JUNIT

Parameterised Test

Agenda (Contd.).

Parameterized test

Objectives

At the end of this module, you will be able to:

- Carry on Parameterized tests

Parameterized test

Parameterised Tests

- New feature added in JUnit 4
- Used to test a method with varying number of Parameters
- Steps for testing a code with multiple parameters
  - The testing class should be annotated with @RunWith(Parameterized.class)
  - The class should have these 3 entities
    - A single constructor that stores the test data
      - Is expected to store each data set in the class fields
    - A static method that generates and returns test data
      - This should be annotated with @Parameters
      - It should return a Collection of Arrays
      - Each array represent the data to be used in a particular test run
      - Number of elements in an array should correspond to the number of elements in the constructor
      - Because each array element will be passed to the constructor for every run
    - A test method

Parameterised Tests

- **Structure of a parameterized test class**

•The method that generates test data must be annotated with @Parameters, and it must return a Collection of Arrays. Each array represents the data to be used in a particular test run. The number of elements in each array must correspond to the number of parameters in the class's constructor, because each array element will be passed to the constructor, one at a time as the class is instantiated over and over.

•The constructor is simply expected to store each data set in the class's fields, where they can be accessed by the test methods. Note that only a single constructor may be provided. This means that each array provided by the data-generating method must be the same size, and you might have to pad your data sets with nulls if you don't always need a particular value.

Example

```
package junit.first;
import junit.first.Stringmanip.*;
import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;

  @RunWith(Parameterized.class)
  public class StringmanipTest2
   {
     // Fields
    private String datum;
    private String expected;
```

Example

```
 /*
       Constructor.
        The JUnit test runner will instantiate this class once for every element in the Collection returned by the method annotated with
         @Parameters.
       */
      public StringmanipTest2(String datum, String expected)
      {
         this.datum = datum;
          this.expected = expected;
       }

       /*
         Test data generator.
         This method is called the the JUnit parameterized test runner and returns a Collection of Arrays.  For each Array in the Collection, each array element corresponds to a parameter in the constructor.
       */
```

Example

```
@Parameters
    public static Collection<Object[]> generateData()
   {
      // In this example, the parameter generator returns a List of
      // arrays.  Each array has two elements: { datum, expected }.
    // These data are hard-coded into the class, but they could be
    // generated or loaded in any way you like.
    Object[][] data = new  Object[][]{
              { “Smita", “SMITA" },
              { “smita", “SMITA" },
              { “SMITA", “SMITA" }
          };
     return Arrays.asList(data);
   }
```

Example

```
 /*
This test method is run once for each element in the Collection returned by the test data generator -- that is, every time this class is instantiated. Each time this class is instantiated, it will have a different data set, which is available to the test method through the
         instance's fields.
        */
       @Test
       public void testUpperCase()
       {
          Stringmanip s = new Stringmanip(this.datum);
         String actualResult = s.upperCase();
         assertEquals(actualResult, this.expected);
      }
    }
```

Handling an Exception

- Two cases are there:
  - Case 1 :We expect a normal behavior and then no exceptions
  - Case 2: We expect an anomalous behavior and then an exception

Case 1:

```
@Test
public void testDiv()
{
try
{
c.div(10,2);
assertTrue(true); //OK
}catch(ArithmeticException
expected)
{
fail("Method should not
fail");
}}
```

Case 2:

```
@Test
public void testDiv()
{
try
{
c.div(10,0);
fail("Method should fail");
}catch(ArithmeticException
expected)
{
assertTrue(true);
} }
```

```
public void div(int a,int b)throws ArithmeticException
{
int c=0;

c=a/b;
System.out.println(c);}
```

What is Wrong with this code?

```
Boolean validate(int primeNumber)
{
for(int i=2;i< Math.sqrt(primeNumber);i++){
   if (primeNumber % i == 0) {
   System.out.println(primeNumber+"is Composite");
   return false;
   }
}
   System.out.println(primeNumber+" is Prime");
   return true;
}
```

What is Wrong with this code? --contd.

- **Test the code in previous page using JUNIT.**

Check with the following set of inputs:

  - **Input Set1**: 3,7,11,15,17,19, 9999
    - **Whether all prime numbers are detected correctly?**
  - **Input Set2**: 2, 4, 20, 48, 71, 881, 991
    - Which are composite numbers are detected correctly?
    - **As per given code, 4 is prime or composite?**
      - *If 4 is printed as prime, then, make changes in the code.*

What is Wrong with this code? --contd.

- As per the given code, for Input set2 the result is:
- 2 is Prime
- **4 is Prime --→ Do you agree with this? Is not, what is problem in the code?**
- 20 is Composite
- 48 is Composite
- 71 is Prime
- 881 is Prime
- 991 is Prime

Assignment

Summary

- Parameterized Tests

Thank You