Explication du Page Object Model (POM) et des Options Cucumber dans la Classe Runner

---

I. Page Object Model (POM) avec Selenium et Java

1. POM

Le **Page Object Model** (POM) est un **design pattern** utilisé en automatisation de tests pour améliorer la maintenance et la réutilisabilité du code. Il consiste à créer une classe pour chaque page ou composant de l'application, encapsulant ainsi les éléments de l'interface utilisateur et les actions associées.

---

2. Avantages du POM

- **Séparation des responsabilités** : Sépare la logique des tests (scénarios de test) de la logique des interactions avec l'interface utilisateur.
- **Réutilisabilité** : Les méthodes d'une page peuvent être réutilisées dans différents tests.
- **Maintenance facile** : Si l'interface utilisateur change, seules les classes POM associées doivent être mises à jour.

---

3. Structure du POM

La structure d'un projet POM inclut généralement :

```
src/main/java
  └── pages
        ├── LoginPage.java
        ├── HomePage.java
        └── UserManagementPage.java
src/test/java
  └── stepdefinitions
        ├── LoginSteps.java
        ├── HomeSteps.java
  └── runner
        └── TestRunner.java
  └── features
        ├── login.feature
        └── home.feature
```

---

4. Exemple de POM

LoginPage.java

```
package pages;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class LoginPage {
WebDriver driver;
@FindBy(id = "username")
WebElement usernameField;

@FindBy(id = "password")
WebElement passwordField;

@FindBy(id = "loginButton")
WebElement loginButton;

public LoginPage(WebDriver driver) {
this.driver = driver;
PageFactory.initElements(driver, this); }
```

```
public void enterUsername(String username) {
usernameField.sendKeys(username);
}
public void enterPassword(String password) {
passwordField.sendKeys(password);
}
public void clickLoginButton() {
loginButton.click();
}
 }
```

LoginSteps.java (Étapes associées au fichier Gherkin)

```
package stepdefinitions;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import pages.LoginPage;

public class LoginSteps {
WebDriver driver;
LoginPage loginPage;

@Given("l'utilisateur est sur la page de connexion")
public void user_is_on_login_page() {
driver = new ChromeDriver();
driver.get("https://example.com/login");
loginPage = new LoginPage(driver);
}

@When("il saisit le nom d'utilisateur {string}")
public void enter_username(String username) {
loginPage.enterUsername(username);
 }

@When("il saisit le mot de passe {string}")
public void enter_password(String password) {
 loginPage.enterPassword(password);
}

@When("il clique sur le bouton de connexion")
public void click_login_button() {
 loginPage.clickLoginButton();
```

```
}
@Then("l'utilisateur est redirigé vers la page d'accueil")
public void user_is_redirected_to_home_page() {
// Code de vérification
}
 }
```

II. Explication des Options Cucumber dans la Classe Runner

La classe **Runner** est utilisée pour exécuter les scénarios définis dans les fichiers Gherkin. Elle est configurée avec des annotations spécifiques pour indiquer où se trouvent les fichiers de test et comment les exécuter.

---

1. Exemple de Classe Runner

TestRunner.java

```
package runner;
import org.junit.runner.RunWith;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
@RunWith(Cucumber.class)
@CucumberOptions (
features = "src/test/java/features", // Chemin vers les fichiers .feature
glue = "stepdefinitions", // Chemin vers les classes Step Definitions
tags = "@Login",  // Exécute uniquement les scénarios avec ce tag
plugin = { "pretty", // Affiche les étapes dans la console
            "html:target/cucumber-reports.html",  // Génère un rapport HTML
"json:target/cucumber.json", // Génère un rapport
JSON "junit:target/cucumber.xml" // Génère un rapport JUnit XML },
monochrome = true, // Améliore la lisibilité des logs dans la console
```

```
dryRun = false // Vérifie les définitions de step sans exécuter les tests )
snippets = CucumberOptions.SnippetType.CAMELCASE // Génère des snippets en camelCas
public class TestRunner {
// Classe vide, utilisée uniquement pour exécuter les tests
}
```

---

2. Options Courantes

Option	Description
`features`	Chemin vers les fichiers `.feature` contenant les scénarios de test.
`glue`	Chemin vers les classes de Step Definitions.
`tags`	Permet de filtrer les scénarios à exécuter en fonction des tags définis dans les fichiers Gherkin.
`plugin`	Génère des rapports dans différents formats (HTML, JSON, XML).
`monochrome`	Si `true`, désactive les caractères de contrôle dans les logs pour une meilleure lisibilité.
`dryRun`	Si `true`, vérifie que toutes les étapes dans les fichiers `.feature` ont une définition.
`snippets`	Détermine le style des snippets générés pour les méthodes de Step Definitions.

3. Utilisation des Tags

Les **tags** permettent de filtrer les scénarios. Par exemple :

- `@Login` : Exécute uniquement les scénarios marqués par `@Login`.
- `@SmokeTest and not @Skip` : Exécute les tests `@SmokeTest`, mais exclut ceux marqués `@Skip`.
- **snippets = CAMELCASE**, les méthodes générées respectent les conventions Java, améliorant ainsi la lisibilité et la cohérence du code.

- **POM** améliore la structuration et la maintenance du code pour les projets Selenium.
- Les **options Cucumber** dans la classe Runner permettent une exécution flexible et bien structurée des tests.