

HOW TO LEARN FROM THIS DOCUMENT, watch here⇒ <https://youtu.be/PG5FRtuCRIY>

### Learn Core Concepts

[2.1 JAVA String Interview Q&A](#)

[2.2 Java Array Interview Q&A](#)

[2.3 Java List Interview Q&A](#)

[2.4 Java MAP Interview Q&A](#)

[2.5 Basics of Object-Oriented Programming \(OOPS\)](#)

[Inheritance](#)

[Polymorphism](#)

[Abstraction](#)

[Encapsulation](#)

### Core Java Interview Q&A

### CODING

[1. Top 20 Java Coding Q&A : Most Frequently Asked](#)

[2. Must Know Java Coding Q&A : Second Most Asked](#)

[3. JAVA CODING INTERVIEW Q&A BANK: 240 Coding Q&A : Commonly Asked](#)

### Java Interview Q&A

[3. Throws & TryCatch with Example From Test Automation](#)

[4. Explain Java Over-riding with an example from Test Automation](#)

[5. How to Implement multithreading in Selenium?](#)

[6. Top 5 things about Java Multithreading](#)

[7. Give a real time scenario example for Polymorphism/method overloading ? Interview qus selenium webdriver/java/method overloading](#)

[8. How to use Java Interface in Test Automation ?](#)

[9. Final vs Finally vs Finalise with Selenium Examples](#)

[10. Method vs Constructor in Java](#)

[11. Common Methods in Pattern](#)

[12. Common Methods in Matcher](#)

[13. Constructor Interview Q&A](#)

[▲ Explain the difference between a default constructor and a parameterized constructor.](#)

[▲ Can constructors be overloaded in Java? Provide an example.](#)

[▲ What is the role of the super keyword in constructors?](#)

[▲ Can a constructor call another constructor in the same class? How?](#)

[▲ How do you use a constructor to initialize a WebDriver in Selenium?](#)

### Test Automation-Java Scenario Based Interview Q&A

[Overloading:](#)

[Overriding:](#)

[Encapsulation:](#)

[Inheritance:](#)

[Enums:](#)

[Generics:](#)

[Strings:](#)

[Array:](#)

[List:](#)

[Map:](#)

[INTERFACE](#)

=====

[More Q&A](#)

[Commonly Asked Java Interview Q&A 2024](#)

Sidharth Shukla

## Section-01

### Learn Core Concepts

2.1 [JAVA String Interview Q&A](#)

2.2 [Java Array Interview Q&A](#)

2.3 [Java List Interview Q&A](#)

2.4 [Java MAP Interview Q&A](#)

### 2.5 Basics of Object-Oriented Programming (OOPS)

Object-Oriented Programming (OOP) is a fundamental programming paradigm used in software development, defined by its use of classes and objects.

It's built on four main principles: Inheritance, Polymorphism, Abstraction, and Encapsulation.

These principles not only help in creating structured and reusable code but also make it easier to understand, maintain, and modify.

### Inheritance

Inheritance allows one class to inherit the properties and methods of another class. It's a way to form a hierarchy between classes, promoting code reusability.

Example:

```
class Vehicle {
```

```

        public void startEngine() {

            System.out.println("Engine started");

        }
    }

class Car extends Vehicle {

    public void openTrunk() {

        System.out.println("Trunk opened");

    }

}

public class Main {

    public static void main(String[] args) {

        Car myCar = new Car();

        myCar.startEngine(); // Inherited method

        myCar.openTrunk(); // Own method

    }

}

```

In this Java example, Car inherits from Vehicle.

Car can use the startEngine method from Vehicle, demonstrating inheritance.

## Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It's the ability of multiple object types to implement the same functionality, which can be achieved either by method overloading or method overriding.

Example:

```

class Bird {

```

```
    public void sing() {  
        System.out.println("Bird is singing");  
    }  
}  
  
class Sparrow extends Bird {  
    public void sing() {  
        System.out.println("Sparrow is singing");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Bird myBird = new Sparrow();  
        myBird.sing(); // Outputs: Sparrow is singing  
    }  
}
```

Here, `Sparrow` overrides the `sing` method of `Bird`. Despite referring to `Sparrow` with a `Bird` reference, the overridden method in `Sparrow` is called.

## Abstraction

Abstraction is the concept of hiding complex implementation details and showing only the necessary features of an object. It can be achieved using abstract classes and interfaces.

Example:

```
abstract class Animal {  
    abstract void makeSound();  
}
```

```
        public void eat() {  
            System.out.println("Animal is eating");  
        }  
    }  
}
```

```
class Dog extends Animal {  
    public void makeSound() {  
        System.out.println("Bark");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Animal myDog = new Dog();  
        myDog.makeSound(); // Outputs: Bark  
        myDog.eat(); // Inherited method  
    }  
}
```

Animal is an abstract class that provides a method `makeSound()`.

Dog provides the specific implementation of this method.

## Encapsulation

Encapsulation is the technique of bundling data (variables) and methods that act on the data into a single unit, often called a class, and restricting access to some of the object's components.

```
class BankAccount {  
    private double balance;
```

```
public void deposit(double amount) {  
    if (amount > 0) {  
        balance += amount;  
    }  
}  
  
public void withdraw(double amount) {  
    if (amount <= balance) {  
        balance -= amount;  
    }  
}  
  
public double getBalance() {  
    return balance;  
}  
}  
  
public class Main {  
    public static void main(String[] args) {  
        BankAccount account = new BankAccount();  
        account.deposit(1000);  
        account.withdraw(500);  
        System.out.println("Balance: " + account.getBalance());  
    }  
}
```

In this example, the balance of the `BankAccount` is kept private. It can only be modified through the `deposit` and `withdraw` methods and read through the `getBalance` method, showcasing encapsulation.

## Section-02

## SECTION -02

=====CLICK THE HYPERLINK BELOW=====

**[Core Java Interview Q&A](#)**

=====




## Section-03

===CLICK THE HYPERLINKS=====

### CODING

1. [Top 20 Java Coding Q&A](#) : Most Frequently Asked
2. [Must Know Java Coding Q&A](#) : Second Most Asked
3. [JAVA CODING INTERVIEW Q&A BANK: 240 Coding Q&A](#) : Commonly Asked

 **Trying for an SDET role?** Challenge yourself with these coding problems!

**Avoid GPT and Google**—try solving these on your own! 💪

=> **Reverse** an Array or String 

=> **Two Pointer Problem:** Remove duplicates from a sorted array in-place.

=> **Sliding Window Problem:** Find two numbers in a sorted array that sum up to a specific target.

=> **Find Occurrences:** Count the occurrences of elements in an array.

=> **Reverse Words in a Sentence:** How would you do it?

=> **Draw a Star Pyramid** using Java

=> **Classic Problems:** Palindrome, Anagram, Prime

=> **Find Max and Min** in an Array

=> **Parenthesis with Stack:** Given a string containing just the characters (, ), {, }, [ and ], determine if the input string is valid.

=> **Regex Pattern:** Given a list of domain names, you need to extract and return all unique subdomains. A subdomain is defined as any part of a domain that appears before the main domain and TLD (top-level domain). For example, in the domain [mail.google.com](mailto:mail.google.com), mail is a subdomain of [google.com](https://www.google.com).

=====

## Section - 04

# Java Interview Q&A

### 3. Throws & TryCatch with Example From Test Automation

While conducting interviews, I've observed that individuals are familiar with the definitions of "throws" and "try-catch" but often lack a comprehensive understanding of their usage.

Let's try to understand with examples..

In Java, try-catch blocks are used for exception handling, while throws is used in method declarations to indicate that a method might throw a specific type of exception. In the context of Selenium, here's a brief explanation:

\*\*\*\*\*

● Try-Catch:

\*\*\*\*\*

👉 Usage:

It is used to catch and handle exceptions within a specific block of code.

👉 Example:

```
try {  
    WebElement element = driver.findElement(By.id("example"));  
    element.click();  
} catch (NoSuchElementException e) {  
    System.out.println("Element not found: " + e.getMessage());  
}
```

👉 When to Use:

Use try-catch when you want to handle exceptions immediately within a particular code block.

👉 Pros:

Provides a localized way to handle exceptions, making the code more robust.

👉 Cons:

Can lead to code duplication if similar exception handling is required in multiple places.

\*\*\*\*\*

## ● Throws:

\*\*\*\*\*

### 👉 Usage:

The biggest problem is handling the exceptions needs to be done in the calling methods too.. It is used in method declarations to indicate that the method may throw a specific type of exception, and the responsibility to handle the exception is delegated to the calling method.

### 👉 Example:

```
public void clickElement() throws NoSuchElementException {  
    WebElement element = driver.findElement(By.id("example"));  
    element.click\(\);  
}
```

### 👉 When to Use:

Use throws when you want to delegate the responsibility of handling exceptions to the calling method.

### 👉 Pros:

Promotes cleaner code by separating the concern of exception handling from the method's logic.

### 👉 Cons:

Requires the calling method to handle the exception or declare it with throws.

## ● Choosing Between Try-Catch and Throws:

👉 Use try-catch when you want to handle exceptions immediately within the current block of code.

👉 Use throws when you want to delegate the responsibility of handling exceptions to the calling method.

=====

## 4. Explain Java Over-riding with an example from Test Automation

When I talk to folks in 1:1, I've noticed that many get confused about how to add new steps to the `@BeforeTest` method when it's set up in a base class.

Using `@Override` is a good way to handle this and it's a good example to mention in interviews. Just stick to relevant examples when discussing this topic during interviews and avoid giving random examples.

### ▲ 1 Overriding in Java:

Overriding is a concept in Java that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. It enables polymorphism, where the same method name can behave differently based on the object it's called on.

### ▲ Steps to implement Overriding

👉 Create a Base Test Class:

Define a base test class where you initialize your `WebDriver` instance and set up common configurations.

```
public class BaseTest {  
  
    protected WebDriver driver;
```

```
    @BeforeTest  
  
    public void setUp() {  
  
        // Initialize WebDriver
```

```
System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");  
  
driver = new ChromeDriver();  
  
driver.manage().window().maximize();  
  
}
```

```
// Other common setup methods can be added here  
  
}
```

👉 Override Setup Method:

Override the setUp() method in subclasses to provide specific setup actions for each test class

```
public class MyTest extends BaseTest {  
  
    @Override  
    @BeforeTest  
    public void setUp() {  
        // Call the setup method from the base class  
        super.setUp();  
  
        // Additional setup specific to this test class  
        // For example, navigating to a specific URL  
        driver.get("https://example.com");  
    }  
}
```

In this example, the BaseTest class contains the setUp() method annotated with @BeforeTest, which initializes the WebDriver instance.

The MyTest class extends BaseTest, and the setUp() method is overridden to provide additional setup specific to the test class.

The overridden method is also annotated with @BeforeTest to ensure it executes before any test method belonging to the <test> tag in the TestNG XML suite file.

=====

## 5. How to Implement multithreading in Selenium?

In the context of test automation with Selenium, multithreading can be a valuable approach for running multiple tests simultaneously, ultimately reducing the overall execution time.

### What is multithreading in Java ?

In Java, multithreading is a feature that allows multiple threads to exist within the context of a single process. A thread is the smallest unit of execution within a program, and it can be thought of as a “lightweight” process that runs independently of others, sharing the same memory space and resources. Multithreading enables concurrent execution of two or more parts of a program, which can improve application performance by utilizing the processing power of modern multicore CPUs more effectively. This allows for tasks to be performed simultaneously, such as updating the user interface while performing calculations in the background, improving the overall responsiveness and user experience. To create and manage threads in Java, you can either extend the Thread class or implement the Runnable interface. The Thread class provides methods for starting, stopping, and synchronizing threads, while the Runnable interface allows you to define the tasks that will be executed by the threads. Java also provides various synchronization techniques, such as locks and semaphores, to coordinate access to shared resources and prevent conflicts between threads. Properly managing and synchronizing threads is crucial to avoid issues like race conditions and deadlocks, ensuring that your application runs smoothly and efficiently.

In Java, multithreading allows multiple threads to exist within the context of a single process, such that they share the same memory and resources.

### Implement multithreading in Selenium

To implement multithreading in Selenium tests using Java, you can make use of the Thread class or the ExecutorService interface from the java.util.concurrent package.

The ExecutorService interface provides a higher-level abstraction for managing threads, making it easier to submit tasks and handle their execution.

Here’s an example that demonstrates how to run Selenium tests using multithreading with the ExecutorService interface and TestNG:

#### ▲ CODE EXAMPLE

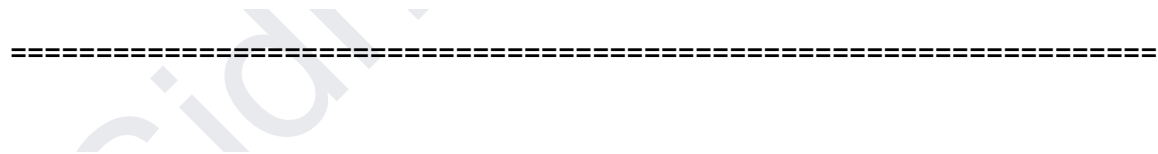
```
public class SeleniumTest {
```

```

private static final int NUM_THREADS = 5;
private static final ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
@Test
public void testGoogleSearch() {
    executor.execute() -> {
        // WebDriver code to perform Google search
    };
}
@Test
public void testAmazonSearch() {
    executor.execute() -> {
        // WebDriver code to perform Amazon search
    };
}
@Test
public void shutdown() throws InterruptedException {
    executor.shutdown();
    executor.awaitTermination(1, TimeUnit.MINUTES);
}
}

```

In this example, we create a thread pool with a fixed number of threads using `Executors.newFixedThreadPool`. Each test method submits a task to the executor, which then executes the task asynchronously in a separate thread. The shutdown method ensures that the executor service is properly shut down after the tests are executed.



## 6. Top 5 things about Java Multithreading

Here are five unique things about Java multithreading:

**Thread Class and Runnable Interface:** Java offers two ways to create threads — by extending the `Thread` class or implementing the `Runnable` interface. The `Thread`

class provides more control over thread creation and management, while the Runnable interface allows for a more flexible and object-oriented approach.

**Thread Synchronization:** Java provides multiple synchronization techniques to manage access to shared resources, such as synchronized methods and blocks, locks, and semaphores. Proper synchronization is essential to avoid issues like race conditions and deadlocks.

**Thread Pools:** Java's ExecutorService framework simplifies the creation and management of thread pools, making it easier to handle multiple concurrent tasks efficiently.

**Thread-local Variables:** Java supports thread-local variables, which are unique to each thread, allowing for more efficient use of resources and better isolation between threads.

**Thread States:** Java threads can be in one of several states, such as NEW, RUNNABLE, BLOCKED, WAITING, TIMED\_WAITING, or TERMINATED. Understanding these states and managing thread transitions is crucial for building responsive and efficient multithreaded applications.

=====

## ***7. Give a real time scenario example for Polymorphism/method overloading ? Interview qus selenium webdriver/java/method overloading***

### **Method Overloading in Java**

Method overloading is a feature in object-oriented programming languages like Java that allows a class to have multiple methods with the same name but with different parameters. These methods can perform similar or related tasks but may accept different types or numbers of parameters.

In method overloading:

1. Methods must have the same name.
2. Methods must be defined within the same class.
3. Methods must have different parameter lists (different types or different numbers of parameters).
4. The return type of the methods may or may not be the same.



When a method is invoked, the compiler determines which version of the overloaded method to call based on the number and types of arguments provided. This decision is made at compile time, and the appropriate method is then called at runtime.

Method overloading provides several benefits:

- It improves code readability and maintainability by allowing developers to use intuitive method names for different variations of a task.
- It enhances code reusability by allowing methods to perform similar tasks with different inputs.
- It provides flexibility in method design, enabling the creation of more intuitive and expressive APIs.

Overall, method overloading is a powerful tool that allows developers to write cleaner, more expressive code while providing flexibility and versatility in method design.

### Scenario: Handling Different Types of Click Actions

In Selenium test automation, you often encounter scenarios where you need to perform various types of click actions on web elements. These actions can include simple clicks, double clicks, and right clicks. By using method overloading, you can create flexible and intuitive methods to handle these different types of click actions efficiently.

```
public class ClickActionsHelper {  
  
    private WebDriver driver;  
    private Actions actions;  
  
    public ClickActionsHelper(WebDriver driver) {  
        this.driver = driver;  
        this.actions = new Actions(driver);  
    }  
  
    // Method for performing a simple click  
    public void click(WebElement element) {  
        actions.click(element).build().perform();  
    }  
  
    // Method for performing a double click  
    public void click(WebElement element, boolean doubleClick) {  
        if (doubleClick) {
```

```

        actions.doubleClick(element).build().perform();
    } else {
        click(element); // Invokes the simple click method
    }
}

// Method for performing a right click
public void click(WebElement element, boolean doubleClick, boolean
rightClick) {
    if (rightClick) {
        actions.contextClick(element).build().perform();
    } else {
        click(element, doubleClick); // Invokes the appropriate
click method based on parameters
    }
}
}

```

In this example, we have a `ClickActionsHelper` class that encapsulates methods for handling different click actions using Selenium's `Actions` class. We use method overloading to define multiple `click` methods with varying parameters:

- The first `click` method takes a `WebElement` parameter and performs a simple click action.
- The second `click` method overloads the first one and adds a boolean parameter to indicate whether it should perform a double click. Depending on the value of this parameter, it either performs a double click action or invokes the simple click method.
- The third `click` method further overloads the second one and adds another boolean parameter to indicate whether it should perform a right click. Similarly, it either performs a right click action or invokes the appropriate click method based on the parameters provided.

By using method overloading in this manner, we can handle different types of click actions with ease and readability, enhancing the maintainability and flexibility of our test automation code.

=====

## 8. How to use Java Interface in Test Automation ?

In Selenium WebDriver, Java interfaces can be used to define page objects and implement the Page Object Model (POM) design pattern.

The Page Object Model is a design pattern that encourages the separation of test logic from the UI layer by creating a separate class for each web page and defining methods to interact with elements on that page.

Here's a simple example demonstrating how to use a Java interface in Selenium WebDriver: Let's say we have a web application with a login page and a home page. We can create interfaces for each page to define the elements and actions specific to that page.

● Interface for the Login page: [LoginPage.java](#)

```
public interface LoginPage {  
    WebElement getUsernameInput();  
    void login(String username, String password);  
}
```

● Implementation of LoginPage interface: [LoginPageImpl.java](#)

```
public class LoginPageImpl implements LoginPage {  
    private WebDriver driver;  
    @FindBy(id = "username")  
    private WebElement usernameInput;  
    public LoginPageImpl(WebDriver driver) {  
        this.driver = driver;  
        PageFactory.initElements(driver, this);  
    }  
    public WebElement getUsernameInput() {  
        return usernameInput;  
    }  
    public void login(String username, String password) {  
        usernameInput.sendKeys(username);  
    }  
}
```

● With this setup, you can create instances of LoginPageImpl classes in your test classes and interact with the elements on the respective pages using the methods defined in the interfaces.

```
WebDriver driver = new ChromeDriver();  
LoginPage loginPage = new LoginPageImpl(driver);  
loginPage.login("username");  
System.out.println("Logged in user: " + loggedInUserName);  
driver.quit();  
*****
```

🔗 If you want to learn Framework Development with Advanced Java concepts refer the End to end Automation & SDET Training with advanced topics on Design Patterns, Generative AI, Pair programming, API with rest Assured & Postman, Selenium, Docker, Jenkins, GIT, Appium along with 1:1 Guidance –demo: <https://lnkd.in/g/CxnJJ7>

\*\*\*\*\*

👉 Arrange 1:1 call here for career guidance and mock interviews : <https://lnkd.in/ddayTw>

\*\*\*\*\*

## 9. Final vs Finally vs Finalise with Selenium Examples

What is the difference between the “final,” “finally,” and “finalize” keywords in Java with examples from Selenium?

### ● “final” Keyword:

In Java, the “final” keyword is used to declare a constant variable, a method that cannot be overridden, or a class that cannot be extended.

A final variable’s value cannot be changed once assigned, a final method cannot be overridden in subclasses, and a final class cannot be subclassed.

Selenium Example:

```
public class LoginPage {  
  
    // Declaring a final constant for the login URL  
    private static final String LOGIN_URL = "https://example.com/login";  
  
    public void goToLoginPage(WebDriver driver) {  
        driver.get(LOGIN_URL); // The LOGIN_URL cannot be changed  
    }  
}
```

In this example, LOGIN\_URL is declared as final, ensuring that the URL for the login page cannot be modified once assigned.

### ● “finally” Block:

The “finally” block is used in exception handling to ensure that a block of code is always executed, regardless of whether an exception is thrown or not. It is placed after the try-catch block and is useful for releasing resources or performing cleanup operations.

Selenium Example:

```
public class TestExample {  
  
    public void testMethod(WebDriver driver) {
```

```

try {
driver.get("https://example.com");
WebElement element = driver.findElement(By.id("someId"));
element.click();
} catch (NoSuchElementException e) {
System.out.println("Element NF");
} finally {
// This block will always execute, even if an exception occurs
driver.quit();
}
}
}

```

In this example, the finally block ensures that the WebDriver is closed (driver.quit()) regardless of whether the test encounters an exception.

### ● "finalize" Method:

The "finalize" method is a protected method defined in the "Object" class. It is called by the garbage collector before an object is garbage collected. It gives the object a chance to perform any necessary cleanup actions before being destroyed. But it is recommended to avoid relying on the "finalize" method for resource cleanup.

```

public class SeleniumTest {

WebDriver driver;

public SeleniumTest() {
driver = new ChromeDriver();
}

@Override
protected void finalize() throws Throwable {
try {
if (driver != null) {
driver.quit(); // Attempt to close the WebDriver before garbage collection
}
} finally {
super.finalize();
}
}
}
}

```

In this example, the finalize method attempts to quit the WebDriver before the object is garbage collected. But, using finalize for this purpose is generally discouraged.

## 10. Method vs Constructor in Java

💛 Today's Topic is Constructor VS Method in java 💚

Let's understand the differences between Constructor & Method, along with an example from automation on how we can use both in real time scenarios

👉 Purpose:

Constructor:

Used to initialize the state of an object during its creation.

Method:

Performs a specific task on an object or returns a value to the caller.

👉 Syntax:

Constructor:

Same name as the class, no return type.

Method:

Any name, must have a return type or void.

👉 Invocation:

Constructor:

Implicitly invoked by the system when an object is created.

Method:

Explicitly invoked by the programmer or other methods.

👉 Overloading and Overriding:

Constructor:

Can be overloaded but not overridden by subclasses.

Method:

Can be both overloaded and overridden by subclasses.

## 👉 Examples

▲ // TestAutomation() constructor to initialize the data members

```
class TestAutomation {  
TestAutomation() //THIS IS CONSTRUCTOR  
{  
// set the system property for Chrome driver System.setProperty("webdriver.chrome.driver",  
"C:\\chromedriver.exe");  
// create a new instance of Chrome driver  
driver = new ChromeDriver();  
// assign the url to be tested  
url = "https://www.bing.com";  
}
```

▲ // launchBrowser() METHOD to launch the browser and navigate to the url

```
public  
void launchBrowser()  
{  
// maximize the browser window  
driver.manage().window().maximize();  
// navigate to the url  
driver.get(url);  
}  
}
```

▲ IMPLEMENTATION:

```
// create an object of TestAutomation using the constructor  
TestAutomation test = new TestAutomation();  
  
// invoke the launchBrowser() method on the object  
test.launchBrowser();
```

In summary, the TestAutomation class has a constructor (TestAutomation()) for initializing data members and a method (launchBrowser()) for browser handling.

## 💚 PRO-TIP FOR SDETs:

A constructor typically has a fixed time and space complexity,  $O(1)$ , as it is only called once when an object is instantiated, whereas methods can have varying time and space complexities depending on the logic, inputs, and number of invocations

## 11. Common Methods in **Pattern**

- **compile(String regex)**: Compiles the given regular expression into a pattern.
- **compile(String regex, int flags)**: Compiles the given regular expression into a pattern with the specified flags, such as **Pattern.CASE\_INSENSITIVE**.
- **split(CharSequence input)**: Splits the given input sequence around matches of this pattern.

## 12. Common Methods in **Matcher**

- **find()**: Attempts to find the next subsequence of the input sequence that matches the pattern.
- **group()**: Returns the matched subsequence.
- **matches()**: Attempts to match the entire input sequence against the pattern.
- **start()**: Returns the start index of the last match.
- **end()**: Returns the end index of the last match.
- **replaceAll(String replacement)**: Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
- **replaceFirst(String replacement)**: Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
- **groupCount()**: Returns the number of capturing groups in the pattern.

## 13. Constructor Interview Q&A

Constructor is one of the most important topic for aspiring Automation QA or SDET Job role, in this post we will look into some of the conceptual questions on Java-Constructor.

▲ Explain the difference between a default constructor and a parameterized constructor.

Answer:

A default constructor is a no-argument constructor that is automatically generated by the compiler if no other constructors are defined. It initializes the object with default values.



A parameterized constructor is one that takes arguments to initialize an object with specific values provided during object creation.

▲ Can constructors be overloaded in Java? Provide an example.

Answer:

Yes, constructors can be overloaded in Java. This means you can have multiple constructors in a class with different parameter lists.

Example:

```
public class WebDriverSetup {
    private String browser;
    private String driverPath;

    // Default constructor
    public WebDriverSetup() {
        this.browser = "Chrome";
        this.driverPath = "/path/to/chromedriver";
    }

    // Parameterized constructor
    public WebDriverSetup(String browser, String driverPath) {
        this.browser = browser;
        this.driverPath = driverPath;
    }
}
```

▲ What is the role of the super keyword in constructors?

Answer:

The super keyword is used to call the constructor of the superclass from a subclass. This is useful when you need to initialize the superclass part of an object from within a subclass constructor.

### ▲ Can a constructor call another constructor in the same class? How?

Answer:

Yes, a constructor can call another constructor in the same class using the this keyword. This is known as constructor chaining.

```
public class BrowserConfig {
    private String browser;
    private String driverPath;

    public BrowserConfig() {
        this("Chrome", "/path/to/chromedriver");
    }

    public BrowserConfig(String browser, String driverPath) {
        this.browser = browser;
        this.driverPath = driverPath;
    }
}
```

### ▲ How do you use a constructor to initialize a WebDriver in Selenium?

Answer:

A constructor can be used to initialize a WebDriver by setting up the necessary configurations and creating an instance of the WebDriver within the constructor.

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class WebDriverInitializer {
    private WebDriver driver;
```

```
public WebDriverInitializer() {
    System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
    this.driver = new ChromeDriver();
}

public WebDriver getDriver() {
    return this.driver;
}
}
```

🚀 Let's connect on 1:1 call: <https://lnkd.in/ddayTwng>

🚀 To learn automation and java with practical examples refer the Masterclass in Automation/SDET: <https://lnkd.in/giCxnJJZ>

## Test Automation-Java Scenario Based Interview Q&A

Overloading:

Question: How would you create overloaded methods for finding elements using Selenium WebDriver?

Answer:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class ElementFinder {
```

```

public WebElement findElement(WebDriver driver, String locator) {
    return driver.findElement(By.xpath(locator));
}

public WebElement findElement(WebDriver driver, By locator) {
    return driver.findElement(locator);
}
}

```

## Overriding:

Question: Describe a scenario where you would override the `toString()` method in a custom `WebElement` class for logging purposes in Selenium.

Answer:

```

import org.openqa.selenium.WebElement;

public class CustomWebElement extends WebElement {
    @Override
    public String toString() {
        return "Custom element with tag name: " + this.getTagName();
    }
}

```

## Encapsulation:

Question: How can encapsulation be applied to manage `WebDriver` instances in Selenium tests?

Answer:

```

import org.openqa.selenium.WebDriver;

public class WebDriverManager {
    private WebDriver driver;

    public WebDriver getDriver() {
        if (driver == null) {
            // Initialize WebDriver here
        }
        return driver;
    }
}

```

```
}
```

## Inheritance:

Question: How does the Page Object Model (POM) utilize inheritance in Selenium tests?

Answer:

```
public class BasePage {  
    // Common page elements and methods  
}  
  
public class HomePage extends BasePage {  
    // Page-specific elements and methods  
}
```

## Enums:

Question: Explain how you could use enums to define browser types for cross-browser testing in Selenium.

Answer:

```
public enum BrowserType {  
    CHROME, FIREFOX, SAFARI, EDGE  
}
```

## Generics:

Question: How would you create a generic method to handle dynamic waits in Selenium?

Answer:

```
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.support.ui.WebDriverWait;  
  
public class WaitUtils {  
    public static <T> T waitFor(WebDriver driver, long timeoutInSeconds, T  
condition) {  
        WebDriverWait wait = new WebDriverWait(driver, timeoutInSeconds);
```

```
    return wait.until(condition);
}
}
```

## Strings:

Question: Provide an example of using strings in Selenium to verify page titles.

Answer:

```
import org.openqa.selenium.WebDriver;

public class PageTitleChecker {
    public boolean isPageTitleCorrect(WebDriver driver, String expectedTitle) {
        return driver.getTitle().equals(expectedTitle);
    }
}
```

## Array:

Question: How could you use arrays in Selenium to store multiple element locators?

Answer:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class ElementLocator {
    private By[] locators;

    public ElementLocator(By... locators) {
        this.locators = locators;
    }

    public WebElement findElement(WebDriver driver) {
        for (By locator : locators) {
            WebElement element = driver.findElement(locator);
            if (element != null) {
                return element;
            }
        }
    }
}
```

```
    return null;
  }
}
```

## List:

Question: How would you use lists in Selenium to store a collection of WebElements?

Answer:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

import java.util.List;

public class ElementList {
    public List<WebElement> findElements(WebDriver driver, By locator) {
        return driver.findElements(locator);
    }
}
```

## Map:

Question: Describe a scenario where you would use a map in Selenium to store test data for data-driven testing.

```
import org.openqa.selenium.WebDriver;
import java.util.Map;

public class TestData {
    public void performTest(WebDriver driver, Map<String, String> testData) {
        // Use test data for test execution
    }
}
```

These scenarios demonstrate how core Java concepts can be applied in the context of Selenium WebDriver for effective test automation

# INTERFACE

## How to use Java Interface in Test Automation ?

In Selenium WebDriver, Java interfaces can be used to define page objects and implement the Page Object Model (POM) design pattern.

The Page Object Model is a design pattern that encourages the separation of test logic from the UI layer by creating a separate class for each web page and defining methods to interact with elements on that page.

Here's a simple example demonstrating how to use a Java interface in Selenium WebDriver:

Let's say we have a web application with a login page and a home page. We can create interfaces for each page to define the elements and actions specific to that page.

● Interface for the Login page: [LoginPage.java](#)

```
public interface LoginPage {  
    WebElement getUsernameInput();  
    void login(String username, String password);  
}
```

● Implementation of LoginPage interface: [LoginPageImpl.java](#)

```
public class LoginPageImpl implements LoginPage {  
    private WebDriver driver;  
  
    @FindBy(id = "username")  
    private WebElement usernameInput;  
  
    public LoginPageImpl(WebDriver driver) {  
        this.driver = driver;  
        PageFactory.initElements(driver, this);  
    }  
    public WebElement getUsernameInput() {  
        return usernameInput;  
    }  
    public void login(String username, String password) {  
        usernameInput.sendKeys(username);  
    }  
}
```



```
}
```

● With this setup, you can create instances of LoginPageImpl classes in your test classes and interact with the elements on the respective pages using the methods defined in the interfaces.

```
WebDriver driver = new ChromeDriver();
LoginPage loginPage = new LoginPageImpl(driver);
loginPage.login("username");
System.out.println("Logged in user: " + loggedInUserName);
driver.quit();
```

=====

More Q&A [Optional]

[Top 50 JAVA Interview Q&A](#)

[OOPS ADVANCED](#)

## Commonly Asked Java Interview Q&A 2024 [IMP]

👉 Java program to remove duplicates characters from given String.

👉 Program Remove the second highest element from the HashMap.

👉 Java program to Generate prime numbers between 1 & given 4 number

👉 How to find the missing values from a sorted array.

👉 Java program to input name, middle name and surname of a person and print only the initials.

👉 Program to Print all Treemap elements?

👉 What is a singleton Design Pattern? How do you implement that in your framework?

👉 Write the Top 5 test cases for Booking Coupons.

👉 What is serialization and deserialization?

👉 What is the Difference between status codes 401 and 402?

👉 Difference between selenium 3 and selenium 4?

👉 What is delegate in Java and where do you use Delegate in your Framework?

👉 How many maximum thread-pool can you open in the TestNG?

👉 What are the Major challenges that come into the picture when you do parallel testing using TestNG and Grid?

👉 How do you integrate your automation framework with the Jenkins pipeline?

👉 What will happen if we remove the main method from the java program?

👉 What is the component of your current Project?

👉 How do you pass parameters in TestNG?

👉 Write the logic of retrying the failed test case with a minimum 3 numbers of time in Automation Testing. Which Interface do you use for it?

👉 What is the OOPs concept in java?

👉 Difference Between Classes and Objects?

👉 What is collection in Java?

👉 In How many ways can we create an object?

👉 Why is Java not 100% Object-oriented?

👉 Can we make a constructor as Static?

👉 How to convert a JSON to java object using Jackson? POJO

👉 What is the difference between Abstraction Class and Interfaces?

👉 Difference between String, StringBuilder, and StringBuffer?

👉 What are other immutable classes in Java apart from String?

👉 Difference between TreeMap and HashMap?

👉 How do you set priorities for test automation, which test needs to be automated first?

👉 How do you set test case priorities for your team?

👉 What are the functional things you need to test on e-commerce sites?

-----

👉 Interview Q&A Package to crack interviews for Automation Testing and SDET with similar examples: <https://lnkd.in/gJrBGExe>

💚 Arrange 1:1 call on career guidance for SDET or Automation: <https://lnkd.in/dF5ADMiU>

📢 E2E Automation-SDET Sessions with long-term mentorship: <https://lnkd.in/giCxnJJ7>

-----