

JUnit tutorial

Készítette: Budai Péter, BME IIT, 2015.

A JUnit alapú egységteszteléshez felhasználandó annotációk leírásait az alábbi URL-en találja meg:

<http://junit.sourceforge.net/javadoc/>

A JUnit keretrendszer használata során felmerülő legtöbb kérdésre a *JUnit FAQ* dokumentum (azon belül is elsősorban a 4., 5. és 7. szakasz) nyújt választ:

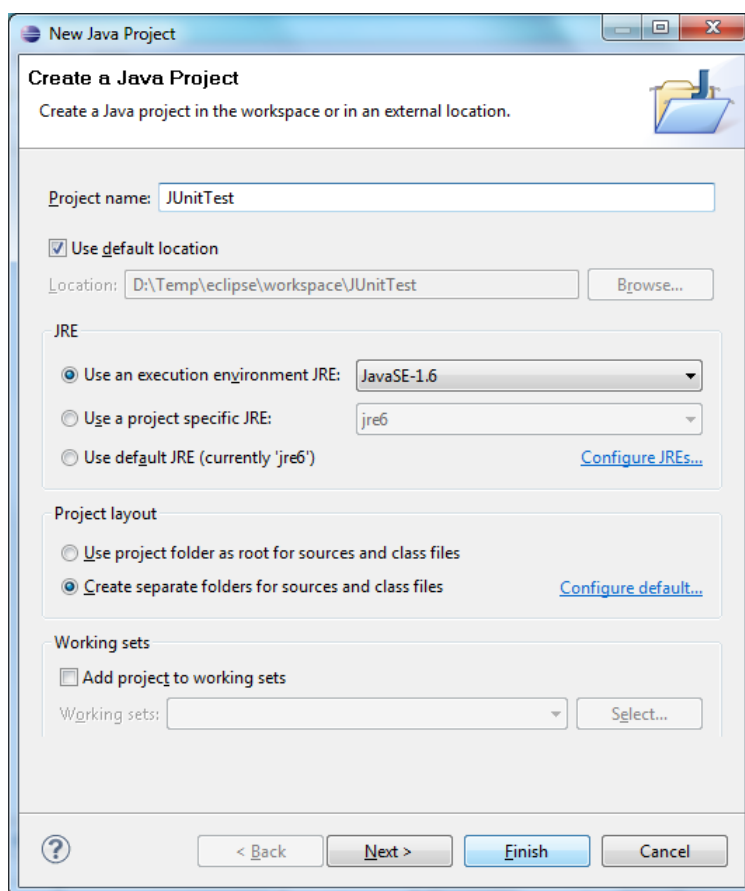
<http://junit.sourceforge.net/doc/faq/faq.htm>

Többek között a parametrikus tesztelés használatára is mutat példát az alábbi *JUnit Tutorial*:

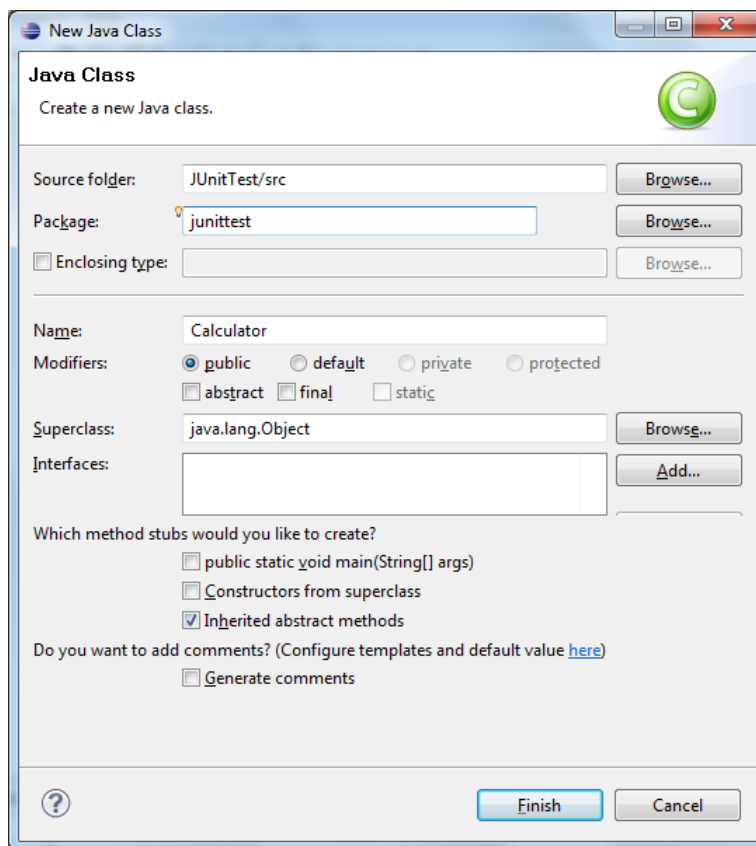
<http://www.mkymong.com/tutorials/junit-tutorials/>

1 Egy egyszerű számológép osztály létrehozása

Készítsünk egy egyszerű számológép osztályt Java nyelven, mely képes lebegőpontos számok szorzására és osztására. Ezen az osztályon fogjuk bemutatni a JUnit keretrendszer legegyszerűbb lehetőségeit. Ehhez hozzunk létre egy új Java projektet az Eclipse-en belül, legyen a neve **JUnitTest**.



A projektben hozzunk létre egy új, **Calculator** nevű osztályt a **junittest** csomagon belül:



Végül valósítsuk is meg a számológépünk funkcióit a **Calculator** osztályban. Ehhez vegyünk fel két metódust, egyet a szorzásnak és egyet az osztásnak. Az osztást megvalósító metódusnál figyeljünk rá, hogy a nullával való osztást elkerüljük. Ilyenkor dobjunk **IllegalArgumentException**!

```
package junittest;

public class Calculator {

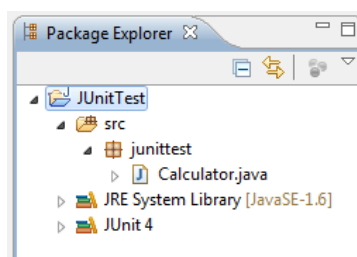
    public double multiply(double a, double b) {
        return a * b;
    }

    public double divide(double a, double b)
        throws IllegalArgumentException {
        if(b == 0) {
            throw new IllegalArgumentException();
        }
        return a / b;
    }
}
```

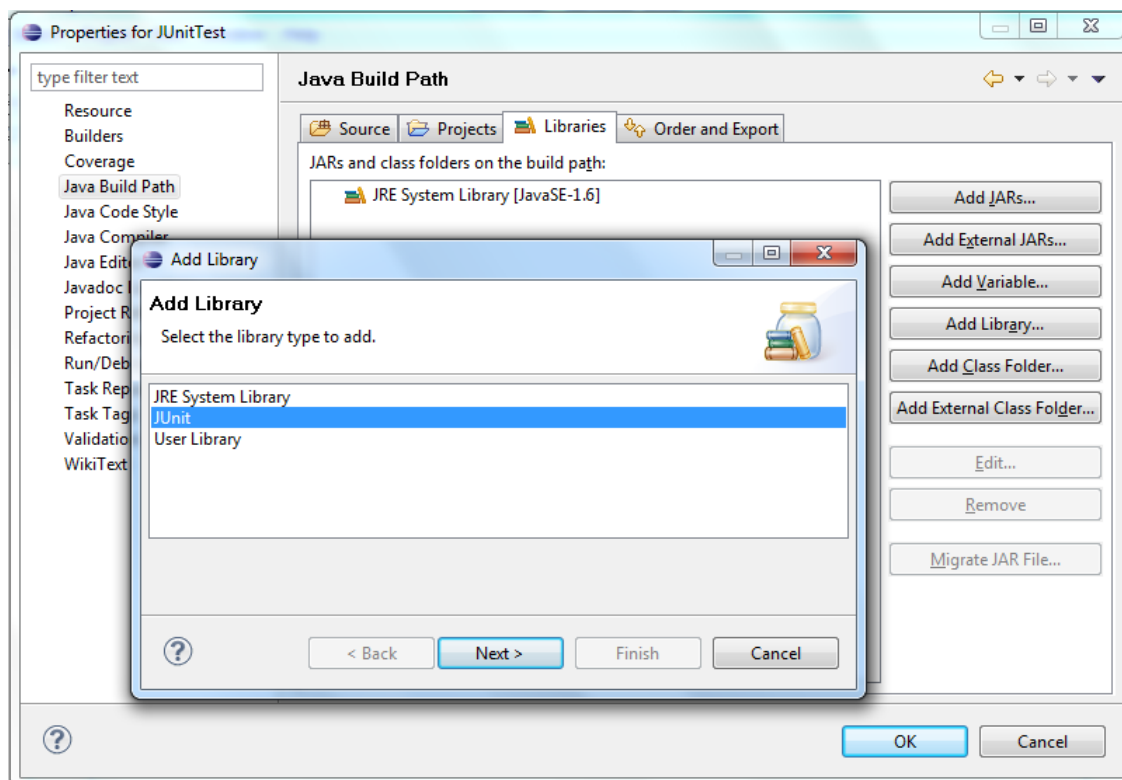
2 A számítógép osztály tesztelése a JUnit 4 keretrendszerrel

A **JUnit** egy nyílt forráskódú Java osztálykönyvtár, mely mára a Java nyelven írt programok egység-tesztelésének (unit testing) egyik legnépszerűbb megoldásává vált. A JUnit keretrendszer olyan osztályokat tartalmaz, melyek kényelmesebbé, átláthatóbbá és megismételhetővé teszik a programok alapvető egységeinek (osztályoknak és metódusoknak) a tesztelését.

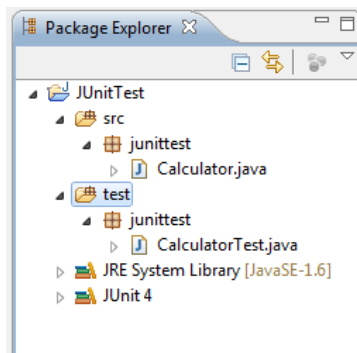
A JUnit keretrendszert általában a különböző fejlesztő-környezetek is támogatják, így az Eclipse is. Ha használni szeretnénk, a JUnit osztálykönyvtárát hozzá kell adnunk a projektünkhöz, ahogy az alábbi ábrán is látható.



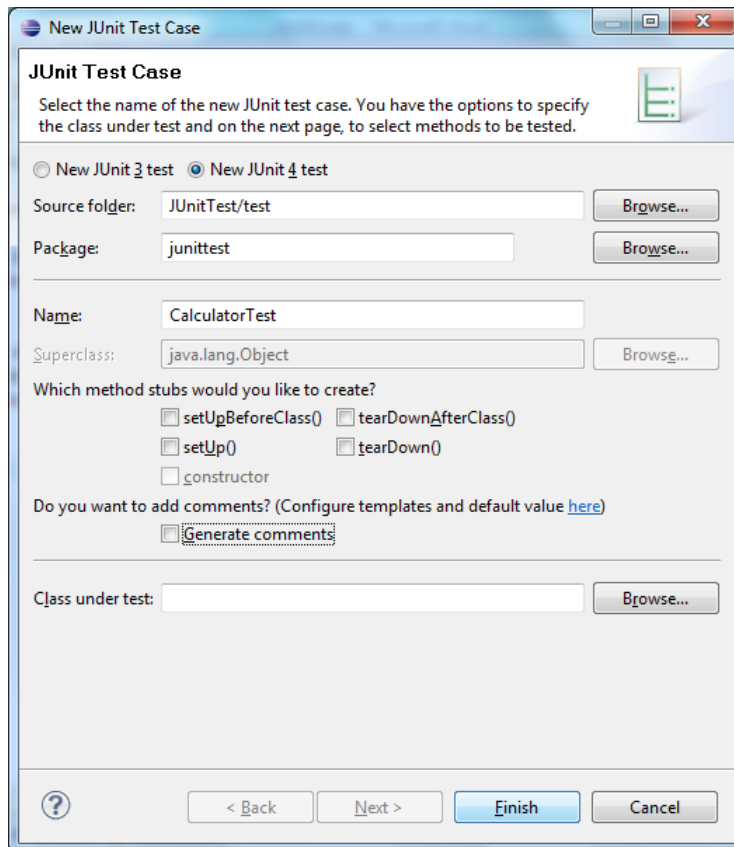
Ehhez a projekt nevén jobb egérgombbal való kattintásra felugró menüből válasszuk a legalsó, **Properties** menüpontot, majd a megnyíló ablakban navigáljunk a **Java Build Path, Libraries, Add Library...** lehetőséghez. Az újabb ablakban válasszuk ki a JUnit könyvtárat, a következő képernyőn pedig a legördülő menüből a **JUnit 4** verziót. A korábbi verziót másképpen kell használni, ezen laborgyakorlat során nem foglalkozunk vele. Az alábbi ábra segít megtalálni a kérdéses dialógusablakot.



Egy szoftver fejlesztése során a tesztesztályokat általában külön kezeljük az alkalmazás logikától, hiszen azok nem kerülnek majd bele a lefordított és összecsomagolt **.jar** fájlba. Ezért az Eclipse projektünkben létre kell hozni egy új forráskönyvtárat a teszt-állományok számára. Ehhez kattintsunk jobb egér-gombbal a projekten, majd válasszuk a **New, Source Folder** lehetőséget. A könyvtár neve legyen **test**, ahogy az alábbi ábrán is látszik.



A tesztállományokat itt ugyanolyan package struktúrába szervezhetjük, mint a programunk többi részét. A konvenció szerint a tesztesztályokat ugyanabba a csomagba helyezzük el, ahol a tesztelt osztály is található. Így a tesztesztályok hozzáférhetnek az alapértelmezett (*default, package*) láthatóságú elemekhez is, a külön forráskönyvtár miatt azonban nem kerülnek bele a végleges szoftverbe.



Most pedig hozzunk létre ez új **JUnit Test Case**-t a **test** könyvtárunkon belül. A neve legyen **CalculatorTest**, és tegyük a **junittest** package-be. Ezt mutatja a fenti ábra. Ha nem találjuk a **JUnit Test Case** opciót, akkor sincs baj, tökéletesen megfelel, ha csak egyszerűen egy új Java osztályt hozzunk létre ugyanilyen névvel, ugyanebben a package-ben.

Amikor a JUnit segítségével végezzük a tesztelést, a különböző teszteseteinket külön Java osztályok és metódusok formájában készítjük el, melyeket a megfelelő annotációkkal (**@Test** és társai) látunk el. Ezek alapján a keretrendszer fogja lefuttatni a teszteket, és a végén összesíti az eredményeket. Egy tesztesztosztályban több tesztmetódust is elhelyezhetünk. A tesztmetódusokban az **org.junit.Assert** osztály **assertXXX()** metódusait használhatjuk arra, hogy összevessük a várt és tényleges eredményeket. Ezek a metódusok automatikusan sikertelennek minősítik az tesztesetet, amint a eltéréseket tapasztalnak a várt és tényleges értékek között. A JUnit ráadásul lehetőséget ad arra is, hogy megjelölhessük, ha valamely tesztmetódus során egy kivétel megjelenése a kívánatos esemény. Ilyenkor a kivétel hiánya jelenti a hibás teszteredményt. Ennek pontos módja a FAQ-ban olvasható.

Most pedig készítsük el a teszteseteinket. A példa kedvéért most meglehetősen felületesek leszünk, és csak egyetlen számpárra próbáljuk ki mindkét műveletet. Viszont azt külön teszteljük, hogy a nullával való osztást jól kezeli-e az osztályunk, és valóban kivétel keletkezik-e.

```
package junittest;

import org.junit.Assert;
import org.junit.Test;

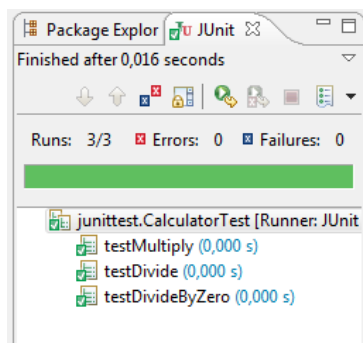
public class CalculatorTest {

    @Test
    public void testMultiply() {
        Calculator calc = new Calculator();
        double result = calc.multiply(5.0, 8.0);
        Assert.assertEquals(40.0, result, 0);
    }

    @Test
    public void testDivide() throws Exception {
        Calculator calc= new Calculator();
        double result = calc.divide(20.0, 4.0);
        Assert.assertEquals(5.0, result, 0);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testDivideByZero() throws Exception {
        Calculator calc = new Calculator();
        calc.divide(10.0, 0.0);
    }
}
```

Ha elkészült a tesztosztályunk, akkor az Eclipse **Package Explorer** ablakában jobb egérgombbal rákattintunk (vagy a projekt nevére), és kiválasztjuk a **Run As/JUnit Test** lehetőséget. A fejlesztőkörnyezet le fogja futtatni nekünk az osztályban található teszteket, és egy ablakban meg is jeleníti azok eredményeit, ahogy az a következő ábrán látható is:



3 Tesztkörnyezetek használata

A JUnit keretrendszerben a tesztmetódusok egymástól függetlenül kerülnek végrehajtásra, az egyik teszt eredménye nem befolyásolja a másikat. Ezt úgy éri el a JUnit, hogy minden tesztmetódus végrehajtásához egy teljesen új példányt hoz létre a tesztszóból.

Összetettebb komponensek esetén gyakori eset, hogy egy adott művelet teszteléséhez a vizsgált objektumot előzőleg egy meghatározott kiindulási állapotba kell hozni. Az ehhez szükséges (néha nagy számú) lépések valójában logikailag nem tartoznak a teszthez. Ráadásul ha több tesztet is felhasználja ugyanazt a kiinduló állapotot, ezeket minden egyes tesztmetódusban meg kellene ismételni. A JUnit ezért lehetőséget ad arra, hogy ezeket a tesztinicializáló (és esetleg -lezáró) kódrészleteket külön metódusokba helyezzük el a tesztszobon belül, melyeket aztán minden tesztmetódus előtt (és után) meghív. Így módon a tesztek számára egy úgynevezett tesztkörnyezetet (**test fixture**) alakíthatunk ki. Az inicializáló és lezáró metódusokat szintén annotációk (**@Before** és **@After**) jelölik.

Most alakítsuk át a tesztszobunkat úgy, hogy számológép objektum létrehozását ne kelljen minden tesztmetódusban elvégezni, ezt helyezzük át a speciális tesztkörnyezet-inicializáló metódusba! A számológépes példánál ezzel sokat nem nyerünk, de az elvet jól demonstrálja.

```
package junittest;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class CalculatorTest {

    Calculator calc;

    @Before
    public void setUp() {
        calc = new Calculator();
    }

    @Test
    public void testMultiply() {
        double result = calc.multiply(5.0, 8.0);
        Assert.assertEquals(40.0, result, 0);
    }

    @Test
    public void testDivide() throws Exception {
        double result = calc.divide(20.0, 4.0);
        Assert.assertEquals(5.0, result, 0);
    }

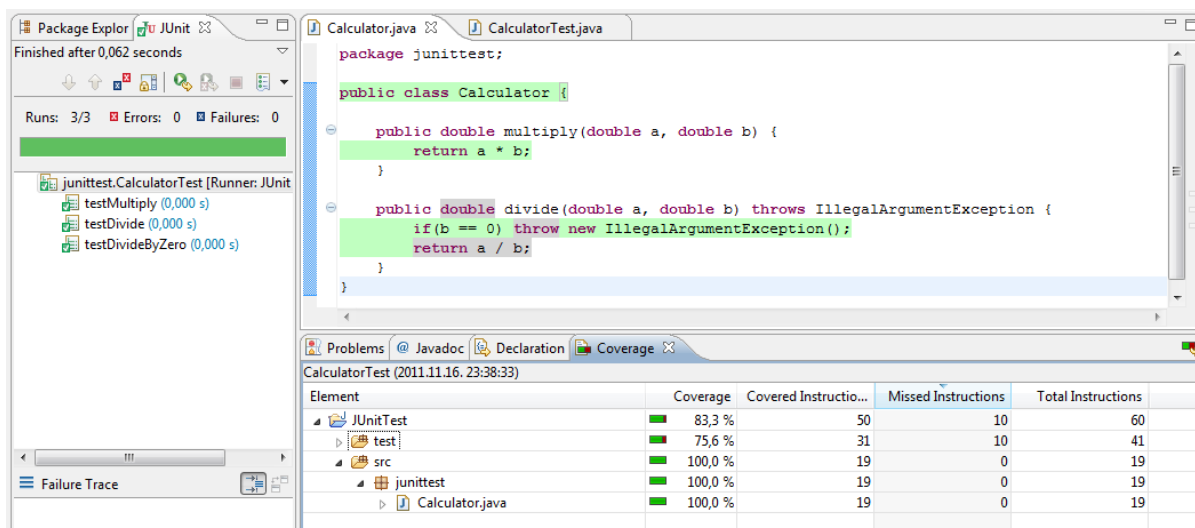
    @Test(expected=IllegalArgumentException.class)
    public void testDivideByZero() throws Exception {
        calc.divide(10.0, 0.0);
    }
}
```

Figyeljünk arra, hogy a létrehozott számológép példányt elérhetővé kell tenni a tesztmetódusok számára, ezért tagváltozóként kellett felvennünk. Ha mindent jól csináltunk és újra lefuttatjuk a teszteket, akkor az eredmény nem változott, a teszt továbbra is sikeres.

4 Kódlefedettség-vizsgálat

Az **Eclipse** képes arra, hogy egy tetszőleges Java program futtatása során (így egy JUnit teszt esetén is) feljegyezze, hogy a végrehajtás mely utasításokat érintette (akár JVM bytecode szinten is), és erről részletes kimutatást ad osztályokra lebontva. Az **Eclipse** lehetővé teszi, hogy egy kattintással elvégezzük a kódlefedettség-mérést, ráadásul a forráskódban is automatikusan megjelöli azokat a programsorokat, melyeket érintett a végrehajtás.

A kódfedettség-mérés nagyon egyszerű, a **Package Explorer** ablakban kattintsunk jobb egérgombbal a projekt nevére, majd most a **Run As** helyett a **Coverage As, JUnit Test** menüpontot kell választanunk. A tesztek megszokott módon történő lefutása után megjelenik egy **Coverage** ablak, ahol fájlokra lebontva megtekinthető a kódlefedettségi statisztika. A fájl nevére duplán kattintva pedig megnyílik a forráskód, melyben megfelelően színezett sorok mutatják a végrehajtás által érintett területeket. Egy példa látható minderre a következő oldali ábrán:



Element	Coverage	Covered Instruction...	Missed Instructions	Total Instructions
JUnitTest	83,3 %	50	10	60
test	75,6 %	31	10	41
src	100,0 %	19	0	19
unittest	100,0 %	19	0	19
Calculator.java	100,0 %	19	0	19

5 Paraméteres tesztelés JUnit segítségével

Főként különböző algoritmusok fejlesztése során tipikus feladat, hogy egy bizonyos műveletet több különböző bemeneti adatra is le kell tesztelni, hogy megbizonyosodjunk arról, hogy a program szélsőséges esetekben is helyesen működik. Ilyenkor minden bemeneti adathoz külön tesztmetódust kellene készíteni, ami szélesebb paramétertartománynál már kezelhetetlenné válik. Szerencsére a JUnit kínál megoldást ennek kikerülésére is a paraméteres tesztek (**parameterized test**) formájában.

Paraméteres teszt esetén a tesztsztyál egy, a megfelelő annotációval (**@Parameters**) ellátott statikus metódusán keresztül adjuk meg a bemeneti adatsorokat, és az osztályban szereplő tesztmetódusokat pontosan annyiszor fogja lefuttatni a keretrendszer, ahány ilyen különböző bemeneti adatsor van. A tesztek egyszerre egy adatsoron kerülnek lefuttatásra, melyet a konstruktorban kap meg a tesztobjektum. Mindezeket egy példával is illusztrálja a *JUnit Tutorial* (lásd fent) megfelelő szakasza.

A statikus metódus **Object** tömbök kollekciójával (**Collection<Object[]>** vagy **List<Object[]>**, stb.) tér vissza. A kollekció elemszáma (**.size()**) adja meg, hogy hányszor kell lefuttatni a tesztet (hányféle különböző adatsor van). A kollekcióban szereplő **Object[]** típusú elemek pedig egy-egy adatsort tartalmaznak (azért tömb, mert előfordulhat, hogy egy teszthez több paraméter tartozhat, például bemeneti értékek és elvárt kimeneti értékek párpai).

Az adatsort tartalmazó, **Object[]** típusú tömb elemszámának (**.length**) meg kell egyeznie a tesztsztyál konstruktora argumentumainak számával. Ugyanis a JUnit a tömb elemeit fogja sorban értékül adni a konstruktor argumentumainak, amikor példányosítja a tesztsztyált. Tehát ha a konstruktornak három argumentuma van és ezekből összesen négyféle ilyen hármas kombinációval kell lefuttatni a tesztet, akkor a metódus egy négyelemű kollekcióval tér vissza, melyben háromelemű tömbök vannak.

A gyakorlatban a konstruktorban kapott paramétereket általában eltároljuk a tesztsztyál egy tagváltozójában, hogy később a tesztmetódusokban használhassuk azokat.

A parametrikus tesztek speciálisak abból a szempontból, hogy az alapértelmezett JUnit végrehajtó motor nem képes lefuttatni őket. Ezért a tesztsztyálunkat egy speciális annotációval (**@RunWith**) kell ellátni, melynek a paraméterében megadhatjuk, hogy a JUnit azt a végrehajtó osztályt (**org.junit.runners.Parameterized**) használja, amely fel van készítve a parametrikus tesztek kezelésére.

Most pedig alakítsuk át a tesztsztyálunkat úgy, hogy ne csak egy-egy számpárral végezze el a szorzás és osztás műveletek tesztelését, hanem öt különböző operanduspárral:

```
package junittest;

import java.util.ArrayList;
import java.util.List;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class CalculatorTest {

    double a;
    double b;
    Calculator calc;

    public CalculatorTest(double a, double b) {
        this.a = a;
        this.b = b;
    }

    @Before
    public void setUp() {
        calc = new Calculator();
    }

    @Test
    public void testMultiply() {
        double result = calc.multiply(a, b);
        Assert.assertEquals(a * b, result, 0);
    }

    @Test
    public void testDivide() throws Exception {
        double result = calc.divide(a, b);
        Assert.assertEquals(a / b, result, 0);
    }

    @Test(expected=IllegalArgumentException.class)
    public void testDivideByZero() throws Exception {
        calc.divide(a, 0.0);
    }

    @Parameters
    public static List<Object[]> parameters() {
        List<Object[]> params = new ArrayList<Object[]>();
        params.add(new Object[] {0.0, 0.0});
        params.add(new Object[] {10.0, 0.0});
        params.add(new Object[] {10.0, 3.0});
        params.add(new Object[] {20.0, 4.0});
        params.add(new Object[] {40.0, 5.0});
        return params;
    }
}
```

Ha lefuttatjuk a parametrikus tesztünket, azt tapasztaljuk, hogy már nem hibátlan eredményt kapunk (lásd az alábbi ábrát). Az első két adatpárnál, ahol a második paraméter értéke nulla volt, az osztást tesztelő metódus nem várt kivételt (**IllegalArgumentException**) kapott. Ez természetesen nem a **Calculator** osztály hibája, a teszteseteink vannak rosszul megválasztva, hiába, a jó tesztek tervezése egy külön tudományterület.

