Interview Questions Automation

1. Explain the oops concept used in framework

Answer:

Object-Oriented Programming (OOP) is a programming paradigm that encourages the organization of code into objects, which are instances of classes. In the context of building a software framework, OOP principles are often applied to create a modular, maintainable, and extensible architecture. Here are some OOP concepts commonly used in framework development:

1. **Classes and Objects**: In OOP, a class is a blueprint or template for creating objects. A framework often defines various classes to represent different components or functionalities. For example, in a web application framework, you might have classes for handling HTTP requests, database connections, and user authentication. Objects are instances of these classes.

2. **Encapsulation**: Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on that data within a single unit, i.e., a class. This helps in hiding the internal implementation details of a class from the outside world. Frameworks use encapsulation to expose a clean and consistent API while keeping the implementation details hidden.

3. **Inheritance**: Inheritance allows one class to inherit properties and methods from another class. Frameworks often use inheritance to create a hierarchy of classes that share common functionality. For instance, a framework might have a base "Controller" class that other controllers inherit from, inheriting common methods and behaviors.

4. **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common base class. This concept is often used in frameworks to enable flexibility. For example, a framework might have a "Plugin" system where different plugins can extend the framework's functionality in a consistent way.

5. **Abstraction**: Abstraction involves simplifying complex systems by modeling classes based on their essential properties and behaviors. Frameworks abstract away low-level details, providing developers with high-level interfaces to work with. For instance, a database abstraction layer in a framework allows developers to work with databases without needing to write raw SQL queries.

6. **Composition**: Composition is the concept of building complex objects by combining simpler objects. Frameworks use composition to create modular and reusable components. For example, a web framework might compose a web page using components like headers, footers, and content areas.

7. **Interfaces**: Interfaces define a contract that classes must adhere to. Frameworks use interfaces to ensure that classes implementing a specific interface provide certain methods or

behaviors. This allows for consistency and helps developers understand how to use various framework components.

8. **Dependency Injection**: Dependency injection is a design pattern used in OOP to provide objects with their dependencies (e.g., services or components) rather than letting them create their dependencies. Frameworks use dependency injection to facilitate loose coupling between components, making it easier to replace or extend functionality.

9. **Design Patterns**: Frameworks often incorporate well-known design patterns like the Singleton pattern, Factory pattern, Observer pattern, and others to solve common architectural problems efficiently. These patterns help ensure a clean and maintainable codebase.

10. **Modularity and Reusability**: OOP encourages breaking down a system into smaller, reusable modules or classes. Frameworks are designed with modularity in mind to make it easy to extend and reuse components in various parts of an application.

In summary, OOP concepts are fundamental to the design and development of software frameworks. They help create organized, maintainable, and extensible architectures that simplify complex software development tasks. By applying OOP principles, framework developers can provide developers with powerful tools and abstractions for building applications efficiently.

2. Explain page object model and how it is different from page factory

Answer:

The Page Object Model (POM) and Page Factory are design patterns used in test automation, particularly for Selenium WebDriver in web application testing.

Page Object Model (POM):
POM is a design pattern that promotes the abstraction of web pages into separate classes. Each class represents a page or a component, encapsulating its locators and methods to interact with elements. This abstraction enhances test maintainability and readability by separating test code from the page structure.

Example:

```
public class LoginPage {
By usernameField = By.id("username");
By passwordField = By.id("password");
By loginButton = By.id("loginButton");

WebDriver driver;

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

```
    public void enterUsername(String username) {
        driver.findElement(usernameField).sendKeys(username);
    }

    public void enterPassword(String password) {
        driver.findElement(passwordField).sendKeys(password);
    }

    public void clickLoginButton() {
        driver.findElement(loginButton).click();
    }
}
```

Page Factory:
Page Factory is an extension of POM that uses annotations to initialize elements lazily, enhancing performance. It allows automatic initialization of web elements when you create an instance of a page class. This reduces code redundancy by eliminating the need to locate elements explicitly.

Example:

```
    public class LoginPage {
    @FindBy(id = "username")
    private WebElement usernameField;

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

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

    public LoginPage(WebDriver 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();
    }
}
```

Difference:
- POM uses explicit element initialization, while Page Factory uses annotations.
- Page Factory reduces code for element initialization.
- Page Factory may improve performance due to lazy loading.
- Both patterns aim to improve test maintainability by abstracting web page interactions, but Page Factory streamlines the process by reducing boilerplate code.

//Boilerplate code refers to sections of code that are repetitive, standard, or required in multiple places within a program but don't directly contribute to the unique functionality of the application. This code is often necessary to set up the environment, initialize components, or handle common tasks but can be considered repetitive and verbose. Developers use design patterns and abstractions to reduce boilerplate code and make their codebase more concise and maintainable.

In the context of test automation with Selenium WebDriver and Page Object Model (POM), here's an example of boilerplate code without Page Factory:

```java
WebDriver driver = new ChromeDriver();
driver.manage().window().maximize();

LoginPage loginPage = new LoginPage(driver);
loginPage.enterUsername("user123");
loginPage.enterPassword("password123");
loginPage.clickLoginButton();

driver.quit();
```

In this code, setting up the WebDriver, maximizing the window, and creating an instance of the LoginPage class is considered boilerplate code because it's common setup and teardown code that appears in many test scripts.

When using Page Factory, some of this boilerplate code is reduced, as the elements are automatically initialized:

```java
WebDriver driver = new ChromeDriver();
driver.manage().window().maximize();

LoginPage loginPage = PageFactory.initElements(driver, LoginPage.class);
```

```
loginPage.enterUsername("user123");
loginPage.enterPassword("password123");
loginPage.clickLoginButton();

driver.quit();
```

In this modified code, Page Factory reduces the repetitive element initialization code, making the test script cleaner and more concise.

3. Explain testNG listener , Anyone listener explain it?

Answer:

TestNG (Test Next Generation) is a popular testing framework in Java used for writing and executing automated tests. TestNG listeners are components that allow you to customize and extend the behavior of your test execution. They provide hooks or callbacks at various stages of test execution, allowing you to perform actions or gather information before, during, or after tests.

Here are some key points about TestNG listeners:

1. **Customization**: TestNG listeners enable you to customize and extend TestNG's default behavior to meet your specific testing needs.

2. **Callbacks**: Listeners provide callback methods that are invoked at specific events during the test execution lifecycle. These events include suite, test, class, method, and more.

3. **Common Use Cases**: TestNG listeners are commonly used for logging, reporting, taking screenshots on test failure, data-driven testing, setting up and tearing down test environments, and more.

4. **Implementing Listeners**: To create a TestNG listener, you need to implement one or more listener interfaces provided by TestNG, such as `ITestListener`, `IInvokedMethodListener`, `ISuiteListener`, etc., and override their callback methods.

5. **Configuration**: Listeners can be configured in your TestNG XML configuration file, where you specify which listeners to use and associate them with test classes, methods, or suites.

Here's an example of a simple TestNG listener that logs messages before and after each test method execution:

```java
import org.testng.ITestListener;
import org.testng.ITestResult;

public class MyTestListener implements ITestListener {
```

```
    @Override
    public void onTestStart(ITestResult result) {
        System.out.println("Test Started: " + result.getName());
    }

    @Override
    public void onTestSuccess(ITestResult result) {
        System.out.println("Test Passed: " + result.getName());
    }

    @Override
    public void onTestFailure(ITestResult result) {
        System.out.println("Test Failed: " + result.getName());
        // Implement screenshot capture or other actions on failure.
    }

    // Other callback methods for different events can be implemented here.
}
```

To use this custom listener, you would configure it in your TestNG XML file:

```xml
<suite name="MyTestSuite">
    <listeners>
        <listener class-name="com.example.MyTestListener" />
    </listeners>
    <!-- Test classes and test methods go here -->
</suite>
```

When you run your tests, TestNG will invoke the callback methods defined in your listener at the appropriate points in the test execution lifecycle, allowing you to customize the behavior of your tests as needed.

Certainly, here are the names of some commonly used TestNG listeners, each in one line:

1. **ITestListener**: Provides callbacks for test-level events such as test start, test success, test failure, etc.

2. **ISuiteListener**: Provides callbacks for suite-level events like suite start, suite finish, etc.

3. **IInvokedMethodListener**: Provides callbacks for method-level events such as method invocation, success, failure, etc.

4. **IReporter**: Allows custom reporting by providing a callback to generate custom test reports.

5. **ITransformer**: Allows you to transform test parameters before they are passed to test methods.

6. **IAnnotationTransformer**: Lets you modify annotations on test classes and methods dynamically.

7. **IHookable**: Allows you to define custom logic for hook methods (e.g., `@BeforeMethod`, `@AfterMethod`) using `IHookable` interface.

8. **IConfigurable**: Allows you to define custom configuration methods for test classes using `@Configuration` annotations.

9. **IExecutionListener**: Provides callbacks for the start and finish of the entire test execution process.

10. **IMethodInterceptor**: Allows you to change the order of test methods execution or filter methods.

These listeners allow you to extend and customize TestNG's behavior to suit your testing needs by implementing their respective interfaces and overriding their callback methods.

4. How to run test cases in parallel in testNG , method vs Class parallel

Answer:

TestNG provides several ways to run test cases in parallel, including method-level and class-level parallel execution. The choice between method and class parallelism depends on your testing requirements and the architecture of your test suite.

Method-Level Parallelism:
In method-level parallelism, individual test methods are executed concurrently in separate threads. This is useful when your test methods are independent and can run in parallel without affecting each other. Here's how to configure method-level parallelism in TestNG:

```xml
<suite name="MyTestSuite" parallel="methods" thread-count="3">
    <!-- Test classes and their methods go here -->
</suite>
```

In this example, the `parallel` attribute is set to "methods," and `thread-count` specifies the number of threads to use for parallel execution. Each test method within the suite will run in parallel up to the specified thread count.

Class-Level Parallelism:
Class-level parallelism involves running entire test classes concurrently. This is useful when you have test methods within the same class that depend on each other's state or resources and should not run concurrently. Here's how to configure class-level parallelism:

```xml
<suite name="MyTestSuite" parallel="classes" thread-count="2">
    <!-- Test classes go here -->
</suite>
```

In this example, the `parallel` attribute is set to "classes," and `thread-count` specifies the number of threads to use for parallel execution. Each test class within the suite will run in parallel up to the specified thread count, but the methods within each class will run sequentially.

Mixed Parallelism:
You can also use a combination of both method-level and class-level parallelism by specifying different parallel attributes for different suites within the same TestNG XML file. For example, you might run some suites with method-level parallelism and others with class-level parallelism.

```xml
<suite name="Suite1" parallel="methods" thread-count="3">
    <!-- Test classes and their methods go here -->
</suite>

<suite name="Suite2" parallel="classes" thread-count="2">
    <!-- Test classes go here -->
</suite>
```

This allows you to optimize parallelism based on your specific testing requirements and the characteristics of your test suite.

In summary, TestNG provides flexibility in running test cases in parallel, allowing you to choose between method-level and class-level parallelism, or even a combination of both, based on your testing needs and the dependencies between your test methods and classes.

5. Explain Overriding and Overloading?

Answer

TestNG provides several ways to run test cases in parallel, including method-level and class-level parallel execution. The choice between method and class parallelism depends on your testing requirements and the architecture of your test suite.

Method-Level Parallelism:

In method-level parallelism, individual test methods are executed concurrently in separate threads. This is useful when your test methods are independent and can run in parallel without affecting each other. Here's how to configure method-level parallelism in TestNG:

```xml
<suite name="MyTestSuite" parallel="methods" thread-count="3">
    <!-- Test classes and their methods go here -->
</suite>
```

In this example, the `parallel` attribute is set to "methods," and `thread-count` specifies the number of threads to use for parallel execution. Each test method within the suite will run in parallel up to the specified thread count.

Class-Level Parallelism:
Class-level parallelism involves running entire test classes concurrently. This is useful when you have test methods within the same class that depend on each other's state or resources and should not run concurrently. Here's how to configure class-level parallelism:

```xml
<suite name="MyTestSuite" parallel="classes" thread-count="2">
    <!-- Test classes go here -->
</suite>
```

In this example, the `parallel` attribute is set to "classes," and `thread-count` specifies the number of threads to use for parallel execution. Each test class within the suite will run in parallel up to the specified thread count, but the methods within each class will run sequentially.

Mixed Parallelism:
You can also use a combination of both method-level and class-level parallelism by specifying different parallel attributes for different suites within the same TestNG XML file. For example, you might run some suites with method-level parallelism and others with class-level parallelism.

```xml
<suite name="Suite1" parallel="methods" thread-count="3">
    <!-- Test classes and their methods go here -->
</suite>

<suite name="Suite2" parallel="classes" thread-count="2">
    <!-- Test classes go here -->
</suite>
```

This allows you to optimize parallelism based on your specific testing requirements and the characteristics of your test suite.

In summary, TestNG provides flexibility in running test cases in parallel, allowing you to choose between method-level and class-level parallelism, or even a combination of both, based on your testing needs and the dependencies between your test methods and classes.

Question : why multiple inheritance are not supported in java

Answer:

Multiple inheritance is the concept of a class inheriting properties and behaviors from more than one superclass or parent class. While multiple inheritance is supported in some programming languages like C++, Java deliberately chose not to support it due to several design considerations. Here are some reasons why multiple inheritance is not supported in Java:

1. **Ambiguity**: Multiple inheritance can lead to a situation known as the "diamond problem," where a class inherits from two classes that have a common ancestor. This can create ambiguity when calling methods or accessing attributes, as it's unclear which superclass's method or attribute should be used.

2. **Complexity**: Multiple inheritance increases the complexity of the language and makes it challenging to determine the order of method resolution when conflicts occur. This complexity can lead to hard-to-maintain code and make it more challenging to understand program behavior.

3. **Maintenance Issues**: Code maintenance becomes problematic when changes are made to one superclass, potentially affecting multiple subclasses. This can lead to unintended consequences and make codebases less maintainable.

4. **Favor Composition over Inheritance**: Java promotes the use of composition over inheritance. By using interfaces and composition, developers can achieve the benefits of code reuse and flexibility without the complications of multiple inheritance.

5. **Interface-based Inheritance**: Java offers a form of multiple inheritance through interfaces. A class can implement multiple interfaces, allowing it to inherit method signatures from multiple sources without the complexities associated with inheriting method implementations.

6. **Clarity and Readability**: Java emphasizes code readability and maintainability. Multiple inheritance can make code less clear and increase the risk of naming conflicts and errors.

To mitigate the issues associated with multiple inheritance, Java encourages developers to use single inheritance with classes and leverage interfaces for achieving multiple inheritance-like behavior. This approach allows for code reuse and flexibility while maintaining a clear and manageable codebase.

Question :Suppose there are 500 test given to you, how would you choose automation tech stack?

Answer

Choosing an automation tech stack for a set of 500 tests involves careful consideration of various factors, including the nature of the tests, project requirements, team expertise, and long-term maintainability. Here's a systematic approach to making this decision:

1. **Understand Test Requirements**:

   - Analyze the types of tests (e.g., unit, integration, functional, regression, load) and their complexity.

   - Identify the platforms and browsers you need to support.

   - Consider the need for parallel execution and distributed testing.

2. **Evaluate Technology Options**:

   - Explore available test automation frameworks and tools for your programming language.

   - Consider the following factors:

     - Community support and active development.

     - Compatibility with the application under test (AUT).

     - Test execution speed and reliability.

     - Support for various testing types (e.g., UI, API, performance).

     - Reporting and logging capabilities.

     - Integration with Continuous Integration (CI) tools.

     - Licensing costs, if any.

- Research popular programming languages and libraries for test automation (e.g., Selenium, Appium, JUnit, TestNG, Cucumber, JBehave).

3. **Assess Team Skillset**:

- Evaluate the existing skills and expertise of your team members in programming languages and frameworks.

- Consider training or upskilling opportunities if the chosen stack requires new skills.

4. **Pilot Testing**:

- Conduct a pilot project to evaluate the selected tech stack with a smaller subset of tests.

- Assess its suitability, performance, and maintainability.

5. **Scalability**:

- Ensure that the chosen stack can scale to accommodate the large number of tests effectively.

- Evaluate the ability to run tests in parallel and distribute them across multiple machines or environments.

6. **Maintainability**:

- Prioritize maintainability by choosing a stack that enables easy test maintenance and updates as the application evolves.

7. **CI/CD Integration**:

- Ensure that the chosen tech stack seamlessly integrates with your CI/CD pipeline for automated testing and reporting.

8. **Reporting and Analysis**:

- Consider tools and frameworks that provide comprehensive reporting and analysis capabilities, as analyzing results from 500 tests can be challenging.

9. **Cost Considerations**:

- Evaluate the budget and licensing costs associated with the selected tools and frameworks.

10. **Long-Term Viability**:

- Choose a tech stack that is likely to remain viable and supported in the long run.

11. **Documentation and Community**:

- Check the availability of documentation, tutorials, and a supportive community for the chosen stack.

12. **Security and Compliance**:

- Ensure that the tech stack aligns with security and compliance requirements, especially if your application deals with sensitive data.

13. **Feedback and Continuous Improvement**:

- Encourage feedback from the testing team to continuously improve the automation strategy and tech stack.

Based on the above considerations, you can make an informed decision about the automation tech stack that best suits your project's needs, balancing factors like functionality, scalability, maintainability, and team expertise. Remember that the goal is to create a robust and efficient automation framework that helps you effectively manage and execute your 500 tests.

Question: How selenium works ? what happens when we I do Webdriver driver = new ChromeDriver()

Answer: Selenium is a popular open-source framework for automating web browsers. It allows you to write scripts in various programming languages to interact with web pages, simulate user actions, and perform automated testing. When you execute the line of code `WebDriver driver = new ChromeDriver();`, several things happen:

1. **Instantiating WebDriver**:

- `WebDriver` is an interface in Selenium that defines a set of methods for interacting with web browsers.

- `ChromeDriver` is a specific implementation of the `WebDriver` interface for the Google Chrome browser.

- When you create an instance of `ChromeDriver`, you are essentially telling Selenium to use the Chrome browser for automation.

2. **Initializing the Chrome Browser**:

- The `ChromeDriver` constructor initializes and launches the Chrome browser on your system. This means that if Chrome is not already open, a new instance of Chrome will be launched.

- The `WebDriver` interface allows you to interact with the browser through your automation script.

3. **Communication with the WebDriver Server**:

- Selenium operates on a client-server architecture. When you create a `ChromeDriver` instance, it acts as a client that communicates with the WebDriver server.

- The WebDriver server is responsible for controlling the browser and translating your script's commands into actions performed by the browser.

- The client (your script) and server communicate using the WebDriver protocol, which is implemented by the WebDriver server for each supported browser (e.g., Chrome, Firefox, Safari).

4. **Session Creation**:

- A new "session" is established between your script and the WebDriver server. This session represents the interaction between your script and the browser.

- The session ID is generated and used for subsequent communication to identify the session.

5. **Interaction and Automation**:

- Once the browser is launched and the session is established, you can use the `driver` object to automate various actions, such as navigating to web pages, interacting with web elements (e.g., clicking buttons, filling forms), and extracting information from web pages.

- Your automation script sends commands to the WebDriver server, which in turn directs the browser to perform these actions.

6. **Execution and Response**:

- As your script executes commands through the `driver` object, the browser carries out the actions accordingly.

- The WebDriver server sends responses and information back to your script, such as success/failure status, element identification, and more.

7. **Cleanup and Termination**:

- After your automation script is finished, you should explicitly close the browser using `driver.quit()` to release system resources and end the WebDriver session gracefully.

In summary, when you create a `ChromeDriver` instance with `WebDriver driver = new ChromeDriver();`, you are initializing the Chrome browser, establishing a communication session between your script and the WebDriver server, and enabling your script to automate browser interactions. Selenium provides a powerful platform for web automation and testing across various browsers.