SOEN 387
WEB-BASED ENTERPRISE APPLICATIONS DESIGN

TUTORIAL – 5

JUnit

Agenda

- Software Unit Testing
- Unit testing framework: components
- Junit Testing Framework: Junit
- JUnit: setting up the testing context
- JUnit 4: assertions
- JUnit 4: test class annotations
- JUnit 4: Test class
- JUnit : Exercise
- Repository
  - Introduction
  - Book.Java
  - Interface: BookRepository.Java
  - BookRepositoryImpl.Java
  - Singleton and thread safe approach
  - ServletContext()
  - book-list.jsp
- Junit example tests for book.java
- In Class Exercise

Software Unit Testing

“testing can reveal only the presence of faults, never their absence.” [Edsger Dijkstra]

Definition: A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality applied to one of the units of the code being tested. Usually a unit test exercises some particular method in a particular context.

Example: add a large value to a sorted list, then confirm that this value appears at the end of the list.

The goal of unit testing is to isolate important parts (i.e. units) of the program and show that the individual parts are free of certain faults.

Unit testing framework: components

- **Tested Class** – the class that is being tested.
- **Tested Method** – the method that is tested.
- **Test Case** – the testing of a class's method against some specified conditions.
- **Test Case Class** – a class performing some test cases.
- **Test Case Method** – a Test Case Class's method implementing a test case.
- **Test Suite** – a collection of test cases that can be tested in a single batch.

Java unit testing framework: JUnit

- In Java, the standard unit testing framework is known as JUnit.
- Test Cases and Test Results are Java objects.
- Using JUnit you can easily and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts.

Thus, every test case has three phases:

1. Set up the context in which the method will be called.
2. Call the method.
3. Observe if the method call resulted in the correct behavior using one or more JUnit assertion method.

JUnit: setting up the testing context

1. If more than one test case share the same context, group them in a test class and use setUp()/tearDown() (JUnit 3) or @Before/@After (JUnit 4) to set the context to the right values before each test case is run.

1. JUnit 4 also enables you to set a number of static values and methods that are shared across all tests if that is desirable. These are managed using @BeforeClass and@AfterClass.

JUnit 4: assertions

The JUnit framework provides a complete set of assertion methods that can be used in different situations:

void assertEquals(boolean expected, boolean actual)

Check that two primitive values or objects are equal.

void assertTrue(boolean condition)

Check that a condition is true.

void assertFalse(boolean condition)

Check that a condition is false.

void assertNotNull(Object object)

Check that an object isn't null.

void assertNull(Object object)

Check that an object is null.

void assertSame(boolean condition)

Tests if two object references point to the same object.

void assertNotSame(boolean condition)

Tests if two object references do not point to the same object.

void assertArrayEquals(expectedArray, resultArray)

Tests whether two arrays are equal to each other.

JUnit 4: test class annotations

@Before

Several tests need similar objects created before they can run. Annotating a **public void** method with **@Before** causes that method to be run before each test method.

@After

If you allocate external resources in a **@Before** method you need to release them after the test runs. Annotating a **public void** method with **@After** causes that method to be run after each test method.

@BeforeClass

Annotating a **public static void** method with **@BeforeClass** causes it to be run once before any of the test methods in the class. As these are static methods, they can only work upon static members.

@AfterClass

Perform the following **public static void** method after all tests have finished. This can be used to perform clean-up activities. As these are static methods, they can only work upon static members.

@Test

Tells JUnit that the **public void** method to which it is attached can be run as a test case.

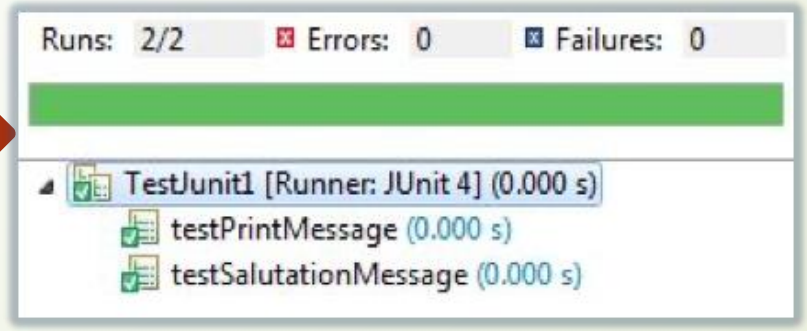
@Ignore

Ignore the test and that test will not be executed.

JUnit 4: Test class

Console output :

```
in before TestJunit1 class
in before TestJunit1 test case
Inside testPrintMessage()
Robert
in after TestJunit1 test case
in before TestJunit1 test case
Inside testSalutationMessage()
Hi!Robert
in after TestJunit1 test case
in after TestJunit1 class
```




```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
public class TestJunit1 {
  String message = "Robert";
  MessageUtil messageUtil = new MessageUtil(message);

  @BeforeClass public static void beforeClass() {
    System.out.println("in before TestJunit1 class");
  }
  @AfterClass public static void  afterClass() {
    System.out.println("in after TestJunit1 class");
  }
  @Before public void before() {
    System.out.println("in before TestJunit1 test case");
  }
  @After public void after() {
    System.out.println("in after TestJunit1 test case");
  }
  @Test public void testPrintMessage() {
    System.out.println("Inside testPrintMessage()");
    assertEquals(message, messageUtil.printMessage());
  }
  @Test public void testSalutationMessage() {
    System.out.println("Inside testSalutationMessage()");
    message = "Hi!" + "Robert";
    assertEquals(message,messageUtil.salutationMessage());
  }
}
```

JUnit : Exercise

1. Given a java file named Calculator.java (see below code):

```
public class Calculator {
  public int evaluate(String expression) {
    int sum = 0;
    for (String summand:
expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

Write a junit test named CalculatorTest.java to assert below statement true:

```
int sum = calculator.evaluate("1+2+3");
```

2. Given a java file named MyClass.java (see below code):

```
public class MyClass {
      public Object multiply(int i, int j) {
            return i*j;
      }

      public Object Divide(int i, int j) {
            if(j == 0 ) {
                  System.exit(0);
            }
            return i/j;
      }
}
```

Write a junit test named MyClassTester.java with two tests:
Test1. Assert true for multiplication of two numbers to be 0
Test2. Assert true for division of two numbers as per your choice

Introduction to Repository

Repository are a means of storing and updating dynamic data upon user request. It can be implemented using servlet and JSP. Following example shows:

BookRepositoryImpl.java skeleton code to add, update and populate information about book.

book.java file to store all the information about a book.

Reference: http://miageprojet2.unice.fr/GestionnaireLivreJEE

Repository/Book.java

```
package com.bookstore;
import java.math.BigDecimal;
import java.util.Date;
public class Book implements Cloneable {
    private String title ;
    private String description ;
    private BigDecimal price ;
    private Date pubDate ;
    private String id ;

    public Book (String id, String title,
String description, BigDecimal price, Date
pubDate) {
          this . id = id;
         this . title = title;
         this . description = description;
         this . price = price;
         this . pubDate = pubDate;
    }
    public Book () {
    }
    public String getTitle () {
          return title ;
    }
    public void setTitle (String title) {
          this . title = title;
    }
    public String getDescription () {
          return description ;
    }
    public void setDescription (String
description) {
          this . description = description;
    }
```

```
public BigDecimal getPrice () {
          return price ;
    }
public void setPrice (BigDecimal price) {
          this . price = price;
    }
public Date getPubDate () {
          return pubDate ;
    }
public void setPubDate (Date pubDate) {
          this . pubDate = pubDate;
    }
public String getId () {
          return id ;
    }
public void setId (String id) {
          this . id = id;
    }
  @Override
  public String toString () {return "Book [title =" + title + ",
  description =" + description+ ", price =" + price + ", pubDate =" +
  pubDate + ", id =" + id
  + "]" ;
    }

  }
```

Creating a template to represent a book

Interface: Repository/BookRepository.java

```
package com.bookstore;

import java.util.List;

public interface BookRepository {

    void add(book b);

    void update (book b);

    List <Book> listBooks ();

}
```

Here we have a model, let's make an interface

BookRepositoryImpl.java

```
public class BookRepositoryImpl {

        private AtomicInteger ai;
        private ConcurrentHashMap<Integer, Book> map;

        private BookRepositoryImpl() {
                ai = new AtomicInteger();
                map = new ConcurrentHashMap<Integer, Book>();

        }

        private static BookRepositoryImpl instance = null;

        public static synchronized <ServletContext> BookRepositoryImpl  getInstance(ServletContext ctx) {
                return instance == null && ctx != null ? (instance = new BookRepositoryImpl()) : instance;
        }

        public void add(Book b) throws Exception {
                b.setId(ai.addAndGet(1));
                map.put(b.getId(), b);
        }
```

ConcurrentHashMap class is thread-safe. At a time any number of threads are applicable for read operation without locking the ConcurrentHashMap object

BookRepositoryImpl.java

```
public List<Book> list() {
                ArrayList<Book> result = new ArrayList(map.values());
                result.sort((b1, b2) -> ("" + b1.getId()).compareTo("" + b2.getId()));
                return result;
        }

        public void update(Book b) throws Exception {
                if (map.containsKey(b.getId())) {
                        map.put(b.getId(), b);

                        return;
                }
                throw new Exception("key not found");
        }
}
```

The singleton design pattern is used to restrict the instantiation of a class and ensures that only one instance of the class exists in the JVM. In other words, a singleton class is a class that can have only one object (an instance of the class) at a time per JVM instance

BookListServlet

```
import com.bookstore.BookRepository;
import com.bookstore.BookRepositoryImp;

@WebServlet ( "/ book /" )
public class BookListServlet extends HttpServlet {

    protected void doGet (HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {

        BookRepository bookRepo = BookRepositoryImp.getInstance(getServletContext());
        request.setAttribute ( "books" , bookRepo.listBooks ());
        getServletContext().getRequestDispatcher(
                "/book-list.jsp" ).forward (request, response);
        }
}
```


The doGet method is called in the case of an HTTP GET request and has the query as the first parameter and the response as the second parameter.

getServletContext()returns "the execution context", ie an object that knows how to speak with the server running the application

book-list.jsp

```
<% @ Page language = " java " contentType = " text / html; charset = UTF-8 " pageEncoding = " UTF-8 "%>
<% @ taglib uri = " http://java.sun.com/jsp/ jstl / core " prefix =" c "%>
<! HTML DOCTYPE >

< html >
< head >
    < title > Book listing </ title >
</ head >
< body >


< table >
    < tr >
        < th > Title </ th >
        < th > Description </ th >
        < th > Price </ th >
        < th > Publication Date </ th >
    </ tr >

    < c : forEach var = "book" items = " $ { books } " >
        < tr >
            < td > $ { book.title } </ td >
            < td > $ { book.description } </ td >
            < td > $ { book.price } </ td >
            < td > $ { book.pubDate } </ td >
        </ tr >
    </ c : forEach >
</ table >

</ body >
  </ html >
```

In Class Exercise

1. Add a link at the top of the JSP display page, to add a book [Hint: `< A href = " $ { pageContext.request.contextPath } / book" > Add Book </ a >`]

1. Add a book and verify if book exists in the repository. Use Try-Catch block to handle any exception.

   [Hint: *Create a BookEditorServletthat will handle this type of URLs, set the name of the Servlet, the mapping, put it in the same package as the existing servlet*]

1. Add a Junit test for repository to very addBook()