1. What is unit testing?

Unit testing is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program, function, procedure, etc. In object-oriented programming, the smallest unit is a method, which may belong to a base/ super class, abstract class or derived/ child class.

2. What is the difference between manual testing and automated testing?

Manual testing is testing of the software where tests are executed manually by a QA Analysts. It is performed to discover bugs in software under development. However, In Automated Software Testing, testers write code/test scripts to automate test execution. Testers use appropriate automation tools to develop the test scripts and validate the software. The goal is to complete test execution in a less amount of time.

In Manual testing, the tester checks all the essential features of the given application or software. In this process, the software testers execute the test cases and generate the test reports without the help of any automation software testing tools. It is a classical method of all testing types and helps find bugs in software systems. It is generally conducted by an experienced tester to accomplish the software testing process.

In Automated Software Testing, testers write code/test scripts to automate test execution. Testers use appropriate automation tools to develop the test scripts and validate the software. The goal is to complete test execution in a less amount of time.

Automated testing entirely relies on the pre-scripted test which runs automatically to compare actual result with the expected results. This helps the tester to determine whether or not an application performs as expected.

4. What are the features of JUnit?

JUnit is a unit testing framework for Java programming language. It plays a crucial role test-driven development and is a family of unit testing frameworks collectively known as xUnit. 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. This approach is like "test a little, code a little, test a little, code a little." 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.

The features of JUnit are:-

* 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.

5. What are the important JUnit annotations? And its usage in coding

The important annotations and its usage are: -

* @Test - This annotation is a replacement of org.junit.TestCase which indicates that public void method to which it is attached can be executed as a test Case.
* @Before - This annotation is used if you want to execute some statement such as preconditions before each test case.
* @BeforeClass - This annotation is used if you want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases.
* @After - This annotation can be used if you want to execute some statements after each Test Case for e.g resetting variables, deleting temporary files, variables, etc.
* @AfterClass - This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.
* @Ignores - This annotation can be used if you want to ignore some statements during test execution for e.g. disabling some test cases during test execution.
* @Test(timeout=500) - This annotation can be used if you want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time.
* @Test(expected=IllegalArgumentException.class) - This annotation can be used if you want to handle some exception during test execution. For, e.g., if you want to check whether a particular method is throwing specified exception or not.

6. What does Assert class?

Assert is a method useful in determining Pass or Fail status of a test case, The assert methods are provided by the class org.junit.Assert which extends java.lang.Object class.There are various types of assertions like Boolean, Null, Identical etc. Junit provides a class named Assert, which provides a bunch of assertion methods useful in writing test cases and to detect test failure.

7. What is Code Coverage?

Code coverage is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases. It also creates some test cases to increase coverage and determining a quantitative measure of code coverage. In most cases, code coverage system gathers information about the running program. It also combines that with source code information to generate a report about the test suite's code coverage.

8. What are the best practices to perform Unit Testing?

The best practices to perform Unit Testing are:-

* Unit Tests Should Be Trustworthy - The test must fail if the code is broken and only if the code is broken. If it doesn't, we cannot trust what the test results are telling us.
* Unit Tests Should Be Maintainable and Readable - When production code changes, tests often need to be updated, and possibly debugged as well. So, it must be easy to read and understand the test, not only for whoever wrote it, but for other developers as well. Always organize and name your tests for clarity and readability.
* Unit Tests Should Verify a Single-Use Case - Good tests validate one thing and one thing only, which means that typically, they validate a single use-case. Tests that follow this best practice are simpler and more understandable, and that is good for maintainability and debugging. Tests that validate more than one thing can easily become complex and time-consuming to maintain. Don't let this happen.
* Unit Tests Should Be Isolated - Tests should be runnable on any machine, in any order, without affecting each other. If possible, tests should have no dependencies on environmental factors or global/external state. Tests that have these dependencies are harder to run and usually unstable, making them harder to debug and fix, and end up costing more time than they save.
* Unit Tests Should Be Automated - Make sure tests are being run in an automated process. This can be daily, or every hour, or in a Continuous Integration or Delivery process. The reports need to be accessible to and reviewed by everyone on the team. As a team, talk about which metrics you care about: code coverage, modified code coverage, number of tests being run, performance, etc. A lot can be learned by looking at these numbers, and a big shift in those numbers often indicates regressions that can be addressed immediately.

9. What is Mocking?

Mocking is primarily used in unit testing. An object under test may have dependencies on other (complex) objects. To isolate the behaviour of the object you want to replace the other objects by mocks that simulate the behaviour of the real objects. This is useful if the real objects are impractical to incorporate into the unit test. In short, mocking is creating objects that simulate the behaviour of real objects.