**Program: List of JUnit annotations.**

Annotations are introduced in JUnit4. Here are the list of annotations and its descriptions

Reference: [org.junit java docs](http://junit.sourceforge.net/javadoc/org/junit/package-summary.html)

**@Test:** The Test annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class then invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded.

|  |  |  |
| --- | --- | --- |
| [?](https://www.java2novice.com/junit-examples/junit-annotations/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10 | public class MyTestClass {      @Test      public void myTestMethod() {          /\*\*           \* Use Assert methods to call your methods to be tested.           \* A simple test to check whether the given list is empty or not.           \*/         org.junit.Assert.assertTrue( new ArrayList().isEmpty() );      }   } | |

**@Test (expected = Exception.class):** Sometimes we need to test the exception to be thrown by the test. @Test annotation provides a parameter called 'expected', declares that a test method should throw an exception. If it doesn't throw an exception or if it throws a different exception than the one declared, the test fails.

|  |  |  |
| --- | --- | --- |
| [?](https://www.java2novice.com/junit-examples/junit-annotations/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12 | public class MyTestClass {      @Test(expected=IOException.class)      public void myTestMethod() {          /\*\*           \* this test performs some IO operations, sometimes we may not           \* get access to the resources, then the method should through           \* declared exception.           \*/         ....         ....      }   } | |

**@Test(timeout=100):** Somethimes we need to mesure the performance interms of time. The @Test annotations provides an optional parameter called 'timeout', which causes a test to fail if it takes longer than a specified amount of clock time (measured in milliseconds).

|  |  |  |
| --- | --- | --- |
| [?](https://www.java2novice.com/junit-examples/junit-annotations/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11 | public class MyTestClass {      @Test@Test(timeout=100)      public void myTestMethod() {          /\*\*           \* The IO operation has to be done with in 100 milli seconds. If not,           \* the test should fail.           \*/         ....         ....      }   } | |

**@Before:** When writing tests, it is common to find that several tests need similar objects created before they can run. Annotating a public void method with @Before causes that method to be run before the Test method. The @Before methods of super classes will be run before those of the current class.

|  |  |  |
| --- | --- | --- |
| [?](https://www.java2novice.com/junit-examples/junit-annotations/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class MyTestClass {        List<String> testList;        @Before      public void initialize() {          testList = new ArrayList<String>();      }        @Test      public void myTestMethod() {          /\*\*           \* Use Assert methods to call your methods to be tested.           \* A simple test to check whether the given list is empty or not.           \*/         org.junit.Assert.assertTrue( testList.isEmpty() );      }   } | |

**@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. All @After methods are guaranteed to run even if a Before or Test method throws an exception. The @After methods declared in superclasses will be run after those of the current class.

|  |  |  |
| --- | --- | --- |
| [?](https://www.java2novice.com/junit-examples/junit-annotations/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class MyTestClass {        OutputStream stream;        @Before      public void initialize() {          /\*\*           \* Open OutputStream, and use this stream for tests.           \*/          stream = new FileOutputStream(...);      }        @Test      public void myTestMethod() {          /\*\*           \* Now use OutputStream object to perform tests           \*/         ...         ...      }          @After      public void closeOutputStream() {            /\*\*             \* Close output stream here             \*/            try{              if(stream != null) stream.close();            } catch(Exception ex){              }      }   } | |

**@BeforeClass:** Sometimes several tests need to share computationally expensive setup (like logging into a database). While this can compromise the independence of tests, sometimes it is a necessary optimization. Annotating a public static void no-arg method with @BeforeClass causes it to be run once before any of the test methods in the class. The @BeforeClass methods of superclasses will be run before those the current class.

The annotations @BeforeClass and @Before are same in functionality. The only difference is the method annotated with @BeforeClass will be called once per test class based, and the method annotated with @Before will be called once per test based.

|  |  |  |
| --- | --- | --- |
| [?](https://www.java2novice.com/junit-examples/junit-annotations/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class MyTestClass {        @BeforeClass      public void initGlobalResources() {          /\*\*           \* This method will be called only once per test class.           \*/      }        @Before      public void initializeResources() {          /\*\*           \* This method will be called before calling every test.           \*/      }        @Test      public void myTestMethod1() {          /\*\*           \* initializeResources() method will be called before calling this method           \*/      }        @Test      public void myTestMethod2() {          /\*\*           \* initializeResources() method will be called before calling this method           \*/      }   } | |

**@AfterClass:** If you allocate expensive external resources in a BeforeClass method you need to release them after all the tests in the class have run. Annotating a public static void method with @AfterClass causes that method to be run after all the tests in the class have been run. All @AfterClass methods are guaranteed to run even if a BeforeClass method throws an exception. The @AfterClass methods declared in superclasses will be run after those of the current class.

The annotations @AfterClass and @After are same in functionality. The only difference is the method annotated with @AfterClass will be called once per test class based, and the method annotated with @After will be called once per test based.

|  |  |  |
| --- | --- | --- |
| [?](https://www.java2novice.com/junit-examples/junit-annotations/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class MyTestClass {        @BeforeClass      public void initGlobalResources() {          /\*\*           \* This method will be called only once per test class. It will be called           \* before executing test.           \*/      }        @Test      public void myTestMethod1() {          // write your test code here...          ...          ...      }        @BeforeClass      public void closeGlobalResources() {          /\*\*           \* This method will be called only once per test class. It will be called           \* after executing test.           \*/      }   } | |

**@Ignore:** Sometimes you want to temporarily disable a test or a group of tests. Methods annotated with Test that are also annotated with @Ignore will not be executed as tests. Also, you can annotate a class containing test methods with @Ignore and none of the containing tests will be executed. Native JUnit 4 test runners should report the number of ignored tests along with the number of tests that ran and the number of tests that failed.

You can also use @Ignore annotation at class level.

|  |  |  |
| --- | --- | --- |
| [?](https://www.java2novice.com/junit-examples/junit-annotations/)   |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10 | public class MyTestClass {      @Ignore      @Test      public void myTestMethod() {          /\*\*           \* This test will be ignored.           \*/         org.junit.Assert.assertTrue( new ArrayList().isEmpty() );      }   } | |