Junit Tests

Checking Current Code Coverage

- We use onap Sonar to track code coverage (sonar.onap.org)
- To see the appc coverage, click on the “appc” project on the front page (make sure you choose the most recent version of appc)
- From the project page, you can click on the coverage percentage to see more detail

Junit Plugin Setup

- The EclEmma plugin for Eclipse IDE allows you to run tests and see code coverage from within Eclipse
- If you go to the “Eclipse Marketplace” under the “Help” menu in Eclipse, you can search for EclEmma and install it from there
- Now, when you right click a class in either the Project Explorer or Outline in Eclipse, you will see the option to run “Coverage As…” > “Junit Test”

Naming Test Classes

- Normally, the test class should start with the word “Test”, followed by the class name you are testing
- One test class is created for each class you are trying to test
- For example, if we are testing the “RestartServer.java” class, the test class could be named “TestRestartServer.java”

Where to put test classes

- Test classes should be placed in the src/test/java folder of the same project that contains the code that you are testing
- The test classes should be in the same package as the class you are testing, but in the src/test/java folder
- For example:
  - The “RestartServer.java” class is part of the appc-iaas-adapter-bundle project
  - It can be found in the “src/main/java/org/onap/appc/adapter/iaas/provider/operation/impl” folder
  - Therefore, the test class should be placed in the “src/**test**/java/org/onap/appc/adapter/iaas/provider/operation/impl” folder

Inside a Test Class

- For any test class, the “org.junit.Test” class needs to be imported
- Each method in your test class should test one function of the code
  - At a minimum, you will want one method in your test class for each method of the class that you are testing
  - If you want to test different paths or options in a method, you will want to make a different method in your test class for each of these
  - For example, you might want to test the functionality of a method based on different input values. Each of these tests should have its own method in your test class
- Each test method in your test class needs the annotation “@Test” on the line before the method declaration

Naming Test Methods

- One way to name the methods in your test class is to start with the word “test” and then the name of the method you are testing
  - For example, if we are testing the “executeProviderOperation()” method, the test method could be named “testExecuteProviderOperation()”
- For cases where you will have multiple test methods for each method being tested, one way to name these is to use an underscore after the method name
  - For example, if we are testing the “executeProviderOperation()” method with a null input value to make sure it functions correctly, we could name this test “TestExecuteProviderOperation_NullInput()”

Checking Behavior

- It is not enough to simply run the code that you are testing. You want to make sure that is running correctly
- Junit provides several functions to do this:
  - Import the “org.junit.Assert” class
  - `Assert.fail(“description of what failed”):`
    - This method marks the test as a failure
    - For example, you might call this if an exception is thrown from the code that you are testing
  - `Assert.assertEquals(expected value, actual value):`
    - This method compares the value you expect to get back, with the value you actually get back
    - If they do not match, it marks the test as a failure
  - More examples can be found in the junit docs

Rules for Tests

- Tests need to run fast
  - Each test should run in under a second
  - A test cannot wait for a timeout or other time based event
- Tests should never make a connection to an external service of any kind
  - The idea of unit tests is that they run quickly and internally in any environment
- The PowerMock framework should not be used
  - Tests done using the PowerMock framework will not be counted towards the Sonar coverage results

Mocking Classes

- Some classes cannot be used in tests
  - For example, a class that goes out to an OpenStack server, gets a list of its inventory, and stores it
  - Since we can’t make connections to remote systems in junit tests, we can’t use this class in a test
- We can create a fake version of this class to be used during the test
  - This will allow us to test the functionality of other classes, even though we can’t test this one class

Org.Mockito

- The mockito framework provides an easy way to create fake or “mocked” versions of classes
  - We can define behavior for these mocked classes so that they can still provide correct return values to the rest of the code that we are testing
  - After the test, we can query the mocked class and confirm whether certain methods were called on it, and if certain values were passed to it
- Mockito does have some limitations
  - Return values can not be set for private methods of mocked classes

Mocking a Class

- Let’s say we are trying to create a mock of a class named “OpenStackProvider.java”
- We will need to import “org.mockito.Mockito”
- To create the mock:
  - `OpenStackProvider openStackProvider = Mockito.mock(OpenStackProvider.class);`

Making the Mocked Class Return values

- By default, the mocked class will always return a default value (null for objects, 0 for ints, etc..) when a method is called on it
- We can change this
- Let's say the OpenStackProvider class has a getProviderName() method
- We want to make this return a real value
  - Import the static class (`import static`…) "org.mockito.Mockito.doReturn"
  - `doReturn("Provider 1 Name").when(openStackProvider).getProviderName();`
  - The red name is the instance of OpenStackProvider that we created on the last slide

Verify that a method was called on the mocked class

- Let's say that the OpenStackProvider class has a "`public void startInstance(String instanceId)`" method
- We want to verify that this method of our mocked version of the class was called with the parameter "instance1"
  - Import the static class "org.mockito.Mockito.verify"
  - `verify(openStackProvider).startInstance("instance1");`
- If the method startInstance was not called during the test with the "instance1" value, this will cause the test to be marked as failure