Best Practices for Unit Testing with Java

Testing is a very important aspect of development and can largely determine the fate of an application. Good testing can catch application-killing issues early on, but poor testing invariably leads to failure and downtime.

Helpful tips for better unit testing in Java.

1. The unit test code should adhere to Four Principles of Simple Design.
   a. Runs and covers all tests
   b. Contains no duplication
   c. Expresses intent of programmers
   d. Minimizes number of classes and methods

2. Use Test Driven Development Judiciously!: The goal is to write tests that cover all the requirements as against simply writing code first that may not even meet the requirements.
3. Write tests for methods that have the fewest dependencies first, and work your way up.
4. Name the unit tests clearly and consistently.
5. Do not initialize the unit test in test class constructor; use an @BeforeEach method instead.
6. Unit Tests Should Be Maintainable and Readable
7. Unit Test code should be easy to read and analyze
8. Name of The Unit Test: The name of the test should include the condition being tested and if necessary, the result.
9. Name unit tests using a convention that includes the method and condition being tested
10. Define the unit tests that run quickly
11. Test only one code unit at a time.
12. Make each test independent of all the other test cases.
13. Unit Tests Should Be Isolated
14. Assert Only What You Want to Test
15. Insert Test Data Right In The Test Method
16. Favor Composition over Inheritance in test case implementations
17. Ensure that test code is separated from production code
18. Do not print anything out inside unit tests at execution time.
19. Do not use static members in a test class. If you have used then re-initialize for each test case: Static members make unit test methods dependent. Instead, write test methods that are completely independent.
20. Mock out all external services and state.
21. Use exact matching when using a mocking framework with mock objects: For example, avoid using Mockito's methods that start with any; prefer configuring and verifying exact parameter values.

22. Make every implementation Testable.
23. Unit Tests Should Verify a Single Use Case: one assertion in one test method.
24. Do not rely on indirect testing: write another test case for each scenario.
25. No dependencies between test conditions.
26. Don't assume the order in which the tests would be run.
27. Avoid Test Interdependence.
28. Define one assertion per Test Method: When a unit test fails, it is much easier to determine what went wrong. If a failed unit test has three assertions, further effort is required to determine which one failed. If a failed unit test has only one assertion, no such effort is required.

When a unit test performs more than one assertion, it is not guaranteed that all of the assertions occur. In particular, if an unchecked exception occurs, the assertions after the exception do not happen; instead, JUnit marks the unit test method as having an error and proceeds to the next test method.

29. Use the most appropriate assertions inside test methods

---

a. **strongest assertions** - asserting on the return value of a method
b. **strong assertions** - verifying that vital dependent mock objects were interacted with correctly
c. **weak assertions** - verifying that non-vital dependent mock objects (such as a logger) were interacted with correctly
d. **non-existent assertions**

30. Define One Condition per Unit Test for testing
31. No Exception Handling within a unit test.
32. Use annotated markers to test for exceptions.
33. Do not write catch blocks that exist only to pass a test.
34. Do not write catch blocks that exist only to print a stack trace
35. Ensure that unit tests run completely in-memory: Avoid the unit tests that make HTTP requests, access a database, or read from the filesystem. These activities are too slow and too unreliable, so they are better left to other types of tests, such as functional or end to end tests.
36. Tests that read from the file system: configure toread from relative path and not absolute path.
37. Do not skip unit tests in final testing cycle, Ensure that every unit test is executed in final sprint cycle.
38. Remove the fakes,mocks, test doubles, hard coded values in production testing.

---

39. Measure Code Coverage frequently: Code coverage measures (in percentage e) how much of the code is executed when the unit tests are run. Normally, code with high coverage has a decreased chance of containing undetected bugs, as more of its source code has been executed in the course of testing. Some best practices for measuring code coverage include:
    a. Use a code coverage tool like Clover, Corbetura, JaCoCo, or Sonar. Using a tool can improve testing quality, as these tools point out areas of the code that are untested, allowing to develop additional tests to cover these areas.
    b. Whenever new functionality is written, immediately write new tests to implement with TDD.
    c. Ensure that there are test cases that cover all the branches of the code, i.e. if/else statements.

40. Externalize test data wherever possible.(Parameterized tests)
41. Use Assertions Instead of Print Statements: avoid cluttered test code.
42. Build tests that have deterministic results.
43. Test negative scenarios and borderline cases, in addition to positive scenarios : A negative test case is a test case that tests if a system can handle invalid data
44. Write small and specific tests by using helper functions, parameterized tests, AssertJ's powerful etc .
45. Write self-contained tests by revealing all relevant parameters insert data right in the test and prefer composition over inheritance.
46. Write dump tests by avoiding the reuse of production code and focusing on comparing output values with hard-coded values.
47. Don't Overuse Variables
48. Apply java tuning parameters.
49. Create unit tests that target exceptions
50. Put assertion parameters in the proper order

51. Do not write your own catch blocks that exist only to fail a test
52. Avoid External Dependencies inside test cases at execution time.
53. Unit Tests Should be Separated From Production Code
54. Unit Tests should be included in CI Builds: All unit tests should be run in a CI Build as soon as the code is committed. This leads to early detection of bugs.
55. Test Where the Logic is
56. Continuously Refactor Test Code
57. Always Write Tests for Bugs
58. Make Unit test cases Short
59. Don't Repeat Yourself (DRY Principle in design of test cases)
60. The characteristics of a good unit test:

- **Fast**. Having a unit test that takes more time to run removes most of the advantages. The longer it takes to run, the more developers will put off running them until just before commit time, or start skipping them all together.
- **Isolated**. It can be tempting to have a bunch of common set-up tasks used by multiple unit tests, but where possible, try to limit this. It ties unit tests together in ways that can be harder to isolate in the future.
- **Succinct**. If your unit tests are hard to set up and complicated to write, it's a good sign that the code you are trying to test is either too complex, or badly structured. Either way, it's a good sign that a code refactoring is required.
- **Easy to Understand**: Good naming of unit tests is vital. You should be able to read a test name and understand what the test is testing, what results the test is expecting, and why it's failing. Do not be afraid to make unit test names long – the more information the better. After all, you should only see them when they are failing.
- **Reliable**. Tests shouldn't be bound to a given environment or rely on external factors to work. They should work 100% of the time, or they are not going to fulfill their purpose.

****