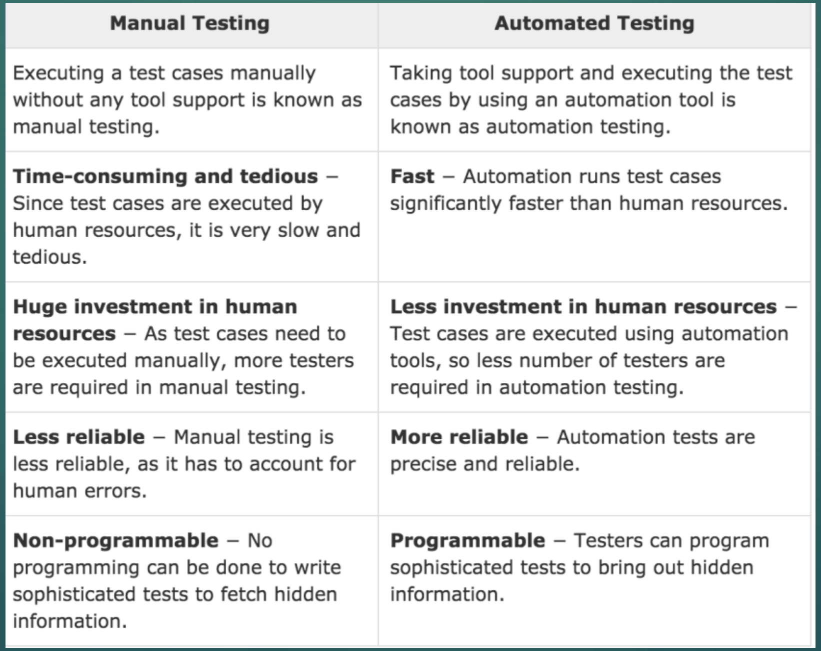
Lab 6

JUNIT TESTING

LAKSHYA TANDON

Testing and its types

- Testing is the process of checking the functionality of an application to ensure it runs as per requirements.
- Unit testing can be done in two ways – manual testing and automated testing.



Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing.	Taking tool support and executing the test cases by using an automation tool is known as automation testing.
Time-consuming and tedious – Since test cases are executed by human resources, it is very slow and tedious.	**Fast** – Automation runs test cases significantly faster than human resources.
Huge investment in human resources – As test cases need to be executed manually, more testers are required in manual testing.	**Less investment in human resources** – Test cases are executed using automation tools, so less number of testers are required in automation testing.
Less reliable – Manual testing is less reliable, as it has to account for human errors.	**More reliable** – Automation tests are precise and reliable.
Non-programmable – No programming can be done to write sophisticated tests to fetch hidden information.	**Programmable** – Testers can program sophisticated tests to bring out hidden information.

JUnit

- JUnit is a **Regression Testing** framework to implement unit testing in Java programming language.
- It can be easily be integrated with any of the following:
  - Eclipse
  - Ant
  - Maven
- JUnit promotes the idea of "first testing then coding", **which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented.**
- It increases the productivity of the programmer and the stability of program code, **which in turn reduces the stress on the programmer and the time spent on debugging.**

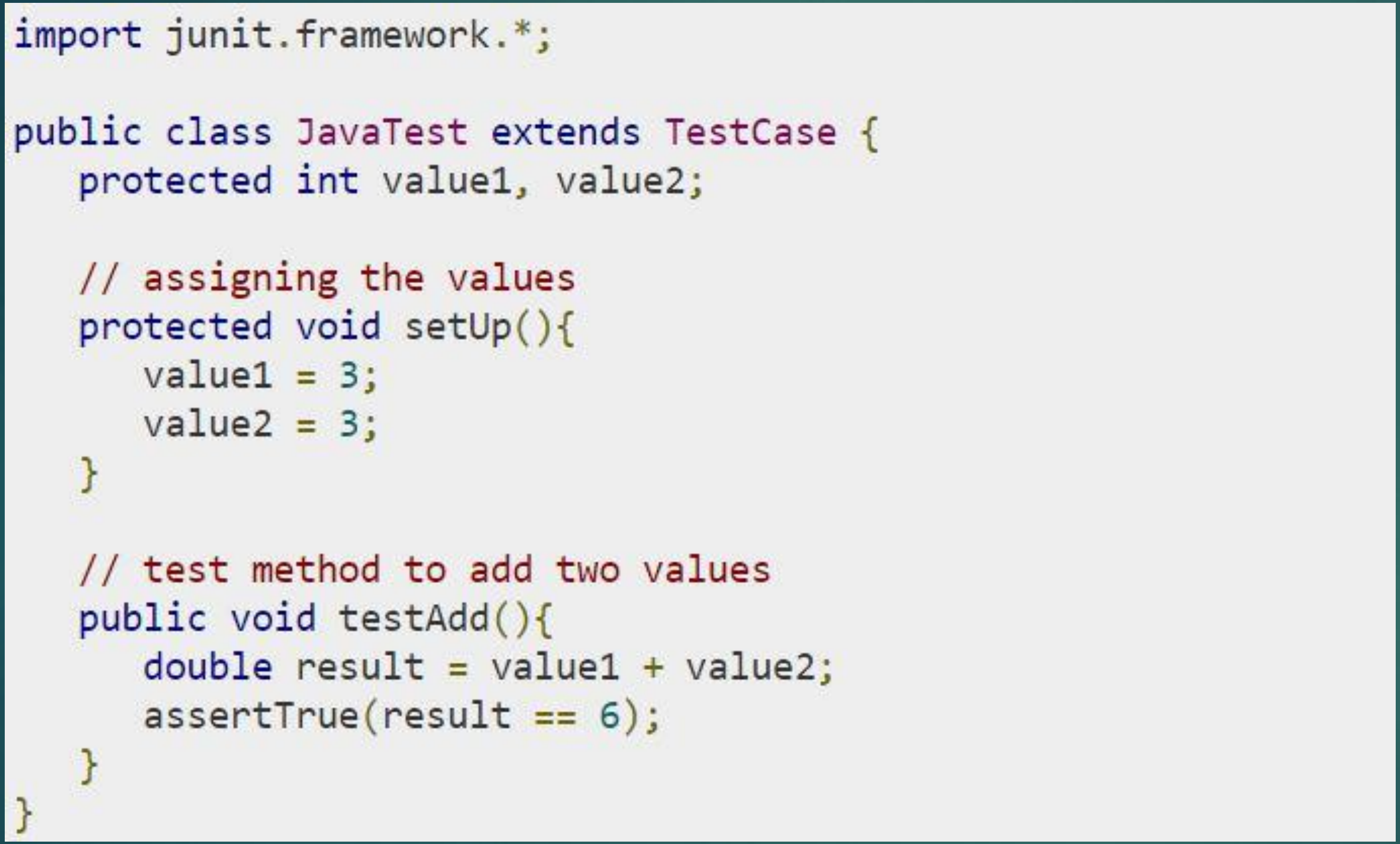
Features of JUnit

- JUnit is an open source framework, which is used for writing and running tests.
- Provides annotations to identify test methods.
- Provides assertions for testing expected results.
- Provides test runners for running tests.
- JUnit tests allow you to write codes faster, which increases quality.
- JUnit is elegantly simple. It is less complex and takes less time.
- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

Features of JUnit Test Framework

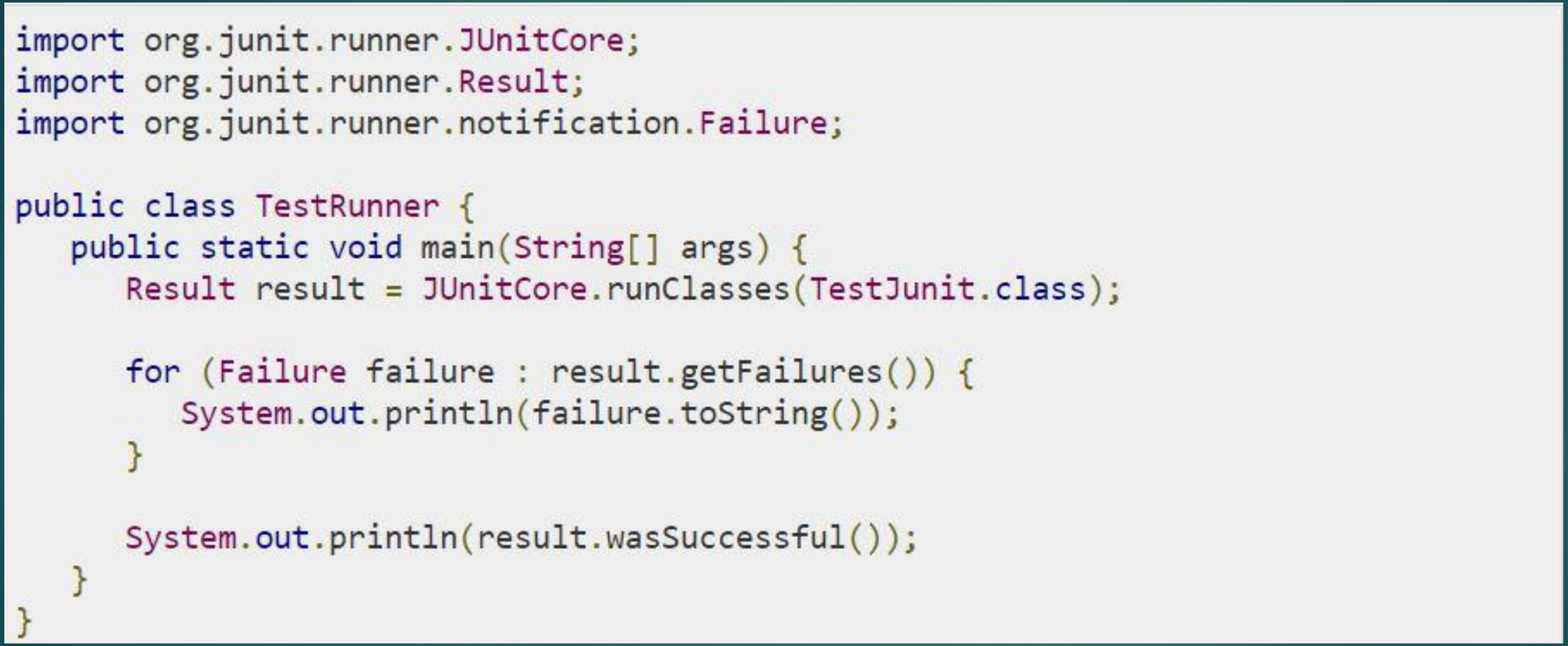
- **Fixtures**-The purpose of a test fixture is to ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable. It includes
  - setUp() method, which runs before every test invocation.
  - tearDown() method, which runs after every test method.
- **Test suites**-A test suite bundles a few unit test cases and runs them together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test.
- **Test runners**-Test runner is used for executing the test cases
- **JUnit classes**-JUnit classes are important classes, used in writing and testing JUnits. Some of the important classes are
  - Assert − Contains a set of assert methods.
  - TestCase − Contains a test case that defines the fixture to run multiple tests.
  - TestResult − Contains methods to collect the results of executing a test case.

Example Fixtures



```java
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   // test method to add two values
   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

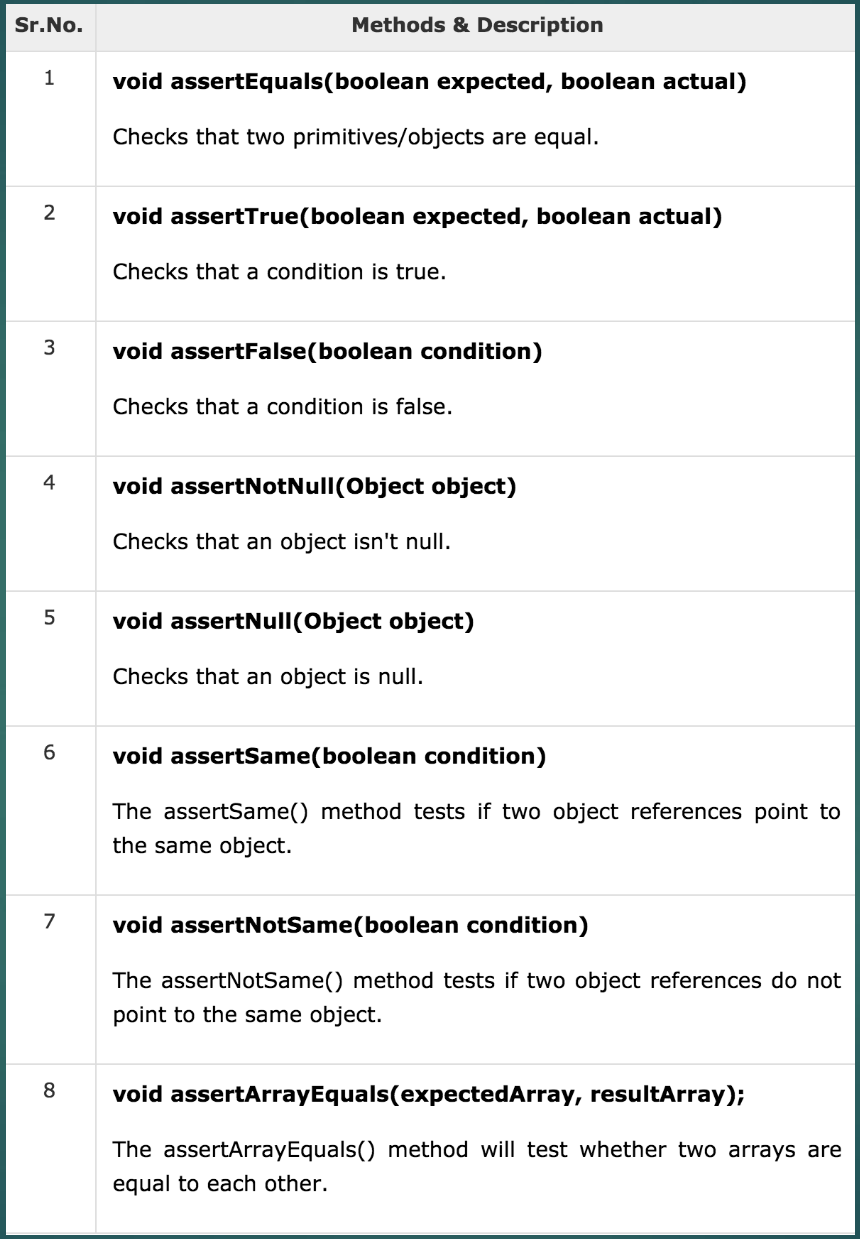
Example Runners – checks TestJunit already exists



```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

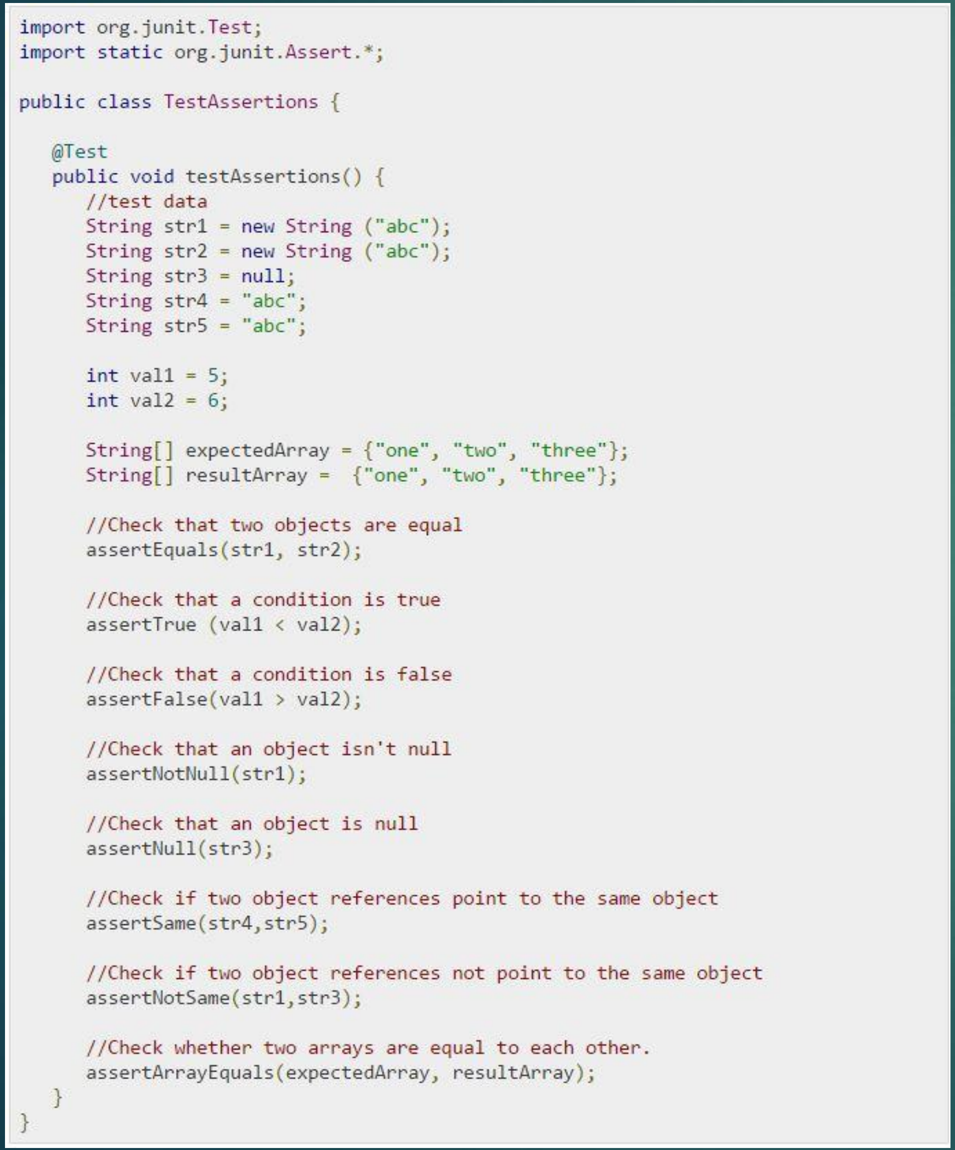
Assertion

- All the assertions are in the Assert class.
- public class Assert extends java.lang.Object
- This class provides a set of assertion methods, useful for writing tests. Only failed assertions are recorded. Some of the important methods of Assert class are as follows



Sr.No.	Methods & Description
1	**void assertEquals(boolean expected, boolean actual)** Checks that two primitives/objects are equal.
2	**void assertTrue(boolean expected, boolean actual)** Checks that a condition is true.
3	**void assertFalse(boolean condition)** Checks that a condition is false.
4	**void assertNotNull(Object object)** Checks that an object isn't null.
5	**void assertNull(Object object)** Checks that an object is null.
6	**void assertSame(boolean condition)** The assertSame() method tests if two object references point to the same object.
7	**void assertNotSame(boolean condition)** The assertNotSame() method tests if two object references do not point to the same object.
8	**void assertArrayEquals(expectedArray, resultArray);** The assertArrayEquals() method will test whether two arrays are equal to each other.

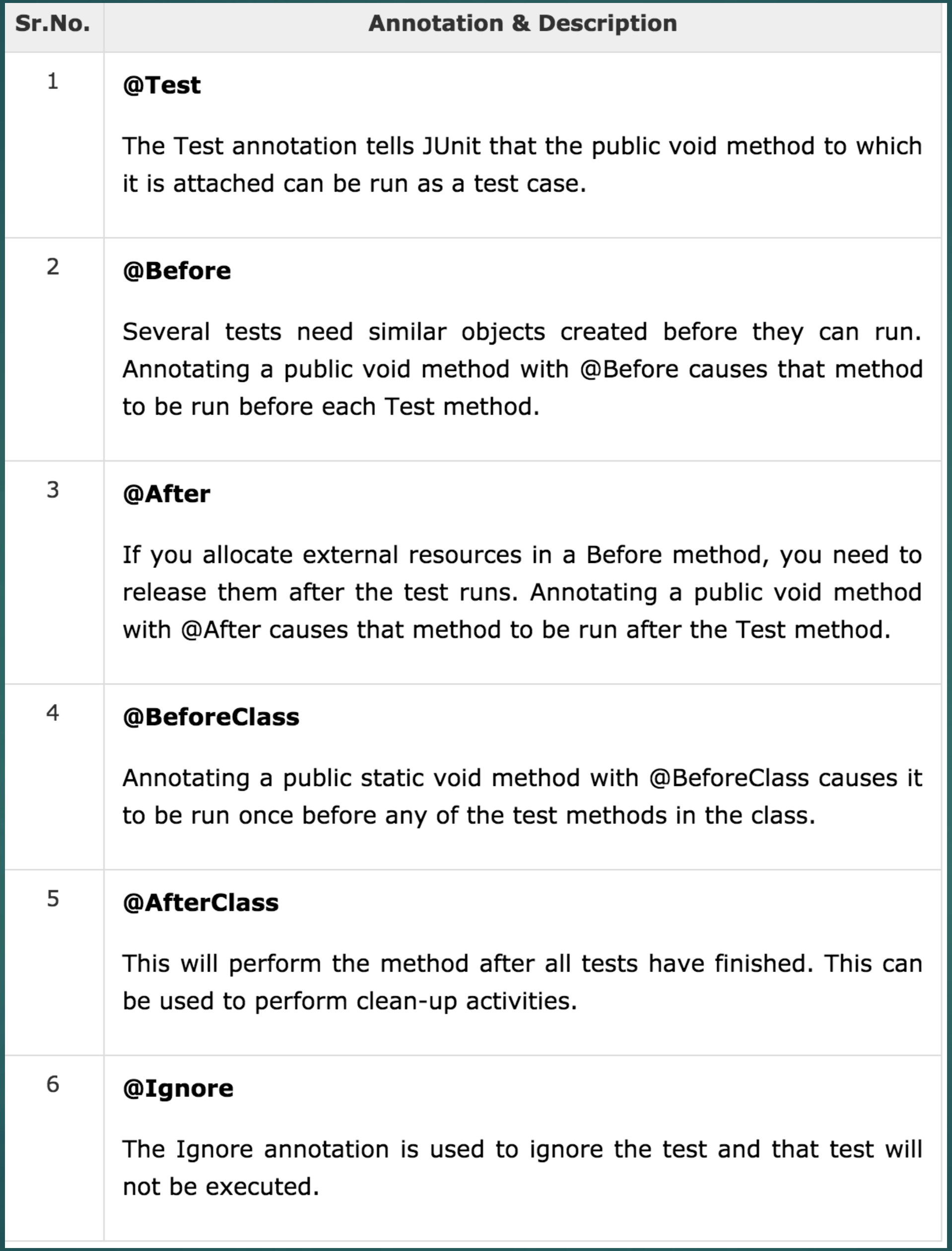
Assertions Example



```java
import org.junit.Test;
import static org.junit.Assert.*;

public class TestAssertions {

   @Test
   public void testAssertions() {
      //test data
      String str1 = new String ("abc");
      String str2 = new String ("abc");
      String str3 = null;
      String str4 = "abc";
      String str5 = "abc";

      int val1 = 5;
      int val2 = 6;

      String[] expectedArray = {"one", "two", "three"};
      String[] resultArray =  {"one", "two", "three"};

      //Check that two objects are equal
      assertEquals(str1, str2);

      //Check that a condition is true
      assertTrue (val1 < val2);

      //Check that a condition is false
      assertFalse(val1 > val2);

      //Check that an object isn't null
      assertNotNull(str1);

      //Check that an object is null
      assertNull(str3);

      //Check if two object references point to the same object
      assertSame(str4,str5);

      //Check if two object references not point to the same object
      assertNotSame(str1,str3);

      //Check whether two arrays are equal to each other.
      assertArrayEquals(expectedArray, resultArray);
   }
}
```

Annotations

- Annotations are like meta-tags that you can add to your code, and apply them to methods or in class. These annotations in JUnit provide the following information about test methods –
  - which methods are going to run before and after test methods.
  - which methods run before and after all the methods, and.
  - which methods or classes will be ignored during the execution.



Sr.No.	Annotation & Description
1	**@Test** The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case.
2	**@Before** Several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before each Test method.
3	**@After** If you allocate external resources in a Before method, you need to release them after the test runs. Annotating a public void method with @After causes that method to be run after the Test method.
4	**@BeforeClass** Annotating a public static void method with @BeforeClass causes it to be run once before any of the test methods in the class.
5	**@AfterClass** This will perform the method after all tests have finished. This can be used to perform clean-up activities.
6	**@Ignore** The Ignore annotation is used to ignore the test and that test will not be executed.