Synchronization

- Program always executes in fast manner.
- It doesn't care about what is happening in application.
- If application takes more time to respond then the program fail.
- We need to make sure that the program consider application speed
- For that we have to provide waiting time to program based on application speed
- We can give wait time in two ways
    - **Implicit wait**: configuring the wait in default way for all page navigations and element identifications.
    - If element takes more time to load then we can use this as default timeout
    - This will not work if we have to wait for some conditions then this will not work.
    - **Explicit wait**: we can provide waiting based application status
    - we can wait for element/alert/frame...etc
    - we can wait for any attribute value
    - we can wait for element status (visibility/clickability)
- There is an advanced waiting technique called **fluent wait**. We can customize the way how we wait for elements/element status/...etc.
- If any wait statement is failed we get timeout exception
- PageLoad Timeout and implicitlywait comes under implicit wait
- //setup page load timeout
    - driver.manage().timeouts().pageLoadTimeout(15, TimeUnit.SECONDS);
- //element wait time
    - driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
- Thread.sleep is given by java for waiting specific number of seconds
    - Thread.sleep(milliseconds);
- For making code to explicitly wait for a condition we have to use webdriverwait class. This class is having a method "until" which will wait for a condition.
- We can provide conditions by using ExpectedConditions class.
- There are predifiened set of conditions to wait for elements, window, frames, alerts and attributes...etc.

Creating Code for explicit wait

```
WebDriverWait w = new WebDriverWait(driver, 15);

//below statemenet will wait for element to be present in source code

w.until(ExpectedConditions.presenceOfElementLocated(By.linkText("Log In / Register")));


//below statement will wait for presence of alert

w.until(ExpectedConditions.alertIsPresent());
```

```
//below statement will wait for attribute value for an element
w.until(ExpectedConditions.attributeContains(element, attribute, value)

//below statement will wait for clickability of an element
w.until(ExpectedConditions.elementToBeClickable(locator));

//below statement will wait for visibility of an element
w.until(ExpectedConditions.visibilityOf(element));
```

Fluent Wait

- If we want to develop custom conditions we can use FluentWait. We can customize the way how ExpectedConditions class works.
- We can wait and perform some operation using fluent wait concept
- Or We can return element if found after waiting
- If explicit wait is failed we get timedout exception
- We have to import Function class from com.google.common.base.Function; when writing fluent wait
- We have to follow a syntax while writing a custom condition
- We have to use Function class. Here we specify input and output datatypes.
- If we specify new Function<WebDriver, WebElement>() then it takes driver as input and return webelement.
- If we specify new Function<WebDriver, boolean>() then it takes driver as input and return boolean value.

Below method will wait for Element and return the element after it exists

```
public static WebElement waitAndReturnElement(WebDriver driver, By locator, int tOut) {
        // Waiting 30 seconds for an element to be present on the page, checking
        // for its presence once every 5 seconds.
        Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
                        .withTimeout(tOut, TimeUnit.SECONDS)
                        .pollingEvery(1, TimeUnit.SECONDS)
                        .ignoring(NoSuchElementException.class);
        WebElement elm = wait.until(new Function<WebDriver, WebElement>() {
```

```
                public WebElement apply(WebDriver driver) {
                        return driver.findElement(locator);
                }
        });
        return elm;
}
```

Below method will wait for Element and return the true or false after finding it

```
public static boolean waitAndReturnElementExistance(WebDriver driver, By locator, int tOut) {
        // Waiting 30 seconds for an element to be present on the page, checking
        // for its presence once every 5 seconds.
        Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
                        .withTimeout(tOut, TimeUnit.SECONDS)
                        .pollingEvery(1, TimeUnit.SECONDS)
                        .ignoring(NoSuchElementException.class);
        boolean elmFound = wait.until(new Function<WebDriver, boolean>() {
                public boolean apply(WebDriver driver) {
                        driver.findElement(locator);
                        return true;
                }
        });
        return elm;
}
```

OOPS

- OOPS is useful for extending the existing system
- There are 3 major concepts in OOPS
- **Encapsulation** : Keeping every related thing under single entity
    - We use it to hide/restrict the data access
    - Ex: there is a class which has emp age variable. The developer wants to restrict the age between 18 and 65.
    - If it is a public variable everybody will access and any value can be assigned
    - If it is a private variable nobody can access
    - If it is a public and final variable everybody can access in read only way
    - In this situation we create variable as private and write methods to get or to set value. These methods are called setters and getters.
- **Inheritance**: This is used to inherit the features from one class to another class.
    - We can use extends keyword to inherit the features
    - All class members without private keyword gets inherited
    - If we use inheritance we can able to implement method overriding concept
    - Writing the same method with same signature is called overriding
    - If we create instance for parent class and take it as parent class then the methods from parent class will be executed. Every class member will be taken from parent class
        - ParentClass variable = new ParentClass();
    - If we create instance for child class and take it as child class then the methods from child class will be executed. If not available in child class then the methods from parent will be executed.
        - ChildClass variable = new ChildClass();
    - If we create instance for child class and take it as parent class then the methods from child class will be executed if ovverridden. If not then the methods from parent will be executed.
    - ParentClass variable = new ChildClass();
- **Polymorphism**: Used to create different forms for the method
    - we can get Method Over Riding and Over loading features in polymorphism
    - Creating different methods with same name but different parameters is called method overloading. It can be overloaded in same class or in child class.
    - Creating different methods with same name and parameters is called method overriding. It can be overrided in child class.
- **Abstract Classes:** Abstract classes are useful to common methods.
    - If a method is shared across multiple classes then instead writing the same method in all classes we can write in one abstract class and extend it.
    - In java8 the difference between interface and abstract classes came down because interfaces are accepting method with body.
    - Before in interface we provide set of rules for classes.
    - If multiple classes are having same method implementation then we create those methods in abstract class and extends that abstract with the class
    - We cannot create instance for abstract class

- You need to extend the abstract class with another class to use abstract class members
- In abstract class we can have abstract and non abstract methods
- We have to use abstract keyword to create abstract method
- A method without body is called abstract method
- Abstract methods must be implemented by extended classes (like interface)
- A class can implement multiple interfaces but not multiple classes
- multiple inheritance is not possible (class a extends b, c) but multi level inheritance is possible