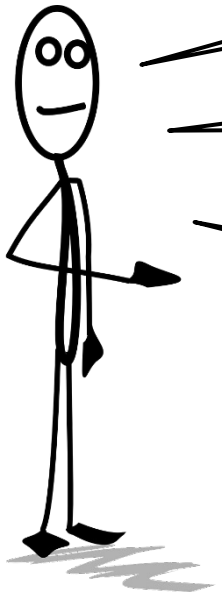
Technische Universität Braunschweig

# Programmieren 1 – Vorlesung #10

Arne Schmidt

# Wiederholung

# Letzte Woche

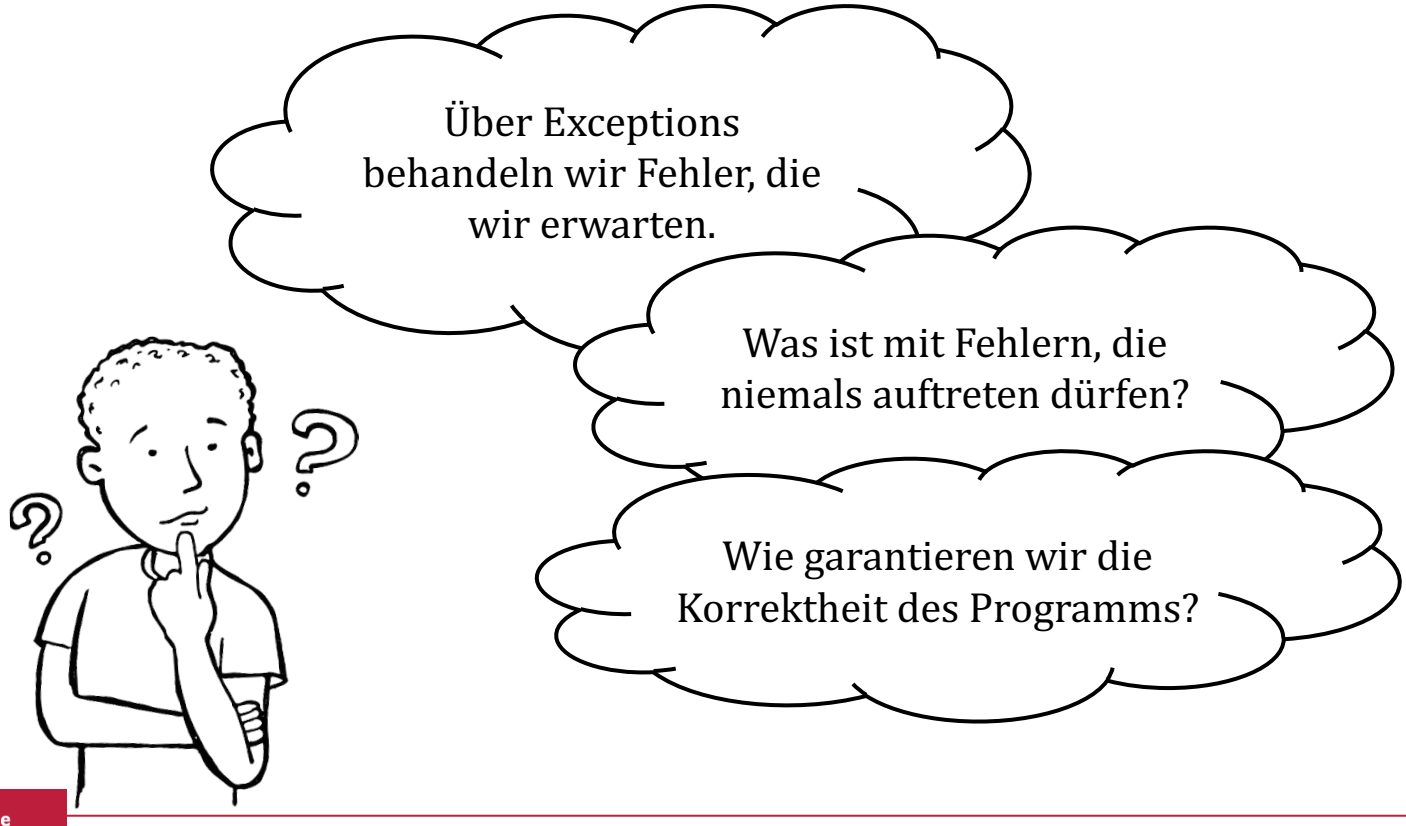


**Ausnahmen** sind Objekte der Klasse **Throwable** oder einer ihrer Unterklassen

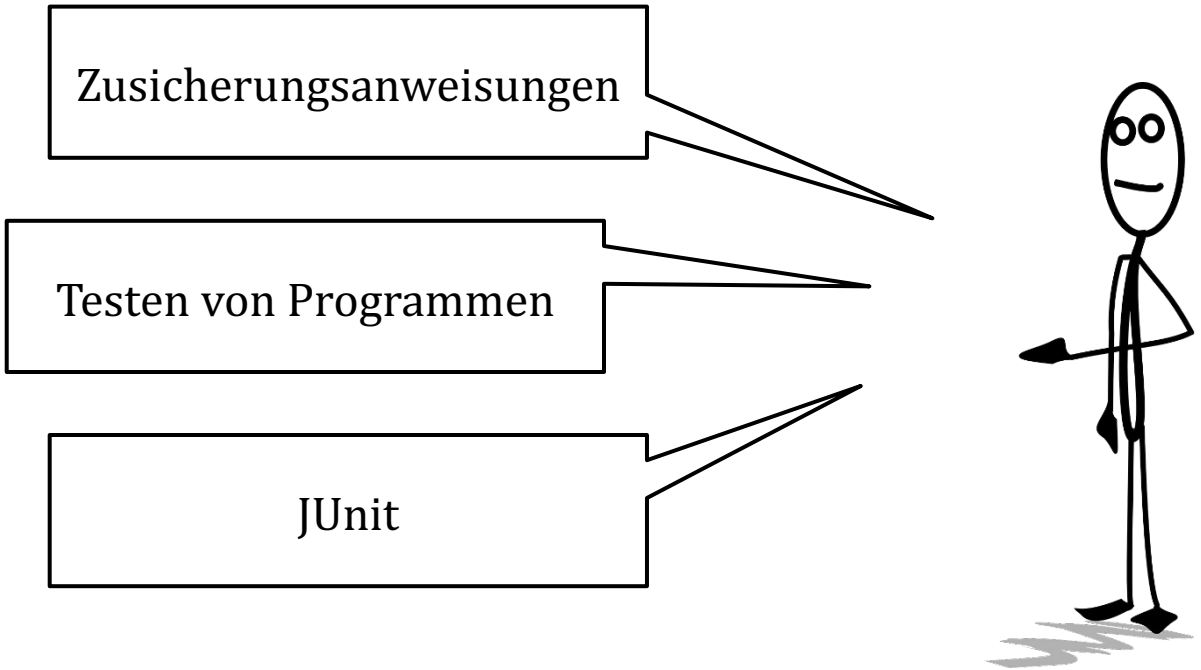
Behandeln von Ausnahmen über **try-catch**-Anweisung;
Auslösen per **throw**-Anweisung

catch-or-throw-Regel

Eigene Ausnahmeklassen


Über Exceptions behandeln wir Fehler, die wir erwarten.
Was ist mit Fehlern, die niemals auftreten dürfen?
Wie garantieren wir die Korrektheit des Programms?

# Heute



# Kapitel 6.2 – Assertions

# assert

Ab Java 1.4 kann mit der Anweisung **assert** überprüft werden, ob eine Bedingung gilt. Im Falle des Scheiterns, wird ein **AssertionError** ausgegeben.

Syntax:
assert Ausdruck1 [: Ausdruck2]

Ausdruck zum Testen mit Rückgabetyp boolean

Ist der Wert von Ausdruck1 true, dann passiert nichts weiter.

Optionaler Ausdruck, dessen Wert an den Konstruktor des AssertionError übergeben wird. Typischerweise eine Nachricht.

# Warum assertion?

> „*Use [exceptions] for conditions you expect to occur; use assertions for conditions that should never occur*“
> – Steve McConnell, Code Complete

> *Assertions are especially useful in large, complicated programs and in high-reliability programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on.*
> - Steve McConnell, Code Complete

# Anwendungen

Die assert-Anweisung wird vor allem für folgende Bedingungen benutzt:

- Vor- und Nachbedingungen
- Invarianten:
  - Schleifeninvariante
  - Kontrollflussinvariante
  - Klasseninvariante

Zwar kann die Korrektheit eines Programm(stück)s mit assert-Anweisungen mathematisch nicht bewiesen werden, dennoch können sie bei sinnvoller Anwendung erheblich zur **Programmsicherheit** beitragen.

Asserts helfen vor allem während der Programmentwicklung zu testen!

# Ein Beispiel – public Methode

```
static public int rest (int x, int y) {
   if (x < 0 || y <= 0) {
      throw new IllegalArgumentException();
   }
   int q = 0;
   int r = x;
   assert x == q * y + r && 0 <= r; // Vorbedingung / Schleifeninvariante
   while (r >= y) {
      r = r - y;
      q = q + 1;
      assert x == q * y + r && 0 <= r; // Schleifeninvariante
   }
   assert x == q * y + r && 0 <= r && r < y; // Nachbedingung
   return r;
}
```

# Ein Beispiel – private Methode

```
static private int rest (int x, int y) {
   assert x >= 0 && y > 0;
   int q = 0;
   int r = x;
   assert x == q * y + r && 0 <= r; // Vorbedingung / Schleifeninvariante
   while (r >= y) {
      r = r - y;
      q = q + 1;
      assert x == q * y + r && 0 <= r; // Schleifeninvariante
   }
   assert x == q * y + r && 0 <= r && r < y; // Nachbedingung
   return r;
}
```

# Kontrollflussinvariante

Assertions statt Kommentare!

```
if (i % 3 == 0) {
...
} else if (i % 3 == 1) {
...
} else { // (i % 3 == 2)
...
}
```

```
if (i % 3 == 0) {
...
} else if (i % 3 == 1) {
...
} else {
assert (i % 3 == 2);
...
}
```

Aufpassen! Modulo (%) kann in Java einen negativen Wert liefern!

# Kontrollflussinvariante – unerreichbare Zustände

Nutze assert false; für nicht Stellen, die nicht erreicht werden sollen.

An dieser Stelle noch besser: Enum verwenden!

```
final int MONTAG = 1, /* ... ,*/ SONNTAG = 7;
int wochentag ;
/* ... */
switch ( wochentag ) {
   case MONTAG :
      /* ... */
   case SONNTAG :
      /* ... */
   default:
      assert false ;
}
```

# Komplexe Zusicherungen

Ggf. lässt sich eine Zusicherung nicht mit einfachen Operationen durchführen.

Betrachte beispielsweise:
int[] a = new int[n];

Es soll nun zugesichert werden, dass a sortiert ist. Mathematisch ist das einfach:

$$\forall i \in \{0, \dots, n-2\}: a[i] \leq a[i+1]$$

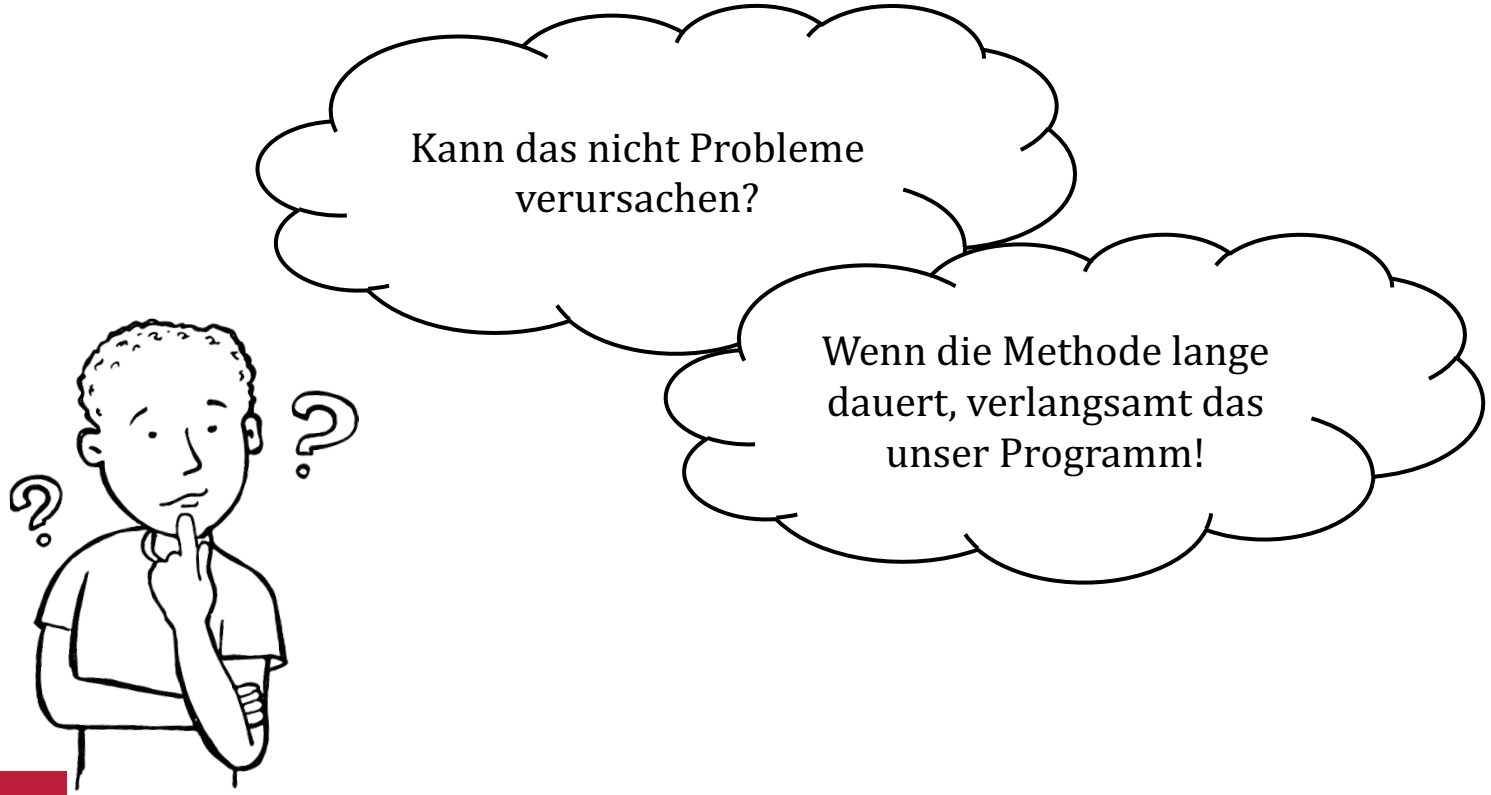
Wie können wir das mit assert testen?

# Assert und Methoden

Auch bei assert-Anweisungen können Methoden aufgerufen werden.

```
boolean isSorted (int [] a) {
   for (i = 0; i < a.length - 1; ++i) {
      if (a[i] > a[i + 1]) {
         return false ;
      }
   }
   return true ;
};
/* ... */
assert isSorted(a);
```

# Laufzeiten und Assertions



# Ein- und ausschalten von Assertions: Bedingtes Übersetzen

Anweisungen innerhalb einer if-Bedingung, die zur Kompilierzeit nicht erreicht werden, übersetzt der Compiler nicht.
Achtung: Generell gilt, dass bei nicht erreichbaren Anweisungen Fehler ausgegeben werden.

```
static final boolean SCHALTER = false // oder true
if ( SCHALTER )
   assert Anweisung ;
```

Sind alle Assertions in dieser Form, können diese ein- und ausgeschaltet werden.

# Ein- und ausschalten von Assertions: java Option

Nach Kompilieren können Assertions ein- und ausgeschaltet werden:

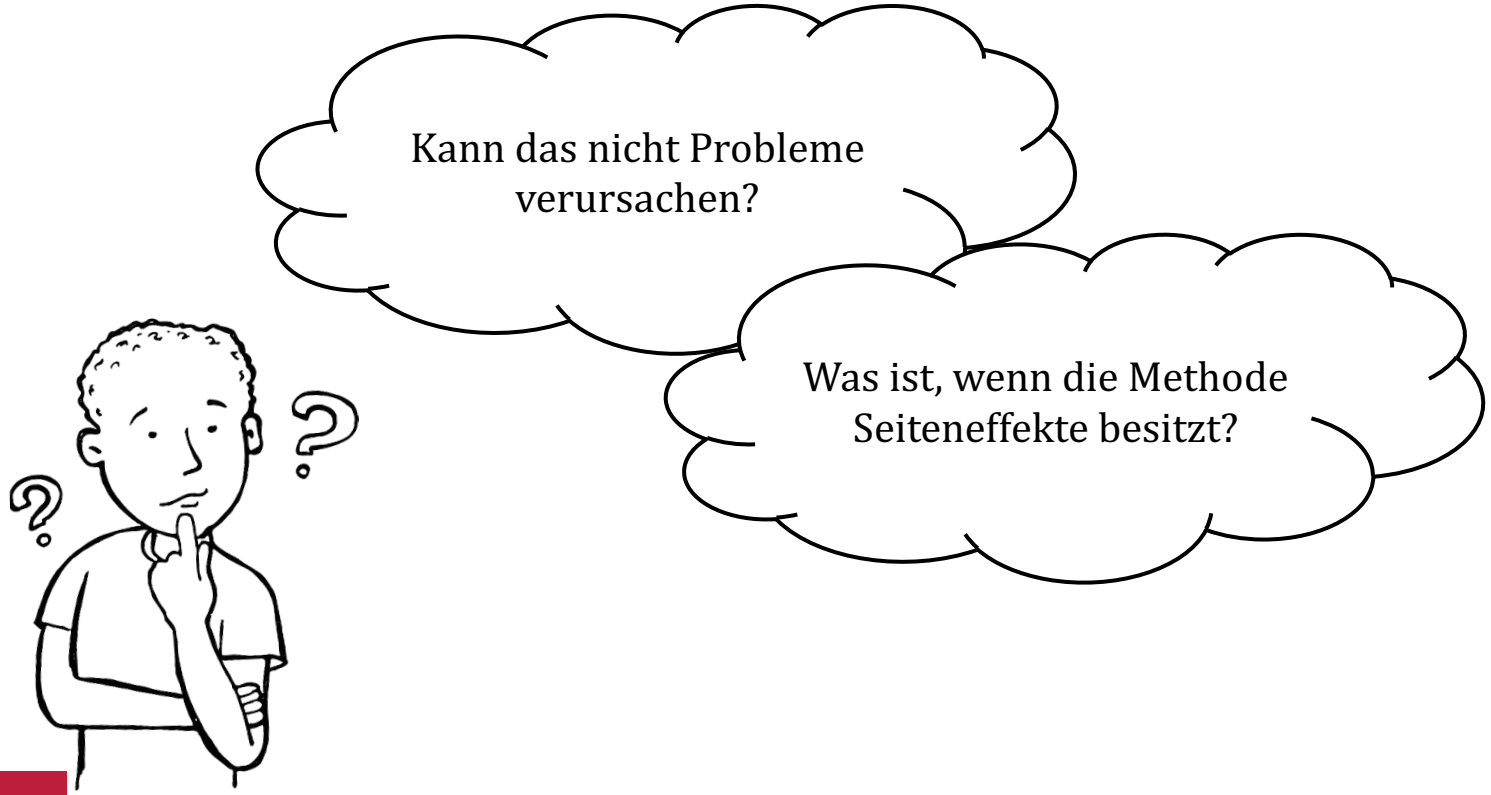
```
java –enableassertions MyJavaFile
java –disableassertions MyJavaFile
```

Oder in Kurzform:

```
java –ea MyJavaFile
java –da MyJavaFile
```

Wichtig:
Standardmäßig sind Assertions ausgeschaltet.

# Assertions und Seiteneffekte



# Vermeide Seiteneffekte in Assertions!

Besitzt eine Methode Seiteneffekte und wird über ein assert aufgerufen, ändert das Ausschalten der Assertions die Logik!

Wir können darüber aber erzwingen, dass Assertions aktiviert werden müssen:

```
static boolean schalter = false ;
assert schalter = true ; // gewollter Seiteneffekt
if (! schalter ) {
   throw new RuntimeException (
      "Zusicherungen müssen eingeschaltet sein .");
   }
```

# Kapitel 6.3 – Testen von Programmen

# Verifikation vs Testen

## Verfikation

- **Mathematischer Nachweis** der (partiellen oder totalen) Korrektheit
- Formaler Beweis **zeigt Abwesenheit** von Fehlern

## Testen

- **Probeweiser Ablauf** eines Programms
- Aussagekraft hängt von getesteten Eingabeparametern ab
- **Zeigt nur Anwesenheit** von Fehlern; nie deren Abwesenheit.

# Was und wie muss getestet werden?

Modultest und Integrationstest, d.h. funktionieren Komponenten im Einzelnen und im Zusammenspiel?

Black-Box-Test
- Implementierung nicht bekannt.
- Wähle Testfälle abhängig von der Beschreibung der Methode / Klasse

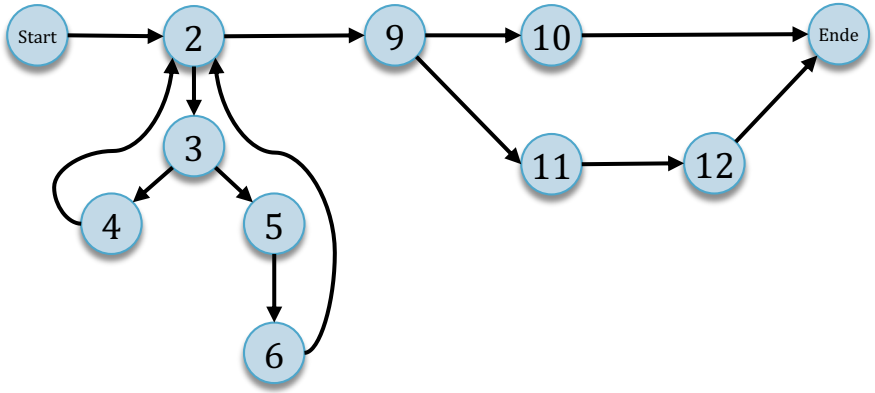
White-Box-Test:
- Implementierung bekannt
- Wähle Testfälle abhängig von der Teststrategie (nächste Slide)

Testfälle können im Allgemeinen beinhalten:
Wertebereiche / Grenzwerte der Daten, unzulässige Eingaben oder zufällige Eingaben.

# Kontrollflussorientierte Testverfahren

```
1.  public static int ggt(int a, int b) {
2.      while (a > 0 && b > 0){
3.          if (a > b){
4.              a -= b;
5.          } else {
6.              b -= a;
7.          }
8.      }
9.      if (b == 0){
10.         return a;
11.     } else {
12.         return b;
13.     }
14. }
```

Kontrollflussgraph



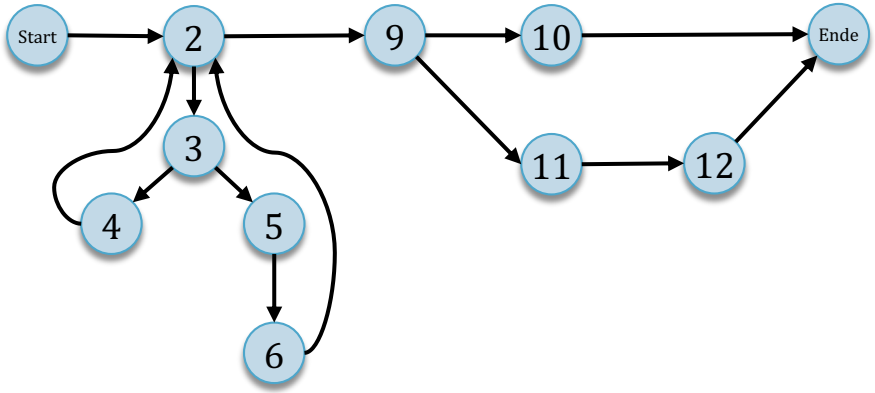

Im $C_0$-Test müssen alle Anweisungen mindestens einmal durchlaufen werden.

Im $C_1$-Test müssen alle Kanten des *Kontrollflussgraphen* mindestens einmal durchlaufen werden.

# Kontrollflussorientierte Testverfahren

```
1.  public static int ggt(int a, int b) {
2.      while (a > 0 && b > 0){
3.          if (a > b){
4.              a -= b;
5.          } else {
6.              b -= a;
7.          }
8.      }
9.      if (b == 0){
10.         return a;
11.     } else {
12.         return b;
13.     }
14. }
```

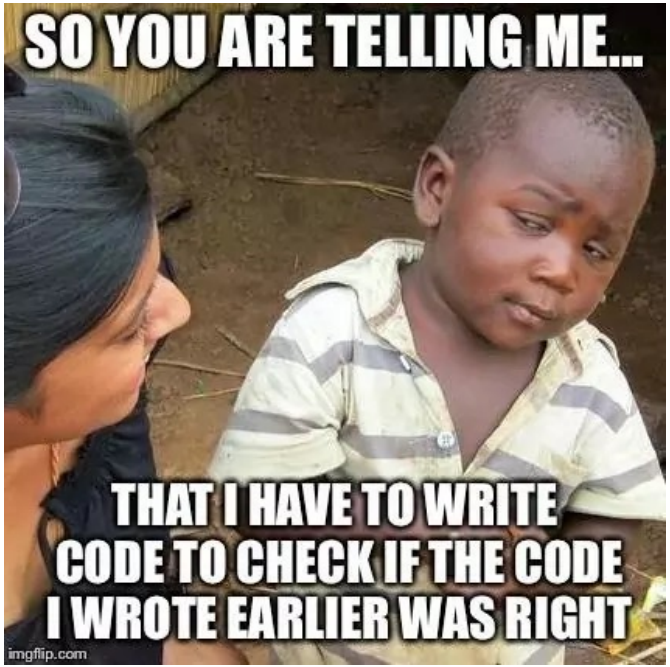
Kontrollflussgraph




In diesem Beispiel besitzen $C_0$ und $C_1$ die gleichen Tests:
(9, 8) testet Anweisungen 2 – 6 und 9 – 10.
(1, 0) testet Anweisungen 2, 9, 11 und 12.

# Wo testen?


Wo schreibt man die Tests?
Einfach in die Klassen rein? Das bläht die nur auf!


SO YOU ARE TELLING ME...
THAT I HAVE TO WRITE CODE TO CHECK IF THE CODE I WROTE EARLIER WAS RIGHT
imgflip.com

# JUnit

JUnit ist ein Framework zum Testen von Java-Programmen und ermöglicht automatisierte Tests einzelner Units, z.B. Tests von Klassen und Methoden.

JUnit ist nicht Bestandteil des JDKs. Es kann über https://junit.org/ heruntergeladen werden.

Um JUnit zu nutzen, muss die Bibliothek im CLASSPATH eingebunden oder als Parameter übergeben werden.

# JUnit Tests

Tests lassen sich mit geringem Aufwand und ohne Dialog durchführen. Damit kann häufig getestet werden.

Schlägt ein Test fehl, wird beispielsweise ein AssertionError ausgelöst.

- Jeder zu testenden Klasse wird mit JUnit eine eigene Testklasse zugeordnet.
- Jeder zu testenden Methode können mehrere Testmethoden geschrieben werden (z.B. für $C_0$- bzw. $C_1$-Tests)

Der Kopf von Testmethoden ist, bis auf den Namen, vorgeschrieben und wird mit @Test annotiert:

```
@Test
public void testMethode(){...}
```

# JUnit: Beispiel – Abessinische Bauernmethode

```
public class Beispiel {
   static boolean odd(long x) {
      if (x == 2 * (x / 2)) {
         return false ;
      }
      return true ;
   }

   static long produkt (long x, long y) {
      long u = x, v = y, w = 0;
      while (v > 0) {
         if ( odd (v)) {
            w = w + u;
         }
         v = v / 2;
         u = u + u;
      }
      return w;
   }
}
```

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

public class BeispielTest {

   @Test
    public void test () {
      long expected = -255;
      long actual = Beispiel.produkt(-15, 17);
      assertEquals(expected, actual);
      expected = 0;
      actual = Beispiel.produkt(3456, 0);
      assertEquals(expected, actual);
   }
}
```

Test: Geht es für alle Kombinationen von Vorzeichen?

# JUnit: Ausführen

Aufruf:

```
javac BeispielTest.java
java –ea org.junit.runner.JUnitCore BeispielTest
```

Ausgabe:

```
JUnit version 4.11
.
Time : 0.019

OK (1 test )
```

# JUnit: Ausführen bei scheiternden Assertions

Ausgabe (wenn expected = 0 zu expected = 1 geändert wird):

```
JUnit version 4.11
.E
Time : 0.026
There was 1 failure :
1) test ( BeispielTest )
java . lang . AssertionError : expected :<1> but was :<0>
        at org . junit . Assert . fail ( Assert . java :88)
        at org . junit . Assert . failNotEquals ( Assert . java :743)
        at org . junit . Assert . assertEquals ( Assert . java :118)
        ...
```

# JUnit: assertEquals

Die Methode assertEquals ist vielfach überladen. Einige Beispiele:

- assertEquals(long e, long a)
  assertEquals(double e, double a)
  Die Werte werden auf numerische Gleichheit getestet.

- assertEquals(double e, double a, double d)
  e und a dürfen höchstens um den Betrag von d voneinander abweichen.

- assertEquals(Object e, Object a)
  e und a werden mit der Methode equals verglichen.

Für boolesche Ergebnisse existiert bspw.

- assertTrue(boolean a)
- assertFalse(boolean a)
  a muss true bzw. false liefern.

# JUnit: Protokollierung

JUnit protokolliert, welche Tests fehlgeschlagen sind.

Um die Übersicht zu behalten, können den assert-Methoden ein String zur Beschreibung des Testfalls übergeben werden, der im Fehlerfall im Protokoll erscheint.

```
assertEquals (" Testfallbeschreibung ", expected , actual);
```

# JUnit: Weitere Tests - Exceptions

Auf Exceptions testen:
In JUnit 4 können erwartete Exceptions in der Annotation festgehalten werden

```
@Test(expected = NameException.class)
```

In Junit 5 funktioniert das feiner, erfordert aber Lamda-Ausdrücke

```
Exception exception = assertThrows(RuntimeException.class,
    () -> {Integer.parseInt("1a"); }
);

String expectedMessage = "For input string";
String actualMessage = exception.getMessage();
assertTrue(actualMessage.contains(expectedMessage));
```

# JUnit: Weitere Tests - Zeitlimit

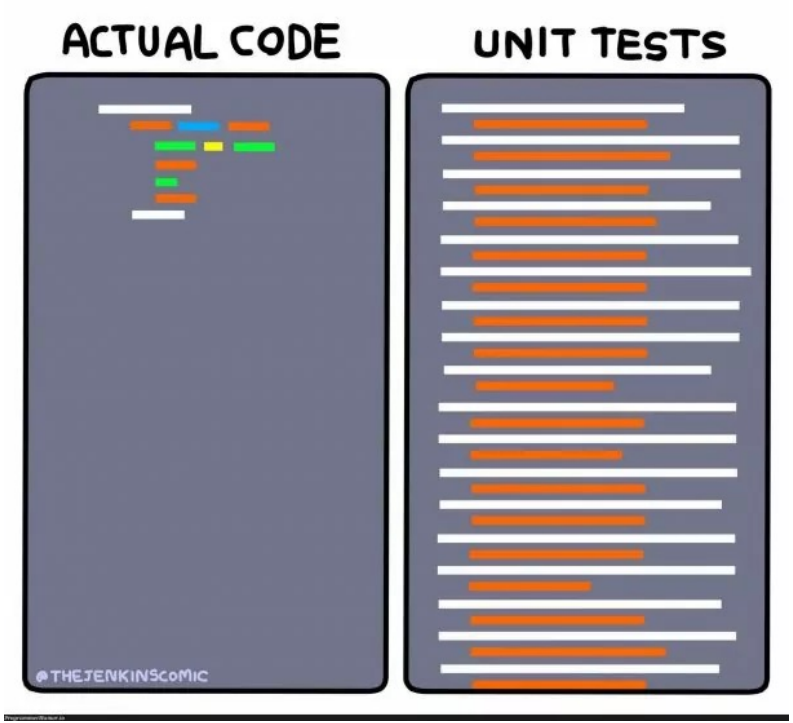
Auf Zeitlimits testen:
In JUnit 4 können Zeitlimits in der Annotation festgehalten werden

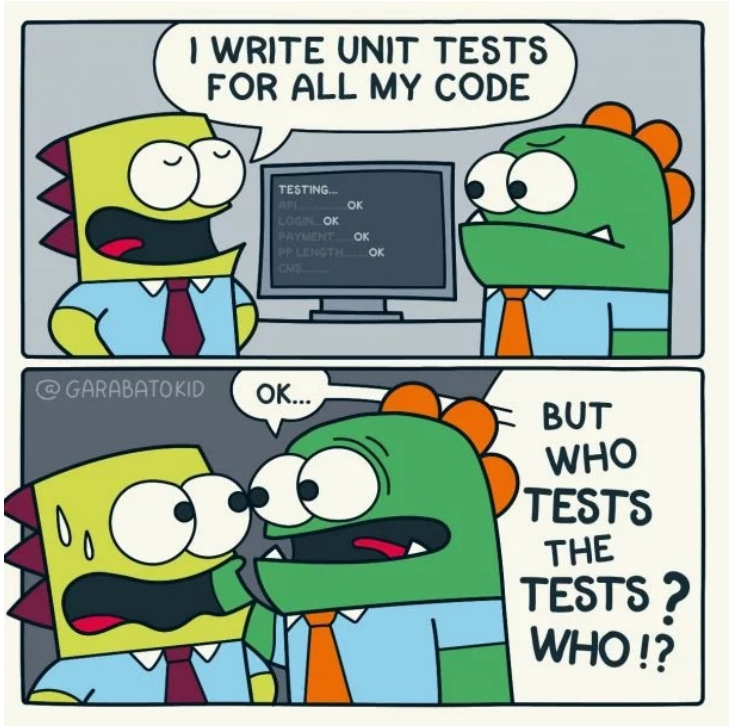
```
@Test(timeout =2000) //Zeitlimit ist 2 Sekunden
```

In Junit 5 funktioniert das feiner, erfordert aber Lamda-Ausdrücke

```
@Test
void testGetValue() throws InterruptedException {
   Assertions.assertTimeout(Duration.ofSeconds(2),
      () -> { meineMethode(); }
   );
}
```

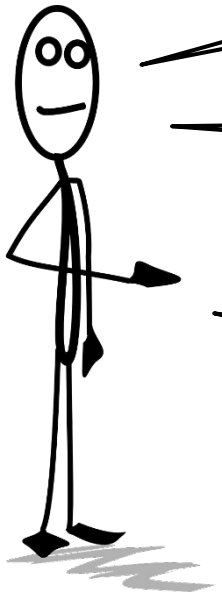
# JUnit: Anmerkungen




Wenig Code kann viele Tests erfordern!

# Zusammenfassung



**Assertions** zum Testen, z.B. für testgetriebene Programmierung

$C_0$- und $C_1$-Tests

**JUnit** für automatisiertes Testen.

# Nächste Woche

Wie können Daten während der Laufzeit eingegeben werden?

Wie können Daten aus Dateien ausgelesen oder in Dateien geschrieben werden?

Wie kann die Ausgabe auf der Konsole möglich gut lesbar angezeigt werden?