How to Call a Static and Non-Static Method from Another Class

Example:

```
public class staticAndNonStaticMethods {
    // Static Method
    public static WebDriver staticClickMethod(WebDriver web) {
        web.findElement(By.linkText("Store")).click();
        return web;
    }

    // Non - Static Method
    public WebDriver nonstaticClickMethod(WebDriver web) {
        web.findElement(By.linkText("Support")).click();
        return web;
    }
}
```

Static Method

A static method belongs to the class itself rather than to any specific instance of the class. You can call it using the class name.

- Loaded at class loading time.
- Faster access and do not require an instance.

```
WebDriver driver = staticAndNonStaticMethods.staticClickMethod(driver);
```

Non-Static Method

A non-static method requires an instance of the class to be invoked. You need to create an object of the class.

- Loaded at runtime, upon object instantiation.
- Require an instance of the class to be called.
-

```
staticAndNonStaticMethods obj = new staticAndNonStaticMethods();
WebDriver driver = obj.nonstaticClickMethod(driver);
```

TestNG Annotations

TestNG is a testing framework inspired by JUnit, designed to cover all categories of tests.

Below are some of the key annotations:

@Test : Marks a method as a test method.

@BeforeMethod : Runs before each test method.

@AfterMethod : Runs after each test method.

@BeforeClass : Runs once before the first test method in the current class.

@AfterClass : Runs once after all test methods in the current class executed.

@BeforeTest : Runs before any test method in the specified test tag.

@AfterTest : Runs after all test methods in the specified test tag.

@BeforeSuite : Runs once before all tests in the suite.

@AfterSuite : Runs once after all tests in the suite.

Example:

1. Before Suite/Test/Class/Method

```java
public class LaunchChromeDriver {
    WebDriver driver;

    @BeforeSuite
    public void beforeSuite() {
        System.out.println("This is Before Suite Method....");
    }

    @BeforeTest
    public void beforeTest() {
        System.out.println("This is Before Test Method.....");
    }

    @BeforeClass
    public void beforeClass() {
        System.out.println("This is Before Class.....");
    }

    @BeforeMethod
    public void beforeMethod() {
        System.out.println("This is Before Method.....");
    }
}
```

2. Test Methods

```java
@Test
public void Test1() {
    System.out.println("This is Test1");
}

@Test
public void Test2() {
    System.out.println("This is Test2");
}
```

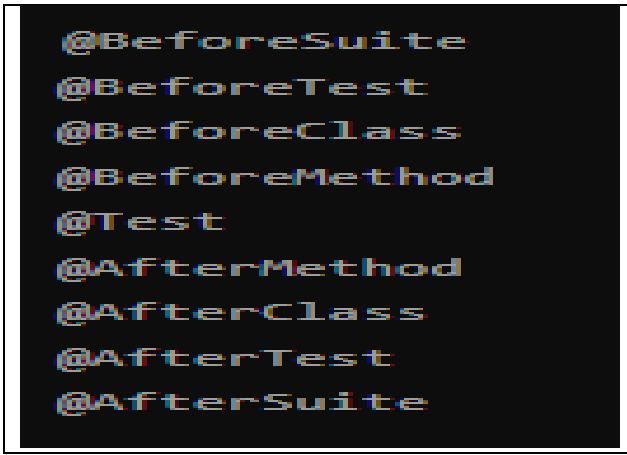
3. After Method/Class/Suite

```java
@AfterMethod
    public void afterMethod() {
    System.out.println("This is After Method.....");
}

@AfterClass
public void afterClass() {
    System.out.println("This is After Class.....");
}

@AfterSuite
public void afterSuite() {
    System.out.println("This is After Suite Method....");
}
```

Execution Order:



Expected Output (in order):

```mathematica
This is Before Suite Method....
This is Before Test Method.....
This is Before Class.....
This is Before Method.....
This is Test1
This is After Method.....
This is Before Method.....
This is Test2
This is After Method.....
This is After Class.....
This is After Suite Method....
```

Why the Order is Important:

- **Isolation of Tests**: By following this sequence, TestNG ensures that no test affects another test's outcome, keeping tests independent.

- **Efficiency**: Using class- and suite-level setup and teardown methods (`@BeforeClass`, `@BeforeSuite`), you avoid repeated expensive setup operations that don't need to be done for every test method.

- **Resource Management**: The teardown annotations (`@AfterMethod`, `@AfterTest`, etc.) are critical for releasing resources, closing connections, and ensuring that each test starts with a clean slate.

- **Reliability and Debugging**: When the execution order is consistent, it becomes easier to debug issues, as you know in which phase something went wrong.

Parallel Execution in TestNG

TestNG allows parallel execution of tests at different levels:

Suite-level Parallelism: Executes multiple test suites concurrently.

Method-level Parallelism: Executes multiple test methods from the same or different classes concurrently.

Class-level Parallelism: Executes multiple test classes concurrently.

Program:

```
public class ClassA {
    @Test
    public void first() {
        System.out.println("class A first " +
            Thread.currentThread().getId());
    }

    @Test
    public void second() {
        System.out.println("class A second " +
            Thread.currentThread().getId());
    }

    @Test
    public void three() {
        System.out.println("class A three " +
            Thread.currentThread().getId());
    }

    @Test
    public void four() {
        System.out.println("class A four " +
            Thread.currentThread().getId());
    }
}
```

```
public class ClassB {
    @Test
    public void first() {
        System.out.println("class B first
            Thread.currentThread().getId());
    }

    @Test
    public void second() {
        System.out.println("class B second
            Thread.currentThread().getId());
    }

    @Test
    public void three() {
        System.out.println("class B three
            Thread.currentThread().getId());
    }

    @Test
    public void four() {
        System.out.println("class B four "
            Thread.currentThread().getId());
    }
}
```

```
public class ClassC {
    @Test
    public void first() {
        System.out.println("class C first " +
            Thread.currentThread().getId());
    }

    @Test
    public void second() {
        System.out.println("class C second " +
            Thread.currentThread().getId());
    }

    @Test
    public void three() {
        System.out.println("class C three " +
            Thread.currentThread().getId());
    }

    @Test
    public void four() {
        System.out.println("class C four " +
            Thread.currentThread().getId());
    }
}
```

```
public class ClassD {
    @Test
    public void first() {
        System.out.println("class D first
            Thread.currentThread().getId());
    }

    @Test
    public void second() {
        System.out.println("class D second
            Thread.currentThread().getId());
    }

    @Test
    public void three() {
        System.out.println("class D three
            Thread.currentThread().getId());
    }

    @Test
    public void four() {
        System.out.println("class D four "
            Thread.currentThread().getId());
    }
}
```

Parallel Executions in Different Levels of XML Files:

Method level Execution.	Class Level Execution	Suite Level Execution

```xml
<suite thread-count="4" name="Suite" parallel="methods">
  <test name="Test">
    <classes>
      <class name="suiteclasslevelexecution.ClassC"/>
      <class name="suiteclasslevelexecution.ClassB"/>
    </classes>
  </test> <!-- Test -->

  <test name="Test2">
    <classes>
      <class name="suiteclasslevelexecution.ClassA"/>
      <class name="suiteclasslevelexecution.ClassD"/>
    </classes>
  </test> <!-- Test2 -->
</suite> <!-- Suite -->
```

```xml
<suite thread-count="3" name="Suite" parallel="classes">
  <test name="Test">
    <classes>
      <class name="suiteclasslevelexecution.ClassC"/>
      <class name="suiteclasslevelexecution.ClassB"/>
      <class name="suiteclasslevelexecution.ClassA"/>
      <class name="suiteclasslevelexecution.ClassD"/>
    </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

```xml
<suite name="MasterSuite" parallel="true" thread-count="2">
    <suite-files>
        <suite-file path="testng.xml"/>
        <suite-file path="testng2.xml"/>
    </suite-files> <!-- Suite-level -->
</suite> <!-- Master Suite -->
```

Class-Level Parallel Execution in TestNG

Class-level parallel execution in TestNG allows multiple test classes to run concurrently in parallel threads. This feature is used to speed up test execution, especially in scenarios where tests are independent of each other and can be run simultaneously without conflicts.

Method-Level Parallel Execution in TestNG

Method-level parallel execution in TestNG allows individual test methods to run concurrently across multiple threads. This is useful for speeding up test execution when each test method is independent and can run in parallel without conflicts.

Suite-Level Parallel Execution in TestNG

Suite-level parallel execution in TestNG allows multiple test suites to run concurrently in different threads. This can be useful when your testing is organized into different test suites, and you want to execute these suites in parallel to reduce the overall execution time.

Advantages:

- **Concurrency**: Entire test Suites,Methods,Classes are run in parallel, with each suite having its own thread.

- **Efficiency**: Maximizes the utilization of system resources by running multiple test Suites,Methods,Classes at the same time.

- **Independent Execution**: Each Suites,Methods,Classes runs independently, which is ideal when suites don't share resources or states.

Locators :

Locators are used to identify web elements in Selenium. Here are the eight locators:

ID: Finds elements by the value of the id attribute.

Name: Finds elements by the value of the name attribute.

Class Name: Finds elements by the value of the class attribute.

Tag Name: Finds elements by the tag name.

Link Text: Finds links by their exact text.

Partial Link Text: Finds links by a portion of the text.

CSS Selector: Finds elements using CSS selectors.

XPath: Finds elements using XML path expressions.

Program:

```
public class EightLocators {
    private static WebDriver driver = new ChromeDriver();

    public static void main(String[] args) throws InterruptedException {
        driver.get("https://github.com/login");

        highlight(driver, driver.findElement(By.id("login_field")));
        highlight(driver, driver.findElement(By.name("password")));
        highlight(driver, driver.findElement(By.className("header-logo")));
        highlight(driver, driver.findElement(By.linkText("Forgot password?")));
        highlight(driver, driver.findElement(By.partialLinkText("Create an ")));
        highlight(driver, driver.findElement(By.tagName("h1")));
        highlight(driver, driver.findElement(By.xpath("/html/body/div[1]/div[3]/main/div/
        highlight(driver, driver.findElement(By.cssSelector("input[name='commit']")));

        driver.close();
    }

    public static void highlight(WebDriver driver, WebElement element) {
        JavascriptExecutor je = (JavascriptExecutor) driver;
        je.executeScript("arguments[0].setAttribute('style','border:2px solid red; backgr
    }
}
```

Output: