Java-Testprogramme mit JUnit schreiben

JUnit ist ein Framework, welches das Schreiben von Testprogrammen in Java unterstützt. Für andere Programmiersprachen gibt es entsprechende Frameworks (NUnit für C#, VisualBasic.net und andere .net-Sprachen, CppUnit für C++, PyUnit für Python etc.). Gemeinsam werden diese Programme als *Familie der* xUnit- *Frameworks* bezeichnet.

1. Bibliotheken und Frameworks

In Java versteht man unter einer Bibliothek (engl. library) eine Sammlung von Klassen, die typischerweise keine selbstständigen Programme sind (technisch: die keine `main`-Methode enthalten). Im Gegensatz dazu ist ein Framework typischerweise ein selbständiges Programm („mit `main`-Methode"), welches ein Benutzer durch das Hinzufügen bestimmter Klassen vervollständigen muss und dadurch an seine speziellen Bedürfnisse anpassen kann.

2. Das Framework JUnit

JUnit ist ein Framework, welches sogenannte *Testfälle* ausführt und prüft, ob sie gelungen oder misslungen sind. Ein Benutzer muss zu diesem Framework nur noch die Testfälle hinzufügen, um es an seine Bedürfnisse anzupassen. Technisch ist ein *Testfall* in JUnit eine *parameterlose Prozedur* (engl. void method with no parameters).

JUnit unterstützt vor allem das Erstellen von Komponenten-Tests (engl. unit tests), bei denen einzelne Methoden oder Klassen getestet werden. Für sogenannte Integrationstests oder Systemtests (bei denen ein vollständiges, umfangreiches Softwaresystem getestet wird) ist JUnit weniger geeignet. Die Grenze zwischen Komponenten-Tests und Systemstests ist aber keine scharfe schwarze Linie, sondern eher eine ziemlich breite Grauzone.

Technisch muss man zwischen JUnit Version 3.8 (für Java 1.4) und JUnit Version 4.x (für Java 5) unterscheiden. Die Version 4.0 wurde im Januar 2006 veröffentlicht, jetzt (Anfang 2009) ist die Version 4.8 aktuell. Die klassische Version 3.8 ist etwas einfacher als die neuen Versionen, funktioniert mit Java 1.4 und mit Java 5 und kann auch unabhängig von einer Java-Entwicklungsumgebung wie Eclipse oder Netbeans eingesetzt werden. Die neuen Versionen (4.0 und Nachfolger) bieten einige neue Befehle und Möglichkeiten, überlassen aber das Anzeigen der Testergebnisse der verwendeten Entwicklungsumgebung (z.B. Eclipse oder Netbeans) oder den sogenannten TestRunner-Programmen der klassischen Version 3.8.

Im Folgenden wird die klassische Version 3.8 von JUnit behandelt.

JUnit ist ein quelloffenes Entwicklungsprojekt (siehe `www.junit.org`). Die Hauptentwickler sind Erich Gamma und Kent Beck.

3. Installation

Von der Adresse `http://sourceforge.net/projects/junit/` kann man sich (nicht nur die neueren Versionen, sondern auch) die klassische Version 3.8.2 von JUnit als `.zip`-Datei herunterladen (ca. 0,5 MB, inklusive Quellcode und Dokumentation). Wenn man die Datei entpackt, entsteht ein Verzeichnis namens `junit3.8.2`, welches unter anderem eine Datei namens `junit.jar` enthält. Den Pfadnamen dieser `.jar`-Datei (z.B. `C:\meineDateien\junit3.8.2\junit.jar`) sollte man in die Umgebungsvariable `CLASSPATH` eintragen. Unter Windows XP: Start, Systemsteuerung, System, Reiter „Erweitert", Knopf „Umgebungsvariablen". Die Variable `CLASSPATH` sollte eine Folge von Pfadnamen enthalten, die durch Semikolons voneinander getrennt sind, z.B. so:

```
c:\meineDateien\junit3.8.2\junit.jar;d:\meineKlassen;d:\otto\seineKlassen
```

Jeder Pfadname sollte eine `.jar`-Datei bezeichnen oder ein Verzeichnis, welches `.class`-Dateien enthält.

4. Ein Beispielprogramm

Die folgende Klasse ist eine Ergänzung für das Framework JUnit (Version 3.8) und dient dazu, die Klasse `java.lang.StringBuilder` zu testen.

```
 1 import junit.framework.Test;      // Eine Schnittstelle
 2 import junit.framework.TestCase;  // Eine Test-Klasse
 3 import junit.framework.TestSuite; // Eine Test-Klasse
 4
 5 public class StringBuilderJut00 extends TestCase {
 6    // ---------------------------------------------------------------------
 7    static public Test suite() {
 8       return new TestSuite(StringBuilderJut00.class);
 9    } // suite
10    // ---------------------------------------------------------------------
11    public void testKonstruktor() {
12       // Initialisiert der Konstruktor mit String-Parameter korrekt?
13       StringBuilder sb = new StringBuilder("ABC");
14       assertEquals("ABC", sb.toString());
15    } // testLeerSB
16    // ---------------------------------------------------------------------
17    public void testAppendString() {
18       // Funktioniert das Anhaengen eines String-Objekts mit append?
19       StringBuilder sb = new StringBuilder();
20       sb.append("Hallo!");
21       assertEquals(6, sb.length());
22       assertEquals("Hallo!", sb.toString());
23    } // testAppendString
24    // ---------------------------------------------------------------------
25    public void testAppendInt() {
26       // Funktioniert das Anhaengen eines int-Wertes mit append?
27       StringBuilder sb = new StringBuilder();
28       sb.append(+123);
29       assertEquals(   3 , sb.length());
30       assertEquals("123", sb.toString());
31    } // probiereAppendInt
32    // ---------------------------------------------------------------------
33 } // class StringBuilderJut00
```

Der Name einer solchen Testklasse (im Beispiel: `StringBuilderJut00`) ist frei wählbar, aber die Klasse muss als Erweiterung der Klasse `junit.framework.TestCase` vereinbart werden (siehe Zeilen 2 und 5).

Die Klasse `StringBuilderJut00` enthält drei Testfälle. Damit sind die Methoden `testKonstruktor`, `testAppendString` und `testAppendInt` gemeint. Wenn der Name eines Testfalls mit `"test"` beginnt, ist „seine Verwaltung“ besonders einfach. Andere Namen sind möglich, aber nicht empfehlenswert. Auf jeden Fall ist ein Testfall eine *Methode*, genauer: Eine öffentliche, parameterlose Objektprozedur (engl. a public non-static void-method with no parameters).

Eine Testklasse wie `StringBuilderJut00` muss eine Klassenmethode namens `suite` enthalten. Die muss ein `TestSuite`-Objekt liefern, welches die Namen aller Testfälle enthält. Solange diese Namen alle mit `"test"` beginnen, kann man die einfache `suite`-Method aus den Zeilen 7 bis 9 übernehmen und muss nur den Klassennamen (im Beispiel: `StringBuilderJut00`) anpassen (ein anderes Beispiel findet man im Abschnitt 7). Die `suite`-Methode muss eine öffentliche, parameterlose Klassenfunktion sein (engl. a public static non-void-method with no parameters).

Ein Testfall besteht aus vorbereitenden Befehlen (das können beliebige Java-Befehle sein) und aus `assert`-Befehlen. Im Beispiel enthält der Testfall `testKonstruktor` *einen* `assert`-Befehl und die anderen beiden Testfälle enthalten je *zwei*.

Ein `assert`-Befehl kann gelingen oder misslingen. Ein `assertEquals`-Befehl gelingt, wenn seine beiden Parameter gleich sind. Ein `assertTrue`-Befehl gelingt, wenn sein Parameter gleich `true` ist, ein `assertNotNull`-Befehl, wenn sein Parameter ungleich `null` ist etc. (siehe den Abschnitt 6 über `assert`-Befehle).

Wenn ein `assert`-Befehl misslingt, wirft er eine `assert`-Ausnahme (d.h. eine Ausnahme des Typs `junit.framework.AssertionFailedError`).

Ein Testfall kann (wie jede Methode) *normal beendet* oder auf Grund einer Ausnahme *abgebrochen* werden. Die folgenden drei Bezeichnungen sind nützlich, wenn man intensiv mit Testfällen umgeht:

Ein Testfall T ist gelungen bedeutet: T wurde normal beendet.

T misslang mit einem assert-Fehler bedeutet: T wurde mit einer `assert`-Ausnahme abgebrochen.

T misslang mit einem unerwarteten Fehler bedeutet: T wurde mit einer anderen (nicht-assert-) Ausnahme abgebrochen.

Im Englischen wird ein `assert`-Fehler als `failure` und ein unerwarteter Fehler als `error` bezeichnet.

Die folgende Daumenregel soll deutlich machen, warum man zwischen `assert`-Fehlern und unerwarteten Fehlern unterscheidet:

Daumenregel: Misslingt ein Testfall mit einem `unerwarteten Fehler` (engl. error), so muss man typischerweise den Testfall (oder eine andere Stelle der Testklasse) korrigieren. Misslingt ein Testfall mit einem assert-Fehler (engl. failure), muss man typischerweise die getestete Methode oder Klasse korrigieren.

Beispiel-01: Angenommen, beim Ausführen der beiden Zeilen

```
13    StringBuilder sb = new StringBuilder("ABC");
14    assertEquals("ABC", sb.toString());
```

im Testfall `testKonstruktor` misslingt der `assert`-Befehl (weil seine beiden Parameter ungleich sind). Dann misslingt der ganze Testfall mit einem `assert`-Fehler. In diesem Fall muss man die getestete Klasse `StringBuilder` prüfen und eventuell korrigieren. Vermutlich enthält (in diesem fiktiven Fall) der in Zeile 13 aufgerufene Konstruktor oder die in Zeile 14 aufgerufene Methode `toString` einen Fehler). Sinn und Zweck einer Testklasse wie `StringbuilderJut00` ist es, solche `assert`-Fehler zu entdecken (bzw. zu zeigen, dass sie abwesend sind).

Beispiel-02: Angenommen, wir hätten im obigen Testfall `testKonstruktor` aus Versehen

```
13    StringBuilder sb = null;
14    assertEquals("ABC", sb.toString());
```

geschrieben. Bevor in Zeile 14 die Methode `assertEquals` aufgerufen werden kann, muss der Wert des Parameterausdrucks `sb.toString()` berechnet werden. Dabei tritt eine `NullPointerException` auf. Deshalb misslingt der Testfall `testKonstruktor` mit einem unerwarteten Fehler (der `assert`-Befehl konnte seinen Vergleich ja noch nicht einmal beginnen). In diesem Fall muss die Zeile 13 der Testklasse (und nicht die getestete Klasse StringBuilder) korrigiert werden.

Es ist aber ziemlich einfach, Fälle zu konstruieren, in denen die obige Daumenregel *versagt*, wie etwa im folgenden Beispiel.

Beispiel-03: Angenommen, wir hätten im Testfall `testKonstruktor` aus Versehen

```
13    StringBuilder sb = new StringBuilder("XYZ");
14    assertEquals("ABC", sb.toString());
```

geschrieben. Dann würde der Testfall mit einem `assert`-Fehler abgebrochen, aber offenbar müsste man trotzdem den *Testfall* korrigieren, und nicht die getestete Klasse `StringBuilder`.

Allgemein gilt: Wenn ein Testprogramm einen Fehler feststellt, dann kann der entweder im *getesteten Programm* oder aber im *Testprogramm* liegen.

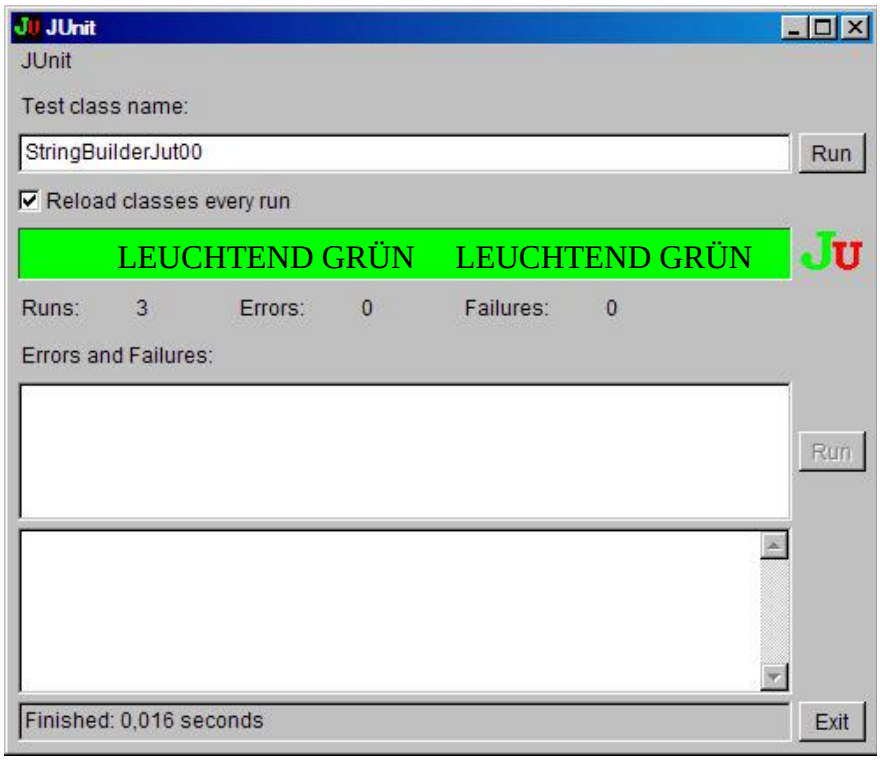
Vergleich: Indem man zwei rauhe Steinplatten aneinander reibt kann man erreichen, dass beide glatter werden. Wenn man ein Programm P mit einem Testprogramm TP testet, kann man in beiden Programmen Fehler entdecken und beseitigen („bis P und TP beide glatt sind“).

Einfache Testprogramme: Indem man Testprogramme besonders einfach und übersichtlich strukturiert (viel einfacher und übersichtlicher als die zu testenden Anwendungsprogramme), kann man sie besonders schnell „weitgehend fehlerfrei“ bekommen. Ab dann werden (fast) alle entdeckten Fehler im zu testenden Programm liegen.

Alle Testfälle in der Klasse `StringBuilderJut00` kann man *ausführen* lassen, indem man in einer Kommandozeile (z.B. in einem `DOS`- oder einem `bash`-Fenster) folgendes Kommando eingibt:

```
> java junit.awtui.TestRunner StringBuilderJut00
```

Nach kurzer Zeit sollte dann ein Fenster wie das nebenstehende aufgehen.



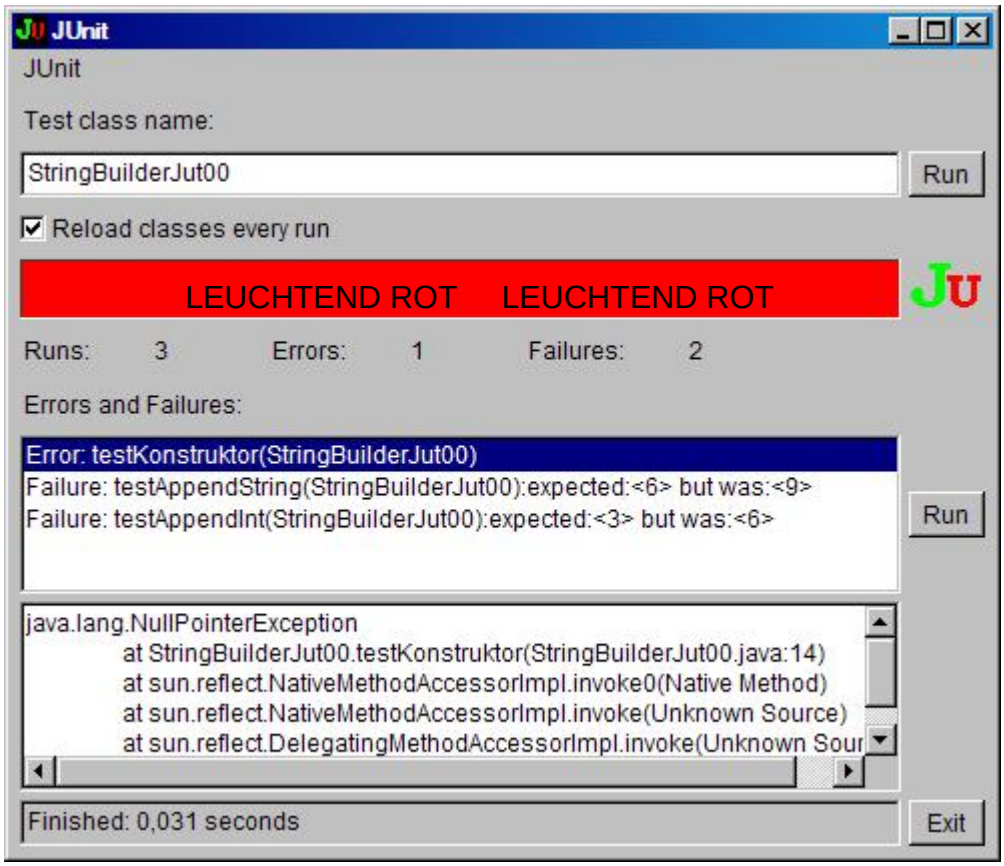
Falls Sie eine schwarz-weiss-Version dieses Papiers vor sich haben: Der hier zweimal mit „LEUCHTEND GRÜN“ beschriftete Balken ist auf dem Bildschirm nicht beschriftet, aber leuchtend grün. Dieser Balken (und der Text darunter) drücken aus, dass der durchgeführte Test vollständig gelungen ist (keiner der 3 Testfälle ist misslungen) und dass seine Durchführung insgesamt 0,016 Sekunden gedauert hat. Ein Test, bei dem der Programmierer irgendwelche Programmausgaben lesen und prüfen muss, dauert in aller Regel deutlich länger :-).

Unsere Testklasse hatte kaum eine Chance, in einer so gründlich ausgetesteten Standardklasse wie `StringBuilder` noch einen Fehler zu entdecken. Um aber zu zeigen, wie in JUnit das Misslingen von Testfällen angezeigt wird, sollen jetzt drei Fehler in die *Testklasse* eingebaut werden, und zwar in alle drei Zeilen, in denen ein `StringBuilder`-Objekt erzeugt wird:

```
...    ...
 13    StringBuilder sb = null;
...    ...
 19    StringBuilder sb = new StringBuilder("ABC");
...    ...
 27    StringBuilder sb = new StringBuilder("ABC");
...    ...
```

Wenn wir dann die fehlerhafte Testklasse `StringBuilderJut00` compiliert haben und erneut das Kommando

`> java junit.awtui.TestRunner StringBuilderJut00`



in einem `DOS`- oder `bash`-Fenster eingeben, sollte ein Fenster mit einem lechtend *roten* Balken (anstelle des leuchtend *grünen* Balkens) erscheinen, etwa so wie hier abgebildet.

Der rote Balken macht deutlich (oder: macht es schwer zu übersehen), dass irgendetwas nicht stimmt. Dem Text unter dem farbigen Balken kann man entnehmen, dass insgesamt 3 Testfälle ausgeführt wurden (`Runs: 3`), dass davon einer mit einem unerwarten Fehler misslungen ist (`Errors: 1`) und dass zwei mit einem `assert`-Fehler misslungen sind (`Failures: 2`).

Im mittleren Teilfenster darunter wird jeder unerwartete Fehler (Error) und jeder `assert`-Fehler (Failure) durch eine einzeilige Meldung beschrieben.

Der in diesem mittleren Teilfenster ausgewählte Fehler wird im untersten Teilfenster dann noch genauer beschrieben: Durch eine Wiederholung der Meldung aus dem mittleren Fenster und einen vollständigen Stack-Trace. An dem Stack-Trace ist häufig vor allem die Zeilen-Nr am Ende der ersten Zeile (im Beispiel: `:14`) besonders interessant, weil sie meist auf eine bestimmte Zeile in der Testklasse (im Beispiel: `StringBuilderJut00`) hinweist.

Man beachte, dass in jedem Testfall höchstens *ein* Fehler auftreten kann, weil jeder Fehler den sofortigen Abbruch des Testfalls bewirkt. Das Auftreten eines Fehlers in *einem* Testfall verhindert aber nicht die Ausführung der *nachfolgenden Testfälle*.

5. JUnit-Testklassen und die zu testenden Klassen

Im vorigen Abschnitt wurde als Beispiel die JUnit-Testklasse `StringBuilderJut00` behandelt. Sie diente dazu, bestimmte Teile der Klasse `StringBuilder` zu testen. In diesem ersten Beispiel gab es also genau *eine* Testklasse und *eine* zu testende Klasse.

Das Framework JUnit läßt einem aber volle Freiheit bei der Entscheidung, *wieviele* Klassen man in *einer* Testklasse testen will oder mit *wieviel* Testklassen man *eine* zu testende Klasse testet. So kann man z.B. in *einer* Testklasse *mehrere* Klassen testen (z.B. weil diese Klassen „eng zusammengehören“ und nur zusammen sinnvoll getestet werden können), oder man kann für *eine* komplizierte Klasse KK *mehrere* Testklassen schreiben, die nur KK testen, etc.

6. assert-Befehle

In JUnint (Version 3.8) gibt es 32 `assert`-Methoden (vereinbart in der Klasse `junit.framework.Assert`). Die folgende Tabelle soll eine Übersicht geben und die „Systematik“ dieser Methoden erkennbar machen:

```
static void assertEquals (             boolean soll, boolean ist)
static void assertEquals (             byte    soll, byte    ist)
static void assertEquals (             char    soll, char    ist)
static void assertEquals (             double  soll, double  ist, double delta)
static void assertEquals (             float   soll, float   ist, float  delta)
static void assertEquals (             int     soll, int     ist)
static void assertEquals (             long    soll, long    ist)
static void assertEquals (             short   soll, short   ist)
static void assertEquals (             String  soll, String  ist)
static void assertEquals (             Object  soll, Object  ist)

static void assertSame   (             Object  soll, Object  ist)
static void assertNotSame(             Object  soll, Object  ist)
static void assertFalse  (                           boolean ist)
static void assertTrue   (                           boolean ist)
static void assertNull   (                           Object  ist)
static void assertNotNull(                           Object  ist)

static void assertEquals (String meld, boolean soll, boolean ist)
static void assertEquals (String meld, byte    soll, byte    ist)
static void assertEquals (String meld, char    soll, char    ist)
static void assertEquals (String meld, double  soll, double  ist, double delta)
static void assertEquals (String meld, float   soll, float   ist, float  delta)
static void assertEquals (String meld, int     soll, int     ist)
static void assertEquals (String meld, long    soll, long    ist)
static void assertEquals (String meld, short   soll, short   ist)
static void assertEquals (String meld, String  soll, String  ist)
static void assertEquals (String meld, Object  soll, Object  ist)

static void assertSame   (String meld, Object  soll, Object  ist)
static void assertNotSame(String meld, Object  soll, Object  ist)
static void assertFalse  (String meld,               boolean ist)
static void assertTrue   (String meld,               boolean ist)
static void assertNull   (String meld,               Object  ist)
static void assertNotNull(String meld,               Object  ist)
```

Die unteren 16 Methoden entsprechen „eins zu eins“ den oberen 16 Methoden, haben aber vorn einen zusätzlichen `String`-Parameter. Damit kann man eine Meldung angeben, die im Falle eines `assert`-Fehlers (zusammen mit dem Namen des Testfalls) ausgegeben wird. Wenn man die Namen der Testfälle sorgfältig wählt, kann man häufig auf eine solche zusätzliche Meldung verzichten.

6.1. assertEquals und die Reihenfolge der Parameter

Mit den 20 Methoden namens `assertEquals` kann man prüfen lassen, ob zwei Parameter eines beliebigen Typs *gleich* sind. Falls die zwei Parameter zu einem primitven Typ gehören, werden sie mit dem Operator == verglichen: `soll == ist`. Falls sie aber zu einem Referenztyp gehören, werden sie mit der Methode `equals` verglichen: `soll.equals(ist)`.

In Fehlermeldungen des JUnit-Rahmensystems wie z.B.

`expected:<"ABC"> but was:<"XYZ">`

wird davon ausgegangen, dass man in einem `assertEquals`-Befehl *zuerst* den erwarteten Soll-Wert und *danach* den tatsächlich vom Testling gelieferten Ist-Wert angibt:

```
assertEquals("ABC", sb.toString()); // Richtige Reihenfolge von Soll und Ist
assertEquals(sb.toString(), "ABC"); // Unglückliche Reihenfolge
```

6.2. assertEquals und assertSame, Gleichheit und Identität

Mit den 2 Methoden namens `assertSame` kann man prüfen lassen, ob zwei Parameter eines Referenztyps *identisch* sind. Dazu werden die beiden Parameter mit dem Operator `==` verglichen.

Beispiel-01: Gleichheit und Identität

```
1 String s1 = new String("Hallo");
2 String s2 = new String("Hallo");
3 String s3 = s2;
4
5 assertEquals(s1, s2); // Gelingt,   s1 und s2 sind       gleich
6 assertSame  (s1, s2); // Misslingt, s1 und s2 sind nicht identisch
7 assertEquals(s2, s3); // Gelingt,   s2 und s3 sind       gleich
8 assertSame  (s2, s3); // Gelingt,   s2 und s3 sind       identisch
```

In den Zeilen 1 bis 3 werden mit `new` zwei `String`-Objekte mit unterschiedlichen Referenzen, aber gleichen Zielwerten erzeugt (beide repräsentieren den String `"Hallo"`). Die Variablen `s1` und `s2` haben unterschiedliche (Referenz-) Werte, zeigen aber auf zwei gleiche Zielwerte. Die beiden Variablen `s2` und `s3` haben gleiche (Referenz-) Werte und zeigen damit auf dasselbe `String`-Objekt.

6.3. Gleitpunktzahlen tolerant vergleichen

Gleitpunktzahlen der Typen `float` und `double` sind in aller Regel mit einem (mehr oder weniger großen) Fehler behaftet. Deshalb ist es selten sinnvoll, sie auf „exakte Gleichheit“ zu prüfen. Häufig sollte man zwei solche Zahlen nur dann als *ungleich* betrachten, wenn sie sich um mehr als eine bestimmte Zahl `delta` unterscheiden. Diese „maxiamle tolerierbare Differenz“ kann und muss man bei den `assertEquals`-Methoden für `float`- und `double`-Zahlen als `delta`-Parameter angeben.

Beispiel-02: Gleichheit trotz kleiner Unterschiede

```
 9 assertEquals(10.0, 11.0,   1.0);   // Gelingt   |10.0 - 11.0  | <= 1.0
10 assertEquals(10.0, 11.001, 1.0);   // Misslingt |10.0 - 11.001| >  1.0
11 assertEquals(10.0, 10.1,   0.1);   // Gelingt   |10.0 - 10.1  | <= 0.1
12 assertEquals(10.0,  9.8,   0.1);   // Misslingt |10.0 -  9.8  | >  0.1
```

Der Befehl in Zeile 9 gelingt, weil der Betrag der Differenz `10.0 - 11.0` nicht größer ist als das angegebene `delta` von `1.0`.

Der Befehl in Zeile 10 misslingt, weil der Betrag der Differenz `10.0 - 11.001` größer ist als das angegebene `delta` von `1.0`).

Man beachte, dass der `delta`-Parameter immer eine *absolute* (noch tolerierbare) *Differenz* angibt. Wenn man z.B. mit Millionen (kg) rechnet, ist eine absolute Differenz von `1.0` häufig tolerierbar.

Rechnet man dagegen mit Millionsteln (z.B. Gramm Dioxin), ist eine absolute Differenz von `1.0` häufig nicht tolerierbar.

Muss man gleichzeitig mit großen Zahlen (z.B. mit Millionen) und kleinen Zahlen (z.B. Millionsteln) rechnen, kann man häufig keine *absolute* Differenz, sondern nur eine *relative* Differenz tolerieren: Zwei Zahlen müssen dann als verschieden betrachtet werden, wenn sie sich z.B. um mehr als 3 Prozent oder um mehr als 0,05 Promille unterscheiden. Wenn man Gleitpunktzahlen mit einer solchen relativen Toleranz vergleichen will, muss man selbst eine entsprechende `assert`-Methode programmieren. Wenn diese Methode eine zu große Differenz feststellt, sollte sie eine der folgenden beiden `fail`-Methoden aufrufen (und dadurch eine `assert`-Ausnahme auslösen):

```
13 static void fail()
14 static void fail(String meld)
```

6.4. Bequeme assert-Methoden

Wer will, kann alle `assert`-Befehle als Aufrufe der `assertEquals`-Methoden formulieren. Es ist sogar möglich, sich auf Aufrufe der Methode `assertTrue` zu beschränken. Die anderen `assert`-Methoden sind aber häufig „bequeme Abkürzungen“, die das Schreiben und Lesen von Testklassen etwas erleichtern.

Die 2 Methoden namens `assertTrue` gelingen, wenn ihr ist-Parameter gleich `true` ist.
Die 2 Methoden namens `assertFalse` gelingen, wenn ihr ist-Parameter gleich `false` ist.

Die 2 Methoden namens `assertNull` gelingen, wenn ihr ist-Parameter gleich `null` ist.
Die 2 Methoden namens `assertNotNull` gelingen, wenn ihr ist-Parameter ungleich `null` ist.

Die 2 Methoden namens `assertNotSame` gelingen, wenn ihr ist- und ihr soll-Parameter nicht identisch sind.

6.5. Zugriff auf assert-Methoden und Klassenhierarchie

Wieso kann man in einer Klasse wie `StringBuilderJut00` „einfach so“ auf die 32 `assert`-Methoden und die beiden `fail`-Methoden zugreifen, die in der Klasse `Assert` definiert sind? Der folgende *Vererbungsbaum* enthält die Antwort:

```
java.lang.Object
   |
   +--- junit.framework.Assert
           |
           +--- junit.framework.TestCase
                   |
                   +--- StringBuilderJut00
```

7. Noch ein Beispielprogramm

Das folgende Beispielprogramm demonstriert einige zusätzliche Möglichkeiten, die JUnit bietet.

```
 1 import junit.framework.Test;      // Eine Schnittstelle
 2 import junit.framework.TestCase;  // Eine Test-Klasse
 3 import junit.framework.TestSuite; // Eine Test-Klasse
 4
 5 public class StringBuilderJut01 extends TestCase {
 6    // ---------------------------------------------------------------------
 7    static public Test suite() {
 8       // Alle mit "test" beginnenden Namen von Testfaellen kann man
 9       // (pauschal) wie folgt in ein TestSuite-Objekt einfuegen:
```

```
10        TestSuite ts1 = new TestSuite(StringBuilderJut01.class);
11
12        // Beliebige Namen von Testfaellen kann man einzeln wie folgt
13        // in ein TestSuite-Objekt einfuegen:
14        ts1.addTest(new StringBuilderJut01("pruefeAppendInt"));
15        // Diese zweite Vorgehensweise setzt einen StringBuilderJut01-
16        // Konstruktor mit einem String-Parameter voraus (siehe unten).
17        return ts1;
18     } // suite
19     // ------------------------------------------------------------------
20     public StringBuilderJut01(String name) {super(name);}
21     // ------------------------------------------------------------------
22     // Eine globale Variable, die von allen Testfaellen benutzt werden kann:
23     StringBuilder sb;
24
25     // Die Methode setUp wird (vom JUnit-Framework) vor jedem
26     // einzelnen Testfall aufgerufen:
27     public void setUp() {
28        sb = new StringBuilder();
29        pln("setUp    wurde aufgerufen!");
30     } // setUp
31
32     // Die Methode tearDown wird (vom JUnit-Framework) nach jedem
33     // einzelnen Testfall aufgerufen:
34     public void tearDown() {
35        pln("tearDown wurde aufgerufen!");
36     }
37     // ------------------------------------------------------------------
38     public void testKonstruktor() {
39        // Initialisiert der Konstruktor mit String-Parameter korrekt?
40        sb = new StringBuilder("ABC");
41        assertEquals("ABC", sb.toString());
42     } // testLeerSB
43     // ------------------------------------------------------------------
44     public void testAppendString() {
45        // Funktioniert das Anhaengen eines String-Objekts mit append?
46        sb.append("Hallo!");
47        assertEquals(6, sb.length());
48        assertEquals("Hallo!", sb.toString());
49     } // testAppendString
50     // ------------------------------------------------------------------
51     public void pruefeAppendInt() {
52        // Funktioniert das Anhaengen eines int-Wertes mit append?
53        sb.append(+123);
54        assertEquals(   3 , sb.length());
55        assertEquals("123", sb.toString());
56     } // probiereAppendInt
57     // ------------------------------------------------------------------
58     static public void main(String[] _) {
59        // Die 3 Testfaelle in suite() ausfuehren lassen:
60        junit.awtui.TestRunner.run(StringBuilderJut01.class);
61     } // main
62     // ------------------------------------------------------------------
63     // Eine Methode mit einem kurzen Namen:
64     static void pln(Object ob) {System.out.println(ob);}
65     // ------------------------------------------------------------------
66  } // class StringBuilderJut01
```

Kurze Erläuterung der hier benutzten zusätzlichen Möglichkeiten:

1. In der Methode `suite` wird der Testfall-Name `pruefeAppendInt`, der nicht mit `"test"` beginnt, in ein `TestSuite`-Objekt eingefügt (Zeile 14).

2. Dazu ist ein Konstruktor mit einem `String`-Parameter erforderlich. Ein solcher Konstruktor wird in Zeile 14 aufgerufen und in Zeile 20 vereinbart.

3. In den Zeilen 27 bis 30 wird eine (parameterlose Objekt-) Prozedur namens `setUp` vereinbart. Diese Methode wird vom Framework JUnit vor jedem einzelnen Testfall ausgeführt.

4. In den Zeilen 34 bis 36 wird eine (parameterlose Objekt-) Prozedur namens `tearDown` vereinbart. Diese Methode wird vom Framework JUnit nach jedem einzelnen Testfall ausgeführt.

5. Die Testklasse enthält jetzt eine `main`-Methode (Zeile 58 bis 61). Somit kann man sie auf zwei Weisen ausführen lassen:
5.1. Indem man das Programm `StringBuilderJut01` startet, etwa so:

```
> java StringBuilderJut01
```

5.2. Indem man (wie bisher) in einem `DOS`- oder `bash`-Fenster das folgende Kommando eingibt:

```
> java junit.awtui.TestRunner StringBuilderJut00
```

8. Testergebnisse als Textzeilen ausgeben

Die Ergebnisse eines JUnit-Tests kann man nicht nur *graphisch anzeigen*, sondern auch als eine Folge von *Textzeilen* ausgeben lassen. Wenn man die fehlerhafte Version der Klasse `StringBuilder-Jut00` verwendet und in einem `DOS`- oder `bash`-Fenster das folgende Kommando eingibt:

```
> java junit.textui.TestRunner StringBuilderJut00
```

erhält man folgende Ausgabe:

```
.E.F.F
Time: 0
There was 1 error:
1) testKonstruktor(StringBuilderJut00)java.lang.NullPointerException
        at StringBuilderJut00.testKonstruktor(StringBuilderJut00.java:14)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
There were 2 failures:
1) testAppendString(StringBuilderJut00)junit.framework.AssertionFailedError:
expected:<6> but was:<9>
        at StringBuilderJut00.testAppendString(StringBuilderJut00.java:21)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
2) testAppendInt(StringBuilderJut00)junit.framework.AssertionFailedError:
expected:<3> but was:<6>
        at StringBuilderJut00.testAppendInt(StringBuilderJut00.java:29)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)

FAILURES!!!
Tests run: 3,  Failures: 2,  Errors: 1
```

Diese Zeilen kann man relativ einfach von einem anderen Werkzeug weiterverarbeiten lassen (was mit der graphischen Ausgabe kaum möglich ist).