1- Take Screenshot :

```
public static String Screenshot(WebDriver driver) {

    TakesScreenshot ts=(TakesScreenshot) driver;

    File src=ts.getScreenshotAs(OutputType.FILE);

    String
path=System.getProperty("user.dir")+"\\src\\test\\java\\ScreenShot\\"+GetCurr
entTime()+".jpg";

    File destination=new File(path);

    try
    {
        FileUtils.copyFile(src, destination);
    } catch (IOException e)
    {
        loggerr.info("Capture Failed "+e.getMessage());
    }

    return path;
}
```

2- Reports:

```
3- @BeforeSuite
       public void setupReport() {

           extent = new ExtentHtmlReporter(new
   File(System.getProperty("user.dir") +
   "\\src\\test\\java\\TestReports\\Web-Automation On Chrome "+
   GetCurrentTime() + " .html"));
           report = new ExtentReports();
           report.attachReporter(extent);
       }

       @BeforeMethod
       public void setUp(Method method){
           BasicConfigurator.configure();
          // ExtentHtmlReporter extent = new ExtentHtmlReporter(new
   File(System.getProperty("user.dir") +
   "\\src\\test\\java\\TestReports\\Web-Automation On Chrome "+
   GetCurrentTime() + " .html"));
   //        report = new ExtentReports();
   //        report.attachReporter(extent);
           logger = report.createTest(method.getName());
           WebDriverManager.chromedriver().setup();
           driver = new ChromeDriver();
           driver.manage().window().maximize();
           driver.manage().timeouts().implicitlyWait(20 ,
   TimeUnit.SECONDS);
           driver.get(Configurations.url);
       }

       @AfterMethod
       public void getResult(ITestResult result) throws IOException{
           if(result.getStatus() == ITestResult.FAILURE){
               logger.log(Status.FAIL,
   MarkupHelper.createLabel(result.getName()+" FAILED ",
   ExtentColor.RED));
               logger.fail(result.getThrowable());
               String temp=  Screenshot(driver);
               logger.fail(result.getThrowable().getMessage(),
   MediaEntityBuilder.createScreenCaptureFromPath(temp).build());
           }else if(result.getStatus() == ITestResult.SUCCESS) {
               logger.log(Status.PASS,
   MarkupHelper.createLabel(result.getName()+" PASSED " ,
   ExtentColor.GREEN));
               String temp=  Screenshot(driver);
               logger.pass(result.getMethod().getMethodName(),
   MediaEntityBuilder.createScreenCaptureFromPath(temp).build());
           }
           else {
               logger.log(Status.SKIP,
   MarkupHelper.createLabel(result.getName()+" SKIPPED ",
   ExtentColor.ORANGE));
               logger.skip(result.getThrowable());
```

```
            }
           // report.flush();
            driver.quit();


        }



        @AfterSuite
        public void tearDownReport() {
            report.flush(); // Generate the final report once after all
    tests
        }

public static String GetCurrentTime () {
    DateFormat format = new SimpleDateFormat("dd MMMM YYYY hh.mm.ss");
    Date date=new Date();
    return  format.format(date);
}
```

3- Waits

In Selenium WebDriver with Java, waits are essential to handle the dynamic loading of elements on a web page. They allow the test to pause and wait for certain conditions to be met before proceeding, which improves reliability by preventing test failures caused by elements not being immediately available. Selenium provides three main types of waits:

1. Implicit Wait

- **Purpose**: Implicit wait tells WebDriver to poll the DOM for a certain amount of time to locate an element. It applies to all elements in the test script, and WebDriver waits for the specified time before throwing a `NoSuchElementException` if the element is not found.
- **Syntax**:

  ```
  java
  Copy code
  driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
  ```

- **When to Use**: Use implicit waits when dealing with elements that take a predictable time to load, as it applies globally.

2. Explicit Wait

- **Purpose**: Explicit wait allows you to define a specific condition for a specific element before the WebDriver proceeds. It is more flexible than an implicit wait as it waits for a condition (such as visibility, clickability, etc.) on an individual element.
- **Syntax**:

  ```
  java
  Copy code
  WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
  WebElement element =
  wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("example
  Id")));
  ```

- **Common Conditions**:
  - `visibilityOfElementLocated()`: Waits for the element to be visible.
  - `elementToBeClickable()`: Waits for the element to be clickable.
  - `presenceOfElementLocated()`: Waits for the element to be present in the DOM.
- **When to Use**: Use explicit waits when you need a specific condition for an element, like waiting for a button to become clickable or a message to appear.

3. Fluent Wait

- **Purpose**: Fluent wait is a more advanced form of an explicit wait that allows more control over polling frequency and conditions. It waits for a condition to be met within a timeout period and lets you specify how frequently WebDriver should check the condition. It also handles exceptions like `NoSuchElementException` by ignoring them.
- **Syntax**:

  ```
  java
  Copy code
  Wait<WebDriver> wait = new FluentWait<>(driver)
      .withTimeout(Duration.ofSeconds(30))
      .pollingEvery(Duration.ofSeconds(5))
      .ignoring(NoSuchElementException.class);
  WebElement element =
  wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("example
  Id")));
  ```

- **When to Use**: Use Fluent Wait when working with elements that load unpredictably or when you need to ignore specific exceptions.

Examples of Using Waits Effectively

1. **Implicit Wait Example**:

   ```
   java
   Copy code
   driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
   driver.findElement(By.id("exampleId")).click();
   ```

2. **Explicit Wait with Custom Condition**:

   ```
   java
   Copy code
   WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(20));
   WebElement element =
   wait.until(ExpectedConditions.textToBePresentInElementLocated(By.id("ex
   ampleText"), "Expected Text"));
   ```

3. **Fluent Wait Example for Unpredictable Elements**:

   ```
   java
   Copy code
   Wait<WebDriver> wait = new FluentWait<>(driver)
       .withTimeout(Duration.ofSeconds(20))
       .pollingEvery(Duration.ofSeconds(2))
       .ignoring(NoSuchElementException.class);
   WebElement element =
   wait.until(ExpectedConditions.presenceOfElementLocated(By.id("dynamicEl
   ement")));
   ```

Summary

- **Implicit Wait**: Sets a global wait time for all elements in the DOM.
- **Explicit Wait**: Waits for specific conditions of a specific element, giving more control.
- **Fluent Wait**: Advanced wait with polling frequency control, useful for highly dynamic elements.

Using these waits properly in your Selenium tests will help avoid test failures due to timing issues and ensure that your tests run more reliably.

Here are examples for each type of wait in Java Selenium.

1. Implicit Wait

This wait is set once and applies to all elements in the WebDriver session. It waits until the element is found or the specified time elapses.

```
java
Copy code
// Set up WebDriver
WebDriver driver = new ChromeDriver();

// Set implicit wait for 10 seconds
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

driver.get("http://example.com");

// Attempt to find an element; it will wait up to 10 seconds
WebElement element = driver.findElement(By.id("exampleElement"));
```

```
element.click();
driver.quit();
```

2. Explicit Wait

This is used for specific conditions and only applies to the selected element.

```
java
Copy code
// Set up WebDriver
WebDriver driver = new ChromeDriver();

driver.get("http://example.com");

// Explicit wait for up to 10 seconds for the element to be visible
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
WebElement element =
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("exampleElemen
t")));

element.click();
driver.quit();
```

3. Fluent Wait

Fluent wait is a more flexible wait that allows you to control polling intervals and exceptions to ignore.

```
java
Copy code
// Set up WebDriver
WebDriver driver = new ChromeDriver();

driver.get("http://example.com");

// Fluent wait with a timeout of 15 seconds, polling every 2 seconds,
ignoring NoSuchElementException
Wait<WebDriver> wait = new FluentWait<>(driver)
    .withTimeout(Duration.ofSeconds(15))
    .pollingEvery(Duration.ofSeconds(2))
    .ignoring(NoSuchElementException.class);

WebElement element =
wait.until(ExpectedConditions.elementToBeClickable(By.id("exampleElement")));

element.click();
driver.quit();
```

4- Annotation :

TestNG annotations provide structure for organizing and managing test flows, making testing more efficient and reliable. Here's a breakdown of common annotations, including practical uses for each:

1. **@BeforeSuite**
   - **Purpose**: Runs once before any tests in the suite begin, making it useful for initializing resources that are needed for the entire suite.
   - **Usage**: Start a server, configure global test settings, or connect to a database.
   - **Example**:

     ```
     java
     Copy code
     @BeforeSuite
     public void setupSuite() {
         System.out.println("Starting the server for test suite.");
     }
     ```

2. **@AfterSuite**
   - **Purpose**: Executes after all tests in the suite are completed, used to clean up or release resources.
   - **Usage**: Close database connections or shut down the server.
   - **Example**:

```
java
Copy code
@AfterSuite
public void teardownSuite() {
    System.out.println("Shutting down server.");
}
```

3. **@BeforeTest**
   - **Purpose**: Runs before any test methods within the `<test>` XML tag, setting up prerequisites at a higher level than individual tests.
   - **Usage**: Initialize web drivers or set up test-specific configurations.
   - **Example**:

```
java
Copy code
@BeforeTest
public void setupTest() {
    System.out.println("Setting up configurations for all
tests.");
}
```

4. **@AfterTest**
   - **Purpose**: Runs after all test methods in a `<test>` XML tag are complete, useful for releasing resources specific to that tag.
   - **Usage**: Tear down web driver instances or other temporary configurations.
   - **Example**:

```
java
Copy code
@AfterTest
public void teardownTest() {
    System.out.println("Cleaning up configurations after test.");
}
```

5. **@BeforeClass**
   - **Purpose**: Runs once before any methods in a class, typically to initialize resources shared by all methods in that class.
   - **Usage**: Open a browser or initialize API connections for a particular test class.
   - **Example**:

```
java
Copy code
@BeforeClass
public void setupClass() {
    System.out.println("Opening browser for test class.");
}
```

6. **@AfterClass**
   - **Purpose**: Executes after all methods in a class have run, useful for class-specific cleanup.
   - **Usage**: Close the browser or disconnect from a service.
   - **Example**:

```
java
Copy code
@AfterClass
public void teardownClass() {
    System.out.println("Closing browser after test class.");
}
```

7. **@BeforeMethod**
   - **Purpose**: Runs before each test method, making it ideal for setting up test-specific prerequisites.
   - **Usage**: Clear cookies, reset configurations, or log in as a particular user.
   - **Example**:

```
java
Copy code
@BeforeMethod
public void setupMethod() {
    System.out.println("Clearing browser cookies before each
test.");
}
```

8. **@AfterMethod**
   - **Purpose**: Runs after each test method, used to clean up anything created or changed during the test.
   - **Usage**: Log out or reset the application state.
   - **Example**:

```
java
Copy code
@AfterMethod
public void teardownMethod() {
    System.out.println("Logging out after each test.");
}
```

9. **@Test**
   - **Purpose**: The main annotation to define an actual test case.
   - **Usage**: Used to specify test cases, often with additional parameters like `priority`, `groups`, and `dataProvider`.
   - **Example**:

```
java
Copy code
@Test(priority = 1, groups = {"smoke"})
public void sampleTest() {
    System.out.println("Executing sample test.");
}
```

10. **@DataProvider**

Order of Execution

The annotations are executed in a specific order:

1. `@BeforeSuite`
2. `@BeforeTest`
3. `@BeforeClass`
4. `@BeforeMethod`
5. `@Test`
6. `@AfterMethod`
7. `@AfterClass`
8. `@AfterTest`
9. `@AfterSuite`