Testing Terminology

- Unit Test
  - Testing methods of a single class
- Integration Test
  - Testing how well two or more classes work together
- Test Suite
  - Collection of test cases
- Test Fixture
  - Code that sets up data needed to run tests
- Test Runner
  - Software that runs the tests and reports results

Ideas of Unit Testing

- Tests are automated (self checked)
- Tests run in isolation of one another
- Tests do not depend on or connect to external resources (File I/O, Networking, DB)
- Ordering of tests does not matter
- Each test starts with a clean data structure (not use one from a previous test which could have changed it)

JUnit Strategy

- For each test
  - Arrange
    - Create the data/objects needed to perform the test
  - Act
    - Perform the test
  - Assert
    - Validate the results

JUnit Tests

➢For each test

- Imports
  - **import** org.junit.Test;
  - **import static** org.junit.Assert.*;
- Place @Test before each test method

```
public class TestString {
  @Test
  public void testLength() {
    String tstString;
    int len;

    /*  arrange  */
    tstString = new String("test");

    /*  act  */
    len = tstString.length();

    /*  assert  */
    assertEquals(len,4);
  }
}
```

Assert Methods

- After getting a result, use an assert() method to verify expectations have been met
  - May need multiple assert() calls to check

- Basic Asserts
  - AssertionError thrown if test fails
  - Optional message is included in Error on failure

```
static void assertTrue(boolean test)
static void assertTrue(String message, boolean test)

static void assertFalse(boolean test)
static void assertFalse(String message, boolean test)
```

Assert Equals

- Primitive parameters compared with ==
- Object parameters compared by calling the equals() method

assertEquals(expectedValue, ***actualValue***)

assertEquals(String message, expectedValue, ***actualValue***)

- For comparing doubles

assertEquals(expected, actual, delta)

```
final double DELTA = 1e-15;

@Test
public void testDelta(){
   double expectedVal = 20.56;
   double computedVal = Calculator.add(10.54, 10.02);
   assertEquals(computedVal, expectedVal, DELTA);
}
```

Assert Same

- Verify two objects are the same object (address comparison)

assertSame(Object expected, Object actual)
assertSame(String message, Object expected, Object actual)

assertNotSame(Object expected, Object actual)
assertNotSame(String message, Object expected, Object actual)

Assert Null

- Verify an object is null

assertNull(Object actualObj)
assertNull(String message, Object actualObj)

assertNotNull(Object actualObj)
assertNotNull(String message, Object actualObj)

Assert Failure

- Used to indicate test has failed
  - For example, reach a point in the code that should never have been reached

fail()

fail(String message)

```
public class TestString {
  @Test
  public void testArrayBoundsException() {
    String tstString;

    /*  arrange  */
    tstString = new String("test");

    /*  act  */
    try {
        char badChar = tstString.charAt(10);
        /* why didn't we get an exception? */
        fail("No exception thrown for bad index");
    } catch (StringIndexOutOfBoundsException ex) {
      /* This is what we expect to happen */
    }
  }
}
```

Expecting Exceptions

- Add expected attribute to @Test to indicate an expected exception
  - Test passes if exception is thrown, fails otherwise
  - Note the example of the previous slide is more precise in case multiple statements could throw the same exception

```
@Test(expected = StringIndexOutOfBoundsException.class)
public void testArrayBoundsException() {
    String tstString;
    int len;

    /*  arrange  */
    tstString = new String("test");

    /*  act  */
    char badChar = tstString.charAt(10);
}
```

Expecting Exceptions

- Caution: what would happen if you wanted to test that a division operation gives an ArithmeticException on a divide by zero, but multiple statements in the test could throw the exception?

```
@Test(expected = ArithmeticException.class)
public void testExFromDivByZero() {
    Calculator calc = new Calculator();
    int firstDiv; divRes;

    /* What if this statement throws exception (because of a bug)? */
    firstDiv = calc.div(10,5);
    divRes = calc.div(firstDiv, 0);
}
```

Timing Out

- Use timeout attribute to cause failure if a test takes too long
  - What if bug introduces an infinite loop?
  - What if code modification makes testcase run 3 times longer?
  - Time limit is in milliseconds

```
@Test(timeout = 1000)
public void testEquals() {
    String tstString1, tstString2;
    int len;

    /*  arrange  */
    tstString1 = new String("test");
    tstString2 = new String("test");

    /*  act and assert  */
    assertEquals(tstString1, tstString2);
}
```

Ignoring Tests

- Use @Ignore to turn off the testing for a function
  - Can be used to add tests before code is actually developed or working

```
@Ignore("Expected to work in release 5.6")
@Test
public void testEquals() {
    String tstString1, tstString2;
    int len;

    /*  arrange  */
    tstString1 = new String("test");
    tstString2 = new String("test");

    /*  act and assert  */
    assertEquals(tstString1, tstString2);
}
```

Test Fixtures

- If tests require some common initial setup, put the code in a function annotated with @Before
  - Code is executed before every test so each test starts with uncorrupted data
  - Can have more than one method with the annotation

```
public class TestString {
  private String tstString;

  @Before
  public void setUp() {
    String testStr = new String("test");
  }

  ...

}
```

- Use methods annotated with @After to cleanup after every test, if necessary

Test Suites

- Use @SuiteClasses to define the test classes in the test suit
- When you run the class, all tests in the suite are run

```
@SuiteClasses(value={StringTests.class,
         CalculatorTests.class,
         HashTableTests.class})
public class NightlyTests{ }
```

@RunWith

- Use @RunWith to specify a different class (test runner) to run the tests
  - Replaces the test runner built in to JUnit
  - For Spring testing use SpringJUnit4ClassRunner.class

```
@RunWith(SpringJUnit4ClassRunner.class)
public class StudentDaoTests {

  ...

}
```

Spring Annotations for Tests

- Use @ContextConfiguration to load an application context

```
@ContextConfiguration("classpath:studentdao-context.xml")
@RunWith(SpringJUnit4ClassRunner.class)
public class StudentDaoTests {

  ...

}
```

Can also use a list of files to create the application context:

```
@ContextConfiguration(locations = {
     "app-context.xml", "utils-context.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class StudentDaoTests {

  ...

}
```

Spring Annotations for Tests

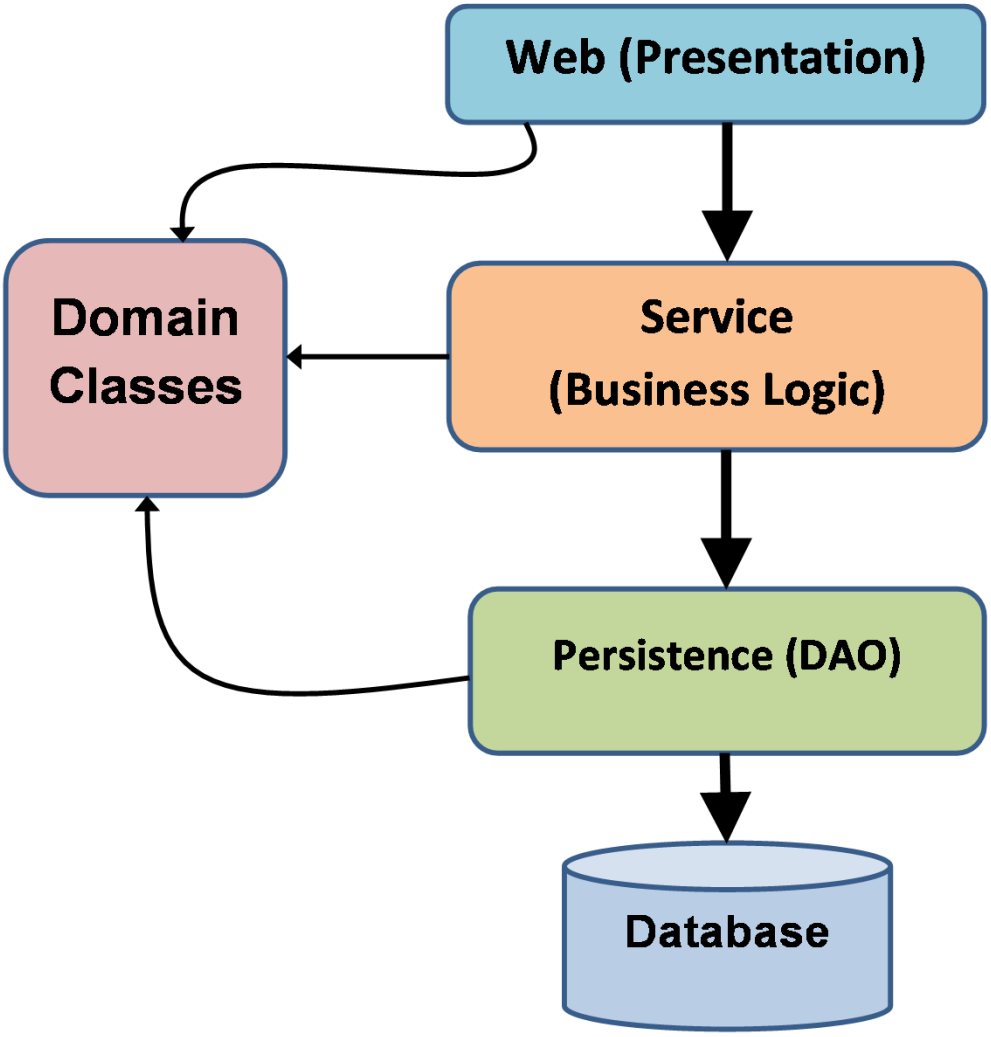
- @DirtiesContext
  - When used on a method, the application context will be reloaded when the method ends

- @TransactionConfiguration
  - Applied at class level to configure transactions (default is to rollback)

A More Realistic Application

- Example: Application for a University
  - Enroll new students
  - Lookup student information
  - Retrieve a student's course history
  - Add courses
  - Add course offerings (specific instances of courses)
  - Register students for a course
  - Lookup course information
  - Find course offerings in a department for a specific semester

Layers of a Web Application



Domain Classes

- Domain classes are the classes needed to represent objects in the problem domain

- Example Domain Classes:
  - Student
    - Name
    - Address
    - Login Account Name
    - GPA
  - Course
    - Name
    - Title
    - Department
  - CourseOffering
    - Course
    - Semester (e.g., Spring 2014)
    - Maximum Enrollment
    - Actual Enrollment
  - CourseEntry
    - Student
    - CourseOffering
    - Grade

The Service Layer

- The Service layer implements the business logic of the application
- Example Service Class: StudentService
  - Enroll new students
  - Lookup student information
  - Retrieve a student's course history

```
public interface StudentService {
    public void addNewStudent(Student stud);
    public void deleteStudent(Student stud);
    public Student findStudentByAcctName(String acctName);
    public void insertStudentCourseItem(StudentCourseItem courseItem);
    public void deleteStudentCourseItem(Student stud, CourseOffering offering);
    public boolean isRegisteredForCourse(Student stud, CourseOffering offering);
    public List<StudentCourseItem> findCourseHistory(long studId);
    public List<StudentCourseItem> findCourseHistoryForSem(long studId,
                                                           String sem, int year);
    public void updateProfile(Student stud);
}
```

The DAO Layer

- The DAO layer consists of Data Access Objects

- All Database operations should be put in the DAO object
  - Keep the Service layer completely independent of the database technology
  - Generally one Domain class should correspond to one DAO class
  - The DAO class will be responsible for persisting the Domain object

```
public interface StudentDAO {
   public Student findStudentById(long id);
   public Student findStudentByAcctName(String acctName);
   public long insertStudent(Student stud);
   public void deleteStudent(Student stud);
   public void updatePhone(Student stud);
   public void updateProfile(Student stud);
}
```

Mocking

- An Object under test may have dependencies on other complex objects

- Mocking
  - Creating objects that simulate the behavior of real objects

- Isolate the test object by replacing other objects with Mocks
  - Mock objects simulate the behavior of other objects
  - Mocked objects can be simpler and more efficient than actual object
  - Can test an object with Mocks even if the real objects are still in development

Mocking Example

- Incorporating a database in testing can be complicated and slow
- If we are only interested in the data (not where it comes from), we can replace the database with a Mock object