LAB. TESTAREA UNITARA

ВВЕДЕНИЕ

Что такое модульное тестирование:

1. Тестирование отдельных функций системы.
2. Как правило, выполняется разработчиком модуля.
3. Может быть легко автоматизировано.
4. Закладывает основу для регрессионного тестирования приложения.

JUnit – библиотека, позволяющая проводить модульное тестирование Java-приложений.

Написание тестов стабилизирует код и позволяет сократить время отладки. Тестирование системы в целом (системное тестирование) не всегда позволяет обнаружить ошибки в отдельных компонентах. Исправление ошибок на ранних стадиях разработки менее затратно.

ПРИМЕР ПРОГРАММЫ

Пусть есть класс, реализующий несколько математических функций:

```
public class CustomMath {
public static int sum(int x, int y){
	return x+y; //возвращает результат слажения двух чисел
}
public static int division(int x, int y){
	if (y==0) {
		throw new IllegalArgumentException("devider is 0");
//бросается исключение
	}
	return(x/y); //возвращает результат деления
}
}
```

ЗАМЕЧАНИЕ

Иногда требуется снабжать программу модульными тестами.

Тесты неудобно хранить в самой программе:

1. Усложняет чтение кода.
2. Такие тесты сложно запускать.
3. Тесты не относятся к бизнес-логике приложения и должны быть исключены из конечного продукта.

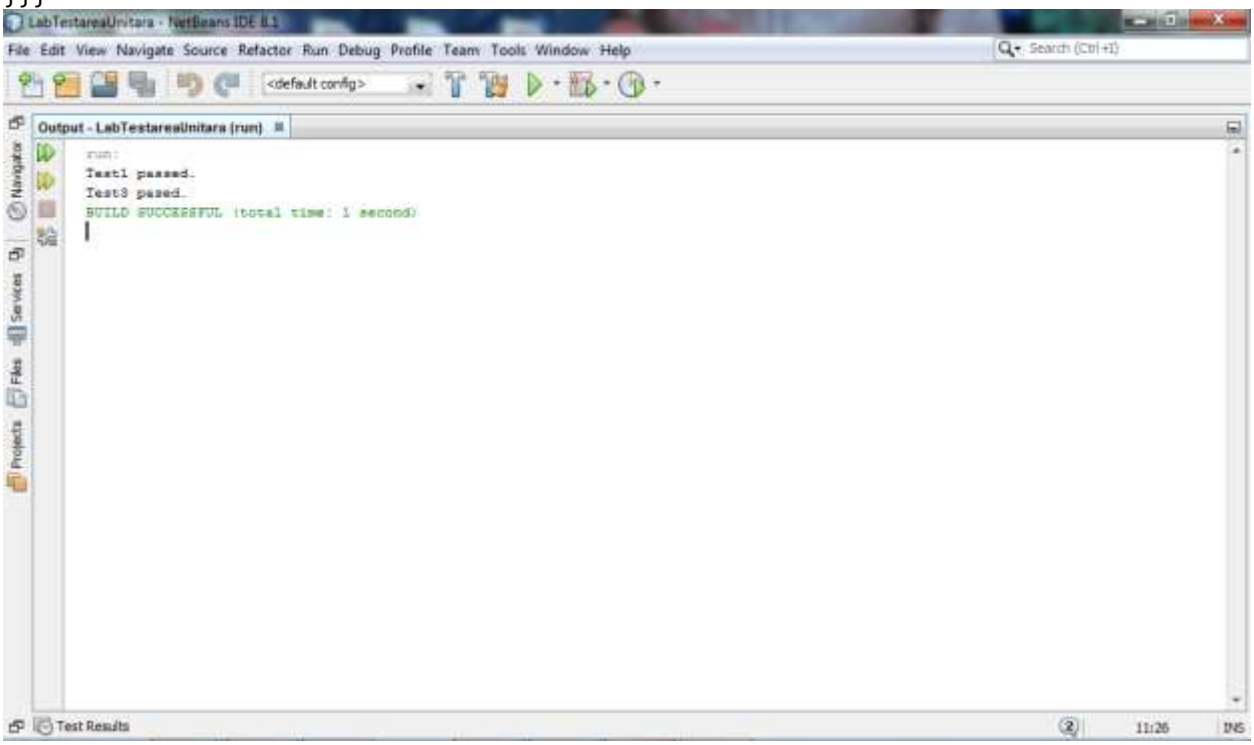
Внешняя библиотека, подключенная к проекту, может существенно облегчить разработку и поддержание модульных тестов. Наиболее популярная библиотека для Java – JUnit.

ВАРИАНТ МОДУЛЬНОГО ТЕСТИРОВАНИЯ БЕЗ БИБЛИОТЕКИ

Некоторые проверки можно поместить в сам класс. Доработаем класс CustomMath

```
public class CustomMath {
public static int sum(int x, int y){
```

```
        return x+y;
}
public static int division(int x, int y){
        if (y==0) {
                throw new IllegalArgumentException("devider is 0");
        }
        return(x/y);
}
public static void main (String[] args){
if (sum(1,3)==4){
//проверяем что при слажении 1 и 3 нам возвращается 4
        System.out.println("Test1 passed.");
}
else {
        System.out.println("Test1 failed.");
}
try {
        int z=division(1,0);
        System.out.println("Test3 failed.");
}
catch (IllegalArgumentException e){
// Тест считается успешным если попытка деления на 0
   //генерируется ожидаемое исключение
        System.out.println("Test3 pased.");
}}}
```



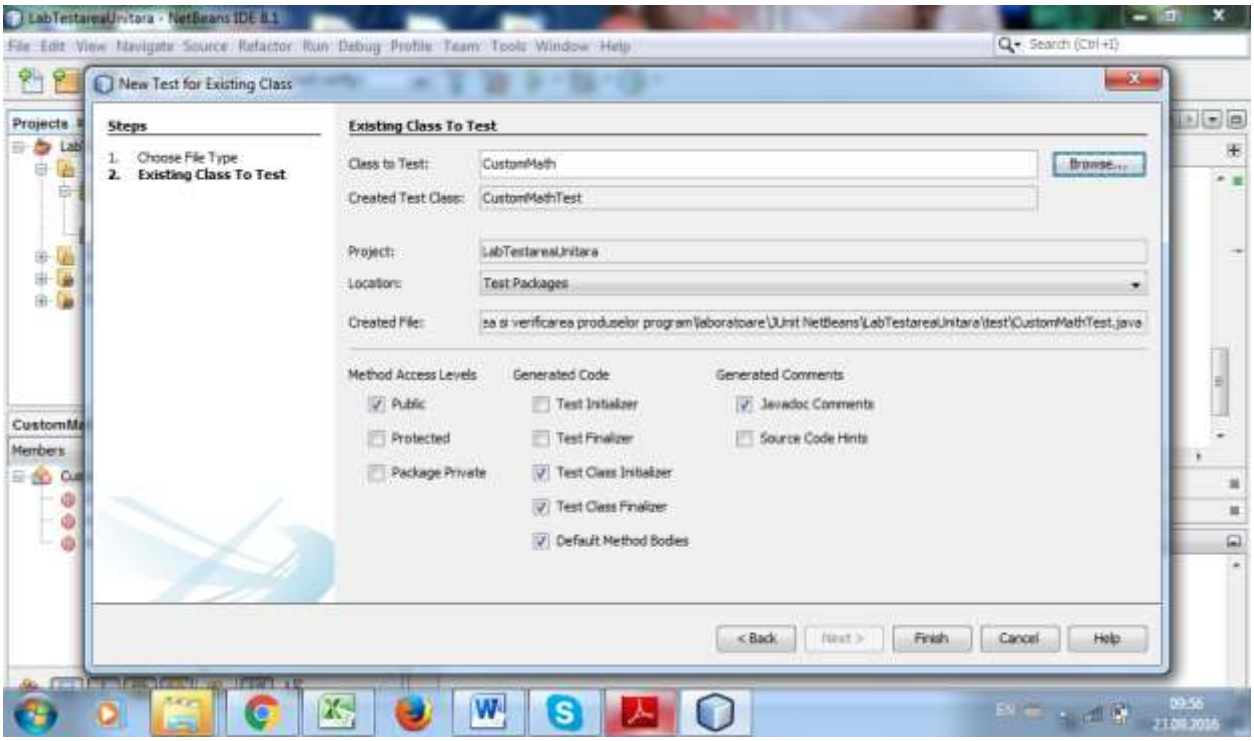
УСТАНОВКА JUNIT

JUnit может быть использован для любого Java-приложения.
Сайт проекта www.junit.org.
Библиотека входит в состав большинства интегрированных сред разработки, в том числе NetBeans.

СОЗДАНИЕ ТЕСТОВОГО МОДУЛЯ

Создание тестового модуля по шаблону может быть произведено с помощью мастера.

Тесты JUnit будут располагаться в ветке Test Packages проекта. Структура папок в Test Packages в общем случае дублирует папки классов Source Packages. Выберите в меню File->New File->… в разделе JUnit пункт Тест для существующего класса («Test for Existing Class»).



Javadoc - форма организации комментариев в коде с использованием ключевых слов, по которым NetBeans определяет существенную информацию. Если класс оформлен с использованием Javadoc – по нему может быть автоматически создана документация, а также работать контекстная подсказка NetBeans (к примеру, показывать назначение функции).

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Andrian
 */
```

```
public class CustomMathTest {

    public CustomMathTest() {
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    /**
     * Test of sum method, of class CustomMath.
     */
    @Test
    public void testSum() {
        System.out.println("sum");
        int x = 0;
        int y = 0;
        int expResult = 0;
        int result = CustomMath.sum(x, y);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

    /**
     * Test of division method, of class CustomMath.
     */
    @Test
    public void testDivision() {
        System.out.println("division");
        int x = 0;
        int y = 0;
        int expResult = 0;
        int result = CustomMath.division(x, y);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

    /**
     * Test of main method, of class CustomMath.
     */
    @Test
    public void testMain() {
        System.out.println("main");
        String[] args = null;
        CustomMath.main(args);
        fail("The test case is a prototype.");
    }

}
```

В коде тестов можно видеть аннотации: информация о назначении методов с символом @ (@BeforeClass, @AfterClass, @Test).

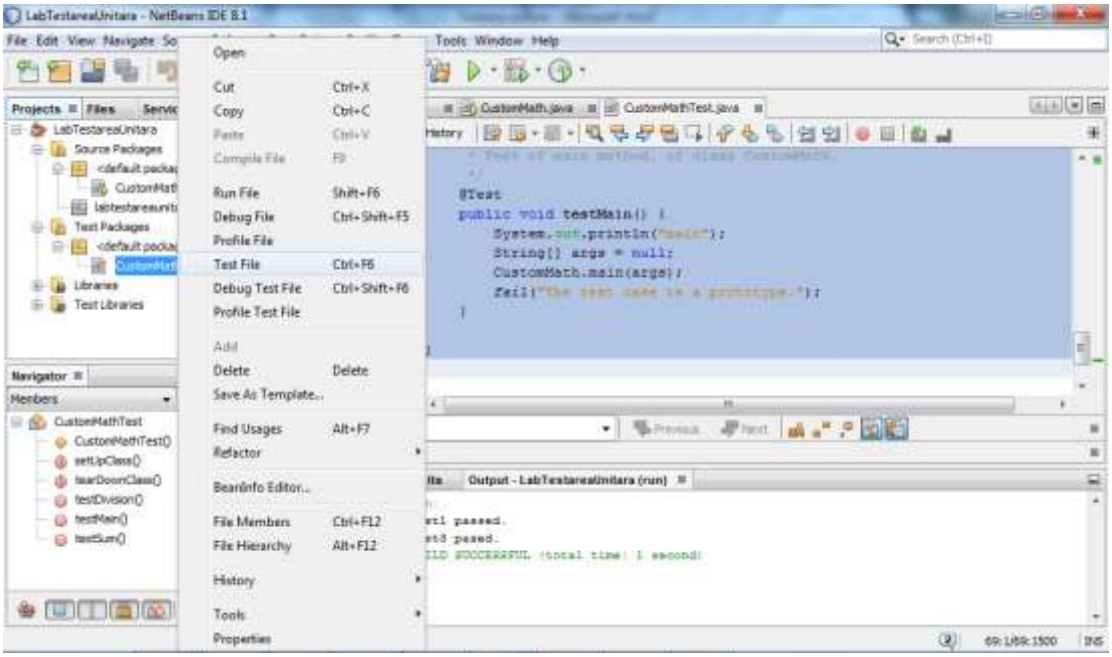
Аннотация @Test отмечает методы, автоматически запускаемые средой тестирования.

@BeforeClass и @AfterClass содержат действия, которые необходимо выполнить до запуска тестов класса (например, подключение к базе данных, или подготовку данных для

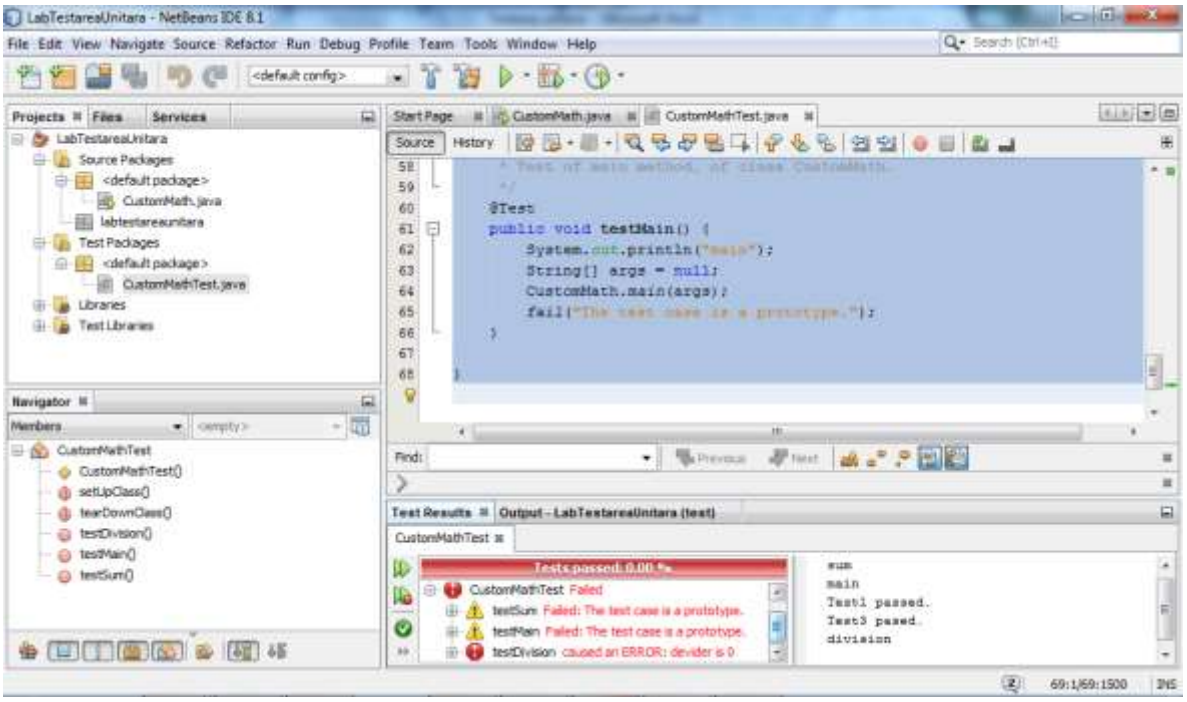
обработки) или после выполнения тестов (например, отключение от базы данных, восстановление исходного ее состояния).

ЗАПУСК СОЗДАННЫХ ПО УМОЛЧАНИЮ ТЕСТОВ

Запуск тестов выполняется через контекстное меню тестируемого класса, пунктом Test File.



Вкладка Test Results отображает выводимые на консоль сообщения (в данном случае размещенные нами в функции main проверки), а также результаты проверки методов тестируемого класса (3 неудачи).



Иерархия классов JUnit:

- java.lang.Object
- org.junit.**Assert**
- org.junit.**Assume**
- java.lang.Throwable (implements java.io.Serializable)
- java.lang.Error
- java.lang.AssertionError

org.junit.**ComparisonFailure**

- org.junit.**Test.None**

Annotation Type Hierarchy

- org.junit.**Test** (implements java.lang.annotation.Annotation)
- org.junit.**Ignore** (implements java.lang.annotation.Annotation)
- org.junit.**BeforeClass** (implements java.lang.annotation.Annotation)
- org.junit.**Before** (implements java.lang.annotation.Annotation)
- org.junit.**AfterClass** (implements java.lang.annotation.Annotation)
- org.junit.**After** (implements java.lang.annotation.Annotation)

Для проверки правильности выполнений метода в JUnit предусмотрена группа методов Assert, проверяющие условия и в случае несовпадения отмечающие тест не пройденным.

Описание методов:

Method Summary

`static void`	**`assertArrayEquals`**`(byte[] expecteds, byte[] actuals)` Asserts that two byte arrays are equal.
`static void`	**`assertArrayEquals`**`(char[] expecteds, char[] actuals)` Asserts that two char arrays are equal.
`static void`	**`assertArrayEquals`**`(int[] expecteds, int[] actuals)` Asserts that two int arrays are equal.
`static void`	**`assertArrayEquals`**`(long[] expecteds, long[] actuals)` Asserts that two long arrays are equal.
`static void`	**`assertArrayEquals`**`(java.lang.Object[] expecteds, java.lang.Object[] actuals)` Asserts that two object arrays are equal.
`static void`	**`assertArrayEquals`**`(short[] expecteds, short[] actuals)` Asserts that two short arrays are equal.
`static void`	**`assertArrayEquals`**`(java.lang.String message, byte[] expecteds, byte[] actuals)` Asserts that two byte arrays are equal.
`static void`	**`assertArrayEquals(java.lang.String message, char[] expecteds, char[] actuals) Asserts that two char arrays`**

	are equal.
static void	**assertArrayEquals(java.lang.String message, int[] expecteds, int[] actuals) Asserts that two int arrays are equal.**
static void	**assertArrayEquals(java.lang.String message, long[] expecteds, long[] actuals) Asserts that two long arrays are equal.**
static void	**assertArrayEquals(java.lang.String message, java.lang.Object[] expecteds, java.lang.Object[] actuals)**

Asserts that two object arrays are equal.

static void	**assertArrayEquals**(java.lang.String message, short[] expecteds, short[] actuals) Asserts that two short arrays are equal.
static void	**assertEquals**(double expected, double actual) **Deprecated.** *Use assertEquals(double expected, double actual, double epsilon) instead*
static void	**assertEquals**(double expected, double actual, double delta) Asserts that two doubles or floats are equal to within a positive delta.
static void	**assertEquals**(long expected, long actual) Asserts that two longs are equal.
static void	**assertEquals**(java.lang.Object[] expecteds, java.lang.Object[] actuals) **Deprecated.** *use assertArrayEquals*
static void	**assertEquals**(java.lang.Object expected, java.lang.Object actual) Asserts that two objects are equal.
static void	**assertEquals(java.lang.String message, double expected, double actual) Deprecated. Use assertEquals(String message, double expected, double actual, double epsilon) instead**
static void	**assertEquals(java.lang.String message, double expected, double actual, double delta) Asserts that two doubles or floats are equal to within a positive delta.**
static void	**assertEquals(java.lang.String message, long expected, long actual) Asserts that two longs are equal.**
static void	**assertEquals(java.lang.String message, java.lang.Object[] expecteds, java.lang.Object[] actuals) Deprecated. use assertArrayEquals**
static void	**assertEquals(java.lang.String message, java.lang.Object expected, java.lang.Object actual) Asserts that two objects are equal.**
static void	**assertFalse(boolean condition) Asserts that a condition is false.**
static void	**assertFalse(java.lang.String message,**

	boolean condition) Asserts that a condition is false.
static void	**assertNotNull(java.lang.Object object) Asserts that an object isn't null.**
static void	**assertNotNull(java.lang.String message, java.lang.Object object) Asserts that an object isn't null.**
static void	**assertNotSame(java.lang.Object unexpected, java.lang.Object actual) Asserts that two objects do not refer to the same object.**
static void	**assertNull**(java.lang.Object object) Asserts that an object is null.
static void	
static void	**assertNull**(java.lang.String message, java.lang.Object object) Asserts that an object is null.
static void	**assertSame**(java.lang.Object expected, java.lang.Object actual) Asserts that two objects refer to the same object.
static void	**assertSame**(java.lang.String message, java.lang.Object expected, java.lang.Object actual) Asserts that two objects refer to the same object.
static <T> void	**assertNotSame(java.lang.String message, java.lang.Object unexpected, java.lang.Object actual)** Asserts that two objects do not refer to the same object.
static void	***fail()*** Fails a test with no message.
static void	*fail(java.lang.String message)* Fails a test with the given message.

Функция fail() принудительно отмечает тест не некую проверку самостоятельно и она не отлавливается функцией assert.

УПРАЖНЕНИЕ 1.

Создайте проект с указанным выше классом CustomMath.

Уберите из метода main класса CustomMath проверку функции sum.

Уберите из метода testSum вызов метода fail. Убедитесь в прохождении теста функцией sum при текущих исходных данных.

Добавьте в отчет текст функции testSum и результат тестирования (скриншот окна test results).

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Исключения могут быть правильным поведением метода при определенных условиях (например, исключение отсутствия файла в случае, если он не доступен). Можно обрабатывать

исключение в тесте с помощью блока try…catch(), либо передавать его далее с помощью ключевого слова throws в описании метода.

Изменим метод testDivision таким образом, чтобы он проверял корректное поведение при делении на 0. Корректным поведением в данном случае является генерация исключения.

УПРАЖНЕНИЕ 2.

Модифицируйте тест testDivision следующим образом:

```
Test

    public void testDivisionByZero() {

        int x = 0;

        int y = 0;

        int expResult = 0;

        try {

            int result=CustomMath.division(x, y);

            assertEquals(expResult, result);

            if(y==0) fail("Деление на ноли не создает исключителънои ситуации");

            }

        catch(IllegalArgumentException e){

            if(y!=0) fail("Генерация исключения при ненулевом знаменателе");

        }

    }
```

Уберите проверку метода division из метода main класса CustomMath.

Запустите тестирование при y=0 и y!=0 (в ручную, последовательно изменяя начальное значение y).

Разместите в отчете текст теста testDivisionByZero и результаты тестирования при различных y.

ПАРАМЕТРИЗОВАННЫЕ ТЕСТЫ

Для проверки бизнес-логики приложений регулярно приходится создавать тесты, количество которых может существенно колебаться. В предшествующих версиях JUnit это приводило к значительным неудобствам - главным образом из-за того, что

изменение групп параметров в тестируемом методе требовало написания отдельного тестового сценария для каждой группы.
В версии JUnit 4 реализована возможность, позволяющая создавать общие тесты, в которые можно направлять различные значения параметров. В результате вы можете создать один тестовый сценарий и выполнить его несколько раз - по одному разу для каждого параметра.
Создание параметрического теста в JUnit 4 производится в пять шагов:
1. Создание типового теста без конкретных значений параметров.
2. Создание метода static, который возвращает тип Collection и маркирует его аннотацией @Parameter.
3. Создание полей класса для параметров, которые требуются для типового метода, описанного на шаге 1.
4. Создание конструктора, которые связывает параметры коллекции шага 2 с соответствующими полями класса, описанными на шаге 3.
5. Указание параметризованного запуска тестов с помощью класса Parametrized.
Рассмотрим указанные шаги поочередно.

ШАГ 1. СОЗДАНИЕ ТИПОВОГО ТЕСТА

В качестве типового тестируемого метода используем метод sum тестируемого класса CustomMath. Для проверки работы суммирования нам понадобятся наборы из трех параментров: два слагаемых и предполагаемое значение суммы. Тестируемый метод класса CustomMath:

```
public static int sum(int x, int y){
      return x+y; //возвращает результат слажения двух чисел
```

ШАГ 2. СОЗДАНИЕ МЕТОДА ВВОДА ПАРАМЕТРОВ

На данном шаге в классе тестов создается метод ввода параметров, объявляемый как static и возвращающий тип Collection. Этот метод необходимо снабдить аннотацией @Parameters. Внутри этого метода вам достаточно создать многомерный массив Object и преобразовать его в список List, как показано в листинге:
Метод ввода параметров с аннотацией @Parameters:

```
@Parameters

    public static Collection sumValues(){

    return Arrays.asList(new Object[][]{

        {1,1,2},

        {-1,1,0},

        {10,15,25}});

    }
```

ШАГ 3. СОЗДАНИЕ ТРЕХ ПОЛЕЙ КЛАССА ДЛЯ ХРАНЕНИЯ ПАРАМЕТРОВ

```
int x,y,sumResult;
```

ШАГ 4. СОЗДАНИЕ КОНСТРУКТОРА

Конструктор, который вы создадите на этом шаге, будет присваивать полям класса значения ваших параметров:

```
public CustomMathTest(int x,int y, int sumResult) {

        this.x=x;

        this.y=y;

        this.sumResult=sumResult;

    }
```

ШАГ 5. ЗАДАНИЕ КЛАССА PARAMETRIZED

И, наконец, на уровне класса необходимо указать, что данный тест должен исполняться с использованием параметров (Parameterized):

```
@RunWith (Parameterized.class)

public class CustomMathTest {
```

ВЫПОЛНЕНИЕ ТЕСТА

Теперь процедура тестирования будет исполняться трижды – в соответствии с числом наборов параметров для тестирования в методе sumValues.

После изменения, класс тестов будет выглядеть:

```
/*

 * To change this license header, choose License Headers in Project Properties.

 * To change this template file, choose Tools | Templates

 * and open the template in the editor.

 */
```

```
import java.util.Arrays;

import java.util.Collection;

import java.util.List;

import org.junit.AfterClass;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.*;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameters;



/**

 *

 * @author Andrian

 */

@RunWith (Parameterized.class)

public class CustomMathTest {

     @Parameters

    public static Collection sumValues(){

    return Arrays.asList(new Object[][]{

        {1,1,2},

        {-1,1,0},

        {10,15,25}});

    }

    int x,y,sumResult;
```

```
    public CustomMathTest(int x,int y, int sumResult) {
        this.x=x;
        this.y=y;
        this.sumResult=sumResult;
    }

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    /**
     * Test of sum method, of class CustomMath.
     */
    @Test
    public void testSum() {
        int expResult=sumResult;
        int result = CustomMath.sum(x, y);
        assertEquals(expResult, result);
      //  fail("The test case is a prototype.");
    }
}
```

УПРАЖНЕНИЕ 3.

Сделайте метод тестирования testDivisionByZero() параметрическим таким образом, чтобы функция проверяла деление на ноль, а также подачу корректных входных данных.

Включите в отчет окончательный вариант классов CustomMath и CustomMathTest, а также скриншот результата тестирования.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ, ПРИ СДАЧЕ РАБОТЫ

УПРАЖНЕНИЕ 4

Расширьте класс тестов таким образом, чтобы в нем применялся метод assertTrue и/или assertFalse. При необходимости, добавьте в тестируемый класс CustomMath метод, который мог бы проверяться таким образом.

Добавленный или измененные методы тестируемого класса и класса тестов включите в отчет.

ИСТОЧНИКИ

Методические указания составлены на основе:

1. Андрей Дмитриев – Модульное тестирование при помощи JUnit. Sun Microsystems.
2. Антон Сабуров – Учебный проект – Студенческий отдел кадров, глава 10. http://java-course.ru/
3. IBM developerWorks – Переходим на JUnit 4. http://www.ibm.com/developerworks/ru/edu/j-junit4/section6.html

Интернет-ресурсы:

1. www.netbeans.org – ресурс посвящен IDE NetBeans.
2. www.java.sun.com – ресурс платформы Java.
3. http://www.ibm.com/developerworks/java/ - материалы по разработке на java.
4. http://www.junit.org/ - ресурс проекта JUnit.

ТРЕБОВАНИЯ К ОТЧЕТУ ПО ЛАБОРАТОРНОЙ РАБОТЕ 2.

Отчет должен содержать:

Стандартный титульный лист (приведен ниже).

Информацию по упражнениям 1-4(указано в упражнениях).

Отчет защищается при наличии проекта, соответствующего лабораторной работе и бумажной копии отчета по работе.