[Print](javascript:window.print())

**Course Transcript**

Working with JUnit Tests

**Grouping Tests**

[1. Course Introduction](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t2)

[2. Creating Test Suites](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t6)

[3. Categories](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t10)

**Finetuning  Tests**

[1. Managing Failure with @Ignore](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t15)

[2. Testing Time](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t19)

[3. Customizing Test Runs](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t23)

**Managing Test Case Data**

[1. Parameters in Tests](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t28)

[2. Managing Data across Multiple Tests](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t32)

[3. Test Execution Ordering](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t36)

**Testing Rules**

[1. Enhancing Tests with Rules](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t41)

[2. Rules for Managing Tests](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t45)

**Theories and Mock Objects**

[1. Theories](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t50)

[2. JUnit and Mock Objects](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t54)

**Maven and JUnit**

[1. Maven and JUnit](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t59)

**Legacy Testing**

[1. Legacy JUnit Tests](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t64)

[2. Legacy JUnit Test Suites](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t68)

**Practice: Modifying Java Code in Test Scenarios**

[1. Exercise: Modify Java Code in Test Scenarios](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#t73)

Course Introduction

Learning Objective

*After completing this topic, you should be able to*

* *start the course*

**1. Introduction to the course**

JUnit provides many features that take us beyond basic testing, allowing for advanced ways to organize test data, execute rules within our tests, and fine-tune our test execution. I'm Tony Lowe, a JAVA architect with over 15 years' experience building and teaching IT solutions. We may also have a need to look back at Legacy test cases and bring them forward into our most recent version of JUnit. This course will round out our knowledge of the capabilities and features of JUnit to maximize how we use the framework.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Creating Test Suites

Learning Objective

*After completing this topic, you should be able to*

* *create JUnit Test suites*

**1. How to create test suites**

Individual JUnit tests focus around a single concept, typically tests to assure a component. On a project, you'll build many components with many, many tests overall. As you develop, you typically execute one test at a time. As a project, you may execute all of these tests at specific times, but it's helpful to manage tests in small groups. This is where test suites are useful. So, within my project here I have a lot of tests, and I can go through and I can run them individually. It's not too bad and run any one of these, I get some results back and keep going and keep going down the list and run them one at a time. Now, this isn't bad if I do it once in a while. But if I have to do this every time I change my code, which is what I want to do for regression testing, it would take a long time. But at the same time, if I have 500 test classes inside of there, to run all of those would take a really long time. And I don't need to do them all for every single little test. And so what I can do is I can create a JUnit test suite. And a JUnit test suite is a very simple class these days, because I have a single class with two annotations and I have a suite defined.   
*In the Eclipse development environment, five files are open on separate tabbed pages – TestParameters.java, TestTarget.java, TestRealEstateDAO.java, and TestSlowBehavior.java.  
  
In the TestParameters.java file, the following code is visible:  
  
@RunWith(Parameterized.class)  
public class TestParameters  
{  
   private int               player1;  
   private int               player2;  
   private int               outcome;  
  
   private RockPaperScissors game;  
  
   public TestParameters(int player1, int player2, int outcome)  
   {  
       this.player1 = player1;  
       this.player2 = player2;  
       this.outcome = outcome;  
   }  
  
   @Parameters(name="Rock, Paper, Scissors winner {0} versus {1} => {2}")  
   public static Collection<Integer[]> addedNumbers()  
   {  
       return Arrays.asList(new Integer[][] {   
               {RockPaperScissors.ROCK, RockPaperScissors.ROCK, 0},  
               {RockPaperScissors.ROCK, RockPaperScissors.PAPER, 2},  
               {RockPaperScissors.ROCK, RockPaperScissors.SCISSORS, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.ROCK, 1},  
  
The presenter switches to the TestTarget.java file. In the file, the following code is visible:  
  
public class TestTarget  
{  
@Test  
public void testAdd()  
{  
MyTarget target = new MyTarget();  
  
int answer = target.basicAdd(2, 2);  
TestCase.assertEquals("Two plus two is four right?", 4, answer);  
}  
  
@Test  
public void badTestCase()  
{  
MyTarget target = new MyTarget();  
  
int answer = target.basicAdd(2, 2);  
TestCase.assertEquals("Adding is fun", 6, answer);  
}  
  
@Test(expected=ArithmeticException.class)  
public void testDivideByZero()  
{  
MyTarget target = new MyTarget();  
  
Next the presenter switches to the TestRealEstateDAO.java file. In the file, the following code is visible:  
  
package myapp.re;  
  
import static org.hamcrest.CoreMatchers.hasItem;  
  
  
public class TestRealEstateDAO  
{  
   private static ConfigurationFactory config;  
   private RealEstateDAO dao;  
   private RealEstate sample;  
     
   @BeforeClass  
   public static void setupHibernate()  
   {  
       config = new ConfigurationFactory();  
   }  
  
   @Before  
   public void setupDAO()  
   {  
       dao = new RealEstateDAO(config.getSessionFactory());  
   }  
  
   @Before  
   public void setupSample()  
  
The presenter switches to the TestSuiteExample.java file, which contains the following code:  
  
package myapp.junit.suite;  
  
import myapp.junit.TestTarget;  
  
@RunWith(Suite.class)  
@SuiteClasses({TestRealEstateDAO.class, TestParameters.class, TestTarget.class})  
public class TestSuiteExample  
{  
  
}*   
  
Now, the two annotations is RunWith and then SuiteClasses and they are both critically important. SuiteClasses obviously here list a bunch of classes that I want to test. In this case, I have the four that were open, and there might only be a fraction of everything I have in my solution. It's focused around some regression, some behavior, something I want to list up for the system. And the RunWith tells it I want to run this as a test suite, and this guy is critical. If I take the RunWith out and I try and run this guy, it actually won't let me run as JUnit. It asks me, okay well what do you want to run? It's actually asking me some other main program I have inside here. I can say **Run As** and there is nothing applicable inside here, there is nothing you can run it as, there is no main program, there is nothing here. So, I need the run as. And then when I have my SuiteClasses here and I go and run this guy, you can see of these four test classes it'll run all of them. And it's going to run them in whatever order it chooses to, and some of them will pass and some of them fail. And it's going to show us the collection of everything that's going on inside here. So  -   
*In the TestSuiteExample.java file, the presenter highlights the @RunWith and @SuiteClasses annotations in the following segment of code:   
  
@RunWith(Suite.class)  
@SuiteClasses({TestRealEstateDAO.class, TestParameters.class, TestTarget.class})  
  
Next he highlights the code @RunWith(Suite.class) and deletes it. The code in the TestSuiteExample.java file is now  
  
package myapp.junit.suite;  
  
import myapp.junit.TestTarget;  
  
@SuiteClasses({TestRealEstateDAO.class, TestParameters.class, TestTarget.class})  
public class TestSuiteExample  
{  
  
}  
  
The presenter clicks the Run button on the toolbar and the Select Configuration dialog box lists two launch configuration options – ParallelRunner and MyTestRunner.  
  
The presenter clicks the Cancel button to close the Select Configuration dialog box.  
  
To further demonstrate the importance of the missing code, the presenter clicks the Run drop-down button on the toolbar and hovers over the Run As drop-down list option. A submenu appears with the grayed out option (none applicable).  
  
The presenter pastes the missing code back into the TestSuiteExample.java file. The full code in the file is now  
  
package myapp.junit.suite;  
  
import myapp.junit.TestTarget;  
  
@RunWith(Suite.class)  
@SuiteClasses({TestRealEstateDAO.class, TestParameters.class, TestSlowBehavior.class, TestTarget.class})  
public class TestSuiteExample  
{  
  
}  
  
The presenter clicks the Run button.*   
  
here's one example, it takes 4.4 seconds to go off and run this guy for all the tests. So, this guy only took less than a half second to run and this guy took 7/10th of the second run, this guy took about 4 seconds, it was a really slow one in there, and then this guy was 2/10th of the second. So, I might say, you know if I look at this, I can do most of the testing in 4/10th of the second. So, let me go and take this guy out of here, I don't need this guy. I can drop this down and then all of a sudden my test runs. And in 4/10th of the second, just under half a second here, I have my test results. So, this is the cool part of a suite, I can customize it, I can focus either based off of a functionality or need, or whatever I have to choose inside of there to get what I wanted. Now, the limitation is when I setup a test suite like this, I have to run every single test method in each one of these classes. I can't say, "Oh, this badTestCase, don't run this guy". There is no option for that. If you want to do that, you want to investigate categories in a little bit more detail. But generally speaking, these tests suites are really great for being able to define different regression, testing suites, or maybe just a focused area for your team, or your group, or what's being worked on. And then being able to have quick tests as well as the comprehensive tests as you move forward.   
*In the JUnit view, the top-level test node is myapp.junit.suite.TestSuiteExample. The test progress details show that 39/39 tests ran, 19 were skipped, one returned an error, and three failed. The top-level test node is expanded and the following tests and associated runtimes are listed below it – myapp.re.TextRealEstateDAO (0.452 s), myapp.junit.parameters.TestParameters (0.007 s), myapp.junit.timeout.TestSlowBehavior (4.002 s), and myapp.junit.TestTarget (0.002 s). The following methods and associated runtimes are listed below the myapp.re.TestRealEstateDAO test child node – testLoadAndDelete (0.235 s), testAdd (0.059), testLoadAll (0.093 s), randomNotADBTest (0.003 s), and testUpdate (0.061 s). The following methods and associated runtimes are listed below the myapp.junit.timeout.TestSlowBehavior test child node – testVerySlow (2.002 s), testSlow (1.000 s), and notJustTime (1.000 s).  
  
The presenter selects the top-level test node, myapp.junit.suite.TestSuiteExample, and refers to its overall runtime of 4.464 seconds.  
  
Beneath the top-level test node, he refers to the child test node myapp.re.TestRealEstateDAO, which took 0.452 seconds to run.  
  
The presenter refers to the child test node myapp.junit.parameters.TestParameters, which took 0.007 seconds to run.  
  
The presenter refers to the child test node myapp.junit.timeout.TestSlowBehavior, which took 4.002 seconds to run.  
  
He refers to the child test node junit.timeout.TestTarget, which took only 0.002 seconds to run. Below it, the following methods and associated runtimes are listed – testDivideByZero (0.000 s), testAdd (0.001 s), badTestCase (0.000 s), and sillyTest (0.000 s). The badTestCase method failed, the sillyTest was skipped, and the testDivideByZero and testAdd methods ran successfully.  
  
The presenter refers again to the overall runtime of the top-level test node, myapp.junit.suite.TestSuiteExample.  
  
To reduce the runtime of the test suite, the presenter removes the TestSlowBehavior class from the TestSuiteExample.java file. The full code in the file is now  
  
package myapp.junit.suite;  
  
import myapp.junit.TestTarget;  
  
@RunWith(Suite.class)  
@SuiteClasses({TestRealEstateDAO.class, TestParameters.class, TestTarget.class})  
public class TestSuiteExample  
{  
  
}  
  
The presenter clicks the Run button.  
  
In the JUnit view, the presenter highlights the top-level test suite, myapp.junit.suite.TestSuiteExample, which now ran in a reduced runtime of 0.436 seconds.   
  
The presenter refers to the following segment of code in the TestSuiteExample.java file:  
  
@RunWith(Suite.class)  
@SuiteClasses({TestRealEstateDAO.class, TestParameters.class, TestTarget.class})  
  
Finally, in the JUnit view, the presenter selects the failed badTestCase method under the myapp.junit.TestTarget child test node.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Categories

Learning Objective

*After completing this topic, you should be able to*

* *use categories in JUnit tests*

**1. Using categories in JUnit tests**

While test suites are valuable for providing a grouping of tests, categories allow that group to be further customized. The way we design tests does not mean we must limit how we execute tests. Categories allow us to mark specific test methods and execute them how we see fit. It's very powerful, so let's take a look. So starting with this real estate test, this is a test that does interactions with the database, and there’s many different types of interactions with the database. And so we can categorize each test method, based off the type of test it's doing. So this Category annotation here allows us to say this is a CRUDTest…another one that's a CRUDTest…another one that's a CRUDTest…and then down here we can see we have a SearchTest. So as opposed to create, read, update, and delete – that's what CRUD stands for – this is a test that does searching on the database instead. So within that, we can define these as either being interfaces or we can define them as being inherited inside of there. So if we look at this, a database test is just an interface, it's just a marker; there’s no required methods, there’s no required implementation. It's just a marker that goes inside of there. SearchTest – again, just a marker. But I can actually have these tests extend out the DatabaseTest. And so the different types of database tests out there are search tests, or CRUD tests, or things like that. So -   
*In the Eclipse development environment, eight files are open on separate tabbed pages – TestRealEstateDAO.java, OrderedTestRealEstateDAO.java, DBTest.java, SearchTest.java, CRUDTests.java, SearchTests.java, CRUDTest.java, and DBTests.java.  
  
The TestRealEstateDAO.java file contains the following code:  
  
package myapp.re;  
  
import static org.hamcrest.CoreMatchers.hasItem;  
  
public class TestRealEstateDAO  
{  
   private static ConfigurationFactory config;  
   private RealEstateDAO dao;  
   private RealEstate sample;  
     
   @BeforeClass  
   public static void setupHibernate()  
   {  
       config = new ConfigurationFactory();  
   }  
  
   @Before  
   public void setupDAO()  
   {  
       dao = new RealEstateDAO(config.getSessionFactory());  
   }  
  
   @Before  
   public void setupSample()  
   {  
       sample = createRealEstate();  
   }  
     
   @After  
   public void sayHi()  
   {  
       System.out.println("Hello");  
   }  
     
   @After  
   public void wipeTable()  
   {  
       Session session = config.getSessionFactory().getCurrentSession();  
       session.beginTransaction();  
       session.createSQLQuery("delete from RealEstate").executeUpdate();  
       session.getTransaction().commit();  
   }  
     
   @AfterClass  
   public static void closeDatabase()  
   {  
       config.getSessionFactory().close();  
   }  
     
   @Test  
   @Category(CRUDTest.class)  
   public void testAdd()  
   {  
       dao.add(sample);  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the one we just added", all, hasItem(sample));  
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testUpdate()  
   {  
       dao.add(sample);  
         
       double updatedValue = 200000;  
       sample.setValue(updatedValue);  
         
       dao.update(sample);  
         
       RealEstate updated = dao.load(sample.getId());  
       assertEquals("New value should be updated", updatedValue, updated.getValue(), 0.001);  
                 
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testLoadAndDelete()  
   {  
       dao.add(sample);  
       long saveId = sample.getId();  
         
       RealEstate added = dao.load(saveId);  
       assertNotNull("Should have loaded the one we just added", added);  
         
       dao.delete(sample);  
         
       RealEstate deleted = dao.load(saveId);  
       assertNull("It should not have been loaded", deleted);  
   }  
  
   @Test  
   @Category(SearchTest.class)  
   public void testLoadAll()  
   {  
       RealEstate r1 = createRealEstate();  
       RealEstate r2 = createRealEstate();  
       RealEstate r3 = createRealEstate();  
       dao.add(r1);  
       dao.add(r2);  
       dao.add(r3);  
  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the ones we just added", all, hasItems(r1, r2, r3));  
       assertEquals("We expect 3 items", 3, all.size());  
   }  
     
     
   @Test  
   public void randomNotADBTest()  
   {  
       fail("Not marked with a category, don't run me");  
   }  
     
   private RealEstate createRealEstate()  
   {  
       sample = new RealEstate();  
       sample.setValue(100000);  
       Address a = new Address();  
       a.setStreet("123 Main");  
       a.setCity("My Town");  
       a.setState("My State");  
       a.setZip("12345");  
       sample.setAddress(a);  
       return sample;  
   }  
}  
  
The presenter highlights the category annotation and CRUDTest class in the following segment of code:  
  
@Test  
@Category(CRUDTest.class)  
public void testAdd()  
  
The presenter highlights the CRUDTest class in the following segment of code:  
  
@Test  
@Category(CRUDTest.class)  
public void testUpdate()  
  
He highlights the CRUDTest class in the following segment of code:  
  
@Test  
@Category(CRUDTest.class)  
public void testLoadAndDelete()  
  
He highlights the SearchTest class in the following segment of code:  
  
@Test  
@Category(SearchTest.class)  
public void testLoadAll()  
  
The presenter clicks the DBTest.java tab in the editor. The DBTest.java file contains the following code:  
  
package myapp.junit.categories;  
  
public interface DBTest  
{  
  
}  
  
Next the presenter clicks the SearchTest.java tab. The SearchTest.java file contains the following code:  
  
package myapp.junit.categories;  
  
public interface SearchTest extends DBTest  
{  
  
}  
  
The presenter clicks the CRUDTests.java tab. The CRUDTests.java file contains the following code:  
  
package myapp.re;  
  
import myapp.junit.categories.CRUDTest;  
  
@RunWith(Categories.class)  
@IncludeCategory(CRUDTest.class)  
@SuiteClasses(TestRealEstateDAO.class)  
public class CRDTests  
{  
  
}*   
  
each of these guys come up and they work within each other to be able to allow us to categorize those things as a group. So now that I have the categories defined, I can define a new type of test suite, such as my CRUDTests here. And so, like a normal test suite, we need to add the RunWith, but in this case, the RunWith…we need to pick the Categories runner. The Categories runner will say, don't just run the whole test suite, run this just looking for these methods or entire classes – you can categorize the Category annotation on an entire class as well – that give us these tests, that yield these types of test. And so I want to include for this the category CRUDTest. And the SuiteClasses I want to look at inside of here, I can have any number of them, but in this case, we just want the TestRealEstate one; just the one that's going on here. So when I go and run this guy, you can see it's going to go through and it's not going to run all the tests, it's just going to run the CRUDTest that was inside of here. And for the TestRealEstate it ran these three: the load, the delete, and the update. Now if we go back to our real estate…TestRealEstate class here, notice it did not run the search one; it did not run this not test over inside of here. So this is actually not a database test at all, and it has no category in it, and so it didn't run it – it ignored it. It only looked for things inside of the CRUDTest category.   
*The presenter closes the DBTest.java and SearchTest.java files.  
  
In the CRUDTests.java file, the presenter highlights the code @RunWith(Categories.class).  
  
Next he highlights the code @IncludeCategory(CRUDTest.class).  
  
The presenter highlights the class specified in the SuiteClasses annotation:  
  
TestRealEstateDAO.class.  
  
The presenter clicks the Run button to run the CRUDTests.java code.   
  
The left-hand pane changes from the Package Explorer view to the JUnit view. In the JUnit view, the runtime of the top-level test node, myapp.re.CRUDTests, is 0.367 seconds.  
  
The presenter expands the top-level test node to reveal the child test node myapp.re.TestRealEstateDAO, which had a runtime of 0.367 seconds.  
  
The presenter expands the myapp.re.TestRealEstateDAO child test node to reveal three test methods and their associated runtimes – testLoadAndDelete (0.240 s), testAdd (0.064 s), and testUpdate (0.062 s).  
  
In the editor, the presenter clicks the TestRealEstateDAO.java tab. The TestRealEstateDAO.java file contains the following code:  
  
package myapp.re;  
  
import static org.hamcrest.CoreMatchers.hasItem;  
  
public class TestRealEstateDAO  
{  
   private static ConfigurationFactory config;  
   private RealEstateDAO dao;  
   private RealEstate sample;  
     
   @BeforeClass  
   public static void setupHibernate()  
   {  
       config = new ConfigurationFactory();  
   }  
  
   @Before  
   public void setupDAO()  
   {  
       dao = new RealEstateDAO(config.getSessionFactory());  
   }  
  
   @Before  
   public void setupSample()  
   {  
       sample = createRealEstate();  
   }  
     
   @After  
   public void sayHi()  
   {  
       System.out.println("Hello");  
   }  
     
   @After  
   public void wipeTable()  
   {  
       Session session = config.getSessionFactory().getCurrentSession();  
       session.beginTransaction();  
       session.createSQLQuery("delete from RealEstate").executeUpdate();  
       session.getTransaction().commit();  
   }  
     
   @AfterClass  
   public static void closeDatabase()  
   {  
       config.getSessionFactory().close();  
   }  
     
   @Test  
   @Category(CRUDTest.class)  
   public void testAdd()  
   {  
       dao.add(sample);  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the one we just added", all, hasItem(sample));  
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testUpdate()  
   {  
       dao.add(sample);  
         
       double updatedValue = 200000;  
       sample.setValue(updatedValue);  
         
       dao.update(sample);  
         
       RealEstate updated = dao.load(sample.getId());  
       assertEquals("New value should be updated", updatedValue, updated.getValue(), 0.001);  
                 
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testLoadAndDelete()  
   {  
       dao.add(sample);  
       long saveId = sample.getId();  
         
       RealEstate added = dao.load(saveId);  
       assertNotNull("Should have loaded the one we just added", added);  
         
       dao.delete(sample);  
         
       RealEstate deleted = dao.load(saveId);  
       assertNull("It should not have been loaded", deleted);  
   }  
  
   @Test  
   @Category(SearchTest.class)  
   public void testLoadAll()  
   {  
       RealEstate r1 = createRealEstate();  
       RealEstate r2 = createRealEstate();  
       RealEstate r3 = createRealEstate();  
       dao.add(r1);  
       dao.add(r2);  
       dao.add(r3);  
  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the ones we just added", all, hasItems(r1, r2, r3));  
       assertEquals("We expect 3 items", 3, all.size());  
   }  
     
     
   @Test  
   public void randomNotADBTest()  
   {  
       fail("Not marked with a category, don't run me");  
   }  
     
   private RealEstate createRealEstate()  
   {  
       sample = new RealEstate();  
       sample.setValue(100000);  
       Address a = new Address();  
       a.setStreet("123 Main");  
       a.setCity("My Town");  
       a.setState("My State");  
       a.setZip("12345");  
       sample.setAddress(a);  
       return sample;  
   }  
}  
  
The presenter highlights the SearchTest class in the following segment of code:  
  
@Test  
@Category(SearchTest.class)  
public void testLoadAll()  
  
The presenter highlights the randomNotADBTest method in the following segment of code:  
  
@Test  
public void randomNotADBTest()  
  
Next the presenter clicks the CRUDTests.java tab. The CRUDTests.java file contains the following code:  
  
package myapp.re;  
  
import myapp.junit.categories.CRUDTest;  
  
@RunWith(Categories.class)  
@IncludeCategory(CRUDTest.class)  
@SuiteClasses(TestRealEstateDAO.class)  
public class CRDTests  
{  
  
}  
  
The presenter highlights the CRUDTest class in the following segment of code:  
  
@RunWith(Categories.class)  
@IncludeCategory(CRUDTest.class)  
@SuiteClasses(TestRealEstateDAO.class)  
public class CRDTests*   
  
I can show you again, for SearchTest, I go and run this suite, again, using the Categories runner, using the SearchTest, the SearchTest interface which categorizes those together. I have run whatever classes were out there, and I just ran one test in this case: the LoadAll inside of there. And so this Categories allows us to be very, very specific, and I can actually run all the database tests whether they are CRUD tests, or whether they are any of those guys, you can see all the tests run. But it did not run that one test that's not marked with a Category – this not a database test – it leaves this guy out intentionally. And so using categories is a really powerful way to allow me to focus in, to get my tests, to just run this type of test. So even better than just a test suite that runs everything, I can define at a method level much more than at class level, which types of tests I want to run, but still keep them grouped together. Because remember, grouping them gives me fixtures, and the Before and After and BeforeClass, and After test tags that I would like to use, and like to reuse, but still be able to customize which methods run as part of a test suite.   
*In the editor, the presenter closes the CRUDTests.java file and clicks the SearchTests.java tab. The SearchTests.java file contains the following code:  
  
package myapp.re;  
  
import myapp.junit.categories.SearchTest;  
  
@RunWith(Categories.class)  
@IncludeCategory(SearchTest.class)  
@SuiteClasses(TestRealEstateDAO.class)  
public class SearchTests  
{  
  
}  
  
The presenter clicks Run to run the SearchTests.java code. In the JUnit view, the top-level test node is myapp.re.SearchTests and its runtime is 0.290 seconds.  
  
In the SearchTests.java file, the presenter highlights the code @RunWith(Categories.class).  
  
Next he highlights the code (SearchTest.class).  
  
In the JUnit view, the presenter expands the top-level test node myapp.re.SearchTests to reveal the myapp.re.TestRealEstateDAO child test node, which had a runtime of 0.290 seconds.   
  
The presenter expands the myapp.re.TestRealEstateDAO child test node to reveal the following test method and its associated runtime – testLoadAll (0.290 s).  
  
In the editor, the presenter clicks the DBTests.java tab. The DBTests.java file contains the following code:  
  
package myapp.re;  
  
import myapp.junit.categories.DBTest;  
  
@RunWith(Categories.class)  
@IncludeCategory(DBTest.class)  
@SuiteClasses(TestRealEstateDAO.class)  
public class DBTests  
{  
  
}  
  
The presenter clicks Run to run the DBTests.java code.  
  
In the JUnit view, the presenter expands the top-level test suite myapp.re.DBTests to reveal the myapp.re.TestRealEstateDAO child test node and its runtime of 0.460 seconds.  
  
He then expands the myapp.re.TestRealEstateDAO child test node to reveal the following test methods and associated runtimes – testLoadAndDelete (0.249 s), testAdd (0.057 s), testLoadAll (0.090 s), and testUpdate (0.064 s).  
  
In the editor, the presenter clicks the TestRealEstateDAO.java tab. The TestRealEstateDAO.java file contains the following code:  
  
package myapp.re;  
  
import static org.hamcrest.CoreMatchers.hasItem;  
  
public class TestRealEstateDAO  
{  
   private static ConfigurationFactory config;  
   private RealEstateDAO dao;  
   private RealEstate sample;  
     
   @BeforeClass  
   public static void setupHibernate()  
   {  
       config = new ConfigurationFactory();  
   }  
  
   @Before  
   public void setupDAO()  
   {  
       dao = new RealEstateDAO(config.getSessionFactory());  
   }  
  
   @Before  
   public void setupSample()  
   {  
       sample = createRealEstate();  
   }  
     
   @After  
   public void sayHi()  
   {  
       System.out.println("Hello");  
   }  
     
   @After  
   public void wipeTable()  
   {  
       Session session = config.getSessionFactory().getCurrentSession();  
       session.beginTransaction();  
       session.createSQLQuery("delete from RealEstate").executeUpdate();  
       session.getTransaction().commit();  
   }  
     
   @AfterClass  
   public static void closeDatabase()  
   {  
       config.getSessionFactory().close();  
   }  
     
   @Test  
   @Category(CRUDTest.class)  
   public void testAdd()  
   {  
       dao.add(sample);  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the one we just added", all, hasItem(sample));  
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testUpdate()  
   {  
       dao.add(sample);  
         
       double updatedValue = 200000;  
       sample.setValue(updatedValue);  
         
       dao.update(sample);  
         
       RealEstate updated = dao.load(sample.getId());  
       assertEquals("New value should be updated", updatedValue, updated.getValue(), 0.001);  
                 
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testLoadAndDelete()  
   {  
       dao.add(sample);  
       long saveId = sample.getId();  
         
       RealEstate added = dao.load(saveId);  
       assertNotNull("Should have loaded the one we just added", added);  
         
       dao.delete(sample);  
         
       RealEstate deleted = dao.load(saveId);  
       assertNull("It should not have been loaded", deleted);  
   }  
  
   @Test  
   @Category(SearchTest.class)  
   public void testLoadAll()  
   {  
       RealEstate r1 = createRealEstate();  
       RealEstate r2 = createRealEstate();  
       RealEstate r3 = createRealEstate();  
       dao.add(r1);  
       dao.add(r2);  
       dao.add(r3);  
  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the ones we just added", all, hasItems(r1, r2, r3));  
       assertEquals("We expect 3 items", 3, all.size());  
   }  
     
     
   @Test  
   public void randomNotADBTest()  
   {  
       fail("Not marked with a category, don't run me");  
   }  
     
   private RealEstate createRealEstate()  
   {  
       sample = new RealEstate();  
       sample.setValue(100000);  
       Address a = new Address();  
       a.setStreet("123 Main");  
       a.setCity("My Town");  
       a.setState("My State");  
       a.setZip("12345");  
       sample.setAddress(a);  
       return sample;  
   }  
}  
  
The presenter highlights the following segment of code:  
  
@Test  
public void randomNotADBTest()  
{  
   fail("Not marked with a category, don't run me");  
}*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Managing Failure with @Ignore

Learning Objective

*After completing this topic, you should be able to*

* *manage failures with @Ignore in JUnit tests*

**1. Using @Ignore to manage failures**

Well, we want our tests to always be passing, it's inevitable that some will fail over time. As features are not yet developed, are being changed, or simply we just don't have time to fix the test. While we still may want that test around, we now have the ability to temporarily ignore it's execution using a simple annotation. So, let's say for our project we're going to use test-driven development. And we know that we're going to do some math library or something out there. And so we know for our math library we want to test addition, subtraction and multiplication and division. Well, I don't have any tests for that yet, I don't know what the features are going to be, but I know I want those test cases, I can start by defining all these test cases. And then throwing in the Ignore tag, and so the Ignore tag simply is just that, when the test gets executed I ignore this test. So, let me show you what that looks like, if I run this test, the test passes, but all four tests inside of here were ignored. So, they don't do anything and they would pass anyway, but they aren't there yet. So, let's just say as we're going to do addition, we're going to go through test-driven development. We're going to build a new class, so I say okay, file, new actually lets come do it this way, let say  **New – Class**. And this is going to be my Math class outside of here, and so for my Math class I'm going to have a public int add method. And I'm going to say int x and int y and that's doing it's job. And then you know I'm going to say return x+y. Excellent, -   
*In the Eclipse development environment, two files are open on separate tabbed pages in the editor – ShowIgnore.Java and TestSlowBehavior.java.  
  
In the ShowIgnore.Java file, the following code is visible:  
  
package myapp.junit.ignore;  
  
import org.junit.Ignore;  
  
public class ShowIgnore  
{  
   @Test  
   @Ignore  
   public void testAddition()  
   {  
  
   }  
     
   @Test  
   @Ignore  
   public void testSubtraction()  
   {  
         
   }  
  
   @Test  
   @Ignore  
   public void testMultiplication()  
   {  
  
The presenter highlights the @Ignore annotation in the following segment of code:  
  
   @Test  
   @Ignore  
   public void testAddition()  
  
The presenter then clicks Run to run the ShowIgnore.Java code.  
  
The left-hand pane changes from the Package Explorer view to the JUnit view. In the JUnit view, the top-level test is myapp.junit.ignore.ShowIgnore and it's runtime is 0.001 seconds. Four methods are listed below it – testMultiplication, testAddition, testDivision, and testSubtraction. All four test methods were skipped.  
  
The presenter clicks the File menu and hovers over the New option. He then closes the menu.  
  
Next he closes the JUnit view and clicks the Link with Editor button in the Package Explorer. In the Java element hierarchy, the myapp.junit.ignore project expands and the ShowIgnore.java test within it is selected.   
  
The presenter right-clicks the myapp.junit.ignore project and selects New - Class from the shortcut menu that opens. The New Java Class dialog box opens. The presenter types Math in the Name text box and saves the changes.  
  
The new Math.java file opens in the editor. It contains the following code:  
  
package myapp.junit.ignore;  
  
public class Math  
{  
  
}  
  
The presenter adds the following code to the Math.java file:  
  
   public int add(int x, int y)  
   {  
       return x + y;  
   }*   
  
so I know this is going to be out there. Now, I'm also going to say public int divide and I'm going to say int x and int y, but I don't know how to implement this yet. So, say return 0, it's not implemented yet, I don't know what's going to happen here and so forth. But again, this is the idea of test-driven development, I define test cases and then I go build them from there. So, here I can build my Math object, not that Java Math object, run my Math object inside of here, and I can go and implement this guy, assertEquals and I can say it expected is 4, then the actual is m.add(2, 2). Now, this Test is ready to go, I want to start executing this Test, now that it's build and ready to go. So, I can just simply take out the Ignore, great I'm ready to go. Look it's working, great, now I'm building this up bit by bit, I can see there's tests that aren't being implemented. I can factor that into my results, I can see that report that comes up and say three tests were ignored. You see three are skipped, it's reported right here it's being skipped. So I know I have more work to do testing, I'm not done with my unit testing. But -   
*The presenter continues to add to the code in the Math.java file, which in full is  
  
package myapp.junit.ignore;  
  
public class Math  
{  
   public int add(int x, int y)  
   {  
       return x + y;  
   }  
     
   public int divide (int x, int y)  
   {  
       return 0;  
   }  
}  
  
The presenter clicks the ShowIgnore.java tab in the editor. In the ShowIgnore.java file, the following code is visible:  
  
package myapp.junit.ignore;  
  
import org.junit.Ignore;  
  
public class ShowIgnore  
{  
   @Test  
   @Ignore  
   public void testAddition()  
   {  
  
   }  
     
   @Test  
   @Ignore  
   public void testSubtraction()  
   {  
         
   }  
  
   @Test  
   @Ignore  
   public void testMultiplication()  
   {  
  
Beneath the testAddition method, the presenter adds to the code to define a Math object. He then removes the @Ignore annotation above the testAddition method. The visible code in the ShowIgnore.java file is now  
  
package myapp.junit.ignore;  
  
import org.junit.Ignore;  
  
public class ShowIgnore  
{  
   @Test  
   public void testAddition()  
   {  
       Math m = new Math();  
       assertEquals(4, m.add(2, 2));  
   }  
     
   @Test  
   @Ignore  
   public void testSubtraction()  
   {  
         
   }  
  
   @Test  
   @Ignore  
   public void testMultiplication()  
   {   
  
The presenter clicks Run to run the ShowIgnore.java code.  
  
The test progress details and hierarchy aren't visible in the JUnit view because only the editor pane is in focus.*   
  
I know that I'm not failing at the same time, they are just not implemented yet and that's a different statistic. It's something else we want to find out from here. Now, this being said there's other reasons that could come up. I can go through and do Ignore the division, because this is not done yet and these aren't implemented at all. I could add this in and it might be failing in and so I want to leave it out. Here is another example, so this is a test that just runs really slow. And as we're going through here and you can see that the first Test takes forever, you know over two seconds to do and it's failing. You know what this test is unimportant, it's a performance test, we know functionally it's working. It's a very odd scenario that doesn't happen very often, I don't want that you have this reported so I just say @Ignore. And when I run this now, put it on testSlow instead of testVerySlow, I can go back and fix that, I'll put on VerySlow, the proper one, I'm going to skip that test that was failing. And so now the third one is failing too, so I can Ignore that one or fix it, I have that choice. So, the Ignore annotation gives us the flexibility of instead of choosing the design of the test, it gives us that extra dimension of not only having tested or passing and failing, but test that we're skipping for now for some reason. So, use this annotation that really improve your process, but without having to change the design of your tests.   
*In the ShowIgnore.java file, the presenter scrolls down and refers to the testDivision method in the following segment of   
code:  
  
   @Test  
   @Ignore  
   public void testDivision()  
   {  
  
   }  
  
The presenter briefly refers to the testSubtraction and testMultiplication methods above the testDivision method.  
  
In the editor, the presenter clicks the TestSlowBehavior.java tab. In the TestSlowBehavior.java file, the following code is visible:  
  
package myapp.junit.timeout;  
  
import static org.junit.Assert.assertEquals;  
  
public class TestSlowBehavior  
{  
   private SlowBehavior target;  
     
   @Before  
   public void setup()  
   {  
       target = new SlowBehavior();  
   }  
     
   @Test(timeout=2000)  
   public void testSlow()  
   {  
       target.doSomethingSlow();  
   }  
  
   @Test(timeout=2000)  
   public void testVerySlow()  
   {  
       target.doSomethingVerySlow();  
  
The presenter clicks the Run button to run the TestSlowBehavior.java test code.  
  
In the JUnit view, the top-level test is myapp.junit.timeout.TestSlowBehavior and its runtime is 4.000 seconds. Three methods are listed below it – testVerySlow, testSlow, and notJustTime. The testVerySlow method failed with a runtime of 2.000 seconds. The testSlow and notJustTime methods both have a runtime of 1.000 seconds.  
  
The presenter adds the @Ignore annotation to the testSlow method. In the TestSlowBehavior.java file, the following code is visible:  
  
import static org.junit.Assert.assertEquals;  
  
public class TestSlowBehavior  
{  
   private SlowBehavior target;  
     
   @Before  
   public void setup()  
   {  
       target = new SlowBehavior();  
   }  
     
   @Test(timeout=2000)  
   @Ignore  
   public void testSlow()  
   {  
       target.doSomethingSlow();  
   }  
  
   @Test(timeout=2000)  
   public void testVerySlow()  
   {  
  
Next the presenter clicks Run to run the modified TestSlowBehavior.java code.  
  
In the JUnit view, the top-level test node myapp.junit.timeout.TestSlowBehavior has a runtime of 2.986 seconds. The following test methods and associated runtimes are listed below it: testVerySlow (1.985 s), notJustTime method (1.000 s), and testSlow (0.000 s). The testSlow method was skipped.  
  
In the TestSlowBehavior.java file, the presenter deletes the @Ignore annotation above the testSlow method and pastes it above the testVerySlow method. In the TestSlowBehavior.java file, the following code is now visible:  
  
import static org.junit.Assert.assertEquals;  
  
public class TestSlowBehavior  
{  
   private SlowBehavior target;  
     
   @Before  
   public void setup()  
   {  
       target = new SlowBehavior();  
   }  
     
   @Test(timeout=2000)  
   public void testSlow()  
   {  
       target.doSomethingSlow();  
   }  
  
   @Test(timeout=2000)  
   @Ignore  
   public void testVerySlow()  
   {  
  
The presenter clicks Run to run the modified TestSlowBehavior.java code.  
  
In the JUnit view, the top-level test node myapp.junit.timeout.TestSlowBehavior has a runtime of 2.002 seconds. The testVerySlow method was skipped and its runtime is 0.000 seconds. The testSlow method's runtime is 0.998 seconds. The notJustTime method has failed with a runtime of 1.004 seconds.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Testing Time

Learning Objective

*After completing this topic, you should be able to*

* *configure timeouts in JUnit tests*

**1. Test execution times**

Most testing is directed at functional requirements and satisfying the behavior within the system. We cannot forget, however, that execution time is a factor for the acceptability of solutions as well. A simple tweak to the test annotation, allows us a really quick check to validate if the elapsed duration of any test is also acceptable. So our functionality under test this time is no functionality at all. We're going to ignore the functional side of this and we are just simply going to have a test that sits around for a second. It takes a long time to execute the code inside of there. And then another one that sits around for 10 seconds. And in this 10 second one it's going to be longer than acceptable duration. We are okay with a second, but we're not okay with 10 seconds. So as we define our tests, again we are not worried about the functionality we're just going to do something slower or something very slow. But when we define the test annotation, we define it with a timeout. It's one of only two attributes I can define a test with. But the time out is saying "In milliseconds how long is acceptable?" So 2000 milliseconds, that's a two seconds, is it too long. If it goes past that I don't want to keep running the test. And so as I go through and I run this test, we can see this very slow test, which would, otherwise would take 10 seconds to execute, dies after two seconds. And it dies with an exception saying and the test timed out after two seconds.   
*In the Eclipse development environment, two files are open on separate tabbed pages in the editor – TestSlowBehavior.Java and SlowBehavior.java.  
  
In the TestSlowBehavior.Java file, the following code is visible:  
  
   private SlowBehavior target;  
     
   @Before  
   public void setup()  
   {  
       target = new SlowBehavior();  
   }  
     
   @Test(timeout=2000)  
   public void testSlow()  
   {  
       target.doSomethingSlow();  
   }  
  
   @Test(timeout=2000)  
   public void testVerySlow()  
   {  
       target.doSomethingVerySlow();  
   }  
  
   @Test(timeout=2000)  
   public void notJustTime()  
   {  
       assertEquals("Basic math is still true", 4, 2 + 2);  
  
The presenter clicks the SlowBehavior.java tab. In the SlowBehavior.java file, the following code is visible:  
  
package myapp.junit.timeout;  
  
public class SlowBehavior  
{  
   public void doSomethingSlow()  
   {  
       try  
       {  
           Thread.sleep(1000);  
       } catch (InterruptedException e)  
       {  
           e.printStackTrace();  
       }  
   }  
  
   public void doSomethingVerySlow()  
   {  
       try  
       {  
           Thread.sleep(10000);  
       } catch (InterruptedException e)  
       {  
           e.printStackTrace();  
       }  
  
The presenter highlights the code Thread.sleep(1000); in the following segment of code:  
  
   public void doSomethingSlow()  
   {  
       try  
       {  
           Thread.sleep(1000);  
       }  
  
He then highlights the code Thread.sleep(10000); in the following segment of code:  
  
   public void doSomethingVerySlow()  
   {  
       try  
       {  
           Thread.sleep(10000);  
       }  
  
The presenter clicks the TestSlowBehavior.Java tab. In the TestSlowBehavior.Java file, the following code is visible:  
       
   private SlowBehavior target;  
     
   @Before  
   public void setup()  
   {  
       target = new SlowBehavior();  
   }  
     
   @Test(timeout=2000)  
   public void testSlow()  
   {  
       target.doSomethingSlow();  
   }  
  
   @Test(timeout=2000)  
   public void testVerySlow()  
   {  
       target.doSomethingVerySlow();  
   }  
  
   @Test(timeout=2000)  
   public void notJustTime()  
   {  
       assertEquals("Basic math is still true", 4, 2 + 2);  
  
The presenter highlights the code target.doSomethingSlow();  
  
He then highlights the code target.doSomethingVerySlow();  
  
Next the presenter refers to the timeout attribute, highlighting the code @Test(timeout=2000).  
  
The presenter clicks the Run button to run the TestSlowBehavior.Java test code.  
  
In the JUnit view, the top-level test is myapp.junit.timeout.TestSlowBehavior and it's runtime is 3.999 seconds. The following test methods and associated runtimes are listed below it: testVerySlow (1.999 s), testSlow (1.000 s),   
and notJustTime (1.000 s). The presenter selects the testVerySlow test method which returned an error.  
  
In the Failure Trace section of the JUnit view, the following exception is listed:  
  
java.lang.Exception: test timed out after 2000 milliseconds*   
  
The second test there, again it's very slow but. I'm sorry, the first test there is slow, but it's still within the timeframe, it takes a second so it's fine. Now the third test that we show inside of here is also is a timeout of two seconds. And just to show you it doesn't change anything, if I do some assertions inside of here it won't close anything there. But this says asserting, that something is known and so it fails this is down to fail, so it's still going to fail. It's still going to fail based off of the assertions. It just could say within that test it's going to keep testing until the time when that's over with. If it goes longer than that time then it's no longer acceptable and we want to be able to change that as it goes. Now as we are doing our tests we might be bucking with this, we might be changing around. So we can say what if one second is the new functional requirement that's going on inside of here. We need it to be done quicker, we need it more responsive for users. So as we run the test here, you see the first one fails, the second one passes. As I keep running it though there's a chance it could fail at the one second boundary. It's not going to do it for me now, so let me just cheat a little bit. We'll make it 990 milliseconds and see if we can get it to cheat and do that. But -   
*In the TestSlowBehavior.Java file, the presenter mistakenly refers to the following code for the second test method:  
  
   @Test(timeout=2000)  
   public void testVerySlow()  
   {  
       target.doSomethingVerySlow();  
  
The presenter corrects his mistake by referring instead to the following code for the fist test method:  
  
   @Test(timeout=2000)  
   public void testSlow()  
   {  
       target.doSomethingSlow();  
  
In the JUnit view, the presenter selects the testSlow method, which ran successfully with a runtime of 1.000 seconds.  
  
Next the presenter scrolls down the TestSlowBehavior.Java file to reveal the full code for the third test method:  
  
   @Test(timeout=2000)  
   public void notJustTime()  
   {  
       assertEquals("Basic math is still true", 4, 2 + 2);  
       target.doSomethingSlow();  
       asserNull("This will fail", target);  
   }  
  
The presenter highlights the code @Test(timeout=2000) above the notJustTime method.  
  
He highlights the code assertEquals("Basic math is still true", 4, 2 + 2); below the notJustTime method.  
  
The presenter highlights the code asserNull("This will fail", target); below the notJustTime method.  
  
In the JUnit view, the presenter selects the notJustTime test method, which failed with a runtime of 1.000 seconds. In the Failure Trace section of the JUnit view, the following error details display:  
  
java.lang.AssertionError: This will fail expected null, but was:<myapp.juni  
at myapp.junit.timeout.TestSlowBehavior.notJustTime(TestSlowBehavio  
  
In the TestSlowBehavior.Java file, the presenter changes the timeout attribute for the testSlow method from 2000 to 1000. The code that's visible is now as follows:  
  
   private SlowBehavior target;  
     
   @Before  
   public void setup()  
   {  
       target = new SlowBehavior();  
   }  
     
   @Test(timeout=1000)  
   public void testSlow()  
   {  
       target.doSomethingSlow();  
   }  
  
   @Test(timeout=2000)  
   public void testVerySlow()  
   {  
       target.doSomethingVerySlow();  
   }  
  
   @Test(timeout=2000)  
   public void notJustTime()  
   {  
       assertEquals("Basic math is still true", 4, 2 + 2);  
       target.doSomethingSlow();  
       asserNull("This will fail", target);  
   }  
}  
  
The presenter clicks Run to run the modified TestSlowBehavior.Java code. In the JUnit view, the testVerySlow method has returned an error with a runtime of 2.000 seconds. The testSlow   
method beneath it ran successfully with a runtime of 1.000 seconds.  
  
The presenter runs the TestSlowBehavior.Java code again. The testVerySlow method throws an exception with a runtime of 1.985 seconds and the testSlow method runs successfully with a runtime of 1.000 seconds.  
  
In the TestSlowBehavior.Java file, the presenter changes the timeout attribute for the testSlow test method from 1000 to 990. The following modified code is now visible:  
  
   private SlowBehavior target;  
     
   @Before  
   public void setup()  
   {  
       target = new SlowBehavior();  
   }  
     
   @Test(timeout=990)  
   public void testSlow()  
   {  
       target.doSomethingSlow();  
   }  
  
   @Test(timeout=2000)  
   public void testVerySlow()  
   {  
       target.doSomethingVerySlow();  
   }  
  
   @Test(timeout=2000)  
   public void notJustTime()  
   {  
       assertEquals("Basic math is still true", 4, 2 + 2);  
       target.doSomethingSlow();  
       asserNull("This will fail", target);  
   }  
}  
  
The presenter clicks Run to run the modified TestSlowBehavior.Java code.*  
  
the second it gets to that point, you can see now it fails. Even within that one thousandth, ten thousandth of a second. Once that tires up it fails and its really that threshold and it goes inside of there. And so the bad news is it doesn't tell me how badly I failed. The good news is it does make me, allow me to set those non-functional requirements, set that time factor inside of these tests. It doesn't help me tell me why. I still have to do the debugging. I have to say, "Hey, was some other process sort of running some other task and that's why it was slow? Did I have a slow database connection this time?" I still have to do that debugging, but it gave me a warning sign. So, again let's say it is one second and it works most of the time then I don't really have to worry about it. It's only when it starts failing that I need to worry about it. Which is why I might do one second or one and a half seconds, to give me a warning well before I get to my two second boundary. Because once you hit that boundary this test just fails, you see it fails hard, it does an exception, I have a stack trace over here. Everything is going on really hard inside of there because it's not wanting all that to happen. It's not wanting anything to happen. Actually this stack trace is because of the behavior, this it interrupts the tread that's going on there, it stops the thread. But again it's crashing this guy hard, it's not letting it go forward. So that is the timeout that's how we can add in the functional element and the non-functional element into our testing.   
*In the JUnit view, the presenter selects the testSlow method, which failed with a runtime of 0.990 seconds. In the   
Failure Trace section, the following error information displays:  
  
java.lang.Exception failure: test timed out after 990 milliseconds.  
at java.lang.Thread.sleep(Native Method)  
at.myapp.junit.timeout.SlowBehavior.doSomethingSlow(SlowBehavior.ja  
  
The presenter refers to the testSlow method's timeout attribute, @Test(timeout=990).  
  
Next the presenter changes the testSlow method's timeout attribute back from 990 to 1000.    
  
He clicks Run to run the modified TestSlowBehavior.Java code. In the JUnit view, the presenter selects the testVerySlow method, which has returned an error with a runtime of 2.000 seconds.  
  
In the Failure Trace section, the following error information is listed:  
  
java.lang.Exception failure: test timed out after 990 milliseconds.  
at java.lang.Thread.sleep(Native Method)  
at.myapp.junit.timeout.SlowBehavior.doSomethingSlow(SlowBehavior.ja  
  
The presenter refers to the stack trace details listed in the Console view below the editor. The full stack trace output is as follows:  
  
java.lang.InterruptedException: Sleep Interrupted  
       at java.lang.Thread.sleep(Native Method)  
       at mapp.junit.timeout.SlowBehavior.doSomethingVerySlow(SlowBehavior.java:20)  
       at mapp.junit.timeout.TestSlowBehavior.testVerySlow(TestSlowBehavior.java:29)  
       at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
       at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)  
       at java.lang.reflect.DelegatingMethodAccessorImpl.invoke(Unkown Source)  
       at java.lang.reflect.Method.invoke(Unknown Source)  
  
The presenter clicks the Native Method hyperlink in the second line of output. This activates the SlowBehavior.Java file in the editor. In the SlowBehavior.Java file, the following code is visible:  
  
public class SlowBehavior  
{  
   public void doSomethingSlow()  
   {  
       try  
       {  
           Thread.sleep(1000);  
       } catch (InterruptedException e)  
       {  
           e.printStackTrace();  
       }  
   }  
  
   public void doSomethingVerySlow()  
   {  
       try  
       {  
           Thread.sleep(10000);  
       } catch (InterruptedException e)  
       {  
           e.printStackTrace();  
       }  
   }  
}  
  
Finally, the presenter highlights the code Thread.sleep(10000); in the following segment of code:  
  
   public void doSomethingVerySlow()  
   {  
       try  
       {  
           Thread.sleep(10000);  
       }*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Customizing Test Runs

Learning Objective

*After completing this topic, you should be able to*

* *customize JUnit test runs*

**1. Customizing JUnit test runners**

JUnit provides a basic test runner that will execute most test cases for you. As we expand the custom features that needs special test conditions or use third party solutions, we need to use the smarter test runner to execute our tests. We can customize any test case that tells JUnit the expected test runner, we wish to use for that test. So, our base test case doesn't use any test runner, it uses the default one, we don't have to do anything with it. When we execute it, it uses whichever runner is chosen by the framework executing at the time. Now, if we know we use a special feature inside of here, I use the RunWith command and inside of that, I have to give it a class object of a runner. And so one example that's really simple is, there is a JUnit4 class that I can do, and I've to do a class and since I do the object instance to pass into this guy, that will allow us to ensure tests are run within the JUnit4 framework. This isn't normally required, this doesn't absolutely have to happen, but if you have a feature that you know, or some way you definitely want to know JUnit4 is used as a test runner, it doesn't change the outcome of this guy, but it might, if you're on a different platform. Again this is optional you don't have to put this in there, it's something you don't need for basic tests.   
*In the Eclipse development environment, five files are open on separate tabbed pages in the editor – TestTarget.java, TestParameters.java, TestSuiteExample.java, CRUDTests.java, and TestMe.java.  
  
In the TestTarget.java file, the following code is visible:  
  
package myapp.junit;  
  
import junit.framework.TestCase;  
  
public class TestTarget  
{  
@Test  
public void testAdd()  
{  
MyTarget target = new MyTarget();  
  
int answer = target.basicAdd(2, 2);  
TestCase.assertEquals("Two plus two is four right?", 4, answer);  
}  
  
@Test  
public void badTestCase()  
{  
MyTarget target = new MyTarget();  
  
int answer = target.basicAdd(2, 2);  
TestCase.assertEquals("Adding is fun", 6, answer);  
}  
  
The presenter refers to the @Test annotations within the TestTarget.java code.  
  
The presenter clicks Run to run the TestTarget.java code.  
  
In the JUnit view, the top-level test node is myapp.junit.TestTarget and its runtime is 0.007 seconds. Beneath it, four test methods are listed with corresponding runtimes – testDivideByZero (0.002 s), testAdd (0.001 s), badTestCase (0.003 s), and sillyTest (0.000 s).  
  
The presenter refers to the default runner information displayed to the right of the top-level test node, which is "Runner: JUnit 4".  
  
On the TestTarget.java tabbed page, the presenter adds the code @RunWith(JUnit4.class). The code visible in the TestTarget.java file is now  
  
package myapp.junit;  
  
import junit.framework.TestCase;  
  
@RunWith(JUnit4.class)  
public class TestTarget  
{  
@Test  
public void testAdd()  
{  
MyTarget target = new MyTarget();  
  
int answer = target.basicAdd(2, 2);  
TestCase.assertEquals("Two plus two is four right?", 4, answer);  
}  
  
@Test  
public void badTestCase()  
{  
MyTarget target = new MyTarget();  
  
int answer = target.basicAdd(2, 2);  
TestCase.assertEquals("Adding is fun", 6, answer);  
}  
  
The presenter clicks the Run button.  
  
In the JUnit view, the test methods' runtime values show no significant changes and the top-level test node displays the same runner information – Runner: JUnit 4.  
  
In the TestTarget.java file, the presenter highlights the newly added code @RunWith(JUnit4.class). He then deletes that code.*   
  
But for interesting tests like our parameterized tests, ones where we have a dataset that's passed into the test each time, it's absolutely essential. So, if I run it underneath the parameterized test runner, you can see it passes in data over and over and over again. The data is loaded and then passed to this same test case again, and all the tests are executed each time on the test case. Now, if I don't have that parameterized loader inside of here, it's got to try and run it as a normal test case and it will fail, because every test case should have an empty constructor, and in a parameterized test we don't do that. We have a constructor where we pass in the parameters. So, again the test runner is essential for certain test cases or features. As I'm doing test suites if I wanted to execute any tests, I have to use this test suite runner, it picks up our SuiteClasses and runs all the different classes, and all the different tests in these classes. This runner is customized to be able to be smart and look at the annotations and load this guy up. We similarly need that for categories or if we're going within a category, we need a runner that's smart enough to say, okay what are all the classes you want to include, and then what's the categories you want to execute within those classes. And so we need these smart custom runners to be able to run these special features. Now, as we said there's third parties out there that work as well.   
*The presenter clicks the TestParameters.java tab in the editor. In the TestParameters.java file, the following code is visible:  
  
package myapp.junit.parameters;  
  
import static org.junit.Assert.assertEquals;  
  
@RunWith(Parameterized.class)  
public class TestParameters  
{  
   private int               player1;  
   private int               player2;  
   private int               outcome;  
  
   private RockPaperScissors game;  
  
   public TestParameters(int player1, int player2, int outcome)  
   {  
       this.player1 = player1;  
       this.player2 = player2;  
       this.outcome = outcome;  
   }  
  
   @Parameters(name="Rock, Paper, Scissors winner {0} versus {1} => {2}")  
   public static Collection<Integer[]> addedNumbers()  
   {  
       return Arrays.asList(new Integer[][] {   
  
On the TestParameters.java tabbed page, the presenter refers to the following segment of code:  
  
@RunWith(Parameterized.class)  
public class TestParameters  
{  
   private int               player1;  
   private int               player2;  
   private int               outcome;  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.parameters.TestParameters and its runtime is 0.001 seconds. The test progress details show 27/27 tests were run, 18 of which were skipped.  
  
The presenter expands the myapp.junit.parameters.TestParameters test node to reveal the following list of successful tests:  
  
[Rock, Paper, Scissors winner0 versus0 => 0] (0.000 s)  
[Rock, Paper, Scissors winner0 versus1 => 2] (0.000 s)  
[Rock, Paper, Scissors winner0 versus2 => 1] (0.000 s)  
[Rock, Paper, Scissors winner1 versus0 => 1] (0.000 s)  
[Rock, Paper, Scissors winner1 versus1 => 0] (0.000 s)  
[Rock, Paper, Scissors winner1 versus2 => 2] (0.000 s)  
[Rock, Paper, Scissors winner2 versus0 => 2] (0.000 s)  
[Rock, Paper, Scissors winner2 versus1 => 1] (0.000 s)  
[Rock, Paper, Scissors winner2 versus2 => 0] (0.001 s)  
  
In the TestParameters.java file, the presenter deletes the code @RunWith(Parameterized.class). The visible code in the file is now:  
  
package myapp.junit.parameters;  
  
import static org.junit.Assert.assertEquals;  
  
public class TestParameters  
{  
   private int               player1;  
   private int               player2;  
   private int               outcome;  
  
   private RockPaperScissors game;  
  
   public TestParameters(int player1, int player2, int outcome)  
   {  
       this.player1 = player1;  
       this.player2 = player2;  
       this.outcome = outcome;  
   }  
  
   @Parameters(name="Rock, Paper, Scissors winner {0} versus {1} => {2}")  
   public static Collection<Integer[]> addedNumbers()  
   {  
       return Arrays.asList(new Integer[][] {   
               {RockPaperScissors.ROCK, RockPaperScissors.ROCK, 0},  
  
The presenter clicks the Run button.  
  
In the JUnit view, the test progress details show 1/1 tests were run with 1 error. The top-level test node myapp.junit.parameters.TestParameters has a runtime of 0.000 seconds. An initialization error is listed beneath it with a runtime of 0.000 seconds. In the Failure Trace section, the following java.lang.Exception is listed – "Test class should have exactly one public zero-argument constructor."  
  
In the TestParameters.java file, the presenter highlights the following line of code:  
  
public TestParameters(int player1, int player2, int outcome)  
  
The presenter clicks the TestSuiteExample.java tab in the editor. In the TestSuiteExample.java file, the following code is visible:  
  
package myapp.junit.suite;  
  
import myapp.junit.TestTarget;  
  
@RunWith(Suite.class)  
@SuiteClasses({TestRealEstateDAO.class, TestParameters.class, TestTarget.class})  
public class TestSuiteExample  
{  
  
}  
  
The presenter highlights the code @RunWith(Suite.class).  
  
Next the presenter highlights the code @SuiteClasses and refers to the three classes specified after it – TestRealEstateDAO, TestParameters, and TestTarget.  
  
The presenter clicks the CRUDTests.java tab. In the CRUDTests.java file, the following code is visible:  
  
package myapp.re;  
  
import myapp.junit.categories.CRUDTest;  
  
@RunWith(Categories.class)  
@IncludeCategory(CRUDTest.class)  
@SuiteClasses(TestRealEstateDAO.class)  
public class CRUDTests  
{  
  
}  
  
The presenter highlights the code @RunWith(Categories.class).  
  
The presenter clicks the TestMe.java tab. In the TestMe.java file, the following code is visible:  
  
package target.mock;  
  
import static org.junit.Assert.assertEquals;  
  
@RunWith(MockitoJUnitRunner.class)  
public class TestMe  
{  
   @Mock  
   private AmazingFeatures mockedObject;  
     
   @Before  
   public void setup()  
   {  
       target = new UUT(mockedObject);  
   }  
     
   @Test  
   public void checkNumber()  
   {  
       when(mockedObject.guessNumber()).thenReturn(5);  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));  
   }*   
  
There is a spring plug-in to be able to run spring tests, that will use the dependency injection aspects of spring, and make your tests a little bit more configurable. And here's an example of the Mackito framework, this uses mock objects and it loads things up. And this test runner will go through and it'll fake data for you, it'll load up data, it'll load up objects, it'll define behaviors that you want to have in your mock objects. So, this is a totally third party, this is not built by JUnit. This is built by another group of people entirely, but they have this plug-in for JUnit, because they know that their framework is much more powerful when working with JUnit. It is something that JUnit is really better off with the extension out there for. And so whether you are working on local testing or building a framework of your own, you can use custom test runners to define the way tests execute. So, let's say you're not out there to build your own magical framework, you can still extend out, you can still build your own types of test runners. So, if you go through you can build a **New** Java **Class** and YourRunner. There is a couple of ways you could do it, you can extend one of the base runners, so you could extend out that JUnit4 runner if you wanted to.   
*The presenter clicks Run to run the TestMe.java code.  
  
In the JUnit view, the top-level test node is target.mock.TestMe and its runtime is 0.133 seconds. The test progress details show 3/3 tests were run successfully.  
  
The presenter expands the top-level test node and refers to the three successful tests listed below it – checkError, checkTest, and checkNumber.  
  
Next the presenter closes the JUnit view. The Package Explorer view opens in its place. A project named Basics and its source folder are expanded.  
  
The presenter right-clicks the myapp package within the source folder and selects New - Class from the shortcut menu. The New Java Class dialog box opens. The presenter types YourRunner in the Name text box and saves the new class.  
  
A new YourRunner.java file opens in the editor. It contains the following code:  
  
package myapp;  
  
public class YourRunner  
{  
  
}  
  
The presenter begins to add to the code.*   
  
JUnit4ClassRunner, you could go and extend this guy out and you can make your own inside of there, or you could just simply extend out the Runner class, it's an abstract class. Now, the runner class has no built-in capabilities for to build anything. You would have to do that on your own. If you extend out the JUnit runner class, you could call the features of JUnit runner classes as a super class to say, hey go on run this stuff, but otherwise you could do it by yourself. You have a description that talks about how the test could be run and then you have the runner itself. I'm not going to get into how you would run tests, because that's way beyond the scope of what we're looking at here. But I want to show you that by extending it out with a notifier you can add in listeners as tests to run, you can fire test failures if circumstances aren't met, you can listen for test failures, you can create and modify the flow of testing as it's running. And so if that's something you want to or need to do, it's a feature within JUnit and you're welcome to pick up on that and build as you wish. Generally though, JUnit allows customization within test cases and even gives you that customization as you want to execute your tests using the test runners.   
*In the YourRunner.java file, the presenter continues adding to the code. The full code is now  
  
package myapp;  
  
import org.junit.internal.runners.JUnit4ClassRunner  
  
public class YourRunner extends JUnit4ClassRunner  
{  
  
}  
  
The presenter modifies the code again. The full code is now  
  
package myapp;  
  
import org.junit.runner.Runner;  
  
public class YourRunner extends Runner  
{  
  
}  
  
Next the presenter right-clicks below the YourRunner class and selects Override/Implement Methods from the shortcut menu. The Override/Implement Methods dialog box opens. It contains a pane that lists the Runner class and methods that can be overridden or implemented for it, each with a checkbox. The methods are getDescription(), run(RunNotifier), and testCount(). The Runner checkbox, and the getDescription() and run(RunNotifier) checkboxes, are selected by default.  
  
The presenter clicks the OK button to save the settings and close the Override/Implement Methods dialog box.  
  
The following code is now visible in the YourRunner.java file:  
  
import org.junit.runner.Description;  
import org.junit.runner.Runner;  
import org.junit.runner.notification.RunNotifier;  
  
public class YourRunner extends Runner  
{  
  
   @Override  
   public Description getDescription()  
   {  
       // TODO Auto-generated method stub  
       return null;  
   }  
  
   @Override  
   public void run(RunNotifier notifier)  
   {  
       // TODO Auto-generated method stub  
  
   }  
  
}  
  
The presenter deletes the second auto-generated TODO comment and types notifier. in its place. A drop-down list automatically lists possible methods for completing the code, including addFirstListener(), addListener(), equals(), fireTestAssumptionFailed(), fireTestFailure(), fireTestFinshed(), fireTestIgnored(), fireTestRunFinished(), fireTestRunStarted, and fireTestStarted.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Parameters in Tests

Learning Objective

*After completing this topic, you should be able to*

* *use parameters in JUnit tests*

**1. Using parameters in JUnit tests**

Test cases are written to check functionality, but the test space can often be expanded dramatically by simply changing the input values to a test. JUnit provides a mechanism, by which we can execute a test many times across a changing set of input parameters. So, the unit under test we're going to look at here is a player of RockPaperScissors. And this has a method here to determine the winner. And so when player 1 chooses RockPaperScissors, player two chooses RockPaperScissors, this tells you who wins. That's basically the method that we're trying to test right now. And so as we come over to the Test, it's really simple. We have an outcome that we expect, and then we're going to pass in what player1 guessed and what player2 guessed. And we want player1 to do rock, paper and scissors and we want player2 to do rock, paper and scissors, and all combinations therein of that. And so what we can do inside of our test, is instead of having to do each combination either as multiple test cases or looping it ourselves, we can create the values of what the player is expecting to send in for player1, player2, what the outcome is as data items for this test. And parameterized the test by creating a constructor that says, hey give me each one of these guys each time we're going to execute the test. And so now I can create a static method inside of here, that's going to use the Parameters annotation. And the Parameters annotation will go through and say, hey this is all the data I want you to run the test for.   
*In the Eclipse development environment, two files are open on separate tabbed pages in the editor – TestParameters.java, and   
RockPaperScissors.java.  
  
In the RockPaperScissors.java file, the following code is visible:  
  
package myapp.rps;  
  
public class RockPaperScissors  
{  
   public static final int ROCK = 0;  
   public static final int PAPER = 1;  
   public static final int SCISSORS = 2;  
     
   public int determineWinner(int player1Choice, int player2Choice)  
   {  
       if (player1Choice == player2Choice)  
       {  
           return 0;  
       }  
         
       if (player1Choice == ROCK)  
       {  
           if (player2Choice == SCISSORS)  
           {  
               return 1;  
           } else   
           {  
               return 2;  
           }  
  
The presenter highlights the code determineWinner(int player1Choice, int player2Choice).  
  
The presenter clicks the TestParameters.java tab in the editor. The TestParameters.java file contains the following code:  
  
package myapp.junit.parameters;  
  
import static org.junit.Assert.assertEquals;  
  
import java.util.Arrays;  
import java.util.Collection;  
  
import myapp.rps.RockPaperScissors;  
  
import org.junit.Before;  
import org.junit.Ignore;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.junit.runners.Parameterized;  
import org.junit.runners.Parameterized.Parameters;  
  
@RunWith(Parameterized.class)  
public class TestParameters  
{  
   private int               player1;  
   private int               player2;  
   private int               outcome;  
  
   private RockPaperScissors game;  
  
   public TestParameters(int player1, int player2, int outcome)  
   {  
       this.player1 = player1;  
       this.player2 = player2;  
       this.outcome = outcome;  
   }  
  
   @Parameters(name="Rock, Paper, Scissors winner {0} versus {1} => {2}")  
   public static Collection<Integer[]> addedNumbers()  
   {  
       return Arrays.asList(new Integer[][] {   
               {RockPaperScissors.ROCK, RockPaperScissors.ROCK, 0},  
               {RockPaperScissors.ROCK, RockPaperScissors.PAPER, 2},  
               {RockPaperScissors.ROCK, RockPaperScissors.SCISSORS, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.ROCK, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.PAPER, 0},  
               {RockPaperScissors.PAPER, RockPaperScissors.SCISSORS, 2},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.ROCK, 2},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.PAPER, 1},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.SCISSORS, 0},  
       });  
   }  
  
   @Before  
   public void setupGame()  
   {  
       game = new RockPaperScissors();  
   }  
  
   @Test  
   public void testDetermination()  
   {  
       assertEquals("Game outcome", outcome, game.determineWinner(player1, player2));  
   }  
     
   @Ignore  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer1()  
   {  
       assertEquals("Invalid Player 1 input", outcome, game.determineWinner(9, RockPaperScissors.PAPER));  
   }  
  
   @Ignore  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer2()  
   {  
       assertEquals("Invalid Player 2 input", outcome, game.determineWinner(RockPaperScissors.PAPER, -2));  
   }  
}  
  
Below the testDetermination() test method, the presenter refers to elements in the following line of code:  
  
   assertEquals("Game outcome", outcome, game.determineWinner(player1, player2));  
  
The presenter scrolls up in the TestParameters.java file and, below the TestParameters class, refers to the following code:  
  
   private int               player1;  
   private int               player2;  
   private int               outcome;  
  
Below the private method RockPaperScissors, the presenter highlights the following line of code:  
  
   public TestParameters(int player1, int player2, int outcome)  
  
The presenter scrolls down in the file and, below the @Parameters annotation, highlights the static method Collection<Integer[]> addedNumbers().  
  
Next he highlights the @Parameters annotation.  
  
The presenter refers to the following segment of code below the @Parameters annotation:  
  
       return Arrays.asList(new Integer[][] {   
               {RockPaperScissors.ROCK, RockPaperScissors.ROCK, 0},  
               {RockPaperScissors.ROCK, RockPaperScissors.PAPER, 2},  
               {RockPaperScissors.ROCK, RockPaperScissors.SCISSORS, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.ROCK, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.PAPER, 0},  
               {RockPaperScissors.PAPER, RockPaperScissors.SCISSORS, 2},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.ROCK, 2},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.PAPER, 1},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.SCISSORS, 0},*   
  
Now, it's a static method, but it doesn't have to define the data statically as we did here. That static method could go to a database, or it could go to a file, it could go to any external data source and pull that data in. It's just static to help JUnit know there is just one of those, and it's going to be executed here without the other additional overhead that's involved. So, as we go and run this test you can see it runs it a whole bunch of times. It runs it based off of this little tag I put inside here where we say well rock, paper, or scissors, the winner is expected to be 0, the parameter that's coming in here. And we have player1 that made this move and player2 that made this move. So in this case 0 means they tied, 1 and 2 is basically the values that go in, or and go on from there. So there's data that goes on from there and checks all these things, and it's all doing as exposed here. Now, the value of what's proper and what's supposed to be, is based off the values that are passed in here. So, if it's Rock Rock it's going to be value 0, if it's Rock Paper it's going to be value 2, if it's Rock Scissor it's value 1. The test isn't checking the data, you as the coder are defining the data, but I'm defining the data separately. So, if I want to create a special test case, and in this case there's a finite number of test cases you can even define inside of here. But what you can do is build an add test dynamically. Let me add a new type of test, new data that somebody comes up with. They just change an input file or change a database, and be able to create very clever tests that way.   
*In the TestParameters.java file, the presenter refers to the following segment of code below the @Parameters annotation:  
  
public static Collection<Integer[]> addedNumbers()  
   {  
       return Arrays.asList(new Integer[][] {   
               {RockPaperScissors.ROCK, RockPaperScissors.ROCK, 0},  
               {RockPaperScissors.ROCK, RockPaperScissors.PAPER, 2},  
               {RockPaperScissors.ROCK, RockPaperScissors.SCISSORS, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.ROCK, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.PAPER, 0},  
               {RockPaperScissors.PAPER, RockPaperScissors.SCISSORS, 2},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.ROCK, 2},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.PAPER, 1},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.SCISSORS, 0},  
  
The presenter clicks the Run button to run the TestParameters.java code.  
  
In the JUnit view, the top-level test node is myapp.junit.parameters.TestParameters and its runtime is 0.003 seconds. The test progress details show 27/27 tests were run, 18 of   
which were skipped.  
  
The presenter expands the myapp.junit.parameters.TestParameters test node to reveal the following list of successful tests:  
  
[Rock, Paper, Scissors winner0 versus0 => 0] (0.000 s)  
[Rock, Paper, Scissors winner0 versus1 => 2] (0.000 s)  
[Rock, Paper, Scissors winner0 versus2 => 1] (0.000 s)  
[Rock, Paper, Scissors winner1 versus0 => 1] (0.000 s)  
[Rock, Paper, Scissors winner1 versus1 => 0] (0.000 s)  
[Rock, Paper, Scissors winner1 versus2 => 2] (0.000 s)  
[Rock, Paper, Scissors winner2 versus0 => 2] (0.001 s)  
[Rock, Paper, Scissors winner2 versus1 => 1] (0.001 s)  
[Rock, Paper, Scissors winner2 versus2 => 0] (0.001 s)  
  
In the TestParameters.java file, the presenter highlights the following code after the @Parameters annotation:  
  
(name="Rock, Paper, Scissors winner {0} versus {1} => {2}")  
  
In the JUnit view, the presenter refers to the following player values in the first test listed below the myapp.junit.parameters.TestParameters test node:  
  
[Rock, Paper, Scissors winner0 versus0 => 0]  
  
Player 1's move is the value to the right of "Versus" (0). Player 2's move is the value to the right of "=>" (0). The winning move is the value to the right of "winner" (0).  
  
The presenter then refers to the following player values in the second test listed below the myapp.junit.parameters.TestParameters test node:  
  
[Rock, Paper, Scissors winner0 versus1 => 2]  
  
In the TestParameters.java file, the presenter refers to the following segment of code:  
  
       return Arrays.asList(new Integer[][] {   
               {RockPaperScissors.ROCK, RockPaperScissors.ROCK, 0},  
               {RockPaperScissors.ROCK, RockPaperScissors.PAPER, 2},  
               {RockPaperScissors.ROCK, RockPaperScissors.SCISSORS, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.ROCK, 1},  
               {RockPaperScissors.PAPER, RockPaperScissors.PAPER, 0},  
               {RockPaperScissors.PAPER, RockPaperScissors.SCISSORS, 2},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.ROCK, 2},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.PAPER, 1},  
               {RockPaperScissors.SCISSORS, RockPaperScissors.SCISSORS, 0},*   
  
Now, one little wrinkle on this is we go back to our Test here. I have two Test cases that are ignored. If I unignore these guys right here and they are actually showing up Ignore there, I can go through and run all the tests, but one of the tests is suddenly failing. If I ignore both of this and so you can when I run each iteration, it runs every test on the data for all those iterations, good or bad. And so these guys aren't even using those parameters, they are using predetermined game winners inside of here. Or we add this outcome variable, so I don't know what the outcome is going to look like inside of there, so I have to be very careful in this test design. Ideally, what I would do is not have these here at all much less Ignore them. I would use a test suite instead. I would have my predefined test and have my special one-off cases defined in the separate class, and use a test suite to run both of these together. So, again you keep it clean, you can see it runs each of these on each of all these tests, but these two happen to be ignored this case. And so that is the parameter feature that allows you to really expand out your test space. Where instead of handling a couple of data items that you have to hard code yourself, you can either handle a bunch of data items that could be hard coded, or even externalize those data items. And be able to make flexible tests that can be expanded overtime with new data. Even if the functionality of the test is exactly the same.   
*Next the presenter scrolls down to the bottom of the TestParameters.java file. The following code is visible:  
  
   {  
       game = new RockPaperScissors();  
   }  
  
   @Test  
   public void testDetermination()  
   {  
       assertEquals("Game outcome", outcome, game.determineWinner(player1, player2));  
   }  
     
   @Ignore  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer1()  
   {  
       assertEquals("Invalid Player 1 input", outcome, game.determineWinner(9, RockPaperScissors.PAPER));  
   }  
  
   @Ignore  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer2()  
   {  
       assertEquals("Invalid Player 2 input", outcome, game.determineWinner(RockPaperScissors.PAPER, -2));  
   }  
}  
  
The presenter deletes the two @Ignore annotations. The code visible in the TestParameters.java file is now  
  
   public void setupGame()  
   {  
       game = new RockPaperScissors();  
   }  
  
   @Test  
   public void testDetermination()  
   {  
       assertEquals("Game outcome", outcome, game.determineWinner(player1, player2));  
   }  
     
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer1()  
   {  
       assertEquals("Invalid Player 1 input", outcome, game.determineWinner(9, RockPaperScissors.PAPER));  
   }  
  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer2()  
   {  
       assertEquals("Invalid Player 2 input", outcome, game.determineWinner(RockPaperScissors.PAPER, -2));  
   }  
}  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.parameters.TestParameters and its runtime is 0.014 seconds. The test progress details show 27/27 tests were run, with 6   
errors and 3 failures. The top-level test node is expanded and the following tests are listed below it:  
  
[Rock, Paper, Scissors winner0 versus0 => 0]  
[Rock, Paper, Scissors winner0 versus1 => 2]  
[Rock, Paper, Scissors winner0 versus2 => 1]  
[Rock, Paper, Scissors winner1 versus0 => 1]  
[Rock, Paper, Scissors winner1 versus1 => 0]  
[Rock, Paper, Scissors winner1 versus2 => 2]  
[Rock, Paper, Scissors winner2 versus0 => 2]  
[Rock, Paper, Scissors winner2 versus1 => 1]  
[Rock, Paper, Scissors winner2 versus2 => 0]  
  
In the JUnit view, the first test listed below the top-level test node is also expanded. The following three test methods are listed below it:  
  
testDetermination[Rock, Paper, Scissors winner 0 versus 0 => 0]  
testBadPlayer1[Rock, Paper, Scissors winner 0 versus 0 => 0]  
testBadPlayer2[Rock, Paper, Scissors winner 0 versus 0 => 0]  
  
The testBadPlayer2 method threw an exception and the testDetermination and testBadPlayer1 methods ran successfully.  
  
In the TestParameters.java file, the presenter refers to the outcome and game.determinewinner variables in the following segment of code:  
  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer1()  
   {  
       assertEquals("Invalid Player 1 input", outcome, game.determineWinner(9, RockPaperScissors.PAPER));  
   }  
  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer2()  
   {  
       assertEquals("Invalid Player 2 input", outcome, game.determineWinner(RockPaperScissors.PAPER, -2));  
   }  
}  
  
The presenter replaces the @Ignore annotation above the testBadPlayer1 and testBadPlayer2 test methods and highlights the following modified code:  
  
   @Ignore  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer1()  
   {  
       assertEquals("Invalid Player 1 input", outcome, game.determineWinner(9, RockPaperScissors.PAPER));  
   }  
  
   @Ignore  
   @Test(expected=RuntimeException.class)  
   public void testBadPlayer2()  
   {  
       assertEquals("Invalid Player 2 input", outcome, game.determineWinner(RockPaperScissors.PAPER, -2));  
   }  
}  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node myapp.junit.parameters.TestParameters lists a runtime of 0.000 seconds. The test progress details show 27/27 tests were run, 18 of   
which were skipped.   
  
The presenter expands the top-level test node to reveal the following list of successful tests:  
  
[Rock, Paper, Scissors winner0 versus0 => 0] (0.000 s)  
[Rock, Paper, Scissors winner0 versus1 => 2] (0.000 s)  
[Rock, Paper, Scissors winner0 versus2 => 1] (0.000 s)  
[Rock, Paper, Scissors winner1 versus0 => 1] (0.000 s)  
[Rock, Paper, Scissors winner1 versus1 => 0] (0.000 s)  
[Rock, Paper, Scissors winner1 versus2 => 2] (0.000 s)  
[Rock, Paper, Scissors winner2 versus0 => 2] (0.001 s)  
[Rock, Paper, Scissors winner2 versus1 => 1] (0.001 s)  
[Rock, Paper, Scissors winner2 versus2 => 0] (0.001 s)  
  
The presenter expands the first test listed below the top-level test node to reveal the following three test methods:  
  
testDetermination[Rock, Paper, Scissors winner 0 versus 0 => 0]  
testBadPlayer1[Rock, Paper, Scissors winner 0 versus 0 => 0]  
testBadPlayer2[Rock, Paper, Scissors winner 0 versus 0 => 0]  
  
The testDetermination method ran successfully and the testBadPlayer1 and testBadPlayer2 methods were ignored.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Managing Data across Multiple Tests

Learning Objective

*After completing this topic, you should be able to*

* *manage data across multiple JUnit tests*

**1. How to manage data across multiple tests**

JUnit tests are simple when they control all the test data. As test scopes spans the data that is managed outside of JUnit, we must use test fixtures to manage external data. Let's look at a sample where JUnit is testing interactions with the database, to see how the After annotation can be used in these circumstances. So, we have a test here that's doing a number of tests on databases. This is doing loads and adds and loading everything for database and updates, and it's going to run across the database. Now, if we go and look really quick at the contents of the database, we start out with an empty table. That's what we want, we want a clean slate to start with. Now, when we run our tests we go through the first time and we're going to the test database, and we have a test that's failing. So, our load test inside of here is doing some real simple tests, we're adding these three items to the database. We'll use the add setting here and then we're loading everything and we're saying, hey we should have these three items. And we have this extra check to say there should be three items that came back, we added three in, we should look at three back and isn't that what we expect? But if we go and look at the database, well there is more than three items inside of there, why? Well, the other test didn't necessarily clean up after themselves. We would have to write every single test to include the delete inside of there, if we want to make that happen. If we're not testing the delete we might not want to include that feature there. So, that feature is something that's not being tested. If it's broken I don't want to break all of my test cases.   
*In the Eclipse development environment, the JUnit view is open and a file named TestRealEstateDAO.java is open in the editor.  
  
In the JUnit view, the test progress details show 5/5 tests were run, one of which failed. The top-level test node myapp.re.TestRealEstateDAO is expanded and five methods are listed below it: testLoadAndDelete, testAdd, testLoadAll, randomNotADBTest, and testUpdate. The randomNotADBTest failed while the others all ran successfully.  
  
The testRealEstateDAO.java file contains the following code:  
  
package myapp.re;  
  
import static org.hamcrest.CoreMatchers.hasItem;  
import static org.hamcrest.CoreMatchers.hasItems;  
import static org.junit.Assert.assertEquals;  
import static org.junit.Assert.assertNotNull;  
import static org.junit.Assert.assertNull;  
import static org.junit.Assert.assertThat;  
import static org.junit.Assert.fail;  
  
import java.util.List;  
  
import myapp.junit.categories.CRUDTest;  
import myapp.junit.categories.SearchTest;  
import myapp.realestate.Address;  
import myapp.realestate.RealEstate;  
import myapp.realestate.RealEstateDAO;  
  
import org.hibernate.Session;  
import org.junit.After;  
import org.junit.AfterClass;  
import org.junit.Before;  
import org.junit.BeforeClass;  
import org.junit.Test;  
import org.junit.experimental.categories.Category;  
  
public class TestRealEstateDAO  
{  
   private static ConfigurationFactory config;  
   private RealEstateDAO dao;  
   private RealEstate sample;  
     
   @BeforeClass  
   public static void setupHibernate()  
   {  
       config = new ConfigurationFactory();  
   }  
  
   @Before  
   public void setupDAO()  
   {  
       dao = new RealEstateDAO(config.getSessionFactory());  
   }  
  
   @Before  
   public void setupSample()  
   {  
       sample = createRealEstate();  
   }  
     
//  @After  
//  public void wipeTable()  
//  {  
//      Session session = config.getSessionFactory().getCurrentSession();  
//      session.beginTransaction();  
//      session.createSQLQuery("delete from RealEstate").executeUpdate();  
//      session.getTransaction().commit();  
//  }  
     
   @AfterClass  
   public static void closeDatabase()  
   {  
       config.getSessionFactory().close();  
   }  
     
   @Test  
   @Category(CRUDTest.class)  
   public void testAdd()  
   {  
       dao.add(sample);  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the one we just added", all, hasItem(sample));  
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testUpdate()  
   {  
       dao.add(sample);  
         
       double updatedValue = 200000;  
       sample.setValue(updatedValue);  
         
       dao.update(sample);  
         
       RealEstate updated = dao.load(sample.getId());  
       assertEquals("New value should be updated", updatedValue, updated.getValue(), 0.001);  
                 
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testLoadAndDelete()  
   {  
       dao.add(sample);  
       long saveId = sample.getId();  
         
       RealEstate added = dao.load(saveId);  
       assertNotNull("Should have loaded the one we just added", added);  
         
       dao.delete(sample);  
         
       RealEstate deleted = dao.load(saveId);  
       assertNull("It should not have been loaded", deleted);  
   }  
  
   @Test  
   @Category(SearchTest.class)  
   public void testLoadAll()  
   {  
       RealEstate r1 = createRealEstate();  
       RealEstate r2 = createRealEstate();  
       RealEstate r3 = createRealEstate();  
       dao.add(r1);  
       dao.add(r2);  
       dao.add(r3);  
  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the ones we just added", all, hasItems(r1, r2, r3));  
       assertEquals("We expect 3 items", 3, all.size());  
   }  
     
   @Test  
   public void randomNotADBTest()  
   {  
       fail("Not marked with a category, don't run me");  
   }  
     
   private RealEstate createRealEstate()  
   {  
       sample = new RealEstate();  
       sample.setValue(100000);  
       Address a = new Address();  
       a.setStreet("123 Main");  
       a.setCity("My Town");  
       a.setState("My State");  
       a.setZip("12345");  
       sample.setAddress(a);  
       return sample;  
   }  
}  
  
The presenter switches to the Command Prompt window where the MySQL 5.6 Command Line Client has been invoked. At the prompt, the presenter enters the command select \* from realestate;  
  
The output is Empty set <0.00 sec>.  
  
The presenter switches back to the Eclipse development environment and clicks Run to run the code in the TestRealEstateDAO.java file.  
  
In the JUnit view, the test progress details show 5/5 tests were run, two of which failed. The top-level test node is expanded and five methods are listed below it – testLoadAndDelete, testAdd, testLoadAll, randomNotADBTest, and testUpdate. The randowmNotADBTest and testLoadAll methods failed while the other three ran successfully.  
  
The presenter refers to the failed testLoadAll method in the JUnit view.  
  
In the TestRealEstateDAO.java file, the presenter refers to the three items that are added, loaded, and checked in the following testLoadAll method code:  
  
   {  
       RealEstate r1 = createRealEstate();  
       RealEstate r2 = createRealEstate();  
       RealEstate r3 = createRealEstate();  
       dao.add(r1);  
       dao.add(r2);  
       dao.add(r3);  
  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the ones we just added", all, hasItems(r1, r2, r3));  
       assertEquals("We expect 3 items", 3, all.size());  
   }  
  
The presenter switches to the MySQL Command Line Client window. At the prompt, the presenter enters the command select \* from realestate;  
  
A tab delimited table with six columns is returned as output. The column headers are id, street, city, state, zip, and value. The table includes five rows. Each row starts with a different ID – 61, 62, 63, 64, or 65 – and then specifies the street address 123 Main, the city entry "My Town", the state entry "My State", and the zip value 12345.  
  
The presenter switches back to the Eclipse development environment. In the editor, the presenter scrolls up in the TestRealEstateDAO.java file to reveal the following code:  
  
   public void testLoadAndDelete()  
   {  
       dao.add(sample);  
       long saveId = sample.getId();  
         
       RealEstate added = dao.load(saveId);  
       assertNotNull("Should have loaded the one we just added", added);  
         
       dao.delete(sample);  
         
       RealEstate deleted = dao.load(saveId);  
       assertNull("It should not have been loaded", deleted);  
  
The presenter highlights the following code line:  
  
       RealEstate deleted = dao.load(saveId);  
       assertNull("It should not have been loaded", deleted);*   
  
So, what we can do instead here is, we can go through and add in an extra method to clean up after ourselves. And that method is being commented out for now so we can show adding it back in. We're adding it what we call wipeTable. The wipeTable method is here to basically go out to the database, and delete everything out of that database. And the After annotation is what signals JUnit to say run this after each one of the test cases. So, now for each of the test cases that's going to execute, we're going to go through and wipe the table and that's when each next test case starts up, it should come into a state where each test should pass. Now, our load test knows there is nothing in the database ahead of time. And when it executes, we can assume that the three items I added are the only items out there, and I can check if the size is proper. So, it gives us that extra little test on top of just, hey were those items out there? Did we actually load them properly? Were they actually the only ones in the database? And if I go back and look at my database table again you can see it is indeed empty. So, the After annotation is something that allows us to do that, and we can have any number of After annotations here. I can go through and add another one outside of here, that just say -   
*The presenter scrolls further up in the TestRealEstateDAO.java file and highlights the following commented out wipeTable method:  
  
//  @After  
//  public void wipeTable()  
//  {  
//      Session session = config.getSessionFactory().getCurrentSession();  
//      session.beginTransaction();  
//      session.createSQLQuery("delete from RealEstate").executeUpdate();  
//      session.getTransaction().commit();  
//  }  
  
The presenter removes the commenting. The code for the wipeTable method, which is now  
  
 @After  
 public void wipeTable()  
 {  
     Session session = config.getSessionFactory().getCurrentSession();  
     session.beginTransaction();  
     session.createSQLQuery("delete from RealEstate").executeUpdate();  
     session.getTransaction().commit();  
 }  
  
The presenter highlights the @After annotation above the wipeTable method.  
  
The presenter then clicks the Run button.  
  
In the JUnit view, the test progress details show 5/5 tests were run, one of which failed. The top-level test node is expanded and five methods are listed below it: testLoadAndDelete, testAdd, testLoadAll, randomNotADBTest, and testUpdate. This time only the randowmNotADBTest test failed and the four others ran successfully.  
  
The presenter refers to the successful testLoadAll method in the JUnit view.  
  
In the editor, the presenter scrolls down in the TestRealEstateDAO.java file and highlights the last line of the following code for the testLoadAll method:  
  
   public void testLoadAll()  
   {  
       RealEstate r1 = createRealEstate();  
       RealEstate r2 = createRealEstate();  
       RealEstate r3 = createRealEstate();  
       dao.add(r1);  
       dao.add(r2);  
       dao.add(r3);  
  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the ones we just added", all, hasItems(r1, r2, r3));  
       assertEquals("We expect 3 items", 3, all.size());  
  
The presenter switches to the MySQL Command Line Client window. At the prompt, the presenter enters the command select \* from realestate;  
  
The output is Empty set <0.00 sec>.  
  
The presenter switches back to the Eclipse development environment. In the editor, the presenter scrolls up in the TestRealEstateDAO.java file and begins adding code between the setupSample and wipeTable methods.*   
  
sayHi. And so we can have a real simple here, print line outside of there that is going to run after every single test. Now, this test is highly involved and we see a lot of tests is going on here, there is lot of logins that's happening inside of here as you can see after each test we say, Hello. And it's got a extra nice little feature inside of here, so we can see clearly for each test when it's starting and stopping. And we can see these are the database command in each one of these tests. So, there is a little side benefit as well in our little test here. But the point being is in JUnit every time it sees an After annotation, it's going to run it and then go from there. It's going to run that in whatever order it chooses, it's not going to guarantee the order that they happen. In this case, the annotation is actually happening before everything is being deleted from the database. So, it's not something we can choose exactly on how it happens, but if we want to do that we would just put all the code in a single After annotation. This is saying, I have separate features and run them separately. So, using the After annotation allows us to clean up after our test cases, and particularly that we have an external data source. We want to keep that external data source clean, and it provides us a powerful way to be able to do that in consistent way, that it doesn't have to be defined within each test case.   
*In the TestRealEstateDAO.java file, the presenter completes the following code between the setupSample and wipeTable methods:  
  
   @After  
   public void sayHi()  
   {  
       System.out.println("Hello");  
   }  
  
The presenter clicks the Run button.  
  
In the JUnit view, the test progress details show 5/5 tests were run, one of which failed. The top-level test node is expanded and five methods are listed below it – testLoadAndDelete, testAdd, testLoadAll, randomNotADBTest, and testUpdate. The randowmNotADBTest test failed and the four others ran successfully.  
  
Below the file, the Console pane contains the test log data output. The presenter resizes the pane, which lists output such as  
  
Hello  
Hibernate: delete from RealEstate  
Hibernate: insert into RealEstate (value, street, city, state, zip) values (?, ?, ?, ?, ?)  
Hibernate: select this\_.id as id1\_0\_0\_, this\_.value as value2\_0\_0\_, this\_.street as street3\_0\_0\_, this\_.city as city  
Hello  
Hibernate: delete from RealEstate  
Hibernate: insert into RealEstate (value, street, city, state, zip) values (?, ?, ?, ?, ?)  
Hibernate: insert into RealEstate (value, street, city, state, zip) values (?, ?, ?, ?, ?)  
Hibernate: insert into RealEstate (value, street, city, state, zip) values (?, ?, ?, ?, ?)  
Hibernate: select this\_.id as id1\_0\_0\_, this\_.value as value2\_0\_0\_, this\_.street as street3\_0\_0\_, this\_.city as city  
Hello  
  
The presenter highlights the "Hello" output logged after each test.  
  
In the file, the presenter highlights the @After annotation above the newly added sayHi() method in the following code:  
  
   @After  
   public void sayHi()  
   {  
       System.out.println("Hello");  
   }  
  
In the Console pane, the presenter refers again to the logged "Hello" messages to show that, in each case, they are logged before the database is cleared by the wipeTable() method. A sample of the output in the Console pane, again, is  
  
Hello  
Hibernate: delete from RealEstate  
Hibernate: insert into RealEstate (value, street, city, state, zip) values (?, ?, ?, ?, ?)  
Hibernate: select this\_.id as id1\_0\_0\_, this\_.value as value2\_0\_0\_, this\_.street as street3\_0\_0\_, this\_.city as city  
Hello  
Hibernate: delete from RealEstate  
  
In the TestRealEstateDAO.java file, the presenter refers again to the separate @After annotations in the following code:  
  
   @After  
   public void sayHi()  
   {  
       System.out.println("Hello");  
   }  
     
   @After  
   public void wipeTable()  
   {  
       Session session = config.getSessionFactory().getCurrentSession();  
       session.beginTransaction();  
       session.createSQLQuery("delete from RealEstate").executeUpdate();  
       session.getTransaction().commit();  
   }*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Test Execution Ordering

Learning Objective

*After completing this topic, you should be able to*

* *configure execution ordering in JUnit tests*

**1. Configuring test execution ordering**

By default JUnit will execute test cases in whatever order it decides, which could vary each time the test is executed. Some test designs must assume a specific order for each test case, however. Let's take a look at how we can specify test order within JUnit. So in this test we're executing this where we want this to go a, b, c, d, e, f, g. It gives us a flow towards how we're explaining these different things inside of the tests. Now it does that because we have the FixMethodOrder annotation on the class. And the option side of there is saying we want to sort them by name. Now if I take that out, you can see it does it in random order. I mean A executes goes last, so it does f, d, b, e, g, c, a. Again there is no prediction inside of there. It could run differently each time; in this case it runs the same way each time. So this is a really important annotation. Now there's not a lot of options inside of there as far as method orders. I have the default and the JVM method order and neither of those really help you predict what's going to go on unless you have a very good understanding of how the JVM is working. And so really name ascending is the most common way to make this happen. Now, when we say it has to do with how tests are created, let's take an example of interacting with the database for a test.   
*In the Eclipse development environment, three files – AssertEqualsNumbers.java,   
OrderedTestRealEstateDAO.java, and TestRealEstateDAO.java – are open on separate tabbed pages in the editor.  
  
The AssertEqualsNumbers.java file contains the following code:  
  
package myapp.junit.asserts;  
  
import static org.junit.Assert.assertEquals;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class AssertEqualsNumbers  
{  
  
   @Test  
   public void a\_IntegerEqualsAsserts()  
   {  
       long longValue = 10;  
       assertEquals(longValue, longValue);  
       assertEquals("Two long values", longValue, longValue);  
         
       short shortValue = 10;  
       int intValue = 10;  
       assertEquals("Any two 'integer' values", intValue, shortValue):  
   }  
     
   @Test  
   public void b\_FloatingEqualAsserts()  
   {  
       double doubleValue = 0.3;  
       assertEquals("Two double values", doubleValue, doubleVallue, 0.00000001);  
   }  
         
   @Test  
   public void c\_DeprecatedEqualsAsserts()  
   {  
       double doubleValue = 0.3;  
       assertEquals("Two double values, no delta", doubleValue, doubleValue);  
   }  
     
   @Test  
   public void d\_DoubleDeltaEqualAsserts()  
   {  
       double doubleValue = 0.3;  
       double secondValue = 0.301;  
       assertEquals("Pass within delta", doubleValue, secondValue, 0.01);  
   }  
     
   @Test  
   public void e\_DoubleDeltaEqualAssertsFail()  
   {  
       double doubleValue = 0.3;  
       double secondValue = 0.301;  
       assertEquals("Failed within delta", doubleValue, secondValue, 0.01);  
   }     
     
   @Test  
   public void f\_NotEquals()  
   {  
       double doubleValue = 0.3;  
       double secondValue = 0.3000001;  
       assertNotEquals("These are not equal", doubleValue, secondValue);  
         
       long longValue = 10;  
       byte second = 9;  
       assertNotEquals("Integers too", longValue, second);  
   }  
     
   @Test  
   public void g\_FloatDeltaEqualAsserts()  
   {  
       float floatValue = 0.3f;  
       float secondValue = 0.301f;  
       assertEquals("Pass within delta", floatValue, secondValue, 0.01f);  
       assertEquals("Still Passes within delta of 0.001", floatValue, secondValue, 0.001f);  
       assertEquals("Failed within delta of 0.0001", floatValue, secondValue, 0.0001f);  
   }  
}  
  
The presenter refers to the seven test methods to be run in alphabetical ascending order according to their names.  
  
The presenter copies and then deletes the following line of code:  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
  
The presenter clicks the Run button.  
  
Only the editor is in focus and so the JUnit view test results and test suite tree are not visible.  
  
The presenter pastes the code @FixMethodOrder(MethodSorters.NAME\_ASCENDING) back into its original location and then deletes the NAME\_ASCENDING option within it.   
  
A drop-down list automatically suggests ways to complete the code after MethodSorters. It lists the following three options:  
  
DEFAULT: MethodSorters - MethodSorters  
JVM: MethodSorters - MethodSorters  
NAME\_ASCENDING: MethodSorters - MethodSorters  
  
The presenter chooses the NAME\_ASCENDING option to replace the original NAME\_ASCENDING option for the MethodSorters class.  
  
Next the presenter switches to the OrderedTestRealEstateDAO.java file.*   
  
So as we are going to test, we want to say we're going to test and add. And that's going to be our first test, we're going to add something to the database. And the next thing we want to test is testing an update. So after we've added an IM database we would test, hey, can I update it? Did it update properly in the database? After that I can load it and then delete it and see if this works. So I add something, I update something, then I delete it, and that step is going to give me each of my test cases. So I have very simple test cases, I have very simple steps. I don't want anything redundant within these test cases. I'm going to test one, then the other, then the other. This is probably what I'd do if I was using it from a gooey. So if I go through and execute this guy, it's going to work just fine. It does a, b and c. Without that, without the name ascending inside of here, who knows what's going to execute? It might not run any of these tests successfully. We'll have to say no, see with the b, c, a. And so b and c failed because there was nothing out there to work on. Now it gives us an indication that the tests are working because when I do an Update or LoadAndDelete, there was no added item out there. But it certainly breaks down my tests. So I can use the NAME\_ASCENDING inside of here to ensure the order of my tests and make sure everything runs right.   
*The OrderedTestRealEstateDAO.java file contains the following code:  
  
package myapp.re.ordering;  
  
import static org.hamcrest.CoreMatchers.hasItem;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class OrderedTestRealEstateDAO  
{  
   private static ConfigurationFactory config;  
   private static RealEstate sample;  
   private RealEstateDAO dao;  
     
   @BeforeClass  
   public static void setupHibernate()  
   {  
       config = new ConfigurationFactory();  
   }  
  
   @Before  
   public void setupDAO()  
   {  
       dao = new RealEstateDAO(config.getSessionFactory());  
   }  
  
   @AfterClass  
   public static void closeDatabase()  
   {  
       Session session = config.getSessionFactory().getCurrentSession();  
       session.beginTransaction();  
       session.createSQLQuery("delete from RealEstate").executeUpdate();  
       session.getTransaction().commit();  
       config.getSessionFactory().close();  
   }  
     
   @Test  
   @Category(CRUDTest.class)  
   public void a\_testAdd()  
   {  
       sample = createRealEstate();  
       dao.add(sample);  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the one we just added", all, hasItem(sample));  
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void b\_testUpdate()  
   {  
       double updatedValue = 200000;  
       sample.setValue(updatedValue);  
         
       dao.update(sample);  
         
       RealEstate updated = dao.load(sample.getId());  
       assertEquals("New value should be updated", updatedValue, updated.getValue(), 0.001);  
                 
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void c\_testLoadAndDelete()  
   {  
       long saveId = sample.getId();  
       RealEstate added = dao.load(saveId);  
       assertNotNull("Should have loaded the one we just added", added);  
         
       dao.delete(sample);  
         
       RealEstate deleted = dao.load(saveId);  
       assertNull("It should not have been loaded", deleted);  
   }  
     
   private RealEstate createRealEstate()  
   {  
       sample = new RealEstate();  
       sample.setValue(100000);  
       Address a = new Address();  
       a.setStreet("123 Main");  
       a.setCity("My Town");  
       a.setState("My State");  
       a.setZip("12345");  
       sample.setAddress(a);  
       return sample;  
   }  
}  
  
The the presenter highlights the three test methods – a\_testAdd(), b\_testUpdate(), and   
c\_testLoadAndDelete().  
  
The presenter then clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.re.ordering.OrderedTestRealEstateDAO and its runtime is 0.255 seconds. The test progress details show 3/3 tests were run   
successfully.  
  
The presenter expands the top-level test node to reveal the following list of successful test methods:  
  
a\_testAdd (0.181 s)  
b\_testUpdate (0.036 s)  
c\_testLoadAndDelete (0.038 s)  
  
In the OrderedTestRealEstateDAO.java file, the presenter copies and then deletes the following line of code:  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
  
The presenter clicks the Run button. Only the editor is in focus and so the JUnit view test results and test suite tree are not visible.  
  
He pastes the code @FixMethodOrder(MethodSorters.NAME\_ASCENDING) back in its original location.*   
  
Now, it's not to say this is the right design. There are test cases where this makes sense. We also can test where we do everything inside of one. So it's probably a better test, it's probably easier to debug when each test case is self-contained. So in this case we see an alternative version. We have an update where we add an item, we update the item, and go from there. Now, the difference here is in this test case I have an After that's wiping the table every single time. So I can add as many items as I want to and I can wipe it afterwards. But it gives you options in your design. JUnit lets you either self-contain each test using the test itself or the before and after options, or it allows you to say, "Hey, I really want these tests to be executed one, two, three, four, five." Now you can't really do it one, two, three, four, five; you have to do its a, b, c as it is inside of here because Java won't allow a method to start with the numbers one. But you could just do a one, a two, a three, a four, a five; however you want to design that, but just keep that as an option. The inside of your test design, if you know you want test to execute in a specific order, you can use this without having to create big long extended tests. And have each step judged on its own as pass or fail, even still. And then not have to have everything as part of each test case. So use that, it's a great effect.   
*The presenter switches to the TestRealEstateDAO.java file, in which the following code is visible:  
  
public void testAdd()  
   {  
       dao.add(sample);  
       List<RealEstate> all = dao.loadAll();  
       assertThat("All should contain the one we just added", all, hasItem(sample));  
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testUpdate()  
   {  
       dao.add(sample);  
         
       double updatedValue = 200000;  
       sample.setValue(updatedValue);  
         
       dao.update(sample);  
         
       RealEstate updated = dao.load(sample.getId());  
       assertEquals("New value should be updated", updatedValue, updated.getValue(), 0.001);  
                 
   }  
  
   @Test  
   @Category(CRUDTest.class)  
   public void testLoadAndDelete()  
  
The presenter highlights the testUpdate() method and runs through its self-contained add and update code.  
  
The presenter scrolls up in the TestRealEstateDAO.java file and highlights the wipeTable() method in the following segment of code:  
  
@After  
   public void wipeTable()  
   {  
       Session session = config.getSessionFactory().getCurrentSession();  
       session.beginTransaction();  
       session.createSQLQuery("delete from RealEstate").executeUpdate();  
       session.getTransaction().commit();  
   }  
  
The presenter switches back to the OrderedTestRealEstateDAO.java file. As an example of the naming convention used   
for its test methods, he highlights the method c\_testLoadAndDelete().*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Enhancing Tests with Rules

Learning Objective

*After completing this topic, you should be able to*

* *enhance JUnit tests with rules*

**1. Using rules to enhance tests**

Rules provides us an extension to give us universal checks across all test cases within a JUnit test, or further details and control within each test. It gives us more granularity and control to really create customized testing, beyond what is provided in basic JUnit. We're going to look at a couple different rules here. And the first thing we can point out is the rule starts off as an attribute. We have this…I'm sorry – we have this attribute here, with an annotation inside of it, that defines the rule as being active throughout this whole test. And here's a universalTimeout inside of here. So we can do a timeout on each individual test – we can say each test lasts a certain amount of time. Or we can provide a rule that says all these tests have to take less than a second. And so when we execute this test, we can see it's going to go through – and each test basically is sitting around for a second – and depending on the test that's being executed – some of them pass, or not…if it…depending on the time that that lasts, you know, how long it's going to go. And so, these are actually out of order from what we see here, but Test 1 – 9 tenths of a second – it's fine. Test 2 actually goes just over a second, so we have a little bit of flexibility inside of there, but that probably includes the work being done by the timeout. So it probably did come in at just under a second, plus had a little stuff that that was happening on the side. Test 3 then, right here, is exactly a second – so it's still…it's pretty accurate inside of here, maybe possibly more accurate than the timeout of the overall rule itself. And then Test 4 is obviously passed it, and it fails; it kills it off by the time it gets just past a second.   
*In the Eclipse development environment, four files – UniversalTimeout.java, ExpectedExceptionsExample.java, ErrorCollectorExample.java, and VerifierExample – are open on separate tabbed pages.  
  
The UniversalTimeout.java file contains the following code:  
  
package myapp.junit.rules;  
  
import org.junit.Rule;  
  
public class UniversalTimeout  
{  
   @Rule  
   public TestRule universalTimeout = new Timeout(1000);  
     
   @Test  
   public void delayOne() throws InterruptedException  
   {  
       Thread.sleep(900);  
   }  
  
   @Test  
   public void delayTwo() throws InterruptedException  
   {  
       Thread.sleep(999);  
   }  
  
   @Test  
   public void delayThree() throws InterruptedException  
   {  
       Thread.sleep(1000);  
   }  
  
   @Test  
   public void delayFour() throws InterruptedException  
   {  
       Thread.sleep(1100);  
   }  
}  
  
The presenter highlights the @Rule annotation and the universalTimeout attribute in the following segment of code:  
  
   @Rule  
   public TestRule universalTimeout = new Timeout(1000);  
  
The presenter then clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.rules.UniversalTimeout and its runtime is 3.909 seconds.The top-level test node is expanded and the following test methods and associated runtimes are listed below it –  delayThree (1.002 s), delayFour (1.005 s), delayOne (0.900 s), and delayTwo (1.001 s). The delayFour test method returned an error while the other three ran successfully.  
  
The presenter compares the order of the four test methods listed in the JUnit view with their order in the UniversalTimeout.java file in the editor.  
  
The presenter selects the delayOne test method in the JUnit view and compares its runtime of 0.900 seconds with the Thread.sleep(900) variable defined for the correlating method in the UniversalTimeout.java file.  
  
The presenter selects the delayTwo test method in the JUnit view and compares its runtime of 1.001 seconds with the Thread.sleep(999) variable defined for the correlating method in the UniversalTimeout.java file.  
  
In the editor, the presenter scrolls down the UniversalTimeout.java file and refers to the Thread.sleep(1000) variable defined for the delayThree test method.  
  
The presenter scrolls further down the file and refers to the Thread.sleep(1100) variable defined for the delayFour test method.  
  
He selects the delayFour test method in the JUnit view and the a java.lang.Eception error message appears in the Failure Trace section, stating that the test timed out after 1000 milliseconds.*   
  
So I wouldn't trust these times too directly, but it just kind of gives you a sense for the granularity within that. But this rule capability, more importantly, allows us to specify for all the test the rules that's going to go on here. If I change this to be 3 quarters of a second, or something like that, then all the test would be adjusted appropriately. I don't have to do it individually multiple times. Now, this also works to help us customize exception handling inside of there. So we have a rule – an annotation we can define for an expected exception. It's a different type of rule class that we create as a parameter – as a property to this test. So inside of here, we have a test that nothing interesting happens, we have a test that we do something, but it doesn't get a problem unless there is an ArithmeticException. And inside the ArithmeticException here, you can see when that happens, we're expecting it to happen and this will no longer fail. So let's go ahead and run this guy real quick. As you can see, all my tests pass, even though we could end up with an error inside of here. So inside of the last one then, you're expecting a NumberFormatException. And so this guy we get a divide by 0, this guy we get a "hey this is not a number", but in both cases we're expecting the exception. We have their expected exception inside of here. And even further here, we can expect that the message is going to contain some specific value. We're saying hey, we're expecting this should say this, and this is what the exception is going to go on. So beyond expecting the exception, this rule allows us to check the details of that exception.   
*The presenter points out the four test method runtimes listed in the JUnit view.  
  
In the UniversalTimeout.java file, the presenter refers to the following line of code:  
  
   public TestRule universalTimeout = new Timeout(1000);  
  
The presenter then switches to the ExpectedExceptionsExample.java file. It contains the following code:  
  
package myapp.junit.rules;  
  
import static org.hamcrest.CoreMatchers.containsString;  
  
public class ExpectedExceptionsExample  
{  
   @Rule  
   public ExpectedException ex = ExpectedException.none();  
     
   @Test  
   public void nothingInteresting()  
   {  
         
   }  
     
   @Test  
   public void something()  
   {  
       ex.expect(ArithmeticException.class);  
       int answer = 10 / Integer.parseInt("0");  
   }  
  
   @Test  
   public void detailedSomething()  
   {  
       ex.expect(NumberFormatException.class);  
       ex.expectMessage("abc"); // any string value contained within  
       ex.expectMessage(containsString("abc")); // same thing, but we can use any matcher  
       int answer = 10 / Integer.parseInt("abc");  
   }  
}  
  
The presenter refers to the @Rule annotation and ExpectedException class in the following segment of code:  
  
   @Rule  
   public ExpectedException ex = ExpectedException.none();  
  
The presenter hovers the cursor over the ExpectedException class and a tooltip appears with the following visible information:  
  
The ExpectedException rule allows in-test specification of expected exception types and messages:  
  
org.junit.rules.ExpectedException  
  
 // These tests all pass.  
 public static class HasExpectedException  {  
        @Rule  
        public ExpectedException thrown= ExpectedException.none();  
   
In the file, the presenter highlights the following code:  
  
   @Test  
   public void nothingInteresting()  
   {  
         
   }           
  
Next the presenter highlights the following code:  
  
   @Test  
   public void something()  
   {  
       ex.expect(ArithmeticException.class);  
       int answer = 10 / Integer.parseInt("0");  
   }  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.rules.ExpectedExceptionsExample and its runtime is 0.009 seconds. The presenter expands the top-level test node to reveal the following three test methods and their runtimes – something (0.006 s), nothingInteresting (0.000 s), and detailedSomething (0.003 s). All three methods ran successfully.  
  
In the ExpectedExceptionsExample.java file, the presenter compares and summarizes the different expected exceptions for the something and detailedsomething test methods in the following code:  
  
   @Test  
   public void something()  
   {  
       ex.expect(ArithmeticException.class);  
       int answer = 10 / Integer.parseInt("0");  
   }  
  
   @Test  
   public void detailedSomething()  
   {  
       ex.expect(NumberFormatException.class);  
       ex.expectMessage("abc"); // any string value contained within  
       ex.expectMessage(containsString("abc")); // same thing, but we can use any matcher  
       int answer = 10 / Integer.parseInt("abc");  
   }*   
  
Was it exactly the exception that we're expecting, and did the message come back saying exactly what we expected. And so the rule gives us even more granularity and control. Here's an example of getting around a feature inside of checking that, basically, when your first assert fails, your test stops. If I want to get multiple errors to be caught inside of here, I can use this error collector rule. And so, within this, I just do a checkThat. It's very similar to an assertThat. And now when I execute this test, you can see it fails, but I can capture all the different failures – so it's both warm and dry out there. Where, in a test – say you're doing web services, and you have multiple data items coming in, there could be multiple failures for a single request. We don't want to fail at the stop one…first one, we want to tell you all the different failures that this thing might have failed at. It forgot to check the first parameter, and the third parameter, and the seventh parameter – whatever it might be. This is a great way to do be able to do multiple tests.The last one we'll talk about here is the Verifier. So the Verifier is saying you do your test, everything happened the way it was supposed to happen, -   
*In the ExpectedExceptionsExample.java file, the presenter hovers over the expectMessage in the following segment of code:  
  
   @Test  
   public void detailedSomething()  
   {  
       ex.expect(NumberFormatException.class);  
       ex.expectMessage("abc"); // any string value contained within  
       ex.expectMessage(containsString("abc")); // same thing, but we can use any matcher  
       int answer = 10 / Integer.parseInt("abc");  
   }  
  
A tooltip displays the following information:  
  
void org.junit.rules.ExpectedException.expectMessage(String substring)  
  
expectMessage  
   
public void expectMessage (String substring)  
  
The presenter switches to the ErrorCollectorExample.java file. It contains the following code:  
  
package myapp.junit.rules;  
  
import static org.hamcrest.CoreMatchers.equalTo;  
  
public class ErrorCollectorExample  
{  
   @Rule  
   public ErrorCollector errors = new ErrorCollector();  
     
   private Weather weather;  
     
   @Before  
   public void setup()  
   {  
       weather = new Weather(8);  
   }  
     
   @Test  
   public void manyErrors()  
   {  
       try  
       {  
           errors.checkThat("It is dry", false, equalTo(weather.willItRainToday()));  
           errors.checkThat("It is warm", false, equalTo(weather.isItFreezingToday()));  
           errors.checkThat("It is sunny", true, equalTo(weather.isItSunnyToday()));  
       } catch (Throwable t)  
       {  
           errors.addError(t);  
       }  
   }  
}  
  
The presenter refers to the ErrorCollector rule defined in the following code:  
  
   @Rule  
   public ErrorCollector errors = new ErrorCollector();  
  
Next the presenter refers to the following segment of code:  
  
           errors.checkThat("It is dry", false, equalTo(weather.willItRainToday()));  
           errors.checkThat("It is warm", false, equalTo(weather.isItFreezingToday()));  
           errors.checkThat("It is sunny", true, equalTo(weather.isItSunnyToday()));  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.rules.ErrorCollectorExample and its runtime is 0.005 seconds. The top-level test node is expanded and the following test method and associated runtime is listed below it – manyErrors (0.005 s).  
  
In the JUnit view, the presenter selects the manyErrors method. In the Failure Trace section, the following failure information is visible:  
  
java.lang.AssertionError" It is dry  
Expected: <true>  
  but: was <false>  
at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)  
  
The presenter switches to the VerifierExample.java file. It contains the following code:  
  
package myapp.junit.rules;  
  
import static org.junit.Assert.fail;  
  
public class VerifierExample  
{  
   @Rule  
   public Verifier verify = new Verifier()   
   {  
       @Override  
       protected void verify() throws Throwable  
       {  
           System.out.println("Hmmm");  
           if (true)  
           {  
               fail("I felt like it");  
           }  
       }  
   };  
     
   @Before  
   public void before()  
   {  
       System.out.println("before");  
   }  
     
   @Test  
   public void testSomething()  
   {  
       System.out.println("Test");  
   }  
  
   @After  
   public void after()  
   {  
       System.out.println("after");  
   }  
}*   
  
but let's do one final check after the test is all said and done to say hey, was this test valid? And so the Verifier object inside of here – another rule – is something that I implement my own copy of. I'm doing it as an in line anonymous class – an in line anonymous class, right here. And so, the Verify method – I do whatever logic I want inside of here, and if this guy turns out not to be valid, I can fail the test. That's basically what's going on inside of here. And so if I run this one here, you can see my test executes. I have my before that executes, I have my test that executes, I have after the test that executes – all that happens as it normally would – and then I run my verifier after it's all said and done. I'm checking whatever; I may be checking the state of the database, I may be checking if a file is out there, I might be doing whatever that needs to be done outside of there. And then this will pass or fail, or otherwise adjust the system based off of each test case that runs. So you could see, rules give us a lot of flexibility and a lot of options to expand our JUnit tests even further. So use them to really make some powerful tests.   
*The presenter highlights the verifier object and runs through the Verify() method logic in the following segment of code:  
  
   @Rule  
   public Verifier verify = new Verifier()   
   {  
       @Override  
       protected void verify() throws Throwable  
       {  
           System.out.println("Hmmm");  
           if (true)  
           {  
               fail("I felt like it");  
           }  
       }  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.rules.VerifierExample and its runtime is 0.000 seconds. The top-level test node is expanded and the following failed test method and associated runtime is listed below it – testSomething (0.000 s). The testSomething method is selected and the following failure information is visible in the Failure Trace section:  
  
java.lang.AssertionError:I felt like it  
at myapp.junit.rules.VerifierExampleS1.verify(VerifierExample.java:22)  
  
Below the editor, the Console pane lists output. The presenter highlights the following lines in the output:   
  
before  
Test  
after  
Hmmm  
  
Finally, the presenter refers again to the verify() method in the following segment of code:  
  
       @Override  
       protected void verify() throws Throwable  
       {  
           System.out.println("Hmmm");  
           if (true)  
           {  
               fail("I felt like it");  
           }  
       }*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Rules for Managing Tests

Learning Objective

*After completing this topic, you should be able to*

* *use rules to manage JUnit tests*

**1. Using rules to manage tests**

An additional use of JUnit rules is to manage external test data and resources across all tests. While the test fixture capability provides many good options, using rules gives us a reusable solution to managing files, database connections, or any customized solution you wish to create. So the first example of a rule and action we can see is the use of temporary folders. This is provided for us as a very common rule you might want to use – as building your classes and your tests. So the temporary folder out there is a rule – you see the annotation for the rule. And then we create a public attribute to create the test temporary folder. This is important – it has to be public inside of here for this rule; it has to work that way. So inside of our test, we're going to create, and we're going to write data to a file. And so the file we're going to create, we're going to create out of that temporary directory. This saves us of having to do the work of getting a exact directory running in, creating it, and cleaning up the file afterwards. We don't have to delete it, we can just put it in this temporary directory, in the temporary folder solution for the rule, we'll create it, give us a space, and then delete it when we're all set and done. It'll clean up after itself. So we write whatever we're going to write out to this file, and we're going to make that happen. So when we go and execute this guy, you can see the location it's writing to is in this local temp folder outside of there. The JUnit creates its own test run of this and it puts it outside of this extra folder, and then this guy won't exist afterwards. If I went and looked at my machine, this folder will be gone; it's a local temp folder and it gets cleaned up.   
*In the Eclipse development environment, four files are open on separate tabbed pages – TempFolderRuleTest.java,   
ExternalResourceRule.java, ReusableExternalResourceRule.java, and ReusableConfig.java.  
  
The TempFolderRuleTest.java file contains the following code:  
  
package myapp.junit.rules.resource;  
  
import static org.junit.Assert.assertTrue;  
  
public class TempFolderRuleTest  
{  
   @Rule  
   public TemporaryFolder temp = new TemporaryFolder();  
   // Must be public!  
     
   @Test  
   public void createDataFile() throws IOException  
   {  
       WriteMyData writer = new WriteMyData();  
       File newFile = temp.newFile("myFile.txt");  
       System.out.println(newFile.getCanonicalPath());  
       writer.writeData(newFile);  
       assertTrue("It should be there now", newFile.exists());  
   }  
}  
  
The presenter highlights the @Rule annotation and then the following line of code below it:  
  
   public TemporaryFolder temp = new TemporaryFolder();  
  
Next the presenter highlights the following two lines of code within the @Test annotation:  
  
       WriteMyData writer = new WriteMyData();  
       File newFile = temp.newFile("myFile.txt");  
  
The presenter then clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.rules.resource.TempFolderRuleTest. The test progress details show that 1/1 tests were run successfully.*   
  
And so the great thing about this is it gives me, regardless of what platform I'm on – whether I'm on Windows or Unix or Linux or Mac or whatever – or whatever the configuration of that machine, the permissions is, JUnit can be configured to manage those files and run all that stuff for me. Now, I only work somewhat with files. More likely I'm going to be working with other external resources – for instance, the database. And as I said, we've seen the ability to deal with this within a test fixture. I can create it using before and after, and before class and after class, and such like that, but I can also do it as a rule and an external resource. And so the rule in external resources allows us to plug-in to before and after the same way. And so I can create a connection to my database – in this case, it's usually an example of Hibernate – and it will create and then set that guy up before and after, you know, before it's going to create it, and after it's going to close it. And so when we do our test, we can do a simple test to hit that database and do something interesting. It's going to be able to manage that connection all wrapped around it. Now, we say reusable – this isn't exactly reusable, I can copy and paste this code, but I could do that with a test fixture as well.   
*The presenter switches to the ExternalResourceRule.java file. It contains the following code:  
  
package myapp.junit.rules.resource;  
  
import myapp.re.ConfigurationFactory;  
  
public class ExternalResourceRule  
{  
   @Rule  
   public ExternalResource ext = new ExternalResource()   
   {  
  
       @Override  
       protected void after()  
       {  
           ConfigurationFactory.getStaticSessionFactory().close();  
       }  
  
       @Override  
       protected void before() throws Throwable  
       {  
           ConfigurationFactory.createStatic();  
       }  
   };  
     
   @Test  
   public void testSomething()  
   {  
       SessionFactory sf = ConfigurationFactory.getStaticSessionFactory();  
       Session s = sf.getCurrentSession();  
       s.beginTransaction();  
       s.createCriteria(RealEstate.class).list();  
       s.getTransaction().commit();  
   }  
}  
  
The presenter highlights the following line of code directly below the @Rule annotation:  
  
   public ExternalResource ext = new ExternalResource()  
  
The presenter highlights the before() and after() variables and runs through the related code for creating and closing the connection to the database in the following segment of code:  
  
       @Override  
       protected void after()  
       {  
           ConfigurationFactory.getStaticSessionFactory().close();  
       }  
  
       @Override  
       protected void before() throws Throwable  
       {  
           ConfigurationFactory.createStatic();  
       }  
  
The presenter refers to the following segment of code:  
  
@Test  
   public void testSomething()  
   {  
       SessionFactory sf = ConfigurationFactory.getStaticSessionFactory();  
       Session s = sf.getCurrentSession();  
       s.beginTransaction();  
       s.createCriteria(RealEstate.class).list();  
       s.getTransaction().commit();  
   }  
  
The presenter then clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.rules.ExternalResourceRule. The presenter expands the top-level test node to reveal a single test method, testSomething. The testSomething method ran successfully with a runtime of 1.071 seconds.  
  
In the ExternalResourceRule.java file, the presenter refers to the following segment of code:  
  
   @Rule  
   public ExternalResource ext = new ExternalResource()   
   {  
  
       @Override  
       protected void after()  
       {  
           ConfigurationFactory.getStaticSessionFactory().close();  
       }  
  
       @Override  
       protected void before() throws Throwable  
       {  
           ConfigurationFactory.createStatic();  
       }  
   };*   
  
So what if I know I'm going to have a bunch of tests, and I want them to be reuseably using this solution? Well I can extend out that reusable…that external config into a reusable external config. So instead of the rule that uses the built in default external resource that's out there, I can create my own – and that looks like this. I create my own class, ReusableConfig, that extends out that external resource. And then I do my before and after – the exact same code we had here, I'm just putting into a class all by itself. And now every single time I have a test that wants to deal with this external resource – in this case, the database – we reuse that config. Its before and its after, I've no additional configuration. And I don't have to put the before in, I don't have to put the afters in, or all the other steps that's out there, and then it's all setup and ready to go. And this is setup and reused across all of them, it's all the tests reuse the same rule setup. It's automatically static within there, as well. So now I execute it, and it does all the same stuff we had before, but it's doing it in a reusable way. So using rules gives us a great way to make a component out of that external resource, to basically reuse code and not copy and paste. And all the great design things we want to do in building our solution, rules allows us to do with building our test cases as well.   
*The presenter switches to the ReusableExternalResourceRule.java file. It contains the following code:  
  
package myapp.junit.rules.resource;  
  
import myapp.re.ConfigurationFactory;  
  
public class ReusableExternalResourceRule  
{  
   @Rule  
   public ResuableConfig ext = new ResuableConfig();    
     
   @Test  
   public void testSomething()  
   {  
       SessionFactory sf = ConfigurationFactory.getStaticSessionFactory();  
       Session s = sf.getCurrentSession();  
       s.beginTransaction();  
       s.createCriteria(RealEstate.class).list();  
       s.getTransaction().commit();  
   }  
}  
  
The presenter refers to the ReusableConfig() variable within the following segment of code:  
  
public class ReusableExternalResourceRule  
{  
   @Rule  
   public ResuableConfig ext = new ResuableConfig();   
  
The presenter clicks the ExternalResourceRule.java tab. On the ExternalResourceRule.java tabbed page, the presenter highlights the code ExternalResource in the following line:  
  
public ExternalResource ext = new ExternalResource()   
  
Next the presenter switches to the ReusableConfig.java file. It contains the following code:  
  
package myapp.junit.rules.resource;  
  
import myapp.re.ConfigurationFactory;  
  
import org.junit.rules.ExternalResource;  
  
public class ResuableConfig extends ExternalResource  
{  
   @Override  
   protected void after()  
   {  
       ConfigurationFactory.getStaticSessionFactory().close();  
   }  
  
   @Override  
   protected void before() throws Throwable  
   {  
       ConfigurationFactory.createStatic();  
   }  
}  
  
The presenter highlights the following line of code:  
  
public class ResuableConfig extends ExternalResource  
  
The presenter refers to the before() and after() variables in the following segment of code:  
  
   @Override  
   protected void after()  
   {  
       ConfigurationFactory.getStaticSessionFactory().close();  
   }  
  
   @Override  
   protected void before() throws Throwable  
   {  
       ConfigurationFactory.createStatic();  
   }  
  
The presenter switches to the ExternalResourceRule.java file. He refers again to the following segment of code:  
  
       @Override  
       protected void after()  
       {  
           ConfigurationFactory.getStaticSessionFactory().close();  
       }  
  
       @Override  
       protected void before() throws Throwable  
       {  
           ConfigurationFactory.createStatic();  
       }  
  
The presenter returns to the ReusableConfig.java file. He refers to the following segment of code:  
  
public class ResuableConfig extends ExternalResource  
{  
   @Override  
   protected void after()  
   {  
       ConfigurationFactory.getStaticSessionFactory().close();  
   }  
  
   @Override  
   protected void before() throws Throwable  
   {  
       ConfigurationFactory.createStatic();  
   }  
}  
  
The presenter switches to the ReusableExternalResourceRule.java file. It contains the following code:  
  
package myapp.junit.rules.resource;  
  
import myapp.re.ConfigurationFactory;  
  
public class ReusableExternalResourceRule  
{  
   @Rule  
   public ResuableConfig ext = new ResuableConfig();    
     
   @Test  
   public void testSomething()  
   {  
       SessionFactory sf = ConfigurationFactory.getStaticSessionFactory();  
       Session s = sf.getCurrentSession();  
       s.beginTransaction();  
       s.createCriteria(RealEstate.class).list();  
       s.getTransaction().commit();  
   }  
}  
  
The presenter refers to the following line of code directly below the @Rule annotation:  
  
   public ResuableConfig ext = new ResuableConfig();    
  
The presenter highlights the following segment of code below the testSomething() variable:  
  
   {  
       SessionFactory sf = ConfigurationFactory.getStaticSessionFactory();  
       Session s = sf.getCurrentSession();  
       s.beginTransaction();  
       s.createCriteria(RealEstate.class).list();  
       s.getTransaction().commit();  
   }  
  
The presenter then clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.rules.resource.ReusableExternalResourceRule and its runtime is 1.069 seconds. The presenter expands the top-level test node to reveal a single test method, testSomething. The testSomething method ran successfully with a runtime of  1.069 seconds.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Theories

Learning Objective

*After completing this topic, you should be able to*

* *work with JUnit theories*

**1. Working with JUnit theories**

Test data generation can be tedious and time consuming. JUnit uses theories to simplify the process by allowing for rapid test data generation and clear mechanisms for externalizing data. It even gives us the ability to check that the data is valid thus reducing false failures. Let's take a look. So a very simple class uses the RunWith on the theories. We have to use that special runner to get this going, but it allows us to within the test parameterize the test. And we're saying we are going to pass in a String and an integer. And so we're going to get a String out of this array of strings, we will get an integer out of this array of integers and it's defined by our DataPoints. And so the DataPoints are saying "These are all the access of data items that's going to be passed inside of here." And so when I execute this test I'm going to go through and say, "Hey, these are the values I'm handed and I'm going to filter out any data, the data items that are invalid using the assumeThat option." This says "Hey, if this one's not valid, skip it, don't use it." So if I'm saying to you mix oil and water and they don't mix and your test— instead of failing a test you just say "Hey, skip this test, it doesn't makes sense, oil and water doesn't work." And so let's look at how this guy behaves when it goes off and runs. So as it executes, you can see I get alligator 1, 2, 3. So I have all of the options against alligator. Then I get bear 1, 2, 3. And then I get camel 1, 2, 3.   
*In the Eclipse development environment, four files are open on separate tabbed pages – TestTheory.java, TheoryWithSuppplier.java, SomeData.java, and TestDataSupplier.java.  
  
The TestTheory.java file contains the following code:  
  
package myapp.junit.theory;  
  
import static org.hamcrest.CoreMatchers.not;  
  
@RunWith(Theories.class)  
public class TestTheory  
{  
   @DataPoints  
   public static String[]  animals = {"aligator", "bear", "camel"};  
  
   @DataPoints  
   public static Integer[] integers = {1, 2, 3};  
  
   @Theory  
   public void someTest(String x, Integer y)  
   {  
       System.out.println(x + " " + y);  
       assumeThat(x, not(startsWith("a"))); // filters out any unwanted data  
       System.out.println("continue " + x + " " + y);  
   }  
}  
  
The presenter highlights the code @RunWith(Theories.class)  
  
The presenter highlights (String x, Integer y) in the following segment of code:  
  
   @Theory  
   public void someTest(String x, Integer y)  
  
Next presenter highlights the string array in the following segment code:  
  
   @DataPoints  
   public static String[]  animals = {"aligator", "bear", "camel"};  
  
The presenter highlights the integer array in the following segment of code:  
  
   @DataPoints  
   public static Integer[] integers = {1, 2, 3};  
  
He highlights the @DataPoints annotation.  
  
Below the @Theory annotation, the presenter runs through the logic of the first two lines in the following segment of code:  
  
   {  
       System.out.println(x + " " + y);  
       assumeThat(x, not(startsWith("a"))); // filters out any unwanted data  
       System.out.println("continue " + x + " " + y);  
   }  
  
The presenter then clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.theory.TestTheory and its runtime is 0.021 seconds. The top-level test node is expanded and a single test method and its associated runtime is listed below it – someTest (0.021 s). The someTest ran successfully.  
  
The presenter refers to the following test run output information in the Console pane below the editor:  
  
aligator 1  
aligator 2  
aligator 3  
bear 1  
continue bear 1  
bear 2  
continue bear 2  
bear 3  
continue bear 3  
camel 1  
continue camel 1  
camel 2  
continue camel 2  
camel 3  
continue camel 3*   
  
So it's going to run every data item in the DataPoint set with every other DataPoint set inside here. So we end up with nine runs out of these two. It's not doing alligator one, bear two, camel three; it's doing all of alligator one, two, three; bear one, two, three; camel one, two, three. And with our assumeThat we're saying, "Hey, if the animal starts with an A, in this case alligator, we're skipping it." You can see continue never executes for alligator, but it does for bear and camel. So the theories again lets us throw some data against the wall and see what pops up. Now this doesn't give us the expected value, it just is saying "Here's a bunch of data to go and execute." But it does give us a little bit of data to be going on with. We'd have to pass in the expected value in a different way or solve for the expected value, or something like that, that would see inside of our solution. So we can also externalize that data. If all of our data had to come from strings typed in, it wouldn't be all that much of a simplification over having to do it other ways. But, if we can externalize that data and say load it from a file, or load it from a spreadsheet, or load it from a database we can all of a sudden get tons and tons of test data to be able to spit at a system. So in this case we have two tests. We have a test for something and a test for something else. This was just printing something out, this was doing our oldest search. It's going to fail tests of data, it doesn't conform to things along the way. One should be able to see that.   
*In the TestTheory.java file, the presenter refers to the following segment of code:  
  
   @DataPoints  
   public static String[]  animals = {"aligator", "bear", "camel"};  
  
   @DataPoints  
   public static Integer[] integers = {1, 2, 3};  
  
Below the @Theory annotation, the presenter highlights the following line of code:  
  
       assumeThat(x, not(startsWith("a"))); // filters out any unwanted data  
  
The presenter refers to the following output in the Console pane:  
  
aligator 1  
aligator 2  
aligator 3  
  
Below the @Theory annotation, the presenter highlights the following line of code:    
  
       System.out.println("continue " + x + " " + y);  
  
The presenter refers again to the following output in the Console pane:  
  
aligator 1  
aligator 2  
aligator 3  
bear 1  
continue bear 1  
bear 2  
continue bear 2  
bear 3  
continue bear 3  
camel 1  
continue camel 1  
camel 2  
continue camel 2  
camel 3  
continue camel 3  
  
In the TestTheory.java file, the presenter refers again to the following segment of code:  
  
   @DataPoints  
   public static String[]  animals = {"aligator", "bear", "camel"};  
  
   @DataPoints  
   public static Integer[] integers = {1, 2, 3};  
  
The presenter switches to the TheoryWithSuppplier.java file. It contains the following code:  
  
package myapp.junit.theory;  
  
import static org.junit.Assert.assertTrue;  
  
@RunWith(Theories.class)  
public class TheoryWithSuppplier  
{  
   @Theory  
   public void testSomething(@SomeData int x)  
   {  
       System.out.println(x + " Something");  
       assertTrue(x < 3);  
   }  
  
  
   @Theory  
   public void testSomethingElse(@SomeData int x)  
   {  
       System.out.println(x + " Something else");  
   }  
}  
  
The presenter refers to the testSomething(@SomeData int x) and testSomethingElse(@SomeData int x) variables.  
  
The presenter highlights the following line of code below the testSomethingElse(@SomeData int x) variable:  
  
       System.out.println(x + " Something else");  
  
The presenter then highlights the following line of code below the testSomething(@SomeData int x) variable:  
  
       assertTrue(x < 3);*   
  
And so the way we do that is when we throw in the theory out there. We throw in the @Theory annotation to say— by the way the @Theory annotation says look for some parameters here. I define another annotation to define where this data is coming from. And so I just have one inside of here, I could have multiples of these. But this is a interface I created. And the interface has some annotations for it to say, "Hey, this is a runtime retention inside there." So it says "Only keep this data through runtime." And it's supplied by which other class. And so you can see I'm creating a class that's going to go provide the data for this guy. Some data is being passed in. And so particularly, I can build an object that loads all test data from the database and then hands it off to the tests as it does inside of here. In this case of an integer but I could have much more complex data if I wanted to. And so here my test data supplier is extending out the pre-made class, parameter supplier and its saying, "Hey, get some values out there." In my case, I'm going through and I'm adding some data. And I have this array of potential assignments going inside of there. So I have a String and an Integer, String and Integer, String an Integer that's being put inside of here. And so that potential assignment class is something that's created by JUnit. It's something that's out there that says "This is the data, information about the data and then this is the value you want", you know, that's going to be stored inside of here.   
*In the TheoryWithSuppplier.java file, the presenter highlights the @Theory annotation.  
  
The presenter highlights @SomeData in the following line of code:  
  
   public void testSomething(@SomeData int x)  
  
Next the presenter switches to the SomeData.java file. It contains the following code:  
  
package myapp.junit.theory;  
  
import java.lang.annotation.Retention;  
  
@Retention(RetentionPolicy.RUNTIME)  
@ParametersSuppliedBy(TestDataSupplier.class)  
public @interface SomeData   
{  
  
}  
  
The presenter highlights the code public @interface SomeData  
  
Next the presenter highlights the code @Retention(RetentionPolicy.RUNTIME)  
  
He highlights the code @ParametersSuppliedBy(TestDataSupplier.class)  
  
The presenter then switches to the TheoryWithSuppplier.java file. He highlights @SomeData int x in the following segment of code:  
  
   @Theory  
   public void testSomething(@SomeData int x)  
  
The presenter switches to the TestDataSupplier.java file. It contains the following code:  
  
package myapp.junit.theory;  
  
import java.util.ArrayList;  
  
public class TestDataSupplier extends ParameterSupplier  
{  
  
   @Override  
   public List<PotentialAssignment> getValueSources(ParameterSignature sig)  
   {  
       List<PotentialAssignment> result = new ArrayList<PotentialAssignment>();  
       result.add(PotentialAssignment.forValue("One", new Integer(1)));  
       result.add(PotentialAssignment.forValue("Any text", new Integer(2)));  
       result.add(PotentialAssignment.forValue("Too big", new Integer(3)));  
       result.add(PotentialAssignment.forValue("Way Too big", new Integer(4)));  
      return result;  
   }  
}  
  
The presenter highlights ParameterSupplier in the following code line:  
  
public class TestDataSupplier extends ParameterSupplier  
  
The presenter refers to the getValueSources() variable in the following code line:  
  
   public List<PotentialAssignment> getValueSources(ParameterSignature sig)  
  
Next the presenter refers to each of the following arrays:  
       result.add(PotentialAssignment.forValue("One", new Integer(1)));  
       result.add(PotentialAssignment.forValue("Any text", new Integer(2)));  
       result.add(PotentialAssignment.forValue("Too big", new Integer(3)));  
       result.add(PotentialAssignment.forValue("Way Too big", new Integer(4)));  
  
The presenter hovers the cursor over PotentialAssignment in the following code line:  
  
       List<PotentialAssignment> result = new ArrayList<PotentialAssignment>();  
  
PotentialAssignment turns into a link and the presenter clicks it. A file containing information about the PotentialAssignment class opens in a separate tabbed page in the editor. It contains the following information:  
  
Class File Editor  
  
Source not found  
  
The JAR of this class file belongs to container 'JUnit 4' which does not allow modifications to source attachments on its entries.  
  
//Compiled from PotentialAssignment.java (version 1.5 : 49.0, super bit)  
public abstract class org.junit.experimental.theories.PotentialAssignment{  
  
//Method descriptor#11 ()V  
//Stack:1, Locals:1  
public PotentialAssignment();  
 0 aload\_0 [this]  
 1 invokespecial java.lang.Object() [1]  
 4 return  
  Line numbers:  
   [pc:0, line:3]  
   [pc:4, line:4]  
  Local variable table:  
   [pc:0, pc:5] local: this index:0 type: org.junit.experimental.theories.PotentialAssignment  
  
The presenter closes the file and switches to the TestDataSupplier.java file. It contains the following code:   
  
package myapp.junit.theory;  
  
import java.util.ArrayList;  
  
public class TestDataSupplier extends ParameterSupplier  
{  
  
   @Override  
   public List<PotentialAssignment> getValueSources(ParameterSignature sig)  
   {  
       List<PotentialAssignment> result = new ArrayList<PotentialAssignment>();  
       result.add(PotentialAssignment.forValue("One", new Integer(1)));  
       result.add(PotentialAssignment.forValue("Any text", new Integer(2)));  
       result.add(PotentialAssignment.forValue("Too big", new Integer(3)));  
       result.add(PotentialAssignment.forValue("Way Too big", new Integer(4)));  
      return result;  
   }  
}  
  
The presenter highlights the code "One" and then Integer in the following code line:  
       result.add(PotentialAssignment.forValue("One", new Integer(1)));*   
  
So this potential assignment out there gives us extra metadata when you go and execute our test. Let's go and execute this test and see what happens. So as I'm going and running through, you can see I have that test value called "Too big" and that's coming out of here. "Too big" is the third value. And that's going to fail the test when I get down to this third test of a search who is less than, x is less than three. And so now I can externalize data. Now remember this is a shotgun approach. There is no expected value. This is much better for say performance testing or some other testing that I'm really just trying to throw a lot of data to see how the system responds. It doesn't give me a clear mechanism for saying this is how the data is valid. Okay, I have input X, Y and the result should be Z. I have to come up with that mechanism on my own. If I'm going to check things that's I have to configure that out on my own as a data being passed in. As I said, you can pass the complex object. You can have the results being passed in that way, but it's not built into the system. It's not built in where its defined for you. But the theories does give you a whole lot of test data with a relatively small amount of work. And even better, I can externalize that test data. So as a test designer I am not responsible for every single test case scenario and data that goes into my test.   
*In the TestDataSupplier file, the presenter refers to the ArrayList<PotentialAssignment>() variable in the following code line:  
  
       List<PotentialAssignment> result = new ArrayList<PotentialAssignment>();  
  
The presenter switches to the TheoryWithSuppplier.java file. It contains the following code:  
  
package myapp.junit.theory;  
  
import static org.junit.Assert.assertTrue;  
  
@RunWith(Theories.class)  
public class TheoryWithSuppplier  
{  
   @Theory  
   public void testSomething(@SomeData int x)  
   {  
       System.out.println(x + " Something");  
       assertTrue(x < 3);  
   }  
  
  
   @Theory  
   public void testSomethingElse(@SomeData int x)  
   {  
       System.out.println(x + " Something else");  
   }  
}  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.theory.TheoryWithSuppplier and its runtime is 0.009 seconds. The top-level test node is expanded and the following test methods and associated runtimes are listed below it – testSomethingElse (0.002 s) and testSomething (0.007 s). The testSomethingElse method ran successfully and the testSomething method returned an error. The testSomething method is selected and the presenter refers to the following error output in the Failure Trace section:  
  
org.junit.experimental.theories.internal.ParameterizedAssertionError: testSomething(Too big)  
Caused by: java.lang.AssertionError  
at myapp.junit.theory.TheoryWithSuppplier.testSomething(TheoryWithSuppplier.java:16)  
...20 more  
  
In the editor, the presenter switches to the TestDataSupplier.java file. It contains the following code:  
  
package myapp.junit.theory;  
  
import java.util.ArrayList;  
  
public class TestDataSupplier extends ParameterSupplier  
{  
  
   @Override  
   public List<PotentialAssignment> getValueSources(ParameterSignature sig)  
   {  
       List<PotentialAssignment> result = new ArrayList<PotentialAssignment>();  
       result.add(PotentialAssignment.forValue("One", new Integer(1)));  
       result.add(PotentialAssignment.forValue("Any text", new Integer(2)));  
       result.add(PotentialAssignment.forValue("Too big", new Integer(3)));  
       result.add(PotentialAssignment.forValue("Way Too big", new Integer(4)));  
      return result;  
   }  
}  
  
The presenter highlights the value "Too big" in the following code line:  
  
       result.add(PotentialAssignment.forValue("Too big", new Integer(3)));  
  
The presenter then switches to the TheoryWithSuppplier.java file. It contains the following code:  
  
package myapp.junit.theory;  
  
import static org.junit.Assert.assertTrue;  
  
@RunWith(Theories.class)  
public class TheoryWithSuppplier  
{  
   @Theory  
   public void testSomething(@SomeData int x)  
   {  
       System.out.println(x + " Something");  
       assertTrue(x < 3);  
   }  
  
  
   @Theory  
   public void testSomethingElse(@SomeData int x)  
   {  
       System.out.println(x + " Something else");  
   }  
}  
  
The presenter highlights the following code line:  
  
       assertTrue(x < 3);  
  
Finally, the presenter highlights the code (@SomeData int x) in the following line:  
  
   public void testSomething(@SomeData int x)*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

JUnit and Mock Objects

Learning Objective

*After completing this topic, you should be able to*

* *work with mock objects in JUnit tests*

**1. Working with mock objects in tests**

To isolate a unit under test, we'll often create test dobs and mock objects to simplify the system setup. Creating these mock objects has become significantly easier with the addition of clever frameworks. Let's look at using one of these frameworks within JUnit to simplify our test creation.So we're going to look at the Mockito framework. And so when we run our test, we have to tell it we're running with the MockitoJUnitRunner. And so what Mockito does is allows us to take our unit under test. And I'm just creating any class or testing, whatever is going to do, and we're going to plug-in the dependency to that class that's entirely made up by our framework. So as our unit under test is being created, you can see our constructor here to set up. We have our setup here, the setup under test, it needs an instance of this mock object to be created. So as part of its constructor, it needs these amazing features to be able to do its work. And so I pass in an implementation of the amazing features. Now, amazing features right now is just the interface. I don't know how it's going to be implemented, maybe it's going to be a web service, maybe it's going to be a database, maybe it's going to be a neural network magic – who knows what it's going to be out there. But what we do know is how we want it to respond for our test cases, how we want it to behave for our test cases. So as we go through and define our test cases, I can take this mock object I'm creating using the mock annotation –   
*In the Eclipse development environment, two files – TestMe.java and AmazingFeatures.java – are open on separate tabbed pages in the editor.  
  
In the TestMe.java file, the following code is visible:  
  
package target.mock;  
  
import static org.junit.Assert.assertEquals;  
  
@RunWith(MockitoJUnitRunner.class)  
public class TestMe  
{  
   private UUT target;  
   @Mock  
   private AmazingFeatures mockedObject;  
     
   @Before  
   public void setup()  
   {  
       target = new UUT(mockedObject);  
   }  
     
   @Test  
   public void checkNumber()  
   {  
       when(mockedObject.guessNumber()).thenReturn(5);  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));  
   }  
  
   @Test  
   public void checkText()  
  
The presenter highlights the following code line:  
  
@RunWith(MockitoJUnitRunner.class)  
  
Below the public class TestMe, the presenter highlights the following code line:  
  
   private UUT target;  
  
The presenter then highlights the individual components in the following segment of code:  
  
   @Before  
   public void setup()  
   {  
       target = new UUT(mockedObject);  
   }  
  
Next the presenter hovers over and clicks the UUT declaration hyperlink in the following code line:  
  
       target = new UUT(mockedObject);  
  
A UUT.java file opens in the editor. It contains the following code:  
  
package target.mock;  
  
public class UUT  
{  
   private AmazingFeatures dependentObject;  
  
   public UUT(AmazingFeatures dependentObject)  
   {  
       this.dependentObject = dependentObject;  
   }  
     
   public int numberDependency(int input)  
   {  
       return dependentObject.guessNumber() + input;  
   }  
     
   public String textDependency(String append)  
   {  
       return dependentObject.guessWord() + append;  
   }  
}  
  
The presenter highlights the following line of code:  
  
   public UUT(AmazingFeatures dependentObject)  
  
Next the presenter switches to the AmazingFeatures.java file. It contains the following code:  
  
package target.mock;  
  
public interface AmazingFeatures  
{  
   public int guessNumber();  
     
   public String guessWord();  
}  
  
The presenter switches back to the TestMe.java file. Again, it contains the following code:  
  
package target.mock;  
  
import static org.junit.Assert.assertEquals;  
  
@RunWith(MockitoJUnitRunner.class)  
public class TestMe  
{  
   private UUT target;  
   @Mock  
   private AmazingFeatures mockedObject;  
     
   @Before  
   public void setup()  
   {  
       target = new UUT(mockedObject);  
   }  
     
   @Test  
   public void checkNumber()  
   {  
       when(mockedObject.guessNumber()).thenReturn(5);  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));  
   }  
  
   @Test  
   public void checkText()  
   {  
       when(mockedObject.guessWord()).thenReturn("Na-na-na-na-na-na");  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }  
  
   @Test(expected=IOException.class)  
   public void checkError()  
   {  
       when(mockedObject.guessWord()).thenThrow(IOException.class);  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }  
}  
  
The presenter refers to the mockObject and @Mock annotation in the following segment of code:  
  
   private UUT target;  
   @Mock  
   private AmazingFeatures mockedObject;*   
  
so the mock annotation here is coming out of the Mockito framework, and it's going to create an instance of this object for us. And so as it creates an instance of this object, it's going to allow us to use some features that is coming out of this static import of Mockito. And so that's where these methods are magically coming from – the when method in particular. And so we're saying, when the mock object calls the method guessNumber, okay, so when the guessNumber method is called on the mock object. So you can see, guessNumber is…guessNumber, guessWord; there's two of these amazing features that's going on inside of here. So when this is called, I want you to return the value 5. And so for this test case, any time guessNumber is called, it's going to return 5. And so as I execute this, I fix both sides with 5 and 15. And so our code that's going on inside of here…so what, you know, we're saying we've a number dependency 10 on our target. Our target is unit under test here. And so when I call unit under test target for number dependency, I'm taking the number I'm guessing, I'm adding the number you have got right inside of there, and so I'm saying 5 plus 10, and the answer is going to be 15. So I'm asserting that we get 15 back, and obviously we do.   
*In the TestMe.java file, the presenter clicks in front of @Mockannotation and a drop-down list displays two options – Mock - org.mockito and Mock - org.mockito.MockitoAnnotations. The presenter hovers the cursor over the second option, Mock - org.mockito.MockitoAnnotations.  
  
In the file, the presenter refers again to the following code line:  
  
   private AmazingFeatures mockedObject;  
  
Below the checkNumber() variable, the presenter highlights the first line in the following code segment:  
  
   {  
       when(mockedObject.guessNumber()).thenReturn(5);  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));  
   }  
  
The presenter expands the code node import static org.junit.Assert.assertEquals; to reveal the following code lines:  
  
import static org.mockito.Mockito.when;  
  
import java.io.IOException;  
  
import org.junit.Before;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.mockito.Mock;  
import org.mockito.runners.MockitoJUnitRunner;  
  
The presenter refers to the following line of code:  
  
import static org.mockito.Mockito.when;  
  
The presenter refers to the two instances of the when() method in the following segment of code:  
  
   @Test  
   public void checkNumber()  
   {  
       when(mockedObject.guessNumber()).thenReturn(5);  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));  
   }  
  
   @Test  
   public void checkText()  
   {  
       when(mockedObject.guessWord()).thenReturn("Na-na-na-na-na-na");  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }  
  
Below the checkNumber() variable, the presenter highlights the mockedObject and the guessNumber method in the following code line:  
       when(mockedObject.guessNumber()).thenReturn(5);  
  
The presenter then switches to the AmazingFeatures.java file. Again, it contains the following code:  
  
package target.mock;  
  
public interface AmazingFeatures  
{  
   public int guessNumber();  
     
   public String guessWord();  
}  
  
The presenter highlights the guessNumber method in the following code line:  
  
   public int guessNumber();  
  
Next the presenter highlights the guessWord method in the following code line:  
  
   public String guessWord();  
  
The presenter returns to the TestMe.java file which, again, contains the following code:  
  
package target.mock;  
  
import static org.junit.Assert.assertEquals;  
  
@RunWith(MockitoJUnitRunner.class)  
public class TestMe  
{  
   private UUT target;  
   @Mock  
   private AmazingFeatures mockedObject;  
     
   @Before  
   public void setup()  
   {  
       target = new UUT(mockedObject);  
   }  
     
   @Test  
   public void checkNumber()  
   {  
       when(mockedObject.guessNumber()).thenReturn(5);  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));  
   }  
  
   @Test  
   public void checkText()  
   {  
       when(mockedObject.guessWord()).thenReturn("Na-na-na-na-na-na");  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }  
  
   @Test(expected=IOException.class)  
   public void checkError()  
   {  
       when(mockedObject.guessWord()).thenThrow(IOException.class);  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }  
}  
  
Below the checkNumber() method, the presenter runs through the logic of the following two code lines:  
       when(mockedObject.guessNumber()).thenReturn(5);  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));  
  
The presenter points out the code private UUT target;.  
  
The presenter switches to the UUT.java file, which contains the following code:  
  
package target.mock;  
  
public class UUT  
{  
   private AmazingFeatures dependentObject;  
  
   public UUT(AmazingFeatures dependentObject)  
   {  
       this.dependentObject = dependentObject;  
   }  
     
   public int numberDependency(int input)  
   {  
       return dependentObject.guessNumber() + input;  
   }  
     
   public String textDependency(String append)  
   {  
       return dependentObject.guessWord() + append;  
   }  
}  
  
The presenter points out the guessNumber and + input elements in the following line of code:  
  
       return dependentObject.guessNumber() + input;  
  
The presenter switches to the TestMe.java file, which, again, contains the following code:  
  
package target.mock;  
  
import static org.junit.Assert.assertEquals;  
  
@RunWith(MockitoJUnitRunner.class)  
public class TestMe  
{  
   private UUT target;  
   @Mock  
   private AmazingFeatures mockedObject;  
     
   @Before  
   public void setup()  
   {  
       target = new UUT(mockedObject);  
   }  
     
   @Test  
   public void checkNumber()  
   {  
       when(mockedObject.guessNumber()).thenReturn(5);  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));  
   }  
  
   @Test  
   public void checkText()  
   {  
       when(mockedObject.guessWord()).thenReturn("Na-na-na-na-na-na");  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }  
  
   @Test(expected=IOException.class)  
   public void checkError()  
   {  
       when(mockedObject.guessWord()).thenThrow(IOException.class);  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }  
}  
  
The presenter refers to the values 5, 10, and 15 in the following line of code:  
  
       assertEquals("I fixed both sides with 5 and 10", 15, target.numberDependency(10));*   
  
So the cool thing about using mocked objects is I don't have to implement that – I don't have to say create a class and return 15. But even better, each test might want something different. One might return 5, one might return 7. Instead of having defined a bunch of different mock objects, or some complicated logic inside of mock objects in order to figure that out, the Mockito framework just tells us right here, "hey, when this  gets called, return this thing", and it will do that the next time it's called. I could change it after the next call if I wanted to. So here, let's look at another one. I can say when I call guessWord, then I return na-na-na-na-na. And so, again I have text inside of here. And so I'm saying, it should be na-na-na-na-na, with an exclamation point in inside of there. So for my implementation this time, I take the text that it's guessing and putting an exclamation point at the end. The last thing we can do here is, if inside the mockito framework, -   
*In the TestMe.java file, the presenter refers to the following line of code:  
  
       when(mockedObject.guessNumber()).thenReturn(5);  
  
Below the checkText() method, the presenter runs through the logic of the following segment of code:  
  
   {  
       when(mockedObject.guessWord()).thenReturn("Na-na-na-na-na-na");  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }*   
  
I can say when guessWord is called – or whatever method is called inside of there – then throw an exception, IO exception, whatever type of exception I want to have happen. So again, instead of having to create a mock object, a mock class that will generate errors, that will cause errors to happen – again, more code you have to do, it might be complex to make it happen, I just tell the mockito framework, "hey, if they call this method, make an exception happen". And so when this happens, you can see my test is expecting the exception, I'm expecting the IO exception, and it indeed gets called when I call the method. So I call text dependency, text dependency says guessWord, and then exceptions thrown. So Mockito is only one of several mock object frameworks that are out there. But taking a look at this one, you can see some of the capabilities and how I can tie that into my chain of tests, and make my chain of test really flexible, and put control of the results in the hands of the test designer and developer, which may or may not be the same person that's developing the code, which gives us a lot of flexibility in our test design.   
*Finally, in the TestMe.java file, the presenter runs through the logic of the following segment of code:  
  
   @Test(expected=IOException.class)  
   public void checkError()  
   {  
       when(mockedObject.guessWord()).thenThrow(IOException.class);  
       assertEquals("I fixed both sides with text", "Na-na-na-na-na-na!", target.textDependency("!"));  
   }*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Maven and JUnit

Learning Objective

*After completing this topic, you should be able to*

* *run JUnit tests with Maven*

**1. Running tests with Maven**

JUnit is most effective when run with every change, to ensure that the code is always working. This integrating JUnit into tools such as Maven, which executes many Java builds these days, allows us to test the code each time it is recompiled. Let's take a look at integrating JUnit within Maven. So I've a project set up within Maven. And so Maven is used to do a lot of things for us, it manages our dependencies, it completes our builds, it gives us conventions that guides our projects, and we can use our JUnit within this as well. So if we look at the setup of the Maven project, you can see the first thing it does for us is it separates the source for our main application, from the source from our test application. So we have what we need over here for executing the application, and separately over here, we define our test cases. And so I don't need to worry about compiler rules or separate projects or anything like that, I can keep everything together, but still have a separation in how things are built and packaged.  
Now, -   
*In the Eclipse development environment, the Package Explorer is open. A Maven project is expanded. It contains eight child nodes – src/main/java, src/main/resources, src/test/java, src/test/resources, Maven Dependencies, JRE System Library, src, and target.  
  
The src/main/java child node contains two packages – target and target.mock. The target package contains two files – Account.java and InsuffientFundsException.java.  
  
The src/test/java child node contains two packages – target.test and target.test.mock. The target.test package contains two files – AccountTest.java and TestTarget.java.  
  
The target child folder under the Maven project contains a file named pom.xml.  
  
The Maven/pom.xml file is open in the editor. It contains the following code:  
  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
<modelVersion>4.0.0</modelVersion>  
<groupId>junit</groupId>  
<artifactId>1</artifactId>  
<version>0.0.1-SNAPSHOT</version>  
<name>junitMaven</name>  
<dependencies>  
<dependency>  
<groupId>junit</groupId>  
<artifactId>junit</artifactId>  
<version>4.8.2</version>  
</dependency>  
<dependency>  
<groupId>org.mockito</groupId>  
<artifactId>mockito-all</artifactId>  
<version>1.9.5</version>  
</dependency>  
</dependencies>  
  
<build>  
<plugins>  
<plugin>  
<groupId>org.apache.maven.plugins</groupId>  
<artifactId>maven-surefire-plugin</artifactId>  
<version>2.17</version>  
</plugin>  
</plugins>  
</build>  
</project>*   
  
within the dependency management framework Maven, I set up what's called a project file, a POM file is what's going to go outside, out there. And I can manage the dependencies, such as my dependency here on JUnit. I can say "this is the version of JUnit I'm running with", I have 4.8.2 is the version I'm running with right here. And if I want to change this, it's just simply as I have to go off and change this number Maven will manage downloading the new version of JUnit, installing that as a dependency, and compiling against it, and running against it as proper. Whatever other dependencies are out there for my framework, I can use those as well. I don't have to now run around and find jar files, and things like that. But this isn't a sales pitch on Maven, this is a sales pitch on JUnit. So let's get back to JUnit. So when I want to go and I want to run now, I can go through into my project, right click, and say run as, and it'll give me a bunch of Maven options of how do I want to execute this. So I can just simply compile the Maven code, I can clean up my project, or I can run what's called Maven test. And so when I do that, it goes through, and we'll look at this whole sequence here. It starts off, it cleans up everything, it executes the compilation. So you can see it compiles, and in this case everything is up-to-date – didn't need to build anything new. It'll build everything, package it up and then run the tests.   
*In the Maven/pom.xml file, the presenter highlights the following code:  
  
<dependency>  
<groupId>junit</groupId>  
<artifactId>junit</artifactId>  
<version>4.8.2</version>  
</dependency>  
  
The presenter highlights the JUnit version number 4.8.2 defined within the same dependency code.  
  
He then highlights the second segment of dependency code:  
  
<dependency>  
<groupId>org.mockito</groupId>  
<artifactId>mockito-all</artifactId>  
<version>1.9.5</version>  
</dependency>  
  
In the Package Explorer, the presenter right-clicks the Maven project and selects the Run As option. This opens a submenu  with the options 1 Java Applet, 2 Java Application, 3 JUnit Test, 4 Maven build, 5 Maven build, 6 Maven clean, 7 Maven generate-sources, 8 Maven install, 9 Maven test, and Run Configurations. The presenter selects the 9 Maven test option.  
  
The test runs and output is returned to the Console pane below the editor. The output is as follows:  
  
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".  
SLF4J: Defaulting to no-operation (NOP) logger implementation  
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.  
[INFO] Scanning for projects...  
[INFO]  
[INFO]                                                                          
[INFO] Building junitMaven 0.0.1-SNAPSHOT  
[INFO]                                                                        
[INFO]  
[INFO] --- maven-resources-plugin:2.5: resources (default-resources) @ 1 ---  
[debug] execute contextualize  
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!  
[INFO] Copying 0 resource  
[INFO]  
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @ 1 ---  
[INFO] Nothing to compile - all classes are up to date  
[INFO]   
[INFO] --- maven-resources-plugin:2.5:testReources (default-testResources) @ 1 ---  
[debug] execute contextualize  
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!  
[INFO] Copying 0 resource  
[INFO]  
[INFO] ---maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @ 1 ---  
[INFO]  
[INFO] Nothing to compile - all classes are up to date  
[INFO]  
[INFO] --- maven-surefire-plugin:2.17:test (default-test) @ 1 ---  
[INFO] Surefire report directory: E:\Apps\Dropbox\JUnit\Maven\target\surefire-reports  
  
                                                        
TESTS  
                                                        
Running target.test.AccountTest  
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.029 sec - in target.test.AccountTest  
Running target.test.TestTarget  
Add  
Add  
Divide  
Tests run: 4, Failures: 1, Errors: 0, Skipped: 1, Time elapsed: 0.002 sec <<< FAILURE! - in target.test.TestTarget  
badTestCase(target.test.TestTarget) Time elapsed: 0.001 sec  <<< FAILURE!  
junit.framework.AssertionFailedError: Adding is fun expected: <6> but was: <4>  
       at junit.framework.Assert.fail(Assert.java:47)  
       at junit.framework.Assert.failNotEquals(Assert.java:283)  
       at junit.framework.Assert.failNotEquals(Assert.java:64)  
       at junit.framework.Assert.failNotEquals(Assert.java:195)  
       at junit.framework.Assert.failNotEquals(Assert.java:28)  
  
The presenter refers to the following output line:  
  
[debug] execute contextualize  
  
The presenter highlights the following output line:  
  
[INFO] Nothing to compile - all classes are up to date*   
  
And so you can see, as a high level report, it ran 9 tests. No failures, no errors, no skips for the account test, one that was going on here, that was being executed. And then, it ran over here, 4 tests inside of this next one, this is the test target inside of here, and one of them failed. And so it gives us the summary of all the test classes inside of there, and it will run every single JUnit you have inside of your system.Now, that can be a good thing; it means that every time I redo my code, every time I'm going to execute it again, as far as a big build…not that I want to run it every time like that, I want to run every test every time I'm going to run my local app. But when I do a big build, it'll make sure everything has been compiled, that is I check in code, and rebuild it. I'll know all of my test stuff is working right away. If something breaks, I know right away that it's broken. Now, locally I don't want to have to do that. I don't want to every time I want to run, have to run every single one of the tests, particularly if my test takes a long time. So what I can do instead is, if I go to my run configurations, -   
*In the Console pane, the presenter highlights and runs through the following AccountTest output details:  
  
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.029 sec - in target.test.AccountTest  
  
The presenter also highlights the following test.TestTarget output details:  
  
Tests run: 4, Failures: 1, Errors: 0, Skipped: 1, Time elapsed: 0.002 sec <<< FAILURE! - in target.test.TestTarget  
  
Next the presenter clicks the Run drop-down button on the toolbar and selects the Run Configurations option. The Run Configurations dialog box opens. It contains a pane that lists available configuration options and a section with six tabs – Main, JRE, Refresh, Source, Environment, and Common.  
  
The Main tabbed page contains a Base directory text box that's currently set to ${workspace\_loc:/Maven} and three buttons – Browse Workspace, Browse File System, and Variables. It contains a Goals text box, which is currently set to Dtest=target.test.AccountTest.test, a Profiles text box, six checkboxes – Offline, Update Snapshots, Debug Output, Skip Tests, Non-recursive, and Resolve Workspace artifacts, and a Threads   
drop-down list box that's set to 1. The tabbed page also contains a table with Parameter Name and Value columns and associated Add, Edit, and Remove buttons, a Maven Runtime drop-down list box that's currently set to Embedded (3.0.4/1.4.0.20130531-2315), a Configure button, and Apply, Revert, Run, and Close buttons.*   
  
I can create my own run configuration, just to say "hey, this one I'm saying AccountOnly". I only want to test the account test inside of here, so I do a –d test as a goal that I'm running inside of here. So when I run my JUnit, excuse me, my Maven build in this case, I'm only going to run this JUnit account test inside of here. So if I run this one, you can see it builds what's needed for that, it runs what's needed for that, and that build was successful here. So in this case, it only ran the account test, those 9 that were successful, 0 failures. The net result is nothing failed, the build was successful. If you notice, when I ran this guy before, and I said "run as Maven test", and it ran everything, the build failed. Because, as far as Maven was concerned, if everything didn't compile properly, if everything run proper tests, then it wasn't a successful build. You should go and look at it. And so here, I had 13 run, and 1 failed, and 1 was skipped, and so you should go ahead and fail this build. So this way you can manage all of your testing, get as often of testing, that's the whole point of automated testing – to get it as often as we can, but still locally be able to just focus on what I'm working on. So use Maven, if you so choose, to great effect to keep JUnit running as often as possible.   
*In the Run Configurations dialog box, the presenter selects AccountOnly from the list of Maven Build configuration option. He highlights the entry in the Goals text box – -Dtest=target.test.AccountTest.test, and clicks the Run button to run the pom.xml code using the AccountOnly configuration.  
  
The presenter scrolls down the Console pane to reveal the following output:  
  
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".  
SLF4J: Defaulting to no-operation (NOP) logger implementation  
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.  
[INFO] Scanning for projects...  
[INFO]  
[INFO]                                                                          
[INFO] Building junitMaven 0.0.1-SNAPSHOT  
[INFO]                                                                        
[INFO]  
[INFO] --- maven-resources-plugin:2.5: resources (default-resources) @ 1 ---  
[debug] execute contextualize  
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!  
[INFO] Copying 0 resource  
[INFO]  
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @ 1 ---  
[INFO] Nothing to compile - all classes are up to date  
[INFO]   
[INFO] --- maven-surefire-plugin:2.17:test (default-test) @ 1 ---  
[INFO] Surefire report directory: E:\apps\Dropbox\JUnit\Java\Maven\target\surefire-reports  
  
                                                        
TESTS                                                       
                                                       
Running target.test.AccountTest  
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.029 sec - in target.test.AccountTest  
  
Results :  
  
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0  
  
[INFO]                                                                           
[INFO] BUILD SUCCESS  
[INFO]                                                                                      
[INFO] Total time: 1.288s  
[INFO] Finished at: Sun May 04 12:05:54 CDT 2014  
[INFO] Final Memory: 7M/184M  
[INFO]                                                                                  
  
The presenter highlights the following line under the TESTS output section:  
  
[INFO] BUILD SUCCESS  
  
The presenter refers to the AccountTest details in the following output segment:  
  
Running target.test.AccountTest  
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.029 sec - in target.test.AccountTest  
  
Results :  
  
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0  
  
[INFO]                                                                           
[INFO] BUILD SUCCESS  
  
Next the presenter clicks the Run drop-down button on the toolbar and selects Run as - 9 Maven Test.  
  
In the Package Explorer view, the presenter right-clicks the Maven project and selects Run As - 9 Maven test.   
  
In the Console pane, the presenter scrolls down to the following output:  
  
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".  
SLF4J: Defaulting to no-operation (NOP) logger implementation  
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.  
[INFO] Scanning for projects...  
[INFO]  
[INFO]                                                                          
[INFO] Building junitMaven 0.0.1-SNAPSHOT  
[INFO]                                                                        
[INFO]  
[INFO] --- maven-resources-plugin:2.5: resources (default-resources) @ 1 ---  
[debug] execute contextualize  
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!  
[INFO] Copying 0 resource  
[INFO]  
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @ 1 ---  
[INFO] Nothing to compile - all classes are up to date  
[INFO]   
[INFO] --- maven-surefire-plugin:2.17:test (default-test) @ 1 ---  
[INFO] Surefire report directory: E:\apps\Dropbox\JUnit\Java\Maven\target\surefire-reports  
  
  
TESTS  
                                                        
Running target.test.AccountTest  
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.03 sec - in target.test.AccountTest  
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.17:test (default-test) on project 1:  
[ERROR]  
[ERROR] Please refer to E:\apps\Dropbox  
  
The presenter scrolls further down the Console view. The following run output details are now visible:  
  
Tests run: 13, Failures: 1, Errors: 0, Skipped: 1  
  
[INFO]                                                                                            
[INFO] BUILD FAILURE  
[INFO]                                                                                                  
[INFO] Total time: 1.291s  
[INFO] Finished at: Sun May 04 12:06:14 CDT 2014  
[INFO] Final Memory: 7M/184M  
[INFO]                                                                                            
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.17:test (default-test) on project 1:  
[ERROR]  
  
The presenter highlights BUILD FAILURE in the test results.  
  
Finally, the presenter highlights the following test results:  
  
Tests run: 13, Failures: 1, Errors: 0, Skipped: 1*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Legacy JUnit Tests

Learning Objective

*After completing this topic, you should be able to*

* *configure JUnit tests for legacy code*

**1. Configuring tests for legacy code**

JUnit has been a solid testing framework for a very long time. The most recent versions take a great advantage of annotations. But many tests were created before this job enhancement. Being aware of the legacy implementation may be helpful. Let's look at some legacy test cases and how they relate to modern JUnit. So this is a test case all ready to go that you might come across in your work. Not this one specifically, but the test cases are already there. Now old school JUnit test cases had to extend out the JUnit class test case. You can see it's imported there and extend that out, and that tied us into the JUnit testing framework. Past that, the way I define tests back then was it had to start with the first four letters being test. So I testSample, testMore, testAir, test whatever. And JUnit would go through all your methods in your class and pick any ones that started with test and execute those as a test. Now as you want it to setUp or tearDown, there's a specific Overrided method to setUp your test or tearDown your test. So you had a little bit less flexibility in that you had to implement those specific methods. But at the same time it still gave you the setUp and tearDown options. There was no options for static test setUp and tearDown. That's built within your class as it is, it's called by yourself, you can figure that out on your own.   
*In the Eclipse development environment, two files are open on separate tabbed pages – OldSchoolTest.java and AnotherOldSchoolTest.java.  
  
The OldSchoolTest.java file contains the following code:  
  
package myapp.junit.old;  
  
import junit.framework.TestCase;  
  
public class OldSchoolTest extends TestCase  
{  
   public OldSchoolTest()  
   {  
       super();  
   }  
  
   public OldSchoolTest(String name)  
   {  
       super(name);  
   }  
  
   @Override  
   protected void setUp() throws Exception  
   {  
       System.out.println("Run before each test");  
   }  
  
   @Override  
   protected void tearDown() throws Exception  
   {  
       System.out.println("Run after each test");  
   }  
  
   public void testSample()  
   {  
       System.out.println("Test Sample");  
         
       super.assertSame("Both are the same", "test", "test");  
       super.assertNotSame("Not the same", "test", new String("test"));  
  
       super.assertEquals("Both are the equals", "test", "test");  
   }  
  
   public void testMore()  
   {  
       System.out.println("Test More");  
         
       super.assertNull("Null", null);  
       super.assertNotNull("Not null", "test");  
         
       super.assertFalse("This is false", false);  
       super.assertTrue("This is true", true);  
   }  
     
   public void testError()  
   {  
       System.out.println("Test Error");  
       if (true)  
       {  
           fail("This was a bad decision");  
       }  
   }  
     
   public void notATest()  
   {  
       fail();  
   }  
}  
  
The presenter highlights the code extends TestCase in the following line:  
  
public class OldSchoolTest extends TestCase  
  
The presenter scrolls down the OldSchoolTest.java file and refers to the first four letters of the following methods: testSample(), testMore(), and testError().  
  
Next the presenter highlights the @Override annotation and refers to the setUp() method in the following code segment:  
  
   @Override  
   protected void setUp() throws Exception  
  
The presenter refers to the setUp() and tearDown() variables in the following segment of code:  
  
   @Override  
   protected void setUp() throws Exception  
   {  
       System.out.println("Run before each test");  
   }  
  
   @Override  
   protected void tearDown() throws Exception  
   {  
       System.out.println("Run after each test");  
   }*   
  
So past that, when I go to execute the tests, when I wanted to assert something or I wanted to make some checks on things all of those were provided for me within the super test class inside of here. So I'll just say super.assertSame, super.assertNotSame, and super is optional inside of there obviously. I'm just showing you where that stuff's coming from in this case. So as it changes out, then those things aren't located there, they're located in other classes. That was one limitation of the design for test, testing in JUnit is everything had to be in that super class. It would have this all the features and stuff. Well now we're not limited to that. But inside of there, you can see if I didn't have the word test to start it up it wouldn't show up as a test. So if I go and run this it runs using the same basic ideas and uses the same outputs inside of here. It runs the three tests: testSample, testMore, testError. notATest doesn't run. There's nothing inside of there that makes that guy run. So this old way was very effective. It worked really, really well but it's not as flexible as the new way.   
*Below the testSample() method, the presenter highlights super.assertSame in the following line of code:  
  
       super.assertSame("Both are the same", "test", "test");  
  
The presenter scrolls up in the OldSchoolTest.java file and highlights TestCase in the following line of code:  
  
public class OldSchoolTest extends TestCase  
  
The presenter scrolls back down in the file and refers to the following lines of code:  
  
       super.assertSame("Both are the same", "test", "test");  
       super.assertNotSame("Not the same", "test", new String("test"));  
  
The presenter scrolls to the bottom of the OldSchoolTest.java file and highlights the following code:  
  
   public void notATest()  
   {  
       fail();  
   }  
  
The presenter then clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.old.OldSchoolTest and its runtime is 0.006 seconds. The top-level test node is expanded and the following test methods and associated runtimes are listed below it – testError (0.004 s), testMore (0.000 s), and testSample (0.001 s). The testError method failed and the testMore and testSample methods ran successfully.  
  
In the editor, the presenter scrolls down in the OldSchoolTest.java file and refers to each of the following test methods – testSample(), testMore(), and testError().  
  
The presenter scrolls down further and refers again to the following notATest() test method code:  
  
  
   public void notATest()  
   {  
       fail();  
   }*   
  
So let's say you want to keep it as it is, do so. It will run just fine. There's no reason you have to rewrite the tests. But if you know you're going to be doing big work on this test and you want to update it, it's not hard to update to the new JUnit. So first of all, we just, hey, we don't have to do anything right there. We can drop that out, we can leave that as it is. So at that point we don't need this second constructor anymore, we would tell the name of the class. We can take that one out. We don't have an Override anymore. We, instead, use the Setup annotations. I'm sorry the Before annotation. And that I will just get this one it is ready to go before hand. It can't throw an exception inside of there. But we can change that there, instead of tearDown we do the After annotation, which is equivalent of the tearDown inside of there. And so we could switch that over very easily and have that look like that. Now instead of the super here we have to use our asserts that are from our -   
*Back at the top of the OldSchoolTest.java file, the presenter begins to run through the code from the top down, modifying it to the updated JUnit framework. The modified OldSchoolTest.java code is currently as follows:  
  
package myapp.junit.old;  
  
import org.junit.After;  
import org.junit.Before;  
  
import junit.framework.TestCase;  
  
public class OldSchoolTest  
{  
   public OldSchoolTest()  
   {  
       super();  
   }  
  
   @Before  
   protected void setUp()  
   {  
       System.out.println("Run before each test");  
   }  
  
   @After  
   protected void tearDown()  
   {  
       System.out.println("Run after each test");  
   }  
  
   public void testSample()  
   {  
       System.out.println("Test Sample");  
         
       super.assertSame("Both are the same", "test", "test");  
       super.assertNotSame("Not the same", "test", new String("test"));  
  
       super.assertEquals("Both are the equals", "test", "test");  
   }  
  
   public void testMore()  
   {  
       System.out.println("Test More");  
         
       super.assertNull("Null", null);  
       super.assertNotNull("Not null", "test");  
         
       super.assertFalse("This is false", false);  
       super.assertTrue("This is true", true);  
   }  
     
   public void testError()  
   {  
       System.out.println("Test Error");  
       if (true)  
       {  
           fail("This was a bad decision");  
       }  
   }  
     
   public void notATest()  
   {  
       fail();  
   }  
}*   
  
other framework as our static import. So again, we have our static import, inside of here. Now I can do a static import of all of them at this point, and save myself a little bit of trouble, and then take out all the super calls. So you can see very easily I can take my framework and redo it as being, oops, yeah there you go, as being a modern JUnit test. Now it will take a little bit time to get all up and working. And we're not going to try and fix all of that right here in front of you. But just know it is as simple as that. It's most of those things have very simple move over equivalents. And then I run it just the same I was running in the class before. The same test runner, everything will run just the same. So whether you choose to leave them as the extended test cases or whether you want to move them in the annotation role it's up to you. But know that you can do that easily or keep them as they are and still use them all within JUnit tests.   
*The presenter continues to modify the oldSchoolTest code to update it to the new JUnit framework. The final code is as follows:  
  
package myapp.junit.old;  
  
import static junit.framework.Assert.\*;  
  
import org.junit.After;  
import org.junit.Before;  
  
import junit.framework.TestCase;  
  
public class OldSchoolTest  
{  
   public OldSchoolTest()  
   {  
       super();  
   }  
  
   @Before  
   protected void setUp()  
   {  
       System.out.println("Run before each test");  
   }  
  
   @After  
   protected void tearDown()  
   {  
       System.out.println("Run after each test");  
   }  
  
   public void testSample()  
   {  
       System.out.println("Test Sample");  
         
       assertSame("Both are the same", "test", "test");  
       assertNotSame("Not the same", "test", new String("test"));  
  
       assertEquals("Both are the equals", "test", "test");  
   }  
  
   public void testMore()  
   {  
       System.out.println("Test More");  
         
       super.assertNull("Null", null);  
       super.assertNotNull("Not null", "test");  
         
       super.assertFalse("This is false", false);  
       super.assertTrue("This is true", true);  
   }  
     
   public void testError()  
   {  
       System.out.println("Test Error");  
       if (true)  
       {  
           fail("This was a bad decision");  
       }  
   }  
     
   public void notATest()  
   {  
       fail();  
   }  
}*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Legacy JUnit Test Suites

Learning Objective

*After completing this topic, you should be able to*

* *configure JUnit test suites for legacy code*

**1. Configuring test suites for legacy code**

If your project has existing chain of tests, they may predate the most recent uses of annotations within JUnit. Test suites have always existed in JUnit, but have changed quite a bit from the legacy to the current approach. If you deal with legacy test suites, it may be useful to see how they can be used or converted into their modern equivalent. So here's an OldSchool test case, a TestSuite. The TestSuite here is a class that can be extended out by you to create a suite of tests. And so – very simply – I have a static method inside of here, called Suite, that returns a test. And the way I do that is I use this test suite class that constructs together, and you can add into that a bunch of different tests. So as I add in tests, I can choose – from an individual class – the name of the test I want to execute. So the constructor in JUnit and OldSchoolTests allow me to pass in a string, and says "hey, testSample". If I go the OldSchoolTest, you can see there is a method inside of there called testSample. And so when I call the suite, it's going to go execute this, this and only this, test cases I've added it in. I can then add another one, or I can just simply add in all the tests inside of a class. I return that as a single test, and it gets run as a suite, as I run within my runner. So I can run this; and you can see it, testSample, testMore, and it tests all of the tests inside of another OldSchoolTest. So it gives you all this combined stuff that's running.   
*In the Eclipse development environment, three files are open on separate tabbed pages in the editor – OldSchoolSuite.java, OldSchoolTest.java, and AnotherOldSchoolTest.java.  
  
The OldSchoolSuite.java file contains the following code:  
  
package myapp.junit.old;  
  
import junit.framework.Test;  
import junit.framework.TestSuite;  
  
public class OldSchoolSuite extends TestSuite  
{  
   public static Test suite()  
   {  
       TestSuite suite = new TestSuite();  
         
       suite.addTest(new OldSchoolTest("testSample"));  
       suite.addTest(new OldSchoolTest("testMore"));  
       suite.addTestSuite(AnotherOldSchoolTest.class);  
  
       return suite;  
   }  
}  
  
The presenter highlights the code extends TestSuite in the line:  
  
public class OldSchoolSuite extends TestSuite  
  
Next the presenter highlights the following line of code:  
  
   public static Test suite()  
  
He highlights the following TestSuite class:  
  
       TestSuite suite = new TestSuite();  
  
The presenter refers to the OldSchoolTest class and the test name "testSample" in the following line of code:  
  
       suite.addTest(new OldSchoolTest("testSample"));  
  
Next the presenter switches to the OldSchoolTest.java file. It contains the following code:  
  
package myapp.junit.old;  
  
import junit.framework.TestCase;  
  
public class OldSchoolTest extends TestCase  
{  
   public OldSchoolTest()  
   {  
       super();  
   }  
  
   public OldSchoolTest(String name)  
   {  
       super(name);  
   }  
  
   @Override  
   protected void setUp() throws Exception  
   {  
       System.out.println("Run before each test");  
   }  
  
   @Override  
   protected void tearDown() throws Exception  
   {  
       System.out.println("Run after each test");  
   }  
  
   public void testSample()  
   {  
       System.out.println("Test Sample");  
         
       super.assertSame("Both are the same", "test", "test");  
       super.assertNotSame("Not the same", "test", new String("test"));  
  
       super.assertEquals("Both are the equals", "test", "test");  
   }  
  
   public void testMore()  
   {  
       System.out.println("Test More");  
         
       super.assertNull("Null", null);  
       super.assertNotNull("Not null", "test");  
         
       super.assertFalse("This is false", false);  
       super.assertTrue("This is true", true);  
   }  
     
   public void testError()  
   {  
       System.out.println("Test Error");  
       if (true)  
       {  
           fail("This was a bad decision");  
       }  
   }  
     
   public void notATest()  
   {  
       fail();  
   }  
}  
  
The presenter scrolls down the OldSchoolTest.java file and highlights the testSample variable in the following line of code:  
  
   public void testSample()  
  
The presenter then returns to the OldSchoolSuite.java file. Again, it contains the following code:  
  
package myapp.junit.old;  
  
import junit.framework.Test;  
import junit.framework.TestSuite;  
  
public class OldSchoolSuite extends TestSuite  
{  
   public static Test suite()  
   {  
       TestSuite suite = new TestSuite();  
         
       suite.addTest(new OldSchoolTest("testSample"));  
       suite.addTest(new OldSchoolTest("testMore"));  
       suite.addTestSuite(AnotherOldSchoolTest.class);  
  
       return suite;  
   }  
}  
  
The presenter refers to the following segment of code:  
  
       suite.addTest(new OldSchoolTest("testSample"));  
       suite.addTest(new OldSchoolTest("testMore"));  
       suite.addTestSuite(AnotherOldSchoolTest.class);  
  
       return suite;  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.old.OldSchoolTest and its runtime is 0.002 seconds. The top-level test node is expanded and the following three items and associated runtimes are listed below it – testSample (0.001 s), testMore (0.001 s), and myapp.junit.old.AnotherOldSchoolTest (0.000 s). The myapp.junit.old.AnotherOldSchoolTest child test node is also expanded and the following test methods and associated runtimes are listed below it – testMore (0.000 s) and testSample (0.000 s).*   
  
Now, this worked really well, but it was a lot of coding. It was a lot of individual tweaking of things to have to pickup. And so the new way of using annotations has a lot less of that going on. Now, you may remember, or you may have seen before – what we need to do to make it new is, we need to add in the RunWith annotation, and say "hey, run it as a suite". So in the OldSchool mechanism, we created it as a TestSuite. That told it to run it as a suite. In the new mechanism, I switch it over to be a suite class and I don't need to extend out test case anymore – don't need that at all. Now, what I do need to specify is the SuiteClasses that are going to be a part of this. You can see I'm going to test with the OldSchoolTest; I'm going to test with the AnotherOldSchoolTest. These are both unmodified, unchanged extends TestCase tests that are out there. But I can also add in new test cases. So TestTarget, in this case, is something that's entirely annotation based. It doesn't matter, JUnit doesn't care, it doesn't require you to specify this is old versus this is new; it can see all of these guys. And at this point, this test suite means nothing. I'm just going to comment it out so you can see I don't need any of this to be in here. I can now run this new test suite; and there is good news and bad news about that. The good news is it runs all of these old and new classes together, you can see OldSchoolTest, AnotherOldSchoolTest, as well as the new annotations based one.   
*In the OldSchoolSuite.java file, the presenter begins modifying the code to update it to the new way of using annotations. The modified OldSchoolSuite.java code is as follows:  
  
package myapp.junit.old;  
  
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;  
import org.junit.runners.Suite.SuiteClasses;  
  
import junit.framework.Test;  
import junit.framework.TestSuite;  
  
@RunWith(Suite.class)  
@SuiteClasses({OldSchoolTest.class, AnotherOldSchoolTest.class, TestTarget.class})  
public class OldSchoolSuite  
{  
   public static Test suite()  
   {  
       TestSuite suite = new TestSuite();  
         
       suite.addTest(new OldSchoolTest("testSample"));  
       suite.addTest(new OldSchoolTest("testMore"));  
       suite.addTestSuite(AnotherOldSchoolTest.class);  
  
       return suite;  
   }  
}  
  
The presenter refers to the unmodified OldSchoolTest and AnotherOldSchoolTest classes in the following line of code:  
  
@SuiteClasses({OldSchoolTest.class, AnotherOldSchoolTest.class, TestTarget.class})  
  
The presenter switches back to the OldSchoolTest.java file, which, again, contains the following code:  
  
package myapp.junit.old;  
  
import junit.framework.TestCase;  
  
public class OldSchoolTest extends TestCase  
{  
   public OldSchoolTest()  
   {  
       super();  
   }  
  
   public OldSchoolTest(String name)  
   {  
       super(name);  
   }  
  
   @Override  
   protected void setUp() throws Exception  
   {  
       System.out.println("Run before each test");  
   }  
  
   @Override  
   protected void tearDown() throws Exception  
   {  
       System.out.println("Run after each test");  
   }  
  
   public void testSample()  
   {  
       System.out.println("Test Sample");  
         
       super.assertSame("Both are the same", "test", "test");  
       super.assertNotSame("Not the same", "test", new String("test"));  
  
       super.assertEquals("Both are the equals", "test", "test");  
   }  
  
   public void testMore()  
   {  
       System.out.println("Test More");  
         
       super.assertNull("Null", null);  
       super.assertNotNull("Not null", "test");  
         
       super.assertFalse("This is false", false);  
       super.assertTrue("This is true", true);  
   }  
     
   public void testError()  
   {  
       System.out.println("Test Error");  
       if (true)  
       {  
           fail("This was a bad decision");  
       }  
   }  
     
   public void notATest()  
   {  
       fail();  
   }  
}  
  
The presenter highlights TestCase in the following line of code:  
  
public class OldSchoolTest extends TestCase  
  
The presenter then switches to the AnotherOldSchoolTest.java file. It contains the following code:  
  
package myapp.junit.old;  
  
import junit.framework.TestCase;  
  
public class AnotherOldSchoolTest extends TestCase  
{  
   public AnotherOldSchoolTest(String name)  
   {  
       super(name);  
   }  
  
   public void testSample()  
   {  
       System.out.println("Test Sample");  
         
       super.assertSame("Both are the same", "test", "test");  
       super.assertNotSame("Not the same", "test", new String("test"));  
  
       super.assertEquals("Both are the equals", "test", "test");  
   }  
  
   public void testMore()  
   {  
       System.out.println("Test More");  
         
       super.assertNull("Null", null);  
       super.assertNotNull("Not null", "test");  
         
       super.assertFalse("This is false", false);  
       super.assertTrue("This is true", true);  
   }  
}  
  
The presenter refers to the code TestCase in the following line:  
  
import junit.framework.TestCase;  
  
The presenter returns to the OldSchoolSuite.java file. He further modifies the code by adding a new TestTarget case. The modified OldSchoolSuite.java code is now  
  
package myapp.junit.old;  
  
import myapp.junit.TestTarget;  
  
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;  
import org.junit.runners.Suite.SuiteClasses;  
  
import junit.framework.Test;  
import junit.framework.TestSuite;  
  
@RunWith(Suite.class)  
@SuiteClasses({OldSchoolTest.class, AnotherOldSchoolTest.class, TestTarget.class})  
public class OldSchoolSuite  
{  
   public static Test suite()  
   {  
       TestSuite suite = new TestSuite();  
         
       suite.addTest(new OldSchoolTest("testSample"));  
       suite.addTest(new OldSchoolTest("testMore"));  
       suite.addTestSuite(AnotherOldSchoolTest.class);  
  
       return suite;  
   }  
}  
  
The presenter hovers the cursor over TestTarget in the following code line:  
  
@SuiteClasses({OldSchoolTest.class, AnotherOldSchoolTest.class, TestTarget.class})  
  
TestTarget turns into a link and the presenter clicks it. A new file named TestTarget.java opens in the editor. In the file, the following code is visible:  
  
package myapp.junit;  
  
import junit.framework.TestCase;  
  
public class TestTarget  
{  
@Test  
public void testAdd()  
{  
MyTarget target = new MyTarget();  
  
int answer = target.basicAdd(2, 2);  
TestCase.assertEquals("Two plus two is four right?", 4, answer);  
}  
  
@Test  
public void badTestCase()  
{  
MyTarget target = new MyTarget();  
  
int answer = target.basicAdd(2, 2);  
TestCase.assertEquals("Adding is fun", 6, answer);  
}  
  
@Test(expected=ArithmeticException.class)  
public void testDivideByZero()  
  
The presenter highlights the following segment of code:  
  
@Test  
public void testAdd()  
{  
  
Next the presenter switches to the OldSchoolSuite.java file. The presenter further modifies the code by commenting out the Test Suite. The modified code is now  
  
package myapp.junit.old;  
  
import myapp.junit.TestTarget;  
  
import org.junit.runner.RunWith;  
import org.junit.runners.Suite;  
import org.junit.runners.Suite.SuiteClasses;  
  
import junit.framework.Test;  
import junit.framework.TestSuite;  
  
@RunWith(Suite.class)  
@SuiteClasses({OldSchoolTest.class, AnotherOldSchoolTest.class, TestTarget.class})  
public class OldSchoolSuite  
{  
//    public static Test suite()  
//    {  
//        TestSuite suite = new TestSuite();  
//          
//        suite.addTest(new OldSchoolTest("testSample"));  
//        suite.addTest(new OldSchoolTest("testMore"));  
//        suite.addTestSuite(AnotherOldSchoolTest.class);  
//  
//        return suite;  
//    }  
}  
  
The presenter clicks the Run button.  
  
In the JUnit view, the top-level test node is myapp.junit.old.OldSchoolSuite and its runtime is 0.000 seconds. The test progress details show that 9/9 tests ran, 1 was skipped, and 2 failed. The top-level test node is expanded and the following three tests are listed below it – myapp.junit.old.OldSchoolTest, myapp.junit.old.AnotherOldSchoolTest, and myapp,junit.TestTarget. The OldSchoolTest child node is expanded and the following test methods and associated runtimes are listed below it – testError (0.000 s), testMore (0.000 s), and testSample (0.000 s). The testError method failed and the testMore and testSample methods ran successfully.   
  
In the OldSchoolSuite.java file, the presenter refers to the classes listed in the following segment of code:  
  
@SuiteClasses({OldSchoolTest.class, AnotherOldSchoolTest.class, TestTarget.class})  
public class OldSchoolSuite  
  
In the JUnit view, the presenter refers to the OldschoolTest, AnotherOldSchoolTest, and the newer TestTarget test.*   
  
The bad news though, is I have to run literally all of the tests in each of those class. So before, in OldSchoolTest, I could skip the error test that was failing. I can't do that within the suite – I would have to choose categories for that, and thus I can tweak it around to be able to do that. Now categories, unfortunately, won't work within OldSchoolTests. It's not going to pick and choose within OldSchoolTests, but I can choose to convert those as well. So if you have a system that has a lot of legacy test suites out there, you can see it's a pretty simple feat to switch those over to new test suites, and to be able to include those together. Or you could simply include the old test suite as part of your new test suite, and bundle it that way as well. However you choose to do that, don't be afraid to keep the OldSchoolTests the way they are, and include them into your existing test suites, and not have to feel like you have to change them all immediately. Upgrading JUnit is a fairly, pretty painless process, as it's highly backwards compatible. So take advantage of that within your old tests and building new tests, in the best and latest and greatest technologies.   
*In the JUnit view, the presenter expands the myapp.junit.TestTarget child test node to reveal the following test methods and associated runtimes – testDividedByZero (0.000 s), testAdd (0.000 s), badTestCase (0.000 s), and SillyTest (0.000 s). The testDivideByZero and testAdd methods ran successfully. The badTestCase method failed and the sillyTest was skipped.  
  
Still in the JUnit view, the presenter expands the myapp.junit.old.AnotherOldSchoolTest child test node to reveal the following test methods and associated runtimes – testMore (0.000 s), testSample (0.000 s).  
  
In the OldSchoolSuite.java file, the presenter refers to the following commented out code:  
  
//        suite.addTest(new OldSchoolTest("testSample"));  
//        suite.addTest(new OldSchoolTest("testMore"));  
  
Finally, in the JUnit view, the presenter selects the testError method below the myapp.junit.old.OldSchoolTest child test node.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

Exercise: Modify Java Code in Test Scenarios

Learning Objective

*After completing this topic, you should be able to*

* *practice modifying existing Java code to accomplish proper testing scenarios*

**1. Modifying Java code in testing scenarios**

This exercise will let you practice your new skills.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a02_it_enus/output/html/course_transcript.html#top)

© 2017 Skillsoft Ireland Limited