**What is the difference between Page Object Model (POM) & Page Factory in Selenium?**

**Page Object Model (POM) & Page Factory in Selenium: Ultimate Guide**

Before we learn about Page Object Model, lets understand -

**Why POM ?**

Starting a UI Automation in Selenium WebDriver is NOT a tough task. You just need to find elements, perform operations on it .

Consider this simple script to login into a website

As you can observe, all we are doing is finding elements and filling values for those elements.

This is a small script. Script maintenance looks easy. But with time test suite will grow. As you add more and more lines to your code, things become tough.

The chief problem with script maintenance is that if 10 different scripts are using the same page element, with any change in that element, you need to change all 10 scripts. This is time consuming and error prone.

A better approach to script maintenance is to create a separate class file which would find web elements , fill them or verify them. This class can be reused in all the scripts using that element. In future if there is change in the web element , we need to make change in just 1 class file and not 10 different scripts.

This approach is called **Page Object Model(POM)**. It helps make code **more readable, maintainable**, and **reusable.**

**What is POM?**

* **Page Object Model** is a design pattern to create **Object Repository** for web UI elements.
* Under this model, for each web page in the application there should be corresponding page class.
* This Page class will find the WebElements of that web page and also contains Page methods which perform operations on those WebElements.
* Name of these methods should be given as per the task they are performing i.e., if a loader is waiting for payment gateway to be appear, POM method name can be waitForPaymentScreenDisplay().

**Advantages of POM**

1. Page Object Patten says operations and flows in the UI should be separated from verification. This concept makes our code cleaner and easy to understand.
2. Second benefit is the **object repository is independent of testcases**, so we can use the same object repository for a different purpose with different tools. For example, we can integrate POM with TestNG/JUnit for functional testing and at the same time with JBehave/Cucumber for acceptance testing.
3. Code becomes less and optimized because of the reusable page methods in the POM classes.
4. **Methods** get **more realistic names** which can be easily mapped with the operation happening in UI. i.e. if after clicking on the button we land on the home page, the method name will be like 'gotoHomePage()'.

**How to implement POM ?**

Simple POM:

It's the basic structure of Page object model (POM) where all Web Elements of the **AUT** and the method that operate on these Web Elements are maintained inside a class file. Task like **verification** should be **separate** as part of Test methods.

**Complete Example**

**TestCase:** Go to Guru99 Demo Site .

Step 1) Go to Guru99 Demo Site

Step 2) In home page check text **"Guru99 Bank"** is present

Step 3) Login into application

Step 4) Verify that the Home page contains text as "Manger Id : demo"

Here are we are dealing with 2 pages

1. Login Page
2. Home Page (shown once you login)

Accordingly we create 2 POM classes

**Guru99 Login page POM**

1. **package** pages;
3. **import** org.openqa.selenium.By;
5. **import** org.openqa.selenium.WebDriver;
7. **public** **class** Guru99Login {
9. WebDriver driver;
11. By user99GuruName = By.name("uid");
13. By password99Guru = By.name("password");
15. By titleText =By.className("barone");
17. By login = By.name("btnLogin");


21. **public** Guru99Login(WebDriver driver){
23. **this**.driver = driver;
25. }
27. //Set user name in textbox
29. **public** **void** setUserName(String strUserName){
31. driver.findElement(user99GuruName).sendKeys(strUserName);;
33. }


37. //Set password in password textbox
39. **public** **void** setPassword(String strPassword){
41. driver.findElement(password99Guru).sendKeys(strPassword);
43. }


47. //Click on login button
49. **public** **void** clickLogin(){
51. driver.findElement(login).click();
53. }


57. //Get the title of Login Page
59. **public** String getLoginTitle(){
61. **return** driver.findElement(titleText).getText();
63. }
65. /\*\*
67. \* This POM method will be exposed in test case to login in the application
69. \* @param strUserName
71. \* @param strPasword
73. \* @return
75. \*/
77. **public** **void** loginToGuru99(String strUserName,String strPasword){
79. //Fill user name
81. **this**.setUserName(strUserName);
83. //Fill password
85. **this**.setPassword(strPasword);
87. //Click Login button
89. **this**.clickLogin();


93. }
95. }

**Guru99 Home Page POM**

1. **package** pages;
3. **import** org.openqa.selenium.By;
5. **import** org.openqa.selenium.WebDriver;
7. **public** **class** Guru99HomePage {
9. WebDriver driver;
11. By homePageUserName = By.xpath("//table//tr[@class='heading3']");


15. **public** Guru99HomePage(WebDriver driver){
17. **this**.driver = driver;
19. }


23. //Get the User name from Home Page
25. **public** String getHomePageDashboardUserName(){
27. **return** driver.findElement(homePageUserName).getText();
29. }
31. }

**Guru99 Simple POM Test case**

1. **package** test;
3. **import** java.util.concurrent.TimeUnit;
5. **import** org.openqa.selenium.WebDriver;
7. **import** org.openqa.selenium.firefox.FirefoxDriver;
9. **import** org.testng.Assert;
11. **import** org.testng.annotations.BeforeTest;
13. **import** org.testng.annotations.Test;
15. **import** pages.Guru99HomePage;
17. **import** pages.Guru99Login;
19. **public** **class** Test99GuruLogin {
21. WebDriver driver;
23. Guru99Login objLogin;
25. Guru99HomePage objHomePage;


29. @BeforeTest
31. **public** **void** setup(){
33. driver = **new** FirefoxDriver();
35. driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
37. driver.**get**("http://demo.guru99.com/V4/");
39. }
41. /\*\*
43. \* This test case will login in http://demo.guru99.com/V4/
45. \* Verify login page title as guru99 bank
47. \* Login to application
49. \* Verify the home page using Dashboard message
51. \*/
53. @Test(priority=0)
55. **public** **void** test\_Home\_Page\_Appear\_Correct(){
57. //Create Login Page object
59. objLogin = **new** Guru99Login(driver);
61. //Verify login page title
63. String loginPageTitle = objLogin.getLoginTitle();
65. Assert.assertTrue(loginPageTitle.toLowerCase().contains("guru99 bank"));
67. //login to application
69. objLogin.loginToGuru99("mgr123", "mgr!23");
71. // go the next page
73. objHomePage = **new** Guru99HomePage(driver);
75. //Verify home page
77. Assert.assertTrue(objHomePage.getHomePageDashboardUserName().toLowerCase().contains("manger id : mgr123"));
79. }

**What is Page Factory?**

Page Factory is an inbuilt page object model concept for Selenium WebDriver but it is very optimized.

Here as well we follow the concept of separation of Page Object repository and Test methods. Additionally with the help of PageFactory class we use annotations **@FindBy** to find WebElement. We use initElements method to initialize web elements

**@FindBy** can accept **tagName, partialLinkText, name, linkText, id, css, className, xpath**as attributes.

Let's look at the same example as above using Page Factory

**Guru99 Login page with Page Factory**

1. **package** PageFactory;
3. **import** org.openqa.selenium.WebDriver;
5. **import** org.openqa.selenium.WebElement;
7. **import** org.openqa.selenium.support.FindBy;
9. **import** org.openqa.selenium.support.PageFactory;
11. **public** **class** Guru99Login {
13. /\*\*
15. \* All WebElements are identified by @FindBy annotation
17. \*/
19. WebDriver driver;
21. @FindBy(name="uid")
23. WebElement user99GuruName;


27. @FindBy(name="password")
29. WebElement password99Guru;


33. @FindBy(className="barone")
35. WebElement titleText;


39. @FindBy(name="btnLogin")
41. WebElement login;


45. **public** Guru99Login(WebDriver driver){
47. **this**.driver = driver;
49. //This initElements method will create all WebElements
51. PageFactory.initElements(driver, **this**);
53. }
55. //Set user name in textbox
57. **public** **void** setUserName(String strUserName){
59. user99GuruName.sendKeys(strUserName);


63. }


67. //Set password in password textbox
69. **public** **void** setPassword(String strPassword){
71. password99Guru.sendKeys(strPassword);
73. }


77. //Click on login button
79. **public** **void** clickLogin(){
81. login.click();
83. }


87. //Get the title of Login Page
89. **public** String getLoginTitle(){
91. **return** titleText.getText();
93. }
95. /\*\*
97. \* This POM method will be exposed in test case to login in the application
99. \* @param strUserName
101. \* @param strPasword
103. \* @return
105. \*/
107. **public** **void** loginToGuru99(String strUserName,String strPasword){
109. //Fill user name
111. **this**.setUserName(strUserName);
113. //Fill password
115. **this**.setPassword(strPasword);
117. //Click Login button
119. **this**.clickLogin();


123. }
125. }

**Guru99 Home Page with Page Factory**

1. **package** PageFactory;
3. **import** org.openqa.selenium.WebDriver;
5. **import** org.openqa.selenium.WebElement;
7. **import** org.openqa.selenium.support.FindBy;
9. **import** org.openqa.selenium.support.PageFactory;
11. **public** **class** Guru99HomePage {
13. WebDriver driver;
15. @FindBy(xpath="//table//tr[@class='heading3']")
17. WebElement homePageUserName;


21. **public** Guru99HomePage(WebDriver driver){
23. **this**.driver = driver;
25. //This initElements method will create all WebElements
27. PageFactory.initElements(driver, **this**);
29. }


33. //Get the User name from Home Page
35. **public** String getHomePageDashboardUserName(){
37. **return** homePageUserName.getText();
39. }
41. }

**Guru99 TestCase with Page Factory concept**

1. **package** test;
3. **import** java.util.concurrent.TimeUnit;
5. **import** org.openqa.selenium.WebDriver;
7. **import** org.openqa.selenium.firefox.FirefoxDriver;
9. **import** org.testng.Assert;
11. **import** org.testng.annotations.BeforeTest;
13. **import** org.testng.annotations.Test;
15. **import** PageFactory.Guru99HomePage;
17. **import** PageFactory.Guru99Login;
19. **public** **class** Test99GuruLoginWithPageFactory {
21. WebDriver driver;
23. Guru99Login objLogin;
25. Guru99HomePage objHomePage;


29. @BeforeTest
31. **public** **void** setup(){
33. driver = **new** FirefoxDriver();
35. driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
37. driver.**get**("http://demo.guru99.com/V4/");
39. }
41. /\*\*
43. \* This test go to http://demo.guru99.com/V4/
45. \* Verify login page title as guru99 bank
47. \* Login to application
49. \* Verify the home page using Dashboard message
51. \*/
53. @Test(priority=0)
55. **public** **void** test\_Home\_Page\_Appear\_Correct(){
57. //Create Login Page object
59. objLogin = **new** Guru99Login(driver);
61. //Verify login page title
63. String loginPageTitle = objLogin.getLoginTitle();
65. Assert.assertTrue(loginPageTitle.toLowerCase().contains("guru99 bank"));
67. //login to application
69. objLogin.loginToGuru99("mgr123", "mgr!23");
71. // go the next page
73. objHomePage = **new** Guru99HomePage(driver);
75. //Verify home page
77. Assert.assertTrue(objHomePage.getHomePageDashboardUserName().toLowerCase().contains("manger id : mgr123"));
79. }


83. }

Complete Project Structure will look like the diagram:

**AjaxElementLocatorFactory**

One of the key advantage of using Page Factory pattern is AjaxElementLocatorFactory Class.

It is working on lazy loading concept, i.e. a timeout for a WebElement will be assigned to the Object page class with the help of AjaxElementLocatorFactory .

Here, when an operation is performed on an element the wait for its visibility starts from that moment only. If the element is not found in the given time interval, test case execution will throw 'NoSuchElementException' exception.

PageFactory is a support class for the Selenium 2 Page Objects pattern which you are using, and the AjaxElementLocatorFactory is the factory for the element locators. In your case the constructor will looks like:

**public GoogleResultsPage() {**

**PageFactory.initElements(new AjaxElementLocatorFactory(driver, 15), this);**

**}**

This code will wait maximum of 15 seconds until the elements specified by annotations will appear on the page

**Summary**

1. Page Object Model is an Object repository design pattern in Selenium WebDriver.
2. POM creates our testing code maintainable, reusable.
3. Page Factory is an optimized way to create object repository in POM concept.
4. AjaxElementLocatorFactory is a lazy load concept in Page Factory pattern to identify WebElements only when they are used in any operation.

**Is object initialization the only difference between POM and Page Factory in Selenium?**

The following info is taken from the[PageFactory class JavaDoc](https://github.com/SeleniumHQ/selenium/blob/master/java/client/src/org/openqa/selenium/support/PageFactory.java) of the initElements() method:

1. **public** **static** <T> T initElements(WebDriver driver,
2. Class<T> pageClassToProxy) {
4. T page = instantiatePage(driver, pageClassToProxy);
5. initElements(driver, page);
6. **return** page;
8. }

* **Instantiate an instance of the given class**, and
* **Set a lazy proxy for each of the WebElement and List<WebElement> fields that have been declared**, assuming that the field name is also the HTML element's "id" or "name".

This means that for the class:

1. **public** **class** Page {
2. **private** WebElement submit;
3. }

there will be an element that can be located using the xpath expression

1. //\*[@id='submit']" or //\*[@name='submit']

* **By default, the element or the list is looked up each and every time a method is called upon it. To change this behaviour, simply annotate the field with the CacheLookup annotation.**
* **To change how the element is located**, use the FindBy annotation.

——————————————————————————————

So, when using PageFactory,

1. WebElement or List<WebElement> variables are declared as **private** in the page object class
2. If the WebElement variable has the same name with the web element id or name attribute, **there is no need to specify a locator for the element**
3. Otherwise, **the element locator is specified with a FindBy annotation**
4. **initElements() method sets a lazy proxy for each web element from the page object class; this proxy associates the web element with its locator**; but the web element is not searched yet
5. when the element is used (by calling methods on it such as click(), getText(), isEnabled(), etc), the following happen:
   1. the locator associated with the web element during the page class initialization is retrieved
   2. the element is searched in the browser dom
   3. if the element is found, the code interacts with it
6. if you dont want the element to be searched in the browser dom every time it is used, the CacheLookup annotation can be used; in this case, the element is cached after being found for the first time

This is just the short story of the Page Factory.

It is very beneficial to read the code and try to understand how it works.

This did wonders for me since, to understand what the code does, I had to learn things that I did not know well such as

* generics
* reflection
* lazy initialization
* proxy classes

It took a while to figure it out but I am very glad that I did it.

Some more Notes:

A Page Object Model is a way of representing an application in a test framework. For every ‘page’ in the application you create a Page Object to reference the ‘page’. A Page Factory is one way of implementing a Page Object Model. Search for “Java Factory” to understand the concept of a Factory.

# The [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory)

In order to support the [PageObject](https://code.google.com/p/selenium/wiki/PageObjects) pattern, WebDriver’s support library contains a factory class.

## **A Simple Example**

In order to use the PageFactory, first declare some fields on a PageObject that are WebElements or List<WebElement>, for example:

package org.openqa.selenium.example;

import org.openqa.selenium.WebElement;

public class GoogleSearchPage {

    // Here's the element

    private WebElement q;

    public void searchFor(String text) {

        // And here we use it. Note that it looks like we've

        // not properly instantiated it yet....

        q.sendKeys(text);

        q.submit();

    }

}

In order for this code to work and not throw a NullPointerException because the “q” field isn’t instantiated, we need to initialise the PageObject:

package org.openqa.selenium.example;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.htmlunit.HtmlUnitDriver;

import org.openqa.selenium.support.PageFactory;

public class UsingGoogleSearchPage {

    public static void main(String[] args) {

        // Create a new instance of a driver

        WebDriver driver = new HtmlUnitDriver();

        // Navigate to the right place

        driver.get("http://www.google.com/");

        // Create a new instance of the search page class

        // and initialise any WebElement fields in it.

        GoogleSearchPage page = PageFactory.initElements(driver, GoogleSearchPage.class);

        // And now do the search.

        page.searchFor("Cheese");

    }

}

### Explanation

The [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory) relies on using sensible defaults: the name of the field in the Java class is assumed to be the “id” or “name” of the element on the HTML page. That is, in the example above, the line:

    q.sendKeys(text);

is equivalent to:

    driver.findElement(By.id("q")).sendKeys(text);

The driver instance that’s used is the one that’s passed to the [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory)‘s initElements method.

In the example given, we rely on the [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory) to instantiate the instance of the PageObject. It does this by first looking for a constructor that takes “WebDriver” as its sole argument (public SomePage(WebDriver driver) {). If this is not present, then the default constructor is called. Sometimes, however, the PageObject depends on more than just an instance of the WebDriver interface. Should this be the case, it is possible to get the [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory) to initialise the elements of an already constructed object:

ComplexPageObject page = new ComplexPageObject("expected title", driver);

// Note, we still need to pass in an instance of driver for the

// initialised elements to use

PageFactory.initElements(driver, page);

## **Making the Example Work Using Annotations**

When we run the example, the [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory) will search for an element on the page that matches the field name of the WebElement in the class. It does this by first looking for an element with a matching ID attribute. If this fails, the [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory) falls back to searching for an element by the value of its “name” attribute.

Although the code works, someone who’s not familiar with the source of the Google home page may not know that the name of the field is “q”. Fortunately, we can pick a meaningful name and change the strategy used to look the element up using an annotation:

package org.openqa.selenium.example;

import org.openqa.selenium.By;

import org.openqa.selenium.support.FindBy;

import org.openqa.selenium.support.How;

import org.openqa.selenium.WebElement;

public class GoogleSearchPage {

    // The element is now looked up using the name attribute

    @FindBy(how = How.NAME, using = "q")

    private WebElement searchBox;

    public void searchFor(String text) {

        // We continue using the element just as before

        searchBox.sendKeys(text);

        searchBox.submit();

    }

}

One wrinkle that remains is that every time we call a method on the WebElement, the driver will go and find it on the current page again. In an AJAX-heavy application this is what you would like to happen, but in the case of the Google search page we know that the element is always going to be there and won’t change. We also know that we won’t be navigating away from the page and returning (which would mean that a different element with the same name would be present) It would be handy if we could “cache” the element once we’d looked it up:

package org.openqa.selenium.example;

import org.openqa.selenium.By;

import org.openqa.selenium.support.CacheLookup;

import org.openqa.selenium.support.FindBy;

import org.openqa.selenium.support.How;

import org.openqa.selenium.WebElement;

public class GoogleSearchPage {

    // The element is now looked up using the name attribute,

    // and we never look it up once it has been used the first time

    @FindBy(how = How.NAME, using = "q")

    @CacheLookup

    private WebElement searchBox;

    public void searchFor(String text) {

        // We continue using the element just as before

        searchBox.sendKeys(text);

        searchBox.submit();

    }

}

### Reducing Verbosity

The example above is still a little verbose. A slightly cleaner way of annotating the field would be:

public class GoogleSearchPage {

  @FindBy(name = "q")

  private WebElement searchBox;

  // The rest of the class is unchanged.

}

## **Notes**

* If you use the [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory), you can assume that the fields are initialised. If you don’t use the [PageFactory](https://code.google.com/p/selenium/wiki/PageFactory), then NullPointerExceptions will be thrown if you make the assumption that the fields are already initialised.
* List<WebElement> fields are decorated if and only if they have @FindBy or @FindBys annotation. Default search strategy “by id or name” that works for WebElement fields is hardly suitable for lists because it is rare to have several elements with the same id or name on a page.
* WebElements are evaluated lazily. That is, if you never use a WebElement field in a PageObject, there will never be a call to “findElement” for it.
* The functionality works using dynamic proxies. This means that you shouldn’t expect a WebElement to be a particular subclass, even if you know the type of the driver. For example, if you are using the [HtmlUnitDriver](https://code.google.com/p/selenium/wiki/HtmlUnitDriver), you shouldn’t expect the WebElement field to be initialised with an instance of HtmlUnitWebElement.