JUnit5

JUnit5

- Junit is the most popular unit testing frameworks in Java ecosystem
- JUnit5 is composed of several different modules from three different sub-projects
  - JUnit5 = Junit Platform + Junit Jupiter + Junit Vintage

It defines the TestEngine API for developing new testing frameworks that runs on the platform

It has all new junit annotations and TestEngine implementation to run tests written with these annotations

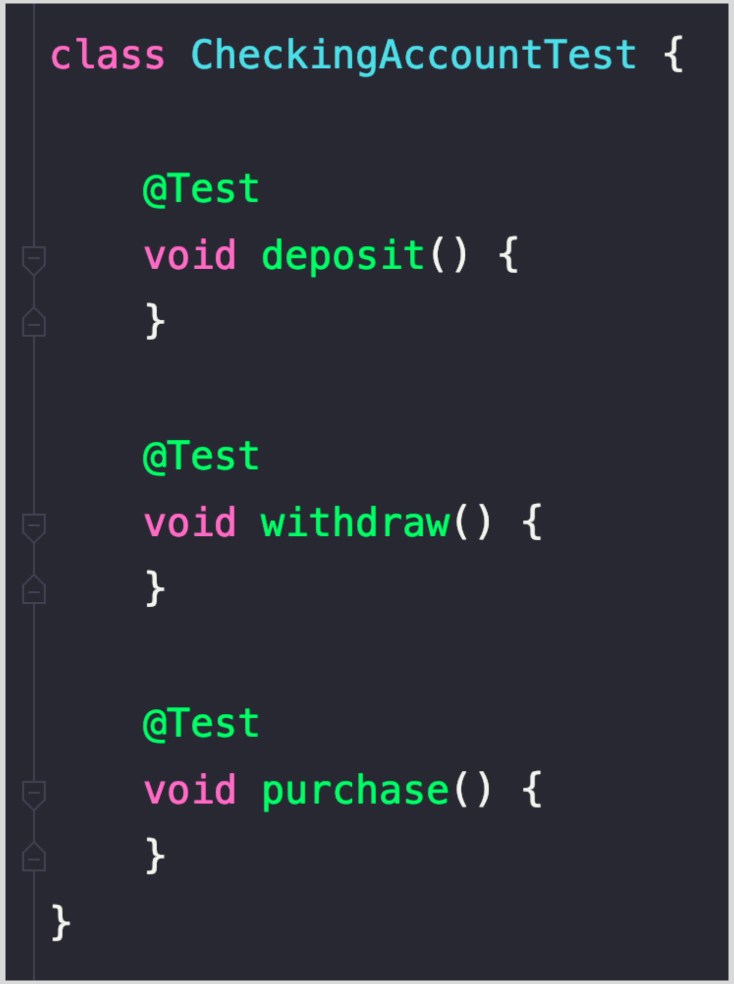
To support running JUnit 3 and JUnit 4 written tests on the JUnit 5 platform.

JUnit 5 vs JUnit 4 Annotations

Feature	JUNIT4	JUNIT5
Declare a test method	@Test	@Test
Execute before all test methods in the current class	@BeforeClass	@BeforeAll
Execute after all test methods in the current class	@AfterClass	@AfterAll
Execute before each test method	@Before	@BeforeEach
Execute after each test method	@After	@AfterEach
Disable a test method/class	@Ignore	@Disabled
Test factory for dynamic tests	NA	@TestFactory
Nested tests	NA	@Nested
Tagging and filtering	@Category	@Tag
Register custom extensions	NA	@ExtendWith

@Test

- It is used to define a certain method is a test method.



```
class CheckingAccountTest {

    @Test
    void deposit() {
    }

    @Test
    void withdraw() {
    }

    @Test
    void purchase() {
    }
}
```

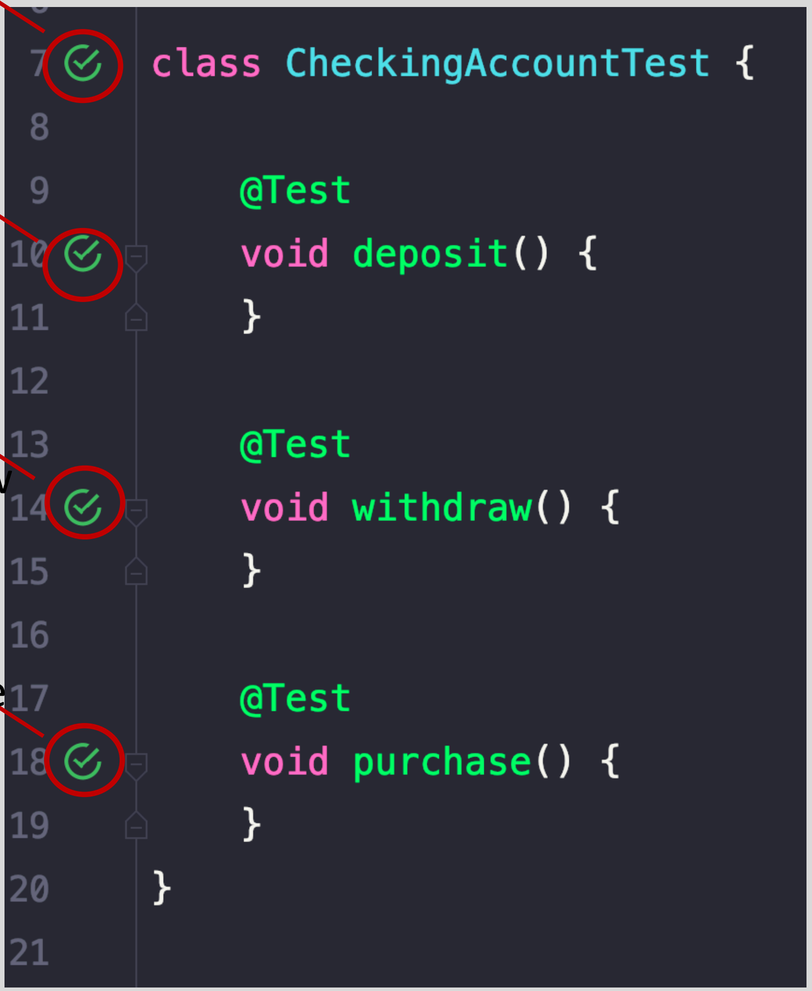
Running Tests

Execute all methods

Execute only deposit

Execute only withdraw

Execute only purchase

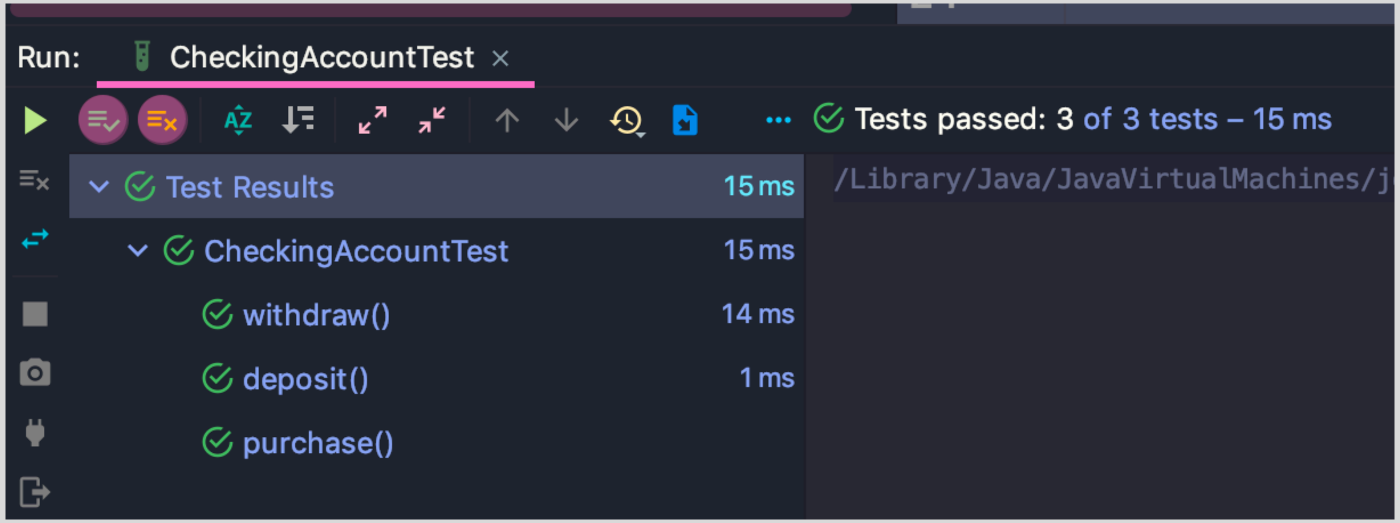


```
class CheckingAccountTest {

    @Test
    void deposit() {
    }

    @Test
    void withdraw() {
    }

    @Test
    void purchase() {
    }
}
```



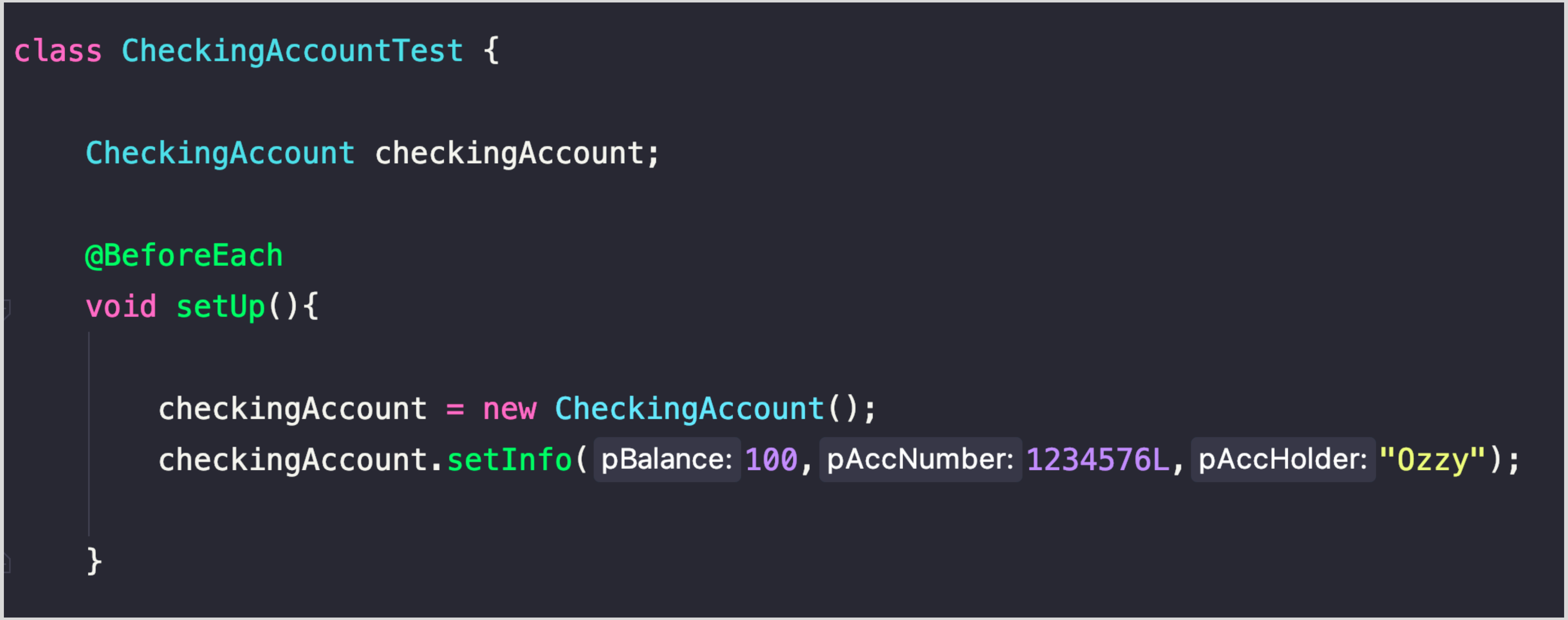
```
Run:  CheckingAccountTest ×
Tests passed: 3 of 3 tests – 15 ms
Test Results                 15 ms    /Library/Java/JavaVirtualMachines/j
  CheckingAccountTest        15 ms
    withdraw()               14 ms
    deposit()                 1 ms
    purchase()
```

Test Results

@BeforeEach

- It is used to signal that the annotated method should be executed before each @Test method in the current class.

```java
@BeforeEach
public void initEach(){
    System.out.println("Before Each initEach() method called");
}
```



```
class CheckingAccountTest {

    CheckingAccount checkingAccount;

    @BeforeEach
    void setUp(){

        checkingAccount = new CheckingAccount();
        checkingAccount.setInfo( pBalance: 100, pAccNumber: 1234576L, pAccHolder: "Ozzy");

    }
```

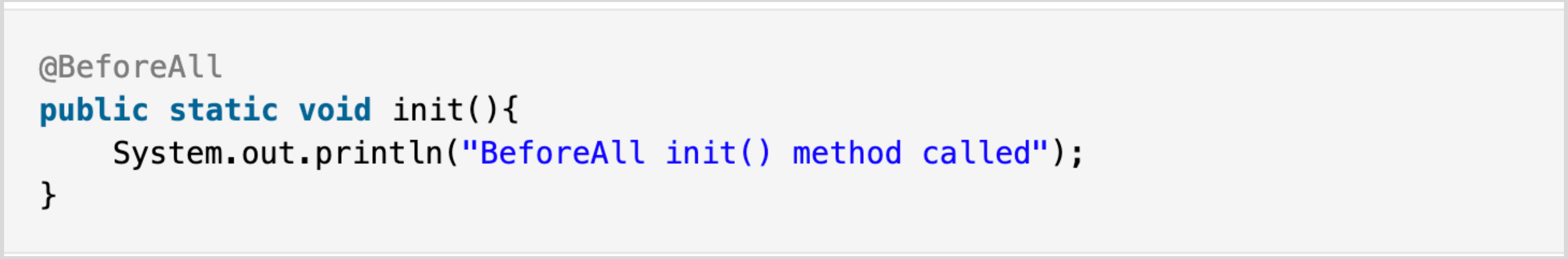
@AfterEach

- It is used to signal that the annotated method should be executed after each @Test method in the current class

```java
@AfterEach
public void cleanUpEach(){
    System.out.println("After Each cleanUpEach() method called");
}
```

@BeforeAll

- It is used to signal that the annotated method should be executed before all tests in the current test class.



```java
@BeforeAll
public static void init(){
    System.out.println("BeforeAll init() method called");
}
```

@AfterAll

- It is used to signal that the annotated method should be executed after all tests in the current test class

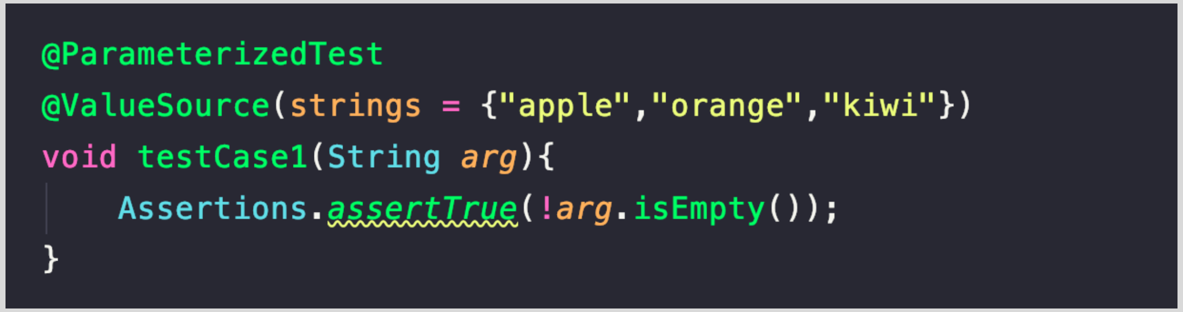
```java
@AfterAll
public static void cleanUp(){
    System.out.println("After All cleanUp() method called");
}
```

Parameterized Tests(@ParameterizedTest)

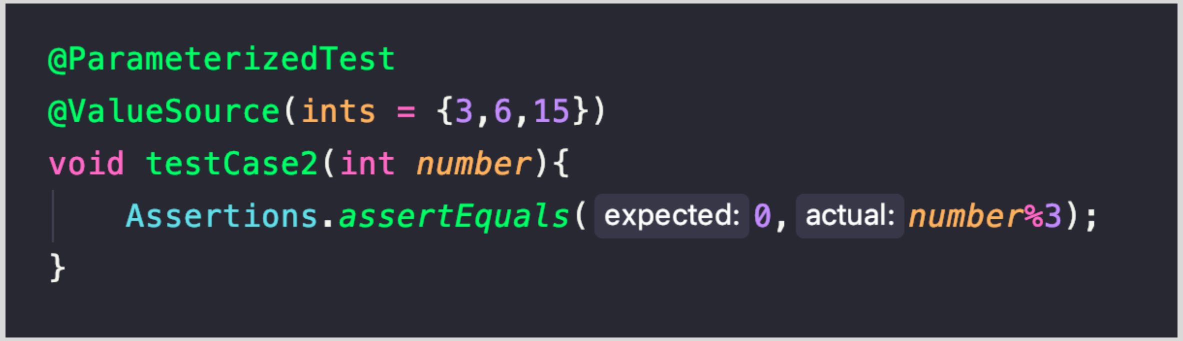
- @ValueSource
- @MethodSource
- @CsvSource
- @CsvFileSource

@ValueSource

- It is used to provide a single parameter per test method
- It lets you specify an array of literals of primitive types



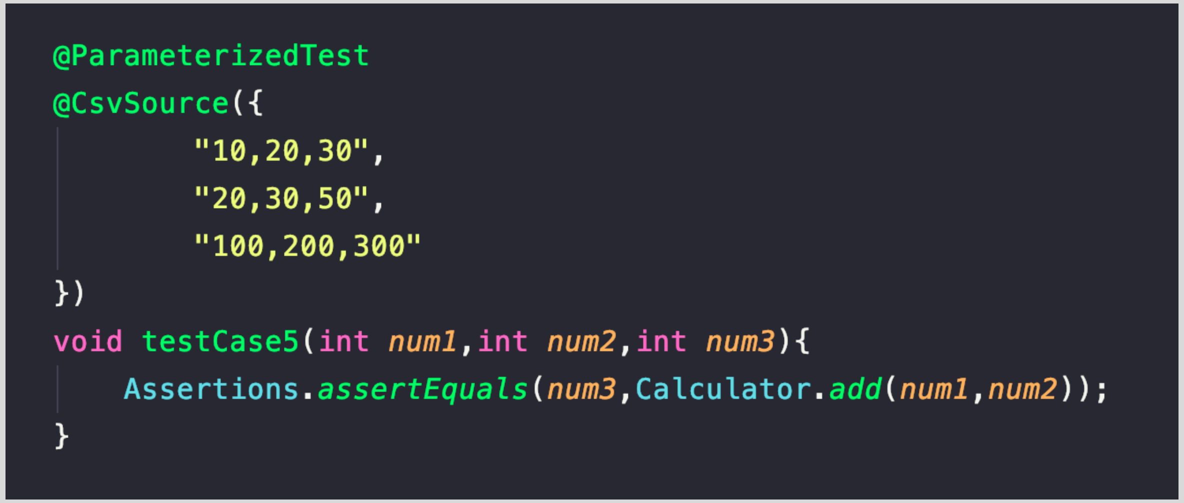
```java
@ParameterizedTest
@ValueSource(strings = {"apple","orange","kiwi"})
void testCase1(String arg){
    Assertions.assertTrue(!arg.isEmpty());
}
```



```java
@ParameterizedTest
@ValueSource(ints = {3,6,15})
void testCase2(int number){
    Assertions.assertEquals( expected: 0, actual: number%3);
}
```

@CsvSource

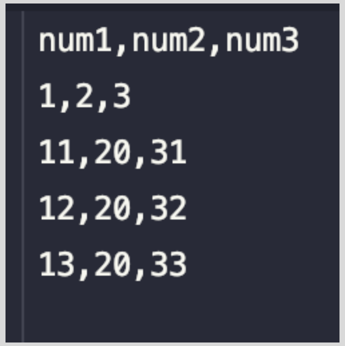
- It is used to run tests that take a comma-separated values as arguments.



```java
@ParameterizedTest
@CsvSource({
        "10,20,30",
        "20,30,50",
        "100,200,300"
})
void testCase5(int num1,int num2,int num3){
    Assertions.assertEquals(num3,Calculator.add(num1,num2));
}
```

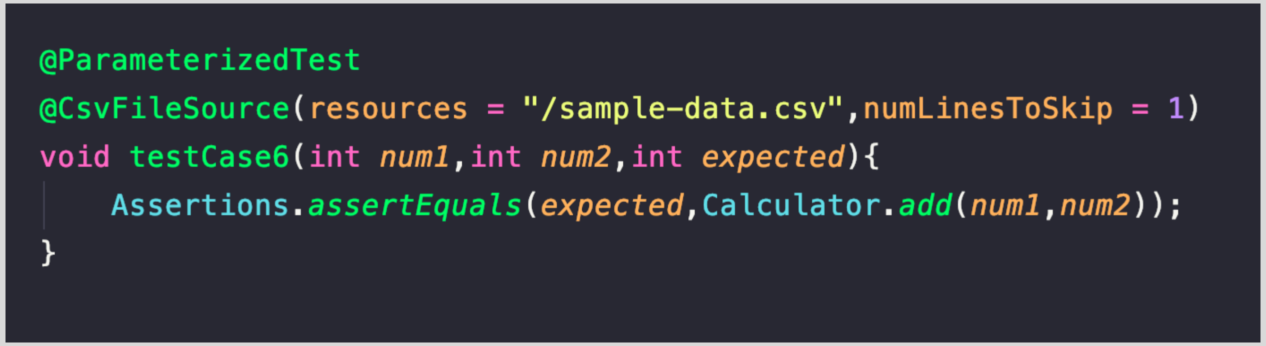
@CsvFileSource

- If we have to write a lot of test data in the test code it can make test less readable. One solution to this is to provide the data in an external CSV file. Each line from the file works as a list of parameters



```
num1,num2,num3
1,2,3
11,20,31
12,20,32
13,20,33
```

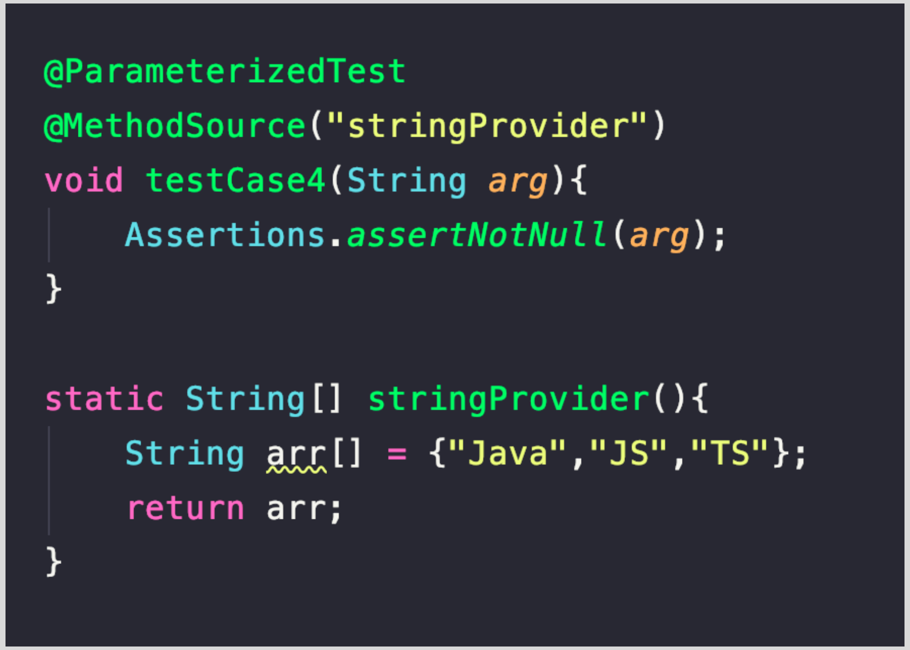
sample-data.csv



```java
@ParameterizedTest
@CsvFileSource(resources = "/sample-data.csv",numLinesToSkip = 1)
void testCase6(int num1,int num2,int expected){
    Assertions.assertEquals(expected,Calculator.add(num1,num2));
}
```

@MethodSource

- It is used to specify a factory method for test arguments.
- This method can be present in the same class or any other class too.
- The factory method should be static and return Stream,Iterable OR array elements.



```java
@ParameterizedTest
@MethodSource("stringProvider")
void testCase4(String arg){
    Assertions.assertNotNull(arg);
}

static String[] stringProvider(){
    String arr[] = {"Java","JS","TS"};
    return arr;
}
```