Eine JUnit-Klasse zum Testen der Klasse StringBuilder

```
 1 // Datei StringBuilderJut.java
 2 /* ---------------------------------------------------------------------
 3 JUnit ist ein Rahmenprogramm zum Erstellen und Ausfuehren von Testprogrammen.
 4 Die Klasse StringBuilderJut wurde mit diesem Rahmenprogramm erstellt und
 5 dient zum Testen der Standardklasse StringBuilder. Wichtige Regeln fuer
 6 das Erstellen einer JUnit-Klasse (JUK) wie StringBuilderJut (wenn man
 7 JUnit Version 3.8 verwendet, fuer die Versionen ab 4.0 gelten etwas andere
 8 Regeln):
 9
10 1. Der Name einer JUK ist frei waehlbar (hier: StringBuilderJut).
11 2. Eine JUK muss die Klasse junit.framework.TestCase erweitern.
12 3. Eine JUK muss eine parameterlose Klassenprozedur namens suite enthalten,
13    die ein Test-Objekt liefert (junit.framework.Test ist eine Schnittstelle,
14    die Klasse junit.framework.TestSuite implementiert diese Schnittstelle).
15 4. Die einzelnen Testfaelle (test cases) muessen als parameterlose
16    Objektprozeduren ("void-Methoden") einer JUK-Klasse realisert werden.
17 5. Die Namen solcher Testfaelle sollten mit "test" beginnen (dann ist
18    alles ganz einfach), koennen aber auch anders lauten (das macht bei 6.
19    ein bisschen mehr Arbeit).
20 6. Die Methode suite muss die Namen aller Testfaelle (im Beispiel sind das
21    die Namen testLeerSB, testAppendString und probiereAppendInt) zu einem
22    TestSuite-Objekt zusammenfassen und dieses Objekt als Ergebnis liefern.
23    Die Methode suite in dieser Klasse zeigt dafuer zwei Vorgehensweisen.
24
25 Diese JUnit-Klasse enthaelt 4 Testfaelle (namens testLeerSB, testAppend
26 probiereAppendInt und assertSyntax). Der Testfall assertSyntax soll nur
27 die Syntax von assert-Befehlen verdeutlichen, fuehrt aber keine sinnvollen
28 Tests durch.
29
30 Die Klasse StringBuilderJut (und damit ihre 4 Testfaelle) kann man auf
31 zwei Arten ausfuehren lassen:
32
33 1. Indem man das Programm StringBuilderJut aufruft (siehe die main-Methode).
34 2. Indem man (in einem bash- oder DOS-Fenster) das folgende Kommando eingibt:
35
36 > java junit.awtui.TestRunner StringBuilderJut
37
38 Dieses Programm setzt Junit Version 3.8 voraus und laeuft NICHT mit
39 Junit Version 4.0 oder spaeteren Versionen.
40 ------------------------------------------------------------------------ */
41 import junit.framework.Test;      // Eine Schnittstelle
42 import junit.framework.TestCase;  // Eine Test-Klasse
43 import junit.framework.TestSuite; // Eine Test-Klasse
44
45 public class StringBuilderJut extends TestCase {    // siehe oben 1. und 2.
46    // ---------------------------------------------------------------------
47    static public Test suite() {                                  // s.o. 3.
48       // Ein TestSuite-Objekt ist im Wesentlichen eine Zusammenfassung
49       // von Namen von Testfaellen. Ein Testfall ist eine parameterlose
50       // Objektprozeudr (non-static void-method with no parameters).
51
52       // Alle mit "test" beginnenden Namen von Testfaellen kann man
53       // (pauschal) wie folgt in ein TestSuite-Objekt einfuegen:
54       TestSuite ts1 = new TestSuite(StringBuilderJut.class);     // s.o. 5.
55
56       // Beliebige Namen von Testfaellen kann man einzeln wie folgt
57       // in ein TestSuite-Objekt einfuegen:
58       ts1.addTest(new StringBuilderJut("probiereAppendInt"));    // s.o. 5.
59       ts1.addTest(new StringBuilderJut("assertSyntax"));
60       // Diese zweite Vorgehensweise setzt einen StringBuilderJut-Konstruk-
```

```
61       // tor mit einem String-Parameter voraus (siehe unten).
62       return ts1;                                                  // s.o. 6.
63    } // suite
64    // ------------------------------------------------------------------------
65    // Ein StringBuilderJut-Konstruktor mit einem String-Parameter:
66    public StringBuilderJut(String name) {super(name);}
67    // ------------------------------------------------------------------------
68    // Eine globale Variable, die von allen Testfaellen benutzt werden kann:
69    StringBuilder sb;
70
71    // Die Methode setUp wird (vom JUnit-Rahmenprogramm) vor jedem
72    // einzelnen Testfall aufgerufen:
73    public void setUp() {
74       sb = new StringBuilder();
75       pln("setUp    wurde aufgerufen!");
76    } // setUp
77
78    // Die Methode tearDown wird (vom JUnit-Rahmenprogramm) nach jedem
79    // einzelnen Testfall aufgerufen:
80    public void tearDown() {
81       pln("tearDown wurde aufgerufen!");
82    }
83    // ------------------------------------------------------------------------
84    public void testLeerSB() {                        // 1. Testfall, s.o. 4.
85       // Ist sb wirklich leer?
86       assertEquals("", sb.toString());
87       assertEquals( 0, sb.length());
88    } // testLeerSB
89    // ------------------------------------------------------------------------
90    public void testAppendString() {                  // 2. Testfall, s.o. 4.
91       // Funktioniert das Anhaengen eines String-Objekts mit append?
92       sb.append("Hallo!");
93       assertEquals("Hallo!", sb.toString());
94       assertEquals(6, sb.length());
95    } // testAppendString
96    // ------------------------------------------------------------------------
97    public void probiereAppendInt() {                 // 3. Testfall, s.o. 4.
98       // Funktioniert das Anhaengen eines int-Wertes mit append?
99       sb.append(123);
100      assertEquals("Fehler bei sb.append(123)", "123", sb.toString());
101      assertEquals("Fehler bei sb.append(123)",    3 , sb.length());
102   } // probiereAppendInt
103   // ------------------------------------------------------------------------
104   // Der folgende Testfall soll die Syntax (und Semantik) von assert-
105   // Befehlen verdeutlichen, fuehrt aber keine sinnvollen Tests durch.
106
107   public void assertSyntax() {                      // 4. Testfall, s.o. 4.
108      String s1 = new String("Hallo");    // s1 und s2 sind gleich, aber
109      String s2 = new String("Hallo");    // nicht identisch
110      int    i1 = 17;                     // i1 und i2 sind gleich und
111      int    i2 = 17;                     // identisch (primitver Typ int)
112      double d1 = 17.0;                   // i1 und d1 sind ungleich
113      Object o1 = null;
114
115      assertSame   ("AB", "AB");          // Gelingt
116      assertSame   (s1, s1);              // Gelingt
117 //   assertSame   (s1, s2);              // Misslingt
118      assertEquals (s1, s2);              // Gelingt
119
120      assertSame   (17, 17);              // Gelingt
121      assertSame   (i1, i2);              // Gelingt
122      assertSame   (i1, 17);              // Gelingt
123      assertEquals (17, 17);              // Gelingt
```

```
124       assertEquals (i1, i2);             // Gelingt
125       assertEquals (i1, 17);             // Gelingt
126 //    assertEquals (i1, d1);             // Misslingt
127
128       assertTrue   (true);               // Gelingt
129 //    assertTrue   (false);              // Misslingt
130 //    assertFalse  (true);               // Misslingt
131       assertFalse  (false);              // Gelingt
132
133       assertNull   (o1);                 // Gelingt
134 //    assertNull   (s1);                 // Misslingt
135
136       // Vergleich von Gleitpunktzahlen mit Toleranz:
137       assertEquals (10.0, 11.0,   1.0);   // Gelingt   |10.0 - 11.0  | <= 1.0
138 //    assertEquals (10.0, 11.001, 1.0);   // Misslingt |10.0 - 11.001| >  1.0
139       assertEquals (10.0, 10.1,   0.1);   // Gelingt   |10.0 - 10.1  | <= 0.1
140 //    assertEquals (10.0,  9.8,   0.1);   // Misslingt |10.0 -  9.8  | >  0.1
141
142       // Bei allen assert-Methoden kann man als ersten Parameter eine
143       // Fehlermeldung (vom Typ String) angeben, die im Falle eines
144       // Misslingens ausgegeben wird:
145       assertSame  ("Oh weia!", i1, i2);
146       assertEquals("Schlimm!", s1, s2);
147       assertTrue  ("????????", true);
148 //    assertNull  ("Ich wusste es!", s1); // Misslingt mit Fehlermeldung!
149       assertEquals("Zu ungenau!", d1, 16.9, 0.2);
150
151       // Vergleiche mit null:
152       assertSame  (null, null);          // Gelingt
153 //    assertSame  (s1,   null);          // Misslingt
154 //    assertSame  (null, s1);            // Misslingt
155       assertEquals(null, null);          // Gelingt
156 //    assertEquals(s1,   null);          // Misslingt
157 //    assertEquals(null, s1);            // Misslingt
158    } // assertSyntax
159    // ------------------------------------------------------------------------
160    static public void main(String[] _) {
161       // Die 4 Testfaelle in suite() ausfuehren lassen:
162       junit.awtui.TestRunner.run(StringBuilderJut.class);
163
164       pln("Die StringBuilderJut.suite() enthaelt " +
165                StringBuilderJut.suite().countTestCases() + " Testfaelle!");
166       pln("StringBuilderJut: Das war's erstmal!");
167    } // main
168    // ------------------------------------------------------------------------
169    // Eine Methode mit einem kurzen Namen:
170    static void pln(Object ob) {System.out.println(ob);}
171    // ------------------------------------------------------------------------
172 } // class StringBuilderJut
173 /* ------------------------------------------------------------------------
174 Ausgabe des Programms StringBuilderJut:
175
176 Die StringBuilderJut.suite() enthaelt 4 Testfaelle!
177 StringBuilderJut: Das war's erstmal!
178 setUp    wurde aufgerufen!
179 tearDown wurde aufgerufen!
180 setUp    wurde aufgerufen!
181 tearDown wurde aufgerufen!
182 setUp    wurde aufgerufen!
183 tearDown wurde aufgerufen!
184 setUp    wurde aufgerufen!
185 tearDown wurde aufgerufen!
186 ------------------------------------------------------------------------
```

```
187 assert-Methoden in Kuerze:
188
189 Es gibt zahlreiche assert-Methoden (assertEquals, assertSame, assertTrue,
190 assertFalse, assertNull, assertNotNull etc.).
191
192 Eine assert-Methode kann misslingen (das zaehlt als Fehler) oder gelingen.
193
194 Die Methoden namens assertEquals gelingen, wenn ihre beiden (letzten)
195 Parameter gleich sind. Diese Parameter duerfen zu einem primitiven Typ
196 (wie int, char, float, ...) oder zu einem Referenztyp (wie String,
197 Comparable, int[], ...) gehoeren. Wenn sie zu einem primitiven Typ
198 gehoeren, werden sie mit dem Operator == verglichen. Wenn sie zu einem
199 Referenztyp gehoeren, werden sie mit equals verglichen. Beide Parameter
200 sollten zum selben Typ gehoeren.
201
202 Die Methoden namens assertSame gelingen, wenn ihre (letzten) beiden
203 Parameter identisch sind. Diese Parameter duerfen zu einem primitiven
204 Typ (wie int, char, float, ...) oder zu einem Referenztyp (wie String,
205 Comparable, int[], ...) gehoeren. Sie werden auf jeden Fall mit dem
206 Operator == verglichen. Beide Parameter sollten zum selben Typ gehoeren.
207
208 Die Methode assertTrue gelingt, wenn ihr (letzter) Parameter gleich true
209 ist. Fuer die Methoden assertFalse und assertNull gilt Entsprechendes.
210
211 Bei allen assert-Methoden kann man als ersten Parameter eine Fehler-
212 meldung (vom Typ String) angeben, die ausgegeben wird, wenn die
213 Methode misslingt.
214
215 Die Methode assertNotSame gelingt, wenn assertSame misslingt (und
216 umgekehrt). Entsprechend fuer assertNotNull und assertNull. Eine Methode
217 assertNotEquals gibt es nicht (wieso?).
218
219 Wenn man Gleitpunktzahlen (vom Typ float bzw. double) mit assertEquals
220 vergleicht, muss man als letzten Parameter eine Toleranz angeben.
221 Nur wenn die beiden verglichenen Zahlen sich um mehr als diese Toleranz
222 unterscheiden, gelten sie als ungleich, sonst als gleich.
223
224 Anmerkung: Diese Toleranz ist eine absolute Zahl (d.h. unabhaengig von
225 der Groesse der verglichenen Zahlen). Wenn man eine relative Abweichung
226 zulassen will (z.B. um 3 Prozent oder um 0.05 Promille etc.), muss man
227 selbst eine entsprechende assert-Methode programmieren.
228 ------------------------------------------------------------------------ */
```