CL309
Object Oriented Analysis and Design Lab
Lab#07

JavaDocs and JUnit Testing

4.1. Using JavaDocs

4.1.1. Introduction

Have you ever had an experience of reading your code months after you last worked on it? You probably had problems remembering exactly what the code does. We can actually do two things to avoid such a situation. One is to strive to make your program simple and readable. The other is to write good documentation.

Javadoc is a convenient, standard way to document your Java code. Javadoc is actually a special format of comments. There are some utilities that read the comments, and then generate HTML document based on the comments. HTML files give us the convenience of hyperlinks from one document to another. Most class libraries, both commercial and open source, provide Javadoc documents.

In this tutorial, we are going to learn several ways to write Javadoc comments with the assistance of Eclipse, and generate HTML files with the Javadoc Export WIzard. There are several tags for Javadoc comments. However, you don't have to try to memorize them all.

Note: Most frequently used tags can be generated by the code template, and you can find others with Content Assist by pressing Ctrl-SPACE.

4.1.2. Types of Javadoc

There are two kinds of Javadoc comments: class-level comments, and member-level comments. Class-level comments provide the description of the classes, and member-level comments describe the purposes of the members. Both types of comments start with /** and end with */. For example, this is a Javadoc comment:

```
/** This is a Javadoc comment */
```

4.1.2.1. Class-level Comments

Class-level comments provide a description of the class, and they are placed right above the code that declares the class. Class-level comments generally contain author tags, and a description of the class. An example class-level comment is below:

```
/**
 * @author OOAD_Instructor
 *
 * The Inventory class contains the amounts of all the
```

```
 * inventory in the CoffeeMaker system.  The types of
 * inventory in the system include coffee, milk, sugar
 * and chocolate.
 */
public class Inventory {

   //Inventory code here

}
```

4.1.2.2. Member-level Comments

Member-level comments describe the fields, methods, and constructors. Method and constructor comments may contain tags that describe the parameters of the method. Method comments may also contain return tags. An example of these member-level comments are below:

```
/**
 * @author OOAD_Instructor
 *
 * The Inventory class contains the amounts of all the
 * inventory in the CoffeeMaker system. The types of
 * inventory in the system include coffee, milk, sugar
 * and chocolate.
 */
public class Inventory {

   /**
    * Inventory for coffee
    */
   private int amtCoffee;

   /**
    * Default constructor for Inventory
    * Sets all ingredients to 15 units
    */
   public Inventory() {
      this.coffee = 15;
   }

   /**
    * Returns the units of coffee in the Inventory
    *
    * @return int
    */
   public int getAmtCoffee() {
      return coffee;
   }

   /**
    * Sets the units of coffee in the Inventory
    *
    * @param int new units coffee
    */
   public void setAmtCoffee(int newCoffee) {
```

```
        this.coffee = newCoffee;
    }
}
```

4.1.3. Tags

Tags are keywords recognized by Javadoc which define the type of information that follows. Tags come after the description (separated by a new line). Here are some common pre-defined tags:

- ***@author [author name]*** - identifies author(s) of a class or interface.
- ***@version [version]*** - version info of a class or interface.
- ***@param [argument name] [argument description]*** - describes an argument of method or constructor.
- ***@return [description of return]*** - describes data returned by method (unnecessary for constructors and void methods).
- ***@exception [exception thrown] [exception description]*** - describes exception thrown by method.
- ***@throws [exception thrown] [exception description]*** - same as ***@exception***.

4.1.4. Javadoc Templates in Eclipse

Eclipse generates Javadoc everytime you create a class, method, or field, using templates built into the IDE. These templates can be modified to create Javadoc in the format that you want. Or you can turn off the Javadoc generation property.

The Javadoc templates are found by selecting **Window > Preferences**. Under the tree on the left of the Preferences dialog, open up **Java > Code Style > Code Templates**.

There is a check box at the bottom of the Code Templates page that says **Automatically add comments for new methods and types.** If you uncheck this box, Javadoc comments will not be generated.

4.1.5. Generate HTML Document

- Right click on the project and select **Export...**. Select **Javadoc**, and click **Next**.
- Make sure the **Javadoc command** refers to the Javadoc command line tool. This tool is provided with JDK. The name is usually *javadoc.exe*. In the lab, you can find *javadoc.exe* in *C:\jdk1.6.1\bin* (that is, the Javadoc command is *C:\jdk1.6.1\bin\ javadoc.exe* .)
- Select the types that you want to create Javadoc for.
- Make sure that the **Use Standard Doclet** radio button is chosen, and **Browse** for a destination folder.
- Click **Next** for more options if you wish; otherwise click **Finish.**

2. JUnit Testing

JUnit is a simple open source Java testing framework used to write and run repeatable automated tests. Unit testing is the process of examining a small "unit" of software (usually a single class) to verify that it meets its expectations or specification.

JUnit features include:

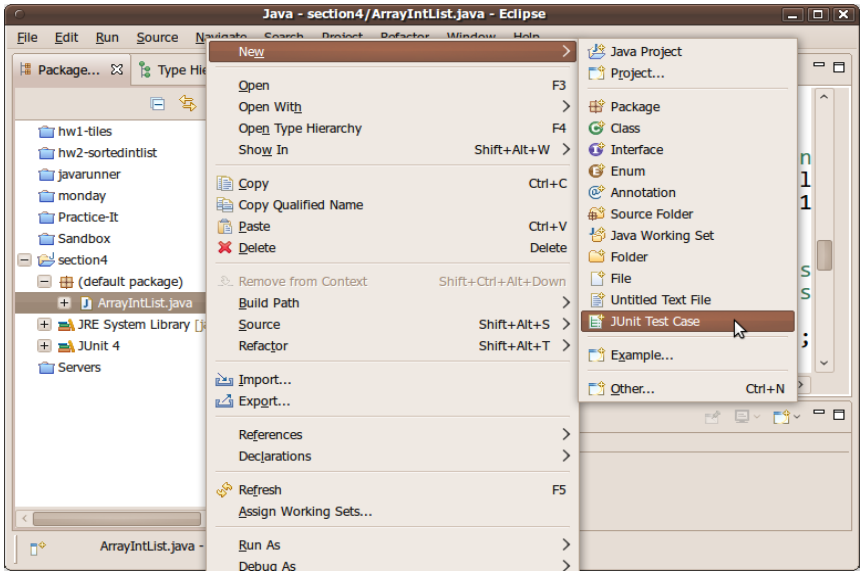
- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test suites for easily organizing and running tests
- Graphical and textual test runners

A unit test targets some other "class under test;" for example, the class `ArrayIntListTest` might be targeting the `ArrayIntList` as its class under test. A unit test generally consists of various testing methods that each interact with the class under test in some specific way to make sure it works as expected. JUnit isn't part of the standard Java class libraries, but it does come included with Eclipse. Or if you aren't using Eclipse, JUnit can be downloaded for free from the JUnit web site at http://junit.org.

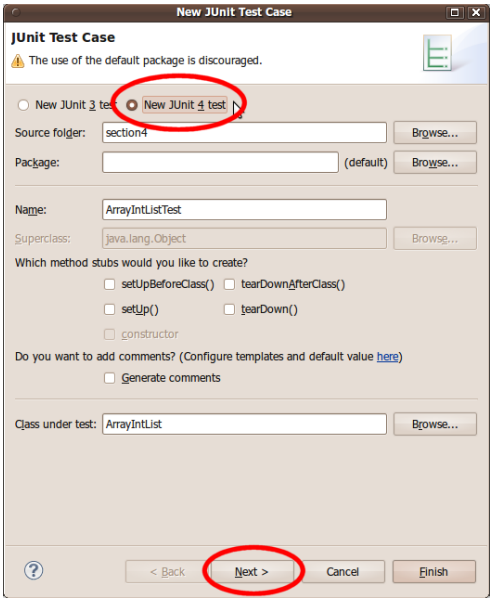
2.1. Creating a JUnit Test Case in Eclipse

A test case is a class which holds a number of test methods. For example if you want to test some methods of a class *Book* you create a class *BookTest* which extends the JUnit *TestCase* class and place your test methods in there.

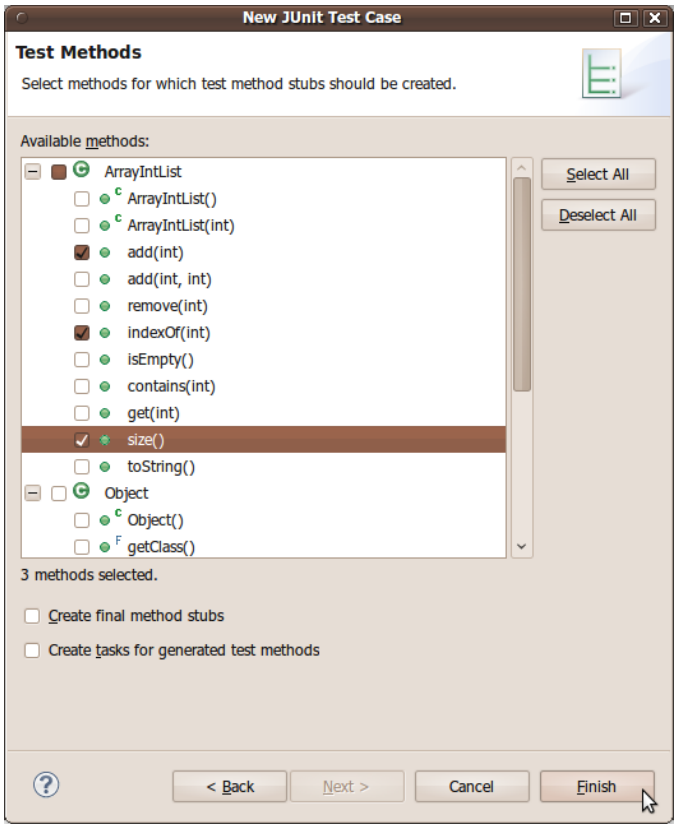
To use JUnit you must create a separate .java file in your project that will test one of your existing classes. In the Package Explorer area on the left side of the Eclipse window, right-click the class you want to test and click New → JUnit Test Case.



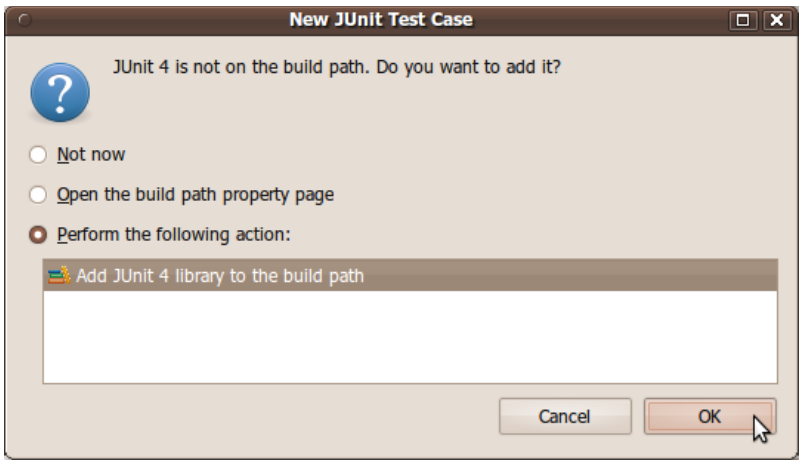
A dialog box will pop up to help you create your test case. Make sure that the option at the top is set to use JUnit 4, not JUnit 3. Click Next.



You will see a set of checkboxes to indicate which methods you want to test. Eclipse will help you by creating "stub" test methods that you can fill in. (You can always add more later manually.) Choose the methods to test and click Finish.



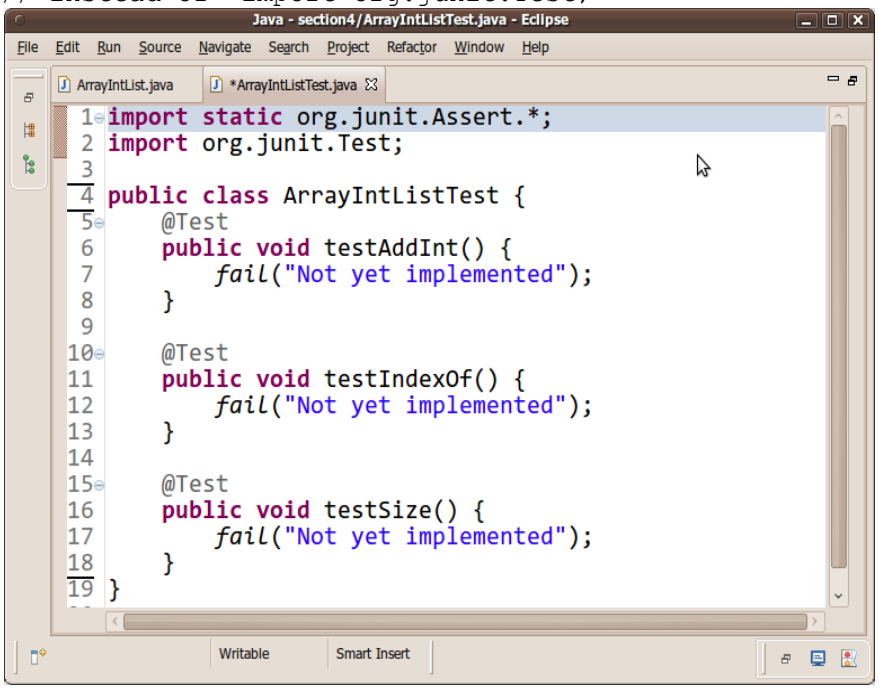
At this point Eclipse will ask whether you want it to automatically attach the JUnit library to your project. Yes, you do. Select "Perform the following action: Add JUnit 4 library to the build path" and press OK.



(If you forget to do add JUnit to your project, you can later add it to your project manually by clicking the top Project menu, then Properties, then Java Build Path, then click Add Library..., and choose JUnit 4 from the list.)

When you're done, you should have a nice new JUnit test case file. I suggest that you change the second `import` statement at the top to say the following:

```
import org.junit.*;    // instead of  import org.junit.Test;
```



2.2. Writing Tests

Each unit test method in your JUnit test case file should test a particular small aspect of the behavior of the "class under test." For example, an ArrayIntListTest might have one testing method to see whether elements can be

added to the list and then retrieved. Another test might check to make sure that the list's size is correct after various manipulations. And so on. Each testing method should be short and should test only one specific aspect of the class under test.

JUnit testing methods utilize *assertions*, which are statements that check whether a given condition is true or false. If the condition is false, the test method fails. If all assertions' conditions in the test method are true, the test method passes. You use assertions to state things that you expect to always be true, such as `assertEquals(3, list.size());` if you expect the array list to contain exactly 3 elements at that point in the code. JUnit provides the following assertion methods:

method name / parameters	description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented.
assertTrue(true) / assertTrue(false)	Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertsEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same. tolerance is the value that the 2 numbers can be off by. So it will assert to true as long as `Math.abs(expected - actual) < tolerance`
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.

Here is a quick example that uses several of these assertion methods.

```
ArrayIntList list = new ArrayIntList();
list.add(42);
list.add(-3);
list.add(17);
list.add(99);

assertEquals(4, list.size());
assertEquals(17, list.get(2));
assertTrue(list.contains(-3));
assertFalse(list.isEmpty());
```

Notice that when using comparisons like `assertEquals`, expected values are written as the left (first) argument, and the actual calls to the list should be written on the right (second argument). This is so that if a test fails, JUnit will give the right error message such as, "expected 4 but found 0".

A well-written test method chooses the various assertion method that is most appropriate for each check. Using the most appropriate assertion method helps JUnit provide better error messages when a test case fails. The previous assertions could have been written in the following way, but it would be poorer style:

```
// This code uses bad style.
assertTrue(list.size() == 4);         // bad; use assertEquals
assertTrue(list.get(2) == 17);        // bad; use assertEquals
if (!list.contains(-3)) {
    fail();                           // bad; use assertTrue
}
assertTrue(!list.isEmpty());          // bad; use assertFalse and delete the !
```

Good test methods are short and test only one specific aspect of the class under test. The above example code is in that sense a poor example; one should not test `size`, `get`, `contains`, and `isEmpty` all in one method. A better (incomplete) set of tests might be more like the following:

```
@Test
public void testAddAndGet1() {
    ArrayIntList list = new ArrayIntList();
    list.add(42);
    list.add(-3);
    list.add(17);
    list.add(99);
    assertEquals(42, list.get(0));
    assertEquals(-3, list.get(1));
    assertEquals(17, list.get(2));
    assertEquals(99, list.get(3));

    assertEquals("second attempt", 42, list.get(0));   // make sure I can get them a second
time
    assertEquals("second attempt", 99, list.get(3));
}

@Test
public void testSize1() {
    ArrayIntList list = new ArrayIntList();
    assertEquals(0, list.size());
    list.add(42);
    assertEquals(1, list.size());
    list.add(-3);
    assertEquals(2, list.size());
    list.add(17);
    assertEquals(3, list.size());
    list.add(99);
    assertEquals(4, list.size());
    assertEquals("second attempt", 4, list.size());   // make sure I can get it a second
time
}

@Test
public void testIsEmpty1() {
    ArrayIntList list = new ArrayIntList();
    assertTrue(list.isEmpty());
    list.add(42);
    assertFalse("should have one element", list.isEmpty());
```

```
    list.add(-3);
    assertFalse("should have two elements", list.isEmpty());
}

@Test
public void testIsEmpty2() {
    ArrayIntList list = new ArrayIntList();
    list.add(42);
    list.add(-3);
    assertFalse("should have two elements", list.isEmpty());
    list.remove(1);
    list.remove(0);
    assertTrue("after removing all elements", list.isEmpty());
    list.add(42);
    assertFalse("should have one element", list.isEmpty());
}

...
```

You might think that writing unit tests is not useful. After all, we can just look at the code of methods like `add` or `isEmpty` to see whether they work. But it's easy to have bugs, and JUnit will catch them better than our own eyes.

Even if we already know that the code works, unit testing can still prove useful. Sometimes we introduce a bug when adding new features or changing existing code; something that used to work is now broken. This is called a *regression*. If we have JUnit tests over the old code, we can make sure that they still pass and avoid costly regressions.

2.3. Annotations

The following table gives an overview of the available annotations in JUnit 4.x.

Annotation	Description
@Test public void method()	The annotation @Test identifies that a method is a test method.
@Before public void method()	Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After public void method()	Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).
@BeforeClass public void method()	Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database.
@AfterClass public void method()	Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database.
@Ignore	Will ignore the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.

@Test(expected=IllegalArgumentException.class)	Fails, if the method does not throw the named exception.
@Test(timeout=100)	Fails, if the method takes longer than 100 milliseconds.

2. Using a test fixture

A test fixture is useful if you have two or more tests for a common set of objects. Using a test fixture avoids duplicating the test code necessary to initialize and cleanup those common objects for each test.

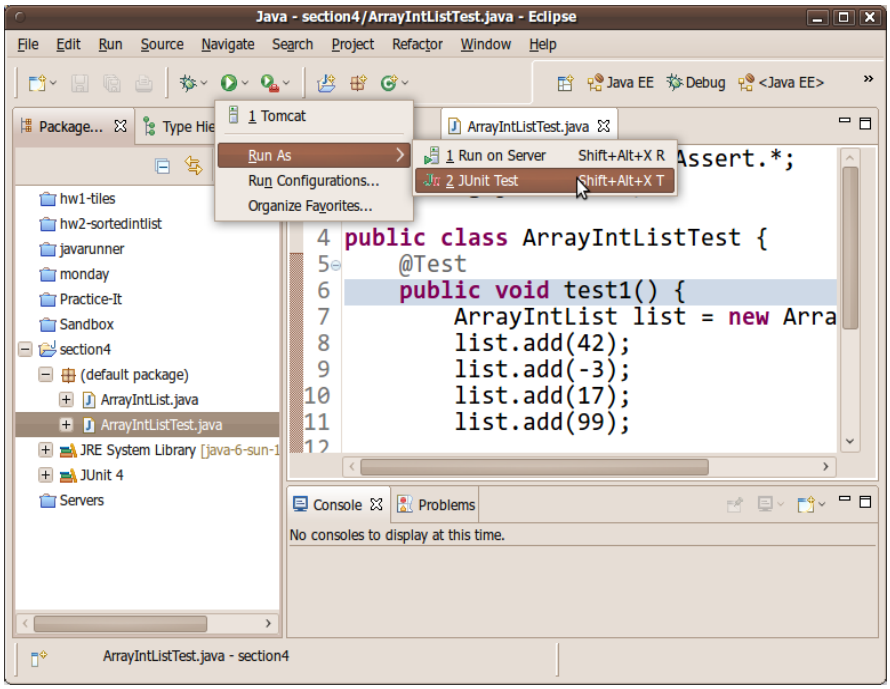
To create a test fixture, define a *setUp()* method that initializes common object and a *tearDown()* method to cleanup those objects. The JUnit framework automatically invokes the *setUp()* method before a each test is run and the *tearDown()* method after each test is run.

The following test uses a test fixture:

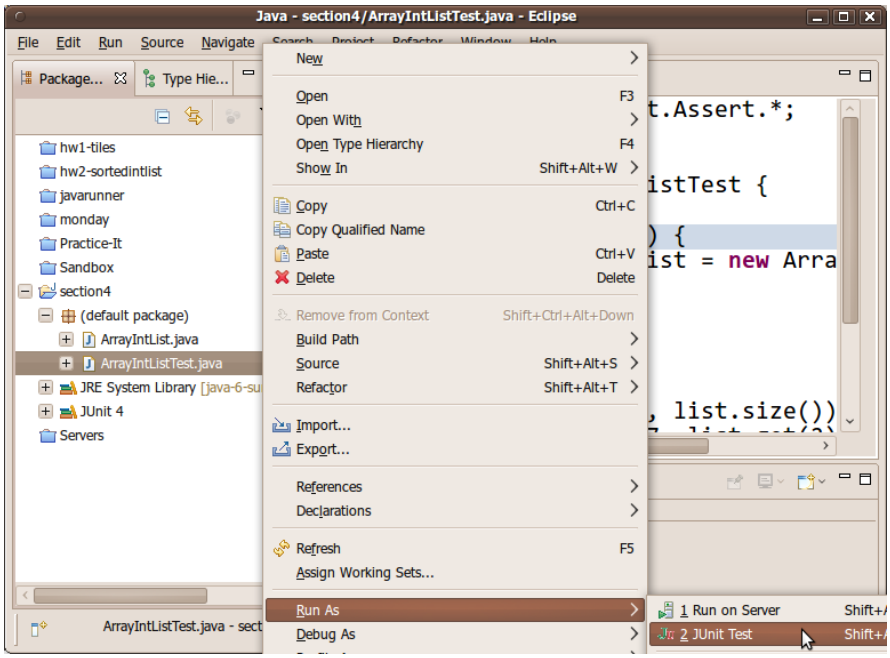
```
public class BookTest2 extends TestCase {

    private Collection collection;

    protected void setUp() {
       collection = new ArrayList();
    }

    protected void tearDown() {
         collection.clear();
    }

    public void testEmptyCollection(){
         assertTrue(collection.isEmpty());
    }
}
```

2.4.Running Your Test Case

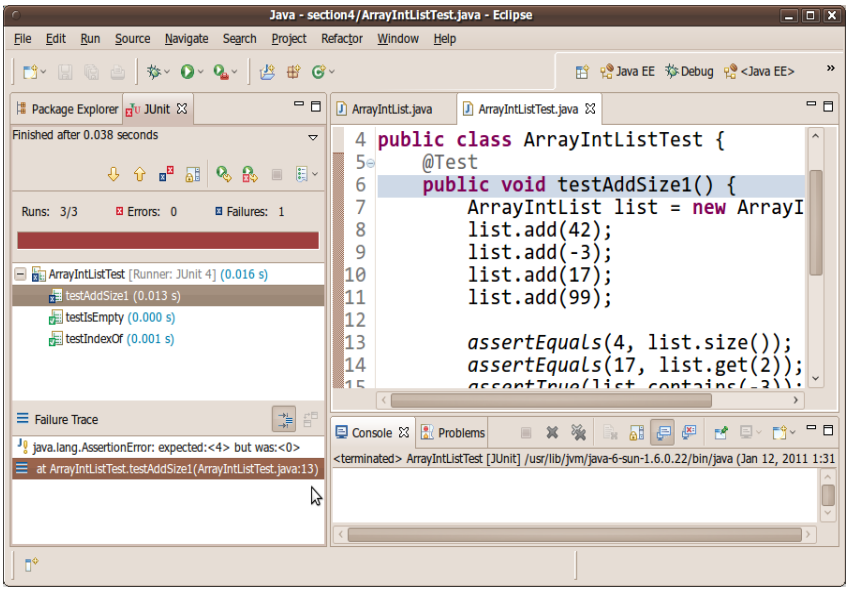
Once you have written one or two test methods, run your JUnit test case. There are two ways to do this. One way is to click the Run button in the top toolbar (it looks like a green "Play" symbol). A menu will drop down; choose to run the class as a JUnit Test.



The other way is to right-click your JUnit test case class and choose Run As → JUnit Test.



A new pane will appear showing the test results for each method. You should see a green bar if all of the tests passed, or a red bar if any of the tests failed. If any tests fail, you can view the details about the failure by clicking on the failed test's name/icon and looking at the details in the pane below.



Most people think that getting a red failure bar is bad. It's not! It is good; it means that you have found a potential bug to be fixed. Finding and fixing bugs is a good thing. Making a red bar become a green bar (by fixing the code and then re-running the test program) can be very rewarding.

2.5.Test Suite

If you have two tests and you'll run them together you could run the tests one at a time yourself, but you would quickly grow tired of that. Instead, JUnit provides an object *TestSuite* which runs any number of test cases together. The suite method is like a main method that is specialized to run tests.

In Junit 4, both **@RunWith** and **@Suite** annotation are used to run the suite test.

The below example means both unit test **JunitTest1** and **JunitTest2** will run together after JunitTest5 is executed.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
        JunitTest1.class,
        JunitTest2.class
})
public class JunitTest5 { }
```