We use Java as a programming language and TestNG as the testing framework.

We use Maven for managing all the dependencies needed for our project and we are using GitHub for version control to check-in our code files.

Our framework is built with Data Driven approach with a combination of Page Object Model. We use Test data from external files because we should never hard-code the test data and we use Apache POI to read Test Data from Excel files.

We use Extent Reports for detailed reporting because it's a very important component of any framework. We are using Log4j2 for logging all the details of the automation workflow for debugging purpose. We take screenshots for failed test cases and we attach screenshots to automation reports also.

Our framework is very modular and we have created page classes for all the common components, not just for particular pages. If there is something common which occurs in different pages, then we create Page Object for that also so that different Test Classes can make use of it. We are avoiding all the redundancy, we follow DRY Principle which means Don't Repeat Yourself so that we have an efficient code out there.

We are using TestNG XML files for handling different Test Suites like Sanity Test Suite and Regression Test Suite and we are using Jenkins to run those different kind of Test Suites.

Whenever there is a deployment, our Sanity Test Suite is triggered and we have multiple Regression Test Suites which run on different schedule. Some are triggered nightly, some are triggered every two days and some are triggered weekly and their schedule also depends on how much time a particular Test Suite takes to complete. As an example, our Sanity Test Suite completes under one hour, which verifies basic functionality. Our nightly suite runs for 8 hours and weekend suite runs for about 24 hours so that we are covering all levels of testing.

How to explain your Automation Framework using Oop's concepts

Inheritance

Polymorphism

Encapsulation

Abstraction

Obsqura ZONE

INTERFACES

We all know the basic statement used in our Automation Framework,

```
WebDriver driver = new ChromeDriver();
```

WebDriver itself is an Interface. So based on the given statements we are initializing ChromeDriver browser using Selenium WebDriver interface. We simply create the objects of the driver classes and work with them. It means we are creating an Object with reference variable as driver of the interface WebDriver. Other examples of interfaces in our framework are WebElement, WebDriverWaits, JavaScripExecutor etc.

```
public class BaseTest
{
public WebDriver driver;

@BeforeMethod
public void init(){
driver = new ChromeDriver();
}
@AfterMethod
public void closeBrowser (){
driver.close();
}
}
```

POLYMORPHISM

There are two types of Polymorphism in Java, they are

method overloading and method overriding.

Polymorphism allows us to perform a task in multiple ways.

We will see with an example on both overloading and overriding.

Compile time polymorphism or Method overloading

A class having multiple methods with same name but different parameters is called Method Overloading.

At compile time, JVM knows which method to invoke by verifying the signature of methods. In any automation frameworks we use implicit waits. This is a classic example of method overloading.

In Implicit wait we use to pass different parameters such as SECONDS, MINUTES, and HOURS etc depending upon the requirements of our Automation framework.

```
public class BaseTest {

public WebDriver driver;

@BeforeMethod
public void init(){
driver = new ChromeDriver();
driver.manage.timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}

@AfterMethod
public void closeBrowser(){
driver.close();
}
}
```

Runtime polymorphism or Method overriding.

Declaring a method in child class which is already present in the parent class is called Method Overriding.

In simple words, overriding means to override the functionality of an existing method. In the below code snippet we assume that Browser value will take from an Excel or XML file as we are considering a Data Driven Framework. We then create the object of required browser and store it in the reference variable of parent interface called 'WebDriver'.

Depending upon which parameter values we pass from the excel sheet, the init method will instantiate and return corresponding WebDriver.

```
public class BaseTest {
   public WebDriver driver;
      @BeforeMethod
       public driver init(String browser){
       if(browser.equals("Chrome")) {
       driver=new ChromeDriver();
        }
       else if(browser.equals("IE")) {
       driver=new InternetExplorerDriver();
       }
  driver.manage.timeouts().implicitlyWait(30, TimeUnit.SECONDS);
    return driver;
     }
   @AfterMethod
   public void closeBrowser(){
   driver.close();
   }
}
```

ENCAPSULATION

Encapsulation is a mechanism of binding code and data together in a single unit.

In other words these classes will have private instance variables and corresponding public getter and setter methods. All the POM classes in any framework are an example of Encapsulation. In the shown POM classes, we declare the data members using @FindBy and initialization of data members will be done using constructor to utilize those methods.

```
public class LoginPage {
private WebDriver driver;
public LoginPage(WebDriver driver) {
this.driver = driver;
PageFactory.initElements(driver , this);
}
@FindBy(xpath = "//*[@id = 'username']")
private WebElement userName;
@FindBy(xpath = "//*[@id = 'password']")
private WebElement userPass;
@FindBy(xpath = "//*[@id = 'login']")
private WebElement loginbutton;
```

```
public void setUserName(String username){
username.sendKeys(username);
}
public void setPassword (String password){
userPass.sendKeys(password);
}
public void clickLoginButton()
{
loginbutton.click();
}
}
```

Obsqura ZONE

ABSTRACTION

Abstraction is the methodology of hiding the implementation of internal details and showing the functionality to the users.

In Page Object Model design, we write locators such as id, name, xpath etc in each Page Class. We utilize these locators in POM class but we can't see these locators implementation in the tests.

Literally we hide the locators from the tests. Hence abstraction concepts are implemented in our framework with all these locators.

Obsqura ZONE

```
public class LogoutPage {
private WebDriver driver;
  public LoginPage(WebDriver driver) {
  this.driver = driver;
  }
  public void logoutApplication(){
  WebElement element =
 driver.findElement(By.id("submit"));
  element.click();
   }
}
```

INHERITANCE

For every manual test case we should develop a 'TestNG' class.

But in every TestNG class there will be some common methods that need to be reused across the automation frameworks.

As explained earlier we create a Base Class to initialize WebDriver interface, WebDriver waits, Property files, Excels, etc.

In order to reuse these interfaces and classes across the framework we use the OOPs concept 'Inheritance'.

Obsqura ZONE

Extending one class into other class is known as Inheritance or IS-A relation, which is achieved with the keyword Extends.

We cannot directly execute 'BaseTest' class as there are no test methods.

Hence we extend this Base Class into all TestNG classes in the framework.

As we know every test methods of TESTNG class must use @Test annotation.

The name of these methods should start with test and end with the class name.

Now upon invoking every Test case, corresponding POM class will be invoked with the help of Aggregation or HAS-A relationship.

Hence IS-A relation and HAS-A relationship concepts are widely used in any automation frameworks.

```
public class ValidLogin extends BaseTest {
  @Test
  public void testValidLogin() {
  LoginPage login =new LoginPage(driver);  // HAS-A relation with POM class
  login.setUserName("Obsqura");
  login.setPassword("Zone");
  login.clickLoginButton();
  }
}
```

Obsqura ZONE

We set all the instance variables as private so that these private variables cannot be accessed directly by other TestNG classes. Then we generate public getter and setter methods corresponding to this private variable. As we set the variables as private and hide their implementation from other classes, this way we achieve encapsulation in our automation framework.