Testen mit JUnit

- Motivation
- Demo – Beispiel (Auszüge)
- Prüf-Methoden
- Nützliches

Motivation: automatisiertes Testen (aus PR1)

- Woher wissen wir, ob eine Funktion einen korrekten Wert für einen bestimmten aktuellen Parameter liefert?

→Wir testen sie, d.h. wir rufen sie auf und prüfen das Ergebnis

- Beispiel:

```
public class Maximum {

    static int max( int value1, int value2 ) {
        if (value1 > value2)
            return value1;
        else
            return value2;
    }

    public static void main( String[] args ) {

        println(max(readInt(), readInt()));

    }

}
```

Frage:
Mit welchem/n Wert(en) rufen wir die Methode auf?

Antwort:

Frage:
Was machen wir, wenn wir einen Fehler entdecken und die Methode verbessert haben?

Antwort:

HOCHSCHULE DER WIRTSCHAFT FÜR MANAGEMENT
UNIVERSITY OF APPLIED MANAGEMENT STUDIES

Motivation: automatisiertes Testen (aus PR1)

- Frage:
  Was machen wir, wenn eine Aufgabe am Computer immer und immer wieder durchgeführt werden soll?

→ Wir schreiben ein Programm, das das für uns tut!

- Am Beispiel: So könnte ein Stück eines **Testprogramms** für die Methode "max" aussehen:

```
public static void main( String[] args ) {

    int result = max(3, 5);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

    result = max(5, 3);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

    result = max(5, 5);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

}
```

Motivation: automatisiertes Testen (aus PR1)

- Das Testprogramm ist praktisch:
  Immer wenn man etwas an "max" geändert hat, sagt es einem "auf Knopfdruck“ (d.h. man muss es nur erneut starten), ob man dabei vielleicht einen Fehler eingebaut hat
- Anmerkung:
  Für „echte“ Funktionen sieht man das nicht so einfach...!

```
public static void main( String[] args ) {

    int result = max(3, 5);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

    result = max(5, 3);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

    result = max(5, 5);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

}
```

Motivation: automatisiertes Testen (aus PR1)

- Das Testprogramm ist aber auch umständlich/geschwätzig
- Die Bibliothek JUnit vereinfacht das Schreiben von Testprogrammen

```
assertEquals(5, max(3, 5));
assertEquals(5, max(5, 3));
assertEquals(5, max(5, 5));
```

kürzerer Testcode mit JUnit

```
public static void main( String[] args ) {

    int result = max(3, 5);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

    result = max(5, 3);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

    result = max(5, 5);
    if (result != 5)
        println("Falsches Ergebnis: 5 erwartet, " + result + " geliefert");

}
```

Motivation: automatisiertes Testen (aus PR1)

- Das Testprogramm ist aber auch umständlich/geschwätzig
- Die Bibliothek JUnit vereinfacht das Schreiben von Testprogrammen

```
assertEquals(5, max(3, 5));
assertEquals(5, max(5, 3));
assertEquals(5, max(5, 5));
```

kürzerer Testcode mit JUnit

Sollwert vorne

Funktionsaufruf zur Bestimmung des Ist-Wertes als 2. Parameter

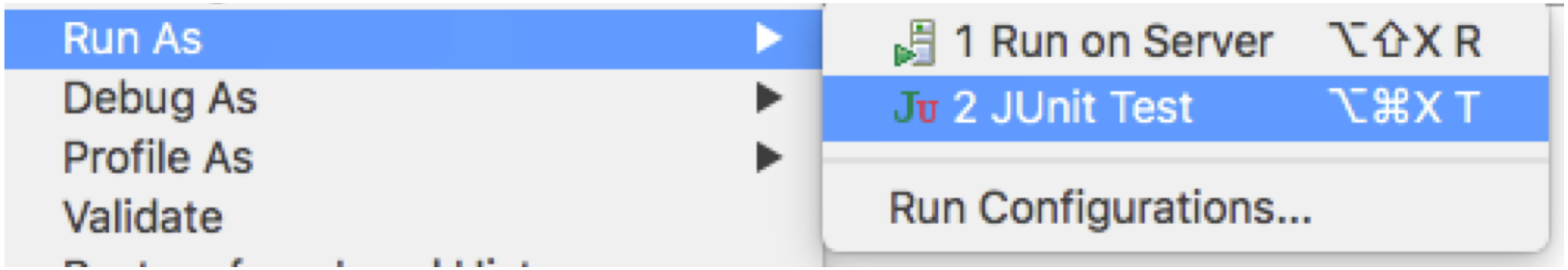
HOCHSCHULE DER WIRTSCHAFT FÜR MANAGEMENT
UNIVERSITY OF APPLIED MANAGEMENT STUDIES

Tests sind auch Programme

```
import static org.junit.Assert.*;
import org.junit.Test;

public class MaximumTest {

    @Test
    public void maximum() throws Exception {

        assertEquals(5, max(3, 5));
        assertEquals(5, max(3, 5));
        assertEquals(5, max(3, 5));

    }

}
```

Alle Methoden einer Test-Klasse, die mit @Test *annotiert* sind, werden nacheinander ausgeführt...

... wenn man die Klasse als JUnit-Test ausführt:

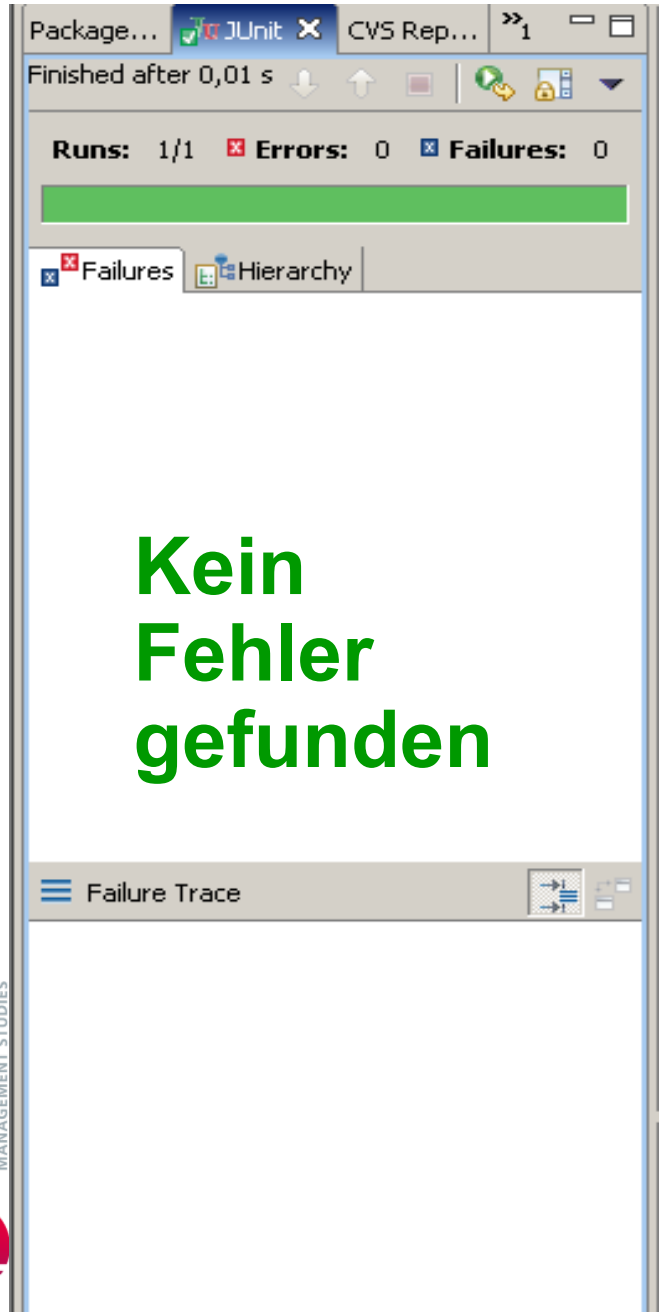


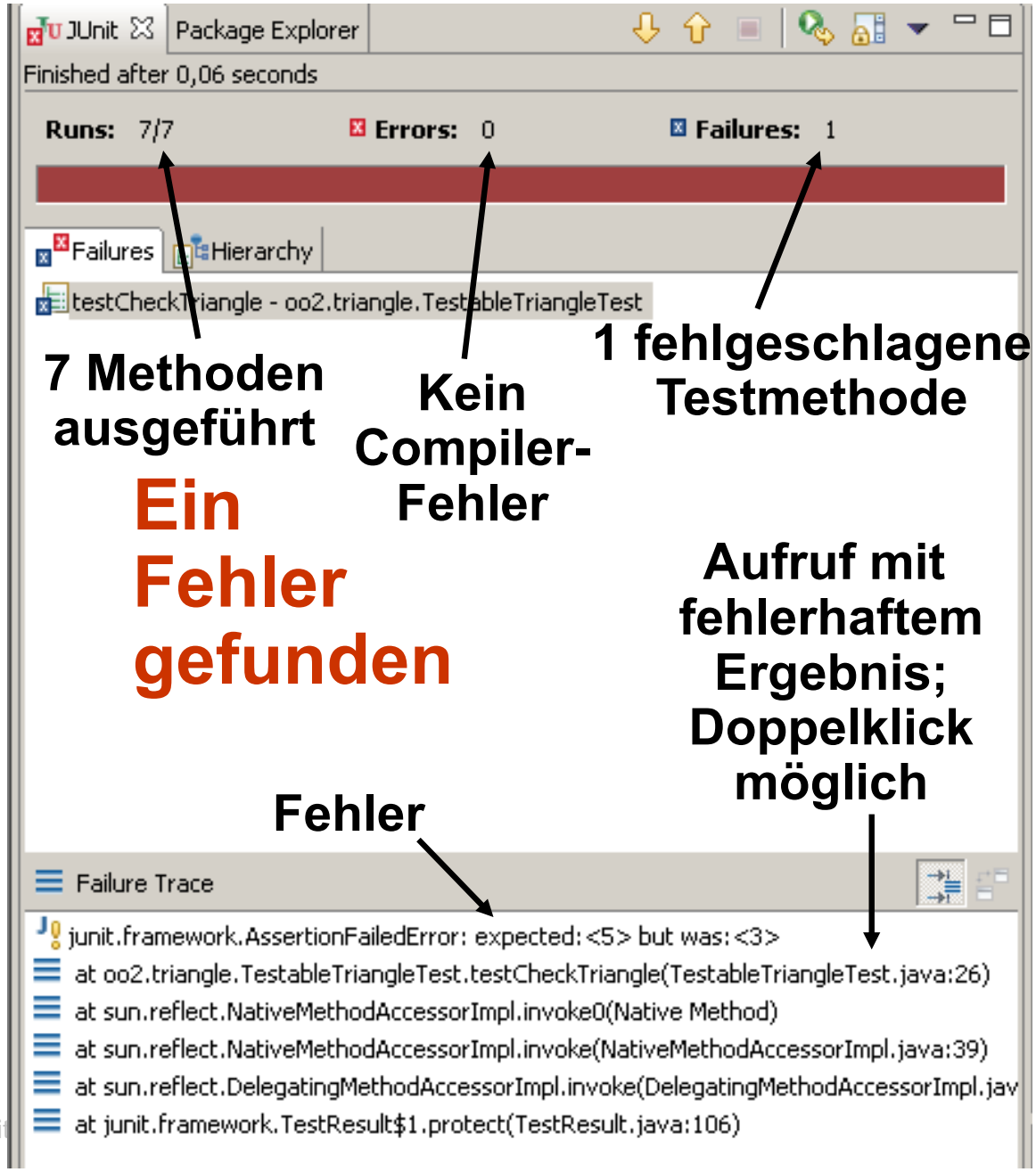
Tests sind auch Programme

```
import static org.junit.Assert.*;
import org.junit.Test;

public class MaximumTest {

    @Test
    public void maximum() throws Exception {

        assertEquals(5, max(3, 5));
        assertEquals(5, max(3, 5));
        assertEquals(5, max(3, 5));

    }

}
```

Ansonsten handelt es sich um eine „ganz normale“ Methode

Testprogramm zu einer Klasse ausführen: negatives / positives Ergebnis





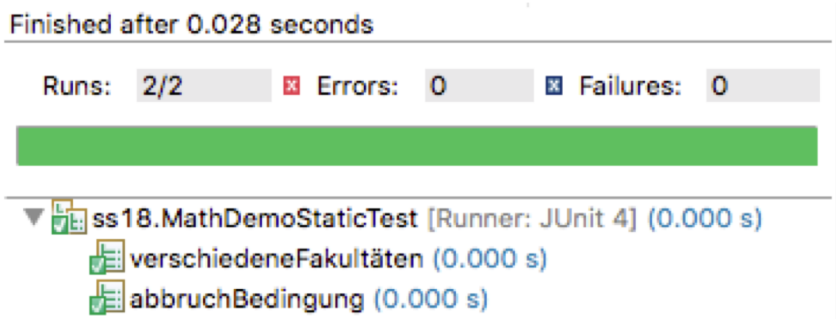
Anderes Beispiel: zu testende Methode in ihrer Klasse

```
public class MathDemoStatic {

    public static int faculty(int n) {
        if (n == 0)
            return 1;
        else
            return n * faculty(n - 1);
    }

}
```

Erste Testmethode in einer Testklasse

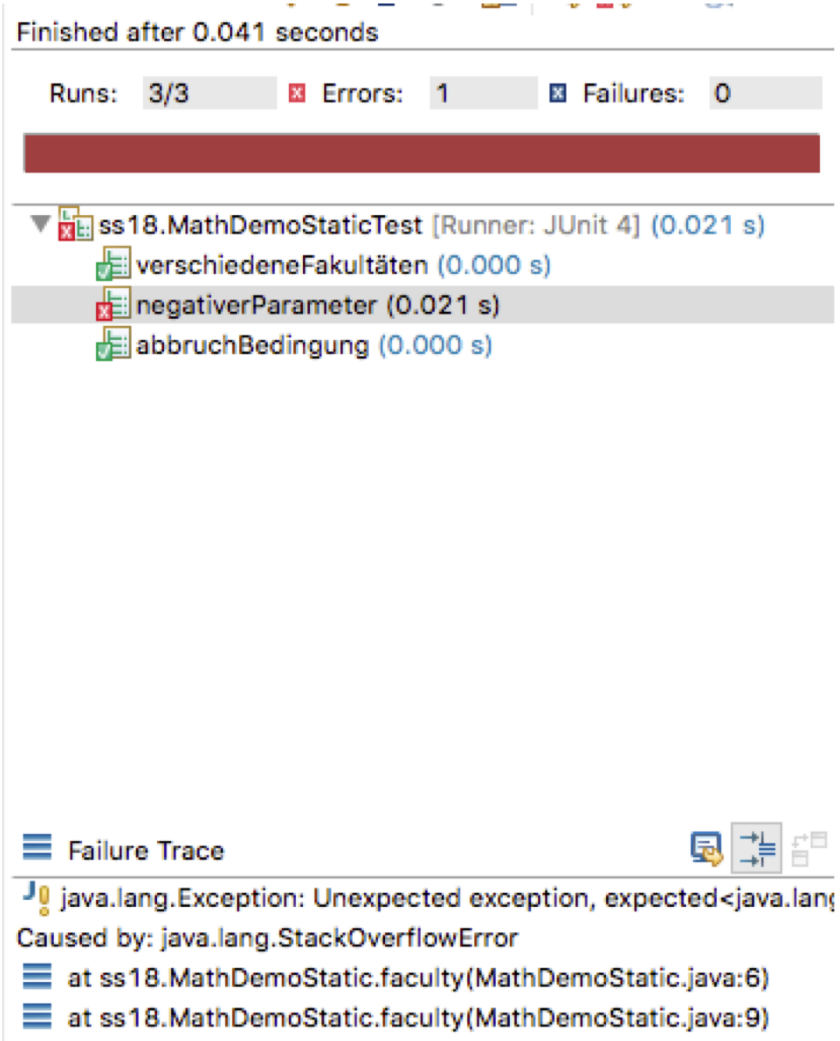
```
public class MathDemoStaticTest {

    @Test
    public void verschiedeneFakultäten() throws Exception {
        assertEquals(1, MathDemoStatic.faculty(1));
        assertEquals(2, MathDemoStatic.faculty(2));
        assertEquals(6, MathDemoStatic.faculty(3));
        assertEquals(24, MathDemoStatic.faculty(4));
    }

}
```

Ausführen *aller* Testmethoden einer Testklasse




```
2
3  import static org.junit.Assert.*;
6
7  public class MathDemoStaticTest {
8
9      @Test
10     public void verschiedeneFakultäten() throws Exception {
11         assertEquals(1, MathDemoStatic.faculty(1));
12         assertEquals(2, MathDemoStatic.faculty(2));
13         assertEquals(6, MathDemoStatic.faculty(3));
14         assertEquals(24, MathDemoStatic.faculty(4));
15     }
16
17     @Test
18     public void abbruchBedingung() throws Exception {
19         assertEquals(1, MathDemoStatic.faculty(0));
20     }
21
22 }
```

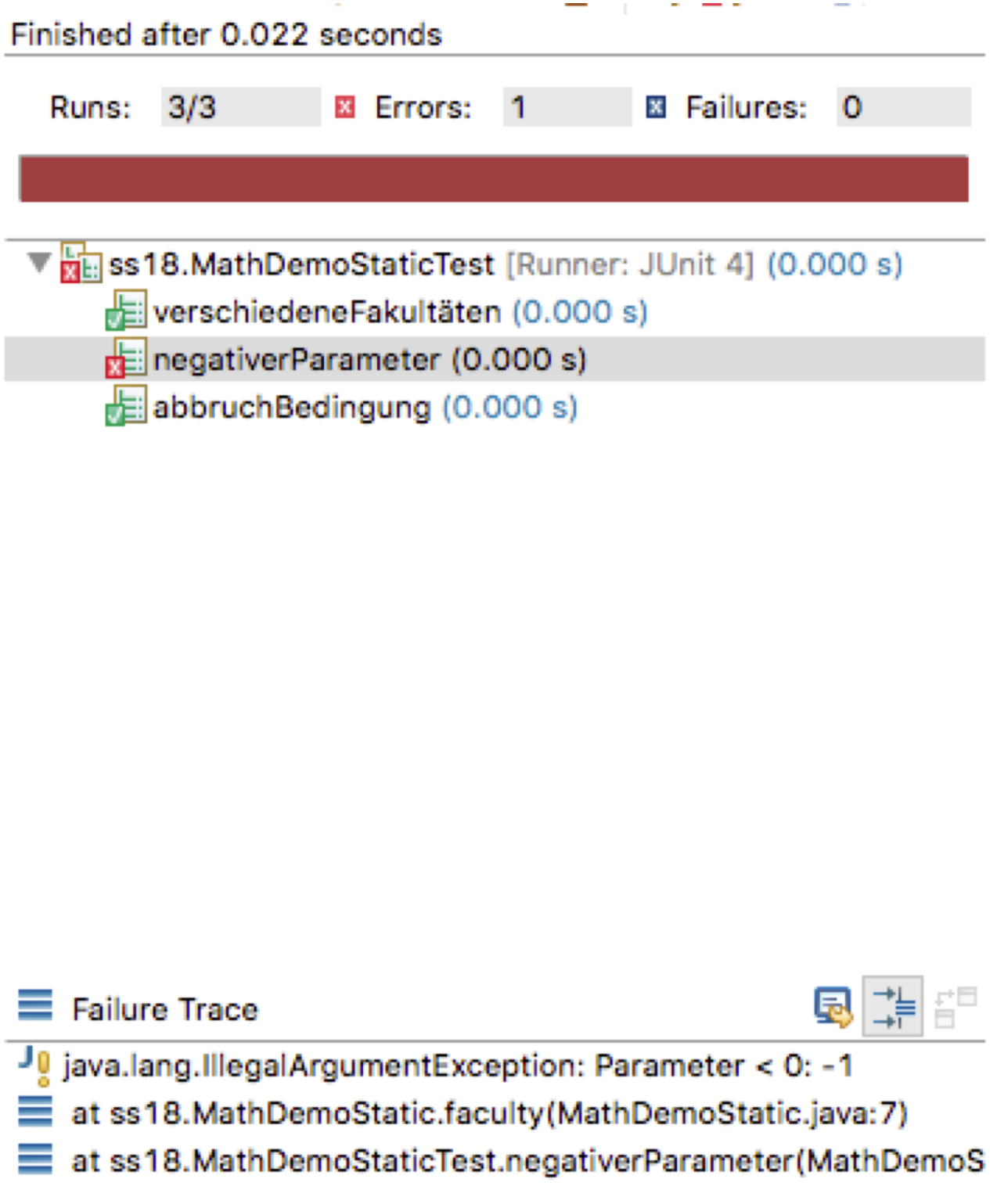
Eine Testmethode schlägt Alarm




```
 7  public class MathDemoStaticTest {
 8
 9      @Test
10      public void verschiedeneFakultäten() throws Exception {
11          assertEquals(1, MathDemoStatic.faculty(1));
12          assertEquals(2, MathDemoStatic.faculty(2));
13          assertEquals(6, MathDemoStatic.faculty(3));
14          assertEquals(24, MathDemoStatic.faculty(4));
15      }
16
17      @Test
18      public void abbruchBedingung() throws Exception {
19          assertEquals(1, MathDemoStatic.faculty(0));
20      }
21
22      @Test(expected=IllegalArgumentException.class)
23      public void negativerParameter() throws Exception {
24          MathDemoStatic.faculty(-1);
25      }
26
27  }
28
29
```

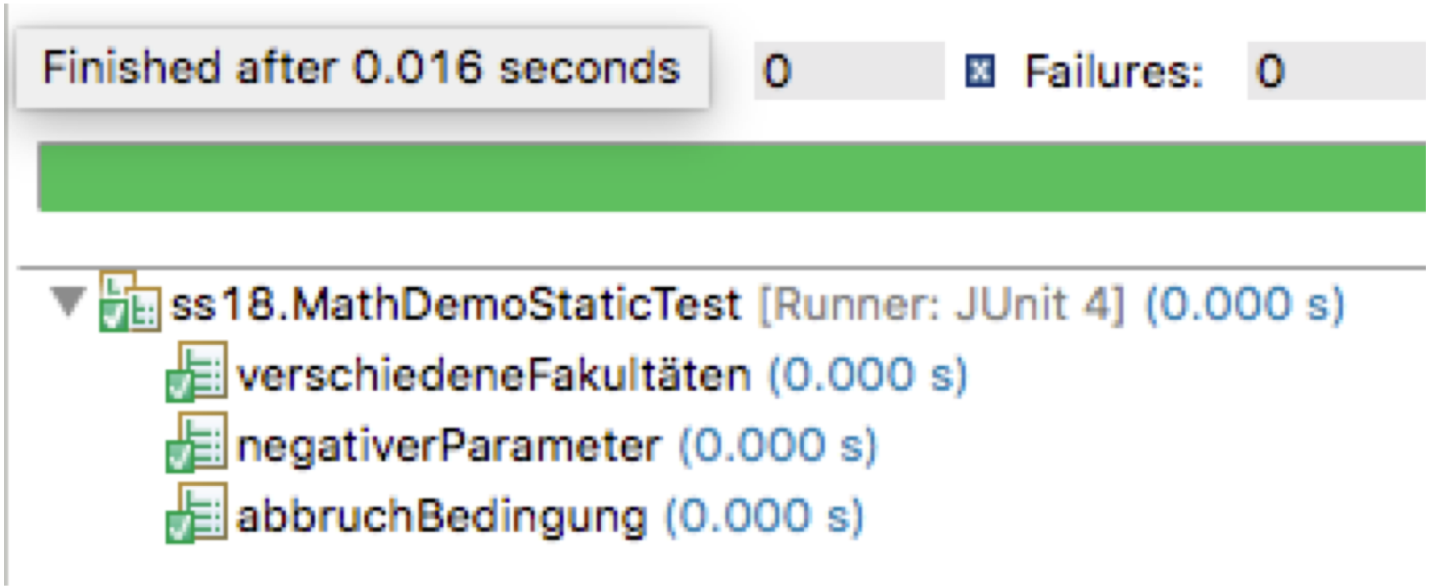
Verbesserte Methode mit Parameterprüfung

```
public class MathDemoStatic {

    public static int faculty(int n) {
        if (n < 0)
            throw new IllegalArgumentException("Parameter < 0: " + n);
        else if (n == 0)
            return 1;
        else
            return n * faculty(n - 1);
    }

}
```



Der Testfall schlägt immer noch fehl

Testmethode korrigiert



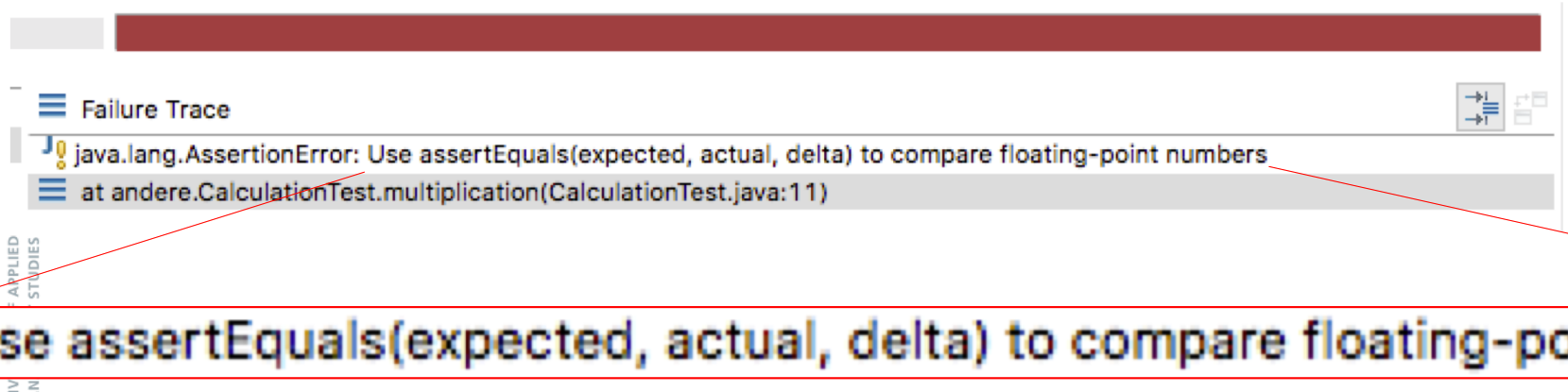

```
@Test(expected=IllegalArgumentException.class)
public void negativerParameter() throws Exception {
    MathDemoStatic.faculty(-1);
}
```

F R A G E N

Umgang mit Rechenfehlern

```
public static double multiply(double value1, double value2) {
    return value1 * value2;
}

@Test
public void multiplication() throws Exception {
    assertEquals(5.6, multiply(2.24, 2.5));
}
```

in Java:
5.600000000000005

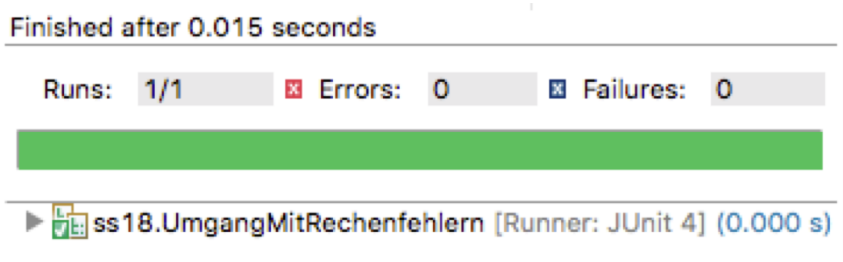


Use assertEquals(expected, actual, delta) to compare floating-point numbers

Umgang mit Rechenfehlern

```
public static double multiply(double value1, double value2) {
    return value1 * value2;
}

@Test
public void multiplication() throws Exception {
    assertEquals(5.6, multiply(2.24, 2.5));
}
```

in Java:
5.600000000000005




```
2
3  import static org.junit.Assert.*;
6
7  public class UmgangMitRechenfehlern {
8
9      @Test
10     public void multiplication() throws Exception {
11         assertEquals(5.6, multiply(2.24, 2.5), 0.0000001);
12     }
```

Weitere Prüfmethoden (importiert aus *Assert*)

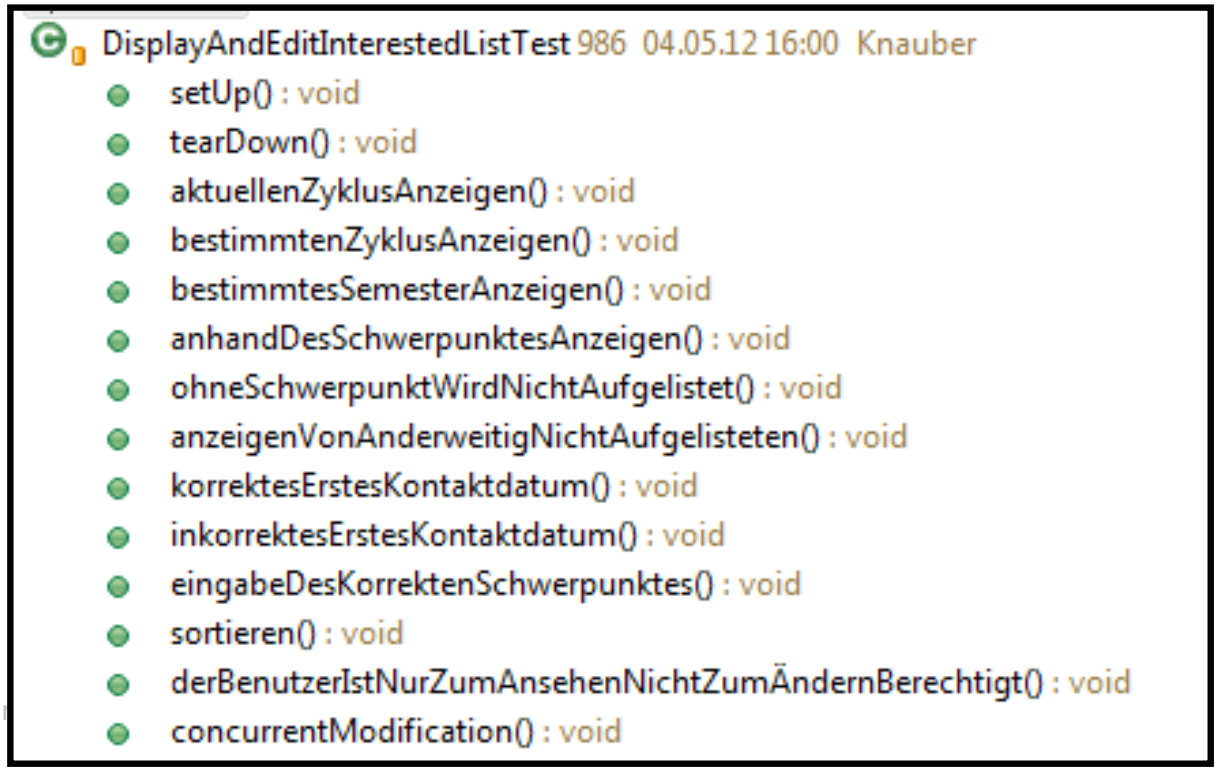
- Konzept:
  An bestimmten Stellen in einem (Test-)Programm soll sichergestellt (engl.: *assert*) werden, dass bestimmte Bedingungen erfüllt sind
- Die Klasse *Assert* bietet viele Prüf-Methoden
  - assertEquals(*Soll-Wert*, *Ist-Wert*); mögliche Typen:
    boolean, byte, char, int, long, short; Object, String (es wird mit *equals* verglichen)
  - assertEquals(*Soll-Wert*, *Ist-Wert*, *delta*); mögliche Typen:
    double, float
  - assertTrue(*Boole'sche Bedingung*)
  - assertFalse(*Boole'sche Bedingung*)
  - assertNull(*Objekt*)
  - assertNotNull(*Objekt*)
  - fail()
- Ist eine der Prüfbedingungen nicht erfüllt, wird ein *AssertionFailedError* ausgelöst
- Alle Methoden existieren auch mit einem zusätzlichen ersten Parameter (Typ String) für eine (hilfreiche!) Meldung

Gute Vorgehensweise

Zu jeder (Produktiv-)Klasse wird eine *Testklasse* mit *Testfällen (Testmethoden)* erstellt

Testfälle entwerfen

- Jede Test-Methode sollte nur einen Testfall prüfen, aber den komplett:
  - Einen geeigneten Zustand vor der zu testenden Methode herstellen
  - Die zu testende Methode aufrufen
  - Das Ergebnis (= Rückgabewert) der Methode überprüfen
  - Den Zustand nach dem Methodenaufruf überprüfen
- Jeder Testfall sollte unabhängig vom Vorgänger funktionieren können
- Die Test-Methoden sollten geeignet benannt sein
  → Fehler auf einen Blick erkennen können



F R A G E N