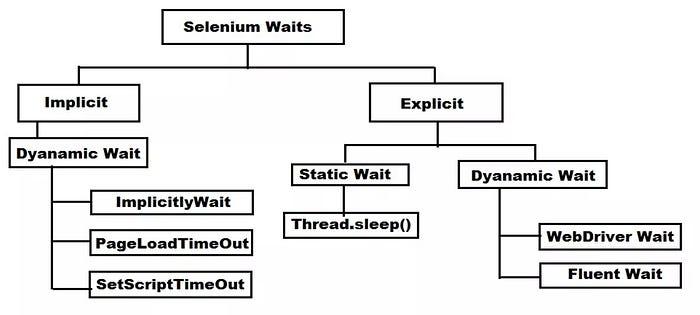
**1- Types of Waits in Selenium**

Selenium offers three types of waits to manage if the element is appeared, disappeared or clickable. These are the types of waits:  
implicit wait, explicit wait, and fluent wait.



# Selenium Timeouts

In Selenium, a TimeoutException is an exception raised when an operation runs out after a specified period. For example, when using the driver.get(); method to load a page, the default timeout is set to 300 seconds. If the page takes longer than 300 seconds to load, a TimeoutException will be raised.

Default Selenium Timeouts:

**Implicit Wait - after the page load**

* ‘Zero’(0) seconds for implicit waits. Here the Selenium Command reports immediately if it cannot find an element.

**Page Load**

* 300 secondsfor page loads

**Script Load**

* 30 secondsfor script timeouts

**1.1- Implicit Wait**Implicit Wait is a global wait applied to all elements in the script. It tells the WebDriver to wait for a certain amount of time before throwing an exception if the element is not immediately available. This wait is applied once, and the WebDriver will continue to wait for the specified time before interacting with each element. The default value is 0.  
here is an example that how we can apply implicit wait:

public static void main(String[] args) {  
 // Set the path of the ChromeDriver executable  
 System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");  
  
 // Create a WebDriver instance (replace with your desired browser)  
 WebDriver driver = new ChromeDriver();  
  
 // Implicitly wait for 10 seconds  
 driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);  
  
 // Navigate to a webpage  
 driver.get("https://example.com");  
  
 // Find an element using Implicit Wait  
 WebElement element = driver.findElement(By.id("myButton"));  
  
 // Perform your desired action on the element (e.g., click it)  
 element.click();  
  
 // Close the browser  
 driver.quit();  
}

**1.2- Explicit Wait**

* Explicit Wait is used to wait for a specific condition to be met before interacting with an element.
* Unlike Implicit Wait, Explicit Wait is applied to specific elements only.
* An explicit wait can only be implemented in cases where synchronization is needed for an element and the rest of the script is working fine.
* Explicit wait can be used to verify the page title, page url or to find the element.
* The Selenium WebDriver provides **WebDriverWait** and **ExpectedCondition** classes for implementing an explicit wait.
* The **ExpectedConditions** class allows us to specify conditions, like the existence or non-existence of an element, in Selenium WebDriver. If the condition is not met within the specified time, an exception is raised.
* The WebDriverWait object will call the ExpectedConditions class method for every 500 milliseconds until it returns successfully or specified time is over.
* Since explicit wait is not set globally, we have the flexibility to utilize different conditions and timeouts for various elements or actions.
* This allows us to fine-tune the waiting behavior for different parts of our test scenarios as needed.

Here is an example of how we can use explicit wait.

public static void main(String[] args) {  
 // Set the path of the ChromeDriver executable  
 System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");  
  
 // Create a WebDriver instance (replace with your desired browser)  
 WebDriver driver = new ChromeDriver();  
  
 // Navigate to a webpage  
 driver.get("https://example.com");  
  
 // Create an explicit wait with a timeout of 10 seconds  
 WebDriverWait wait = new WebDriverWait(driver, 10);  
  
 // Define a condition for the element you want to interact with (e.g., a button with ID "myButton")  
 By elementLocator = By.id("myButton");  
  
 // Wait until the element is present and clickable  
 WebElement element = wait.until(ExpectedConditions.elementToBeClickable(elementLocator));  
  
 // Perform your desired action on the element (e.g., click it)  
 element.click();  
  
 // Close the browser  
 driver.quit();  
 }

**1.3- Fluent Wait**Fluent Wait is more flexible than both Implicit and Explicit Waits. It allows you to define a custom polling frequency and exception handling while waiting for an element.

here is an example of how we can use fluent wait:

public static void main(String[] args) {  
 // Set the path of the ChromeDriver executable  
 System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");  
  
 // Create a WebDriver instance (replace with your desired browser)  
 WebDriver driver = new ChromeDriver();  
  
 // Navigate to a webpage  
 driver.get("https://example.com");  
  
 // Define a Fluent Wait with a timeout of 10 seconds and polling every 500 milliseconds  
 FluentWait<WebDriver> wait = new FluentWait<>(driver)  
 .withTimeout(Duration.ofSeconds(10))  
 .pollingEvery(Duration.ofMillis(500))  
 .ignoring(Exception.class);  
  
 // Find the element you want to interact with (e.g., a button with ID "myButton")  
 WebElement element = driver.findElement(By.id("myButton"));  
  
 // Wait until the element is present wait.until(ExpectedConditions.*presenceOfElementLocated(By locator)*);  
  
 // Perform your desired action on the element (e.g., click it)  
 element.click();  
  
 // Close the browser  
 driver.quit();  
 }

**Page Load & Script Load:**

In Selenium, page load time is the time it takes for a page and its content to download, including scripts. Script load time is the time it takes for scripts to run, get more data, and populate the page.

* Generally, page load waits are triggered until the DOM loads before letting the WebDriver proceed. For example, if an automated test clicks on a website’s Add to Cart button, WebDriver will execute the next line in the script only when the page loads completely. In such cases, there’s no need to instruct WebDriver to wait for page load separately.
* It is usually necessary to introduce a wait time in the Selenium script when navigating Ajax-based or dynamic elements that may continue to load after the page loads.
* Certain elements may only become visible after the page loads or after a user action but be available for interaction after a few seconds have passed. Think of a dropdown menu with dynamic values. It’s always available on the DOM, but its values are populated based on the user’s action.

**2- which one is better?**

The choice between Implicit Wait, Explicit Wait, or Fluent Wait depends on your specific test scenarios and requirements. Each type of wait has its advantages and use cases:

**2.1- Implicit Wait:**1)Suitable for scenarios where most of the elements on the page have similar loading times.  
2) Easy to set up since it’s applied globally once for the entire WebDriver instance.  
3) Useful for basic synchronization needs across the entire test script.  
4) Can lead to longer wait times for elements that load quickly, as it will wait for the maximum specified time for every element.  
5) Less flexible compared to Explicit and Fluent Waits.

**2.2- Explicit Wait:**1)Preferred for scenarios where you need to wait for specific conditions before interacting with elements.  
2) Allows fine-grained control over waiting for individual elements, improving test reliability and speed.  
3) Provides various ExpectedConditions (e.g., elementToBeClickable, presenceOfElementLocated, etc.) to wait for specific element states.  
4) More reliable and precise than Implicit Wait since it waits only for the specified conditions and the defined timeout.

**2.3- Fluent Wait:**1) Offers more advanced features, such as custom polling frequency and exception handling during waiting.  
2) Useful for handling situations where the element loading time may vary or is unstable.  
3) Allows you to define a custom condition using the Function interface.  
4) Provides more flexibility and control compared to Implicit and Explicit Waits.  
5) Requires more code to set up and may be more complex for simple scenarios where Explicit Wait is sufficient.

In general, Explicit Wait is the most commonly used and recommended wait strategy. It offers better control and precision over waiting conditions, making test scripts more reliable and less prone to synchronization issues. However, in certain complex scenarios with unpredictable element loading times, Fluent Wait might be the preferred choice.

**3- Best Practices**

These are some best practices that needs to be considered when using Selenium waits:  
**3.1-** **Avoid using explicit wait for every element:** While explicit waits provide more control, using them excessively can lead to bloated and less maintainable code. Consider using implicit waits for elements that are expected to be available on most pages and use explicit waits selectively for elements with specific waiting conditions.  
**3.2- Use explicit wait instead of implicit wait:** Implicit waits can cause our tests to take longer than necessary to fail if the element cannot be located. Explicit and fluent waits are better because they allow us to wait for a specific condition to occur before proceeding further.

* Implicit waits set a global timeout for all elements, causing unnecessary delays in case of element absence, leading to longer test execution times.
* Explicit and fluent waits, such as WebDriverWait, provide more control and precision by waiting only for specific conditions, improving test reliability and efficiency.

**3.3- Use fluent waits when necessary:** We should use fluent waits when we need to verify a specific condition repeatedly until it becomes true, as they offer greater control over the waiting mechanism.

* Fluent waits are beneficial when dealing with dynamically changing elements or when a condition needs to be checked multiple times.
* They allow us to customize the polling interval, exception handling, and wait duration, making them suitable for complex scenarios with varying element loading times.

**3.4- Use reasonable wait times:** Wait times that are too short can cause false negatives, while wait times that are too long can unnecessarily increase the total execution time of the test.

* Select wait times wisely based on the application’s responsiveness and expected element loading times.
* Short wait times may lead to false test failures, while excessively long wait times can slow down test suites and waste valuable execution time.

**4- Problems that might arise due to inefficient use of waits**When using Selenium for web automation, not using proper wait techniques can lead to various issues, as it may cause the script to generate wrong results.Here are some of the issues that may arise if you do not use wait properly in Selenium:

**4.1- Element not found:** If the script tries to interact with an element before it is present in the DOM, it will raise a “NoSuchElementException.”

**4.2- StaleElementReferenceException:** If an element is located, but the DOM is refreshed or the element is removed from the DOM before interacting with it, this exception is thrown.

**4.3- Element not interactable:** If an element is present but not yet fully loaded or enabled, trying to interact with it (e.g., clicking, typing) will result in an error.

**4.4- Synchronization issues:** Without proper wait statements, Selenium may proceed to the next step of the script before a particular action is completed, leading to synchronization issues.

**4.5- Inconsistent test results:** Without appropriate waits, test results may be inconsistent across different runs, making it challenging to identify and reproduce issues consistently.

**4.6- Slower test execution:** Using excessive waits (e.g., thread.sleep) can make your test scripts slower, as they introduce unnecessary delays, impacting the overall efficiency of the test suite.

**4.7- Flaky tests:** Tests that are sensitive to timing and lack proper waits are more likely to fail intermittently, making them unreliable and difficult to maintain.

**4.8- Inefficient resource utilization:** Without proper waits, the browser may continuously retry to find an element, consuming unnecessary resources and potentially slowing down the test execution.