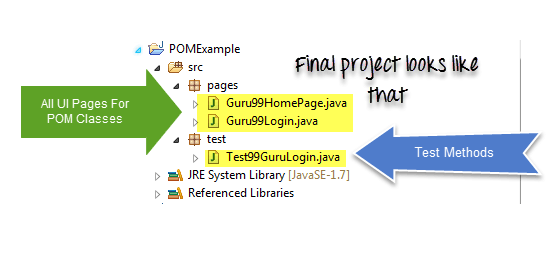
**What is Page Object Model?**

**Page Object Model (POM)** is a design pattern, popularly used in test automation that creates Object Repository for web UI elements. The advantage of the model is that it reduces code duplication and improves test maintenance.

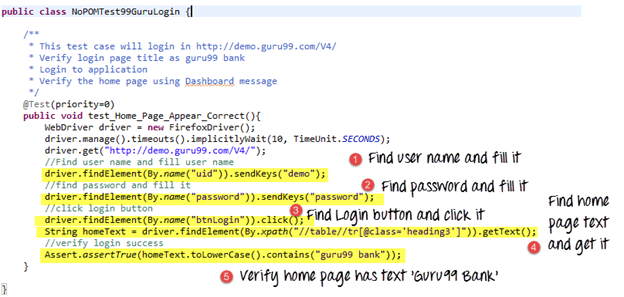
Under this model, for each web page in the application, there should be a corresponding Page Class. This Page class will identify the WebElements of that web page and also contains Page methods which perform operations on those WebElements. Name of these methods should be given as per the task they are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



**Why Page Object Model?**

Starting an 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



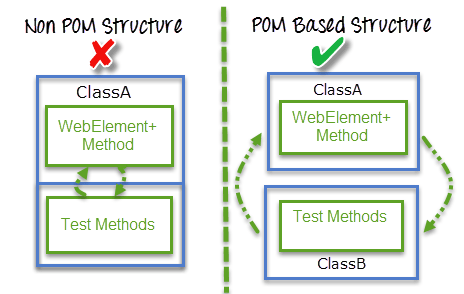
As you can observe, all we are doing is finding elements and filling values for those elements.

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 a change in the web element, we need to make the change in just 1 class file and not 10 different scripts.

This approach is called Page Object Model in Selenium. It helps make the code more readable, maintainable, and reusable.



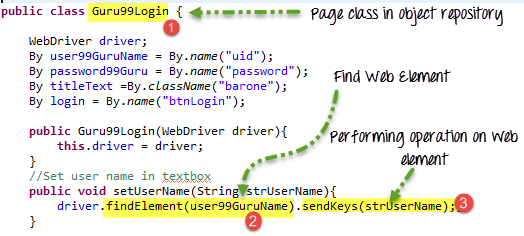
**Advantages of POM**

1. Page Object Design Pattern says operations and flows in the UI should be separated from verification. This concept makes our code cleaner and easy to understand.
2. The Second benefit is the object repository is independent of test cases, so we can use the same object repository for a different purpose with different tools. For example, we can integrate Page Object Model in Selenium with TestNG/JUnit for functional[Testing](https://www.guru99.com/software-testing.html)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:

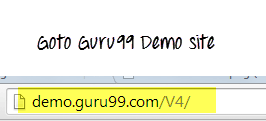
It’s the basic structure of Page object model framework where all Web Elements of the **AUT** and the method that operate on these Web Elements are maintained inside a class file.A task like verification should be separate as part of Test methods.



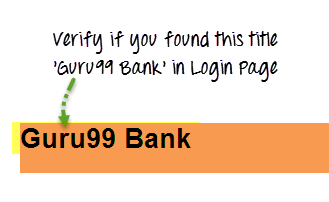
**Complete Example**

**TestCase:** Go to Guru99 Demo Site.

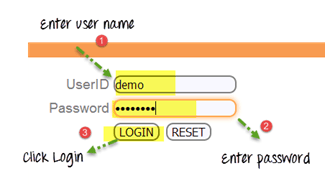
Step 1) Go to Guru99 Demo Site



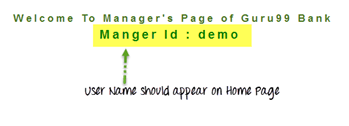
Step 2) In home page check text “Guru99 Bank” is present



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 in Selenium classes

**Guru99 Login page POM**

package pages;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

public class Guru99Login {

WebDriver driver;

By user99GuruName = By.name("uid");

By password99Guru = By.name("password");

By titleText =By.className("barone");

By login = By.name("btnLogin");

public Guru99Login(WebDriver driver){

this.driver = driver;

}

//Set user name in textbox

public void setUserName(String strUserName){

driver.findElement(user99GuruName).sendKeys(strUserName);

}

//Set password in password textbox

public void setPassword(String strPassword){

driver.findElement(password99Guru).sendKeys(strPassword);

}

//Click on login button

public void clickLogin(){

driver.findElement(login).click();

}

//Get the title of Login Page

public String getLoginTitle(){

return driver.findElement(titleText).getText();

}

/\*\*

\* This POM method will be exposed in test case to login in the application

\* @param strUserName

\* @param strPasword

\* @return

\*/

public void loginToGuru99(String strUserName,String strPasword){

//Fill user name

this.setUserName(strUserName);

//Fill password

this.setPassword(strPasword);

//Click Login button

this.clickLogin();

}

}

**Guru99 Home Page POM in Selenium**

package pages;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

public class Guru99HomePage {

WebDriver driver;

By homePageUserName = By.xpath("//table//tr[@class='heading3']");

public Guru99HomePage(WebDriver driver){

this.driver = driver;

}

//Get the User name from Home Page

public String getHomePageDashboardUserName(){

return driver.findElement(homePageUserName).getText();

}

}

**Guru99 Simple POM in Selenium Test case**

package test;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.testng.Assert;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.Test;

import pages.Guru99HomePage;

import pages.Guru99Login;

public class Test99GuruLogin {

String driverPath = "C:\\geckodriver.exe";

WebDriver driver;

Guru99Login objLogin;

Guru99HomePage objHomePage;

@BeforeTest

public void setup(){

System.setProperty("webdriver.gecko.driver", driverPath);

driver = new FirefoxDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.get("http://demo.guru99.com/V4/");

}

/\*\*

\* This test case will login in http://demo.guru99.com/V4/

\* Verify login page title as guru99 bank

\* Login to application

\* Verify the home page using Dashboard message

\*/

@Test(priority=0)

public void test\_Home\_Page\_Appear\_Correct(){

//Create Login Page object

objLogin = new Guru99Login(driver);

//Verify login page title

String loginPageTitle = objLogin.getLoginTitle();

Assert.assertTrue(loginPageTitle.toLowerCase().contains("guru99 bank"));

//login to application

objLogin.loginToGuru99("mgr123", "mgr!23");

// go the next page

objHomePage = new Guru99HomePage(driver);

//Verify home page

Assert.assertTrue(objHomePage.getHomePageDashboardUserName().toLowerCase().contains("manger id : mgr123"));

}

**What is Page Factory in Selenium?**

**Page Factory in Selenium** is an inbuilt Page Object Model framework concept for Selenium WebDriver but it is very optimized. It is used for initialization of Page objects or to instantiate the Page object itself. It is also used to initialize Page class elements without using “FindElement/s.”

Here as well, we follow the concept of separation of Page Object Repository and Test Methods. Additionally, with the help of class PageFactory in Selenium, 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**

package PageFactory;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.support.FindBy;

import org.openqa.selenium.support.PageFactory;

public class Guru99Login {

/\*\*

\* All WebElements are identified by @FindBy annotation

\*/

WebDriver driver;

@FindBy(name="uid")

WebElement user99GuruName;

@FindBy(name="password")

WebElement password99Guru;

@FindBy(className="barone")

WebElement titleText;

@FindBy(name="btnLogin")

WebElement login;

public Guru99Login(WebDriver driver){

this.driver = driver;

//This initElements method will create all WebElements

PageFactory.initElements(driver, this);

}

//Set user name in textbox

public void setUserName(String strUserName){

user99GuruName.sendKeys(strUserName);

}

//Set password in password textbox

public void setPassword(String strPassword){

password99Guru.sendKeys(strPassword);

}

//Click on login button

public void clickLogin(){

login.click();

}

//Get the title of Login Page

public String getLoginTitle(){

return titleText.getText();

}

/\*\*

\* This POM method will be exposed in test case to login in the application

\* @param strUserName

\* @param strPasword

\* @return

\*/

public void loginToGuru99(String strUserName,String strPasword){

//Fill user name

this.setUserName(strUserName);

//Fill password

this.setPassword(strPasword);

//Click Login button

this.clickLogin();

}

}

**Guru99 Home Page with Page Factory**

package PageFactory;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.support.FindBy;

import org.openqa.selenium.support.PageFactory;

public class Guru99HomePage {

WebDriver driver;

@FindBy(xpath="//table//tr[@class='heading3']")

WebElement homePageUserName;

public Guru99HomePage(WebDriver driver){

this.driver = driver;

//This initElements method will create all WebElements

PageFactory.initElements(driver, this);

}

//Get the User name from Home Page

public String getHomePageDashboardUserName(){

return homePageUserName.getText();

}

}

**Guru99 TestCase with Page Factory concept**

package test;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.firefox.FirefoxDriver;

import org.testng.Assert;

import org.testng.annotations.BeforeTest;

import org.testng.annotations.Test;

import PageFactory.Guru99HomePage;

import PageFactory.Guru99Login;

public class Test99GuruLoginWithPageFactory {

String driverPath = "C:\\geckodriver.exe";

WebDriver driver;

Guru99Login objLogin;

Guru99HomePage objHomePage;

@BeforeTest

public void setup(){

System.setProperty("webdriver.gecko.driver", driverPath);

driver = new FirefoxDriver();

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.get("http://demo.guru99.com/V4/");

}

/\*\*

\* This test go to http://demo.guru99.com/V4/

\* Verify login page title as guru99 bank

\* Login to application

\* Verify the home page using Dashboard message

\*/

@Test(priority=0)

public void test\_Home\_Page\_Appear\_Correct(){

//Create Login Page object

objLogin = new Guru99Login(driver);

//Verify login page title

String loginPageTitle = objLogin.getLoginTitle();

Assert.assertTrue(loginPageTitle.toLowerCase().contains("guru99 bank"));

//login to application

objLogin.loginToGuru99("mgr123", "mgr!23");

// go the next page

objHomePage = new Guru99HomePage(driver);

//Verify home page

Assert.assertTrue(objHomePage.getHomePageDashboardUserName().toLowerCase().contains("manger id : mgr123"));

}

}

# **Selenium Maven**

Maven is the latest build testing tool. It has several new features as compare to Ant, like dependency, etc.

Maven is a project build or project management tool. It is used to check the compilation issues between framework components whenever multiple test engineer integrates their files into the same framework.

It always maintained the monitor, framework components, or build, and it provides build status modification, whenever modification happens in the framework.

It provides '**build success**' message if no compilation issues in the framework or else provide '**build failure**' message.

Maven has new features like dependency, which is used to download the dependency jar from the internet before the test execution.

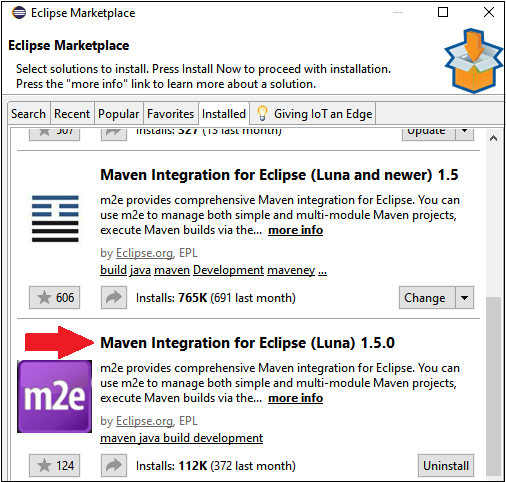
With the help of Maven, we can execute the test scripts in the command line without an eclipse. And it always provides the framework folder structure.

Maven Eclipse plug-in

It is a default plug-in for the latest Eclipse versions like Mars, Luna, oxygen, which is used to create a Maven project through Eclipse.

**Installing Maven plug-in for Eclipse and use it with Selenium TestNG**

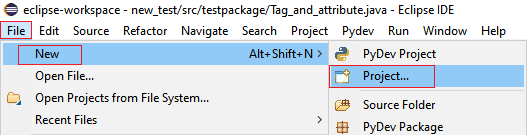
Most of the time, Maven plug-in is automatically installed in the Eclipse, but if it is not present, we will go to the **Eclipse Market Place** and search for **Maven** and download the **M2E integrated version** from there.



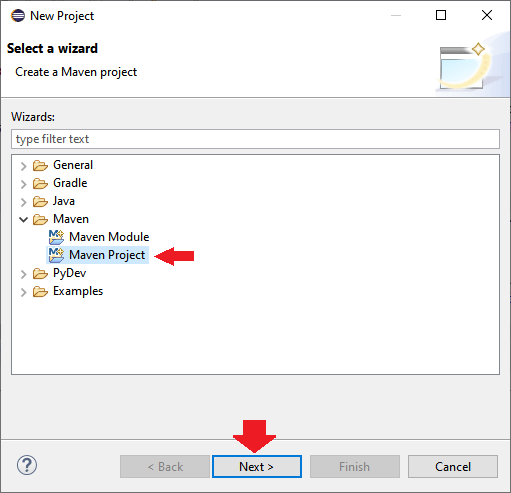
**Steps to create Maven project**

To create a Maven project, follow the below steps:

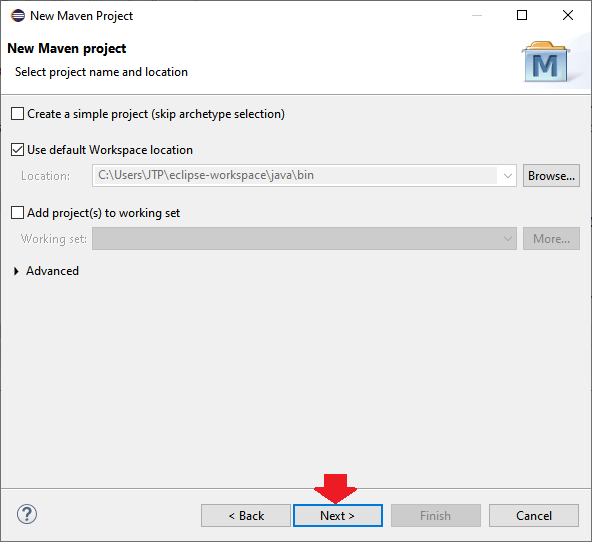
* Go to the **File → New → Project**



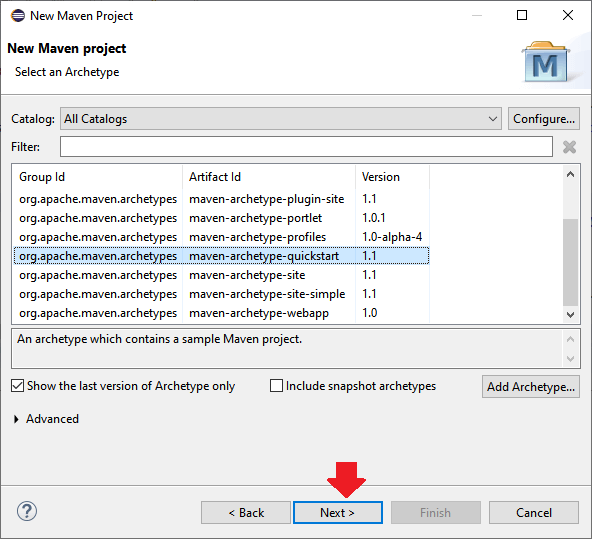
* Then, expand the **Maven** folder and select the **Maven Project** options, and click on the **Next**



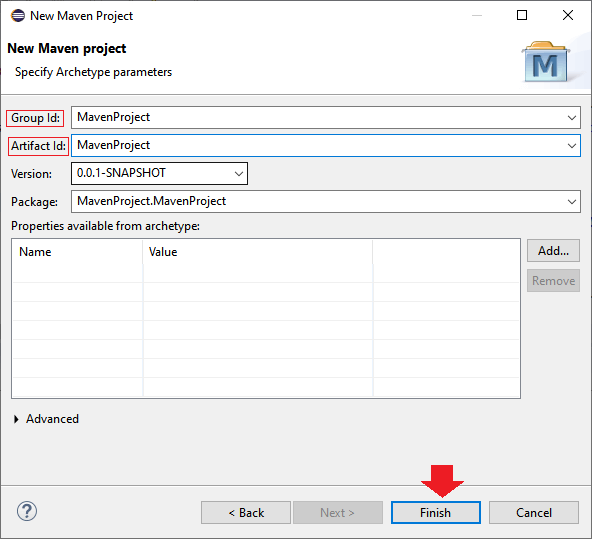
* Again click on the **Next** button for the further process.



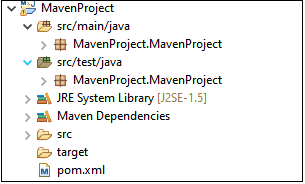
* Click on the **Next** button without doing anything.



* After click on the Next button, we will provide the **Group ID** and **Artifact ID** in our case, and we will give the Group id as **MavenProject** and Artifact ID as **MavenProject**, and click on the **Finish**button as we can see in the below screenshot:



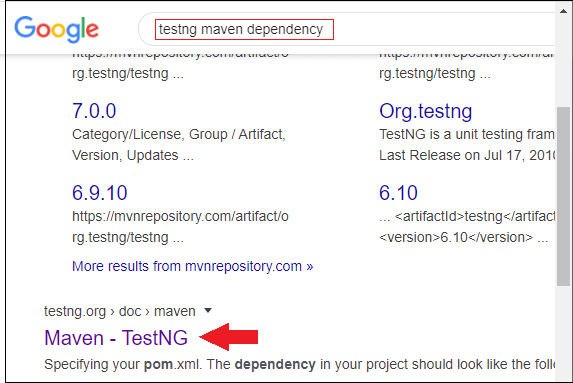
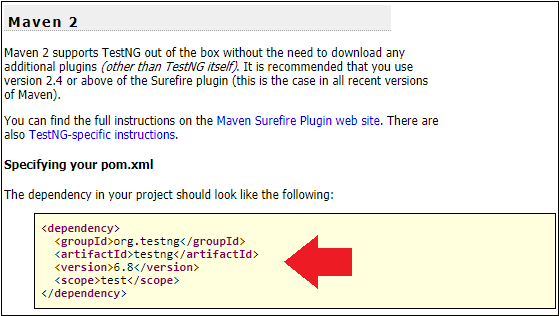
* Once we are done with creating the Maven project, our **Maven folder structure** will look like this:



How to handle or download dependency Jar using Maven

Before we start writing a Maven code, we need to add the general dependencies like [TestNG](https://www.javatpoint.com/testng-tutorial) and [Selenium](https://www.javatpoint.com/selenium-tutorial) in the pom.xml file.

So for this, we will follow the below process:

* Automatically we get the Pom.xml file within the same project.
* To download the dependency jar for **TestNG** and **Selenium**, we should write a dependency code for all the tools in the Pom.xml file.
* To get the dependency code, go to the Google search and type "**TestNG Maven dependency**", and click on the given link.  
  
* And copy the dependency code of TestNG and paste in the pom.xml file.  
  
* In the **scope** section, change the value from **test** to **compile** and final dependency code will look like this:

**<dependency>**

**<groupId>**org.testng**</groupId>**

**<artifactId>**testng**</artifactId>**

**<version>**6.8**</version>**

**<scope>**compile**</scope>**

**</dependency>**

Then, we will add the Maven dependency for Selenium, so for this, we will follow the same process as before:

**Getting Apache POI library**

Apache POI is the pure Java API for reading and writing Excel files in both formats XLS (Excel 2003 and earlier) and XLSX (Excel 2007 and later). To use Apache POI in your Java project:

**For non-Maven projects:**

* **For Maven projects:** Add the following dependency to your project’s pom.xml file:
  + For Excel 2003 format only:

|  |  |
| --- | --- |
|  | <dependency>      <groupId>org.apache.poi</groupId>      <artifactId>poi</artifactId>      <version>VERSION</version>  </dependency> |

* + For Excel 2007 format:

|  |  |
| --- | --- |
|  | <dependency>  <groupId>org.apache.poi</groupId>  <artifactId>poi-ooxml</artifactId>  <version>VERSION</version>  </dependency> |

* + The latest stable version of Apache POI is 3.11 (at the time of writing this tutorial).

**The Apache POI API Basics**

There are two main prefixes which you will encounter when working with Apache POI:

* **HSSF**: denotes the API is for working with Excel 2003 and earlier.
* **XSSF**: denotes the API is for working with Excel 2007 and later.

And to get started the Apache POI API, you just need to understand and use the following 4 interfaces:

* **Workbook**: high level representation of an Excel workbook. Concrete implementations are: **HSSFWorkbook**and **XSSFWorkbook**.
* **Sheet**: high level representation of an Excel worksheet. Typical implementing classes are **HSSFSheet**and **XSSFSheet**.
* **Row**: high level representation of a row in a spreadsheet. **HSSFRow**and **XSSFRow**are two concrete classes.
* **Cell**: high level representation of a cell in a row. **HSSFCell**and **XSSFCell**are the typical implementing classes.

Now, let’s walk through some real-life examples.

## **Reading from Excel File Examples**

Suppose you want to read an Excel file whose content looks like the following screenshot:

This spreadsheet contains information about books (title, author and price).

### **A Simple Example to Read Excel File in Java**

Here’s a dirty example that reads every cell in the first sheet of the workbook and prints out values in every cell, row by row:

package net.codejava.excel;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

import java.util.Iterator;

import org.apache.poi.ss.usermodel.Cell;

import org.apache.poi.ss.usermodel.Row;

import org.apache.poi.ss.usermodel.Sheet;

import org.apache.poi.ss.usermodel.Workbook;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

/\*\*

 \* A dirty simple program that reads an Excel file.

 \* @author www.codejava.net

 \*

 \*/

public class SimpleExcelReaderExample {

    public static void main(String[] args) throws IOException {

        String excelFilePath = "Books.xlsx";

        FileInputStream inputStream = new FileInputStream(new File(excelFilePath));

        Workbook workbook = new XSSFWorkbook(inputStream);

        Sheet firstSheet = workbook.getSheetAt(0);

        Iterator<Row> iterator = firstSheet.iterator();

        while (iterator.hasNext()) {

            Row nextRow = iterator.next();

            Iterator<Cell> cellIterator = nextRow.cellIterator();

            while (cellIterator.hasNext()) {

                Cell cell = cellIterator.next();

                switch (cell.getCellType()) {

                    case Cell.CELL\_TYPE\_STRING:

                        System.out.print(cell.getStringCellValue());

                        break;

                    case Cell.CELL\_TYPE\_BOOLEAN:

                        System.out.print(cell.getBooleanCellValue());

                        break;

                    case Cell.CELL\_TYPE\_NUMERIC:

                        System.out.print(cell.getNumericCellValue());

                        break;

                }

                System.out.print(" - ");

            }

            System.out.println();

        }

        workbook.close();

        inputStream.close();

    }

}

Get Value from a cell

private Object getCellValue(Cell cell) {

    switch (cell.getCellType()) {

    case Cell.CELL\_TYPE\_STRING:

        return cell.getStringCellValue();

    case Cell.CELL\_TYPE\_BOOLEAN:

        return cell.getBooleanCellValue();

    case Cell.CELL\_TYPE\_NUMERIC:

        return cell.getNumericCellValue();

    }

    return null;

}

**Reading Other Information**

* Get a specific sheet:

|  |  |
| --- | --- |
|  | Sheet thirdSheet = workbook.getSheetAt(2); |

* Get sheet name:

|  |  |
| --- | --- |
|  | String sheetName = sheet.getSheetName(); |

* Get total number of sheets in the workbook:

|  |  |  |  |
| --- | --- | --- | --- |
|  | int numberOfSheets = workbook.getNumberOfSheets();   * Get sheet by name:  |  |  | | --- | --- | |  | Sheet sheet = sheet.getSheetIndex(name); | |

XSSFSheet sheet = workbook.getSheetAt(index);

XSSFRow row=sheet.getRow(0);

**for**(**int** i=0;i<row.getLastCellNum();i++){

//System.out.println(row.getCell(i).getStringCellValue().trim());

**if**(row.getCell(i).getStringCellValue().trim().equals(colName.trim()))

col\_Num=i;

}

sheet = workbook.getSheetAt(index);

row = sheet.getRow(rowNum-1);

**if**(row==**null**)

**return** "";

cell = row.getCell(col\_Num);