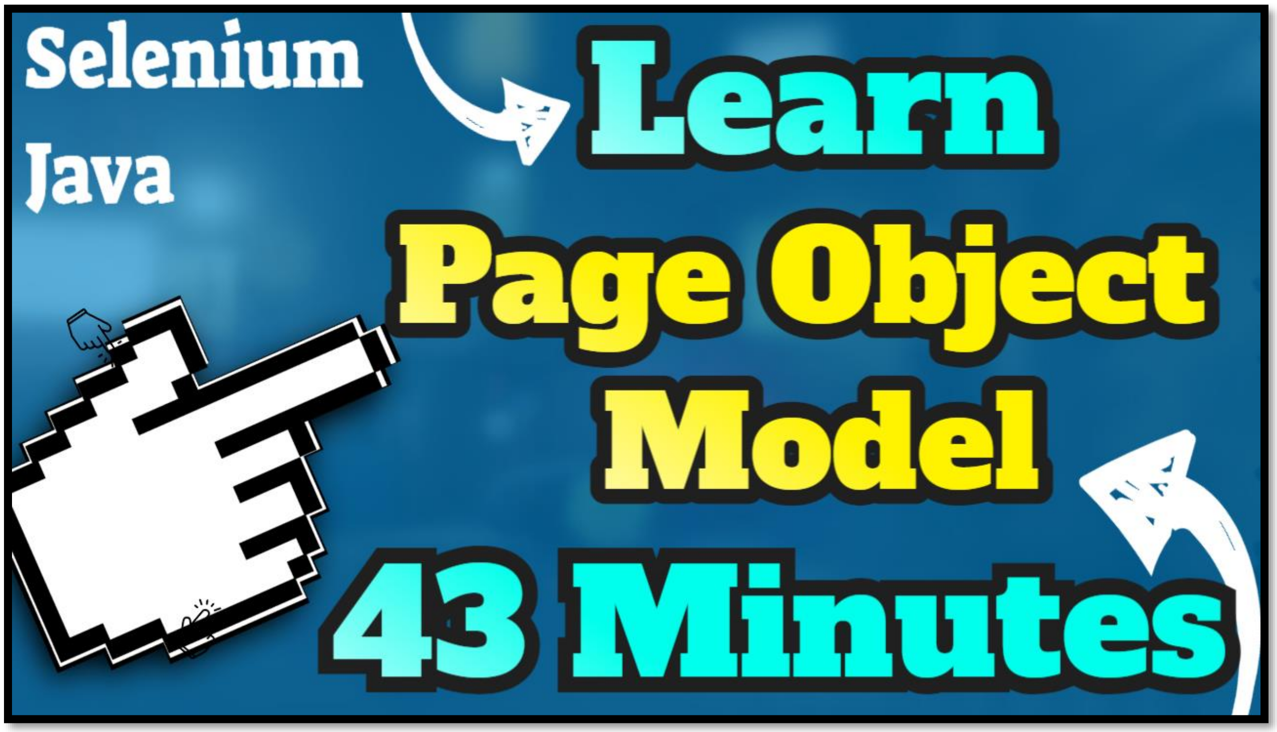
Rex Jones II

Learn Page Object Model For Selenium



Video Playlist https://www.youtube.com/playlist?list=PLfp-cJ6BH8u_CynFLk3yzd8Kl1naC0a2T

Introduction

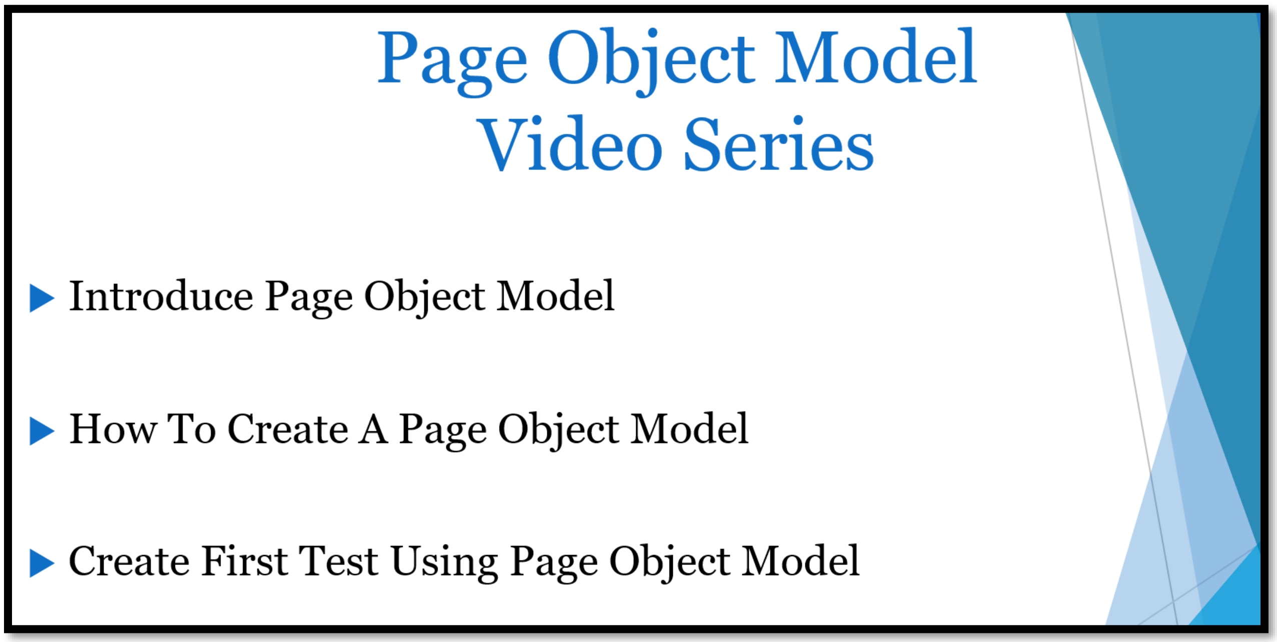
By popular demand, I will cover Page Object Model. Page Object Model is a design pattern that's used as a component in our automation framework. I'm going to show you how I create a Page Object Model step by step from scratch.

Right now, I release videos 3 times a week on Monday, Wednesday, and Friday. In the future, I plan to start releasing videos 5 times a week. If you are interested, subscribe to my YouTube channel then click the bell icon. You can also connect with me on LinkedIn, follow me on Twitter, and Facebook. After this video, I'm going to place the transcript and presentation on GitHub.

Page Object Model allows us to represent the behavior of our application and only update our code in one place.

In this introduction, I will speak about What Is A Page Object Model, Why Page Object Models Are Important, and the Benefits of A Page Object Model. After the intro, I will demo How To Create A Page Object Model, then we are going to Create our First Test Using the Page Object Model.

Rex Jones II

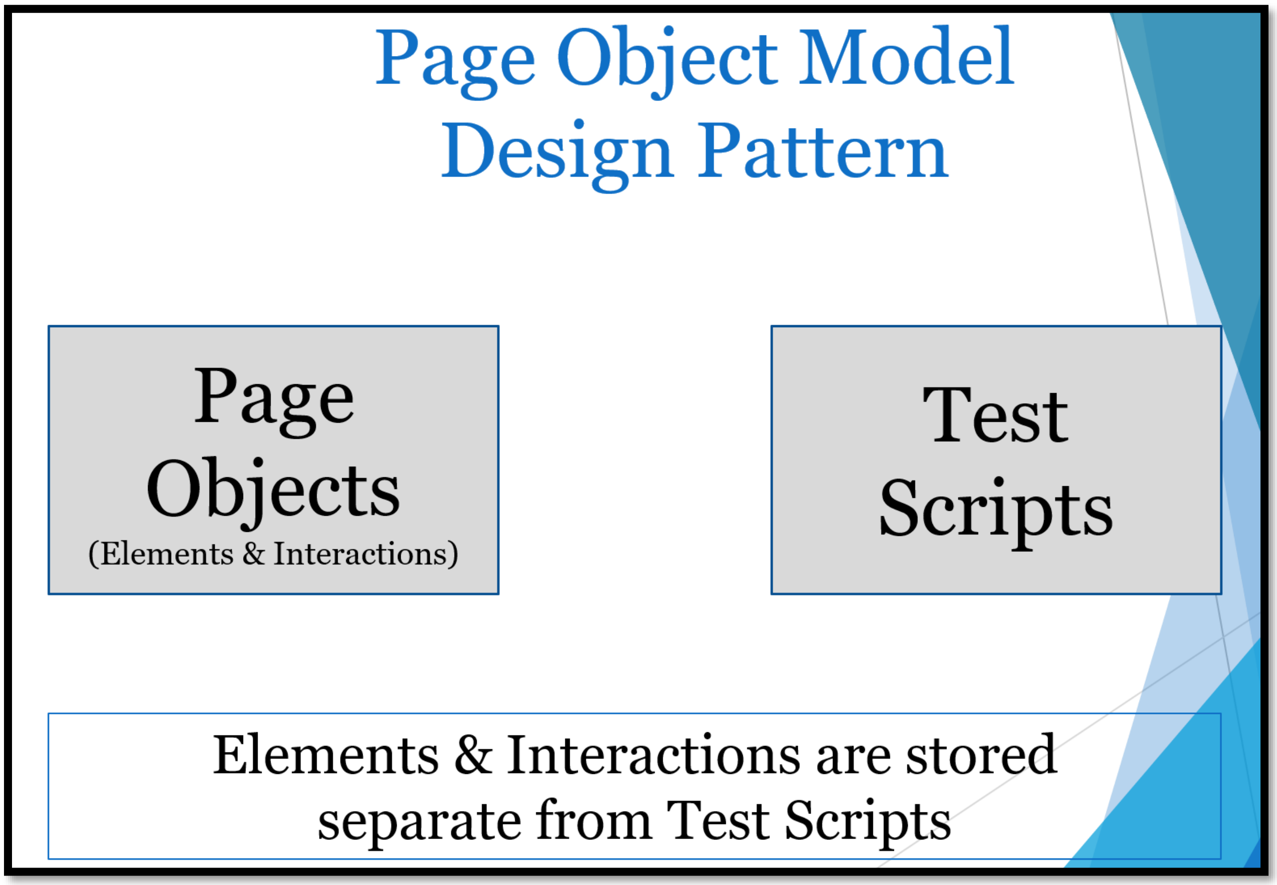


Page Object Model Video Series

- Introduce Page Object Model
- How To Create A Page Object Model
- Create First Test Using Page Object Model

What Is A Page Object Model

Introduction - What is a Page Object Model? A Page Object Model is a popular design pattern with classes that represent each page of an application. In some cases, it will be the entire page and other cases it will be part of a page. The purpose of Page Object Model is to serve as a pattern for the application we are testing. Our elements and interactions are stored separate from the Test Scripts.



Page Object Model Design Pattern

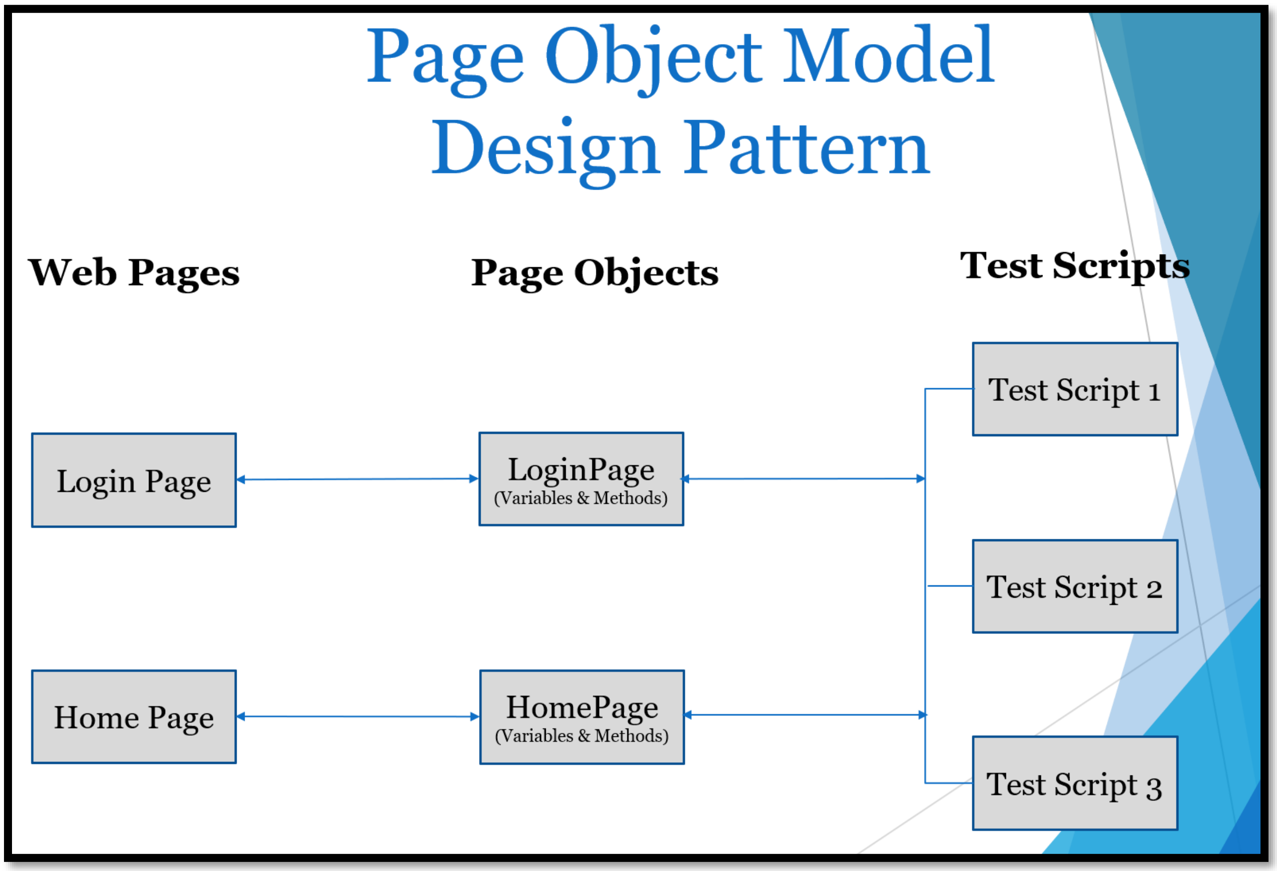
Page Objects
(Elements & Interactions)

Test Scripts

Elements & Interactions are stored separate from Test Scripts

Each web page or section of a web page is represented by a Page Object. In this example, we see the Web Pages are Login Page and Home Page. Both pages have their own Page Object called LoginPage and HomePage.

The Page Object is a class and within that class we have class members. We see the class members are variables and methods. Variables are defined as elements of the page while methods are implemented as interactions with those elements. The Test Scripts reuse a Page Object when it needs to interact with a particular page.



Everything starts by creating a base and the correct page object. Here's the catch, there is more than 1 way for designing our Page Object Model. So, people will do things a little different.

Why it's important – DRY

Why is the Page Object Model important? The Page Object Model is important because it helps with the DRY principle. DRY stands for Don't Repeat Yourself. This principle provides support to us by not repeating the same code. We store each field and method in an object repository. The object repository is designed to help reduce overhead for our test by storing properties and actions in one location. If your test and you notice that you are writing your test and you have the same code over and over then that code is a candidate for the DRY principle.

The DRY principle falls under the OOP's concept which is an acronym for Object-Oriented Programming.

Object-Oriented Programming is a foundation for Page Object Model. We have the opportunity to use all 4 OOP concepts with Page Object Model. The 4 OOP concepts are Encapsulation, Inheritance, Polymorphism, and Abstraction. For example, Encapsulation hides the details from our Test Script. Inheritance allows us to extend a class, and Polymorphism can be used to override a method. Abstraction prevents us from creating an object from the abstract class. If you want to brush up on your knowledge of the OOP's concept, by watching my Playlist which include videos 108 – 121.

Benefits of Page Object Model

The benefits of a Page Object Model is code reusability, code readability, and code maintainability. Code reusability is when we reuse code in multiple locations in our program. That prevents us from rewriting the same code. Code readability means the code is easy to follow. It's easy to follow because the names are descriptive and there are less lines of code. Code maintainability means it will take less time to make a change to our code. Therefore, we can expect to revisit the code again in the future.

That's it for the introduction to Page Object Model. Next, I'm going to show you how I Create A Page Object Model.

When creating a Page Object Model, I prefer to start with a Base Page. The purpose of a Base Page is to have a class with common actions that can be used across different Page Objects. Recall from the introduction, a Page Object is a class with variables and methods. The variables define elements of the page while methods are implemented as interactions with those elements.

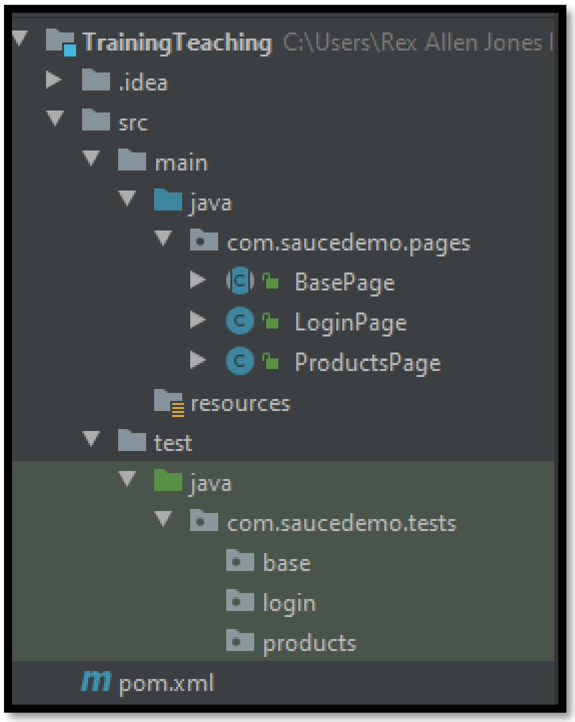
In this video, I will demo step by step, how to create a Base Page and how to create 2 Page Objects. The Page Objects will be LoginPage and ProductsPage. My plan is to explain each concept and not just show you the concepts.

If you are interested in more videos, you can subscribe to my YouTube channel and click the bell icon. You can also follow me on Twitter, connect with me on LinkedIn and Facebook. After this video, I will create a transcript and place the transcript and code on GitHub.

Create Page Object Model

BasePage

Here's the layout of our Page Object Model. We have 2 folders. One for pages and one for tests. The Base Page and Page Objects are placed in the com.saucedemo.pages folder then I placed the Base Test and Test Scripts in the com.saucedemo.tests folder. The url of our application is saucedemo.com.



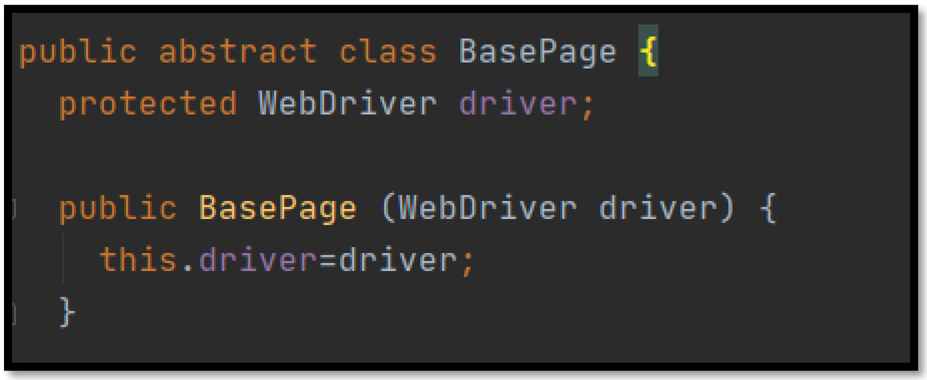
```
TrainingTeaching C:\Users\Rex Allen Jones I
  .idea
  src
    main
      java
        com.saucedemo.pages
          BasePage
          LoginPage
          ProductsPage
      resources
    test
      java
        com.saucedemo.tests
          base
          login
          products
  pom.xml
```

Let's start the Page Object Model with an abstraction BasePage class: public abstract class BasePage { }. Abstract classes cannot be instantiated. Therefore, I am making the BasePage abstract so an object is not created from this class. With Inheritance, the role of this BasePage is to operate as a parent class and each Page class operates as the child class.

Since the BasePage is a parent class, it should contain class members that's also available on other pages in our application. Some variables that's available on other pages are page url, page title, and an object that waits for an element. When it comes to the driver, protected WebDriver driver; we can take more than 1 approach. At least 3 approaches. One approach; we can take is inheriting WebDriver throughout all of our Page classes: from the beginning of our Page class to the end of our Page class then instantiate the object in our Base Test.

The 2nd approach is to declare WebDriver as public static and make sure the driver always retains the last reference then add an if statement to check if the driver is null. The first 2 approaches is better for the DRY principle. However, in this session, I am going with a different approach and create a constructor then pass in WebDriver as an argument.

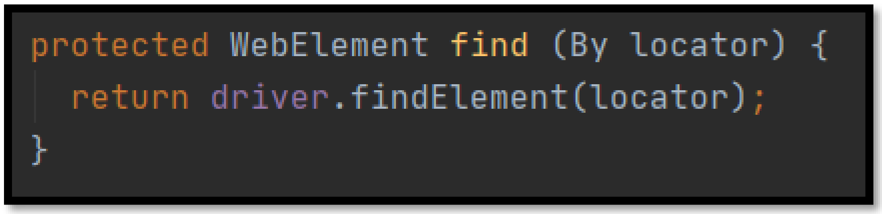
```
public BasePage(WebDriver driver) {
  this.driver = driver;

}
```



```
public abstract class BasePage {
  protected WebDriver driver;

  public BasePage (WebDriver driver) {
    this.driver=driver;
  }
```

The driver will be used to access the browser and locate elements. findElement is the Selenium method we use to locate elements so let's write,
protected WebElement find(By locator) {
  return driver.findElement(locator);

}



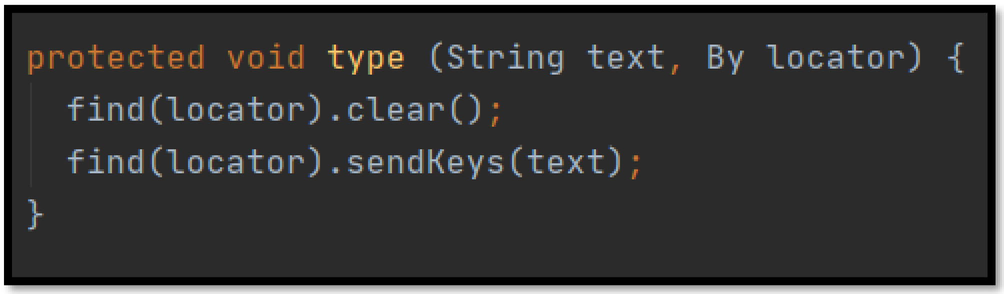
```
protected WebElement find (By locator) {
  return driver.findElement(locator);
}
```

This will be the only place in my Page Object Model that you will see driver.findElement. I will never write it again. We can reuse this find method in our BasePage and all of our Page classes.

Typing data and clicking an element are more generic methods that is common to all pages in our class. So, let's write

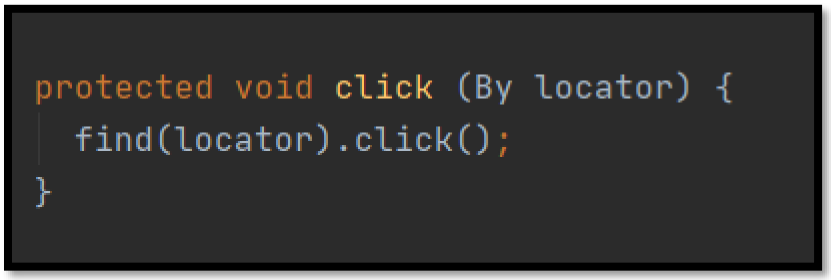
protected void type (String text, By locator) { }
We pass in String for the data our Test Script will type. The By locator is a parameter for locating an element. Look how the previous method reduces code duplication. I will not rewrite driver.findElement but only write find(locator).clear(). If there is data in a field then clear the data and write find(locator).sendKeys(text). That's smooth and that's all we write to type data. I will use this same process for the next method.

Rex
Jones II



```
protected void type (String text, By locator) {
  find(locator).clear();
  find(locator).sendKeys(text);
}
```

The next method is protected void click(By locator) {
  find(locator).click();
}

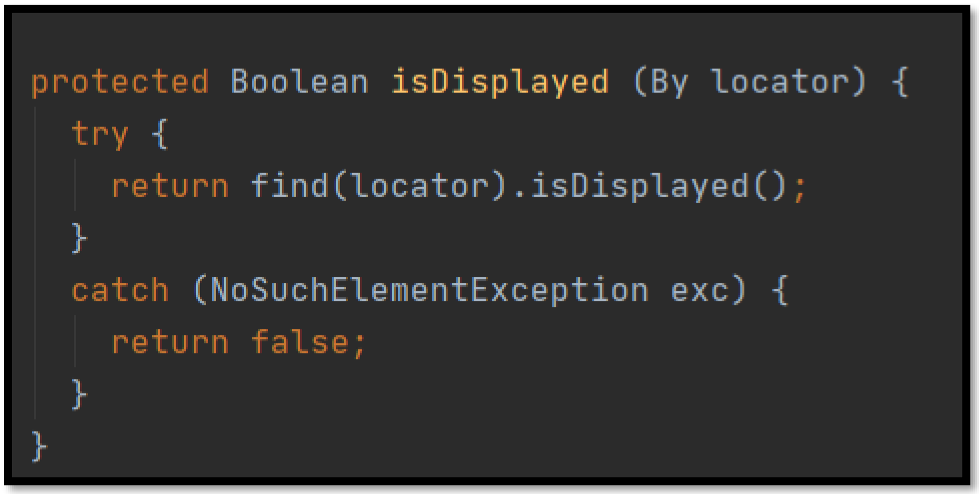


```
protected void click (By locator) {
  find(locator).click();
}
```

That's all we need to click an element. This same click method will click a button, click a link, and click any other element that can be clicked on a web page. Let's write 1 more method.

How about we check if an element is displayed?
protected Boolean isDisplayed (By locator) {
  return find(locator).isDisplayed()

} Know what, let's wrap this return statement in a try/catch block.
try {
  return find(locator).isDisplayed();
}
catch (NoSuchElementException exc) {
  return false;
}



```
protected Boolean isDisplayed (By locator) {
  try {
    return find(locator).isDisplayed();
  }
  catch (NoSuchElementException exc) {
    return false;
  }
}
```

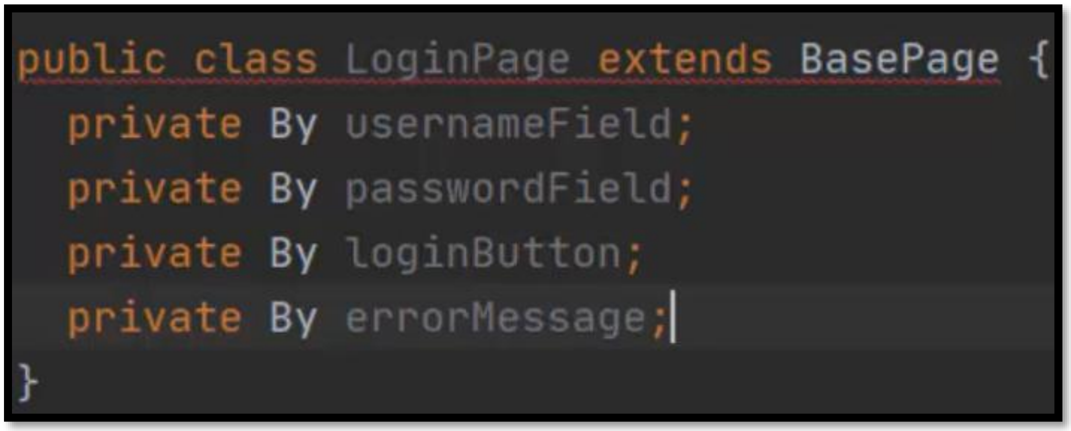
If an element is not on a page then our code will catch the NoSuchElementException and return false. I added Boolean as the return type because isDisplayed returns true or false. These other 2 methods have void as the return type because they do not return a value. This find method has WebElement as the return type because findElement returns a WebElement.

The objective of this BasePage is to allow all of the Page classes to inherit the base variables and methods. Next is the Login Page.

LoginPage

For convention, it is best to have a class name that represents a page in our application. That's why I have LoginPage as the class name. In the application, we have some elements on the Login page: a username field, password field, login button, and error message. Our page class will have properties. We can also say fields to represent these 4 elements. The LoginPage extends our BasePage and thanks to inheritance we see the methods created in the BasePage: isDisplayed, click, type, and find.

Those 4 methods will perform actions on the private By usernameField;, private By passwordField;, private By loginButton;, and private By errorMessage;

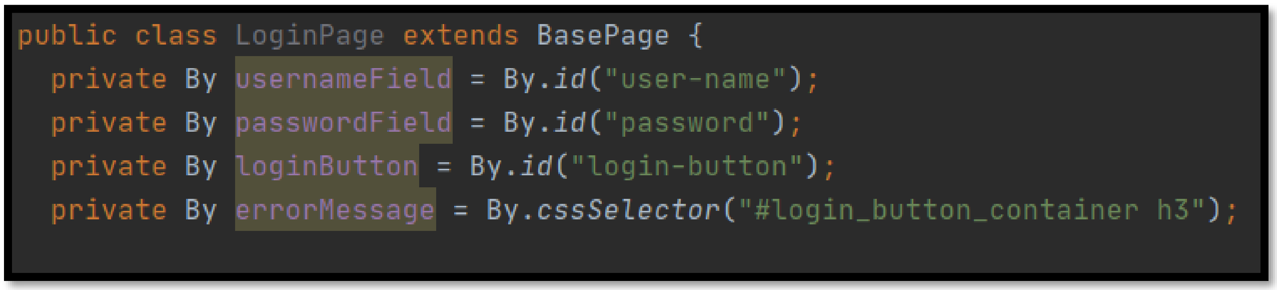


```
public class LoginPage extends BasePage {
  private By usernameField;
  private By passwordField;
  private By loginButton;
  private By errorMessage;
}
```

With Encapsulation, the fields are private so only methods in this LoginPage class have access to the fields. Remember, By is a class for locating elements and the values are going to be locators for each element.

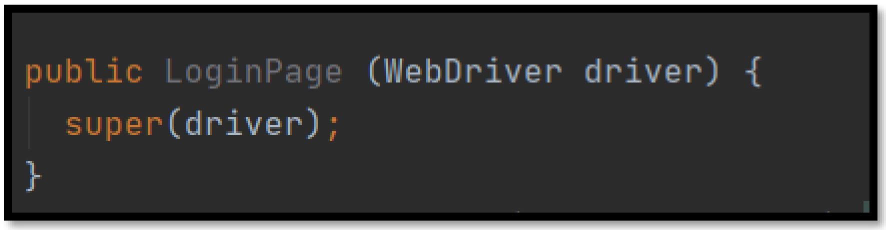
Let's go back to the application and get their values. Right click, Inspect username. We see the form tag has username, password, and Login button. All of the elements have an id attribute. Username has a value of user-name. Okay – cool. Password has a value of password. The Login button has a value of login-button. Inspect the error message and it does not contain an id attribute but the closest id is located in a parent tag. Let's use CSS Selector for the error message. CTRL + F and write #login_button_container h3. Bingo 1 of 1, Copy the value. You can watch my Playlist Customize XPath & CSS Values if you would like to know how I created this value.

The value for usernameField = By. and we see the Selenium locators: className, cssSelector, id, linkText, name, partialLinkText, tagName, and xpath. We are going to use id("user-name"); passwordField = By.id("password"); loginButton = By.className("login-button"); and errorMessage = By.cssSelector paste ("login_button_container h3")



```
public class LoginPage extends BasePage {
  private By usernameField = By.id("user-name");
  private By passwordField = By.id("password");
  private By loginButton = By.id("login-button");
  private By errorMessage = By.cssSelector("#login_button_container h3");
```

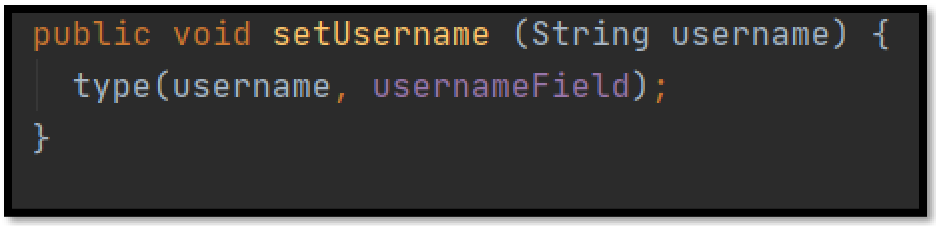
The constructor is public LoginPage(WebDriver driver) {
super(driver); We are calling super then passing in the driver to load the constructor from our BasePage. The constructor is created to bring in the WebDriver so the driver can access the browser.



```
public LoginPage (WebDriver driver) {
  super(driver);
}
```

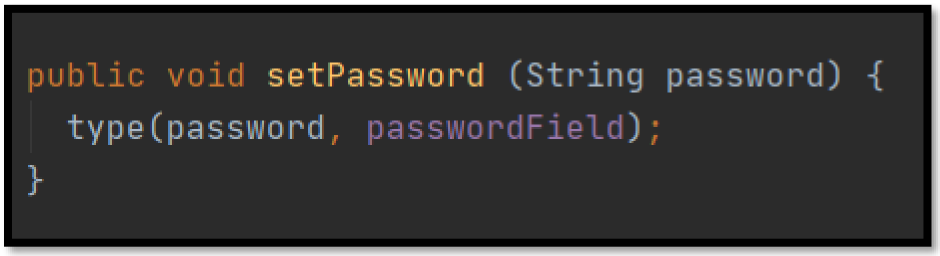
We have our constructor and fields. Now, let's work on the methods which will perform actions on these elements. There are 4 types of methods we can use in our Page Object Model. We have getter and setter methods that comes with encapsulation. Also, we have transition methods and convenience methods. The convenience methods can also be called helper methods. The getter method gets the state of our application and setter methods helps us to create a test that interacts with our application. A transition method is important when our application changes to a different page. It may change when we click a button or click a link. A convenience method is created when combining more than 1 method into a single method. I'm going to demo all 4.

First, we are going to use the setter method by setting the username. public void setUsername (String username) {
type the (username, in the usernameField) type came from the BasePage, username is the parameter and usernameField is the property variable. That's all we write to type for the username. Finish – Done Deal



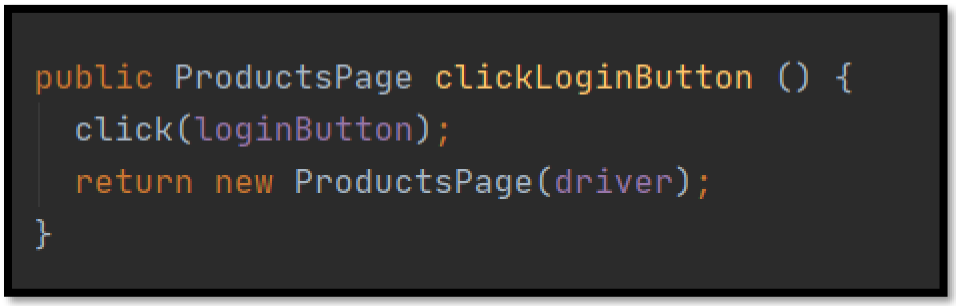
```
public void setUsername (String username) {
  type(username, usernameField);
}
```

} Same with Password. We write public void setPassword(String password) {
type(password, in the passwordField)



```
public void setPassword (String password) {
  type(password, passwordField);
}
```

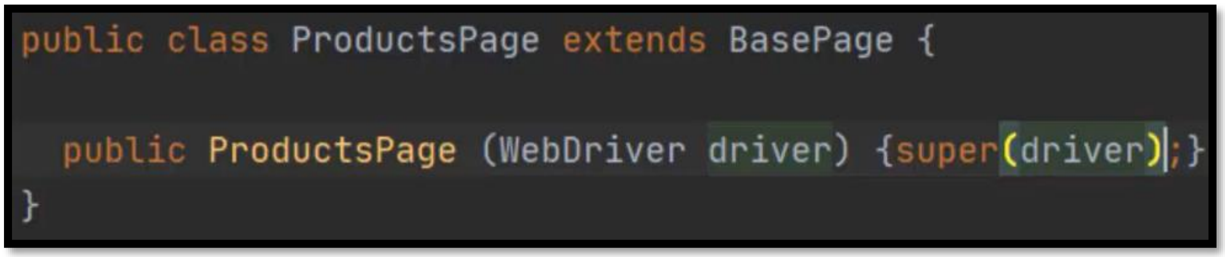
} Next is the transition method public ProductsPage clickLoginButton { }. Do you see why it's called a transition method? We are going to set the username, set the password, and click the Login button then the LoginPage transitions to the ProductsPage. We click(loginButton) / return new ProductsPage pass in the (driver);.



```
public ProductsPage clickLoginButton () {
  click(loginButton);
  return new ProductsPage(driver);
}
```

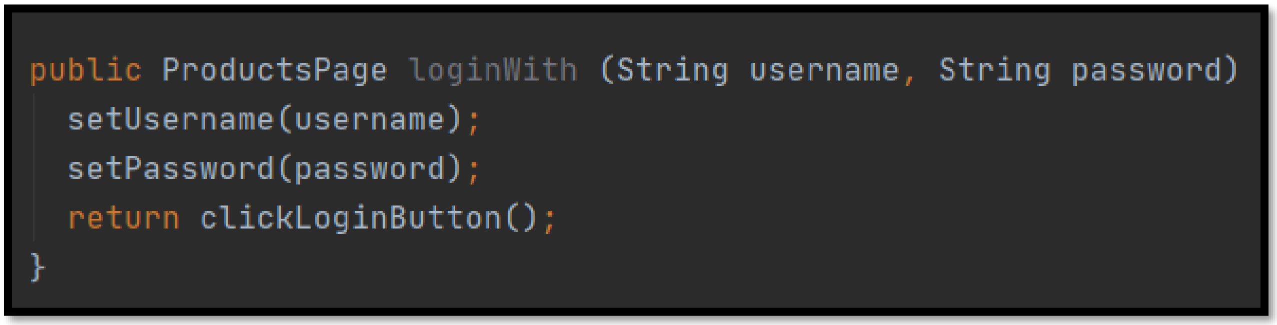
In this scenario, we return the page we transition to and not use the keyword void. An error shows up because we have a constructor that's needed in the ProductsPage.

ProductsPage extends BasePage { }
public ProductsPage (WebDriver driver) {super(driver);} Save. Go back to LoginPage and now the error goes away.



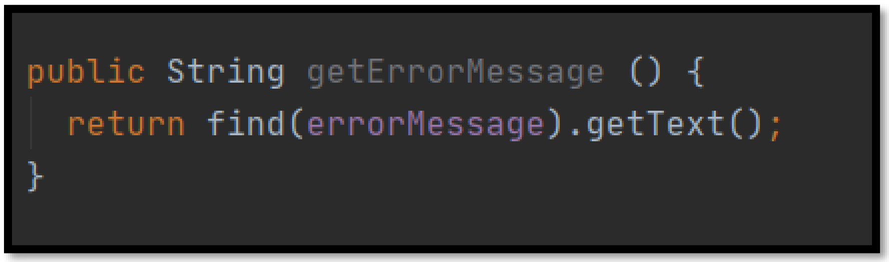
```
public class ProductsPage extends BasePage {

  public ProductsPage (WebDriver driver) {super(driver);}
}
```

Let's create a convenience method so our test also has an opportunity to call one method for logging into the application: setUsername, setPassword, and clickLoginButton. This method will be
public ProductsPage loginWith (String username, String password) { }
  setUsername(username);
  setPassword(password); then
  return clickLoginButton();



```
public ProductsPage loginWith (String username, String password)
  setUsername(username);
  setPassword(password);
  return clickLoginButton();
}
```

Someone may wonder, why do we have an individual method for username, password and clicking the LoginButton plus a convenience method? That convenience method and 3 individual methods defeat the purpose of code duplication. You are right but this is an exception. It's good to have an individual method because your test may include a negative test. For example, you might want to enter a username without a password, click the Login button then confirm the application returns an error message. Let's add a getter method for the error message.
public String getErrorMessage() { }
  return find(errorMessage).getText();



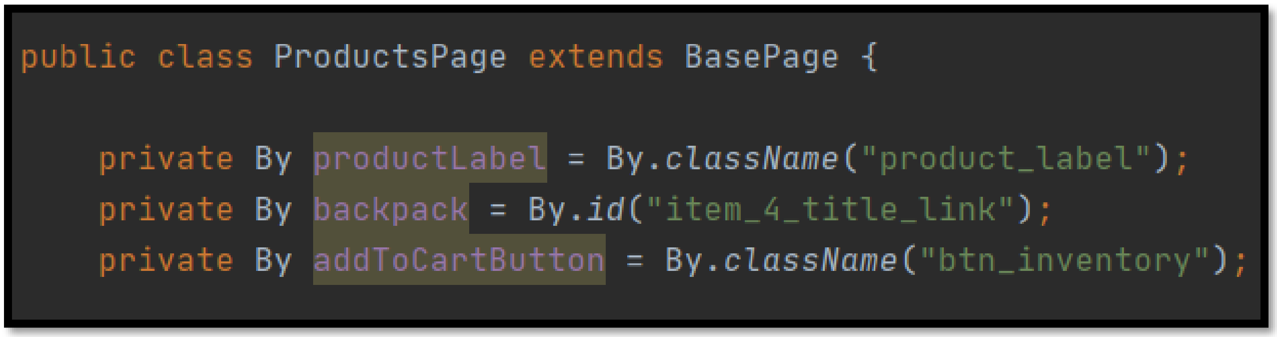
```
public String getErrorMessage () {
  return find(errorMessage).getText();
}
```

We have String return type because the method getText returns a String. We are finished with the Page Object for LoginPage. Now, let’s create a Page Object for the ProductsPage.

ProductsPage

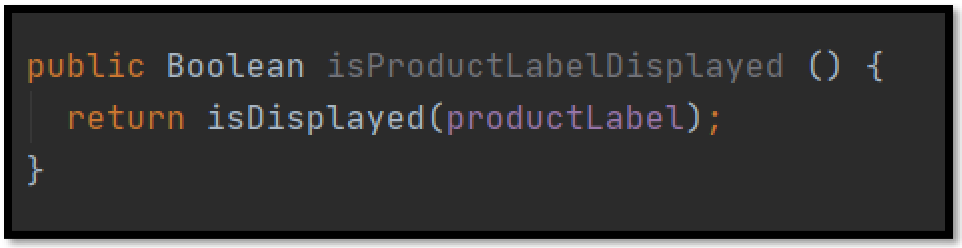
Go back to Swag Labs and inspect the Products label. Class is product_label. Also inspect the backpack title and we are going to use id item_4_title_link. Click the backpack then inspect the Add To Cart button. Let’s test this class value using CSS Selector and write .btn_inventory. Bingo. Okay, we got our values. Let’s go to the ProductsPage class.

We already have our constructor. So, let’s add those 3 properties and their values.
private By productLabel = By.className(“product_label”); / private By backpack = I think it’s By.id(“item_4_title_link”) / private By addToCartButton = By.className(“btn_inventory”);



```
public class ProductsPage extends BasePage {

    private By productLabel = By.className("product_label");
    private By backpack = By.id("item_4_title_link");
    private By addToCartButton = By.className("btn_inventory");
```

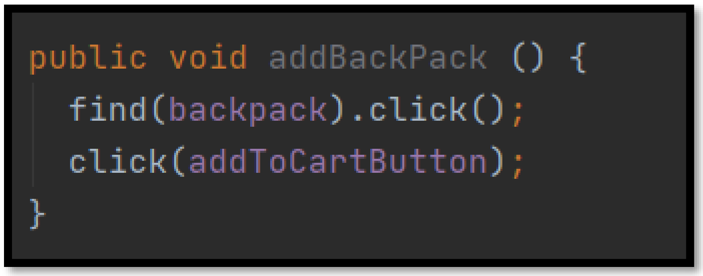
The first method is to check whether the Products label is displayed.
public Boolean isProductLabelDisplayed() { }
return isDisplayed(productLabel);



```
public Boolean isProductLabelDisplayed () {
  return isDisplayed(productLabel);
}
```

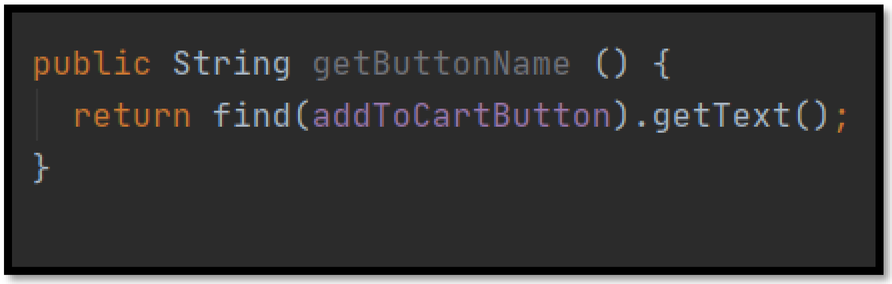
Next is add backpack. public void addBackPack () {
First, we want to find the (backpack) then click(); the backpack. / After clicking the backpack, we click the (addToCartButton);

}



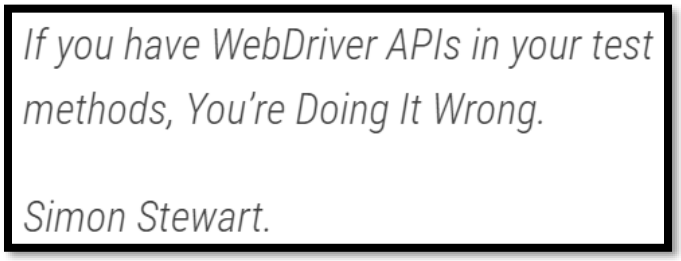
```
public void addBackPack () {
  find(backpack).click();
  click(addToCartButton);
}
```

We are going to also going to get the button name after adding backpack to the cart.
public String getButtonName () { }
  return find(addToCartButton).getText();



```
public String getButtonName () {
  return find(addToCartButton).getText();
}
```

I have a quote from Simon Stewart, the creator of Selenium WebDriver, “If you have WebDriver APIs in your test methods, You’re Doing It Wrong”.

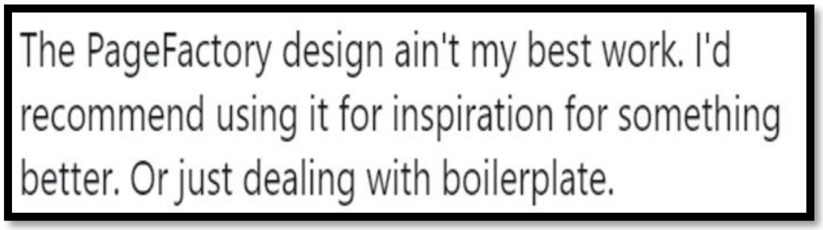


If you have WebDriver APIs in your test methods, You’re Doing It Wrong.

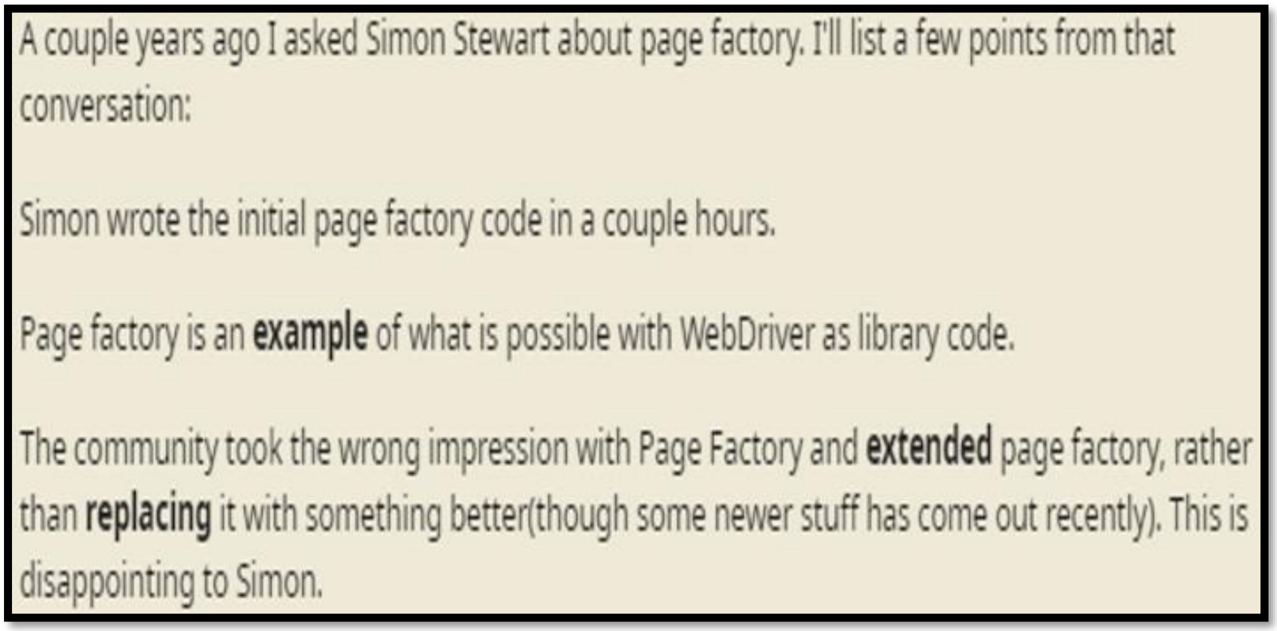
Simon Stewart.

The WebDriver API commands are methods like click, clear, getText, and isDisplayed. These commands belong in our Page Object class and not in our Test Scripts. However, when it comes to assertions, I prefer to include them in my Test Scripts. There is not a rule that say assertions belong in our Page Object class or Test Scripts but I think it’s best to keep them evenhanded neutral. Here’s the difference, in our Page Object, the assertion will have 1 outcome: either pass or fail. But, in our Test Script, the assertion has balance so we can use it for a negative test and a positive test.

I have another quote. You probably noticed I did not speak about PageFactory. That's because Simon Stewart who is also the creator of PageFactory said PageFactory is not his best work.


The PageFactory design ain't my best work. I'd recommend using it for inspiration for something better. Or just dealing with boilerplate.

In a different message, he said the community took the wrong impression with PageFactory.


A couple years ago I asked Simon Stewart about page factory. I'll list a few points from that conversation:

Simon wrote the initial page factory code in a couple hours.

Page factory is an **example** of what is possible with WebDriver as library code.

The community took the wrong impression with Page Factory and **extended** page factory, rather than **replacing** it with something better(though some newer stuff has come out recently). This is disappointing to Simon.

If you want to watch the Selenium Conference from 2017, he also said some more things about PageFactory.

Next, we are going to create a BaseTest then execute our first test using the Page Object Model and I'll see you in the next session.

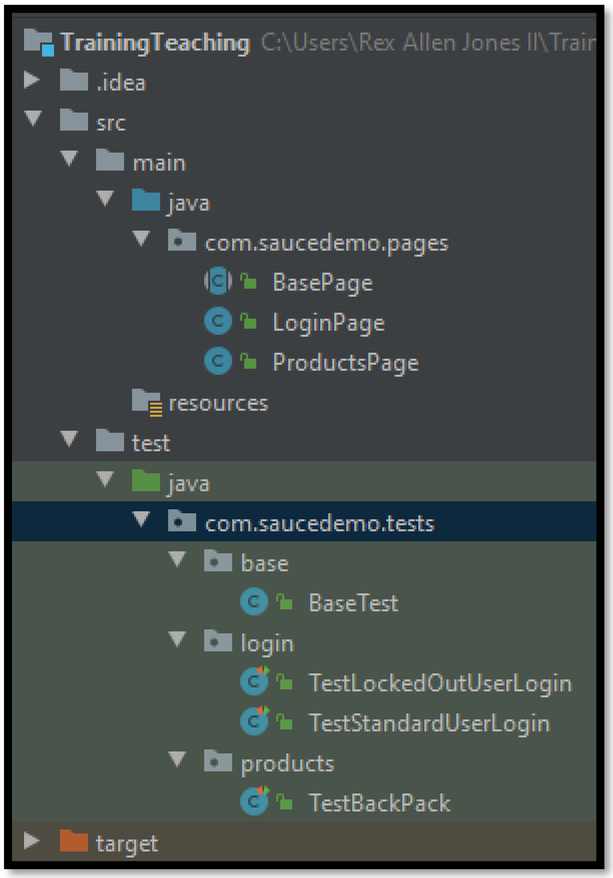
Create Test Using Page Object Model

BaseTest

We create a test using the Page Object Model by starting with a BaseTest. One of the reasons for creating a BaseTest is to set up WebDriver and to tear down WebDriver.

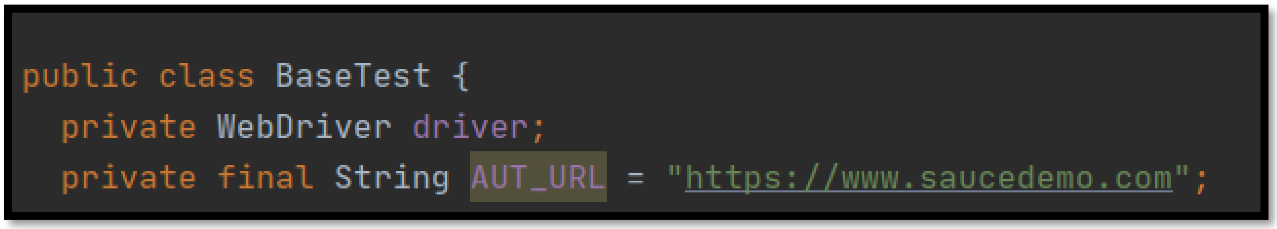
Here's the BaseTest and 3 Test Scripts. 2 of the Test Scripts is for the Login functionality and 1 Test Script is for the Products functionality.

Rex Jones II



```
TrainingTeaching C:\Users\Rex Allen Jones II\Trair
.idea
src
  main
    java
      com.saucedemo.pages
        BasePage
        LoginPage
        ProductsPage
    resources
  test
    java
      com.saucedemo.tests
        base
          BaseTest
        login
          TestLockedOutUserLogin
          TestStandardUserLogin
        products
          TestBackPack
target
```

For the BaseTest, the variables are private WebDriver driver; and the url is private final String Application Under Test AUT_URL = “https://www.saucedemo.com”;

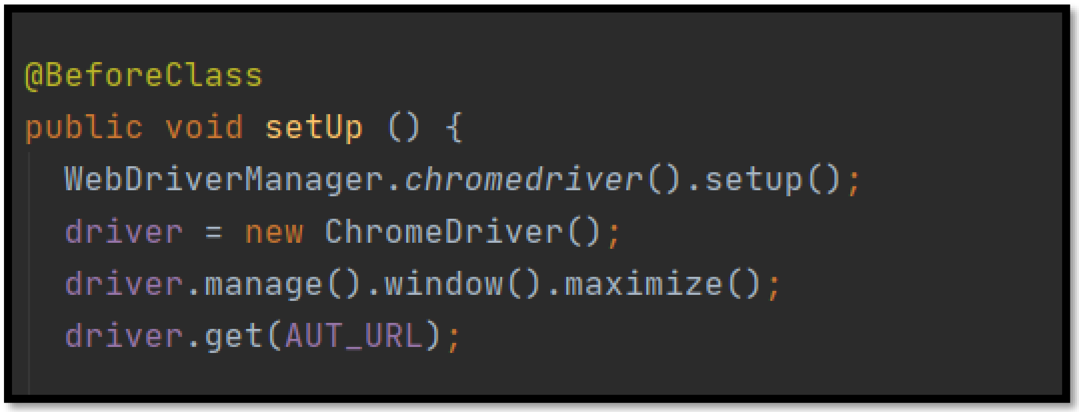


```java
public class BaseTest {
  private WebDriver driver;
  private final String AUT_URL = "https://www.saucedemo.com";
```

Some fields in our Page Object Model may include final and may not include final. It depends on how you want to create your design pattern. Writing final is a way of making sure the value does not change. In this case, the value for AUT_URL cannot be changed.

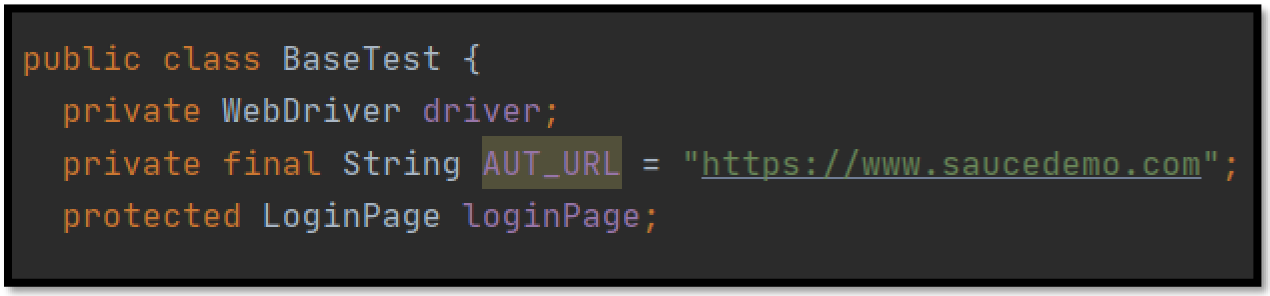
Let’s start with the annotations for TestNG. @BeforeClass / public void setup () { }
We can use System.setProperty to set the arguments for webdriver and a path to the executable file. However, I’m going to write WebDriverManager.chromedriver().setup(). You can watch video 142 if you want more info about WebDriverManager. It’s a library for automating the management of our drivers.

Initialize the driver by writing driver = new ChromeDriver (); / Maximize the window driver.manage().window().maximize() / load our application by writing driver.get(AUT_URL);



```
@BeforeClass
public void setUp () {
  WebDriverManager.chromedriver().setup();
  driver = new ChromeDriver();
  driver.manage().window().maximize();
  driver.get(AUT_URL);
```

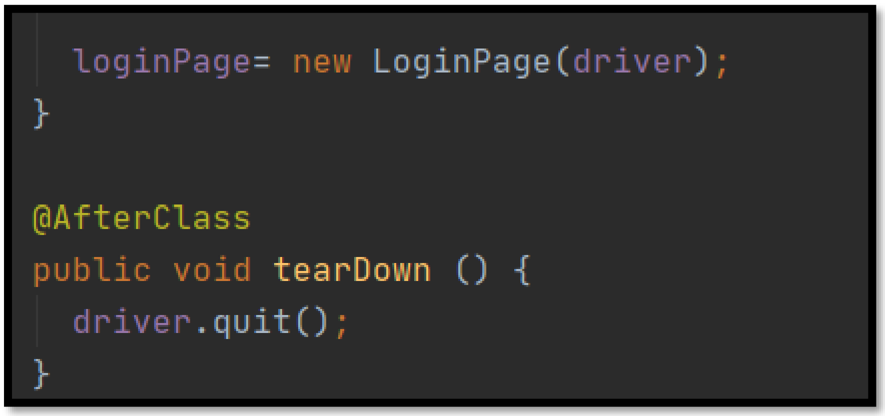
After loading the application, let's implement a way for our 3 Test Scripts to access the LoginPage. protected LoginPage loginPage; I marked this field protected so each Test Script has access to the LoginPage after inheriting the BaseTest.



```
public class BaseTest {
  private WebDriver driver;
  private final String AUT_URL = "https://www.saucedemo.com";
  protected LoginPage loginPage;
```

Create an instance by writing loginPage = new LoginPage(driver);

Next is the annotation for @AfterClass. This configuration will tear down our test automatically after executing the Test Script. public void tearDown () { }
  driver.quit();



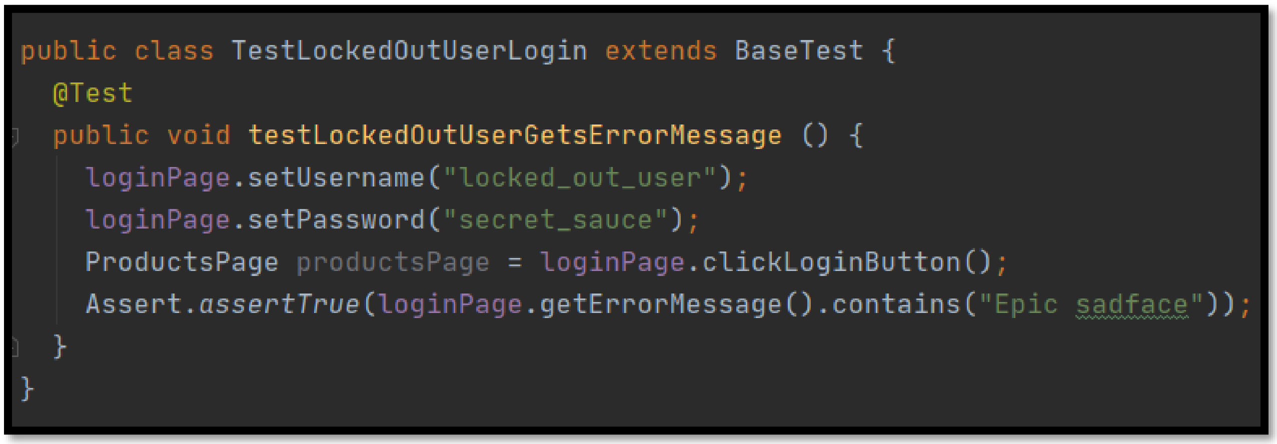
```
    loginPage= new LoginPage(driver);
}

@AfterClass
public void tearDown () {
    driver.quit();
}
```

We can write more in this BaseTest but that's all I'm going to write for now. The benefit of having a BaseTest is to avoid setting up and tearing down every Test. Now, let's move on to creating our Login test.

Login Test

For the Login LockedOutUserTest, we inherit the BaseTest by writing extends BaseTest { }
The annotation is @Test / public void testLockedOutUserGetsErrorMessage () {. Now, we write the logic for testing the locked out user.
From the loginPage., we the setUsername to ("locked_out_user"); / Also, the loginPage. will setPassword to ("secret_sauce") / then click the button on the loginPage.cl. Do you see how the intellisense shows the method clickLoginButton() and ProductsPage? This lets us know that calling clickLoginButton has a transition which returns the ProductsPage. The application makes a transition by changing from the LoginPage to the Products page. Therefore, we assign clickLoginButton to ProductsPage productsPage =

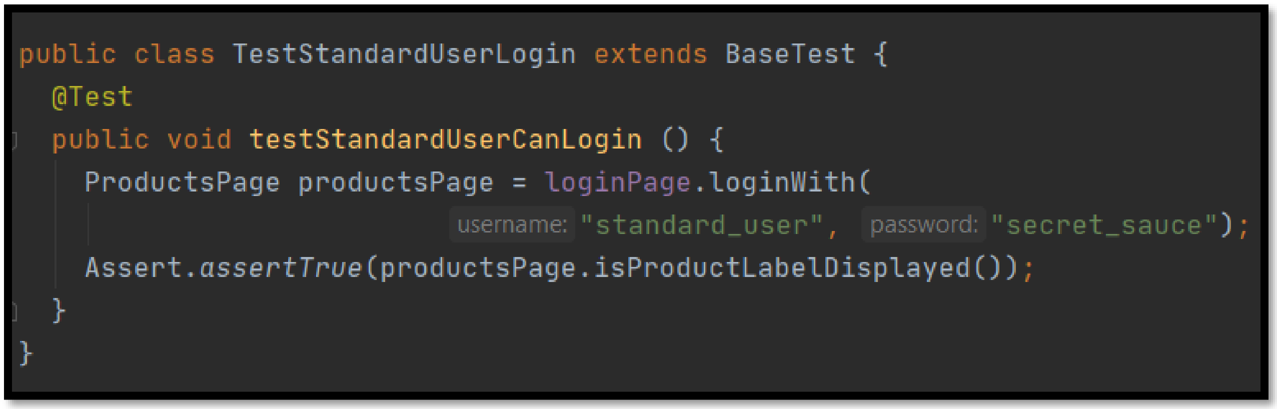
Thankfully, we did not add a failed assertion to our Page Object class. In this Test Script, we expect an error because the user is locked out and we want the test to pass. Go back to the application. The username is locked_out_user, password is secret_sauce and we see the message contains Epic sadface. Go back to our Test Script and write Assert.assertTrue(loginPage.getErrorMessage(). Make sure the message contains("Epic sadface"));



```java
public class TestLockedOutUserLogin extends BaseTest {
  @Test
  public void testLockedOutUserGetsErrorMessage () {
    loginPage.setUsername("locked_out_user");
    loginPage.setPassword("secret_sauce");
    ProductsPage productsPage = loginPage.clickLoginButton();
    Assert.assertTrue(loginPage.getErrorMessage().contains("Epic sadface"));
  }
}
```

Let's Run. We see the test Passed. Next is to test a user can log into the application then verify the Products label is displayed. You see, I have each test in separate classes but they can easily be in the same class. Either way is okay. On a project, I would group all test related to the Locked Out user in one class and all test related to a Standard User in a different class because we may have more than 1 Test Script for each user type.

First, we extend BaseTest. @Test / public void testStandardUserCanLogin () { } This time, let's use the convenience method: On the loginPage. we will loginWith("standard_user", "secret_sauce"); then transition to the ProductsPage productsPage = . Now, verify the Products label is displayed. Assert.assertTrue(productsPage.isProductsLabelDisplayed());



```java
public class TestStandardUserLogin extends BaseTest {
  @Test
  public void testStandardUserCanLogin () {
    ProductsPage productsPage = loginPage.loginWith(
                        username: "standard_user",  password: "secret_sauce");
    Assert.assertTrue(productsPage.isProductLabelDisplayed());
  }
}
```

Let's Run. The Test Passed. Let's create one more test really quick for the Products page.

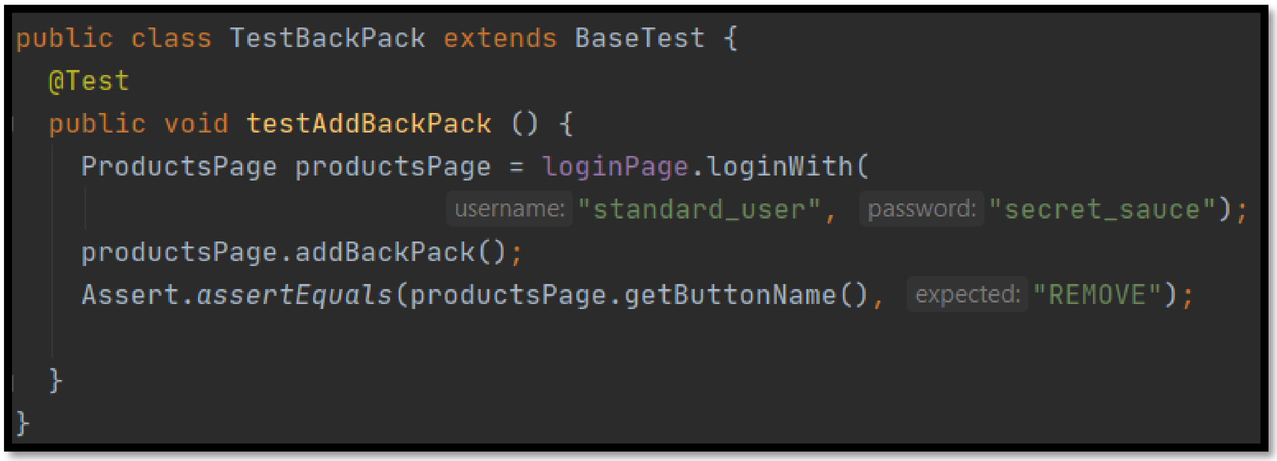
Products Test

Go to the TestBackPack class and it also extends BaseTest @Test / public void testAddBackPack () { }

On the loginPage.loginWith("standard_user", "secret_sauce"); assign to the ProductsPage productsPage =. At this point our test is on the productsPage. and the test will addBackPack(). After adding a backpack,

Rex Jones II

we are going to verify the button name for Add to Cart changes to REMOVE. Assert.assertEquals(productsPage.getButtonName(), we expect the name to be "REMOVE"); in all caps.



```
public class TestBackPack extends BaseTest {
  @Test
  public void testAddBackPack () {
    ProductsPage productsPage = loginPage.loginWith(
                     username: "standard_user",  password: "secret_sauce");
    productsPage.addBackPack();
    Assert.assertEquals(productsPage.getButtonName(),  expected: "REMOVE");

  }
}
```

This time, let's run all 3 tests at the same time. We see all 3 Test were a success. That's it for creating a Base Test and creating 3 Test Scripts using the Page Object Model.

If you like this video, consider following me on Twitter and connecting with me on LinkedIn and Facebook. You can also subscribe to my channel and click the bell icon. I have more videos coming. The documentation and code be uploaded to GitHub.

Social Media Contact

- ✓ YouTube https://www.youtube.com/c/RexJonesII/videos
- ✓ Facebook http://facebook.com/JonesRexII
- ✓ Twitter https://twitter.com/RexJonesII
- ✓ GitHub https://github.com/RexJonesII/Free-Videos
- ✓ LinkedIn https://www.linkedin.com/in/rexjones34/