@TestInstance(TestInstance.LifeCycle.PER\_CLASS)

This will create only one instance for the class and then run on the same instance.

Using this would help us get rid of the static keyword as only one instance is created.

Assumptions:  
Assumptions is actually a bit similar to these conditional statements.

Write assume true in any of the methods  
assumeTrue(Boolean value).

So, here when you run this I am assuming that this value should be true.  
So, if this value is not true, Hey Junit please don’t run this test.   
Because this test doesn’t make sense in that context. Alrght? Does it make sense, it’s similar to the @enabled @disabled but this gives you a programmatic control, you can actually write a code to figure out this value which executes to say okay what that value is and accordingly will run or skip the test.

So, lets’ say your test requires a server to be up to run, if the server is down it doesn’t even make sense to run the test as the test is anyway going to fail. So this is not your code fault basically, so if the server is down and your test fails this is a problem right ? So your test basically gets dinged . So, in this cases rather than having your test flaky, like I don’t know whether my logic failed or any other external issue. So , we can make an assumption and say that hey junit if the server is up then only run the test else don’t run this test and rather give me an false report that the test has failed.  
Let’s take a simple example regarding that.

Boolean isServerUp = true;  
assumeTrue(isServerUp);

AssertAll: So, basically for a particular method lets say multiply, we need to check various outcomes, like what will happen if I multiply with a –ve number, or with 0 , so we will need to create multiple methods to test each case right. So instead of that we can use this annotation. Let me show you how it looks like

@Test

void multiplyTest() {

Additon additon = new Additon();

// int expected = 2;

// int actual = additon.multiply(1, 2);

// assertEquals(expected, actual);

assertAll(

() -> assertEquals(2, additon.multiply(1, 2),”pas the string”),

() -> assertEquals(0, additon.multiply(1, 0), ”pas the string”),,

() -> assertEquals(-2, additon.multiply(1, -2), ”pas the string”),  
);

}

@Nested--- now there Is one more way by which we can integrated these multiple cases in a method. Without having to write it in a method.   
So, we can have mane methods but we can at the same time put all these methods under a class and use @Nested