**Junit for unit Testing**

Junit for Unit Testing in Java, It’s a testing Framework which is used by the developers to write the test code for their code.

--- It’s an Open Source testing Framework.

--- It allows creating and running the test cases.

--- It has annotations and methods for writing the test cases.

Annotations

* @Test – marks a method as test case
* @BeforeClass -Invoked once before all the test cases
* @Before – Invoked before each test method
* @After--- Invoked before each test method
* @AfterClass- Invoked once after all the test cases
  + org.junit.Assert
  + org.junit.TestCase
  + org.junit.TestResult
  + org.junit.TestSuite

Jar File

Junit4.12.jar or junit5.jar

Assert Methods

---  assertEquals(expected, actual) -  It checks if the two values are equal .

--- assertTrue(condition) – verifies the condition is true

---assertNull(Object) – checks if an object is null

---assertFalse(condition) verifies the condition is false .

--- assertNotNull(Object) - checks if an object is not  null.

**JUnit Annotations Example**

Let’s create a class covering important JUnit annotations with simple print statements and execute it with a test runner class:

**Step 1)** Consider below java class having various methods which are attached to above-listed annotations:

**JunitAnnotationsExample.java**

import static org.junit.Assert.assertEquals;   
import static org.junit.Assert.assertFalse;   
   
import java.util.ArrayList;   
   
import org.junit.After;   
import org.junit.AfterClass;   
import org.junit.Before;   
import org.junit.BeforeClass;   
import org.junit.Ignore;   
import org.junit.Test;  
  
public class JunitAnnotationsExample {  
  
 private ArrayList<String> list;  
  
 @BeforeClass  
 public static void m1() {  
 System.*out*.println("Using @BeforeClass , executed before all test cases ");  
 }  
  
 @Before  
 public void m2() {  
 list = new ArrayList<String>();  
 System.*out*.println("Using @Before annotations ,executed before each test cases ");  
 }  
  
 @AfterClass  
 public static void m3() {  
 System.*out*.println("Using @AfterClass ,executed after all test cases");  
 }  
  
 @After  
 public void m4() {  
 list.clear();  
 System.*out*.println("Using @After ,executed after each test cases");  
 }  
  
 @Test  
 public void m5() {  
 list.add("test");  
 assertFalse(list.isEmpty());  
 assertEquals(1, list.size());  
 }  
  
 @Ignore  
 public void m6() {  
 System.*out*.println("Using @Ignore , this execution is ignored");  
 }  
  
 @Test(timeout = 10)  
 public void m7() {  
 System.*out*.println("Using @Test(timeout),it can be used to enforce timeout in JUnit4 test case");  
 }  
  
 @Test(expected = NoSuchMethodException.class)  
 public void m8() {  
 System.*out*.println("Using @Test(expected) ,it will check for specified exception during its execution");  
 }  
}

**Step 2)**let’s create a test runner class to execute above test:

**TestRunner.java**

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(JunitAnnotationsExample.class);  
 for (Failure failure : result.getFailures()) {  
 System.*out*.println(failure.toString());  
 }  
 System.*out*.println("Result=="+result.wasSuccessful());  
 }  
}

| **S.No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | void assertEquals(boolean expected, boolean actual) | It checks whether two values are equals similar to equals method of Object class |
| 2. | void assertFalse(boolean condition) | functionality is to check that a condition is false. |
| 3. | void assertNotNull(Object object) | “assertNotNull” functionality is to check that an object is not null. |
| 4. | void assertNull(Object object) | “assertNull” functionality is to check that an object is null. |
| 5. | void assertTrue(boolean condition) | “assertTrue” functionality is to check that a condition is true. |
| 6. | void fail() | If you want to throw any assertion error, you have fail() that always results in a fail verdict. |
| 7. | void assertSame([String message] | “assertSame” functionality is to check that the two objects refer to the same object. |
| 8. | void assertNotSame([String message] | “assertNotSame” functionality is to check that the two objects do not refer to the same object. |

**JUnit Assert Class**

This class provides a bunch of assertion methods useful in writing a test case. If all assert statements are passed, test results are successful. If any assert statement fails, test results are failed.

As you seen earlier, below table describes important Assert methods and description:

**JUnit Test Cases Class**

To run multiple test, TestCase class is available in **org.junit.TestCase** packages. @Test annotation tells JUnit that this public void method (Test Case here) to which it is attached can be run as a test case.

Below table shows some important methods available in **org.junit.TestCase**class:

| **S.No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | int countTestCases() | This method is used to count how many number of test cases executed by **run(TestResult tr)** method. |
| 2. | TestResult createResult() | This method is used to create a **TestResult** object. |
| 3. | String getName() | This method returns a string which is nothing but a **TestCase**. |
| 4. | TestResult run() | This method is used to execute a test which returns a **TestResult** object |
| 5. | void run(TestResult result) | This method is used to execute a test having a **TestResult** object which doesn’t returns anything. |
| 6. | void setName(String name) | This method is used to set a name of a **TestCase.** |
| 7. | void setUp() | This method is used to write resource association code. e.g. Create a database connection. |
| 8. | void tearDown() | This method is used to write resource release code. e.g. Release database connection after performing transaction operation. |

**JUnit TestResult Class**

When you execute a test, it returns a result (in the form of **TestResult** object). This TestResult object can be used to analyze the resultant object. This test result can be either failure or successful.

See below table for important methods used in org.junit.TestResult class:

| **S.No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | void addError(Test test, Throwable t) | This method is used if you require add an error to the test. |
| 2. | void addFailure(Test test, AssertionFailedError t) | This method is used if you require add a failure to the list of failures. |
| 3. | void endTest(Test test) | This method is used to notify that a test is performed(completed) |
| 4. | int errorCount() | This method is used to get the error detected during test execution. |
| 5. | Enumeration<TestFailure> errors() | This method simply returns a collection (Enumeration here) of errors. |
| 6. | int failureCount() | This method is used to get the count of errors detected during test execution. |
| 7. | void run(TestCase test) | This method is used to execute a test case. |
| 8. | int runCount() | This method simply counts the executed test. |
| 9. | void startTest(Test test) | This method is used to notify that a test is started. |
| 10. | void stop() | This method is used to test run to be stopped. |

**JUnit Test Suite Class**

If you want to execute multiple tests in a specified order, it can be done by combining all the tests in one place. This place is called as the test suites.

See below table for important methods used in **org.junit.TestSuite** class:

| **S.No.** | **Method** | **Description** |
| --- | --- | --- |
| 1. | void addTest(Test test) | This method is used if you want to add a test to the suite. |
| 2. | void addTestSuite(Class<? extends TestCase> testClass) | This method is used if you want to specify the class while adding a test to the suite. |
| 3. | int countTestCases() | This method is used if you want to count the number of test cases. |
| 4. | String getName() | This method is used to get the name of the test suite. |
| 5. | void run(TestResult result) | This method is used to execute a test and collect test result in **TestResult** object. |
| 6. | void setName(String name) | This method is used to set the name of **TestSuite**. |
| 7. | Test testAt(int index) | This method is used if you want to return the test at given index. |
| 8. | int testCount() | This method is used if you want to return a number of tests in the Suite. |
| 9. | static Test warning(String message) | This method returns a test which will fail and log a warning message. |

**Summary**

* JUnit provides a portable API, which provides all important classes and Selenium annotations useful in writing a unit test.
* **Classes which are very useful while writing a test case**
* **Important and frequently used JUnit annotations list**@Before@BeforeClass@After

@AfterClass

@Test

@Ignore

Debugging in IDE (Intellij /Eclipse)

----- Set the break points.

----- Start the program in debug mode

---- Inspect the variables during the execution

---- Going through the code

*Garbage Collection* tracks each and every object available in the JVM heap space, and removes the unused ones.

Basically, *GC* works in two simple steps, known as Mark and Sweep:

* **Mark –**this is where the garbage collector identifies which pieces of memory are in use and which aren’t.
* **Sweep –**this step removes objects identified during the “mark” phase.

**Advantages:**

* No manual memory allocation/deallocation handling because unused memory space is automatically handled by *GC*
* No overhead of handling [***Dangling Pointer***](https://en.wikipedia.org/wiki/Dangling_pointer)
* Automatic [***Memory Leak***](https://en.wikipedia.org/wiki/Memory_leak) management (*GC* on its own can’t guarantee the full proof solution to memory leaking; however, it takes care of a good portion of it)

**Disadvantages**

* Since JVM has to keep track of object reference creation/deletion, this activity requires more CPU power than the original application. It may affect the performance of requests which require large memory.
* Programmers have no control over the scheduling of CPU time dedicated to freeing objects that are no longer needed.
* Using some GC implementations might result in the application stopping unpredictably.
* Automatized memory management won’t be as efficient as the proper manual memory allocation/deallocation.

**GC Implementations**

JVM has four types of *GC* implementations:

* Serial Garbage Collector
* Parallel Garbage Collector
* G1 Garbage Collector
* Z Garbage Collector

**3.1. Serial Garbage Collector**

This is the simplest GC implementation, as it basically works with a single thread. As a result, **this *GC* implementation freezes all application threads when it runs**. Therefore, it’s not a good idea to use it in multi-threaded applications, like server environments.

However, there was [an excellent talk](https://www.infoq.com/presentations/JVM-Performance-Tuning-twitter-QCon-London-2012) given by *Twitter* engineers at QCon 2012 about the performance of *Serial Garbage Collector,* which is a good way to understand this collector better.

The Serial GC is the garbage collector of choice for most applications that don’t have small pause time requirements and run on client-style machines. To enable *Serial Garbage Collector*, we can use the following argument:

java -XX:+UseSerialGC -jar Application.javaCopy

**3.2. Parallel Garbage Collector**

It’s the default *GC* of the *JVM*from Java 5 until Java 8 and is sometimes called Throughput Collectors. Unlike *Serial Garbage Collector*, it **uses multiple threads for managing heap space,** but it also freezes other application threads while performing *GC*.

If we use this *GC*, we can specify maximum garbage collection *threads and pause time, throughput, and footprint* (heap size).

The numbers of garbage collector threads can be controlled with the command-line option *-XX:ParallelGCThreads=<N>*.

The maximum pause time goal (a hint to the garbage collector that [pause times](https://docs.oracle.com/en/java/javase/17/gctuning/ergonomics.html#GUID-C15D02E5-E783-4A0D-8A6B-D57A36A23F77) of *<N>* milliseconds or less are desired) is specified with the command-line option *-XX:MaxGCPauseMillis=<N>*.

The time spent doing garbage collection versus the time spent outside of garbage collection is called the maximum throughput target and can be specified by the command-line option *-XX:GCTimeRatio=<N>.*

The maximum heap footprint (the amount of heap memory that a program requires while running) is specified using the option *-Xmx<N>.*

To enable *Parallel Garbage Collector*, we can use the following argument:

java -XX:+UseParallelGC -jar Application.javaCopy

**3.3. G1 Garbage Collector**

*G1 (Garbage First) Garbage Collector* is designed for applications running on multi-processor machines with large memory space. It’s available from the *JDK7 Update 4* and in later releases.

Unlike other collectors, the *G1* collector partitions the heap into a set of equal-sized heap regions, each a contiguous range of virtual memory. When performing garbage collections, *G1* shows a concurrent global marking phase (i.e. phase 1, known as *Marking)* to determine the liveness of objects throughout the heap.

After the mark phase is complete, *G1* knows which regions are mostly empty. It collects in these areas first, which usually yields a significant amount of free space (i.e. phase 2, known as *Sweeping).* That’s why this method of garbage collection is called Garbage-First.

To enable the *G1 Garbage Collector*, we can use the following argument:

java -XX:+UseG1GC -jar Application.javaCopy

**3.4. Java 8 Changes**

*Java 8u20* has introduced one more *JVM* parameter for reducing the unnecessary use of memory by creating too many instances of the same *String.* This optimizes the heap memory by removing duplicate *String* values to a global single *char[]* array.

We can enable this parameter by adding ***-XX:+UseStringDeduplication*** as a *JVM* parameter.

**3.5. Z Garbage Collector**

[*ZGC (Z Garbage Collector)*](https://www.baeldung.com/jvm-zgc-garbage-collector)is a scalable low-latency garbage collector that debuted in Java 11 as an experimental option for Linux. *JDK* 14 introduced  *ZGC* under the Windows and macOS operating systems. *ZGC* has obtained the production status from Java 15 onwards.

*ZGC* performs all expensive work concurrently,**without stopping the execution of application threads for more than 10 ms**, which makes it suitable for applications that require low latency. It uses**load barriers with colored pointers** to perform concurrent operations when the threads are running, and they’re used to keep track of heap usage.

Reference coloring (colored pointers) is the core concept of *ZGC*. It means that *ZGC* uses some bits (metadata bits) of reference to mark the state of the object. It also **handles heaps ranging from 8MB to 16TB in size**. Furthermore, pause times don’t increase with the heap, live-set, or root-set size.

Similar to *G1, Z Garbage Collector*partitions the heap, except that heap regions can have different sizes.

To enable the *Z Garbage Collector*, we can use the following argument in *JDK* versions lower than 15:

java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC Application.javaCopy

From version 15 on, we don’t need experimental mode on:

java -XX:+UseZGC Application.javaCopy

We should note that *ZGC* isn’t the default Garbage Collector.

**What is an Object?**

* An object is a collection of data and actions.
* An object is an instance of a class.
* Objects have states and behaviors.

In the real-world, we can find so many objects around us, for example Cars, Birds, Humans etc. All these objects have a state and behavior. If we consider a Car then it have some data speed, lights on, direction, etc. and have some actions turn right, accelerate, turn lights on, etc.

If you compare the java object with a real world object, both of them have similar characteristics. Java objects also have a state and behavior. A Java object's state is stored in fields and behavior is shown via methods.

Technically speaking Car, Bird and Human are considered as Class in Java.  Brian Christopher is an object of human and Vehicle XKMV-669 is the object of car.

**Creating Object**

Using new keyword is the most common way to create an object in java.

***Syntax:-***

*ClassName Obj.Name = new ClassName();*

*// Human brianChristopher= new Human();*

*// Car  vehicleXKMV\_669 = new Car();*

The first statement creates a new Human object and second statement creates Car object. This single statement performs three actions, Declaration, Instantiation, and Initialization.

Here, Human brianChristopher is a variable declaration which simply declares to the compiler that the name brianChristopher will be used to refer to an object whose type is Human, the new operator instantiates the Human class (thereby creating a new Human object), and Human initializes the object.

**Object Lifecycle**

In Java, it has seven states in Object lifecycle. They are,

1. Created
2. In use
3. Invisible
4. Unreachable
5. Collected
6. Finalized
7. De-allocated

**Created**

The following are the some actions performed when an object is created. New memory is allocated for an object.

Once the object has been created, assuming that it is assigned to some variable and then it directly moves to the In Use state.

**In use**

Objects that are held by at least one strong reference are considered to be “In Use”.

**Invisible**

An object is in the “Invisible” state when there are no longer any strong references that are accessible to the program, even though there might still be references.

**Unreachable**

An object enters an “unreachable” state when no more strong references to it exist. When an object is unreachable then it is a state for collection.

It is important to note that not just any strong reference will hold an object in memory. These must be references that chain from a garbage collection root.

Garbage collection roots are a special class of variable that includes,Temporary variables on the stack

**Collected**

An object is in the “collected” state when the garbage collector has recognized an object as unreachable and readies it for final processing as a precursor to de-allocation. If the object has a finalize method, then it is marked for finalization.

**Finalized**

An object is in the “finalized” state if it is still unreachable after it’s finalize method, if any, has been run. A finalized object is awaiting de-allocation. If you are considering using a finalizer to ensure that important resources are freed in a timely manner, you might want to reconsider. To lengthening object lifetimes, finalize methods can increase object size.

**De-allocated**

The de-allocated state is the final step in garbage collection. If an object is still unreachable after all the above work has done, then this is the state for de-allocation.

**Dangling pointer**

More generally, dangling references and wild references are references that do not resolve to a valid destination. Dangling pointers arise during object.