JUnit

- An open source framework to write and run tests. JUnit features include:
  - Assertions for testing expected results
  - Annotations for identifying methods that specify a test
  - Test runners for running tests
- JUnit was written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)
- Available as a stand-alone application and built into Eclipse.
  - The current version of the Eclipse JDT has the JUnit plug-in built in.
Write a Test Class with JUnit

public class Hello{
    public String say(){
        return "Hello";
    }
    public String echo(String s){
        if (s == null){
            throw new Exception();
        }
        return s;
    }
}

public class HelloTest {
    private Hello h;
    @Before
    public void setUp() {
        h = new Hello();
    }
    @After
    public void tearDown() {}
    @Test
    public void testSay() {
        assertEquals("Hello", h.say());
    }
}
Steps to write a test case with JUnit

- Define a Test class.
- Optionally override the methods annotated with @before, @after to create or release object(s) under test.
- Define one or more public testXXX() methods (annotated with @Test). Within each method,
  - Call the method being tested and get the actual result
  - Assert what the correct result should be with one of the provided assert methods
  - These steps can be repeated as many times as necessary
- Test methods are independent of each other.
## JUnit Annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@Test public void method()</code></td>
<td>The annotation <code>@Test</code> identifies that a method is a test method.</td>
</tr>
<tr>
<td><code>@Before public void method()</code></td>
<td>Executes the method before EACH test. This method sets up the test data.</td>
</tr>
<tr>
<td><code>@After public void method()</code></td>
<td>Executes the method after EACH test. This method can cleanup the test data.</td>
</tr>
<tr>
<td><code>@BeforeClass public static void method()</code></td>
<td>Executes the method once, before the start of ALL tests.</td>
</tr>
<tr>
<td><code>@AfterClass public static void method()</code></td>
<td>Executes the method once, after ALL tests have have been finished.</td>
</tr>
<tr>
<td><code>@Ignore</code></td>
<td>Ignores the test method.</td>
</tr>
<tr>
<td><code>@Test (expected = Exception.class)</code></td>
<td>Fails, if the method does not throw the named exception.</td>
</tr>
<tr>
<td><code>@Test(timeout=100)</code></td>
<td>Fails, if the method takes too long.</td>
</tr>
</tbody>
</table>
Test Fixtures

- A test fixture is the data (both objects and primitives) that are needed to run tests
  - JUnit supports sharing the setup code
- public void setUp() (annotated with @Before)
  - Sets up the test data (fixture).
  - Called before EVERY test case method.
- public void tearDown() (annotated with @After)
  - Tears down the test fixture.
  - Called after EVERY test case method.
- public static void setUpBeforeClass() (@BeforeClass)
- public static void tearDownAfterClass() (@AfterClass)
JUnit Assert Methods

- JUnit assert is a collection of static methods defined for checking actual values against expected values.

- We only used assertEquals in previous example, but there are additional assert methods.

- JUnit static import
  - To use these assert methods without mentioning the class name, the static import statement is usually added at the beginning of each test class.
  - import static org.junit.Assert.*;
## JUnit Assert Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fail([String])</code></td>
<td>Causes the test to fail.</td>
</tr>
<tr>
<td><code>assertsEquals([String message], expected, actual)</code></td>
<td>Test if the values are the same.</td>
</tr>
<tr>
<td><code>assertNull([message], object)</code></td>
<td>Checks if the object is null.</td>
</tr>
<tr>
<td><code>assertNotNull([message], object)</code></td>
<td>Check if the object is not null.</td>
</tr>
<tr>
<td><code>assertSame([String], expected, actual)</code></td>
<td>Check if both variables refer to the same object.</td>
</tr>
<tr>
<td><code>assertNotSame([String], expected, actual)</code></td>
<td>Check that both variables refer not to the same object.</td>
</tr>
<tr>
<td><code>assertTrue([message], boolean condition)</code></td>
<td>Check if the boolean condition is true.</td>
</tr>
<tr>
<td><code>assertFalse([message], boolean condition)</code></td>
<td>Check if the boolean condition is false.</td>
</tr>
</tbody>
</table>
Testing Exceptions (two different ways)

```java
public void testEcho()
{
    try {
        h.echo();
        fail("should raise an exception");
    } catch (Exception e) {
        //expected
    }
}
```

```java
@Test(expected = Exception.class)
public void testEcho()
{
    h.echo();
}
```
Test Suite

- Test Suite is a composite of Tests, or a combination of test classes.
  - A suite usually contains a set of test classes, but a suite can contain another suite and so on.
- Most IDEs create suites for you. If you have to create suites yourself:
  ```java
  import org.junit.runner.RunWith;
  import org.junit.runners.Suite;
  import org.junit.runners.Suite.SuiteClasses;
  @RunWith(Suite.class)
  @SuiteClasses({ MyClassTest.class, MySecondClassTest.class })
  public class AllTests {
  }
  ```
Test Runner

- Running a TestSuite will automatically run all of its subordinate Test class instances and TestSuite instances.
- Running a Test class will automatically invoke all of its annotated @Test methods. In other words, one Test class could result in multiple tests.
  - All test methods can be executed in an arbitrary order.
- For each @Test method
  - Run @Before method if defined
  - Run @Test method steps
  - Run @After method if defined
Integration Tests With JUnit

- JUnit is primarily used for unit test.
- There is no clear line between unit test and integration test.
  - A specific method may involve the integration logic.
- Two ways to do integration tests with JUnit
  - Do unit test, as previously discussed, on a method that includes the logic of calling other objects.
  - Define a testXXX() method that tests a specific transaction, instead of a method. In this testXXX(), call related objects to check if the transaction is implemented correctly.
Note that

- The version we are using is JUnit 4
  - The previous version is JUnit 3, which did not support annotations (@Before, @After, @Test, …).

- Additional information
  - JUnit Pocket Guide (Search “JUnit” at http://proquest.safaribooksonline.com)
  - JUnit Tutorial
    http://www.vogella.com/articles/JUnit/article.html