciones
* La forma de trabajo es llamar al método stream() sobre una colección para poder trabajar
* Se utilizan funciones Lambda
* Se realizan llamadas con los siguientes métodos:
  + filter(): filtra una lista después de checkear una condición. Es un booleano.
  + map(): transforma una lista u objeto en otro objeto, sea del mismo u otro tipo. Es una función.
  + collect(): devuelve una lista, sea un Set, List, TreeSet,...
  + forEach(): itera la lista creada.

Para las tablas, si queremos ir de un elemento al siguiente utilizamos **following-sibling::element**



Por otro lado, vamos a aprender a **Paginar** para estudiar como hacerlo en Selenium.





Finalmente, vamos a filtrar con Streams una tabla de una página web.

15. SELENIUM RELATIVE LOCATORS

Vamos a ver qué locators se pueden utilizar para este caso:

* above(): permite coger el elemento encima del que estamos
* below(): permite coger el elemento debajo del que estamos
* toLeftOf(): permite coger el elemento a la izquierda del que estamos
* toRightOf(): permite coger el elemento a la derecha del que estamos

La sintaxis es:

*driver.findElement(withTagName(“XX”).above(WebElement))*

Sólo se utiliza desde Selenium 4 hacia delante.

Para poder trabajar con estos métodos, hay que bajarse primero el paquete en cuestión



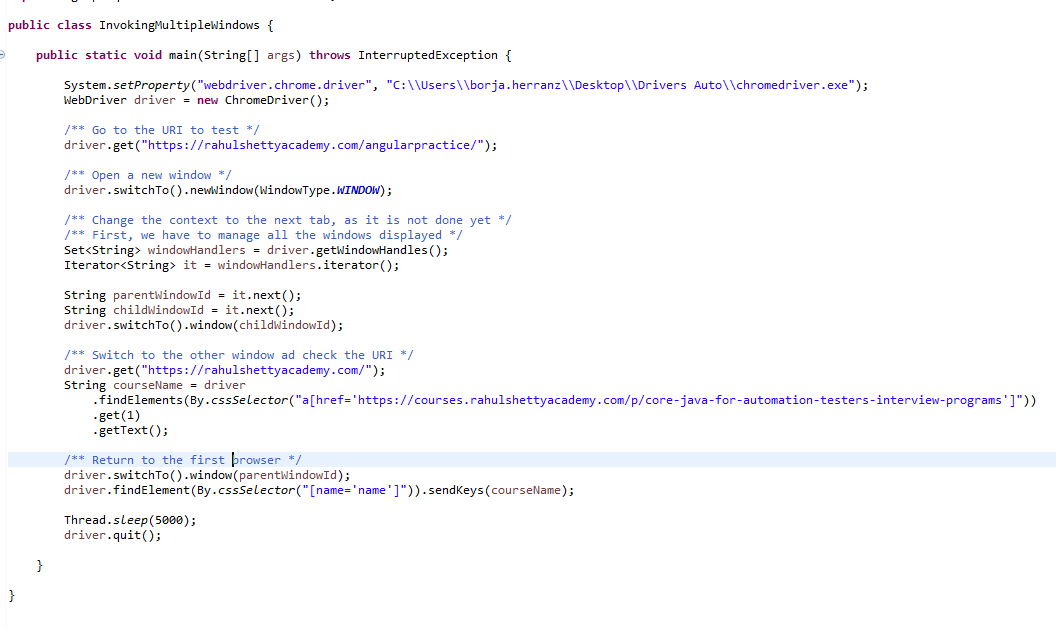
Hay que tener en cuenta que los relative locators no soportan los elementos **flex**



Así quedaría utilizar todos los relative locators.

**Invoking multiple windows using Selenium**

Para esto, necesitamos Selenium 4, sino no funcionará.



En este caso, se utiliza el método *newWindowHandles()* para poder crear una nueva ventana.

Luego, como aprendimos en su día, cambiamos el contexto a la otra ventana y operamos. En este caso, vamos a coger el primer elemento de la página de los cursos y lo vamos a pegar como el nombre en la anterior página.

**Capture Element Screenshots**

Vamos a coger la captura de pantalla únicamente del elemento que queremos, no de toda la página.

Esto solo se puede realizar en la versión de Selenium 4.

Para ello, se ha de utilizar el FileUtils como ya vimos



**Height and Width of WebElement for UX validation**

En la version Selenium 4 se pueden ver el ancho y la altura de un element (de un paddler)

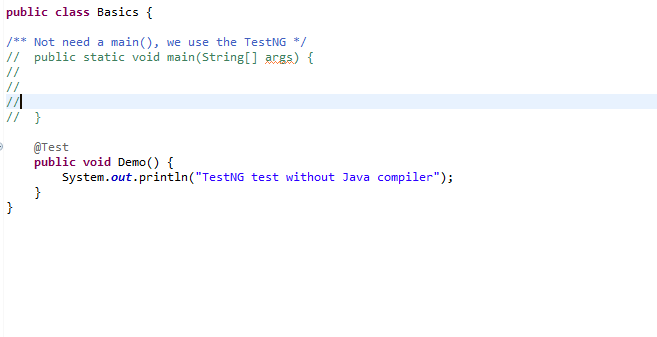
**SECCION 16: TESTNG**

Vamos a aprender a utilizar TestNG como framework para testear. Se le llama ‘Test Framework’

Para correr los métodos con TestNG, es necesario:

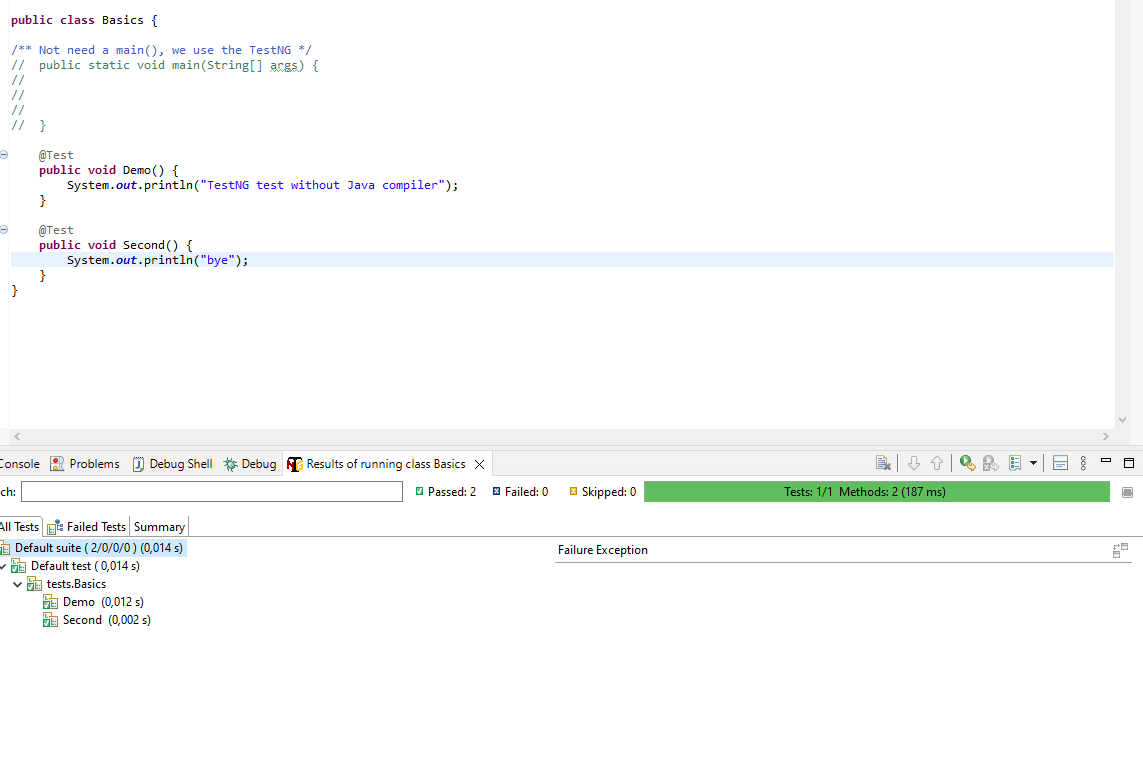
* Instalar el Plugin del marketplace
* Utilizar la anotación @Test

No es necesario que haya un main para ejecutarlo.



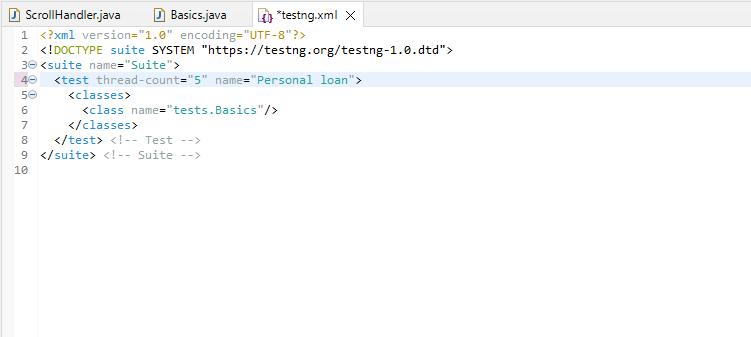
Recordar que en TestNG se pueden ejecutar varios programas a la vez, al contrario que con el main.

Se pueden ejecutar uno detrás de otro en la misma clase Java, ¿pero cómo les ponemos el orden?



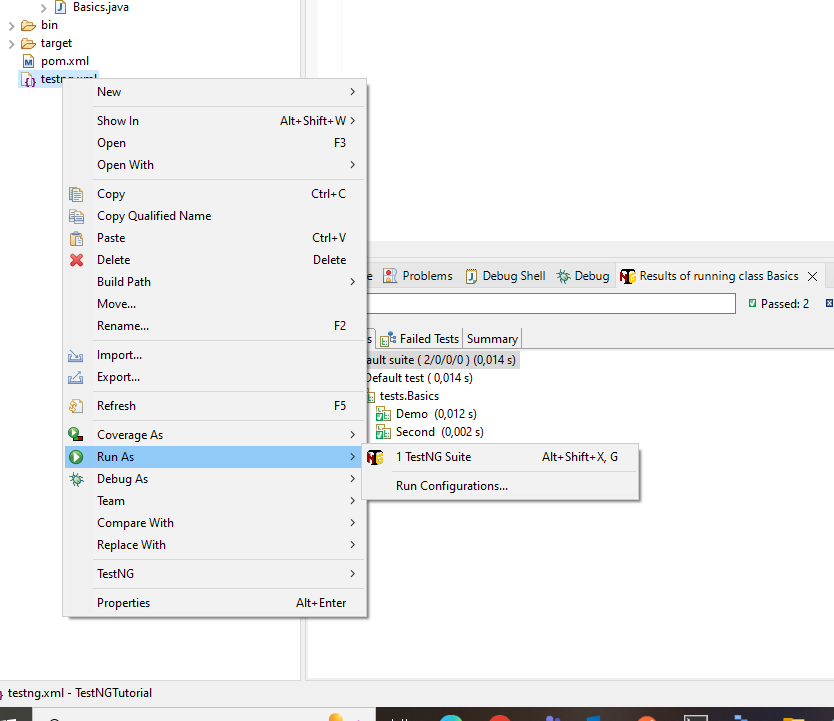
Vamos a definirlo en un documento XML llamado *testng.xml*. Aquí se definen los siguientes parámetros:

* Test Suite: es el cojunto de carpetas de prueba, con diferentes módulos,...
* Test Folder: es el módulo, tiene diferentes casos de pruebas.
* Test Cases: son los casos de prueba que se pueden incluir en el módulo. Es la clase en la que estarán



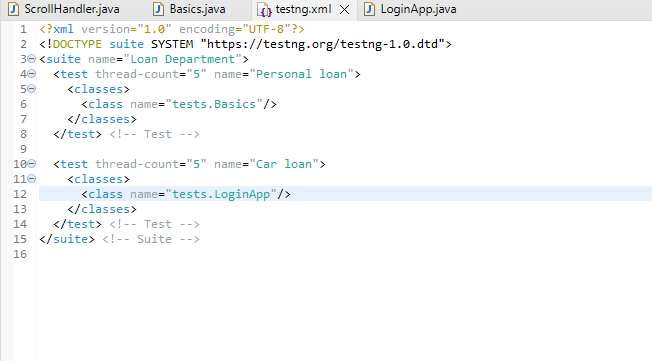
Para ejecutar todos los test cases definidos en cada clase, añadimos:

1. Primero añadimos todas las clases a ejecutar en el XML bajo la etiqueta clases
2. Corremos los test cases como una suite en el XML



Vamos a ver el siguiente concepto que es **priorizar los casos de prueba** con TestNG.

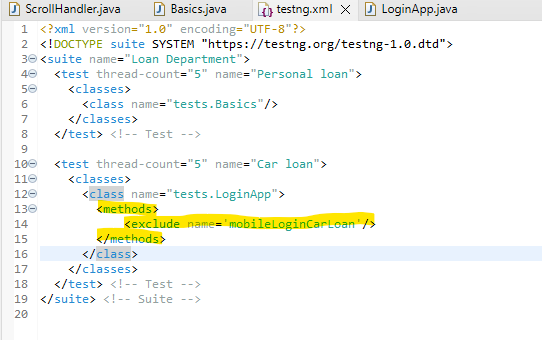
A veces el cliente nos dice que tests ejecutar, cual no... todo se configura en el XML file. Se utiliza esta modularidad para poder **definir los test folders en función de la funcionalidad.**



De momento, para ordenarlo, es subir la etiqueta ‘test’ hacia arriba

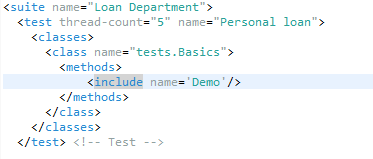
Por último, vamos a **saltarnos (skip)** algunas de las pruebas. Para ello, en el archivo testng.xml se ha de añadir un tag llamado **methods**, que permitirá saltarse estos métodos excluyéndolos.

Usamos el tag **exclude** para ello.



En este caso vamos a quitar el método mobileLoginCarLoan.

También se pueden incluir métodos con la etiqueta **include**



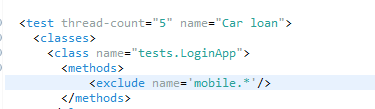
Con ello, se excluirán el resto de los métodos salvo el indicado. La funcionalidad es la inversa.

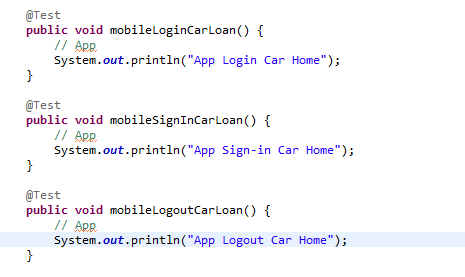
**Ejecutar los casos de prueba a nivel de paquete con RegEx**

Vamos a hacer lo mismo de excluir e incluir, pero en vez de hacerlo uno a uno. Para ello, vamos a utilizar **RegEx**.

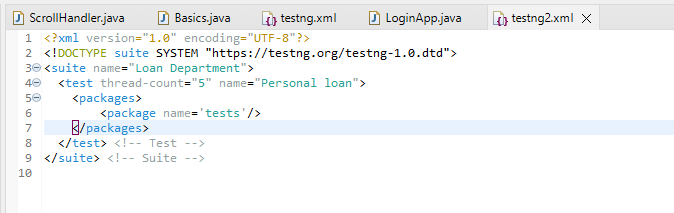
Lo primero, es utilizar una convención para nombrar los métodos o test cases con el nombre del módulo que las contiene para poder trabajar.

Para ello se utiliza la nomenclatura ‘**testcase.\*’** tanto para incluir como para quitar.





También se puede excluir a **nivel de paquete**. Para ello, en el XML file se indica el paquete que se quiera incluir o excluir.



**TestNG Annotations**

Vamos a aprender las anotaciones de TestNG.

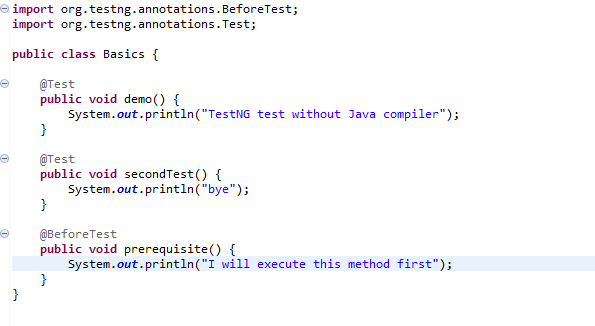
Hay datos que utilizamos varias veces... para que no usemos los automation test cases, vamos a borrar la caché de los datos de base de datos para poder empezar.

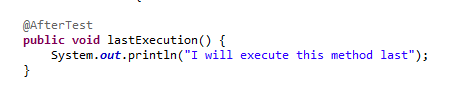
1. Lo primero, es **limpiar los datos**.
2. Luego **ejecutaremos todos los casos de prueba**.

Para poder utilizarlo, hay que utilizar las siguientes etiquetas:

* **@BeforeTest**: se ejecuta como primer método antes de ejecutar el resto de test cases dentro del **test folder**. Será lo que conozcamos como **prerequisitos**. Tiene **alta prioridad**.  
    
  Lo vamos a utilizar para limpiar los datos,... Se utilizan también para Selenium, Apium, REST API,...
* **@AfterTest:** se ejecuta como último método después de ejecutartodos los métodos dentro del mismo **test folder**. Será la **condición de salida**. Tiene **baja prioridad.**

Hay otras anotaciones, como BeforeClass, AfterClass, BeforeMethod, AfterMethod,… cada uno de ellos tiene una funcionalidad que se deberá estudiar y entender dentro de la jerarquía de test suite, test folder,...

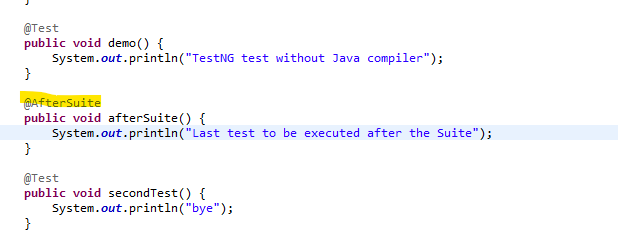


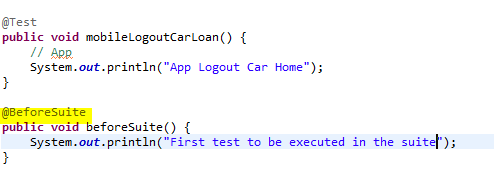


Por otro lado, a nivel de suite, hay otras dos anotaciones que afectan a nivel de Suite:

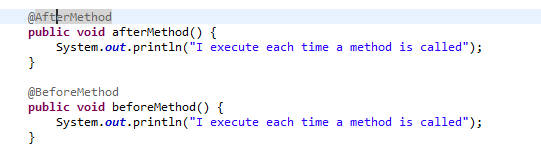
* **@BeforeSuite:** se ejecutará antes de que todos los casos de prueba del Suite se ejecuten. Tiene **la prioridad más alta**.
* **@AfterSuite**: se ejecutará después de ejecutar todos los casos de prueba incluidos en el Suite. Tiene **la prioridad más baja**.
* **@BeforeMethod:** se ejecuta antes de ejecutar cada bloque o cada método dentro de una clase. Si hay 4 métodos, se ejecutará cuatro veces.
* **@AfterMethod:** se ejecuta al final de cada bloque o método dentro de ua clase. Si hay 4 métodos, se ejecutará 4 veces.

En **REST API** se utilizan para los headers, keys,... Lo mismo para Appium y Selenium, son necesarios.



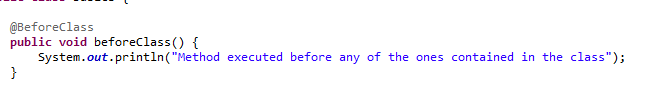


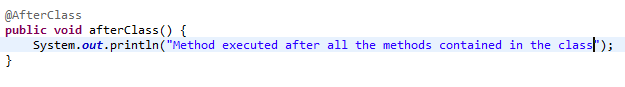
After and Before methods



Lo mismo para las anotaciones:

* @BeforeClass: se ejecuta al inicio de la clase, antes de cualquier método del mismo (haya ya otras anotaciones)
* @AfterClass: se ejecuta al final de la clase, después de cualquier método de la misma (haya ya otras anotaciones)





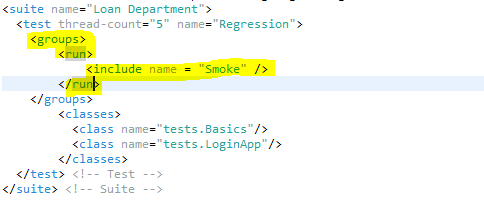
Otra cosa a tener en cuenta es que TestNG ejecuta todos los métodos por **orden alfabético**, por lo que no confundirse cuando uno se ejecuta antes, cual después,... hay un orden establecido.

**TESTNG GROUPS**

Vamos a ver cómo emplear únicamente una serie de test cases específicos al incluirlos en **grupo**. Esto se llama grupos de TestNG.

Hay que añadir un tag parar ello dentro de la anotación @Test(group=’group’)

Por otro lado, en el XML debemos usar la etiqueta *groups* para ello



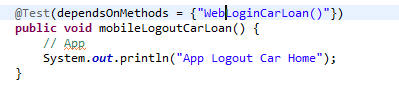
Hay que incluir la etiqueta ‘run’ e ‘include’ para estos casos.

**AUTOMATION HELPER**

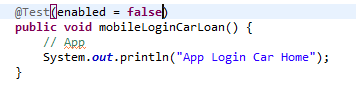
Vamos a usar el keyword **dependance**

Para ello, vamos a aplicar orden a la ejecución de los casos de prueba.

Para ello se ha de utilizar el atributo **dependsOnMethods** dentro de la anotación @Test(dependsOnMethods={}). De esta forma, hacemos depender unos tests de otros y damos orden al conjunto.

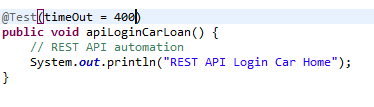


Por otro lado, está el atributo **enabled**, que permite saltarse algunos de los casos de prueba (por ejemplo, si sabemos si hay un bug reportado,...)



Finalmente, hay un último atributo llamado **timeout** que permite poner un timeout para todos aquellos proyectos que tengan un tiempo de espera mayor que X segundos y que no pasan la prueba del implicit wait.

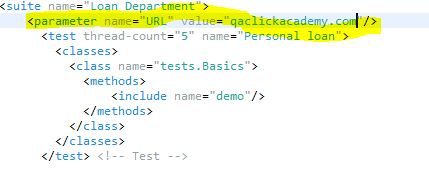
Se define en milisegundos:

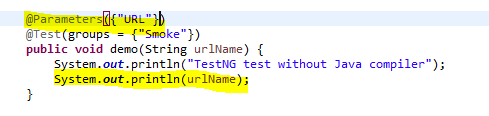


**Parameterizar el archivo XML de TestNG**

Al igual que indicamos en las distintas clases cómo se han de ejecutar los tests con las anotaciones, se puede realizar lo mismo en el archivo XML de TestNG.

En este caso, vamos a ver como pasar parámetros a las funciones definiendo la etiqueta parameters dentro del archivo XML. Por otro lado, en la clase a utilizar, se hace inciso a la anotación **@Parameters** para poder definir estos.





En este caso, se pasa el valor **qaclickacademy.com** como valor para el parámetro. Se pasa en la anotación como una lista de parámetros.

Por otro lado, estos parámetros se pueden pasar a nivel de:

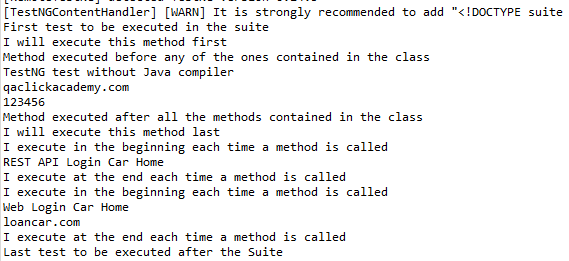
* Test: como el ejemplo enseñado anteriormente
* Suite: común para todos los test folders.

Todos estos parámetros se pasan en este archivo XML file. Luego se pasarán como **DataProvider**.

Se puede hacer override del valor dependiendo de si se define como parámetro global de todo el suite o a nivel del test case.



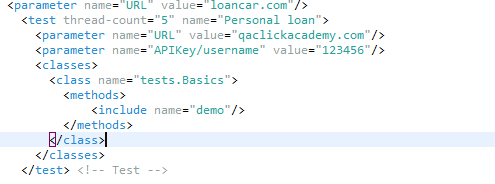
**Resumen de cómo se ejecutan los test cases en base a las anotaciones**

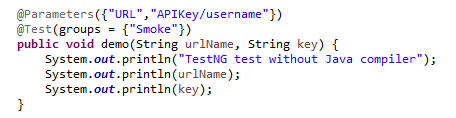


**DataProvider Annotations**

Vamos a ver cómo definir diferentes datos y como pasarlos como parámetro a las funciones con las que trabajamos.

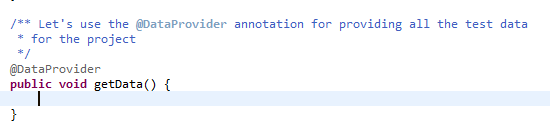
Por ejemplo, se pueden definir varios parámetros en el archivo XML de TestNG y luego inyectarlos como variables en el código. De esta forma se abstrae la lógica de negocio con respecto a los datos.





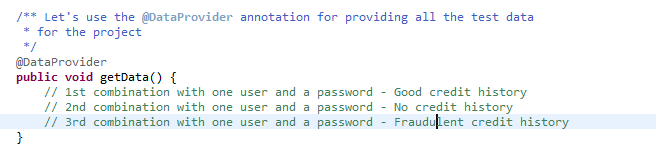
Con **@DataProvider** vamos a pasar datos a nivel de método, no de test folder. Todos estos datos, como el usuario, password,... se les pasa varias veces. Teniendo en cuenta que un suite solo aplica a un proyecto, ¿cómo pasar varios parámetros comunes? Es por eso que entra el concepto de **datasets**.

Lo primero es definir los datos que se van a utilizar en el test. Más comunmente llamado **test data**.



Se define a nivel de módulo.

Se pasan tantas combinaciones como pruebas se tengan pensadas



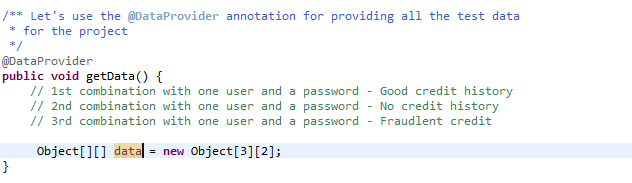
Se ha de pasar la clase Object para empezar a instanciar datos.

Se creará un array de datos para poder empezar; tantos como combinaciones de pruebas haya. Para el caso de prueba, son 3 combinaciones de datos.

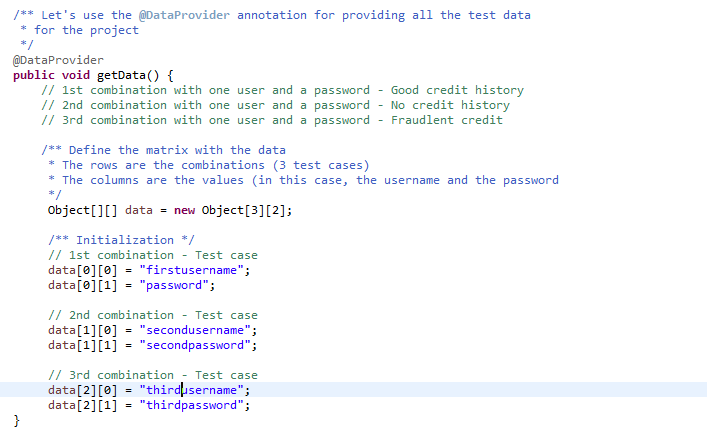
Tal que Object[3][2]:

* Hay 3 combinaciones posibles para las pruebas (o tres casos de prueba)
* Hay 2 posibles valores para los datos (user, password). Lo que contengan es indiferente

Vamos a crear un array de X x X dimensiones

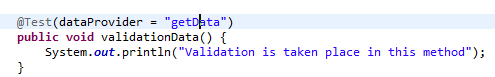


Después de inicializar la matriz, vamos a asignarle los datos necesarios.



Ahora vamos a usar estos valores a los test cases en los que se van a utilizar.

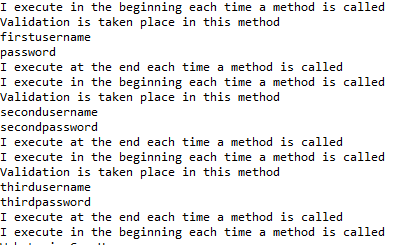
Para ello, dentro de la anotación @Test vamos a pasar como atribute **dataProvider** este dato creado



Se pasa el nombre del método del DataProvider.

A continuación, **se pasan a la función los argumentos a inyectar**. En este caso, se van a pasar el usuario y la contraseña.

El método que analiza cada caso de prueba se ejecutará tantas veces como datos se hayan pasado



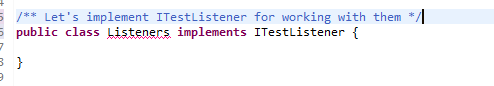
Es por ello, que **si hay un @BeforeMethod o @AfterMethod** **se ejecutarán todas las veces que se comience o acabe una prueba.**

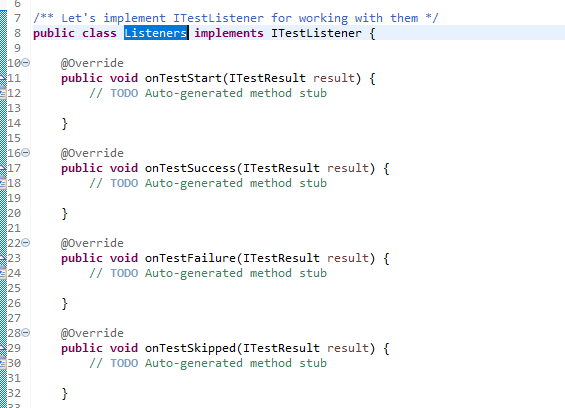
**LISTENERS INTERFACE IN TESTNG FRAMEWORK**

Hay escenarios en los que debe haber listeners para lanzar un trigger cuando se debe realizar un screenshot, si depende de otro método,... que se lance antes o después de un método.

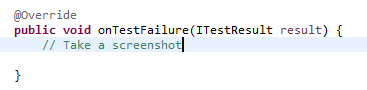
De esta forma se organiza como un bloque específico.

Lo primero es importar el paquete ItestListener.





Todos estos métodos se trigearan cuando se cumpla la condición. Es un listener.

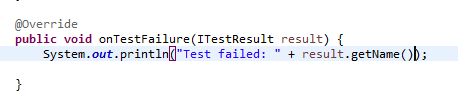


A continuación se le ha de indicar al archivo XML donde se encuentra la clase con los listeners para que pueda cogerlos.

Va a revisar todos los métodos indicados con la anotación @Test.

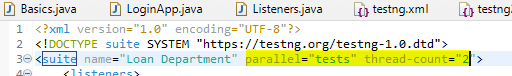
**Running tests in parallel and reporting**

Vamos a coger todos los resultados de los tests con ItestListener. Hay un método que permite coger todos los resultados en un objeto: **result.getName()**.



Recuerda, se coge del objeto **ITestResult.**

**Para pasar todos los tests en paralelo**, se ha de tener en cuenta el rendimiento. Hay escenarios que permiten hacerlo. Hay que pasar el parámetro **parallel** y el paquete. Por último, para indicar el número de tests en paralelo se le ha de indicar en el atributo **threads-count.** Todo dentro del tag suite.



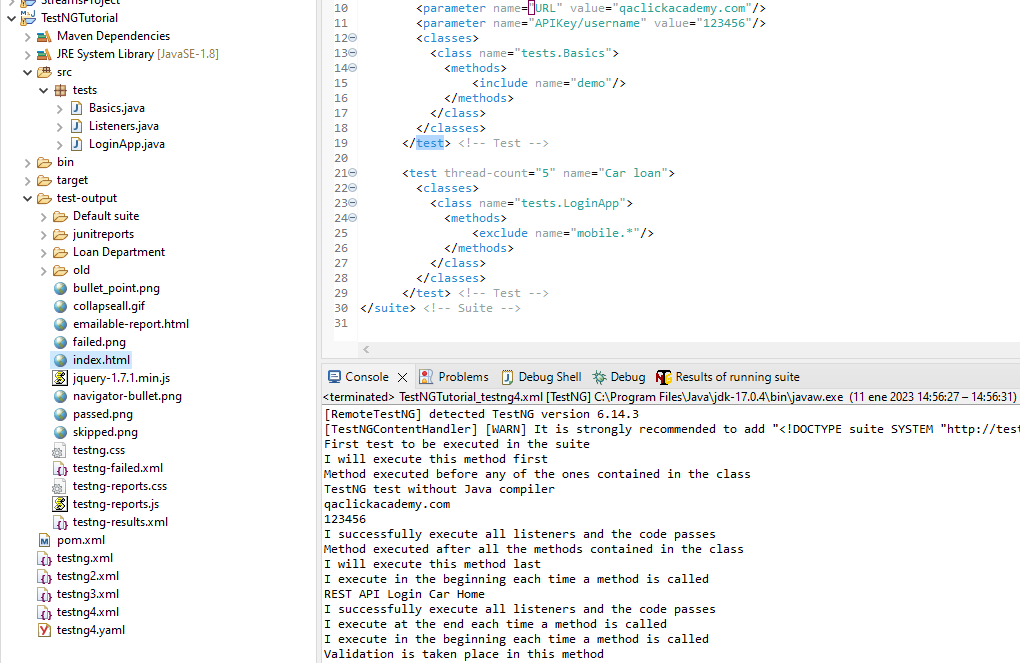
De esta forma van más rápidos.

No tiene sentido en **Appium**, pero si en web.

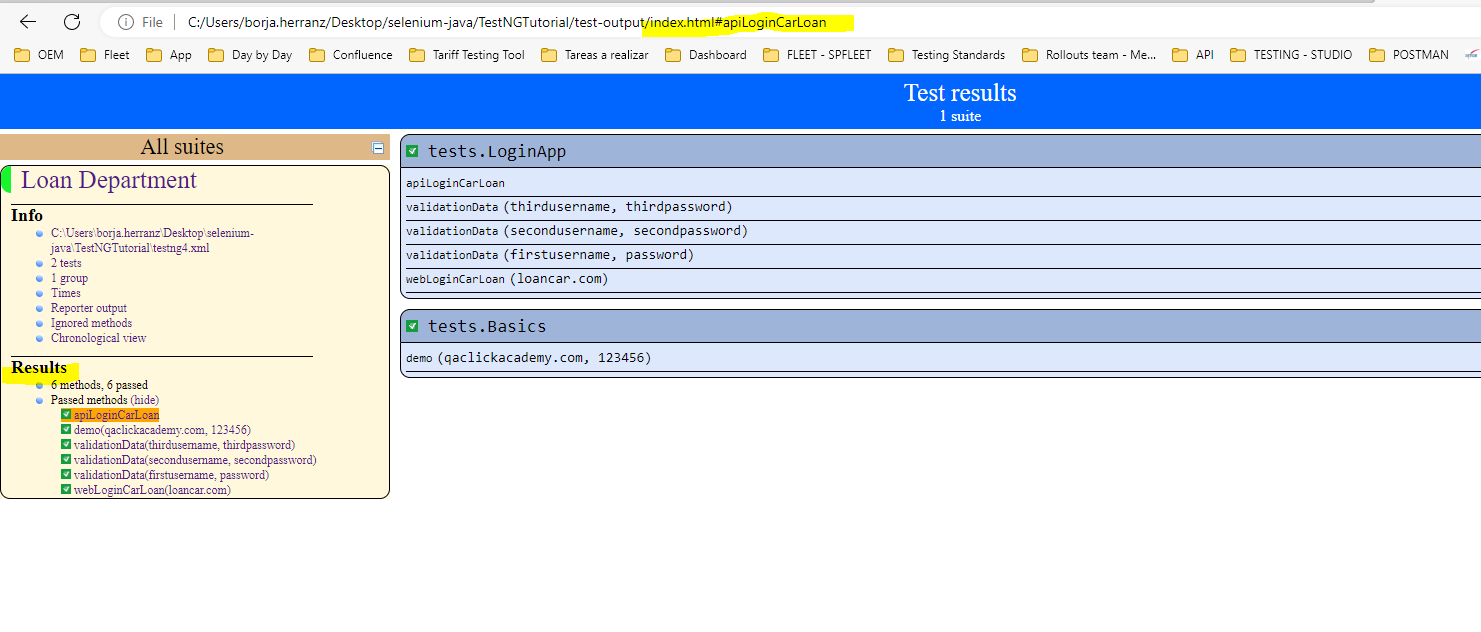
Incluso las clases se pueden ejecutar de forma paralela.



Para conseguir los **reportes, se ha dep ulsar F5 sobre el proyecto y se abrirá la carepta test-output.** En la misma, se ha de pinchar el enlace **index.html**



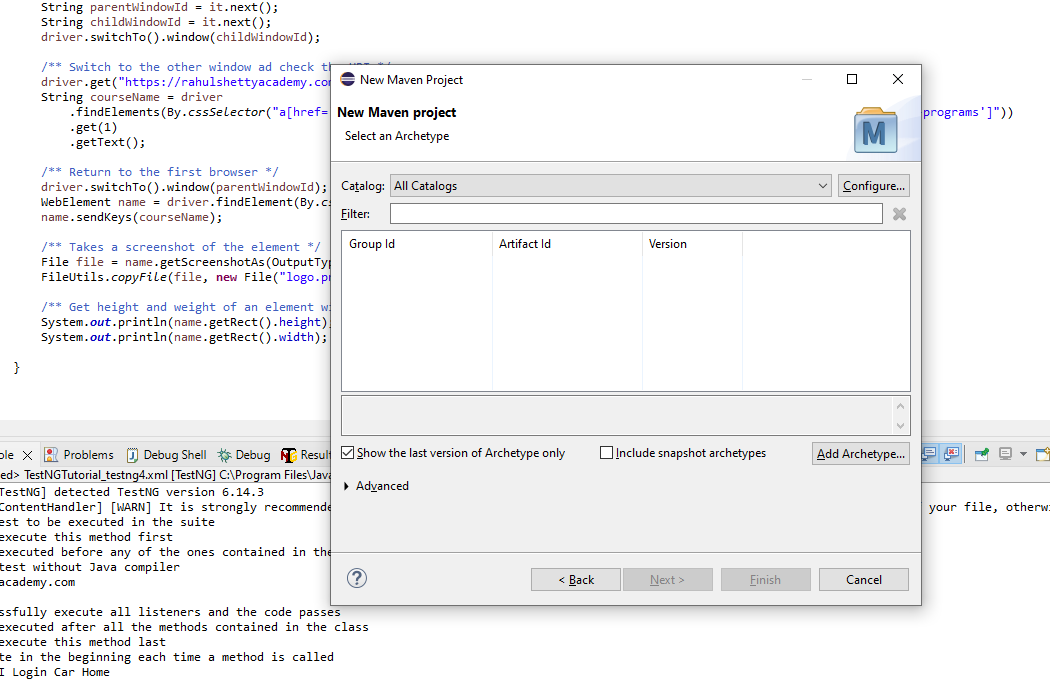
Pinchando en este enlace se nos dirigirá a una página con los resultados, errores,... de cada test ejecutado.



**FRAMEWORK – SELENIUM / TESTNG**

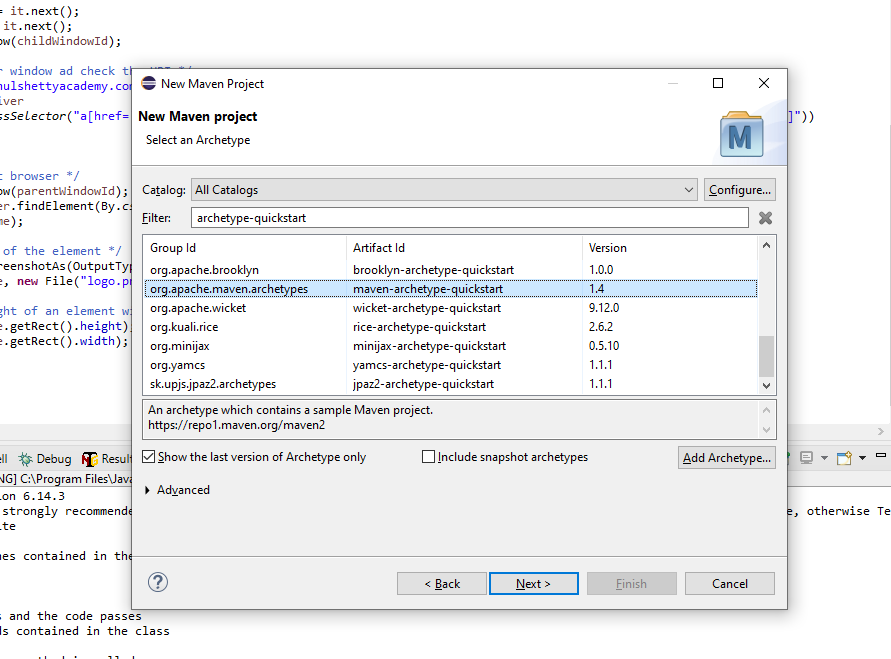
Vamos a crear un proyecto con un concepto E2E para ver una aplicación real de testing.

**Paso 1: Crear un proyecto Maven**



En caso de saber el arquetipo, se puede escoger de la lista.

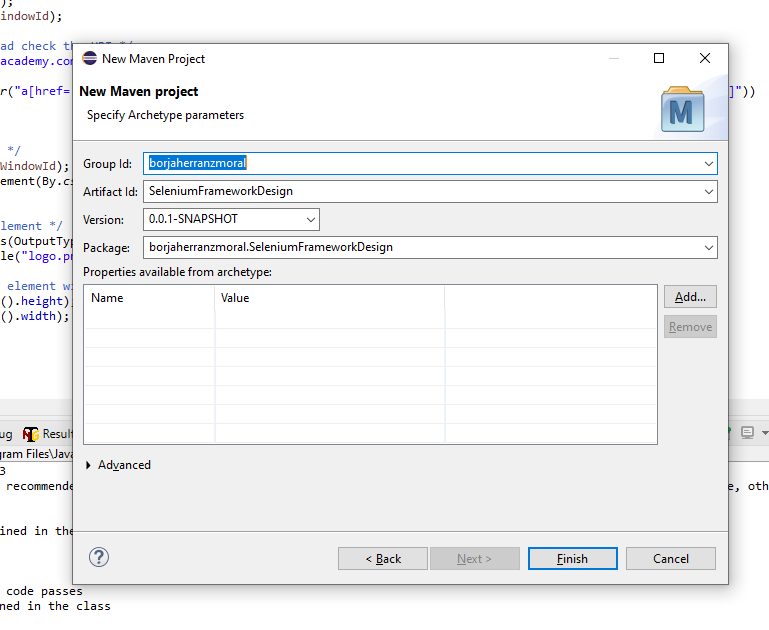
En este caso, vamos a escoger **archetype-quickstart**.



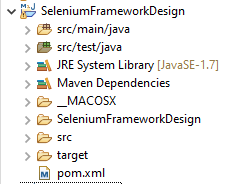
A continuación se describe la estructura del proyecto Maven:

* Artifact Id: Nombre del proyecto Maven
* Group Id: compañía o parent que tendrá varios proyectos. Es una forma de organizarlos
* Version
* Package: indica la estructura de paquetes

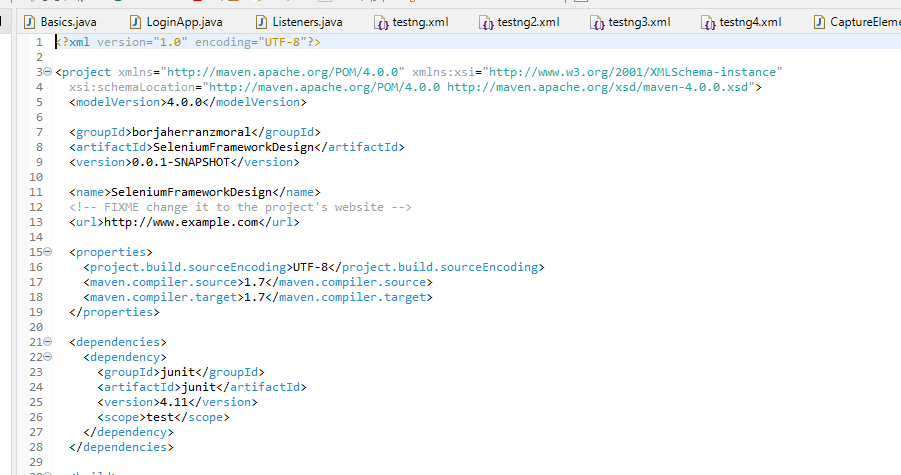
Con estos tres detalles, cualquiera puede trabajar en este proyecto si se lo descarga.



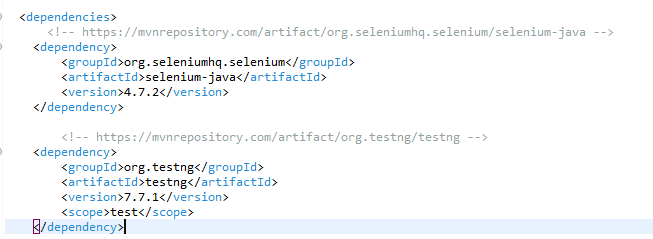
Se creará un proyecto ‘pelado’ para trabajar en él.



Lo primero a tener en cuenta es el **pom.xml**, que tendrá la estructura con las dependencias del proyecto.



Por ejemplo, para este proyecto, vamos a incluir las dependencias para TestNG y Selenium



**Paso 2: Utilizar la página web a testear**

En este caso, vamos a utilizar esta página web

<https://rahulshettyacademy.com/client/auth/login>

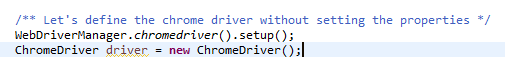
Vamos a ver diferentes productos, y vamos a revisar mensajes de aceptación, ver la cesta antes de comprar, meter un modo de pago y ver que se valida la compra de forma correcta.

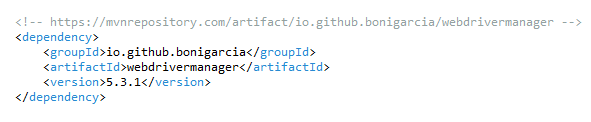
Luego se va a comprobar que se borran items del carro,... borrar pedidos.

**Paso 3: Hacer el login (primer paso)**

Lo siguiente es crear la clase que se dedicará a crear un usuario y/o acceder a la aplicación.

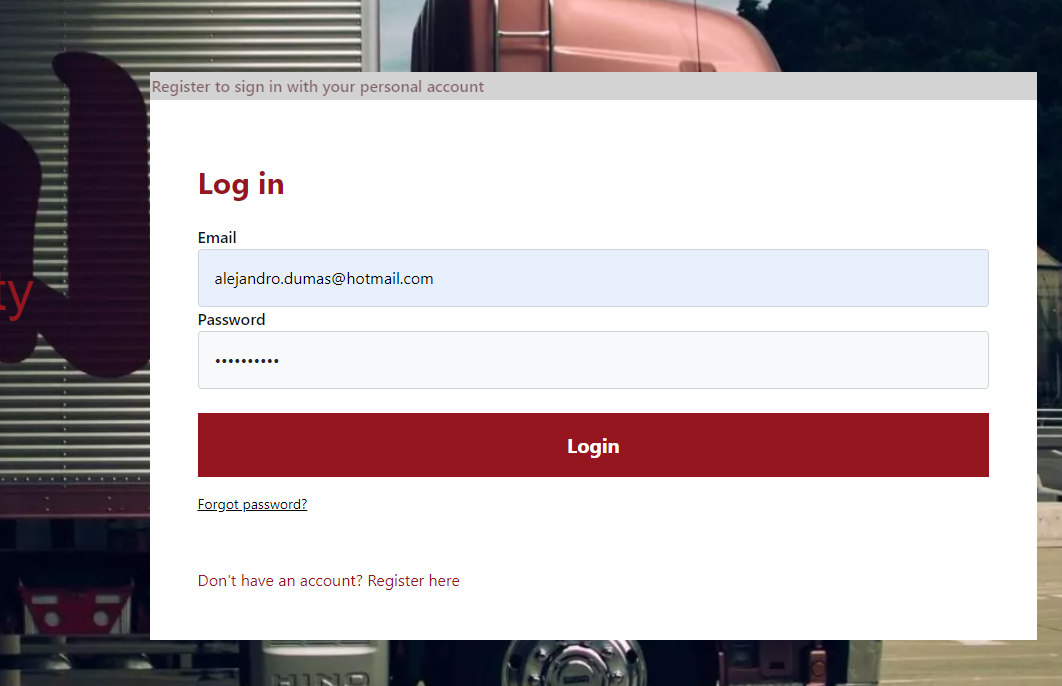
Para ello, vamos a utilizar el WebDriverManager que permite gestionar las conexiones con las diferentes URI sin necesitar utilizar definir el properties.





Esto solo funciona en las nuevas versiones de Selenium, del 4 hacia delante.

Registrado el usuario, vamos a intentar hacer Login.

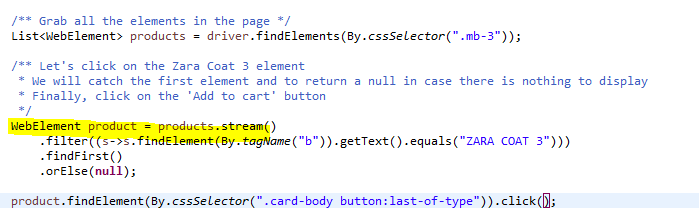




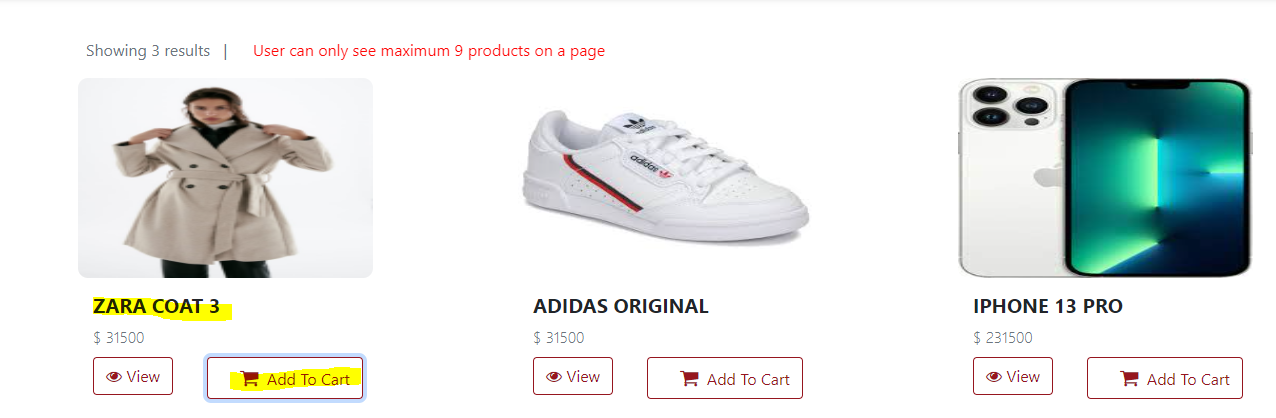
**Paso 4: Añadir el artículo al carrito**

Lo primero que se ha de hacer es encontrar el elemento que se va a añadir, filtrándolo por contenido; y luego añadirlo al carrito.

Aquí se ha de aplicar los Streams(), ya que ayudarán a funcionar correctamente la aplicación.



Por último, para el css selector, y dado que tenemos los botones de ‘View’ y ‘Add to Cart’ se ha de utilizar el atributo sibling **last-of-type** para que nos de correctamente el elemento a clickar.



**Paso 5: Esperas explícitas**

Vamos a usar las esperas activas para ver qué el objeto ha sido añadido a la cesta antes de ir a la página de la cesta.

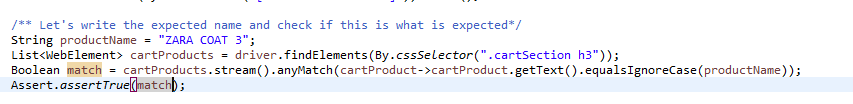
Se va a revisar que salga el mensaje de ‘Correcto’ y esperar antes de que se vaya al carrito. Como aparece después de un tiempo, vamos a usar **Explicit Wait**.

Para ello, se ha de añadir esperas para todos los elementos que queramos verificar, así como para después de logearse en la aplicación



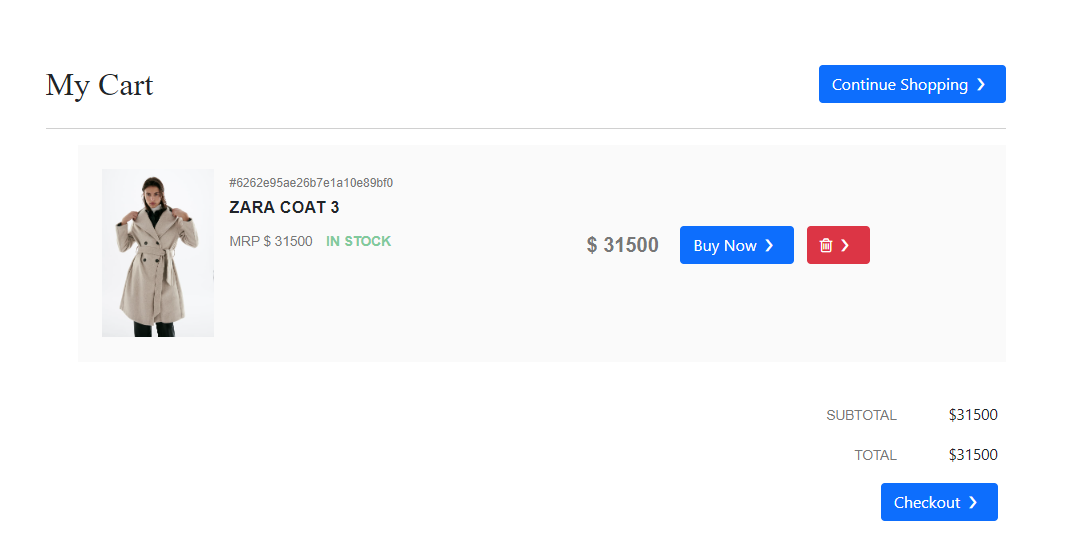
**Paso 6: Revisar que en la cesta está el producto seleccionado,**

El siguiente paso es revisar que todo funciona tal y como se espera en la página de la cesta. Revisaremos aquellos productos seleccionados en la página.



Vamos a poner un booleano para ver si está el elemento que queremos revisar. Para ello, vamos a utilizar Streams().

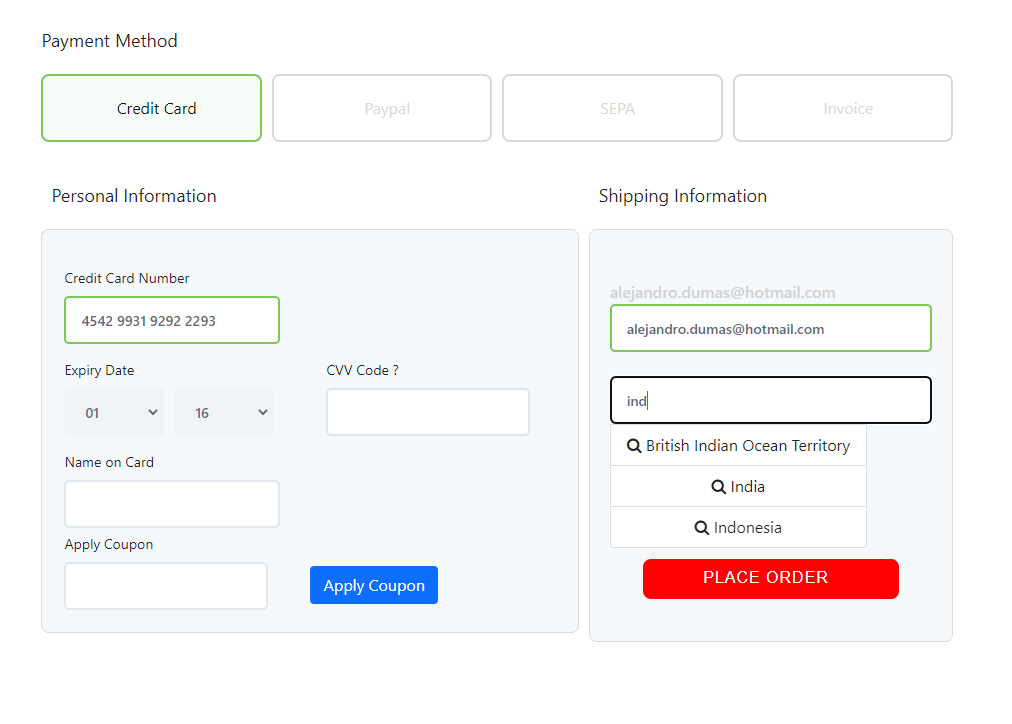
Finalmente, pulsamos el botón de Checkout.



Como todas las clases se llaman igual, se ha de hacer de padre a hijo para llegar al elemento.

**Paso 7: Seleccionar el mercado y revisar correo**

Para este último capítulo de esta lección, vamos a utilizar campos que se autocompletan.



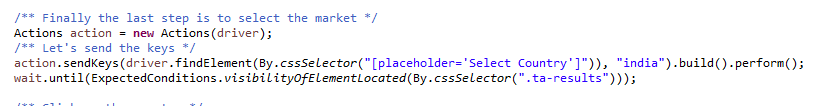
En este caso, los vamos a tratar de otra forma que en lecciones anteriores. **Vamos a utilizar las acciones**.

Vamos a mandar unas letras, revisar que está el mercado que queremos, y seleccionarlo.

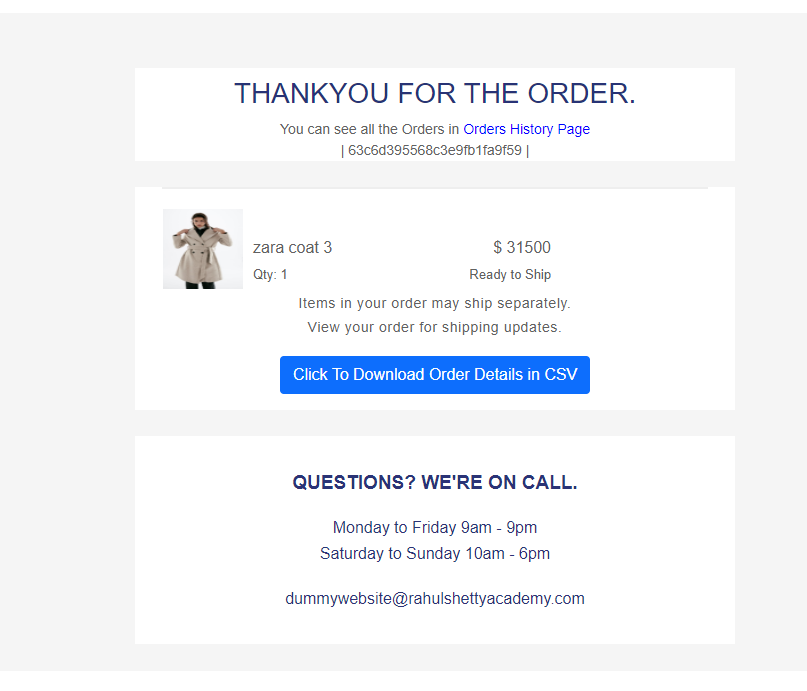
Para ver la documentación: <https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/interactions/Actions.html>

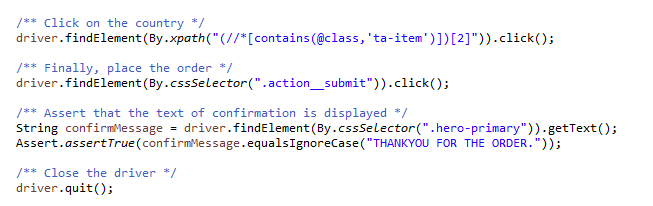
Recuerda que permite realizar acciones complicadas como clickar y arrastrar,...

Además vamos a meter esperas explícitas hasta que se muestre todo el contenido.



Finalmente, después de seleccionar el país, vamos a apretar el botón para mandar la orden y finalmente vamos a meter un Assert para revisar que esté correcto y cerramos el navegador





Los siguientes pasos será refactorizar todo esto que hemos hecho metiendo frameworks y metodologías de trabajo

**FRAMEWORK – PAGE OBJECT MODEL**

Vamos a introducir el concepto de modelo de desarrollo basado en páginas.

Por el momento utilizamos los localizadores en una sola clase que se ejecutan de forma secuencial. El problema es que los métodos cambian durante el ciclo de desarrollo, teniendo problemas de mantenimiento.

Para no tener que revisando cada localizador, o modificarlo según conveniencia.

Como no queremos tener que estar revisando el código, vamos a separar la lógica de los localizadores en otro archivo. Aún así, tampoco es mantenible tener 200 localizadores para diferentes clases.

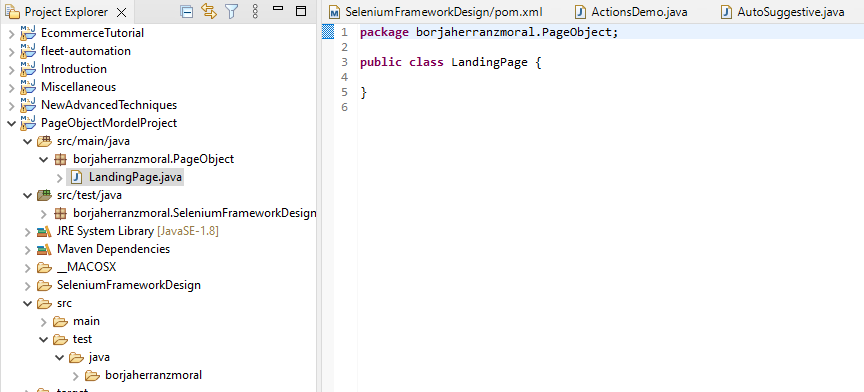
Aquí entra el concepto de **Page Object Model**. Imaginamos que tenemos 4 clases con todos los localizadores. Vamos a meter todos aquellos que compartan funcionalidad en una misma clase. Es decir, si utilizamos 4 veces el login en cuatro clases diferentes, ¿por qué no hacerlo en una única clase?

Es una forma de trabajar, no un framework. Así se quita la abstracción de separar los localizadores.

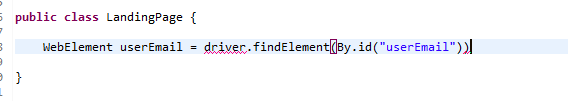
**Paso 1: Refactorizar el código**

Vamos a refactorizar el código anterior para poder utilizar el nuevo modelo. Para testear, vamos a separar el proyecto.

Lo primero es en la carpeta de **src/main**, vamos a crear la clase **LandingPage**. Al contrario que en el caso anterior, vamos a usar el source.



Lo primero que vamos a hacer en esta página es meter las credenciales del usuario que va a probar el login, con todos los objetos,...



¿Pero donde metemos el driver?

Se creo en el proyecto de **StandAloneTest**. Hay que mandarselo.

Primero vamos a utilizar el constructor.

Hay que instanciar la clase a utilizar en el test

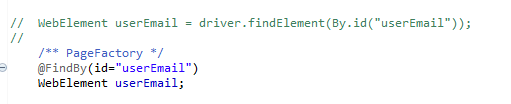


A continuación, en **LandingPage.java** vamos a pasarle el parámetro al constructor

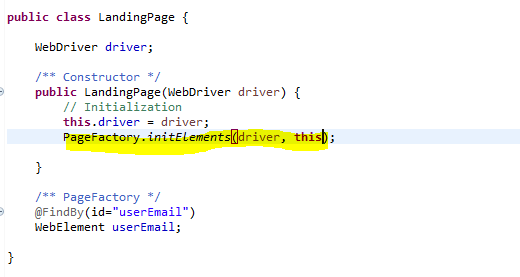


A continuación vamos a meter el concepto de **PageFactory**, que permite reducir la sintaxis para crear el objeto WebElement.

En este caso, vamos a utilizar anotaciones para ello



Sin embargo, no todo está hecho, ya que no se ha inicializado el driver para que el PageFactory lo coja. ¿Cómo lo hacemos? Directamente en el constructor, que será el primer método en ejecutarse.



Se le pasa como argumento el driver y la clase a ejecutar para inicializar los elementos. **Sino no lo va a entender**.

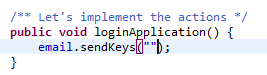
Vamos a meter todos los pasos como son el password, email y el login



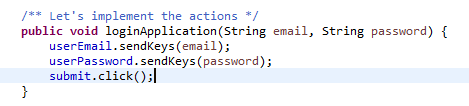
**Paso 2: Implementar la funcionalidad**

Una vez definidos los atributos, vamos a incluir la funcionalidad de la clase. En este caso, apretar al botón de logearse y mandar anteriormente las credenciales.

Lo primero es meter las credenciales

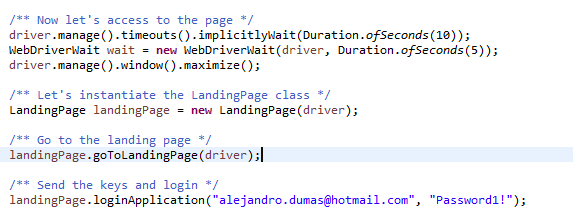


**No vamos a hardcodear el email, porque queremos que se reutilice para varios casos**



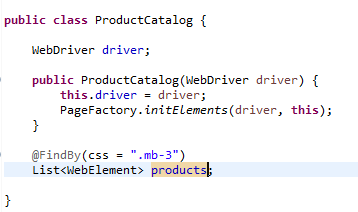
**Los pasamos como parámetro de la función**

De la misma forma, vamos a separar la lógica para acceder a la página de inicio

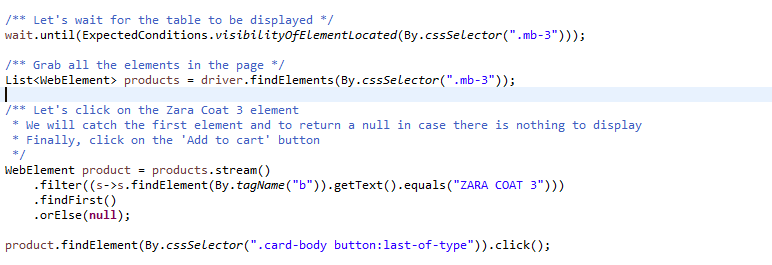




Repetimos el proceso con la parte del catálogo



Sin embargo, lo siguiente es meter esperas activas



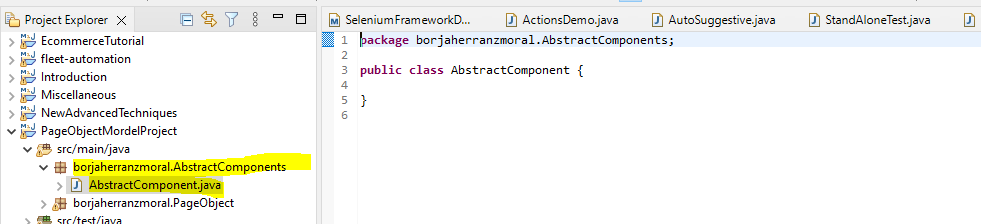
Al ser común en varias partes, vamos a abstraer la lógica para reutilizar el código.

**Paso 3: Abstraer componentes para ser reutilizados**

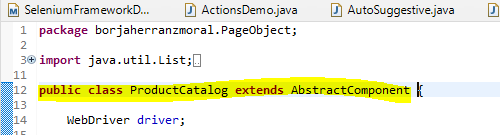
Vamos a evitar duplicar código y reusar aquello que ya tenemos.

En este caso, vamos a reutilizar las esperas explícitas.

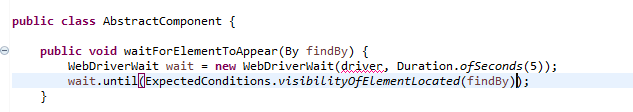
Para ello, creamos un nuevo paquete y metemos las escuchas activas en una nueva clase que repita el comportamiento



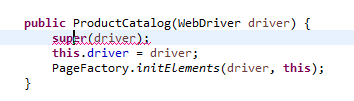
Vamos a utilizar herencia para que la clase del **LandingPage** herede estas escuchas,

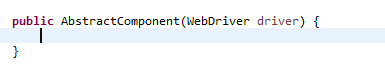


Finalmente, metemos todo en un método con las esperas



Finalmente, para hacer constancia del **driver**, en el constructor vamos a mandarle el elemento de hijo a padre con la palabra reservada **super(driver)**.

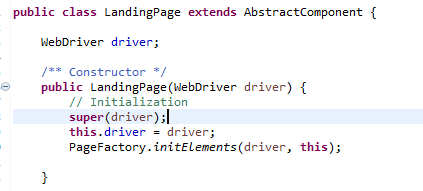




Finalmente, creamos un atributo similar

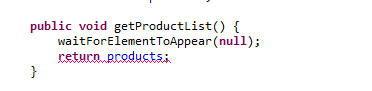


Vamos a extender las esperas activas en todas las clases del proyecto



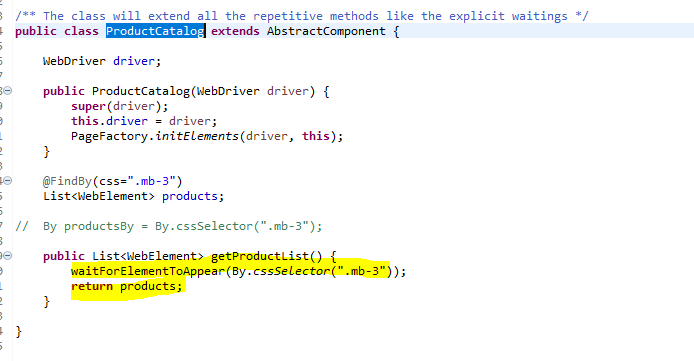
Así es como quedaría la clase y todos los elementos únicos

Ahora vamos a seguir implementando los métodos propios de la clase **ProductCatalog**

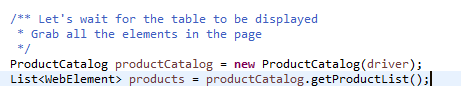


Se crea un nuevo método para recibir los productos y juntar las esperas explicitas definidas en la página con aquellos elementos que se repiten.

En este caso, lo que se va a hacer es pasarle como argumento el atributo a esperar y luego devolver la lista.



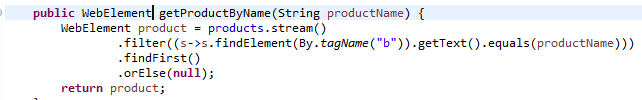
Implementamos la nueva clase en el código



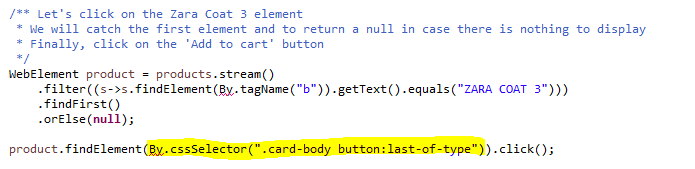
**Paso 4: Más implementaciones**

En este caso, vamos a ver como trabajar con los Streams().

En este caso, vamos a meter una nueva nueva clase que busque si un elemento está dentro del catalogo. Para ello, generalizamos la clase

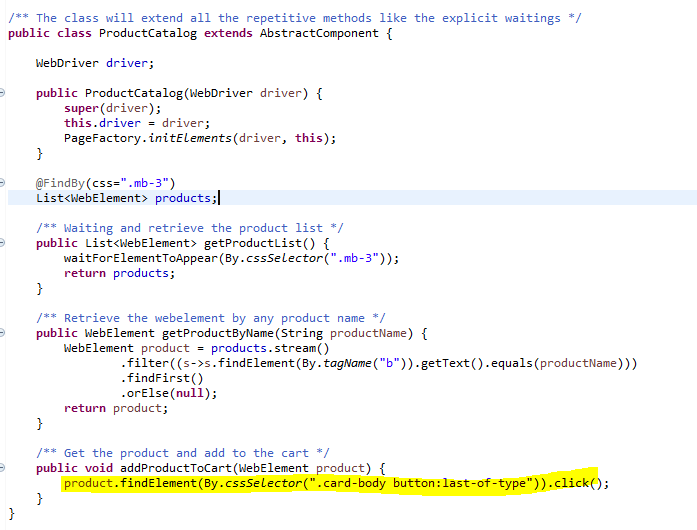


En este caso, no podemos meter este locator como un @FindBy



El problema es que encuentra el elemento, pero debemos pasarle un By. Por lo que no nos permite

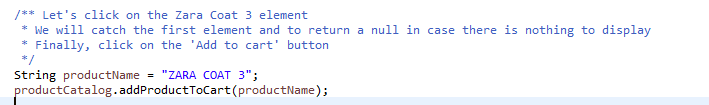
De esta forma, la clase de ProductCatalog queda como:



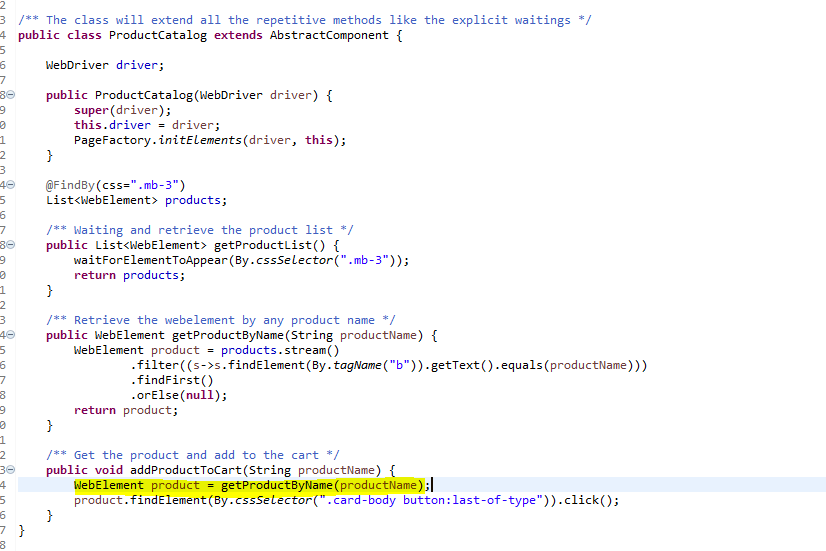
Se puede separar la lógica del by, pero al no reutilizarla, no tiene sentido.

Sin embargo, en los videos lo hacen metiendo las clases dentro del pageObject y abstrayendo los pasos.

Clase que testea

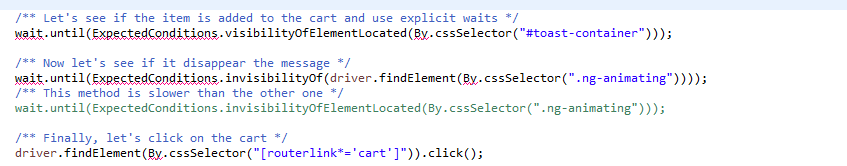


Clase que contiene el modelo



Vamos a seguir este procedimiento.

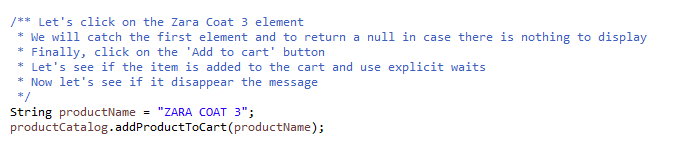
Por último, a continuación tenemos una serie de escuchas activas seguidas antes de añadir al carrito para ver que se muestra el mensaje



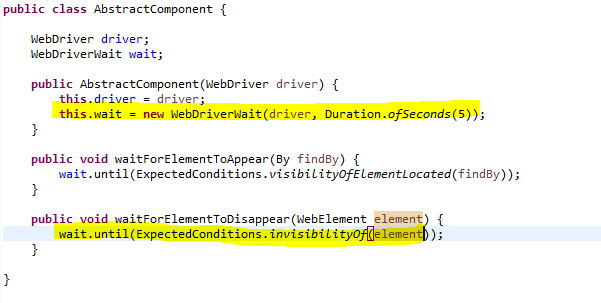
Vamos a meter todo en la clase de **abstraccion**

Así es como quedaría.

Clase controladora

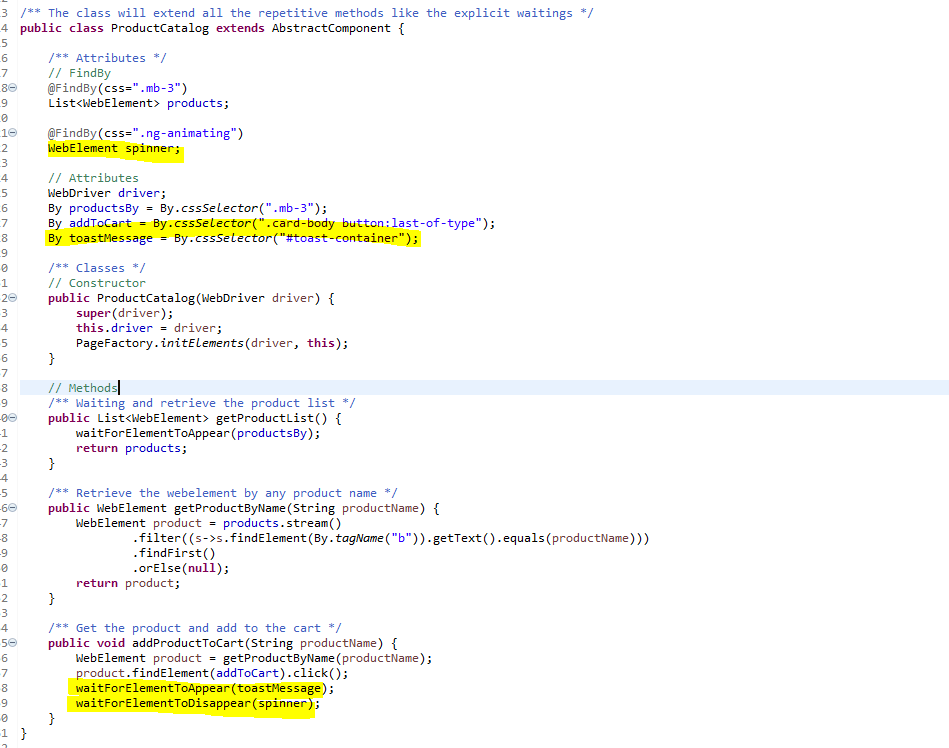


Clase abstracta



En este caso, hemos metido una duración de 5 segundos en el constructor ya que es común a toods los casos (revisar)

Por último, vamos a revisar que desaparezca el elemento en la página creada para el catálogo



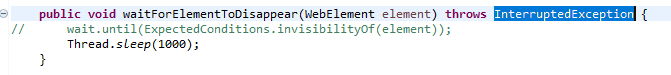
Como podemos ver, se ha separado el spinner ya que se puede entender como un WebElement. Por otro lado, se ha separado la lógica para los atributos.

De esta forma el spinner se ha utilizado como un PageFactory.

\*\* Apaño

Vamos a cambiar el invisibility por iun Thread.sleep(1000) ya que no desaparece rápidamente. Se debe a un fallo en la aplicación del tío. Está esperando que desaparezca otro spinner.

De esta forma quedaría como:



**Paso 5: Más métodos abstractos**

Hemos visto que el header, el footer,... todos son comunes a todas las páginas. Entonces, ¿por qué no utilizar la clase específica para ello?

Vamos a meter ahí todo lo que es ir a la cesta, confirmar,...

**Ojo, las validaciones o aserciones no pueden ir en PageObject**

Como estamos instanciando muchas clases... no tiene sentido. Vamos a hacerlo de tal manera que solo tengamos que instanciar una única clase

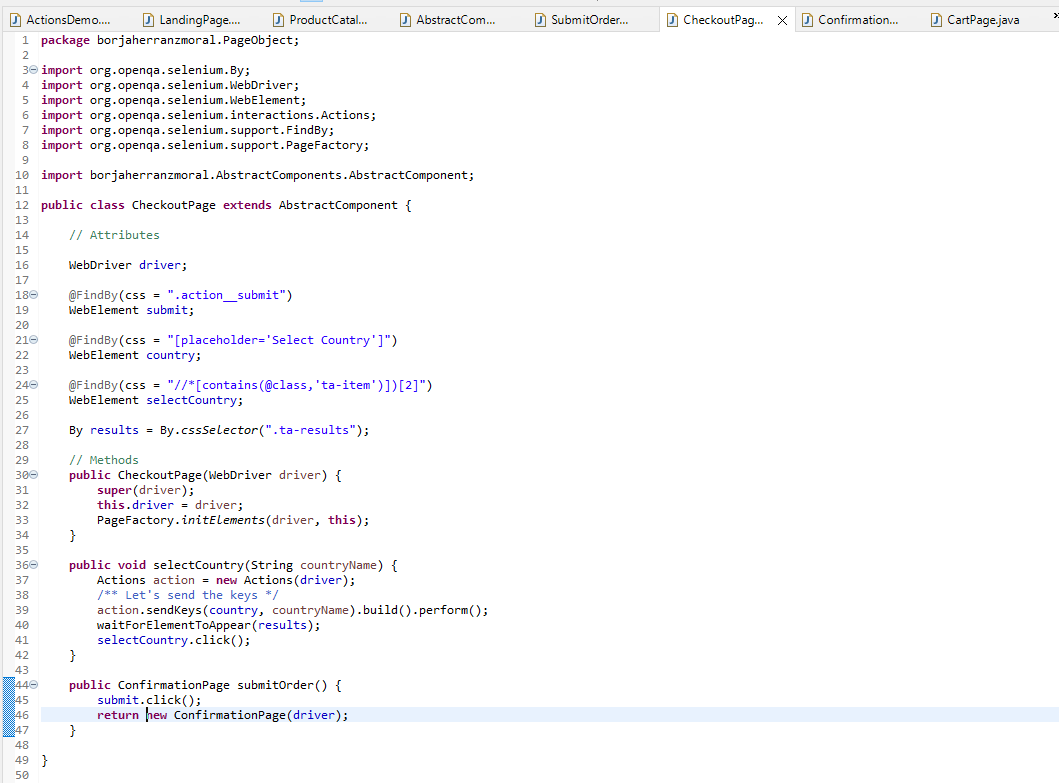
**Paso 6: Completar todo el proceso**

Vamos a terminar el proceso de meter todas las clases en un PageObjectModel y así cerrar este capítulo.

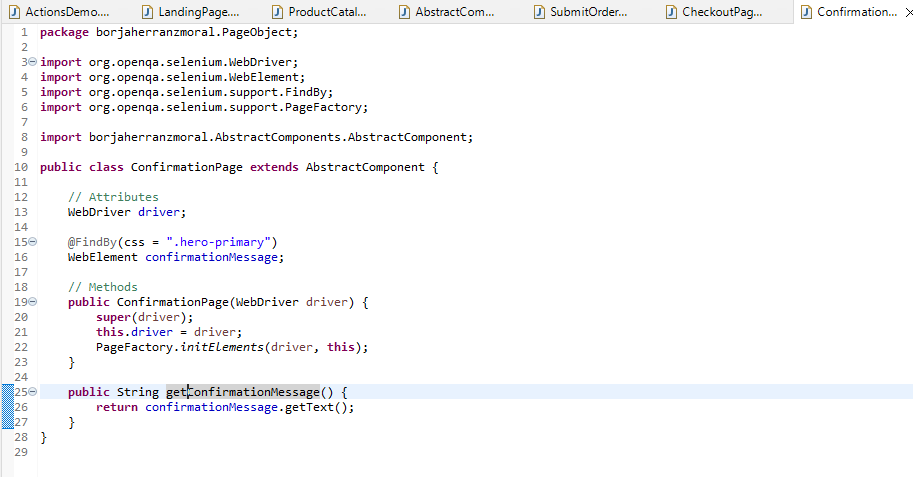
Vamos a crear la clase de **CheckoutPage** para el checkout, y otra clase **ConfirmationPage** para confirmar la compra.

Finalmente, el código para añadir el carrito queda como

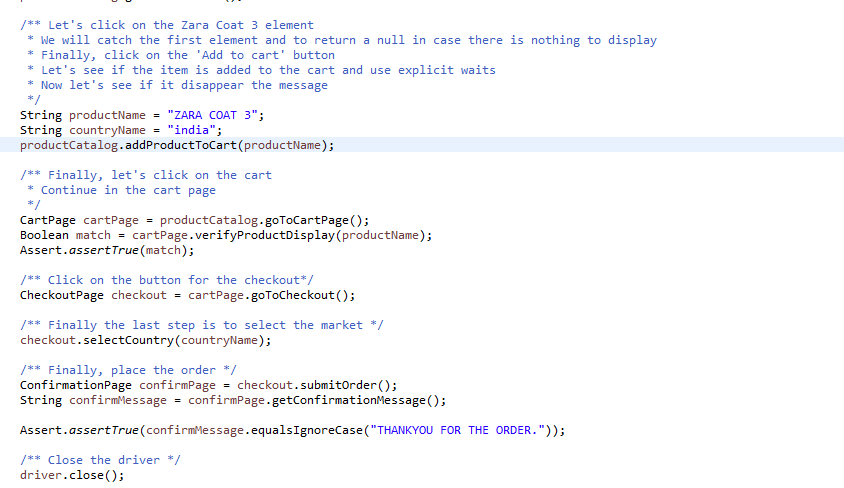
**CheckoutPage**



**ConfirmationPage**



**MAIN**



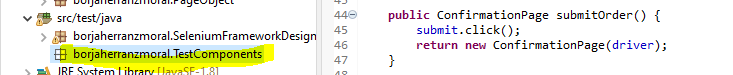
**CAPÍTULO 19: Test Configurations**

**Paso 1: Configurar los métodos de configuración del Base Test**

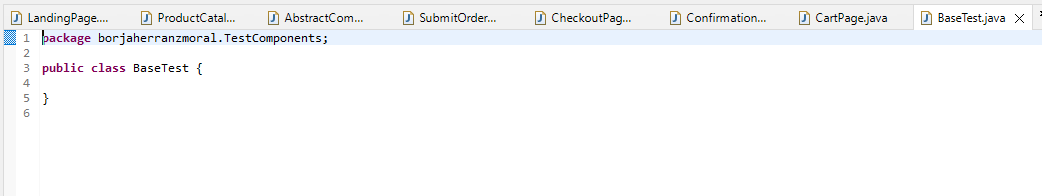
Al igual que hicimos con aquellos elementos comunes, se ha de configurar aquellos elementos que apliquen a todos los casos de prueba.

Es una configuración del browser, seteamos aquellos elementos comunes. Lo meteremos en el **BaseTest**.

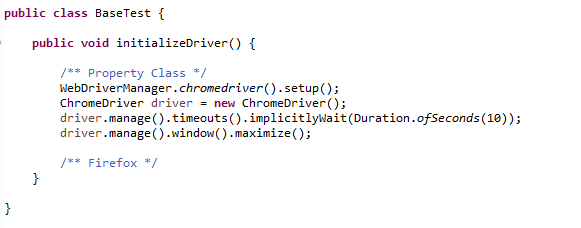
Lo primero es crear el paquete dentro de la carpeta **src/test**



En esta carpeta meteremos la clase **BaseTest**



En esta clase meteremos la configuración para lanzar el browser de Chrome, Firefox,... Inicializar el driver.



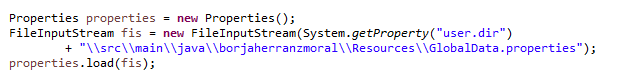
A continuación, para configurar todos los recursos necesarios, en **src/main**, vamos a crear un archivo **File** que contendrán estos recuros.

¡!!OJO!!! Este archivo debe llevar la extensión *.properties* para poder funcionar correctamente.

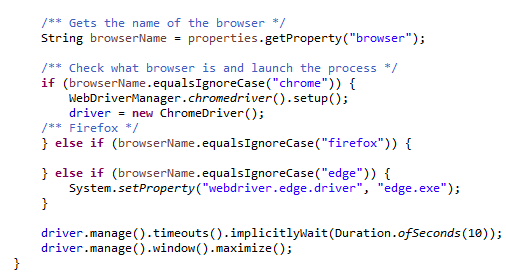


Una vez creada la propiedad, vamos a extenderla en nuestras clases. Para ello, utilizamos la clase **Properties** de Java dentro de nuestro inicializador.

Para cargar el documento se ha de utilizar un FileInputStream para cargar el documento



Finalmente, creamos un bloque de control para determinar que navegador se está utilizando para instanciar el driver.



Se deben cargar las propiedades con el getProperty, que recoge aquellos elementos del archivo .properties.

Por otro lado, se usa el user.dir para navegar hasta la carpeta en la que estamos trabajando.



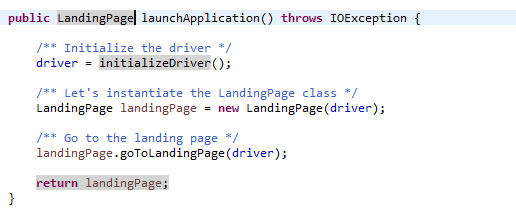
**Paso 2: Inicializar el Driver y usar el BeforeMethod annotation**

Vamos a crear un método para lanzar la aplicación desde el IDE

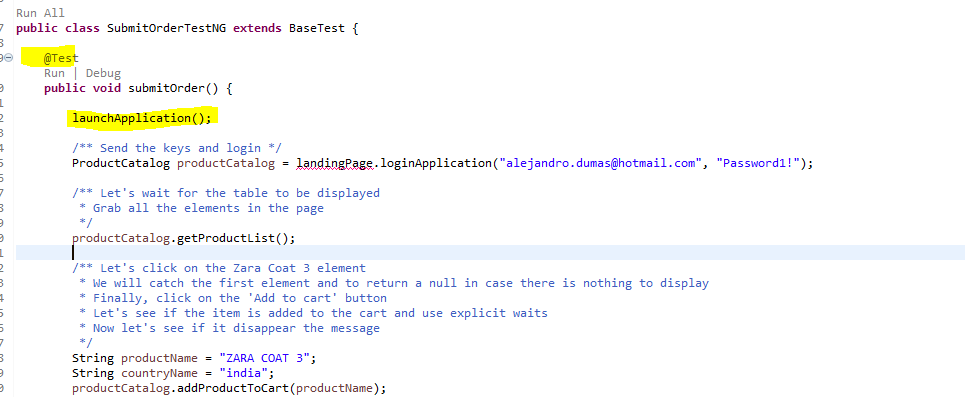
Vamos a meter este método en la clase **BaseTest.java** para poder ejecutarlo en todos los test cases.

Cambiamos el tipo del método initializeDriver() para que retorne el driver que utilizaremos en la aplicación.

Lo siguiente es llegar al LandingPage, así que lo meteremos del método de testing principal aqui:

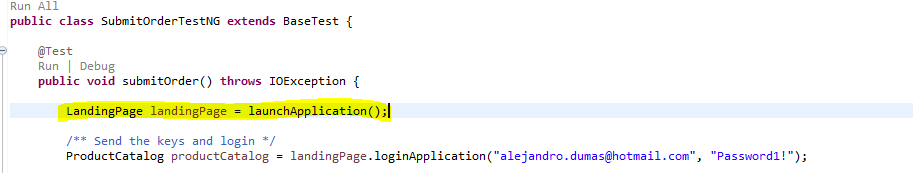


Finalmente, si queremos meterlo en la clase principal, nos pide que el elemento sea estático. Sin embargo, vamos a utilizar las anotaciones de TestNG para poder correrlo sin problema.



Ahora, vamos a la chicha

Como hemos metido el objeto, se lo pasamos como variable en la clase principal

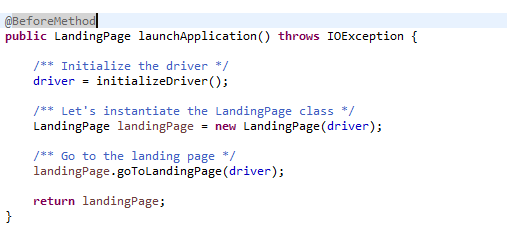


**Paso 3: New ErroValidation test per framework standard**

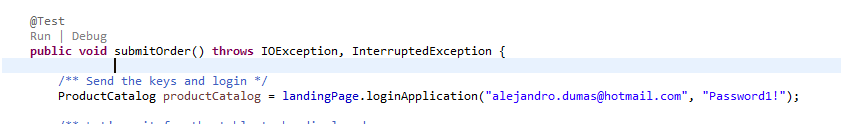
El primer paso que queremos es que el método *launchApplication* se lance cada vez.

Por ello, utilizamos las anotaciones de TestNG, concretamente @BeforeMethod

Funcionará ya que la clase principal está heredando. Es como meter los **prerequisitos** en un test case

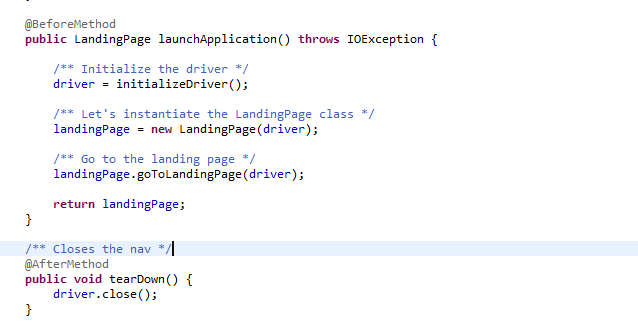


El instanciar la clase LandingPage es local al método de *launchApplication*. Entonces, ¿como somos capaces de lanzarlo? Sacando por fuera la instanciación



De esta forma, quitamos la instanciación ya que lo estamos haciendo en cada paso

Vamos a hacer lo mismo para cerrar el navegador



Con esto ya tendríamos los métodos comunes a todos los tests

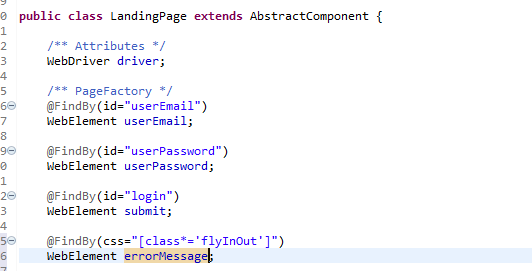
A continuación, vamos a crear otro método para los errores de validación

Esta clase hará pruebas incorrectas, por ejemplo:

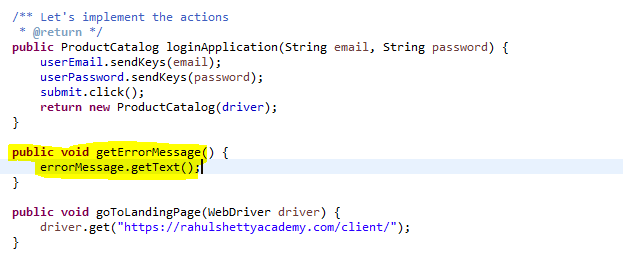
Para buscar el mensaje de error, le mandamos el siguiente



Esto significa que empieza con flyInOut.



Finalmente, decimos que coja el mensaje de error, para luego hacer la aserción



Además, hay que usar una espera explícita para que aparezca

**Paso 4: Implement Test Strategy (tests divided into modules)**

Vamos a crear varios test cases en una misma clase. **No necesitamos una clase por cada test**.

Marcándolos con la anotación @Test podemos hacerlo.

Por ejemplo, podemos tener un **loginValidation()**, **submitErrorValidation()**,...

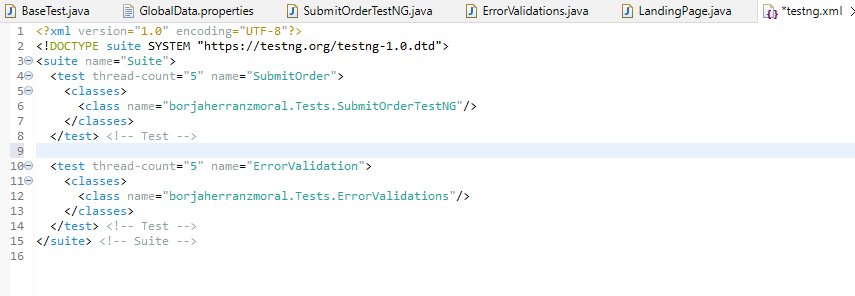
No hace falta crear una clase por ello

Hay que desarrollar la **estrategia para testear**

Por ejemplo, todos los que dependan del **Login**, en la clase para el login,... otros para el submit,... Y en esas clases meter todos los métodos de testeo. De esta manera, limitamos el scope y no es necesario correr todos los test a la vez.

Luego veremos como correrlos en **paralelo. Pero, esto es una tarea de todos los testers y cambia de proyecto a proyecto**.

Después, convertiremos el proyecto a un TestNG project y separaremos los test cases en el testng.xml.



Ya luego lo podremos correr y ver el resultado de las ejecuciones

**Paso 5: Create new test methods with dependency attributes based on the testing strategy**

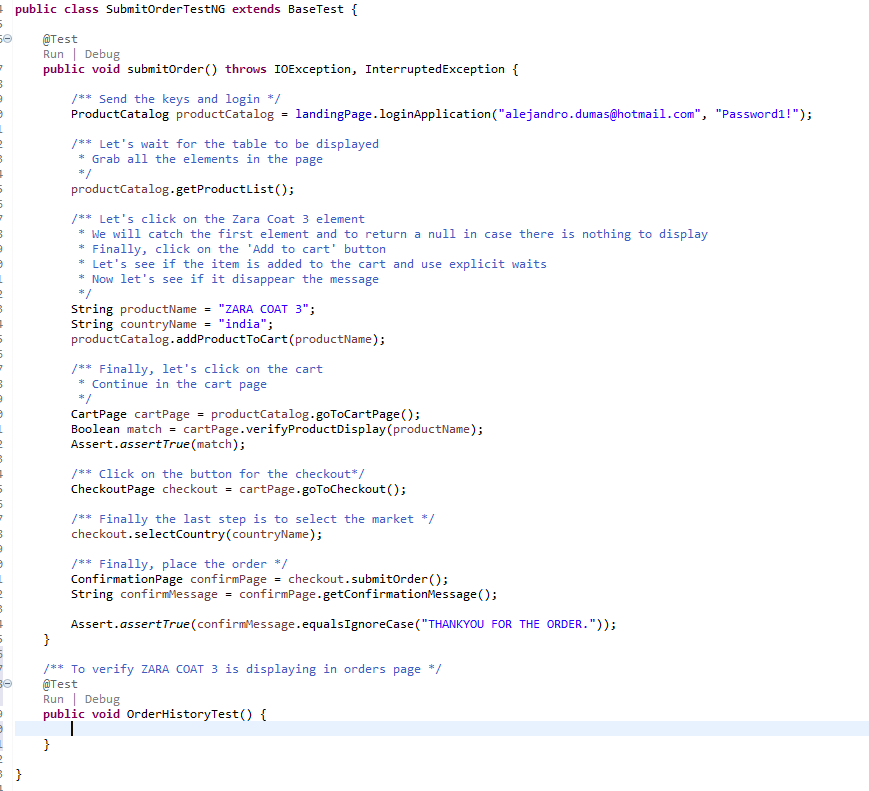
Vamos a probar si el botón aparece en la página para pedir en la página. SI está la orden.

Vamos a utilizar otra función

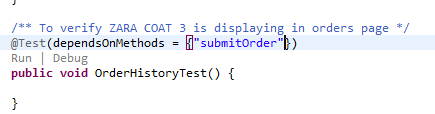
En este caso, vamos a separar la funcionalidad en dos, para no mezclarlo todo en el mismo escenario

Cuanto más pequeñas las funciones, mejor

En este caso, queremos que los tests se realicen de manera secuencial, de tal forma que el primero sea el de **SubmitOrder()** y el segundo el de **OrderHistoryTest()**



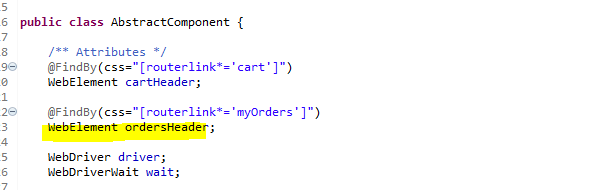
Lo que queremos es que una vez que corra el primer metodo, lancemos el segundo. Se añade así en la anotación

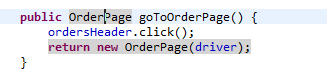


Vamos a crear una nueva página, similar a CartsPage, y meter toda la funcionalidad



Abstract



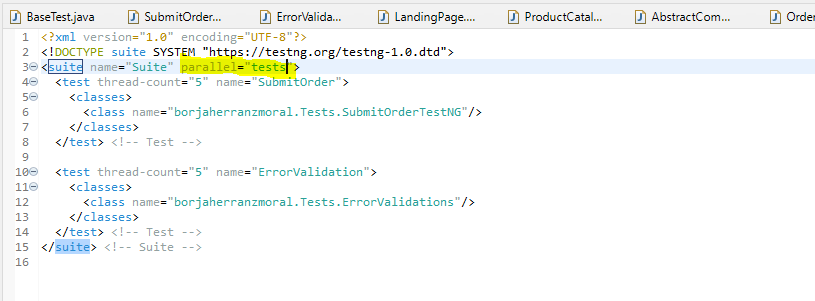


**Paso 6: Correr los tests en paralelo**

Para poder correr los tests en paralelo, necesitamos implementar en el archivo testng.xml las anotaciones adecuadas para hacerlo.

Se debe hacer a nivel de **suite**

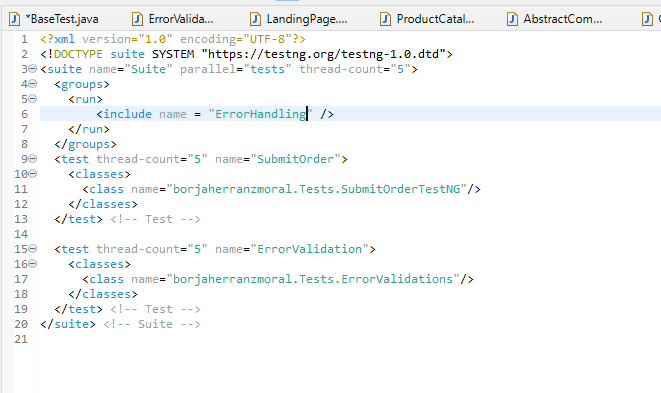
Se mete la anotacion parallel = ‘tests’



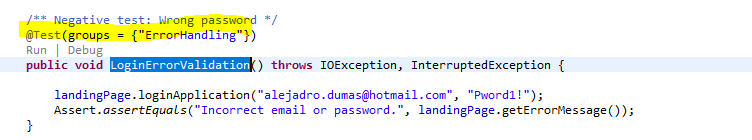
También se puede controlar el número de métodos que corren de forma paralela con el atributo **thread-count**. Puede ir de 5 a 1 método simultáneo.



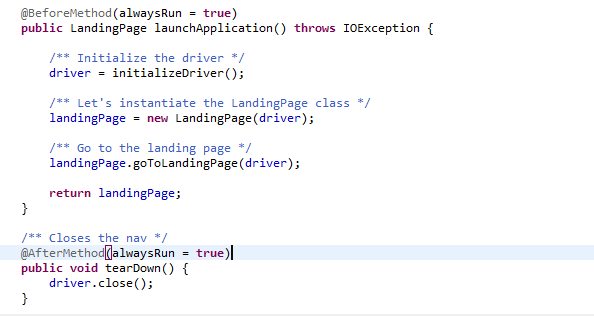
Por último, vamos a agrupar a aquellos tests que queremos lanzar



Igualmente, hemos metido un grupo para sólo incluir en el test case los que funcionan con el ErrorHandling



Finalmente, como vemos que no corre, dando un NullPointerException. Eso es porque en el código corre antes de que se inicialice la página. De esta forma, no encuentra el driver y no funciona nada. Hay que indicar a los métodos @BeforeMethod y @AfterMethod que se inicialicen siempre



**TEST STRATEGY**

**Paso 1: Parametrización en TESTNG**

Esta parte ya la hemos visto cuando TestNG, pero vamos a aplicarlo en nuestro proyecto,

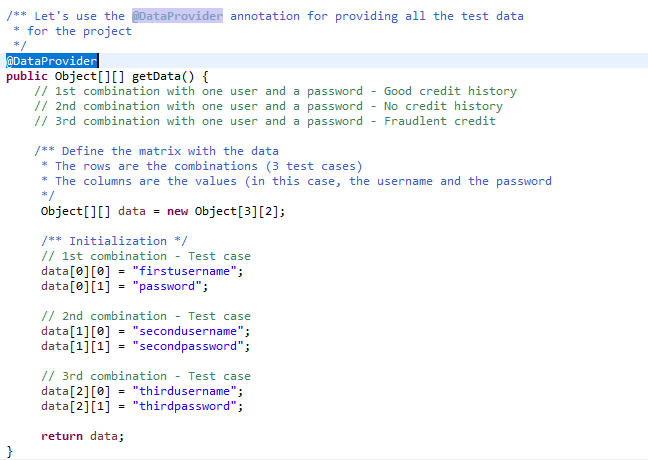
Vamos a usar el Data Provider, junto al JSON. Con ello recogeremos los datos

**¿Qué es Data Provider?**

Nos ayuda a coger los datos, pasándole varios Data Sets.

Para ello, se **usa la anotación @DataProvider**, y se le pasa el objeto que queremos

Por ejemplo, anteriormente:

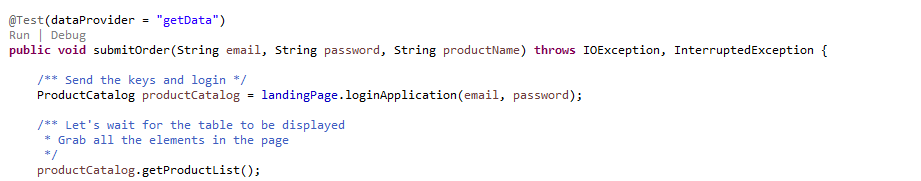


En el caso que queremos utilizar, para usar dos DataSets.

Lo primero es que definimos los datos a utilizar



Finalmente, utilizando TestNG mandamos los valores como parámetros



En este caso, se envía el **email, password, y el nombre del producto**

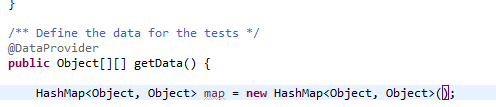
Con esto tenemos parameterizados todos

**Paso 2: Integración del HashMap para enviar datos**

No es correcto tener todos los datos en la clase, porque esto puede cambiar con el tiempo. Es mejor enviar estos valores como un Hashmap.

De esta forma no tenemos que cambiar la dimensión del objeto data.

Para ello, dentro de la clase DataProvider vamos a pasarle este parámetro



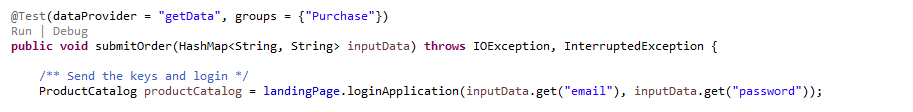
De esta forma, nos queda como:



Finalmente, para que todo cobre coherencia, cambiamos de objeto a String



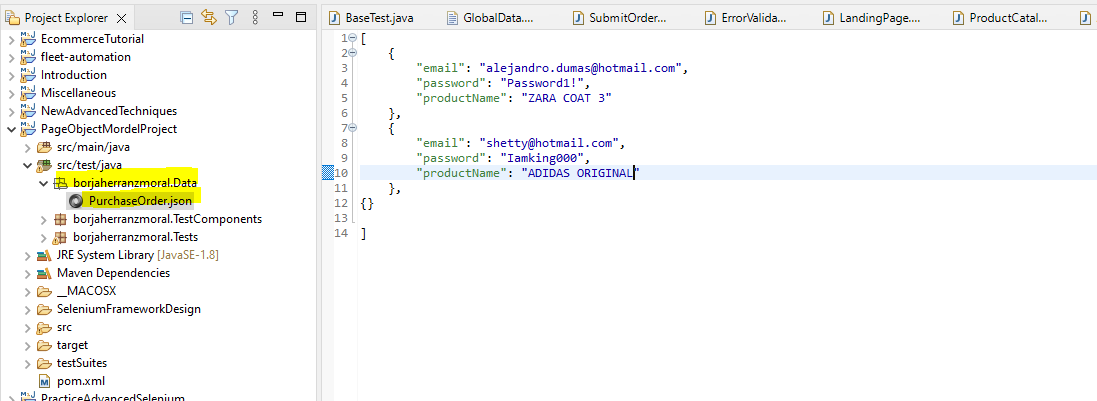
El Hasmap se pasará a la función



**Paso 3: Lectura de archivos JSON y crear la lista de Hashmaps**

Para pasar los archivos como un JSON, lo primero es paquetizar y separar la clase que contendrá los datos.

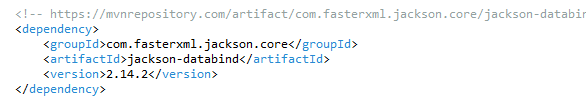
Para ello, creamos un JSON con la estructura que queremos pasar



A continuación, debemos ser capaces de recibir este objeto para poder tratarlo.

Vamos a escanear el JSON en los Hashmap ya creados.

Para escanear el Json hay que importar en el proyecto MAVEN el Jackson



A continuación, en una clase externa, creamos todos los datos que vamos a utilizar, partiendo del JSON que hemos pasado.



Debemos ser capaz de parsear el archivo con el ObjectMapper

Una vez hecho, le pasaremos los datos al fichero que ejecuta las aserciones.

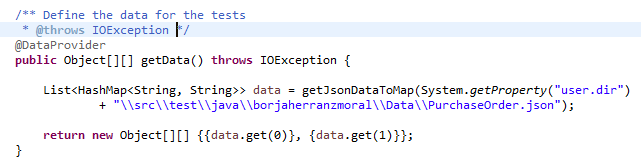
Lo mandamos como List<HashMap<String, String>> ya que es un archivo JSON con keys y values, y tenemos más de un Hashmap (varios objetos)

Sin embargo, para que esto funcione correctamente, el Hindú decide ponerlo todo en la clase de BaseTest. De esta forma se puede heredar del método principal

**BaseTest**



Finalmente lo pasamos como parámetro en el SubmitOrder.java y ya funcionaría

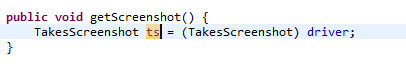


**Paso 4: Tomar capturas de pantalla**

A continuación, vamos a usar la funcionalidad para tomar capturas de pantalla

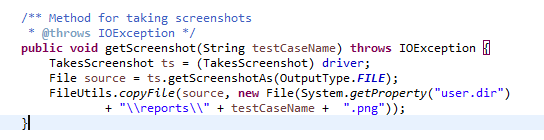


Lo primero vamos a castear el driver para poder coger capturas de pantalla

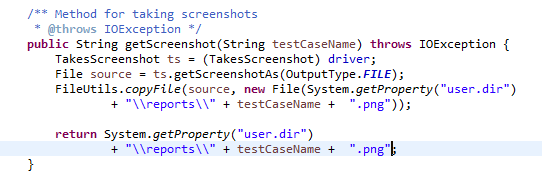


Ahora ya tiene la habilidad de tomar capturas de pantalla

Finalmente, debemos mandar la ruta de destino. Creamos la carpeta **reports** para hacer esto



Finalmente, vamos a mandar esto a ExtendReports, por eso le vamos a pasar el String

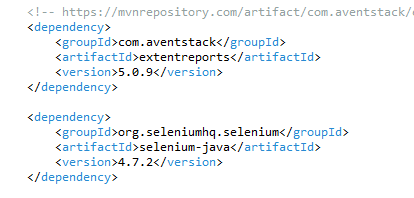


**EXTENT REPORTS**

**Paso 1: ¿Qué es?**

Es una herramienta para sacar reportes con capturas de pantalla, que luego lo podremos linkar a tickets de JIRA

Lo primero es descargarse ExtentReports de Maven Repository



A continuación nos creamos una clase para hacer demos de Selenium

Sería como lo que hacemos con BrowserStack o con Serenity



Creamos la clase así como lo que se utilizaría. En este caso, **ExtentSparkReporter** generará el informe, y **ExtentReports** mandará el report execution

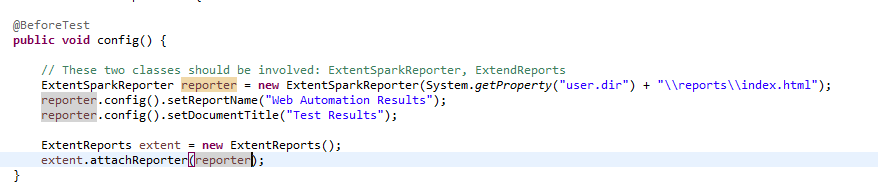
Le ponemos un nombre



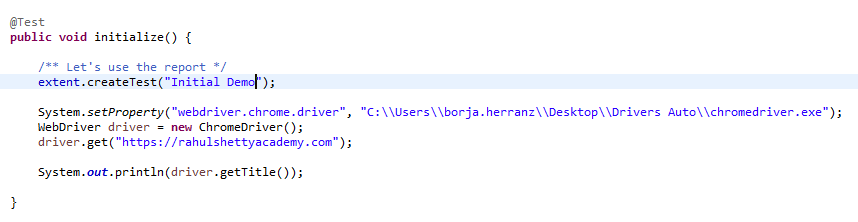
Llamamos el config()

Finalmente, después de nombrarlo, vamos a crear el report instanciando la clase

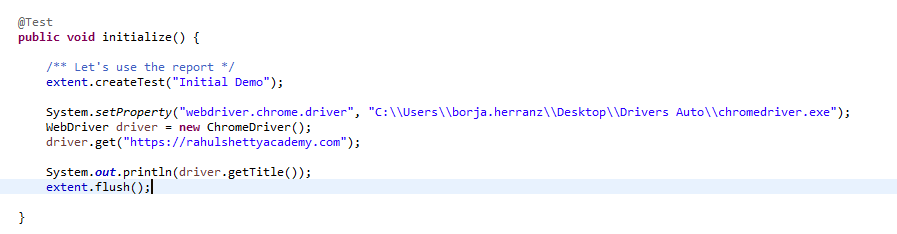
Por último, le pasamos el reporter, que es la configuración, y ya hemos generado el reporter



Finalmente, utilizamos el extent en nuestro test



No lo vamos a utilizar en cada test, pero sí para este caso de prueba

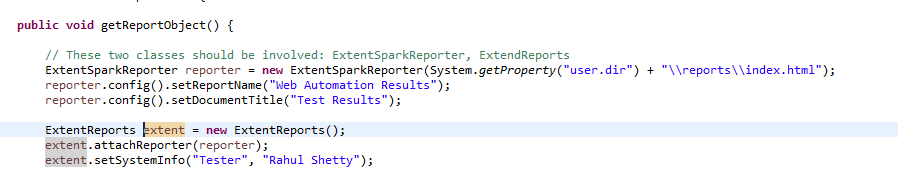


Flush(): permite escribir el test para ti

**Paso 2: Aplicarlo a nuestro proyecto**

Ahora vamos a aplicar lo aprendido a nuestro proyecto para sacar resultados.

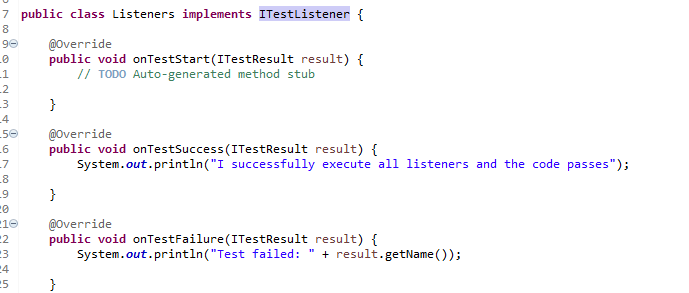
Para utilizar ExtentReports y cuado coger las capturas de pantalla, vamos a usar listeners.



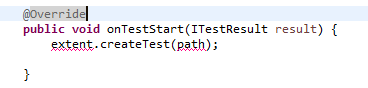
Creamos la clase para los reportes en la parte de recursos

Como no vamos a crear un test cada vez que falle, lo que queremos es que haya Listeners para que los lance cada X. Vamos a extender de la clase Listeners.

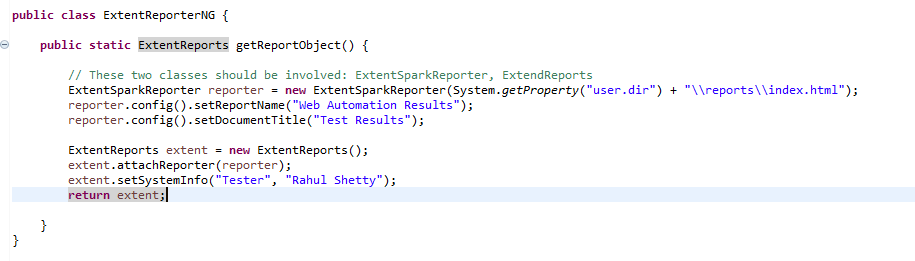
Para ello, creamos una nueva **clase Java** para poder manejarlo.



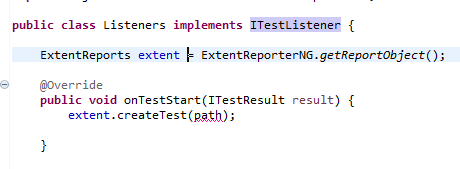
De esta forma, cada vez que lancemos el test, vamos a crear el test.



A continuación, en la clase que implementa los tests, vamos a devolver el objeto



Vamos a poner el método como **static** para poder acceder desde el classname.



Ya con ello, le pasamos el path y ya estaría

Esto es como queda la clase por el momento

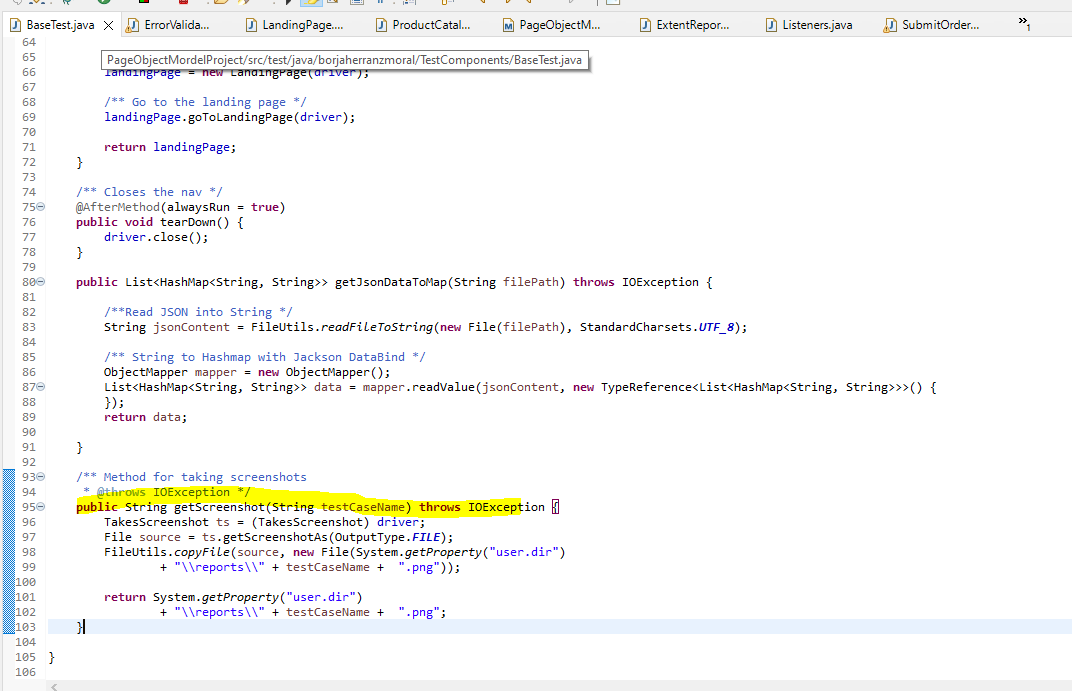


* Por cada test, se crea un nuevo test
* Por cada test pasado, se manda un log de test pasado
* Por cada test failldo, se manda una captura de pantalla y un mensaje con el **result.getThrowable()**

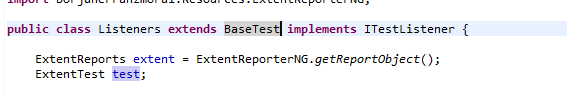
**Paso 3: Adjuntar screenshots a los reportes**

Vamos a pasar los métodos para pasar los screenshots.

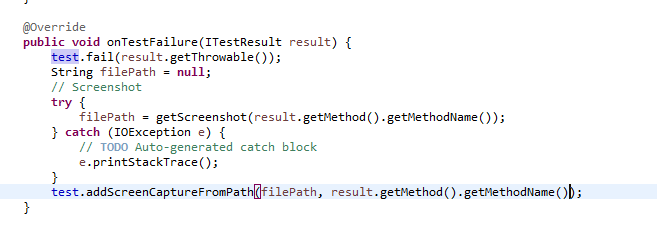
Lo primero, es pasar el método en el BaseTest para poder extenderlo a continuación en nuestra clase de Listeners (tener en cuenta que hemos implementado una interfaz)



En la clase de Listeners



Finalmente, ponemos la captura cuando falla



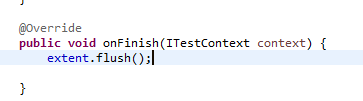
Hay que meterlo en el reporte cuando falle

Por último, hay que indicar en el **testng.xml** de la existencia de estos Listeners, sino no los ejecutará



Sin embargo, esto no acaba aquí. Hay que usar flush() para generar el reporte.

Hay que ponerlo al finalizar el test en el método de ItestListener **onFinish()**



El último problema es meter el Driver en el Listeners. Lo vamos a pasar como argumento del **getScreenshot**. Sólo así se podrá utilizar.

Para ello, con la interfaz ItestResult debemos ser capaces de capturar el **driver**



Hay que pasarle este driver, que es el que verdaderamente sabe la lógica del fallo.