|  |  |
| --- | --- |
| <strong>question</strong> <strong>mark</strong> | 3d human with a red <strong>question</strong> <strong>mark</strong> | Damián Navas ...  Unit Test Automation Training content  Adarsh Mehrotra & Rishi Khurana | Abstract  It is envisaged that by end of this course, the readers will be familiar in writing Automated Unit Test (AUT) scripts with jUnit on Java. The course has been prepared to cover Java code base with and without Maven so that teams not using Maven can still learn and start implementing AUT while they graduate to Maven. |

Table of Contents

[1. Basics 2](#_Toc523845368)

[2. Benefit Cost Analysis to be done before writing Unit Test Automation 3](#_Toc523845369)

[3. SetUp with IntelliJ IDE and Java code base with jUnit without Maven 4](#_Toc523845370)

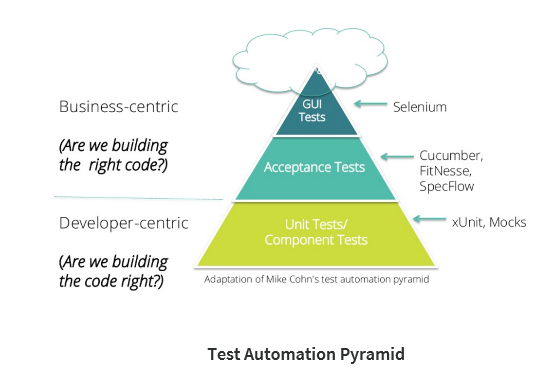
[4. Examples and Practical Illustration 5](#_Toc523845371)

[5. SetUp with IntelliJ IDE and Java code base with jUnit with Maven 16](#_Toc523845372)

[6. Summary 19](#_Toc523845373)

# Basics

* 1. *What is Unit Testing*
  + Goal of unit testing is to isolate each part of a program and show that the individual parts work correctly

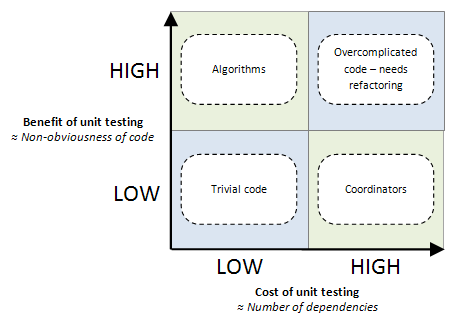


* 1. *Why AUT*

Automation helps in developing a Continuous Delivery solution that is a continuously tested DevOps software delivery pipeline.

* 1. *Salient Points with AUT*
     1. Unit tests are designed for code that has no external dependencies, such as calls to the database or web services
     2. They focus on a specific behavior in a small section of a system under test (SUT).
     3. The key characteristics exhibited by good Unit tests are-
        1. Run fast (they have short setups, run times, and break downs).
        2. Run in isolation (you should be able to reorder them).
        3. Use data that makes them easy to read and to understand.
        4. Use real data (or copies of production data) when they need to.

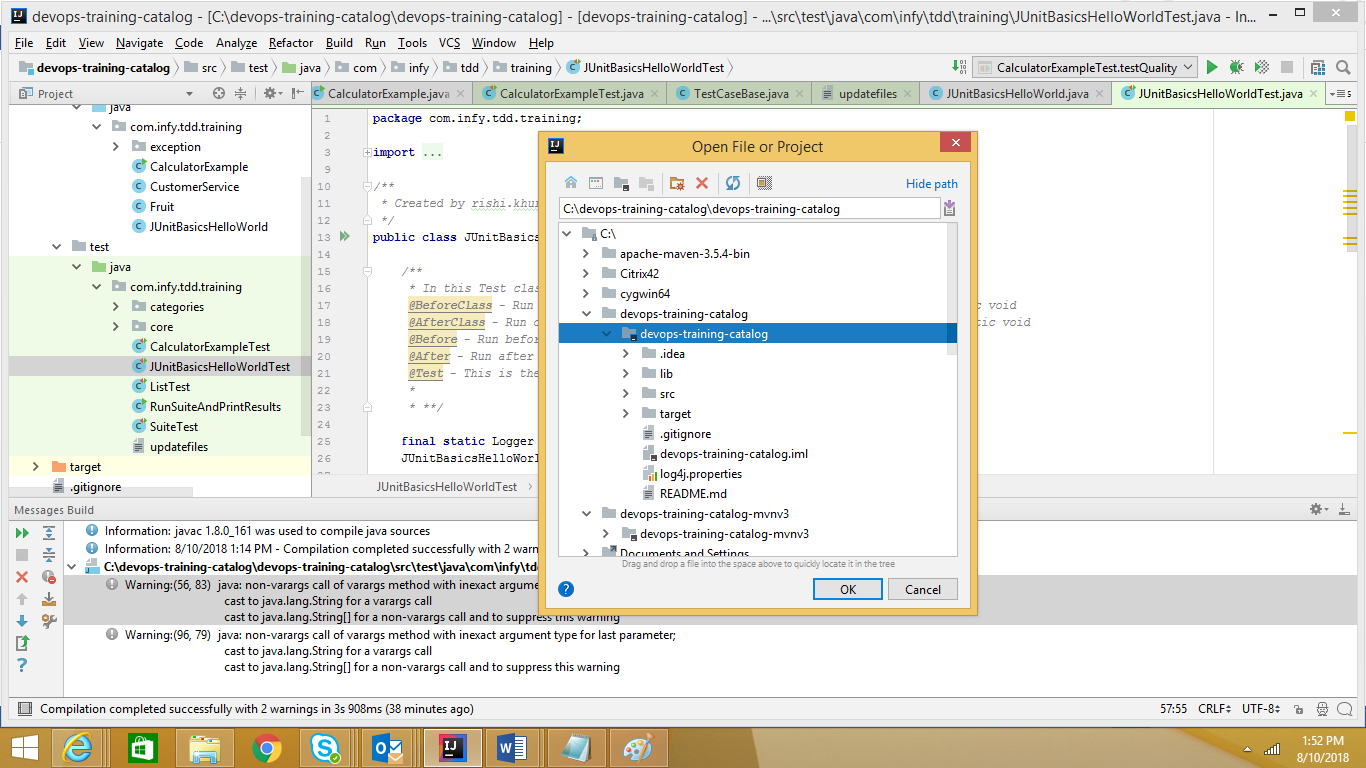
# Benefit Cost Analysis to be done before writing Unit Test Automation



* 1. **Complex code with few dependencies (top left)** - Self-contained algorithms for business rules or for things like sorting or parsing data. This cost-benefit argument goes strongly in favor of unit testing this code, because it’s cheap to do and highly beneficial.
  2. **Trivial code with many dependencies (bottom right)** - This quadrant is called “coordinators”, because these code units tend to glue together and orchestrate interactions between other code units. This cost-benefit argument is in favor of not unit testing this code: it’s expensive to do and yields little practical benefit.
  3. **Complex code with many dependencies (top right)** - This code is very expensive to write with unit tests, but too risky to write without.
  4. **Trivial code with few dependencies (bottom left)** - In cost-benefit terms, it doesn’t matter whether this is unit tested it or not.

# SetUp with IntelliJ IDE and Java code base with jUnit without Maven

* 1. *Pre-requisites* 
     1. Install Intellij IDE (community edition 4.2)
     2. Install Java 1.8
     3. Import the project marked as attachment at the end of the document by clicking File->Open



* + 1. If the JDK is not installed, then options with IntelliJ like ‘Run’, ‘Run With coverage’ will not appear. In that case, please install latest JDK from the [Oracle](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) website.

* + 1. The set-up is complete

# Examples and Practical Illustration

* 1. Basics and Hello world

Test Class - AJUnitBasicsHelloWorldTest.java

Src class - JUnitBasicsHelloWorld.java

* + 1. Create the first Test class called ‘JUnitBasicsHelloWorldTest.java’
    2. Import relevant libraries –



* + 1. Log4j is used for logging whereas asserts are introduced via Hamcrest.
    2. The class extends to a base class that provides logging facilities.



* + 1. The entire details on logging can be found by referring to log4j.properties file.
    2. There are basic annotations that should be understood before writing any Junit Test case. The details are-



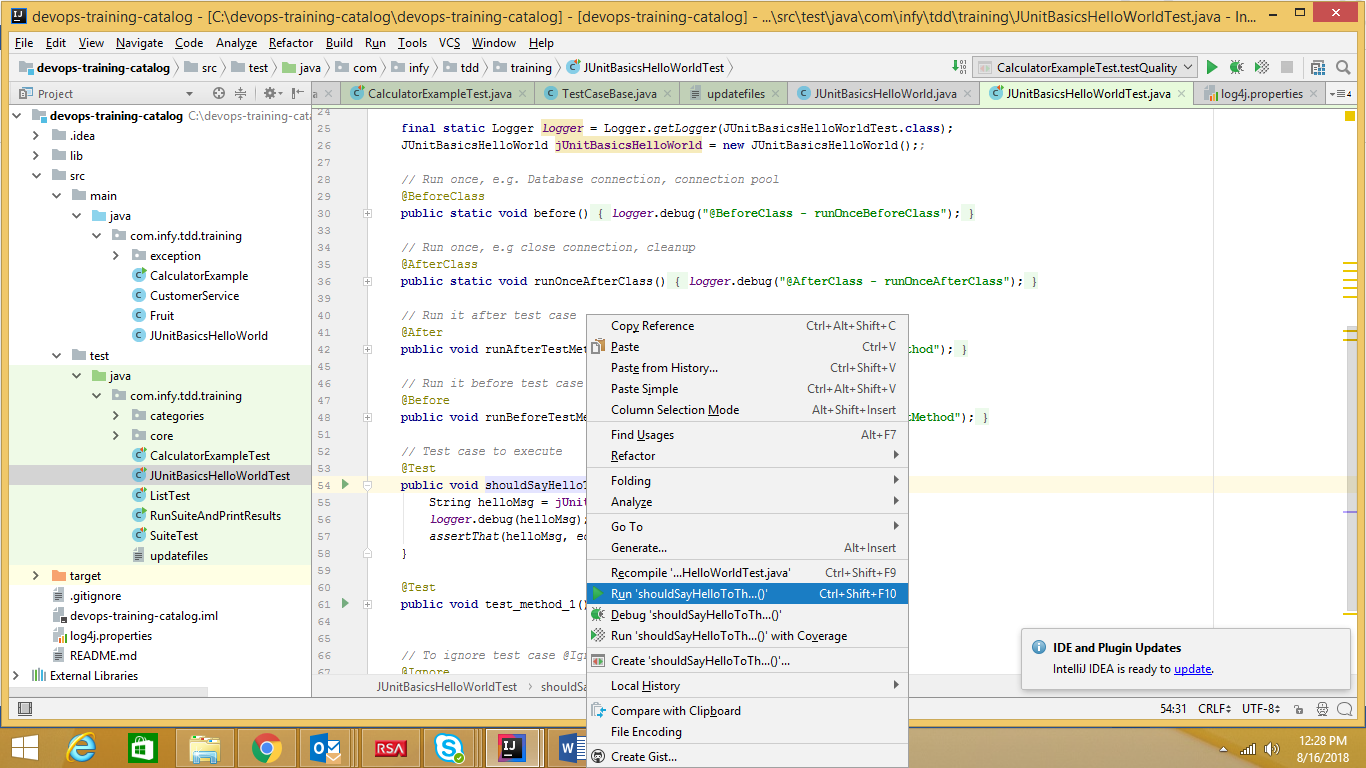
* + 1. The simple method shown below is written to validate the functionality of class ‘JUnitBasicsHelloWorld’. The class returns “**Hello Students!!”** and the expectation of our test case is that it will also return the same.



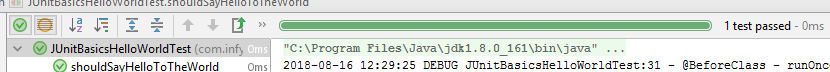
* + 1. Every single test starts with @Test annotation. We invoke the test class and use Assert to verify a statement as fail or pass depending upon the result it returns. ‘Logger.debug’ is used to write back the relevant content to the logs. The location of the logs can be found by opening the log4j.properties file



* + 1. Right-click on the class to say run the test –



* + 1. The results of the first test get displayed.



***WHAT I LEARNT***

*Usage of ‘Import Libraries’ to get started*

*Usage of logger class*

*First test and its results*

* 1. Writing tests and running the Class

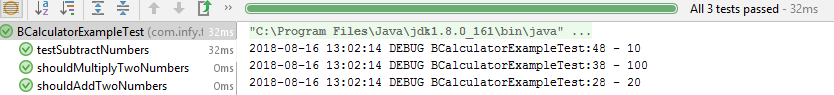
Test Class - BCalculatorExampleTest.java

Src class - CalculatorExample.java

* + 1. Fundamentals of the class remain the same i.e.
       1. The class extends ‘TestCaseBase’ so as to handle logging.
       2. Library references do not change
       3. Logger method present to record the details
    2. There are basic arithmetic operations tested here. Let us take 1 example-



* + 1. As a best practice, we need to test for ‘assertNotNull’. If the value mentioned is ‘21’ instead of 20, the test will show a failed result with a clear view on Actual vs Expected
    2. Run the entire class to see a cumulative test result count by right-clicking on the class and ‘Run’. The results appear like –



***WHAT I LEARNT***

*More Test cases*

*Best practice on Assert*

*Test case failure*

*Getting test results from the entire class*

* 1. Testing Exception

Test Class - CExceptionTests.java

Src class - CalculatorExample.java

* + 1. There are three ways to test exceptions. The first way is Expected Exception-



* + 1. This way is used when you only want test the type of exception
    2. The second way to test exception is using Try-Catch block-



* + 1. The Try-Catch test should have ‘fail()’. The above test passes because that is the expected behavior. With this, you have to write the block around the tested behavior, and write the assertion in the catch block, that may be fine but many find that this style interrupts the reading flow of a test.
    2. The most optimum way to test exception is to create a rule.



* + 1. The rule lets you indicate not only what exception you are expecting, but also the exception message you are expecting.
    2. We can also find out if it has embedded state we want to verify using ‘thrown.expect’ & finding error code.



***WHAT I LEARNT***

*Testing Exceptions*

*Best practice on testing exceptions*

* 1. Ignore a Test

Test Class - DIgnoreTests.java

Src class - NA

* + 1. The @Ignore test annotation is used to ignore particular tests or group of tests in order to skip the build failure.
    2. This is particularly helpful when multiple developers are working on same code base. If there is a flakey test written by ‘x’ developer that can be ignored so that it does not hamper other dev work.
    3. To ignore a test in JUnit you can either comment a method, or delete the @Test annotation; but the test runner will not report such a test.
    4. Alternatively, you can add the @Ignore annotation in front or after @Test. Test runners will report the number of ignored tests, along with the number of tests that ran and the number of tests that failed.



***WHAT I LEARNT***

*@Ignore Tests*

*Why is it important*

* 1. How to test a List

Test Class - EListTest.java

Src class – CalculatorExample.java

* + 1. List consists number of paired similar type of items or an object.
    2. While using List/Collection, the following helper methods come handy-
       1. **not** - matches if the wrapped matcher doesn’t match and vice-versa
       2. **equalTo** - test object equality using the equals method.
       3. **is** - decorator for equalTo to improve readability.
       4. **hasToString** - test Object.toString.
       5. **instanceOf**, **isCompatibleType** - test type.
       6. **notNullValue, nullValue** - test for null.
       7. **sameInstance** - test object identity.
       8. **hasEntry, hasKey, hasValue** - test a map contains an entry, key or value.
       9. **hasItem, hasItems** - test a collection contains elements.
       10. **hasItemInArray**- test an array contains an element



* + 1. While assert, instead of iterating over a List and comparing individual element in List, ” *is()”* method can be used to compare list’s element.
    2. Similarly, an element from List can be find easily using helper method “*hasItems (ELEMENT TO BE SEARCHED)”*.
    3. To verify the element’s existence in a List, helper method *“contains(ELEMENT IN SEQUENCE)”* or *“containsInAnyOrder (EXACT\_ELEMENTS\_IN ANY ORDER)”* can be used.
    4. Size of List can be compared in assert using hasSize (NUMBER OF ELEMENTS).
    5. Even after instantiating any collection, it may have no elements, if not added. this can be easily verified using not (*IsEmptyCollection.empty()*).

***WHAT I LEARNT***

*Testing List and methods that can help*

*Best practices while handling tests on Lists*

* 1. Timeout in Tests

Test Class - FTimeOutExample.java

Src class – CalculatorExample.java

* + 1. Tests that 'runaway' or take too long, can be automatically failed. There are two options for implementing this behavior.
    2. Option 1 - Timeout parameter on @Test Annotation



* + 1. The above example fails because the timeout is specified 1000 ms whereas the test is running infinite.
    2. There is another example on similar lines-



* + 1. Option 2 - Timeout Rule
    2. The Timeout Rule applies the same timeout to all test methods in a class.



* + 1. The timeout specified in the Timeout rule applies to the entire test fixture, including any @Before or @After methods.

***WHAT I LEARNT***

*How to measure performance of tests using@TimeOut*

*Local methods & Global timeOut*

* 1. Run Entire Suite of Tests

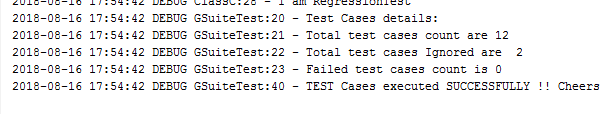
Test Class - GSuiteTest.java & HRunSuiteAndPrintResults.java

Src class – NA

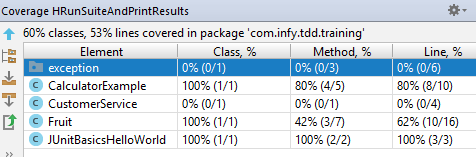
* + 1. In absence of Maven, the entire set of tests can be run as depicted in GsuiteTest class. It requires @RunWith annotation



* + 1. In order to customize reports on total test count, failed tests, passed tests, we prepare another class ‘HRunSuiteAndPrintResults’. All details can be logged.
    2. The entire suite can be run by doing right-click and ‘Run HRunSuiteAndPrintResults’. The results are as follows-



* + 1. We can run the class with coverage that will depict Class, Method & Line coverage.



***WHAT I LEARNT***

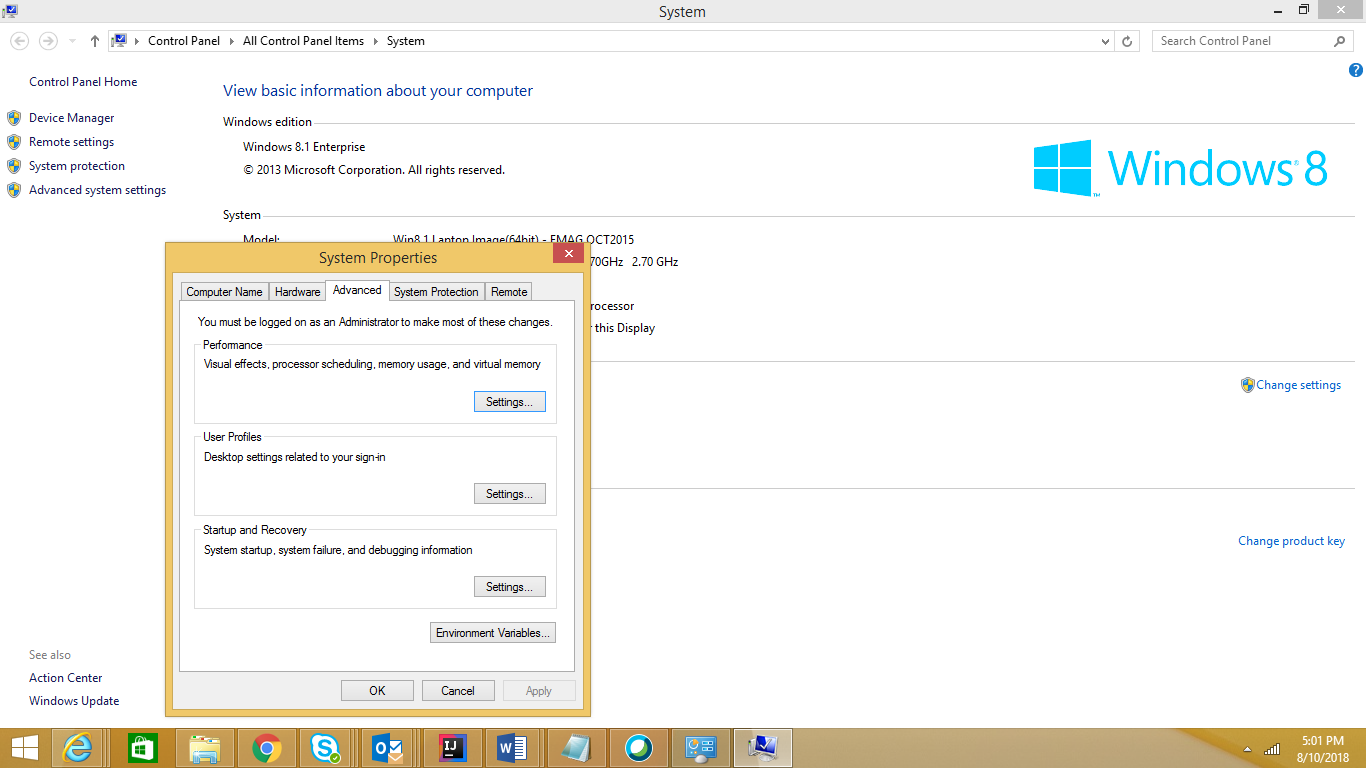
*Running entire set of tests across all classes*

*Customize the test report*

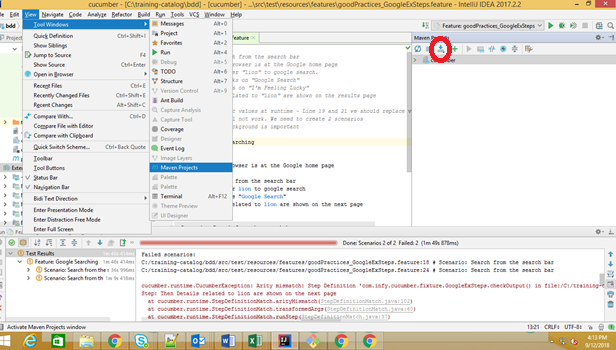
*Class, method and line coverage and its importance*

# SetUp with IntelliJ IDE and Java code base with jUnit with Maven

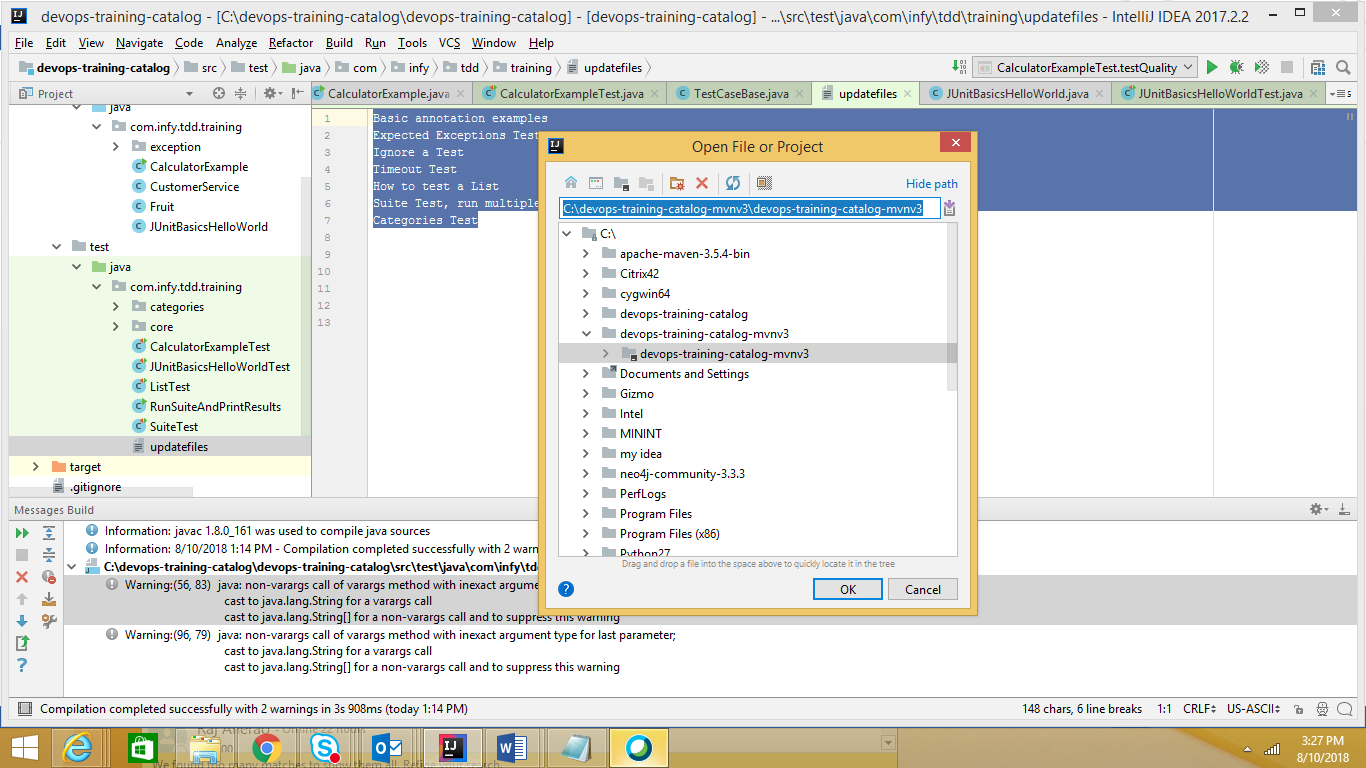
* 1. Installation and pre-requisites
     1. Install Intellij IDE (community edition 4.2)
     2. Install Java 1.8
     3. If the JDK is not installed, then options with IntelliJ like ‘Run’, ‘Run With coverage’ will not appear. In that case, please install latest JDK from the [Oracle](http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html) website.
     4. Install Maven for 2nd example (https://maven.apache.org/download.cgi)
     5. Add the classpath for maven-



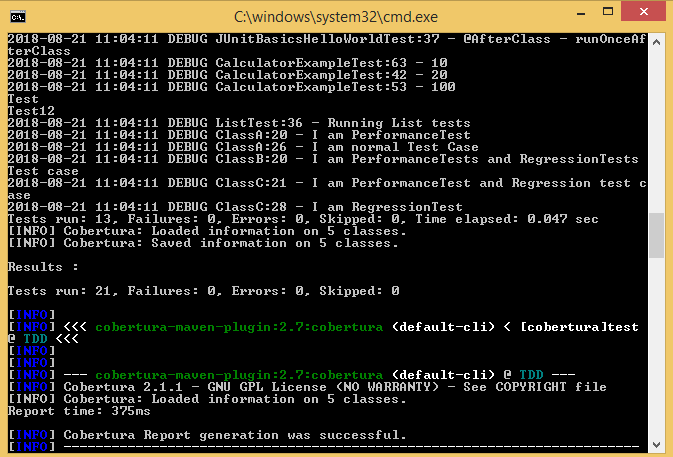
* + 1. Class path should have –
       1. C:\Program Files\Java\jdk1.8.0\_161\bin; C:\apache-maven-3.5.4-bin\apache-maven-3.5.4\bin;
    2. Run **mvn** command from the folder where Maven is installed-
* mvn coberture:cobertura
  + 1. If there are Maven related dependency issues, kindly click to get Maven Projects. Then, click on the icon marked in RED to get latest. The system may need to connect on Internet for first time —



* + 1. We import the project as depicted below-



* 1. Examples
     1. The examples are covered from No.1 to 7 are the same and can be run with Maven. The mechanism to run this is as follows-
        1. Go to Command prompt and in the directory where the solution is. It is placed in C:/Training Catalog/AUT/mvn-tdd
        2. Run mvn:cobertura cobertura



* + - 1. The Cobertura report with all details can be found at the following location-

C:/Training Catalog/AUT/tdd-mvn/target/site/cobertura/index.html

* + - 1. The report can be opened in any browser
      2. Individual classes can be clicked to get their detailed report.



* + - 1. The area marked in PINK in the above report shows non-coverage.
      2. The results of coverage between IntelliJ and Cobertura vary slightly as the internal algorithms to calculate the line coverage across both are different.

***WHAT I LEARNT***

*Running entire set of tests across all classes via Maven-Cobertura*

*Location of test results*

*Interpretation of test results*

# Summary

The above examples will help create a basic infrastructure on writing automated Unit tests and set up a robust unit testing practice. There are some advanced techniques like resource utilization measurements, automated documentation generation, hard assets vs soft asserts that we have skipped to keep the content simple.