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

**Course Transcript**

JUnit Fundamentals

**Getting Started with JUnit**

[1. Course Introduction](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t2)

[2. Unit Testing with JUnit](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t6)

[3. Setting up JUnit in Eclipse](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t10)

[4. Defining @Test](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t14)

[5. Running from Java Code](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t18)

**Assertions**

[1. General Assertions](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t23)

**Equality Asserts**

[1. Number Equality](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t28)

[2. Object Equality](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t32)

[3. Array Equality](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t36)

**AssertThat statements**

[1. Basic assertThat Options](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t41)

[2. Combination assertThat Statements](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t45)

[3. Collections and assertThat](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t49)

[4. Custom assertThat Matchers](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t53)

**Assume Statements**

[1. Assume versus Assert](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t58)

**Setting up Test Cases and Fixtures**

[1. Setting up Test Cases](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t63)

[2. Test Fixtures](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t67)

**Exercise: Modify existing java code to accomplish proper testing scenarios**

[1. Modifying Java Code to Ensure Proper Testing](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#t109)

Course Introduction

Learning Objective

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

* *start the course*

**1. Introduction to the course**

JUnit gives us an early check for defects in code. But conducting unit tests can be slow and painful, particularly if you need to repeat testing. I'm Tony Lowe a Java architect with over fifteen years experience in building and teaching IT solutions. JUnit is designed to help alleviate some of the pain by providing a simple framework for driving unit tests and executing them again and again over time. It doesn't get rid of the need to plan or maintain tests entirely, but it does give you a solid foundation to build your tests. In this course, we will look at how JUnit helps organize and execute your test cases.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Unit Testing with JUnit

Learning Objective

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

* *understand the basics of unit testing with JUnit*

**1. Unit testing and JUnit**

The expectation for quality in software is ever-rising, only to be rivaled by the increasing complexity of solutions and demand for quicker development cycles. In order to meet this demand, we need to increase our rigor and test coverage, and also productivity in executing tests. Unit testing gives us an early look at quality, and JUnit gives us  automation and repeatability to allow us more testing more often. So looking into the idea of unit testing, unit testing by its very nature is breaking down a system into its components. So if I look in front of me right now, I have a pretty complex computer system that needs to be defined by its individual components. So if I wanted to play a video, and I come up to my computer system, and, say, my video is not making any sound, there’s a lot of different places I might be looking for the sound not coming through. So the first question is which of these components are working and which isn't working? So is my router working properly? Am I getting an Internet connection? Is my computer receiving information? Is my computer playing the video I am trying to look at? Then, is my computer sending a signal off to my monitor? Is it sending a signal off to my receiver? And then, is my receiver then continuing signal on to the speakers and making a sound?   
*Heading: System Debugging is Hard  
  
A diagram shows a computer, a monitor, a receiver, a router, and speakers.*  
  
This debugging of the system could be really hard if I don't know where I can check, and which pieces are working and which pieces aren't working. I need to have multiple looks at each piece, and know when which pieces are working and which pieces aren't working. And even more, each of these pieces are made up of sub-pieces themselves. And each of these units need to be each working perfectly in order for my entire system to work as expected. And so what I would do to increase my understanding of my system is I would add unit tests around each of my pieces. So I would test my router – is my Internet connection working? I would test my computer – hey, are the basic features running? Is the operating system up and running? And then hey, is the Internet working? I would test the monitor – hey, turn it on, can I receive the signal from the computer? The receiver – is that working by itself? Maybe I wouldn't plug in the computer at all, I would plug in a different device. I would use the radio on it just to test the receiver is working. And then the speakers as well – are they making sound? And unit tests wrap around each component individually to ensure and make sure that they are making sense. So the development of these tests might also include integration tests and functional tests. As far as my test planning goes, I want to test each component on its own, and I want to test these interconnections between them. So where JUnit comes in is I have my unit under test; and it could be a straight unit test, it could be an integration test, or some other test of that nature.   
*Heading: System Debugging is Hard  
  
In the diagram that shows a computer with a monitor, receiver, a router, and speakers, a list alongside the router includes the items configuration, modem, cabling, and ISP. A list alongside the computer contains the items sound card, video card, motherboard, connectors, and software.  
  
Heading: Unit Testing Checks Components  
  
A diagram contains five labels – Router, Computer, Monitor, Receiver, and Speakers. A line connects the Computer label to the Monitor label. Each label is added to a separate box titled Unit Tests.  
  
A box that connects to the link between the Computer and Monitor labels displays the text "Integration Tests and Functional Tests focus on these."*   
  
And so I am going to have my test planning and my test data – that's where I am going to start with this. I want to understand what am I trying to do with this. I add inside of there automated tests, and that's where my JUnit is helping. I am actually going to be writing code, and that code can come in many forms, but it's going to be something that's going to drive my solution. It doesn't matter if it's a web solution or an android solution, or it could just simply very well be some tiny, little class or two that I am trying to test with here. I am going to found that all within the JUnit framework though. It's going to help me with executing tests, with executing validations inside of tests, and also with results – capturing the results and sending those on – and then all that's going to run within either my interactive development environment – I may be using Eclipse or NetBeans or IntelliJ – or I could be using a continuous integration solution that's going to be running every time I do a build. So I might be using Maven to build my solution, or I could be using Hudson, or one of those other tools that's going to go off and its going to build my code, execute tests, and then give me a report for that all the time. So the idea behind all of this is I want to constantly be testing. I want to use JUnit to automate those tests, and then allow me to get better quality through each change in my system. And within all that, I am going to be able to improve my testing cycles and improve the time it take to do my testing cycles.   
*Heading: JUnit Automation  
  
The labels UUT Software, Test Plans/Data, and Automates Test Cases are shown within a box titled Unit Tests.  
  
Test Plans/Data is linked to the label "Start with planned tests" and Automated Test Cases is linked to the label "Code tests w/ JUnit help."  
  
A box labeled JUnit Framework links to the label "to Help more with tests/execution results."  
  
A box labeled Interactive Development Environment Continuous Integration Solution links to the label "Provide a platform to execute tests."*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Setting up JUnit in Eclipse

Learning Objective

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

* *set up JUnit in Eclipse*

**1. Setting up the JUnit shell in Eclipse**

JUnit provides us a platform for making reusable tests, while Eclipse gives us helpful tools for developing code. So it's only natural for us to integrate both of these tools. We can take advantage of Eclipse to help create and run all of these great JUnit tests we are going to build. So let's look at how we can integrate these two. So we have basic Eclipse environment. There are some projects going on and everything is ready, but we are going to start with a brand new project here. So I am going to create a new Java project. So this is going to be JUnitSetupProject – just to show you some of the things we can do to set up within JUnit. And so that's all we need for now – it's just a plain basic Java project. There is just source and the basic JRE library ready to go. So the first thing we can sit here and show you is that we can add in the JUnit libraries we need very quickly through the build path. And so if we go to the **Libraries** tab inside of the build path, and actually go into **Add Library**, we can see inside of Eclipse, **JUnit** is already setup as a library and it's already downloaded with Eclipse. We don't have to go download and find any .jar files within that. So I can select that – that will even allow me to select which version of JUnit I want to run. I am going to run with the most recent for now, which is **4**. You can see that .jar file is a part of my Eclipse build, right there. Okay, so we can see we have an integrated JUnit inside of Eclipse.   
*The Eclipse development environment includes a pane that's currently open on the Package Explorer; a central code editing pane, which is currently blank; and a pane with five tabs – Problems, Javadoc, Declaration, Search, and Console. The Package Explorer currently lists four entries – Basics, Database, Maven, and zzzLibraries.  
  
The presenter selects File - New - Java Project. The New Java Project dialog box contains a Project name text box, a Use default location checkbox – which is selected, and a Location text box with a Browse button. It also contains options in JRE, Project layout, and Working sets sections.  
  
The presenter types in JUnitSetupProject in the Project name text box and clicks Finish.  
  
The new project, JUnitSetupProject, is listed in Package Explorer. The presenter expands the node for it to show two child nodes – src and JRE System Library.  
  
The presenter right-clicks the JUniteSetupProject node and selects Properties from the shortcut menu that opens.  
  
The Properties dialog box for the project opens on the Java Build Path page. The page contains four tabs – Source, Project, Libraries, and Order and Export. The Libraries tabbed page is open by default. It contains a pane that currently lists the JRE System Library [JavaSE-1.7]. It also contains a set of buttons for adding JARs, variables, libraries, and class folders, and for editing the content of the pane.  
  
The presenter clicks the Add Library button. The Add Library Wizard lists available libraries. JUnit is already selected. The presenter clicks Next. A JUnit Library page lists the current and source locations of the library and contains a JUnit library version drop-down list box. The presenter selects JUnit 4 from the drop-down list and clicks Finish. In the Properties for JUnitSetupProject dialog box, he then clicks OK.*   
  
Now we can go even farther than this. Now that we have the build setup, so it's ready to build, I can come over here and right-click and say **New**. Let me come down to **Other** here. And underneath Other, we can see that there's a lot of options for what we can build. And there is a **JUnit** option because I selected it earlier, but that's normally hidden underneath your **Java** folder. So you can see the base folder is Java, and then underneath there is **JUnit**. And I can build either a **JUnit Test Case**or a Test Suite from this part. So I hit **Next**, and here is where I can select my Package, and then I can give it whatever name…whatever name I want to use inside of there. Now there's other options for adding in some of these basic methods I can define later on, and I can even point to a class I am going to be testing with this class if I so choose – it will actually help set up that class further. But for now, we don't have that class, so we are just going to get started.   
*The presenter right-clicks the src node in Package Explorer and selects New – Other from the shortcut menu. The New dialog box contains a Wizards text box, set by default to the prompt "type filter text", and a pane that lists available wizards.  
  
The presenter scrolls through the list, pointing out the top-level Java option, the JUnit node, and the two options below JUnit – JUnit Test Case and JUnit Test Suite. The presenter selects JUnit Test Case and then clicks Next.  
  
The New JUnit Test Case Wizard launches. It contains New JUnit3 test and New JUnit4 test radio buttons and Source folder and Package name text boxes, each with an associated Browse button. The Source folder text box is set to JUnitSetupProject/src. It contains a Name text box, a Superclass text box set to java.lang.Object, and checkboxes for choosing which method stubs to create – including setUpBeforeClass(), setUp(), constructor, tearDownAfterClass(), and tearDown(). It also contains a Generate comments checkbox, and a Class under test text box with a Browse button.  
  
The presenter types sample in the Package text box and FirstUnit in the Name text box.*   
  
So here is our basic JUnit test. Now building a test that way – very limited use – it only does a couple of little things for us, the first of which is to give us our first sample test case. And we will learn more about these as we go. But the test is…the test annotation is what defines this as being a test. And then it gives us this import here. So imports our junit.Test tag and it imports the static import for Asserts. So what that does for us is there is this class out there called Assert – and I will import that here real quick, just to give you a feel for that – that has all these static methods that allow us to do JUnit calls. There's plenty of these calls here to help us do our testing. Now what the static insert allows to do, instead of having to call something like assertTrue underneath the call of a static test and I have to put Assert.assertTrue every single time, by creating this static import, I can call it without that. I can just simply say assertTrue(true), so whatever test would go inside of there…and then I don't even need to import that class – these are always going to be available.   
*The presenter clicks Finish. In Eclipse, the file named FirstUnit.java is open. It contains the following default code:    
  
package sample;  
  
import static org.JUnit.Assert.\*;  
  
import org.JUnit.Test;  
  
public class FirstJUnit  
{  
  
   @Test  
   public void test()  
   {  
       fail("not yet implemented");  
   }  
  
         
}  
  
Above the line fail("not yet implemented");, the presenter types Assert. A drop-down list automatically provides a long list of suggestions for completing the code, including methods like assertArrayEquals, assertEquals, assertNotEquals, assertSame, and assertNull. In each case, the methods are listed along with their variables.  
  
The presenter types assertTrue after the keyword Assert. This causes the system to cross out the keyword and add an input field for typing a condition after the assertTrue method.  
  
The presenter then deletes the line and simply types  
  
assertTrue(true);  
  
The presenter also then deletes the line  
  
import static org.JUnit.Assert.\*;  
  
In full, the code is now  
  
package sample;  
  
import org.JUnit.Test;  
  
public class FirstJUnit  
{  
  
   @Test  
   public void test()  
   {  
   assertTrue(true);  
       fail("not yet implemented");  
   }  
  
         
}*   
  
Now one extra thing you are going to want to set up inside of here to even help you further, is to set up your context assist appropriately. So if I go up to the **Window** and go to **Preferences** inside of here, underneath the Preferences you have your **Java** preferences; underneath the **Editor**, **Content Assist**, and **Favorites**, there's these options to be able to have these classes that I will automatically include as static imports. So what you would do, if you don't have these already, is you would hit**New Type** and then **Browse**. And you pick whatever class you want to; for JUnit, you want the **Assert** class – you can see we have selected it before, so it shows up as a memory one – and then we go through and we add that in. And then when we add that in, you can see it adds in **junit.framework.Assert.\***. This other guy here from the Hamcrest framework is another great little tool we will use eventually. So these two are things that I always set up when I am going to JUnit. And that will help me do something really cool, like assert, and it will automatically pop up all these assert options. And this is all stuff from that assert class or the hamcrest class, whatever you do inside of there. So it's another great way I can integrate JUnit and Eclipse together.   
*The presenter selects Window - Preferences. The Preference dialog box includes a pane with a tree structure for navigating to the required category of preferences. Currently Java - Editor - Content Assist - Favorites is selected. In the main pane, the Favorites page includes a pane that lists static members and types that contain static members. It also contains four buttons – New Type, New Member, Edit, and Remove.  
  
By default, the pane lists two items – JUnit.framework.Assert and org.hamcrest.CoreMatches.  
  
The presenter clicks New Type. The New Type dialog box contains an Enter a fully qualified type text box and a Browse button. The presenter clicks Browse and the Type Selection dialog box opens. It contains a Choose type name text box and a pane that lists available types. The presenter types Assert in the Choose type name text box and then selects the Assert - JUnit.framework option from those listed. He then clicks OK in the Type Selection and New Type dialog boxes.  
  
The presenter closes the Preferences dialog box. In the FirstUnit.java file, he positions the cursor below the line assertTrue(true); and starts typing the word "assert. A drop-down list automatically lists various methods that start with assert. The presenter then deletes the word.*   
  
So now I have my test. The final great feature of the JUnit inside of here is I can run directly from my tools. So if I go through and I hit Run…okay, it's going to save and launch that…it's going to automatically launch this as a JUnit test. And over here, you can see it ran it within a JUnit test. And so over here I get the results. So my test failed, and it failed on this line because of the assertion "Not yet implemented" – that's this guy right here. So again, I am testing for true…let's make sure that's going…I take that line out, I run it again…oh look, it passed. And if I had a test suite, it would show me all the tests that passed in the suite. And if I had multiple tests – so I could, let's say, add a second test inside of here – just say I have a test2 and I **Run** this guy, it shows me all the tests underneath, all the tests that I passed. What it did just right there – see all the tests – if I click on an individual test inside here and run that…like this, I click on the name and **Run** it…it only runs that test. It doesn't run all the tests inside of the suite. So if you are just trying to perfect that one test, Eclipse is very smart to allow you to do testing. At this point, your tools are ready to go. Now you can go off and do some testing.   
*The presenter clicks the Run button. A Save and Launch dialog box lists the FirstUnit.java file and the presenter clicks OK.  
  
A JUnit pane displays the results, which show that the test failed.  
  
The presenter deletes the code line  
  
fail("not yet implemented");  
  
He clicks Run again and this time, the test passes.   
  
The presenter adds a second test to the code. The full code is now  
  
package sample;  
  
import org.JUnit.Test;  
  
public class FirstJUnit  
{  
  
   @Test  
   public void test()  
   {  
   assertTrue(true);  
       fail("not yet implemented");  
   }  
  
   @Test  
   public void test2()  
   {  
   assertTrue(true);  
       fail("not yet implemented");  
   }  
         
}  
  
The presenter clicks Run again and both tests – listed as test and test2 – are listed as having passed.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Defining @Test

Learning Objective

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

* *understand how to use the @Test annotation*

**1. Using the @Test annotation in Eclipse**

Defining tests in JUnit takes advantage of annotations to allow us the most flexible implementation. The test annotation is the foundation of our testing, marking any method as a test case. Let's look at how we can use this annotation to drive our JUnit tests. In front of us here, we have a existing test class, we have some target we're going to go off and test, there is this other class out there and we can actually take a look at that, that has an Add method and a Divide method. It does whatever it's going to, it can have any complexity of things inside of here and in our case it's pretty simple. Now, we have any number of methods we could add into this class, what makes a method a test case is this Test annotation, this is coming out of our JUnit library, and we have this annotations imported and included as part of our test. Now, this does nothing to the functionality, it just simply flags this so when this class is executed as a JUnit suite, it pulls out anything with the Test annotation and it runs that as a test case. So, in our method here, in our class here, excuse me, we can see I have a add test case, I have a badTestCase here, I have a method here that's not a test case, you can add any number of these inside of there, and then two more tests here down at the bottom.   
*In the Package Explorer in Eclipse, the myapp.junit node is expanded and the TestTarget.java file is selected. The code editing pane contains the following code:   
  
package myapp.junitJUnit;  
  
import junitJUnit.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);  
       }  
  
public void someOtherMethod()  
       {  
               Throw new RuntimeException();  
       }  
         
}  
       @Test(expected=ArithmeticException.class)  
       public void testDivideByZero()  
       {  
               MyTarget target = new MyTarget();  
                 
               int answer = target.basicDivide(100, 0);  
       }  
         
       @Test  
       @Ignore  
       public void sillyTest()  
       {  
               TestCase.assertTrue(false);  
       }  
         
}*   
  
So, let's run this, and you can see, of these five methods I've declared inside of my class, only four of them run as test cases. And the last one actually doesn't run at all. So, the reason for that is it just picks up the ones with the annotations. If I wanted this to run as a test case, well I can do that, I just put an @Test annotation outside of there, when I save it up, it'll now detect this as a test inside of our suite of tests, inside of this class.So, when I run it, now I've five test cases. Now, notice, there is no naming convention required. It used to be in JUnit, a long time ago, before annotations, that all test cases had to start with the 4 letters T-E-S-T, that's no longer the case, here I say testAdd, but then I've a badTestCase, someOtherMethod. Down here, I have another one called testDivideByZero. I can have any number of methods inside of a class, one or a million, or a thousand, however much you want to write, that will create test cases inside of here. Now, this test case right here is failing, the one we just added here with someOtherMethod, it's failing because it throws an exception. Now, the expectations in test cases is if an exception was thrown, something bad happened. Any time it catches an exception within JUnit, it's going to fail the test case.   
*The presenter clicks Run. A JUnit tab, which opens automatically above the Package Explorer tab, lists four test cases under myapp.junit.testTarget. They are testDivideByZero, test Add, badTestCase, and sillyTest.  
  
In the code, the presenter adds the annotation @Test above the following code:  
  
public void SomeOtherMethod()  
  
       {    throw new RuntimeException ();  
       }  
  
The code is now:    
  
      @Test  
       public void SomeOtherMethod()  
       {  
            throw new RuntimeException ();  
       }  
  
The presenter clicks Run. Now five test cases are listed – testDivideByZero, test Add, badTestCase, sillyTest, and someOtherMethod.*   
  
Now, as we're testing our code, there are situations where DivideByZero is appropriate. Our Divide method inside of here, if we would provide a zero value as part of the divide, should throw a DivideByZero exception. It can't give you a number, if I divide something by zero, the result is infinity, I can't represent infinity inside of Java, so I return an exception instead. So, as we want to test this, we have to handle for an exception. The old way of doing this would be to do a try catch statement, to catch the exception and then pass the test appropriately from there. Now, that we have the annotation though, we can actually throw into the annotation the expected characteristic, and there is two actually here, expected and time out, expected is going to say I expect this type of exception to show up. An ArithmeticException or any general exception, any class is going to categorize any of those. When this exception shows up, when I DivideByZero, you can see I have a passed test case. If I'd said here, perhaps instead, a nullPointerException, let me change that off here, and I ran that as a test case, it would fail the test case, because I was expecting a DivideByZero exception, I got a nullPointerException instead. So, it's very clever, and it will make sure that not just any exception happens, but that specific exception happens.   
*The presenter highlights the code  
  
       @Test(expected=ArithmeticException.class)  
       public void testDivideByZero()  
       {  
               MyTarget target = new MyTarget();  
                 
               int answer = target.basicDivide(100, 0);  
       }  
  
The presenter selects excepted in the line @Test(expected=ArithmeticException.class) and a drop-down list opens. It contains the options expected: Class<? extends java.lang.Throwable> - Test and timeout: long - Test. With expected: Class<? extends java.lang.Throwable> - Test selected. In addition, a pop-up box provides the syntax for expected and an explanation, as follows:  
  
public abstract Class<? extends Throwable> expected  
  
Optionally specify expected, a Throwable, to cause a test method to succeed if an exception of the specified class is thrown by the method.  
Default:  
org.junit.Test.None.class  
  
The presenter changes the line @Test(expected=ArithmeticException.class) to  
  
@Test(expected=NullPointerException.class)  
       public void testDivideByZero()  
       {  
               MyTarget target = new MyTarget();  
                 
               int answer = target.basicDivide(100, 0);  
  
The presenter then clicks Run. Under myapp.junit.TestTarget, five tests are listed – testDivideByZero, test Add, someOtherMethod, badTestCase, and sillyTest. The test named testDivideByZero failed, returning an unexpected exception.*   
  
If I go back and put this back where it was, and go back to ArithmeticException, let's say I divide it by ten instead of zero, and I run this as our test case, here, that DivideByZero test is not working because it was expecting ArithmeticException and it didn't get it, it fails that test case, it's not appropriate there. Notice as well, the difference between, hey this is a big bad thing versus, just say, hey this test worked, but it failed. That when we get the exception otherwise it saying if something bad went wrong, it wants you to expect and to anticipate exceptions. So, that's a little bit about testing, using the Test annotation, and so go forth and test.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Running from Java Code

Learning Objective

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

* *run a JUnit test from Java code*

**1. Running a JUnit test from Java**

JUnit tests are built to be modular and can be executed from a variety of platforms. Let's take a look at how we can run JUnit tests from within any Java-based solutions, and even expand that to take advantage of parallel executions of tests. Inside of development, you'll run most of your tests from Eclipse. Past that though, you might want to run tests in different platforms. And the JUnitCore class gives us the runClasses method that let's us do that. This runs from a main classes, like you see here, or any other solution you want to run. So, this method takes any list of classes, it's taking advantage of that cool little Java ability to have multiple parameters passed in. If we take a look at these classes real quick, all they have to do is have a Test annotation inside of them, they don't subclass anything, they don't implement an interface, just it will look through this class for anything with the Test annotation. So, here is my for test counting 1 to 100, here is my while test counting 1 to 100. The only thing that's unique about them is the Test annotation. So, when I go and run this, it's going to run it, and it'll run one test after the other, the For test then the While test. And then it gives me some stats. I can get the stats by querying that class to say how many test did you run? How many tests did you ignore? How long did it take? So, in this case, it took 9 milliseconds, and then how many of these tests failed? And I just got lucky, none of them failed this time, and I can go through for any of the failures and print out what the failures were, if I did have any of those that came back.   
*In Eclipse, three Java files are open on separate tabs – ParallelRunner.java, ParrallelFor.java, and ParallelWhile.java.  
  
The ParallelRunner.java file contains the following code:   
  
package myapp.junit.parallel;  
  
import org.junit.experimental.ParallelComputer;  
import org.junit.runner.Computer;  
import org.junit.runner.JUnitCore;  
import org.junit.runner.Result;  
import org.junit.runner.notification.Failure;  
  
public class ParallelRunner  
{  
   public static void main(String[] args)  
   {  
       //Computer c = new ParallelComputer(true, true);  
       Result result = JUnitCore.runClasses( ParallelFor.class, ParallelWhile.class);  
         
       System.out.println("Ran " + result.getRunCount());  
       System.out.println("Ignored " + result.getIgnoreCount());  
       System.out.println("Time " + result.getRunTime());  
       System.out.println("Failures " + result.getFailureCount());  
         
       for (Failure failure : result.getFailures())  
       {  
           System.out.println(failure.toString());  
       }  
   }  
}  
  
The presenter switches to the ParallelFor.java file, which contains the code   
  
package myapp.JUnit.parallel;  
  
import org.junit.Test;  
  
public class ParallelFor  
{  
   @Test  
   public void runFor()  
   {  
       for(int i = 0; i < 100; i++)  
       {  
           System.out.println("For " + i);  
       }  
   }  
}  
  
The presenter then switches to the ParallelWhile.java file. It contains the following code:  
  
package myapp.JUnit.parallel;  
  
import org.junit.Test;  
  
public class ParallelWhile  
{  
   @Test  
   public void runWhile()  
   {  
       int i = 0;  
       while (i++ < 100)  
       {  
           System.out.println("While " + i);  
       }  
   }  
}  
  
The presenter returns to the ParallelRunner.java file and clicks Run. Below the file, the Console pane lists a series of 99 For tests and then 100 While tests. The final output is  
  
Ran 2  
Ignored 0  
Time 9  
Failures 0*   
  
Now, I can get fancier with this, I can go through and add what's called a Computer. Now, the Computer is a strategy for how do I execute these tests. And so I add the Computer as the first parameter here, you can see the second option is to have a Computer plus the class. Now, with the basic Computer from the JUnit library, here you can see JUnit.runner.computer, I can go through and run the tests and it does all of nothing different. Here is the For loop, runs to 1 to 99, and the While loop runs 1 to a 100 in this case. And so it ran these guys and exactly the same as without the Computer. I can change that though, by adding in the ability to do parallel computing, and I do that using the ParallelComputer class. The ParallelComputer class coming from the experimental part of JUnit, right now this is a test feature, but it allows me to define something that has two parameters that goes in, do I want to use classes, and then do I want to use methods inside of there? I want to test both of those guys, I want to go through both of these guys as I'm running these in parallel.   
*In the ParallelRunner.java file, the presenter removes the comment symbols from the line  
  
               //Computer c = new Computer (); //= new ParallelComputer(true, true). The code is now  
  
               Computer c = new Computer (); = new ParallelComputer(true, true).  
  
He also adds the variable c to the next line in the code, which is now  
  
       Result result = JUnitCore.runClasses(c, ParallelFor.class, ParallelWhile.class);(true, true);  
  
The full code in the file is now as follows:  
  
package myapp.junit.parallel;  
  
import org.junit.experimental.ParallelComputer;  
import org.junit.runner.Computer;  
import org.junit.runner.JUnitCore;  
import org.junit.runner.Result;  
import org.junit.runner.notification.Failure;  
  
public class ParallelRunner  
{  
   public static void main(String[] args)  
   {  
       Computer c = new //ParallelComputer(true, true);  
       Result result = JUnitCore.runClasses(c, ParallelFor.class, ParallelWhile.class);  
         
       System.out.println("Ran " + result.getRunCount());  
       System.out.println("Ignored " + result.getIgnoreCount());  
       System.out.println("Time " + result.getRunTime());  
       System.out.println("Failures " + result.getFailureCount());  
         
       for (Failure failure : result.getFailures())  
       {  
           System.out.println(failure.toString());  
       }  
   }  
}  
  
The presenter clicks Run and the Console pane lists a series of For tests, followed by a series of While tests, numbered from 1 to 100.*   
  
In this case, I have one method and I have one, you know, two classes and one method in each, it's going to run these two guys in parallel. So, now let's go through and run, and see the difference that's going on inside of there, I'll save it and watch it. And so now you can see immediately, it does not run all the for and then all the while, it does them both simultaneously. So, For and While, each kickoff and they are running pretty much in parallel the whole time, it's creating threads appropriately and it's doing everything it needs to do, it runs the test and it gives me the results when they are all set and done. So, if you have some massive test suite, that really takes a long time to go, you can use this parallel computing strategy to speed things up if you are going to run it by hand. And in many cases you will use some sort of continuous integration framework that will run your testing, but let's just say you have an app build or something simple, and it takes that really long time to execute all your tests, this is one way to work around it. You can create your own Java to run wherever you want to, a servlet, it could be a command-line, it could be however you wish to deploy this, but this is how JUnit can be executed and be able to run individual tests, groups of tests, or even groups of tests in parallel.   
*In the ParallelRunner.java file, the presenter removes the comment symbols from before the code ParallelComputer(true,true);  
  
The updated code line is now  
  
               Computer c = new ParallelComputer(true, true);  
The presenter clicks Run. This time, each For test is immediately followed by a While test.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

General Assertions

Learning Objective

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

* *understand JUnit assertions*

**1. Using assertions**

In order to automate tests we need to assure the environment is as expected before we test, and then afterwards the test results are as we also expected. JUnit provides many possible assertions, but we can start with a few that will form the foundations of most of our test case. Let's take a look. Now, a lot of the tests we're going to do, we're going to be comparing the results to an expected value. We'll use assertEquals or assertThat. But assertTrue and assertFalse are basic assertions that are going to help us know, is our environment ready to go. And then check the states of complex things that might not be directly expected outside of tests. So, in this case we have this little helper Weather class outside here. And the Weather class is going to give us three different answers, is it going to rain today, isItSunnyToday or isItFreezingToday?  And this might not even be the unit under Test, but we can use assertTrue and assertFalse to say, hey are we ready to Test in this case, this is one way it could be used.   
*In Eclipse, three Java files are open on separate tabbed pages – GeneralAssertions.java, Weather.java, and ErrorCollectorExample.java.   
  
The GeneralAssertions.java file contains the following code:  
  
package myapp.junit.asserts;  
  
import org.junit.Test;  
  
public class GeneralAssertions  
{  
   private Weather weather = new Weather(7);  
     
   @Test  
   public void showAssertTrue()  
   {  
       assertTrue("No Rainbow today", weather.isItSunnyToday() && weather.willItRainToday());  
       System.out.println("OK, test the rainbow stuff");  
   }  
  
   @Test  
   public void showAssertFalse()  
   {  
       assertFalse("Unsafe to drive", !(weather.isItFreezingToday() && weather.willItRainToday()));  
       System.out.println("Test the new vehicle");  
   }  
  
The presenter switches to the Weather.java file. It contains the code  
  
package myapp.junit.asserts;  
  
public class Weather  
{  
   private int answer;  
     
   public boolean willItRainToday()  
   {  
       return answer % 2 == 0;  
   }  
     
   public boolean isItSunnyToday()  
   {  
       return answer % 3 == 0;  
   }  
     
   public boolean isItFreezingToday()  
   {  
       return answer % 5 == 3;  
   }  
      
   public Weather(int answer)  
   {  
       super();  
       this.answer = answer;  
   }  
  
The presenter then navigates back to the  GeneralAssertions.java file.*   
  
So, in the first Test we only could do the Test if there is a rainbow. And so we're going to say, hey is it sunny and rainy today that's the only way we're get a rainbow. And so when we go through and run our Test here we can see our assertTrue test is not the case, it's not sunny and there's no rainbow out. So, both of those, either of those return false since we can't run the rainbow Test, but we can say, hey it was good, we see the assertFalse test came up positive so it's not freezing and raining. So, if it's raining it's okay, if it's freezing it's okay we can drive, but it's both freezing and raining we can't drive, so that's the case here. So we tested the new vehicle. Now, if we change the Weather here real quick, let's change the Weather and we run this again. We can see now it's not safe to drive. We can look here it says, hey Unsafe to Drive, AssertionError this Test did not run, but the second Test did run. So, you can see it was OK, to test the rainbow stuff. So we first asserted yes it's sunny and it's raining, and we went off and ran the Test so the Test passed.   
*With the GeneralAssertions.java file open, the presenter clicks the Run button. The JUnit tab lists three tests – showAssertFalse, showAssertTrue, and showFail. The presenter selects showAssertTrue and highlights the result  
  
java.lang.AssertionError: No Rainbow today  
  
In the Console pane, the output is the message Test the new vehicle.  
  
In the GeneralAssertions.java file, the presenter changes the code line private Weather weather = new Weather(7); to  
  
   private Weather weather = new Weather(6);.  
  
The presenter then clicks Run again. The JUnit tab lists the tests showAssertFalse, showAssertTrue, and showFail. With showAssertFalse selected, the results include the error message  
  
java.lang.AssertionError: Unsafe to drive  
  
A green icon with a tick indicates that the second test, showAssertTrue, ran successfully. With that test selected, the Console pane contains the message OK, test the rainbow stuff.   
  
The presenter changes the line private Weather weather = new Weather(6); back to  
  
private Weather weather = new Weather(7);*   
  
Now, applying this into your code as you design the Test you can use these assertions to say, hey is it as expected and you can put your own text inside of here to say what went wrong. It's not to say that the unit under Test failed. It's just saying what we expected for this Test to pass didn't happen. So ignore this Test results, we're not going to bother testing at this point. As we hit the Assert, if it's not true, we don't execute the rest of the code. You can see only the rainbow, the vehicle was not tested, it stopped at this point. The assertTrue and false can also be used after the test to say what the scenario was afterwards. So, there is no limitation to however this is called, it's just another example. Now, another use we can see inside of there is there is a Fail method. So the third method here that we have not talked about is, we're saying like if for whatever reason it's freezing today or it'll rain today we're going to say, hey the weather is bad and call off for plans. So again this little example inside of here, but right in this case the weather has been bad in each of these cases, it's either been raining or cold or both based off of the test up above. So, if we change the Weather one more time, then we'll make the weather a little bit nicer. As we go and execute this test, you can say we're no longer failing.   
*In the GeneralAssertions.java file, the presenter scrolls to the following code:  
  
   @Test  
   public void showAssertFalse()  
   {  
       assertFalse("Unsafe to drive", !(weather.isItFreezingToday() && weather.willItRainToday()));  
       System.out.println("Test the new vehicle");  
   }  
  
   @Test  
   public void showFail()  
   {  
       if (weather.isItFreezingToday() || weather.willItRainToday())  
       {  
           fail("Weather is bad, call off our plans");  
       }  
   }  
}  
  
The JUnit tab lists the three tests showAssertFalse, showAssertTrue, and showFail. WIth showFail selected, the following error message displays:  
java.lang.AssertionError: Weather is bad, call off our plans   
  
The presenter then navigates to the line of code private Weather weather = new Weather(6); and changes it to  
  
private Weather weather = new Weather(7);  
  
The presenter clicks Run. This time, the showFail test passes.*   
  
So, the fail execute saying whatever reason you want to say. I don't have to have something not be equal, I don't want to have some condition not be true or false. I can have whatever circumstances, whatever logic, whatever if than else, or looping, or whatever would come up. If some situation is not cool as I determined by my logic, I can just fail the test. So, if don't fail then assume to pass. If I get to the end of a method inside of Test it's assumed to be passed. But I can use the fail to just say, okay at this point I don't like the state of things and fail the Test. So, these basic assertions assertTrue, assertFalse and then fail, allow a foundation to use whatever logic you can design to check your states, and to check your tests.   
*The presenter then highlights the line of code,     fail("Weather is bad, call off our plans"); in the code group:   
  
   @Test  
   public void showFail()  
   {  
       if (weather.isItFreezingToday() || weather.willItRainToday())  
       {  
           fail("Weather is bad, call off our plans");  
       }  
   }  
}*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Number Equality

Learning Objective

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

* *understand how to use Assert number equality methods in JUnit tests*

**1. Number equality assertions**

As our algorithms manipulate numbers, we need to validate that the results are as we expect. When dealing with floating point numbers, it has the extra wrinkle of accounting for precision of values. JUnit, though, gives us a toolbox for asserting these facts about numbers. We have a very simple test here that goes through all the different variations of dealing with numbers. For integers, we have a very basic assert statement – assert two values are equal to each other. It's basically saying our two long values are equal to each other, and it works for all integer types because, as we know, Java will automatically cast any shorter integer types, ints, or characters, or shorts, or anything inside of there into the long type. And so we can just assert any two long values are equal, and we can even add a note if we want to provide more details about what the values mean. As we said, short values integers are any values that go inside of there. Now floating points, then, have that wrinkle inside of here; so here is a double instead, and as I am going to do with assertEquals, I need to provide a precision involved. How close do these guys have to match in order for me to consider them the same?   
 *In Eclipse, a file named AssertEqualsNumbers.java is open. It contains the following code:  
  
package myapp.junit.asserts;  
  
@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, doubleValue, 0.00000001);  
  
       float floatValue = 0.3f;  
       assertEquals("Two float values", floatValue, floatValue, 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, 0.001);  
   }  
     
   @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);  
   }  
}*   
  
Here is the same thing with floating point values. So there's two separate methods in this case – one for doubles, one for floats – that provide both of these checks. Each of these methods provides a little bit of extra data. This data would normally mean something like, "Hey, these are two waypoints in navigation" or these are two – whatever the calculation is going to mean – but we just have whatever text we want to put inside of here. Now the method for comparing two double values without a precision still exists. At this point, I am going to go ahead and run the test, so you can see if, in this case – case c – where I run it without precision, it will fail. And it's going to give us an assertion that says, "Hey, you cannot use this anymore. You have to use the one that has delta. Don't use this one anymore. Use that instead." So just know, you cannot change…you cannot compare two double values without using the precision value that goes along with it. Now that being said, here is an assert – the next one here where I have two values that are close to each other, but they don't match. One is 0.3, the other one is 0.3 with a little bit of extra on top of there. But this test still passes because they are within the delta value. You can see the delta of 0.01 right here, it's still okay, I am okay for rounding here for these two numbers to pass.   
*The presenter clicks Run. The JUnit tab lists seven tests – a\_IntegerEqualsAsserts, b\_FloatingEqualAsserts, c\_DeprecatedEqualsAsserts, d\_DeprecatedEqualsAssets, e\_DoubleDeltaEqualAssertsFail, f\_NotEquals, and g\_FloatDeltaEqualAsserts. Three of the tests failed – c\_DeprecatedEqualsAsserts, e\_DoubleDeltaEqualAssertsFail, and g\_FloatDeltaEqualAsserts.  
  
The presenter selects c\_DeprecatedEqualsAsserts and points to the following error message in the output:  
  
java.lang.AssertError:Use assertEquals(expected, actual, delta)  
  
In the AssertEqualsNumbers.java file, the presenter highlights the values 0.3 and 0.301 in the following code:  
  
   @Test  
   public void d\_DoubleDeltaEqualAsserts()  
   {  
       double doubleValue = 0.3;  
       double secondValue = 0.301;  
       assertEquals("Pass within delta", doubleValue, secondValue, 0.01);  
   }  
  
The presenter also highlights the line  
  
assertEquals("Pass within delta", doubleValue, secondValue, 0.01);*   
  
I do the same test, though, with a little bit smaller delta; say I want these to be more accurate. Then all of a sudden, you can see over here test e is failing. It doesn't match within the delta. I was expecting 0.3, I got 0.301. Now I also have a counter to that to say these two values are not equal. I can do that for double values, I can do that for long values. I can assert the two values are not equal, so you don't have to do a comparison and flip it, you can just simply call the NotEquals method on either one of these. Now the other thing we can check here – we can show you inside of here – is on floating point numbers. Here is where you can do those asserts where, with a greater and greater level of granularity it passes some, it passes some, and then finally it doesn't pass. So you can see, two numbers do not have to match up exactly for floating points either. And as the granularly changes inside the two different…each of the different tests, it might pass some tests and not the others. So comparing numbers, there's a lot of options for inside of JUnit. And particularly floating point numbers; as rare as they are used in Java, if you need them, JUnit provides you a lot of ways to compare those numbers and not have to worry about exact values.   
*Next the presenter selects the test e\_DoubleDeltaEqualsAssetrsfail on the JUnit tab. He points out the following error message in the test output:  
  
Java.lang.AssertionError: Failed within delta expected:<0.3> but was:<0.301>  
  
In the AssertEqualsNumbers.java file, the presenter highlights the code  
  
   @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);  
   }  
  
The presenter then scrolls down to focus on the code  
  
@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 highlights the following lines:  
  
   assertEquals("Still Passes within delta of 0.001", floatValue, secondValue, 0.001f);  
       assertEquals("Failed within delta of 0.0001", floatValue, secondValue, 0.0001f)*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Object Equality

Learning Objective

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

* *understand how to use Assert object equality methods in JUnit tests*

**1. Asserting object equality methods**

Object-oriented solutions rely on the proper configuration and modeling of data. As this code executes, we must validate that an object graph is proper in order to determine our test results. JUnit assertion gives us the methods to help complete that validation. Our first Test we can look at here takes two strings and uses the assertEquals method. It takes the "Two strings" and compares them and as you can see over here, the result says yes they are the same string. I can do the same assertEquals test on any object though. We have two objects being created from the same class here. They have the same values and because the .equal method implemented in the class say it's true, the JUnit class says it's true. So you can see in the SampleClass we have our hashcode, we have our .equals and as long as the .equals returns true, then the assertEquals method is going to assertTrue for those two objects. Over here, I have two more objects I'm creating and these two objects are indeed NotEquals. And so when I do the assertEquals on these two objects the Test fails, because the objects weren't the same. If they were not supposed to be equal I'm expecting two objects to be different. I use the assertNotEquals here and I'm going to check, hey make sure that these two objects do not indeed have the same values, use the .equals method and the assertion will make sure that they are not the same.   
*In Eclipse, two files are open on separate tabbed pages – AssertEqualsObjects.java and SampleClass.java. The JUnit tab lists ten tests – a\_StringEquals,  b\_ObjectEquals, c\_ObjectEqualsFail, d\_ObjectNotEquals, e\_SameObject, f\_NullObjectFail, g\_NullObjects, h\_Null, i\_NotNull, and j\_ObjectArray.      
  
The AssertEquals Objects.java file contains the following code:  
  
package myapp.junit.asserts;  
  
import static org.junit.Assert.assertEquals;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class AssertEqualsObjects  
{  
   @Test  
   public void a\_StringEquals()  
   {  
       String first = "objectEquals";  
       String second = "objectEquals";  
       assertEquals("Two strings", first, second);  
   }  
     
   @Test  
   public void b\_ObjectEquals()  
   {  
       SampleClass first = new SampleClass(42, "a value");  
       SampleClass second = new SampleClass(42, "a value");  
       assertEquals("Two .equals Objects", first, second);  
   }  
  
   @Test  
   public void c\_ObjectEqualsFail()  
   {  
       SampleClass first = new SampleClass(42, "a value");  
       SampleClass second = new SampleClass(42, "another value");  
       assertEquals("Two not .equals Objects", first, second);  
   }  
  
   @Test  
   public void d\_ObjectNotEquals()  
   {  
       SampleClass first = new SampleClass(42, "a value");  
       SampleClass second = new SampleClass(42, "another value");  
       assertNotEquals("Two not .equals Objects", first, second);  
   }  
     
   @Test  
   public void e\_SameObject()  
   {  
       String first = "objectEquals";  
       String nonPooled = new String("objectEquals");  
       assertEquals("Two strings not ==", first, nonPooled);  
       assertNotSame("Not the same object intentionally", first, nonPooled);  
       assertSame("Not the same object", first, nonPooled);  
   }  
  
   @Test  
   public void f\_NullObjectFail()  
   {  
       String first = "objectEquals";  
       String nullObject = null;  
       assertEquals("One is null", first, nullObject);  
   }  
     
   @Test  
   public void g\_NullObjects()  
   {  
       String first = null;  
       String second = null;  
       assertEquals("Both are null", first, second);  
   }  
  
   @Test  
   public void h\_Null()  
   {  
       String check = null;  
       assertNull("This should be null", check);  
   }  
  
   @Test  
   public void i\_NotNull()  
   {  
       String check = "value";  
       assertNotNull("This should be not be null", check);  
   }  
     
   @Test  
   public void j\_ObjectArray()  
   {  
       int [] first = {1, 2, 3};  
       int [] sameOne = first;  
       assertEquals("Two array pointers", first, sameOne);  
       int [] second = {1, 2, 3};  
       assertEquals("Two equal arrays", first, second);  
   }  
}  
  
The presenter starts by highlighting the first test. The relevant code is  
  
   @Test  
   public void a\_StringEquals()  
   {  
       String first = "objectEquals";  
       String second = "objectEquals";  
       assertEquals("Two strings", first, second);  
  
On the JUnit tab, the presenter selects the test a\_StringEquals. A green icon with a tick mark shows that the test passed.   
  
The presenter switches to the SampleClass.java file. The file contains the following code:   
  
package myapp.junit.asserts;  
  
public class SampleClass  
{  
   private int someNumber;  
   private String someString;  
  
   public SampleClass(int someNumber, String someString)  
   {  
       super();  
       this.someNumber = someNumber;  
       this.someString = someString;  
   }  
   @Override  
   public int hashCode()  
   {  
       final int prime = 31;  
       int result = 1;  
       result = prime \* result + someNumber;  
       result = prime \* result + ((someString == null) ? 0 : someString.hashCode());  
       return result;  
   }  
   @Override  
   public boolean equals(Object obj)  
   {  
       if (this == obj)  
           return true;  
       if (obj == null)  
           return false;  
       if (getClass() != obj.getClass())  
           return false;  
       SampleClass other = (SampleClass) obj;  
       if (someNumber != other.someNumber)  
           return false;  
       if (someString == null)  
       {  
           if (other.someString != null)  
               return false;  
       } else if (!someString.equals(other.someString))  
           return false;  
       return true;  
   }  
   public int getSomeNumber()  
   {  
       return someNumber;  
   }  
   public String getSomeString()  
   {  
       return someString;  
   }  
}  
  
The presenter returns to the AssertEqualsObjects.java file and navigates to the following code:  
  
   @Test  
   public void c\_ObjectEqualsFail()  
   {  
       SampleClass first = new SampleClass(42, "a value");  
       SampleClass second = new SampleClass(42, "another value");  
       assertEquals("Two not .equals Objects", first, second);  
   }  
  
The presenter highlights the line  
  
assertEquals("Two not .equals Objects", first, second);   
  
The presenter then navigates to the following code:  
  
@Test  
   public void d\_ObjectNotEquals()  
   {  
       SampleClass first = new SampleClass(42, "a value");  
       SampleClass second = new SampleClass(42, "another value");  
       assertNotEquals("Two not .equals Objects", first, second);  
   }  
  
In the code, the presenter highlights the line  
  
assertNotEquals("Two not .equals Objects", first, second);*   
  
Now, the .equals checks the values are the same, the equal-equals checks are they the same object and JUnit gives us that option as well. So, in this case I have two strings, I'm using strings here in this case, to show I have objectEquals and a String created outside the String pool, and they are indeed equals the same String. They are not the same object, because one is in the string pool and one is not. And when we do an assertSame here we can see this next test fails, they are not the same object. So, if I say assert not the same, that's okay, because they are indeed not the same. But if I say  assert the same it's using the equal-equal method, and then they are saying these are not the same String. So, if you need to make sure in your object graph two objects are the same object, you say assert the same. You want to make sure that they are not the same object use assert not the same. Now, one of the quirks inside of assertEquals if one of the objects is null, obviously the other one is not, it's going to fail. So, our nullObject fails over here. If both of the objects are null in this case, in Test g you can see that's okay, if both the objects are null they are indeed the same, they are not required to have values inside of here. If I want to check that they are indeed null though I use assertNull, I can take any variable and say this variable should be null. And at this point that's null, you're pass,   
*In the AssertEqualsObjects.java file, the presenter navigates to the following code:  
  
   @Test  
   public void e\_SameObject()  
   {  
       String first = "objectEquals";  
       String nonPooled = new String("objectEquals");  
       assertEquals("Two strings not ==", first, nonPooled);  
       assertNotSame("Not the same object intentionally", first, nonPooled);  
       assertSame("Not the same object", first, nonPooled);  
   }  
  
Next the presenter focuses on the code   
  
   @Test  
   public void f\_NullObjectFail()  
   {  
       String first = "objectEquals";  
       String nullObject = null;  
       assertEquals("One is null", first, nullObject);  
   }  
  
The presenter highlights the line  
  
assertEquals("One is null", first, nullObject);  
  
On the JUnit tab, the presenter selects the test f\_NullObjectFail. A red icon with a cross indicates that the test failed.  
  
Next the presenter navigates to the code  
       
   @Test  
   public void g\_NullObjects()  
   {  
       String first = null;  
       String second = null;  
       assertEquals("Both are null", first, second);  
   }  
  
On the JUnit tab, the presenter selects the test g\_NullObject. A green icon with a tick shows that it passed.*  
  
or I can check a variable is not null. So, if in my object graph, I'm expecting an object at some point and it shouldn't be null I can say, hey assert this guy is not indeed null. Last note inside of here, you can try and use some of this stuff on an array. So, if you put an arrays objects inside of these guys you can say, hey are these two arrays equal? Well in the first case it works out that they are indeed equals, because the assertEquals on an array will check an equal-equal on the two arrays, it does not do a deep comparison of the arrays. So, if the two are indeed pointing to the same array the first case, two array pointers does not fail. But the second case, if I change this to be the same values, but a different instance of an array it will fail. If I want to do array comparisons, there is a whole other branch of JUnit that you can look at to do those array comparisons. So, you can see if I'm checking out an object graph, JUnit gives us plenty of options to be able to look at that, and be able to determine if an object graph does as I expect it to be.   
*The presenter navigates to the code   
  
   @Test  
   public void i\_NotNull()  
   {  
       String check = "value";  
       assertNotNull("This should be not be null", check);  
   }  
     
The presenter highlights the line  
  
assertNotNull("This should be not be null", check);  
  
Finally the presenter focuses on the following code:  
  
   @Test  
   public void j\_ObjectArray()  
   {  
       int [] first = {1, 2, 3};  
       int [] sameOne = first;  
       assertEquals("Two array pointers", first, sameOne);  
       int [] second = {1, 2, 3};  
       assertEquals("Two equal arrays", first, second);  
   }  
}*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Array Equality

Learning Objective

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

* *understand how to use Assert array equality methods in JUnit tests*

**1. Asserting array equality methods**

If our solution includes arrays, we need to ensure the manipulation of array data is done with integrity. JUnit assertions provide us with methods that deal directly with arrays, and gives us the ability to perform this analysis. Let's see how JUnit can help. The assertEquals is what we use to compare two objects. Within arrays if I have two arrays that point to the same array object, and there are two variables that are pointing to the same array object, and I use assertEquals that will actually pass. But if I try and I use assertEquals on two arrays that are the same value, it will fail. So, you can see over here it says the two equals two arrays are not equal, it only does an equal-equals it doesn't do a deep equality check on arrays. So, in order to check two arrays I can have two obviously equal arrays, but use the assertArraysEquals instead. This check will go through each of the values and check, hey, they are indeed the same, and you can see our second test here passes.   
*In Eclipse, the AssertArrayEquals.java file is open and the JUnit table lists eight tests – a\_NotEquals, b\_ArrayEquals, c\_ArrayEqualsFail, d\_ArrayEqualsFailSize, e\_TypeFail, f\_ManyTypes, g\_MixedTypes, and h\_NotEquals.  
  
The AssertArrayEquals.java file contains the following code:   
  
package myapp.junit.asserts;  
  
import static org.junit.Assert.assertArrayEquals;  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class AssertArrayEquals  
{  
   @Test  
   public void a\_NotEquals()  
   {  
       int [] first = {1, 2, 3};  
       int [] sameOne = first;  
       assertEquals("Two array pointers", first, sameOne);  
       int [] second = {1, 2, 3};  
       assertEquals("Two equal arrays", first, second);  
   }  
     
   @Test  
   public void b\_ArrayEquals()  
   {  
       int [] first = {1, 2, 3};  
       int [] second = {1, 2, 3};  
       assertArrayEquals("Two equal arrays", first, second);  
   }  
  
   @Test  
   public void c\_ArrayEqualsFail()  
   {  
       int [] first = {1, 2, 3};  
       int [] second = {1, 2, 4};  
       assertArrayEquals("Two unequal arrays", first, second);  
   }  
  
   @Test  
   public void d\_ArrayEqualsFailSize()  
   {  
       int [] first = {1, 2, 3};  
       int [] second = {1, 2};  
       assertArrayEquals("Two very different", first, second);  
   }  
  
   @Test  
   public void e\_TypeFail()  
   {  
       Object[] first = {"string"};  
       Object[] second = {new SampleClass(5, "e")};  
       assertArrayEquals("Two very different", first, second);  
   }  
     
   @Test  
   public void f\_ManyTypes()  
   {  
       double [] first = {1.0, 2.0, 3.0};  
       double [] second = {1.0, 2.0, 3.0};  
       assertArrayEquals("Two double arrays", first, second, 0.01);  
         
       String[] string1 = {"A", "B"};  
       String[] string2 = {"A", "B"};  
       assertArrayEquals("Two String arrays", string1, string2);  
         
       SampleClass [] any1 = {new SampleClass(5, "d")};  
       SampleClass [] any2 = {new SampleClass(5, "d")};  
       assertArrayEquals("Two Any Object arrays", any1, any2);  
   }  
  
   @Test  
   public void g\_MixedTypes()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       assertArrayEquals("Two Mixed Object arrays", any1, any2);  
   }  
     
   @Test(expected=AssertionError.class)  
   public void h\_NotEquals()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "e"), "string", new Integer(3)};  
       try  
       {  
           assertArrayEquals("Two Mixed Object arrays", any1, any2);  
           fail("The arrays should be different");  
       } catch (AssertionError e)  
       {  
         
       }  
   }  
}  
  
In the file, the presenter navigates to the code  
  
@Test  
   public void a\_NotEquals()  
   {  
       int [] first = {1, 2, 3};  
       int [] sameOne = first;  
       assertEquals("Two array pointers", first, sameOne);  
       int [] second = {1, 2, 3};  
       assertEquals("Two equal arrays", first, second);  
   }  
  
On the JUnit tab, the presenter points out that the output for the selected test, a\_NotEquals, includes a java.lang.AssertionError message stating that two equal arrays were expected.   
  
In the file, the presenter focuses on the code  
  
   @Test  
   public void b\_ArrayEquals()  
   {  
       int [] first = {1, 2, 3};  
       int [] second = {1, 2, 3};  
       assertArrayEquals("Two equal arrays", first, second);  
   }  
  
On the JUnit tab, the presenter selects the b\_ArrayEquals test, which is listed as having passed.*  
  
If we have two arrays that are indeed not the same values again 1, 2, 3 versus 1, 2, 4, and the values inside of there you can see in this case the values are different. It expected a three, but it got a four and it's very detailed in what it reports to. It says here exactly the element that was different and it tells you the position of that element. It was in the last position in this case, the element two. Zero, one, two being the third element. If the arrays are of different sizes, you see the first array is of size three. The second array is a size two. That obviously fails as well and JUnit tells you very much that there are different array lengths. I expected a length of three, I got a length of two, so they did not match up. It's a very smart check around this one call to assertArrayEquals. If the classes have different types associated with them. Here I have an array that contains a string, and the second one has an array that contains a separate class that I built elsewhere. You can see it actually is saying, hey I expected a string, but instead I got an instance of the class. So, our assertArrayEquals does a lot of different checking on the values, to make sure that they are truly the same value, there's no fooling it.   
*The presenter navigates to the code  
  
 @Test  
   public void c\_ArrayEqualsFail()  
   {  
       int [] first = {1, 2, 3};  
       int [] second = {1, 2, 4};  
       assertArrayEquals("Two unequal arrays", first, second);  
   }  
  
The presenter highlights the lines int [] first = {1, 2, 3};  and int [] second = {1, 2, 4};.   
  
On the JUnit tab, the presenter selects the c\_ArrayEqualsFail test, which is marked as having failed. He points to the output message  
  
Two unequal arrays:arrays first differed at element [2]; expected <3> but was <4>  
  
The presenter navigates to the code  
  
@Test  
   public void d\_ArrayEqualsFailSize()  
   {  
       int [] first = {1, 2, 3};  
       int [] second = {1, 2};  
       assertArrayEquals("Two very differnet", first, second);  
  
The presenter highlights the line  
  
int [] second = {1, 2};  
  
On the JUnit tab, the presenter selects the d\_ArrayEqualsFailSize test, which is marked as having failed. He points to the output  
  
java.lang.AssertionError: Two very different: array lengths differed, expected.length=3 actual.length=2   
  
Next the presenter navigates to the code  
  
    @Test  
   public void e\_TypeFail()  
   {  
       Object[] first = {"string"};  
       Object[] second = {new SampleClass(5, "e")};  
       assertArrayEquals("Two very different", first, second);  
   }  
  
On the JUnit tab, the presenter selects the e\_TypeFail test, which is marked as having failed.*   
  
It will work just fine on many different types though. You can see this first Test we have inside of here has two arrays of doubles, and it checks it just fine. You see many types, the test here passes perfectly with flying colors each time. It works fine with arrays of strings. It works fine with arrays of classes. So, it doesn't matter integers, doubles, strings, classes, it will check any of those different types and it will even check arrays of mixed types. You see I have my own custom object, plus a string, plus an Integer, and they are all containing the same values. And so as it does a .equals on each one of these, they turn out just perfectly fine and inserts these arrays are indeed equals. The trick inside of there is there is no NotEquals, I can't assert the arrays are indeed not equals. The only trick I can think of to work around that is, in this case the two arrays are defined above, and the first object I'm tweaking the values, they are indeed not the same. When this test passes, this assertion checks, it will actually fail. If I take out this right here for a second, **Save** it and rerun my test, it's going to fail. These are indeed not equal arrays, but by putting the assertion check here and say, hey I'm expecting an AssertionError here. I'm only doing the one check anyway, it allows me to cheat a little bit. I can handle that exception.   
*The presenter scrolls down to the following code:   
  
@Test  
   public void f\_ManyTypes()  
   {  
       double [] first = {1.0, 2.0, 3.0};  
       double [] second = {1.0, 2.0, 3.0};  
       assertArrayEquals("Two double arrays", first, second, 0.01);  
         
       String[] string1 = {"A", "B"};  
       String[] string2 = {"A", "B"};  
       assertArrayEquals("Two String arrays", string1, string2);  
         
       SampleClass [] any1 = {new SampleClass(5, "d")};  
       SampleClass [] any2 = {new SampleClass(5, "d")};  
       assertArrayEquals("Two Any Object arrays", any1, any2);  
   }  
  
   @Test  
   public void g\_MixedTypes()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       assertArrayEquals("Two Mixed Object arrays", any1, any2);  
   }  
  
The presenter then highlights the lines  
  
       double [] first = {1.0, 2.0, 3.0};  
       double [] second = {1.0, 2.0, 3.0};  
       String[] string1 = {"A", "B"};  
       String[] string2 = {"A", "B"};  
       SampleClass [] any1 = {new SampleClass(5, "d")};  
       SampleClass [] any2 = {new SampleClass(5, "d")};  
  
The presenter scrolls down further to reveal the code  
  
@Test  
   public void g\_MixedTypes()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       assertArrayEquals("Two Mixed Object arrays", any1, any2);  
   }  
     
   @Test(expected=AssertionError.class)  
   public void h\_NotEquals()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "e"), "string", new Integer(3)};  
       {  
           assertArrayEquals("Two Mixed Object arrays", any1, any2);  
  
}  
  
The presenter highlights the lines  
  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "d"), "string", new Integer(3)};  
  
Next the presenter navigates to the code  
  
@Test  
   public void g\_MixedTypes()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       assertArrayEquals("Two Mixed Object arrays", any1, any2);  
   }  
     
   @Test(expected=AssertionError.class)  
   public void h\_NotEquals()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "e"), "string", new Integer(3)};  
       try  
       {  
  
The presenter highlights the code lines   
  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "d"), "string", new Integer(3)};  
  
The presenter then navigates to the code  
  
   @Test(expected=AssertionError.class)  
   public void h\_NotEquals()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "e"), "string", new Integer(3)};  
               assertArrayEquals("Two Mixed Object arrays", any1, any2);  
       {  
        
}  
  
  
The presenter highlights the line  
  
assertArrayEquals("Two Mixed Object arrays", any1, any2);  
  
The presenter then navigates to the test that starts with the line @Test(expected=AssertionError.class). He removes the line  
  
 (expected=AssertionError.class)  
  
The full code for the relevant test is now  
  
@Test  
   public void h\_NotEquals()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
  
The presenter clicks Run and the test fails. The presenter then changes the code back to   
  
@Test(expected=AssertionError.class)  
   public void h\_NotEquals()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "e"), "string", new Integer(3)};  
  {  
       assertArrayEquals("Two Mixed Object arrays", any1, any2);  
          
}*   
  
So, the only other trick I could say you could do is if you have many checks inside of there, you could do a try and a catch around this guy for the AssertionError. And this would allow you to pass it appropriately. And if not, you could fail inside of there and say something along the lines of, The arrays should be different, and you can spell that properly too. All right, and so that's the check you could do in there to get the NotEquals working, but there is no NotEquals on its own. Outside of that though, the assertArrayEquals gives us a ton of options for doing a deep dive into arrays, and validating that indeed as I have manipulated an array, the values have been done and changed with integrity.   
*The presenter then adds the lines of code, try, fail("the arrays should be different");, and catch (assertionError e) to the public void h\_NotEquals() code group. The full code is now  
  
@Test(expected=AssertionError.class)  
   public void h\_NotEquals()  
   {  
       Object[] any1 = {new SampleClass(5, "d"), "string", new Integer(3)};  
       Object[] any2 = {new SampleClass(5, "e"), "string", new Integer(3)};  
       try  
       {  
           assertArrayEquals("Two Mixed Object arrays", any1, any2);  
           fail("The arrays shoudl be different");  
       } catch (AssertionError e)  
       {  
         
       }  
   }  
}*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Basic assertThat Options

Learning Objective

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

* *understand how to use assertThat methods in JUnit tests*

**1. Using assertThat methods**

As our solution logic complexity grows, so does our test validation logic. With the addition of the assertThat validation check, JUnit provides us with a framework for completing complex validations. Let's look at a few of these basic options assertThat enables. So the assertThat statement starts off with some really basic checks here. We can give that a reason or not, though that is highly preferable to give that reason of what you are checking for – that logic, "Hey, if this failed, this is the reason it failed. The weather is bad," or whatever the reason would be outside of there. The first assertThat option we can look at, we take the tested value here, in this case "Hello matcher logic", and we say, hey, assertThat anything is true. It doesn't matter what the value is, just anything in there is going to be perfectly fine. So the check here – it doesn't matter what the string is passing, you can even have a null value here – this is going to return true for absolutely anything that's passed in. And it's not terribly, terribly useful, but if there is a need for it, hey, run with it.   
*In Eclipse, a file named AssertThatBasics.java is open and the JUnit tab lists 11 tests – \_Anything, \_ContainsString, \_EndsWith, \_EqualTo, \_InstanceOf, \_Negation, \_NotEqualTo, \_NotNull, \_Null, \_Same, and \_StartsWith.  
  
In the AssertThatBasics.java file, the following is visible:  
  
package myapp.junit.asserts.that;  
  
import static org.junit.Assert.assertThat;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class AssertThatBasics  
{  
   @Test  
   public void \_Anything()  
   {  
       String tested = "Hello matcher logic";  
       String check = "matcher";  
       assertThat("Anything passes", tested, anything(check));  
   }  
  
   @Test  
   public void \_Null()  
   {  
       String tested = null;  
       assertThat("Is it null?", tested, nullValue());  
   }  
  
   @Test  
   public void \_NotNull()  
   {  
       String tested = "";  
       assertThat("Is it not null?", tested, notNullValue());  
   }  
  
The presenter focuses on the code  
  
   @Test  
   public void \_Anything()  
   {  
       String tested = "Hello matcher logic";  
       String check = "matcher";  
       assertThat("Anything passes", tested, anything(check));  
   }  
  
The presenter highlights the word "anything" in the code line  
  
assertThat("Anything passes", tested,  anything(check));.*   
  
The next check here is, very simply, is it a nullValue? So we say that the value passed in right here, if it's a null value, then it's okay. If it’s not a null value, the check will fail. By the way, all these tests passed, so we're just kind of walking through the capabilities of these. But I can show you here, if I change this to a notNullValue and I run the test, then you can see down here, one of them fails. The null check fails inside of there. So it's…all these just really simply check whatever the criteria you put out there, and as long as it meets that criteria, it passes just fine. So let's just run that one more time  and see…yep, everything is shiny, everything runs just fine again. So the notNull is the other check. So in this case, I have a value and as long as that value here is not null, assertThat is totally okay with that. So you can see, instead of using the other types of asserts where we do an assert null or assert notNull or whatnot, we are using the assertThat that can use the basic just assertion: hey, this is null, this is not null. What we will see later on is that these don't look terribly different or complex now, but when we get to more advanced to assertThat options, we can actually combine them and then can use them in interesting ways. So we will build on this in additional looks at assertThat later on.   
*The presenter navigates to the code  
  
   @Test  
   public void \_Null()  
   {  
       String tested = null;  
       assertThat("Is it null?", tested, nullValue());  
   }  
  
The presenter highlights nullValue. He then refers to the JUnit tab, which indicates that all the tests have passed.  
  
The presenter changes the line String tested = null; to  
  
String tested = " ";  
  
The full code for the relevant test is now  
  
 @Test  
   public void \_Null()  
   {  
       String tested = " ";  
       assertThat("Is it null?", tested, nullValue());  
   }  
  
The presenter clicks Run. On the JUnit tab, he selects the test  \_Null, which is marked as having failed. The output includes the message  
  
java.lang.AssertionError: Is it null?   
  
The presenter changes the code back to  
  
 @Test  
   public void \_Null()  
   {  
       String tested = null;  
       assertThat("Is it null?", tested, nullValue());  
   }  
  
The presenter clicks Run and the JUnit tab indicates that all the tests have passed.  
  
Next the presenter navigates to the code  
  
   @Test  
   public void \_NotNull()  
   {  
       String tested = "";  
       assertThat("Is it not null?", tested, notNullValue());  
   }*   
  
So the EqualTo option is kind of like assert equals, but you can see we have a value – is it equal to some other value? So is equals equal to the other value? We are asking are they equal, and there’s two ways we can assert that. This is kind of a syntax language how you want to define your assertions – we have equalTo and we have is. So the value is the other value. So it's just kind of another syntact-ey way of making it sugary in our syntax – they do exactly the same thing. NotEquals – just simply has Not. So the value is not the other value, and again, all this within the assertThat. The assertThat, as we have said, it's taking a value and some matcher. And then all of these different things we are looking at here are matchers, and those matchers are coming out of the assertThat inside of here – the static import of assertThat. They are open inside of here. And then we have these core matchers from Hamcrest that we are using – and so that's where we are seeing this magic that comes in. Moving on down the list, we have the Same, and so we are saying the equal equals, we are checking the equal equals there, so is it sameInstance or is it theInstance? These will mean the same thing – again, more syntactic sugar. And we can ask if it is an instance of. So there is an instanceOf matcher out there, and then that isA matcher; in that object-oriented sense of this, is this a version of another thing? So is this object an instance of whatever class? Is it a…is it syntactic sugar we can throw on top of there?   
*The presenter scrolls down in the file to reveal the following code:  
  
   @Test  
   public void \_EqualTo()  
   {  
       String tested = "equals";  
       String check = "equals";  
       assertThat("Are they equals?", tested, equalTo(check));  
       assertThat("Are they equals?", tested, is(check));  
   }  
  
   @Test  
   public void \_NotEqualTo()  
   {  
       String tested = "separate";  
       String check = "equals";  
       assertThat("They are not equal", tested, not(check));  
   }  
  
The presenter focuses on the   
code  
  
   @Test  
   public void \_EqualTo()  
   {  
       String tested = "equals";  
       String check = "equals";  
       assertThat("Are they equals?", tested, equalTo(check));  
       assertThat("Are they equals?", tested, is(check));  
   }  
  
The presenter highlights various elements in the code, including, "equals",  "equalTo", and "is".  
  
The presenter then focuses on the code  
  
   @Test  
   public void \_NotEqualTo()  
   {  
       String tested = "separate";  
       String check = "equals";  
       assertThat("They are not equal", tested, not(check));  
   }  
  
The presenter highlights "not" and "tested". A drop-down list includes the following two options:  
  
assertThat(T actual,Matcher<?super T>matcher): void - Assert  
assertThat(String reason, T actual,Matcher<?super T>matcher): void - AssertThe presenter then navigates to the top of the file and clicks to expand the code at the line import static org.junit.Assert.assertThat;  
  
The following code becomes visible:  
  
import static org.hamcrest.CoreMatchers.\*;  
  
import java.io.Serializable;  
  
import myapp.junit.asserts.SampleClass;  
  
import org.junit.FixMethodOrder;  
import org.junit.Test;  
import org.junit.runners.MethodSorters;  
  
The presenter scrolls back down to the following code:  
  
@Test  
   public void \_Same()  
   {  
       SampleClass first = new SampleClass(3, "same");  
       SampleClass same = first;  
       assertThat("==?", first, sameInstance(same));  
       assertThat("==?", first, theInstance(same));  
   }  
     
   @Test  
   public void \_InstanceOf()  
   {  
       String tested = "Excited!";  
       assertThat("Is instanceof?", tested, instanceOf(Serializable.class));  
       assertThat("Is a?", tested, isA(Serializable.class));  
   }  
  
   @Test  
   public void \_ContainsString()  
   {  
       String tested = "Hello matcher logic";  
       String check = "matcher";  
       assertThat("Does it contain a string?", tested, containsString(check));  
   }  
  
The presenter highlights sameInstance and  theInstance in the code  
  
@Test  
   public void \_Same()  
   {  
       SampleClass first = new SampleClass(3, "same");  
       SampleClass same = first;  
       assertThat("==?", first, sameInstance(same));  
       assertThat("==?", first, theInstance(same));  
   }  
  
The presenter then focuses on the code  
  
   @Test  
   public void \_InstanceOf()  
   {  
       String tested = "Excited!";  
       assertThat("Is instanceof?", tested, instanceOf(Serializable.class));  
       assertThat("Is a?", tested, isA(Serializable.class));  
  
the presenter highlights the elements instanceOf(Serializable.class) and isA(Serializable.class).*   
  
We can ask, does this matcher contain a string? So this is kind of like a star logic. Is the string matcher located anywhere within the string up above? So yes, it is: "Hello matcher logic" contains the string matcher inside of it. We can get more detail than that, and we can say, does it start with this? So we say, hey, we have this string up here – does it start with a parenthesis? Indeed it does. So it matches there. And we could also complement that with the endsWith. And so the endsWith – does it end with an exclamation point? Yes, it's a very excited statement – right there, it ends with the exclamation point. The…where we can get into the complex stuff then, is we can say, hey, we want to make sure there is no excitement in this. So we want to say, does it not end with this value…and this is where our matching gets really interesting. We can have that sugar of syntax, say, hey, whats the not end with whatever check here? So as long as it doesn't have the exclamation point, this one is going to be okay. So these are some basic options we are going to look at with an assertThat that really help us to start decorating and making much more complex assertions and validations, as our logic grows within our systems.   
*The presenter scrolls down in the file to reveal the following code:  
  
   @Test  
   public void \_ContainsString()  
   {  
       String tested = "Hello matcher logic";  
       String check = "matcher";  
       assertThat("Does it contain a string?", tested, containsString(check));  
   }  
  
   @Test  
   public void \_StartsWith()  
   {  
       String tested = "(side bar)";  
       String check = "(";  
       assertThat("Starts with?", tested, startsWith(check));   
  
The presenter scrolls down in the file, revealing the code  
  
  
  
  
   @Test  
   public void \_EndsWith()  
   {  
       String tested = "Excited!";  
       String check = "!";  
       assertThat("Ends with?", tested, endsWith(check));  
   }  
  
   @Test  
   public void \_Negation()  
   {  
       String tested = "Neutral";  
       String check = "!";  
       assertThat("Negation", tested, not(endsWith(check)));  
   }  
}  
  
The presenter focuses on the code  
  
   @Test  
   public void \_EndsWith()  
   {  
       String tested = "Excited!";  
       String check = "!";  
       assertThat("Ends with?", tested, endsWith(check));  
   }  
  
The presenter highlights endsWith(check)) and "!".   
  
The presenter then focuses on the   
code  
  
   @Test  
   public void \_Negation()  
   {  
       String tested = "Neutral";  
       String check = "!";  
       assertThat("Negation", tested, not(endsWith(check)));  
   }  
  
The presenter highlights "!" and not(endsWith(check).*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Combination assertThat Statements

Learning Objective

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

* *understand how to use combined assertThat statements in JUnit tests*

**1. Using combined assertThat statements**

Combination logic is not limited to just our code within our solution. Traditionally, we would have to drive equally complex Java logic, to validate the covering test cases. Using assertThat within the JUnit enables us to combine checks within a test-driven syntax. Let's take a look. We're back to using assertThat and if you remember the assertThat option goes through, and it takes in the value that we're testing with and then any matchers. And so we're going to look at a couple of complex matchers here, that allows us to do some combination logic that gets to the heart of exactly what I search for. So, given this string right here we might want to do a check saying, hey this string needs to start with a parenthesis and it needs to contain the word bar, and it needs to end with an exclamation point. If all of these three are true then it's a good string. It's something that should have been produced out of our system. Now, as we look at here this test passes, because our string in here indeed contains all of these criteria. We can also look at logic such as anyOf. So we're saying, hey this string up here does it contain anyOf, a comma, a period, a exclamation point, or a question mark. We're saying we want punctuation inside here, if this has any of these items, it's an okay string. So, you can say indeed it has a comma, it has a question mark, it doesn't need all of those.   
*In Eclipse, a file named AssertThatCombination.java is open and the JUnit tab lists five tests – \_AllOf, \_AnyOf, \_Combination, \_Either, and \_Regex.   
  
The following code is visible in the AssertThatCombination.java file:  
  
package myapp.junit.asserts.that;  
  
import static org.hamcrest.CoreMatchers.allOf;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class AssertThatCombination  
{  
   @Test  
   public void \_AllOf()  
   {  
       String tested = "(side bar)!";  
       assertThat("Many Criteria", tested, allOf(startsWith("("),   
                                                 containsString("bar"),   
                                                 endsWith("!")));  
   }  
  
   @Test  
   public void \_AnyOf()  
   {  
       String tested = "does this, being a fragment, contain punctuation?";  
       assertThat("Many Criteria", tested, anyOf(containsString(","),  
                                                 containsString("."),  
                                                 containsString("!"),  
                                                 containsString("?")));  
   }  
  
   @Test  
   public void \_Combination()  
   {  
       String tested = "this should avoid special characters";  
       assertThat("Negated anyOf", tested, not(anyOf(containsString(","),  
                                                     containsString("."),  
                                                     containsString("!"),  
                                                     containsString("?"))));  
   }  
  
  
The presenter focuses on the code   
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class AssertThatCombination  
{  
   @Test  
   public void \_AllOf()  
   {  
       String tested = "(side bar)!";  
       assertThat("Many Criteria", tested, allOf(startsWith("("),   
                                                 containsString("bar"),   
                                                 endsWith("!")));  
   }  
  
In the code, the presenter highlights startsWith("("), containsString("bar"), and    endsWith("!"))).   
  
The presenter then focuses on the code  
  
   @Test  
   public void \_AnyOf()  
   {  
       String tested = "does this, being a fragment, contain punctuation?";  
       assertThat("Many Criteria", tested, anyOf(containsString(","),  
                                                 containsString("."),  
                                                 containsString("!"),  
                                                 containsString("?")));  
  
The presenter highlights the three strings in the containsString statements.*   
  
Now, if I change my code here a little bit, let's just say I put a space up in front of the side bar. I'm going to change my criteria here obviously and then now you can see my AllOf test fails. If I remove all the punctuation, this wouldn't work anymore. It's taking each one of these checks and if any of them are true it's okay. Or if all of these are not true it's not okay, I put this back and everything is happy again. Our combination logic is checking for all of these things and it's a pretty simple syntax. Don't forget, we can also negate it, let's say we don't want these special characters, we're going to string it doesn't have special characters. Well, again we have our anyOf logic just like we had up above, but we throw a not in front of it. Our matchers allowed us to basically string these together. So, I have a matcher here saying, hey does this contain a comma. Take the value of that and these other three as well and combine them together with anyOf. Again, you can have any number. If we look at the anyOf the syntax inside here the same with the allOf, I have that cool little Java feature that allows any number of parameters to be passed in. It's an array of parameters once it gets inside the implementation. And then the not takes any other matcher, and it looks for the exact opposite of what's being returned here. These are all returning true or false when it's all said and done.   
*The presenter navigates back to the code  
  
   @Test  
   public void \_AllOf()  
   {  
       String tested = "(side bar)!";  
       assertThat("Many Criteria", tested, allOf(startsWith("("),   
                                                 containsString("bar"),   
                                                 endsWith("!")));  
   }  
  
In the line String tested = "(side bar)!";, the presenter adds a space before (side bar). The updated line is  
  
String tested = " (side bar)!";  
  
The presenter clicks Run. On the JUnit tab, the \_AllOf test is marked as having failed.  
  
The presenter removes the extra space from the code and clicks Run, and the test passes.   
  
The presenter then focuses on the code  
   @Test  
   public void \_Combination()  
   {  
       String tested = "this should avoid special characters";  
       assertThat("Negated anyOf", tested, not(anyOf(containsString(","),  
                                                     containsString("."),  
                                                     containsString("!"),  
                                                     containsString("?"))));  
   }  
  
  
The presenter highlights each of the strings within the containsString statements. The presenter then selects containsString and a drop-down list opens. In the list, the presenter highlights the option  
  
anyOf(Iterable<Matcher<?superT>>matchers) : AnyOf<T> - CoreMatchers*   
  
Now, you might be saying to yourself this looks awfully like a workaround for regular expressions. And if your preference is to deal with the regular expressions in that syntax, you can do that inside of the matchers to. So, we can go through and we can say assertTrue, and we can pick the value and say, does this match some regular expressions. So, we can use regular expressions inside of our JUnit logic. We're just not using the assertThat anymore just using assertTrue, and it straight matches logic inside of regular expressions. So, don't think that we have to throw out regular expressions, but it's definitely an option within our testing suite.   
*In the AssertThatCombination.java file, the presenter scrolls down to reveal the code  
  
   @Test  
   public void \_Regex()  
   {  
       String tested = "this should avoid special characters";  
       assertTrue("Using a regex instead", tested.matches("^[^<>%$,.!?]\*$"));  
   }  
     
   @Test  
   public void \_Either()  
   {  
       String tested = "(side bar)!";  
       assertThat("Either", tested, either(endsWith("!"))  
                                       .or(startsWith("("))  
                                       .and(notNullValue()));  
   }  
     
}  
  
The presenter focuses on the code  
  
   @Test  
   public void \_Regex()  
   {  
       String tested = "this should avoid special characters";  
       assertTrue("Using a regex instead", tested.matches("^[^<>%$,.!?]\*$"));*   
  
Now, the last thing we can look at here, is when you want to do that and and or logic with inside of your checks. There is nothing wrong with that either that's a great option as well. And so here I can say either. So I either want this to end with a exclamation point or start with a parenthesis, and not be a null value. I want that logic to hold true and as long as they are all you know fine inside of there, it's all going to work. So, again I could change this to a null value. And it's going to go through and it's going to fail. I put it back and it needs to either start with a exclamation point or be both of those. And then you know I can do all this different test cases and you know and check it out. But again, this combination logic allows us to do exactly the checks, that we would be doing on the other side that we're normally doing with testing. But instead of having coded in Java syntax, and if then statement, and equals, and all that sort of stuff. We've really great testing logic that allows us to basically throw us together in a very test-like, data-like natural language-driven syntax for being able to put together test cases. And so using the combinations within assertThat is a great way of really simplifying your test.   
*In the AssertThatCombination.java file, the presenter focuses on the code  
  
   @Test  
   public void \_Either()  
   {  
       String tested = "(side bar)!";  
       assertThat("Either", tested, either(endsWith("!"))  
                                       .or(startsWith("("))  
                                       .and(notNullValue()));  
   }  
     
}  
  
The presenter highlights the elements either(endsWith("!")), or(startsWith("("), and(notNullValue())).The presenter then navigates to the code line String tested = "(side bar)!", and changes it to String tested = null;. The full code for the relevant test is now as follows:  
  
   @Test  
   public void \_Either()  
   {  
       String tested = null;  
       assertThat("Either", tested, either(endsWith("!"))  
                                       .or(startsWith("("))  
                                       .and(notNullValue()));  
   }  
     
The presenter clicks Run. On the JUnit tab, the \_Either test is shown to have failed, with output that includes an AssertionError message.   
  
The presenter changes the line String tested = null line back to  
  
String tested = "(side bar)!"  
  
The full code for the relevant test is now:  
  
 @Test  
   public void \_Either()  
   {  
       String tested = "(side bar)!";  
       assertThat("Either", tested, either(endsWith("!"))  
                                       .or(startsWith("("))  
                                       .and(notNullValue()));  
   }  
     
}  
  
The presenter then clicks Run and the test passes.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Collections and assertThat

Learning Objective

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

* *understand how to use assertThat methods with collections in JUnit tests*

**1. Using assertThat with collections**

Dealing with a large amount of data via a collection in Java coding is actually quite simple. Validating that data however can be rather complex. Using assertThat, we can help to simplify working within collections during testing. Now, you can see we can check data individually as well as a whole within a collection. So, for this Test we're cheating a little bit, we're starting with an array, but we're immediately turning that array into a Collection. And we've changed into a list and we're treating that list as a Collection, just to kind of show you that we can work with many different types of collections, within the Test we're doing inside of here. So, my first assertThat here is saying, hey, does the Collection you're working with, you know the test that the Collection inside of here have some specific item in it. So, we're saying, does it have the value "c" in it. And you can see, because they all pass right now it certainly does have that item. If it didn't, like it was asking if there's an "f" inside of here for instance, it would fail, it's not inside that Collection. So, this works not just for strings, this works for any data types. It's going to go through whether it's a string, or whether it's a custom object, or whether it's any data object that's coming in, it's going to see is this collection containing it. It's going to do the looping and all that extra work for you, so you don't have to code that yourself.   
*In Eclipse, a file named AssertThatCollections.java is open and the JUnit tab lists four tests – \_CheckCollection, \_CollectionContains, \_CollectionFilter, and \_CollectionSearch.   
  
The AssertThatCollections.java file contains the following code:   
  
import static org.hamcrest.CoreMatchers.endsWith;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class AssertThatCollections  
{  
   @Test  
   public void \_CollectionContains()  
   {  
       String [] testArray = {"a", "b", "c", "d", "e"};  
       Collection<String> tested = Arrays.asList(testArray);  
       assertThat("Is the item contained", tested, hasItem("c"));  
   }  
  
   @Test  
   public void \_CheckCollection()  
   {  
       String [] testArray = {"a", "b", "c", "d", "e"};  
       List<String> tested = Arrays.asList(testArray);  
       assertThat("Any item meets the criteria", tested, hasItems("b", "e"));  
   }  
     
   @Test  
   public void \_CollectionSearch()  
   {  
       String [] testArray = {"dog", "cat", "tiger", "mouse", "unicorn"};  
       Set<String> tested = new HashSet<String>(Arrays.asList(testArray));  
       assertThat("Any item meets the criteria", tested, hasItem(startsWith("uni")));  
   }  
     
The presenter focuses on the code:  
  
   @Test  
   public void \_CollectionContains()  
   {  
       String [] testArray = {"a", "b", "c", "d", "e"};  
       Collection<String> tested = Arrays.asList(testArray);  
       assertThat("Is the item contained", tested, hasItem("c"));  
   }  
  
The presenter highlights the lines   
  
String [] testArray = {"a", "b", "c", "d", "e"};  
Arrays.asList(testArray);  
       assertThat("Is the item contained", tested,  
  
On the JUnit tab, an icon shows that the test \_CollectionContains passed.   
  
The presenter navigates to the code line  
  
assertThat("Is the item contained", tested, hasItem("c"));  
  
He changes the line to  
  
assertThat("Is the item contained", tested, hasItem("f"));  
  
The full code for the relevant test is now   
  
 @Test  
   public void \_CollectionContains()  
   {  
       String [] testArray = {"a", "b", "c", "d", "e"};  
       Collection<String> tested = Arrays.asList(testArray);  
       assertThat("Is the item contained", tested, hasItem("f"));  
   }   
  
The presenter clicks Run. On the JUnit tab, an icon shows that the test  \_CollectionContains failed, and its output includes an AssertionError message.  
  
The presenter returns to the line  
  
assertThat("Is the item contained", tested, hasItem("f")  
  
He changes the line back to  
  
assertThat("Is the item contained", tested, hasItem("c"));  
The presenter clicks Run and the test passes.*   
  
There's many different checks we can do with this now. We say assertThat, we have another collection inside of here, same data that were coming in, not only does it have one item. Let's say does it have multiple items, we want it to say does it have both "b" and "e" now. And you can see this guy indeed has "b" and "e" and so it passes. If I have added more items to this list, I'd have to make sure all those items are inside of there, and it's going to give me that quick check. And that syntax is fantastic for that, I'm just simply adding additional parameters for that. And this is another one of those really cool Java methods, that I can have any number of that I want. I don't have to have 1 or 2 or 5, I can have 50 or 20 or 2 or 1. You can see by the syntax we have here, that's going to go through and check the individual items are located within our collection. Now, we can also go through and check details within those items, we can use the matchers and nest those matchers. So, we have the next check down inside of here. We have a different, a little bit more interesting array here, longer values and we're saying, hey does this have an item, you know the same check we have out there, look within another matcher here that startsWith these three characters. So, are there any unis out there, yeah there is a "unicorn" for sure. And so if I change my "unicorn" to a "bicorn" out here, then all of a sudden when I run my test it doesn't work anymore, it doesn't contain that value out there anymore. So, whatever check is going to be here I can nest these checks, I can put them together just like I can with any matchers.   
*In the AssertThatCollections.java file, the presenter focuses on the code  
  
   @Test  
   public void \_CheckCollection()  
   {  
       String [] testArray = {"a", "b", "c", "d", "e"};  
       List<String> tested = Arrays.asList(testArray);  
       assertThat("Any item meets the criteria", tested, hasItems("b", "e"));  
   }  
  
The presenter clicks hasItems in the code and a drop-down list provides two options. The options inlude, hasItems(Matcher<? super T> ... itemMatchers) : Matcher... and hasItems(T ... items) : matcher<Iterable<T>> CoreMatchers.   
  
The presenter then focuses on the code  
  
   @Test  
   public void \_CollectionSearch()  
   {  
       String [] testArray = {"dog", "cat", "tiger", "mouse", "unicorn"};  
       Set<String> tested = new HashSet<String>(Arrays.asList(testArray));  
       assertThat("Any item meets the criteria", tested, hasItem(startsWith("uni")));  
   }  
     
In the code, the presenter highlights the elements:   
  
{"dog", "cat", "tiger", "mouse", "unicorn"};  
hasItem(startsWith("uni")));  
  
The presenter then changes the code line String [] testArray = {"dog", "cat", "tiger", "mouse", "unicorn"}; to String [] testArray = {"dog", "cat", "tiger", "mouse", "bicorn"};. the full code is now  
  
   @Test  
   public void \_CollectionSearch()  
   {  
       String [] testArray = {"dog", "cat", "tiger", "mouse", "bicorn"};  
       Set<String> tested = new HashSet<String>(Arrays.asList(testArray));  
       assertThat("Any item meets the criteria", tested, hasItem(startsWith("uni")));  
   }    
  
  
  
The presenter clicks Run. The JUnit tab shows that the \_CollectionSearch test has failed, with output that includes an AssertionError message.   
  
The presenter then changes the code line, String [] testArray = {"dog", "cat", "tiger", "mouse", "bicorn"};. the full code is now:  back to:  {"dog", "cat", "tiger", "mouse", "unicorn"}; hasItem(startsWith("uni")));. The full code for the relevant test is now   
  
   @Test  
   public void \_CollectionSearch()  
   {  
       String [] testArray = {"dog", "cat", "tiger", "mouse", "unicorn"};  
       Set<String> tested = new HashSet<String>(Arrays.asList(testArray));  
       assertThat("Any item meets the criteria", tested, hasItem(startsWith("uni")));  
   }  
     
The presenter highlight the code elements {"dog", "cat", "tiger", "mouse", "unicorn"}; and hasItem(startsWith("uni")));.  
  
The presenter then clicks Run. The JUnit tab shows that the test has passed.*   
  
Now, the really cool thing inside here is the everyItem. So let's say I want to go to each item in the array and make sure they all conform to, or perhaps don't conform to, I can use the not inside of there. Some rule is going on and this is exactly what I'm doing here. We're saying for everyItem in the collection it should not end with "fish". And look how readable that syntax is using the matcher, so everyItem should not end with "fish". Once you figure out how the syntax works, it's really, really easy to code it, really easy to read. And so again our list has no fishes inside of here. If all of a sudden, I'm adding in a "dogfish" and a "catfish" and a "tigerfish", and I have several of them in there, then guess what it's no longer going to pass that test. We don't like "fish" inside of our collection here and so we're going to get them out.   
*The presenter scrolls down to reveal the code  
  
  @Test  
   public void \_CollectionFilter()  
   {  
       String [] testArray = {"dog", "cat", "tiger", "mouse", "unicorn"};  
       List<String> tested = Arrays.asList(testArray);  
       assertThat("All items meets the criteria", tested, everyItem(not(endsWith("fish"))));  
   }  
}  
  
The presenter highlights the code element everyItem(not(endsWith("fish").  
  
The presenter then changes the code line String [] testArray = {"dog", "cat", "tiger", "mouse", "unicorn"};  to  
  
String [] testArray = {"dogfish", "catfish", "tigerfish", "mouse", "unicorn"};  
  
The full code for the relevant test is now  
  
@Test  
   public void \_CollectionFilter()  
   {  
       String [] testArray = {"dogfish", "catfish", "tigerfish", "mouse", "unicorn"};  
       List<String> tested = Arrays.asList(testArray);  
       assertThat("All items meets the criteria", tested, everyItem(not(endsWith("fish"))));  
   }  
}  
  
The presenter clicks Run and the JUnit tab shows that the \_CollectionFilter test has failed, with an AssertionError message in its output.    
  
The presenter then removes the changes to the code. The code for the text is now  
  
  @Test  
   public void \_CollectionFilter()  
   {  
       String [] testArray = {"dog", "cat", "tiger", "mouse", "unicorn"};  
       List<String> tested = Arrays.asList(testArray);  
       assertThat("All items meets the criteria", tested, everyItem(not(endsWith("fish"))));  
   }  
}*   
  
I want to go back and point out through my tests, notice up here I started with a list. I have either a Collection posing as a list, I have straight list itself, they work just fine. I can also work with sets, I took my array turn into a list, threw that into a set and I can work with sets inside of there, and again the last one is just a list. So, all the different basic types of collections I can use within this mechanism. And it doesn't matter what type I'm using, that's the beautiful thing about it. I can just test, hey this bunch of data you have out there, does it conform to whatever rule I'm making up. So, the assertThat working within our collections gives us a great way of passing some tests, or failing some tests, or just making those checks we need to do, even when our data is hidden within a collection.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Custom assertThat Matchers

Learning Objective

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

* *understand how to use custom assertThat matchers in JUnit tests*

**1. Using custom assertThat matchers**

As we customize our code, particularly data types, it's often valuable to customize our testing as well. The assertThat mechanism in JUnit, allows us a great deal of flexibility for providing custom validation rules, and then applying them within the standard matcher syntax. As we see here we have some basic testing that uses our CustomMatcher. Now let's start by taking a look at that CustomMatcher. In order to build a CustomMatcher we have to extend out the BaseMatcher. And the BaseMatcher is provided for us within JUnit. You can see actually it's in the hamcrest framework part of the JUnit. And we need to tell it which type of class we're going to be testing within this. And so once we say, hey we have the SampleClass it's our class, our customized data type we created. We want to be able to use that and make a test specifically for that. And that test becomes reusable for every type of test that wants to work with that data from now on. And so when we go off and create the Matcher, we're handed in the sample test that we're going to go through and workout. So, we have to have this static Matcher method here to setup and get our Matcher ready, it needs to return off the Matcher we have working on here. And so we go through and build an instance of our own little CustomMatcher within a static instance, that's the first little rule that helps us to build this Matcher more quickly.   
*In Eclipse, two files are open on separate tabbed pages – AssertThatCustom.java, which is currently active, and CustomMatcher.java. The JUnit tab lists two test cases – basicCustomMatcher and customMatcherAdvanced.    
  
The AssertThatCustom.java file contains the following code:   
  
package myapp.junit.asserts.that;  
  
import static org.hamcrest.CoreMatchers.either;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class AssertThatCustom  
{  
  
   @Test  
   public void basicCustomMatcher()  
   {  
       SampleClass tester = new SampleClass(29, "custom matcher test");  
       SampleClass expected = new SampleClass(1, "custom");  
         
       assertThat("Run a custom matcher", tester, CustomMatcher.excitedSample(expected));  
   }  
     
   @Test  
   public void customMatcherAdvanced()  
   {  
       SampleClass tester = new SampleClass(29, "custom matcher test");  
       SampleClass expected = new SampleClass(1, "custom");  
         
       assertThat("Run a custom matcher", tester, either(CustomMatcher.excitedSample(expected)).or(equalTo(expected)));  
   }  
}  
  
The presenter switches to the CustomMatcher.java file. It contains the code   
  
package myapp.junit.asserts.that;  
  
import myapp.junit.asserts.SampleClass;  
  
import org.hamcrest.BaseMatcher;  
import org.hamcrest.Description;  
import org.hamcrest.Matcher;  
  
public class CustomMatcher extends BaseMatcher<SampleClass>  
{  
   private SampleClass expected;  
     
   public static Matcher<SampleClass> excitedSample(SampleClass expected)  
   {  
         
       return new CustomMatcher(expected);  
   }  
     
   public CustomMatcher(SampleClass expected)  
   {  
       this.expected = expected;  
   }  
  
   @Override  
   public boolean matches(Object o)  
   {  
       SampleClass s = (SampleClass) o;  
       if (s.getSomeNumber() > 10)  
       {  
           return s.getSomeString().contains(expected.getSomeString());  
       }  
       return true;  
   }  
  
   @Override  
   public void describeTo(Description d)  
   {  
       d.appendText("Customize description ");  
   }  
}  
  
The presenter focuses on the code  
  
public class CustomMatcher extends BaseMatcher<SampleClass>  
{  
   private SampleClass expected;  
     
   public static Matcher<SampleClass> excitedSample(SampleClass expected)  
   {  
         
       return new CustomMatcher(expected);*   
  
Now, the constructor then takes the expected value to what we're expecting to see, as part of our custom match. Let me store that off to be able to do our test, and what we're expected to do is to provide a couple overrides, the matches is the main one. So you hand us whatever Object we're going to check against, and we're going to see if it needs the rule to match our expected value. Now, obviously we're not doing something as simple as equals to. If we're going to do that we just use the equalsTo that exist, we have our own custom rule inside here. So, the custom rule I made up in this case, is if whatever number you passed in is greater than 10, instead of having to match that text of some string exactly, it just need to contain the text of the expected string that we have going on there. So, we're going to have more text out there, just need to contain this text of the expected value. And as long as it does that this will return true, if it doesn't it'll return false. So, up here then you can see, the describe is going to give us a custom description of what this failure means. If something goes wrong, it allows us to give some extra text, to give some context for what this test is doing. In this case, I'm not giving a lot of context because I have just made it up, but I want to show you this description as we go through and look at the test work.   
*The presenter highlights the code  
   
   public CustomMatcher(SampleClass expected)  
   {  
       this.expected = expected;  
   }  
  
The presenter then highlights the code elements matches(Object o) and s.getSomeNumber within the following code:  
  
   @Override  
   public boolean matches(Object o)  
   {  
       SampleClass s = (SampleClass) o;  
       if (s.getSomeNumber() > 10)  
       {  
           return  
  
Next the presenter focuses on the code  
  
   @Override  
   public void describeTo(Description d)  
   {  
       d.appendText("Customize description " );  
   }  
}  
  
The presenter highlights the string "Customize description".*  
  
So, in the proper scenario our expected values here you can see doesn't really matter the data here, we're not looking at the number, but we're saying it must contain the value custom. And so for the SampleClass up here when we run our test we pass in the value, and then we give a new CustomMatcher here that's working with the expected value. Now, in the hamcrest we would not see this precursor here. If we wanted to get rid of this we could put that static import up above, but I wanted to just see where this was coming from. This is that static call that we created here to go and create our CustomMatcher for us, so I don't have to create an instance of it myself inside here. I want it to match up with the framework of how the rest of the matchers work. And so I got this and I have my excitedSamples. When I run my test you can see it obviously passes. This guy up here is above 10 and it has that value inside of it. If it wasn't above 10, the test results are going to be the same, because it's not checking inside of there. But if I don't, if I change the value that's inside of here, if I make this, I custom with not the text inside of there it doesn't match anymore, then all of a sudden my assertion fails.   
*The presenter returns to the AssertThatCustom.java file. He focuses on the code  
   @Test  
   public void basicCustomMatcher()  
   {  
       SampleClass tester = new SampleClass(29, "custom matcher test");  
       SampleClass expected = new SampleClass(1, "custom");  
         
       assertThat("Run a custom matcher", tester, CustomMatcher.excitedSample(expected));  
   }  
  
The presenter highlights CustomMatcher in the code.   
  
The presenter switches to the CustomMatcher.java file and highlights the code  
   
   public static Matcher<SampleClass> excitedSample(SampleClass expected)  
   {  
         
       return new CustomMatcher(expected);  
   }  
  
The presenter then returns to the AssertThatCustom.java file.  
  
The JUnit tab shows that two tests, basicCustomMatcher, and customMatcherAdvanced, have passed.   
  
In the AssertThatCustom.java file, the presenter navigates to the line  
  
SampleClass tester = new SampleClass(29, "custom matcher test");  
  
He changes the line to  
  
SampleClass tester = new SampleClass(9, "custom matcher test");  
  
The full code in the relevant test is now  
  
   @Test  
   public void basicCustomMatcher()  
   {  
       SampleClass tester = new SampleClass(9, "custom matcher test");  
       SampleClass expected = new SampleClass(1, "custom");  
  
The presenter clicks Run. The JUnit tab indicates that all the tests have passed.  
  
The presenter changes the line SampleClass(1, "custom");  to SampleClass(1, "customf"); The full code for the relevant test is now  
  
   @Test  
   public void basicCustomMatcher()  
   {  
       SampleClass tester = new SampleClass(9, "custom matcher test");  
       SampleClass expected = new SampleClass(1, "customf");  
  
The presenter clicks Run. The JUnit tab shows that the basicbasicCustomMatcher test has failed.*   
  
And as I'm looking at the failure you can see this has AssertionError. We ran a CustomMatcher inside of here and expected it's coming through here, and it's giving us some custom descriptions inside of here. Now, ideally when we say expected inside here for the description, we would go through and do more than just say out here. We'd be expecting and we could say, for example we got a expected.getSomeString, and Got, then the way that the matchers work on the other one. And we would say the value that was coming in, you know it was coming in inside of there. Well, we'll just leave it that for now, we don't get more complex so we spend more time in that. But let me go through and run this again and you can see as I said expected and then customf, but was and then its showing something else inside of here. So, I would have to customize a little bit further to show exactly inside of there what's going on. All right, so that's the basic one. Now, as we build our CustomMatcher we're not limited, we can use that CustomMatcher with either or logic, or all of logic, or whatever other matchers are going to combine that. And we're creating an instance of a matcher that can work with any other matcher out there. So, this is a really powerful way to expand your testing vocabulary, to take your custom data either objects and be able to test with them appropriately, with standard reusable tests.   
*The presenter points out the following error message in the output for the basicbasicCustomMatcher test:  
  
java.lang.AssertionError:Run a custom matcher  
Expected:Customize description  
     but was <myapp.junit.asserts.SampleClass@53c4c653>  
  
The presenter switches to the CustomMatcher.java file. He focuses on the code  
  
 @Override  
   public void describeTo(Description d)  
   {  
       d.appendText("Customize description");  
   }  
  
The presenter changes the line d.appendText ("Customize description"); to  
  
d.appendText("Customize description " + expected.getSomeString() );  
  
The presenter switches to the AssertThatCustom.java file and clicks Run. The JUnit tab shows that the test has failed. The output includes the following error message:  
  
java.lang.AssertionError: Run a custom matcher Expected: Customize description customf but was <myap.junit.asserts.SampleClass@53c4c653>*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Assume versus Assert

Learning Objective

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

* *contrast Assume with Assert and use Assume statements in JUnit tests*

**1. Assume versus assert statements**

Buggy test data can be as big of an issue as buggy code, and it can equally slow down a project. JUnit helps to filter out bad test cases, by allowing the test designer to declare assumptions about the test data, within a test case. Assume statements give us a precheck, to check that the state of a test is acceptable before moving on. So we're going to look at a bunch of different assumes. Now, assume is mostly like an assert. In an assert, it will fail a test if it doesn't come up true. So an assume, we're going to go check to say "hey, is the test data okay?" So inside of the assume, we say "okay, assumeTrue or assumeFalse." We're saying whatever the result is here, and it could be a combination of logic, it could be something I've received from input. Whatever it is, I assume these are true or false, and the text gives me the warning of – if something went wrong – what this means. So these are real simple. I can check notNull, say "hey, here's one or many." You can see our notNull takes advantage of that cool Java feature of having multiple parameters – an array of parameters that goes in, that allows me to check all of these things. So if I pass in four parameters, I'm expecting them all not to be null. I can check them here and assume that they are not null.   
*In Eclipse, a file named AssumeSamples.java is open and the JUnit tab lists five test cases – matcherWithThat, badInputData, no Exception, assumeBoolean, and notNull.  
  
The AssumeSamples.java file contains the following code:   
  
package myapp.junit.assumes;  
  
import static org.hamcrest.CoreMatchers.containsString;  
  
import org.junit.Before;  
import org.junit.Test;  
  
public class AssumeSamples  
{  
   @Test  
   public void assumeBoolean()  
   {  
       assumeFalse("This should be false", false);  
       assumeTrue("This should be true", true);  
       System.out.println("Boolean ok");  
   }  
  
   @Test  
   public void notNull()  
   {  
       assumeNotNull("one", "two", "many more");  
       System.out.println("None were null");  
   }  
  
   @Test  
   public void noException()  
   {  
       try  
       {  
           Integer.parseInt("1");  
       } catch (NumberFormatException e)  
       {  
           assumeNoException("Should not fail", e);  
       }  
       System.out.println("No exception happened");  
  
The presenter initially focuses on the code  
   @Test  
   public void assumeBoolean()  
   {  
       assumeFalse("This should be false", false);  
       assumeTrue("This should be true", true);  
       System.out.println("Boolean ok");  
  
The presenter then focuses on the code  
   @Test  
   public void notNull()  
   {  
       assumeNotNull("one", "two", "many more");  
       System.out.println("None were null");  
   }*   
  
If I'm doing something within my testing that I need to check for an exception, I can go through it and say "hey, I assumed that an exception wasn't going to happen, but it did, and so I'm going to stop executing it." So this is most likely going to be your setting up data for the test. I'm opening up a file to be able to send to the test. If the failure, if the exception happens inside the test itself, it's probably going to be a straight fail. But this is saying "hey, something happened in the test case that – I assumed there was no exception, but one did happen." We could do matchers inside of assume as well. assumeThat uses any matcher that I would use, just like if I was doing assert. And so those matchers might be any check, or any combinations of checks, but I can check my test data using that powerful matcher mechanisms the same way. Now, the question is, is what happens if the test data doesn't align? So you see, all of those up above, they run just fine. When I'm checking criteria, turns out everything is okay, turns out no exception happened, turns out the Booleans and none of them are null, all those just keep running the test cases when they pass.   
*The presenter focuses on the code  
  
@Test  
   public void noException()  
   {  
       try  
       {  
           Integer.parseInt("1");  
       } catch (NumberFormatException e)  
       {  
           assumeNoException("Should not fail", e);  
       }  
       System.out.println("No exception happened");  
  
In the code, the presenter highlights the line  
  
assumeNoException("Should not fail", e)  
  
The presenter then scrolls down to reveal the following code:   
  
@Test  
   public void matcherWithThat()  
   {  
       assumeThat("check any criteria", "my string", containsString("my"));  
       System.out.println("That turned out OK");  
   }  
     
   @Test  
   public void badInputData()  
   {  
       int testData = 0;  
       assumeThat("Must not be zero", testData, not(equalTo(0)));  
         
       int answer = 10/testData;  
   }  
  
The presenter then highlights the line  
  
assumeThat("check any criteria", "my string", containsString("my"));*   
  
Well when they fail, they do not fail the test case. If I look at bad data input over here, it didn't fail. I've zero errors out of here, but I have one that was skipped. I ran all 5 of them, but one of them I skipped, because I had an assumption inside of here. And the assumption says it must not be zero, expected zero not zero. Okay, so in this case, my 'must not be zero' – that's the rule I set up. I'm saying this data that's coming in, the parameters must not be zero, the data load of the database must not be zero. Whatever is going to happen, I set this up as part of my test case. If I get a zero as part of the data coming in, I don't want to fail the test case. The unit under test did not fail, my test design failed. And so I can test my test, if you will, in that the data is coming in, where data is coming from, and then it'll give me some granularity in reporting my results. So I'm not failing the test, I'm failing the test execution. And so instead of giving the error to my developers to work on, I'm giving my error to my testers to work on – if that's the way you're using it. At least, as a developer, at least I know I need to go and fix my test cases, if I'm the one building these things.   
*On the JUnit tab, the presenter selects the badInputData test, which failed. The output includes the following message:  
  
org.junit.internal.AssumptionViolatedException: Must not be zero: got: <0>, expected: not <0>  
  
The presenter navigates to the code  
  
  @Test  
   public void badInputData()  
   {  
       int testData = 0;  
       assumeThat("Must not be zero", testData, not(equalTo(0)));  
         
       int answer = 10/testData;  
   }  
  
The presenter then scrolls down to reveal the code  
  
   @Before  
   public void assumeInSetup()  
   {  
       assumeTrue("It works in setup too", true);  
   }  
}*   
  
So the assumes are not only limited to being used inside of the tests. I can put an assume inside of a setup – before, after, wherever I want to put inside of there as well. So assumeTrue inside of here, it works in the setup too. The difference is if I fail this, if I say…if I change this true to false, if something fails the assumptions, it's going to be executed before every single one of the test cases. And so if it doesn't change, like in this case it's for all the test cases, I end up skipping all the test cases. I don't execute any of them, nothing gets printed out here. But remember, the before runs before each test, so it's not guaranteed to fail all of them. It's only going to fail the ones that have that be the case as they are executing. So if there is something that validates and changes between each one of them, some of them will pass, some of them will fail. But the assume gives us a powerful way to differentiate between test failures inside the unit under test, versus test failures inside the test case itself. So use the assumes to that great affect.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Setting up Test Cases

Learning Objective

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

* *configure JUnit test cases*

**1. Configuring JUnit test cases**

Most features we develop require more than a single test, but instead a series of test cases, which often derive from a common setup, JUnit can be constructed to include this common setup and manage execute before, and even make reusable across to all your test cases all of that manage by JUnit. So, in this test we have here, we have a really simple test, we're going to go play with this SimpleDateFormat. And we can have any number of setups that we need to do for this. So, we need to setup for this some test data, but we also need to setup our simple test object inside here. And so traditionally, we'd set this up using some methods, and we can continue to do that within JUnit. But now instead of having to conform to some fancy rule of what setup looks like, I'm using the Before annotation and say, hey this method needs to be run before each test case. And so when I go and run each of the Test cases and in this guy I have two Test cases, they are both passing. But as we run the Test cases, you can see when I put my System.out.println here, it runs twice. Each test gets its own copy, each test gets the data resetup, so you don't have to worry about cleaning up data or anything like that. It's going to automatically get its own fresh copy each way along the road, and I can have any number of these Before statements.   
*In Eclipse, a file named Setuptest.java is open and the JUnit tab lists two test cases – badFormatTest and format.   
  
The Setuptest.java file contains the following code:  
  
p.junit.before;  
  
import java.text.ParseException;  
  
@FixMethodOrder(MethodSorters.NAME\_ASCENDING)  
public class SetupTest  
{  
   private String testDate;  
   private SimpleDateFormat format;  
     
   @Before  
   public void setupFormat()  
   {  
       format = new SimpleDateFormat("mm/DD/yyyy");  
   }  
  
   @Before  
   public void setupDate()  
   {  
       testDate = "01/01/2000";  
       System.out.println("Data ready");  
   }  
     
   
   @Test  
   public void format() throws ParseException  
   {  
       Date test = format.parse(testDate);  
   }  
  
   @Test(expected=ParseException.class)  
   public void badFormatTest() throws ParseException  
   {  
       testDate = "abcd";  
       Date test = format.parse(testDate);  
   }  
  
The presenter initially focuses on the code  
   @Before  
   public void setupFormat()  
   {  
       format = new SimpleDateFormat("mm/DD/yyyy");  
   }  
  
The presenter then points out the two @Test cases.  
  
The presenter clicks Run. The JUnit tab shows that both tests have passed.*   
  
If I want to I can have another method inside of here, public void myFunSetup inside of here. And this can do anything I want to do, I just do a System.out.println again to say "I ran too". Now, if I don't do anything, if I just have the method out here JUnit doesn't know what to do. Nothing is changed in our output, everything runs exactly as it has before, because it's that magic Before that I need inside of here that I'm going to add inside here. So, you can see this is out of the JUnit class here. The Before annotation lets it know this guy should run beforehand and it runs before each test item. It doesn't necessarily give a predictable way of knowing what order these run in. So ideally, if they are order dependent you should probably code them as such. But if they are totally independent of each other, it doesn't matter which order they run in as these just don't really, they just only to be gets done individually. This is a great chance to be able to separate out the logic, separate out in the methods, and that have any number of this Before methods to go and set everything up.   
*The presenter navigates to immediately above the code  
  
   @Test  
   public void format() throws ParseException  
   {  
       Date test = format.parse(testDate);  
   }  
  
He then adds the code  
  
   @Before  
   public void myFunSetup()  
   {  
       System.out.println("I ran too");  
   }  
  
The presenter clicks Run and the JUnit tab lists myapp.junit.before.SetupTest as having passed.*   
  
And it doesn't matter whether your Test is just doing a simple test, or whether you're doing fancy exception handling, or whatever happens inside the test, JUnit is going to manage the flow. It's going to allow you to setup your Test upfront, execute each test and then have that go through again and again for each test that happens. And just to show that that works just the same, I'm going to go through one last time and say, I can have anyNumberOfTests. And I'm going to go and add in additional test inside of here. And then when the test can do nothing, I'm just going to put a Test annotation inside of here, just to show you as we add in the one additional Test. I have an extra bit of a curly bracket inside of there, but I'm adding in the extra test inside of here. When I run this, I have an additional test that's running and again I have Data Ready, I Ran, Data Ready, I Ran and then the third time Data Ready and I Ran. Each one runs and each one gets its own copy, each one is executed. So, within our Test design we had to setup data, we had to have predictable data, and now we have a very predictable way of knowing that that data has been setup. We can code it in a very modular way, and we can let JUnit know through a simple annotation this guy needs to run Before the test. And it needs to setup that data and we need to run that to have everything run great.   
*The presenter scrolls down and focuses on the code  
   @Test(expected=ParseException.class)  
   public void badFormatTest() throws ParseException  
   {  
       testDate = "abcd";  
       Date test = format.parse(testDate);  
   }  
     
Immediately below this, the presenter adds the following code:  
   @Test  
   public void anyNumberOfTests()  
   {  
            
   }  
}  
  
The presenter clicks Run and the new test anyNumberOfTests is listed on the JUnit tab as having passed.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Test Fixtures

Learning Objective

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

* *work with JUnit test fixtures*

**1. Configuring JUnit test fixtures**

To maximize the productivity in writing and executing test cases, we want to design test fixtures that provide common solutions to how we set up tests, execute, and then clean up afterwards. JUnit describes this testing process, and gives us hooks to allow us to create our own test fixtures. So this is a complex test. It's going to be testing a data access object, which goes to a database, it does database interactions, and it's kind of an expensive test. Whether I am doing services or databases, there is a lot involved in getting that test set up. And if I create a connection to the database for every little, tiny, minor test I do, that would be a very slow test and be very server-resource intensive. So I don't want to do that. So instead, JUnit gives us the option to be able to set up something for all the test cases. And so we will come back to this test in a second. But let's go over here and look at how that works. So using our annotations, we can see there is a bunch of annotations like BeforeClass. So the BeforeClass is saying, before I run any other tests inside of this class, I want to do this setup. I am going to do it one time for the whole class. Then I have the Before annotation, which runs before each test. Obviously the Test annotation, which is the tests that are going to go on – I have any number of these – I can have any number of Before – I can actually have any number of any of these annotations. Then a number tests inside of here. And then I have the After annotation. We are just saying after each test, do this cleanup.   
*In Eclipse, two files are open on separate tabbed pages – BeforeCallDemo.java and TestRealEstateDAO.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();  
   }  
  
The presenter switches to the BeforeCallDemo.java file. It contains the following code:   
  
import org.junit.After;  
  
  
public class BeforeCallDemo  
{  
   @BeforeClass  
   public static void showBeforeClass()  
   {  
       System.out.println("BeforeClass");  
   }  
  
   @Before  
   public void showBefore()  
   {  
       System.out.println("Before");  
   }  
  
   @Test  
   public void one()  
   {  
       System.out.println("one");  
   }  
  
   @Test  
   public void two()  
   {  
       System.out.println("two");  
   }  
  
   @Test  
   public void three()  
   {  
       System.out.println("three");  
   }  
     
   @After  
   public void showAfter()  
   {  
       System.out.println("After");  
   }  
  
   @AfterClass  
   public static void showAfterClass()  
   {  
       System.out.println("AfterClass");  
   }  
}  
  
The presenter focuses on the various annotations in the code, including @Test, @BeforeClass, @Before, and @After.  
  
The presenter clicks Run and the JUnit tab lists three tests – one, two, and three – as having passed.*   
  
And then finally, the AfterClass annotation says that after you have run all the tests that you want to do, here is the final step you want to get to complete it off. And so for this little, quick, simple test, I just simply have some print lines, just so we can see the flow of this. When I go and run this, obviously everything passes – there is no checks to make it fail. But if I look at the flow of this guy running, you can see BeforeClass runs one time. Then the Before runs before the first test, and then the After runs after the first test. Before runs again before the second test, and then After runs after that second test…Before third test and then After. And then finally, after all that's done, we have the After test – the Before and After class sandwiching in all the testing that happens inside of there. So that's the hooks that JUnit provides. That's the process it provides for creating your test cases. So going back to our data access object, we can see in this case, our Before is setting up our hibernate framework. Hibernate is a database connection framework – it's not really important – but we are setting that guy up. And then before each different test that's going to go inside of there, we are going to go and set up the data access object we are going to use.   
*The presenter highlights the annotation @AfterClass.  
  
The presenter opens the Console pane below the file. It lists the output  
  
BeforeClass  
Before  
one  
After  
Before  
twoAfter  
Before  
three  
After  
AftrerClass  
  
The presenter switches to the TestRealEstateDAO.java file and focuses on the code  
  
   @Before  
   public void setupDAO()  
   {  
       dao = new RealEstateDAO(config.getSessionFactory());  
   }*   
  
And so setting up the hibernate framework, just like connecting to the database, is kind of labor intensive. It's got to load some config files, it's got to connect to the database, it's got to create connections; it's got to get those all set up and ready to use. But for each test, we want our own session; we want our own object to go and test with, so there is no state. And so this works great. I set up the framework, and then I set up each object I am going to work with – which is exactly what your solution would do in the…in the wild…the one that's actually running in production. And so then, again, I can have any number of Befores, I can create some value I am going to be playing with, something I am going to be working on. And then I am going to go through, and after the test is over, I can do something like wiping the table. So I can put stuff into the database table – I might read, I might write, I might update, I might delete – who knows what it's going to do. But whatever it's going to do, afterwards I want to go wipe that stuff out of the database table. And so I am…again, opening up a transaction, I am doing some SQL, delete it, and then I am committing to get off – whatever I need to do to wipe it up afterwards.   
*The presenter scrolls down to reveal the following code:   
  
@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));  
   }  
  
The presenter highlights the code  
  
@After  
   public void wipeTable()  
   {  
       Session session = config.getSessionFactory().getCurrentSession();  
       session.beginTransaction();  
       session.createSQLQuery("delete from RealEstate").executeUpdate();  
       session.getTransaction().commit();  
   }*   
  
Now after all the tests are over with, after all that class work is done, I want to close the database. So I want to clean up my database connection, turn that up, close that out properly, and go on from there. And so the tests, you know, at this point, are almost irrelevant…we are not doing…but we can see we can add, we can update, we can load and delete, we can load everything…all the different tests we want to have going on with inside of there. But they are all running within a common environment where they are set up, shut down as a class, and individually as test cases, in a way that we can predict and we can know what's working within our system. And whether I add five more test cases or a hundred more test cases, we know they are all built in an environment that's stable and predictable, and is not going to break down the line as we do more testing. Or if it does, we have one place to fix it. So using these tests fixtures really helps us design tests and build tests in a stable, predictable, and easier-to-use environment.   
*The presenter then focuses on the code  
  
   @AfterClass  
   public static void closeDatabase()  
   {  
       config.getSessionFactory().close();  
   }  
  
The presenter highlights the line  
  
config.getSessionFactory().close();  
  
The presenter scrolls down to reveal the code  
  
@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;  
   }  
}  
  
In the code, the presenter focuses on the methods dao.add(sample), dao.update(sample), dao.delete(sample), dao.loadAll(), and createRealEstate().*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

Modifying Java Code to Ensure Proper Testing

Learning Objective

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

* *practice modifying Java code to ensure proper testing scenarios*

**1. Modifying Java code for testing**

It's time to put what you've been learning into practice. See how you do with the exercise that follows.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_jtut_a01_it_enus/output/html/course_transcript.html#top)

© 2017 Skillsoft Ireland Limited