JUnit 4 Testing

Frank Walsh

Annotations

Intro to Annotations

- Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate.
- Annotations have a number of uses, among them:
  - Information for the compiler — Annotations can be used by the compiler to detect errors or suppress warnings.
  - Compiler-time and deployment-time processing — Software tools can process annotation information to generate code, XML files, and so forth.
  - Runtime processing — Some annotations are available to be examined at runtime.
- Annotations can be applied to a program's declarations of classes, fields, methods, and other program elements

Annotations in Java

- The annotation appears first, often (by convention) on its own line, and may include elements with named or unnamed values.
- The annotation must itself be already defined and explicitly imported if necessary:
- Annotations are defined using a special syntax:

```
1 import documentation.Author;
2
3 @Author(name = "fxwalsh", date = "27/9/2017")
4 public class MyClass {
5
6 }
7
```

```
package documentation;

public @interface Author {
    String name();
    String date();
}
```

Built in Annotations

- There are three annotation types that are predefined by the language specification itself:
  - **@Deprecated**— indicates that the marked element is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation.
  - **@Override -** informs the compiler that the element is meant to override an element declared in a superclass. It not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked with @Override fails to correctly override a method in one of its superclasses, the compiler generates an error.
  - **@SuppressWarnings** - tells the compiler to suppress specific warnings that it would otherwise generate

JUnit 4 and Annotations

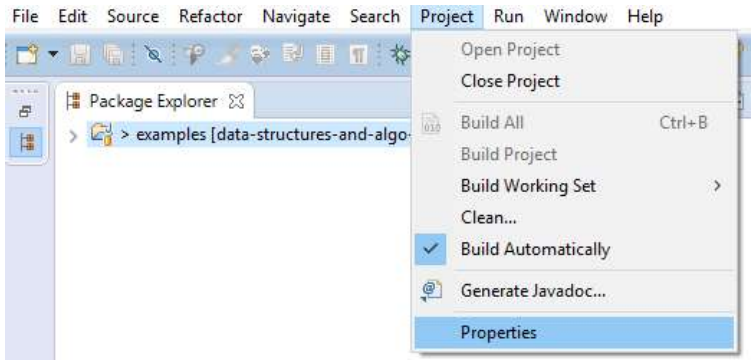
- JUnit uses annotations
  - @Before - run before each test
  - @After - run after each test
  - @Test - the test itself
- No need to extend TestCase

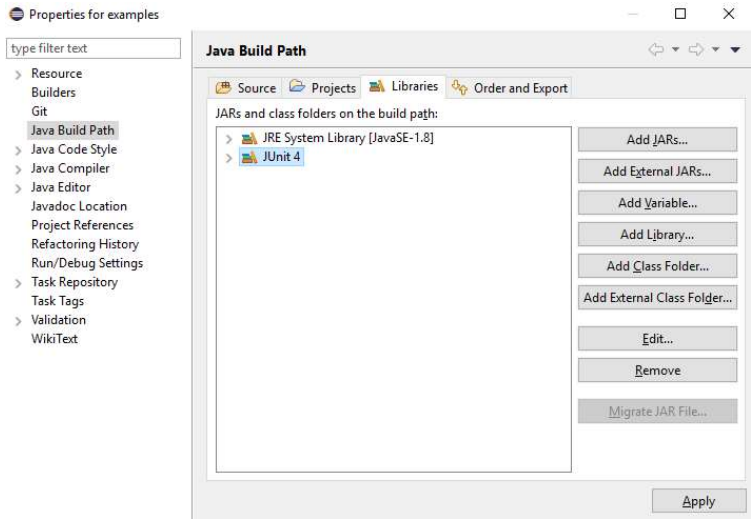
```
1  import static org.junit.Assert.*;
2
3  import org.junit.After;
4  import org.junit.Before;
5  import org.junit.Test;
6
7  public class MyTestCase {
8
9      @Before
10     public void setUp()
11     {
12         //stuff to do BEFORE EACH
13     }
14
15     @After
16     public void tearDown()
17     {
18         //stuff to do AFTER EACH t
19     }
20
21     @Test
22     public void test() {
23         fail("Not yet implemented"
24     }
25 }
```

First Test

Adding Junit to build path

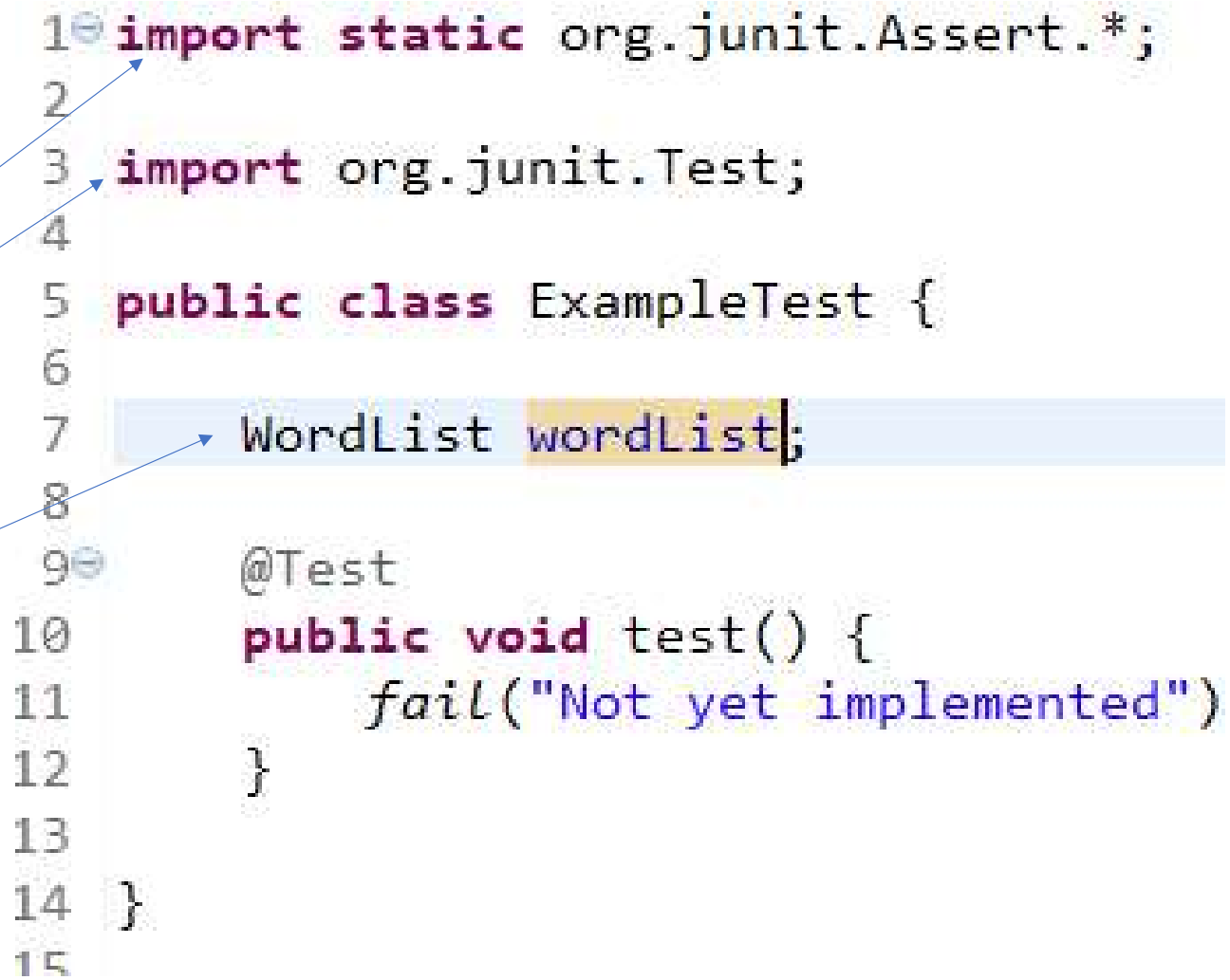
- Make sure you have the JUnit 4 library on your Java project's path. In Eclipse. Select the project in the project view and select Project -> properties. This will open the properties window
- Select *Java Build Path* and select the libraries tab.
- Select *Add Library…* button and add JUnit 4 library to the path.





Create New Test Case

- A Junit 4 test need to import relevant classes (However, Eclipse will do this automatically if you create the test *(File->new->Junit Test Case)*
- Typically include a declaration of the class being tested



```
1  import static org.junit.Assert.*;
2
3  import org.junit.Test;
4
5  public class ExampleTest {
6
7      WordList wordList;
8
9      @Test
10     public void test() {
11         fail("Not yet implemented");
12     }
13
14 }
15
```

JUnit 4 – Just once

- You can declare one method to be executed just once, when the class is first loaded
  - This is for time consuming setup, such as connecting to a data source.
- You can also declare *one* method to be executed *just once* after all the tests have been completed.

```
@BeforeClass
public static void setUpClass(){
    // one-time initialization code
}

@AfterClass
public static void tearDownClass(){

}
```

Junit – before each test

- You can define one or more methods to be executed before each test
- Typically such methods initialize values, so that each test starts with a fresh set

```
@Before
public void setUp(){
    wordList=new WordList("https://www.myurl.com/my_word_list");
}

@After
public void tearDown(){
    //tear down stuff...
}
```

Junit - a test (finally)

- A test method is annotated with @Test, takes no parameters, and returns no result
- Here's a failing test autogenerated by Eclipse.

```
    @Test
    public void test() {
        fail("Not yet implemented");
    }
}
```

Assertions

- Tests use **Assertions to** check if code is behaving as you expect.
- An assertion is a simple method call that verifies that something is true.

```
@Test
public void testSimpleStuff() {
      int a=2;
      assertTrue(a==2);
}
```

More Assertions

- Junit provides lots of assertion methods. Try to be as “expressive” as possible:
- Here’s the same test again using assertEquals(…):

```
@Test
public void testSimpleStuff() {
      int a=2;
      assertEquals(a,2);
}
```

Planning Tests

- Method to test: A static method designed to find the largest number in a list of numbers.
- The following tests would seem to make sense:

  [7, 8, 9] -> 9

  [8, 9, 7] -> 9

  [9, 7, 8] -> 9

  (supplied test data ->expected result)

```
public static int largest (int[] list)
{
...
}
```

More Test Data + First Implementation

- Already have this data:

  [7, 8, 9] -> 9

  [8, 9, 7] -> 9

  [9, 7, 8] -> 9
- What about this set:

  [7, 9, 8, 9] -> 9

  [1] -> 1

  [-9, -8, -7] -> -7

```
public static int largest (int[] list)
{
  int index, max = Integer.MAX_VALUE;

  for (index = 0; index < list.length - 1; index++)
  {
    if (list[index] > max)
    {
      max = list[index];
    }
  }
  return max;
}
```

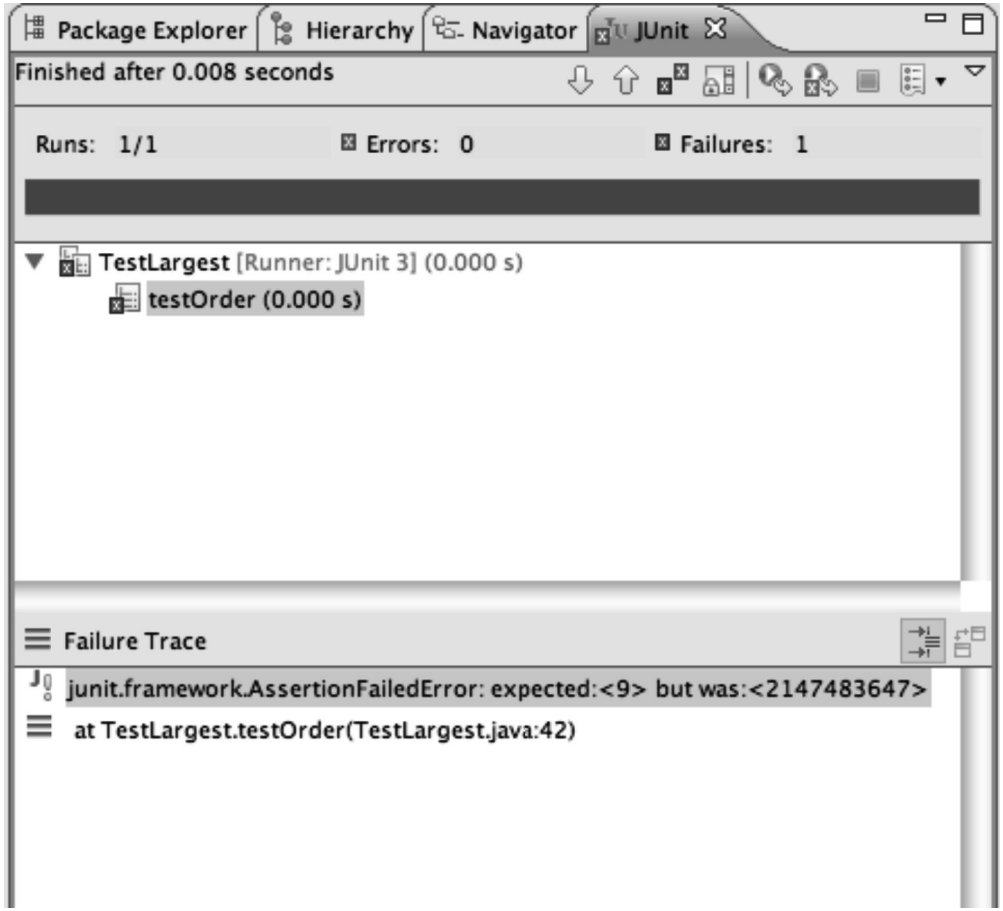
Writing the Test

- This is a TestCase called TestLargest.
- It has one Unit Test - to verify the behaviour of the largest method.

```
import junit.framework.TestCase;

public class TestLargest extends TestCase
{
  public TestLargest (String name)
  {
    super(name);
  }

  public void testOrder ()
  {
    int[] arr = new int[3];
    arr[0] = 8;
    arr[1] = 9;
    arr[2] = 7;
    assertEquals(9, Largest.largest(arr));
  }
}
```

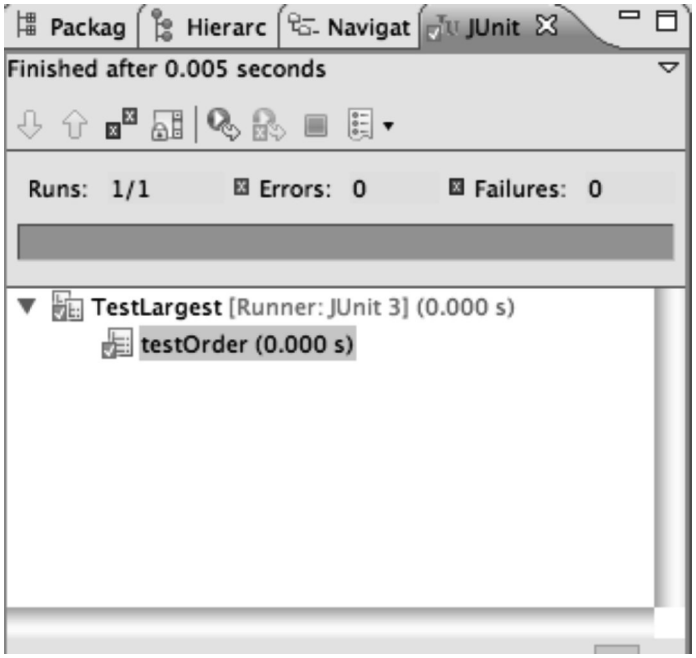
Running the Test

- Why did it return such a huge number instead of our 9
- Where could that very large number have come from?



Bug

- First line should initialize max to zero, not MAX_VALUE.



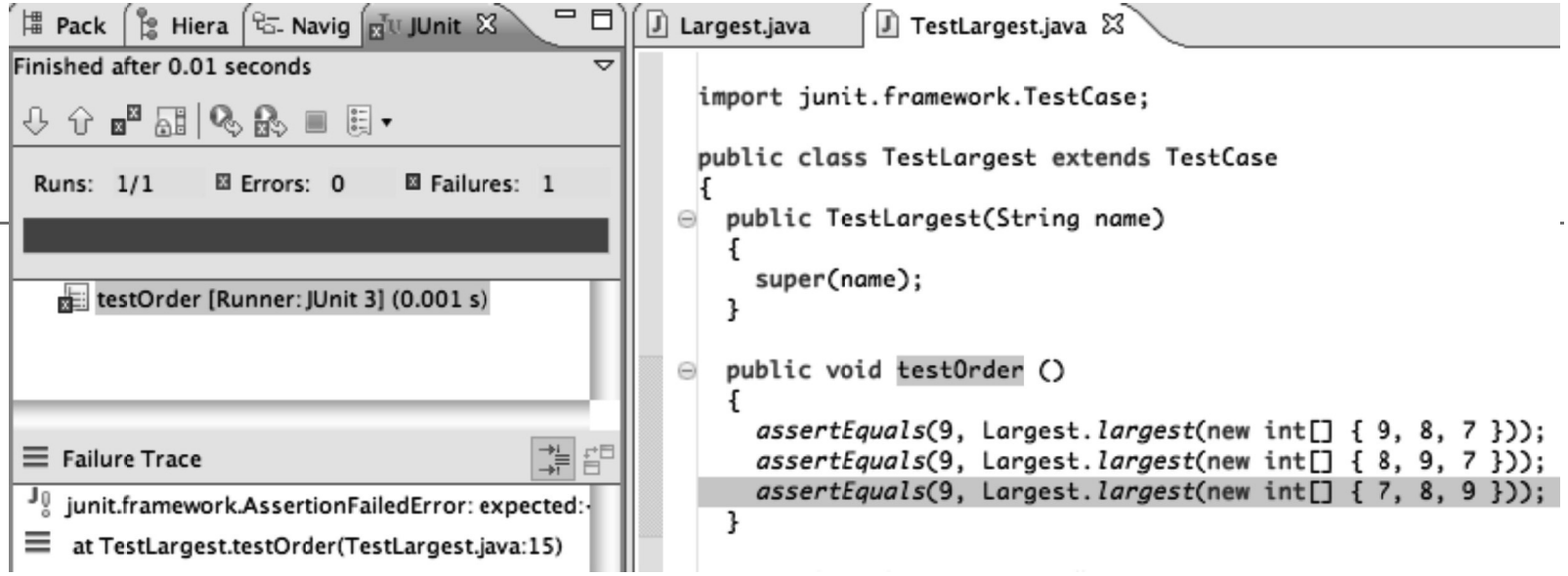

```
public static int largest (int[] list)
{
  //int index, max = Integer.MAX_VALUE;
  int index, max = 0;

  for (index = 0; index < list.length - 1; index++)
  {
    if (list[index] > max)
    {
      max = list[index];
    }
  }
  return max;
}
```

Further Tests

- What happens when the largest number appears in different places in the list - first or last, and somewhere in the middle?
  - Bugs most often show up at the “edges”
  - In this case, edges occur when when the largest number is at the start or end of the array that we pass in
- Aggregate into a single unit test:

```
public void testOrder ()
{
  assertEquals(9, Largest.largest(new int[] { 9, 8, 7 }));
  assertEquals(9, Largest.largest(new int[] { 8, 9, 7 }));
  assertEquals(9, Largest.largest(new int[] { 7, 8, 9 }));
}
```

Failure + Fix

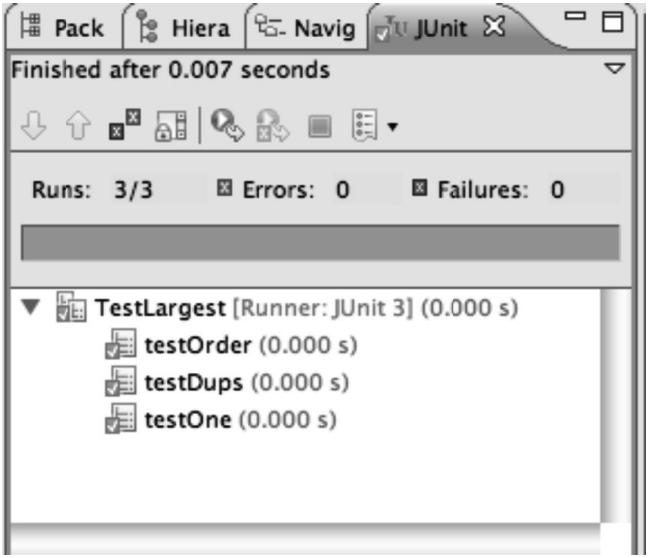



```
public static int largest (int[] list)
{
  int index, max = 0;
  //for (index = 0; index < list.length - 1; index++)
  for (index = 0; index < list.length; index++)
  {
    if (list[index] > max)
    {
      max = list[index];
    }
  }
  return max;
}
```

Further Boundary Conditions

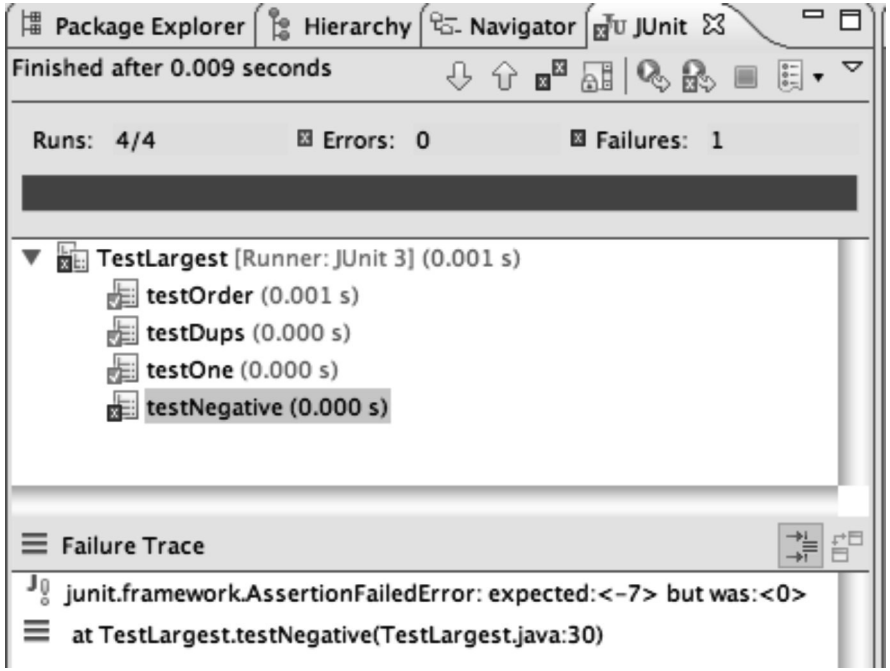
```
public void testDups ()
{
  assertEquals(9, Largest.largest(new int[] { 9, 7, 9, 8 }));
}

public void testOne ()
{
  assertEquals(1, Largest.largest(new int[] { 1 }));
}
```

- Now exercising multiple tests



Failure on testNegative

```
public void testNegative ()
{
  int[] negList = new int[] { -9, -8, -7 };
  assertEquals(-7, Largest.largest(negList));
}
```



fix testNegative

- Choosing 0 to initialize max was a bad idea;
- Should have been MIN VALUE, so as to be less than all negative numbers as well

```
public static int largest (int[] list)
{
  //int index, max = 0;
  int index, max = Integer.MIN_VALUE;

  for (index = 0; index < list.length; index++)
  {
    if (list[index] > max)
    {
      max = list[index];
    }
  }
  return max;
}
```

Expected Errors?

- If the array is empty, this is considered an error, and an exception should be thrown

```
public static int largest (int[] list)
{
  int index, max = Integer.MIN_VALUE;

  if (list.length == 0)
  {
    throw new RuntimeException("Empty list");
  }
  for (index = 0; index < list.length; index++)
  {
    if (list[index] > max)
    {
      max = list[index];
    }
  }
  return max;
}
```

```
public void testEmpty ()
{
  try
  {
    Largest.largest(new int[] {});
    fail("Should have thrown an exception");
  }
  catch (RuntimeException e)
  {
    assertTrue(true);
  }
}
```