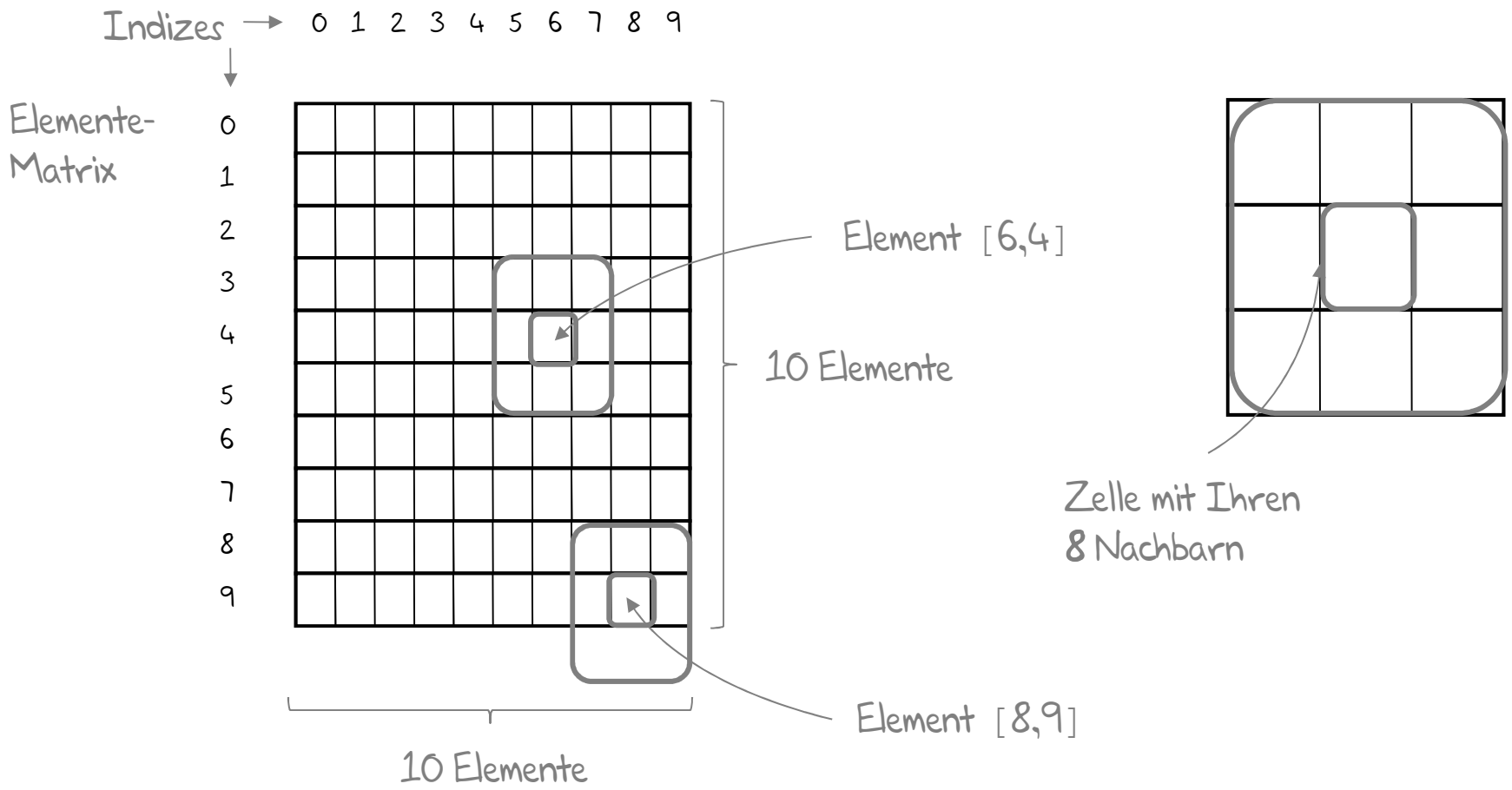
Conway's Game of Life

Man stelle sich vor, eine Welt bestünde nur aus einer 2 -dimensionalen Matrix:



Jeder Eintrag, wir nennen ihn jetzt mal Zelle, kann innerhalb dieser Welt zwei Zustände annehmen: sie ist entweder lebendig oder tot. Jede Zelle interagiert dabei von Generation zu Generation mit ihren maximal 8 Nachbarn.

Die vier Spielregeln

Die Überführung der Welt zu einer neuen Generation, also die Überführung aller Zellen in ihren neuen Zustand, unterliegt den folgenden vier Spielregeln:

1. jede lebendige Zelle, die weniger als zwei lebendige Nachbarn hat, stirbt an Einsamkeit
2. jede lebendige Zelle mit mehr als drei lebendigen Nachbarn stirbt an Überbevölkerung
3. jede lebendige Zelle mit zwei oder drei Nachbarn fühlt sich wohl und lebt weiter
4. jede tote Zelle mit genau drei lebendigen Nachbarn wird wieder zum Leben erweckt

Die Idee besteht nun darin, eine konkrete oder zufällige Konstellation von lebendigen und toten Zellen in dieser Matrix vorzugeben. Das bezeichnen wir dann als die erste Generation. Die zweite Generation wird durch die Anwendung der vier Regeln auf jede der Zellen erzeugt. Es wird geprüft, ob Zellen lebendig bleiben, sterben oder neu entstehen.

Wird ein solches System beispielsweise mit einer zufälligen Konstellation gestartet, so erinnert uns das Zusammenspiel von Leben und Tod z.B. an Bakterienkulturen in einer Petrischale, daher der Name "Spiel des Lebens".

Erster Projektentwurf

Unsere Aufgabe wird es nun sein, eine sehr einfache Version von Conways Game of Life zu implementieren. Schön wäre auch eine Generationsfolge, damit wir die sich ändernde Welt visuell verfolgen können.

Hier ein Grobentwurf:

Identifizierte Teilaufgaben sind:

1. Grobarchitektur des Programms festlegen: Interaktion der Funktionen, Austausch der Daten und Programmschleife.
2. Datentyp für die zweidimensionale Weltmatrix definieren, dabei Lösungen für die Randproblematik finden.
3. Variable Programmparameter festlegen: Weltmatrixgröße, Verteilung der lebendigen und toten Zellen in der Startkonfiguration.
4. Funktion initWelt implementieren, die eine Welt erzeugt und entsprechend der vorgegebenen Verteilung füllt und zurückgibt.
5. Funktion zeigeWelt implementieren, die die aktuelle Welt auf der Konsole ausgibt.
6. Funktion wendeRegelnAn implementieren, die eine erhaltene Generationsstufe der Welt nach den Spielregeln in die nächste Generationsstufe überführt und das Resultat zurückgibt.

Im Folgenden werden wir die Teilaufgaben Schritt für Schritt erarbeiten.

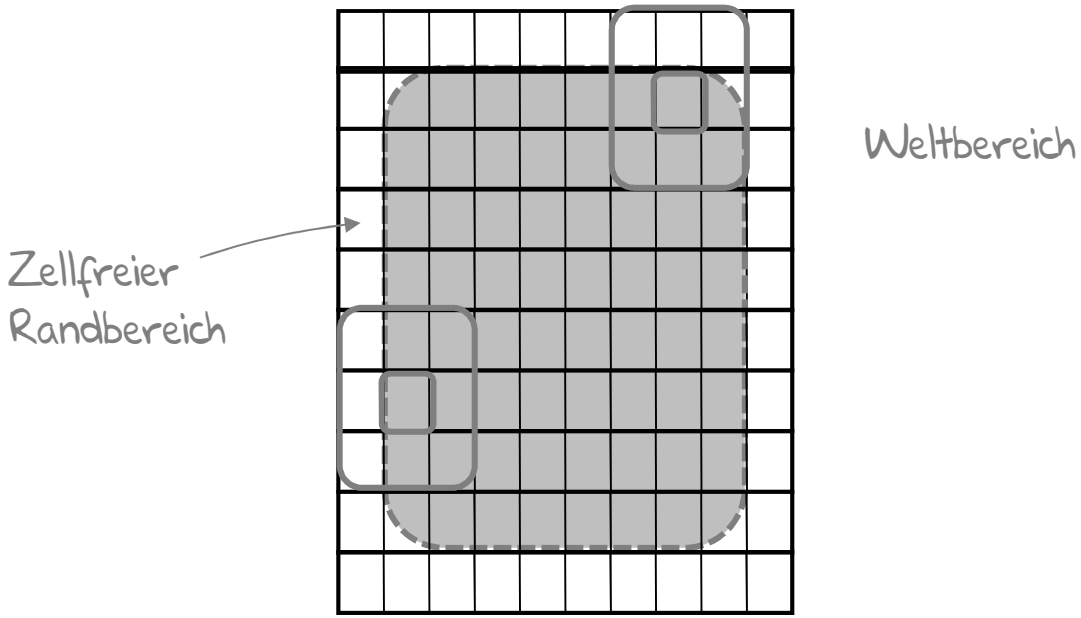
Grobarchitektur des Programms

```
public class GamesOfLife {
   // global definierte Konstanten für die beiden Dimensionen
   final static int DIM1 = <Dimension 1>;
   final static int DIM2 = <Dimension 2>;

   // liefert eine zufällig initialisierte Welt
   public static boolean[][] initWelt() {
   }

   // gibt die aktuelle Welt aus
   public static void zeigeWelt(boolean[][] welt) {
   }

   // wendet die 4 Regeln an und gibt die
   // Folgegeneration wieder zurück
   public static boolean[][] wendeRegelnAn(boolean[][] welt){
   }

   public static void main(String[] args) {
      boolean[][] welt = initWelt();

      System.out.println("Startkonstellation");
      zeigeWelt(welt);

      for (int i=1; i<=10; i++){
         welt = wendeRegelnAn(welt);
         System.out.println("Generation "+i);
         zeigeWelt(welt);
      }
   }
}
```

Welt und Randproblematik

In der main-Funktion werden wir eine zweidimensionale Matrix welt anlegen. Für die möglichen Zellzustände ist es naheliegend, eine Matrix vom Typ boolean zu verwenden. Eine tote Zelle wird durch false und eine lebendige Zelle entsprechend durch true repräsentiert.

Es gibt nun verschiedene Möglichkeiten mit der Randproblematik umzugehen. Wenn wir also die acht Nachbarn eines Weltelements erfragen, wollen wir gerne auf eine IndexOutOfBoundsException verzichten. Trotzdem soll die Abfrage der Nachbarn direkt und nicht abhängig von der jeweiligen Position sein. Ein legitimer Weg, wie er hier zu sehen ist, zeigt unseren Lösungsansatz:



Festlegung der Parameter

Eine Welt der Dimension 10x10 sollte für den Einstieg genügen. Wir werden für den zusätzlichen Rand insgesamt also eine 12x12 Matrix benötigen.

Für die Startkonstellation wünschen wir uns 60% lebendige Zellen mit einer zufälligen Verteilung.

Zufällige Initialisierung der Welt

Angenommen, wir wollen einen Zufallswert erzeugen, der in 60% der Fälle true liefert:

```
if (Math.random() > 0.4)    // Zufallszahl im Intervall [0,1)
   welt[x][y] = true;       // 60% true
else
   welt[x][y] = false;      // 40% false
```

Zielgerichteter ist sicherlich:

```
welt[x][y] = Math.random() > 0.4;
```

Damit haben wir das Kernstück der Funktion initWelt bereits fertig. Wir erzeugen eine Welt vom Typ boolean[][] und füllen anschließend alle Elemente mit der gerade vorgestellten Möglichkeit mit Zufallswerten und geben diese dann an den Aufrufer zurück. Die Methode initWelt könnte also wie folgt aussehen:

```
public static boolean[][] initWelt() {
   boolean[][] welt = new boolean[DIM1][DIM2];

   for (int y=0; y<DIM2; y++)
      for (int x=0; x<DIM1; x++)
         if (y<1 || y>DIM2-1 || x<1 || x>DIM1-1)
            welt[x][y] = false;                // Rand
         else
            welt[x][y] = Math.random() > 0.4; // 60% lebendig

   return welt;
}
```

initWelt zweite Variante

In kürzerer Variante:

```
public static boolean[][] initWelt() {
   boolean[][] welt = new boolean[DIM1][DIM2];

   for (int y=1; y<DIM2-1; y++)
      for (int x=1; x<DIM1-1; x++)
         welt[x][y] = Math.random() > 0.4; // 60% lebendig

   return welt;
}
```

Anzeigen der Welt

Die Ausgabe einer zweidimensionalen Matrix haben wir bereits schon gesehen. An dieser Stelle ist es vielleicht nochmal interessant darauf hinzuweisen, dass wir in der Matrix Elemente vom Datentyp boolean gespeichert haben und ein vergleichender Ausdruck, wie welt[i][j]==true nicht notwendig ist:

```
public static void zeigeWelt(boolean[][] welt) {
   for (int y=1; y<DIM2-1; y++) {
      for (int x=1; x<DIM1-1; x++) {
         if (welt[x][y])
            System.out.print("X");
         else
            System.out.print(" ");
      }
      System.out.println();
   }
   System.out.println();
}
```

Wir geben natürlich nur die veränderbaren Elemente der Welt ohne Rand aus. Ein X steht für eine lebendige und ein Leerzeichen für eine tote Zelle.

Die vier Spielregeln anwenden I

Eine Hauptinformation für die Entscheidung, welche der Regeln greift, ist die Anzahl der vorhandenen Nachbarn. Es lohnt sich darüber nachzudenken, ob wir dafür eine eigene Funktion anbieten wollen.

Für die Nachbarschaftsberechnung einer Zelle benötigen wir die Weltmatrix sowie die Position mit x- und y-Koordinaten. Dann müssen wir in zwei Schleifen nur noch die lebenden Zellen aufsummieren und gegebenenfalls die Zelle selbst nochmal abziehen:

```
public static int anzNachbarn(boolean[][] welt, int x, int y) {
   int ret = 0;
   for (int i=x-1; i<=x+1; ++i)
      for (int j=y-1; j<=y+1; ++j)
         if (welt[i][j])
            ret += 1;

   // einen Nachbarn zuviel mitgezählt?
   if (welt[x][y])
      ret -= 1;

   return ret;
}
```

Wenn uns die Nachbarschaftsfunktion zur Verfügung steht, können wir den Abschnitt zu den Spielregeln sehr übersichtlich gestalten.

Die vier Spielregeln anwenden II

Um nun eine Folgegeneration zu berechnen, müssen wir eine zweite, leere welt_neu erzeugen. Anschließend gehen wir systematisch über alle Elemente der Matrix welt und zählen die jeweiligen Nachbarn.

Die Regeln liefern dann eine eindeutige Entscheidung, welchen Wert wir in der neuen Matrix an die gleichen Stelle schreiben:

```
public static boolean[][] wendeRegelnAn(boolean[][] welt) {
   boolean[][] welt_neu = new boolean[DIM1][DIM2];
   int nachbarn;

   for (int y=1; y<DIM2-1; y++)
      for (int x=1; x<DIM1-1; x++) {
         nachbarn = anzNachbarn(welt, x, y);

         if (welt[x][y]) {
            if ((nachbarn < 2)  || (nachbarn > 3))  // Regel 1, 2
               welt_neu[x][y] = false;

            if ((nachbarn == 2) || (nachbarn == 3)) // Regel 3
               welt_neu[x][y] = true;
         } else {
            if (nachbarn == 3)                      // Regel 4
               welt_neu[x][y] = true;
         }
      }

   return welt_neu;
}
```

Die vier Spielregeln anwenden III

Schauen wir uns kurz mal die vorliegenden Regeln an:

```
if (welt[x][y]) {
   // Regel 1, 2:
   if ((nachbarn < 2)  || (nachbarn > 3))
      welt_neu[x][y] = false;

   // Regel 3:
   if ((nachbarn == 2) || (nachbarn == 3))
      welt_neu[x][y] = true;
} else {
   // Regel 4:
   if (nachbarn == 3)
      welt_neu[x][y] = true;
}
```

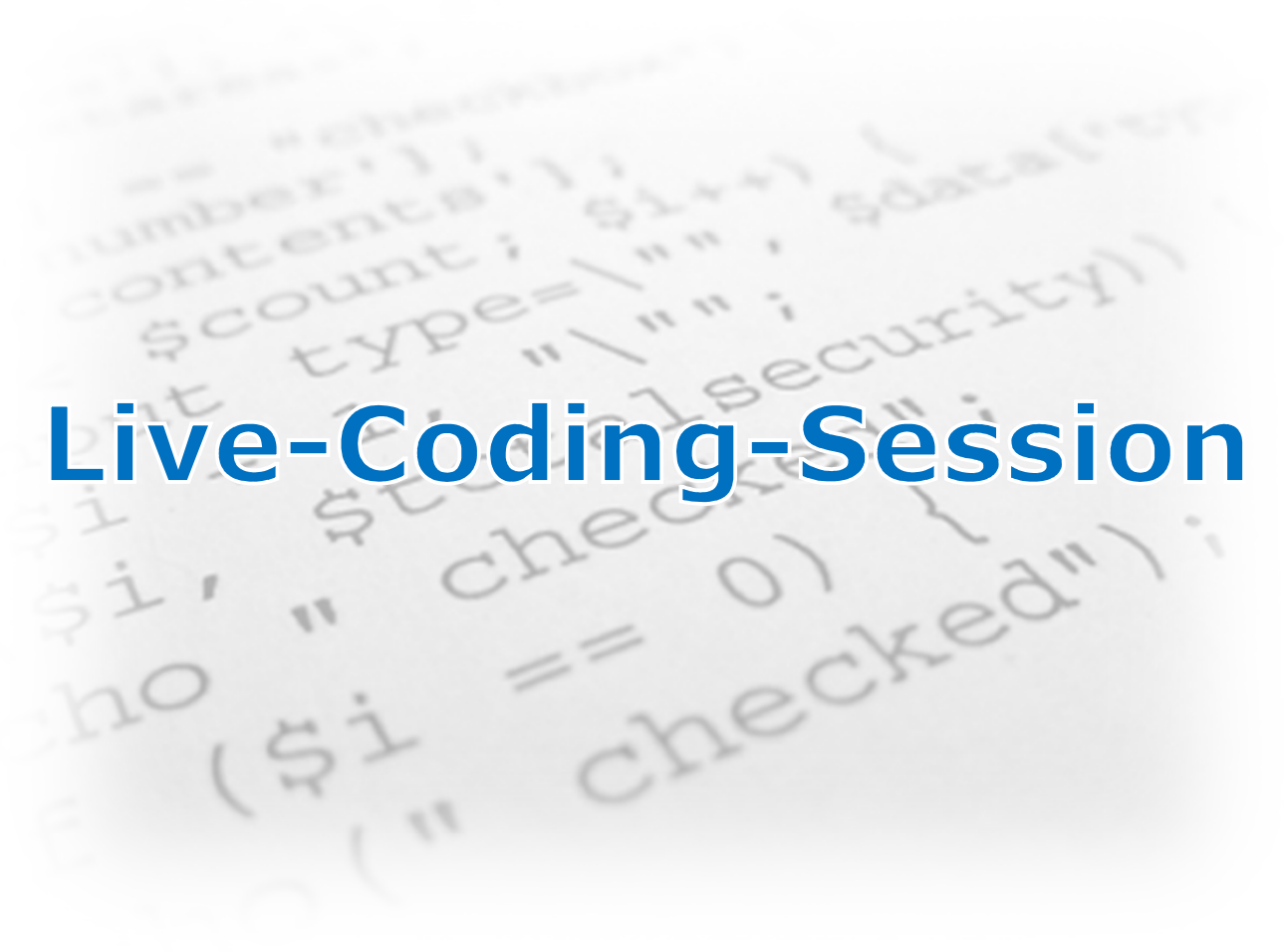
Das können wir mit etwas „Logik“ kürzen zu:

```
welt_neu[x][y] = (welt[x][y] && (nachbarn == 2)) || (nachbarn == 3);
```

Testlauf

Mit den vorgestellten Programmteilen haben wir jetzt ein voll funktionsfähiges Spiel des Lebens:

```
Startkonstellation
 XXX   X
 X    X XX
 XXX
X    X XX
X    XX XX
  X   X XX
 XXX
   X  X
 XXXX XX
  XXX

Generation 1
 XX    XXX
X      XXX
XXX   X
X X XX XX
 X   X
  XX XX XX
 X X   X
     XXX
 X    XX
        XX
```



Live-Coding-Session

Ausblick

Wer jetzt denkt, dass Conway's Game of Life nur reine Spielerei ist, der täuscht sich.

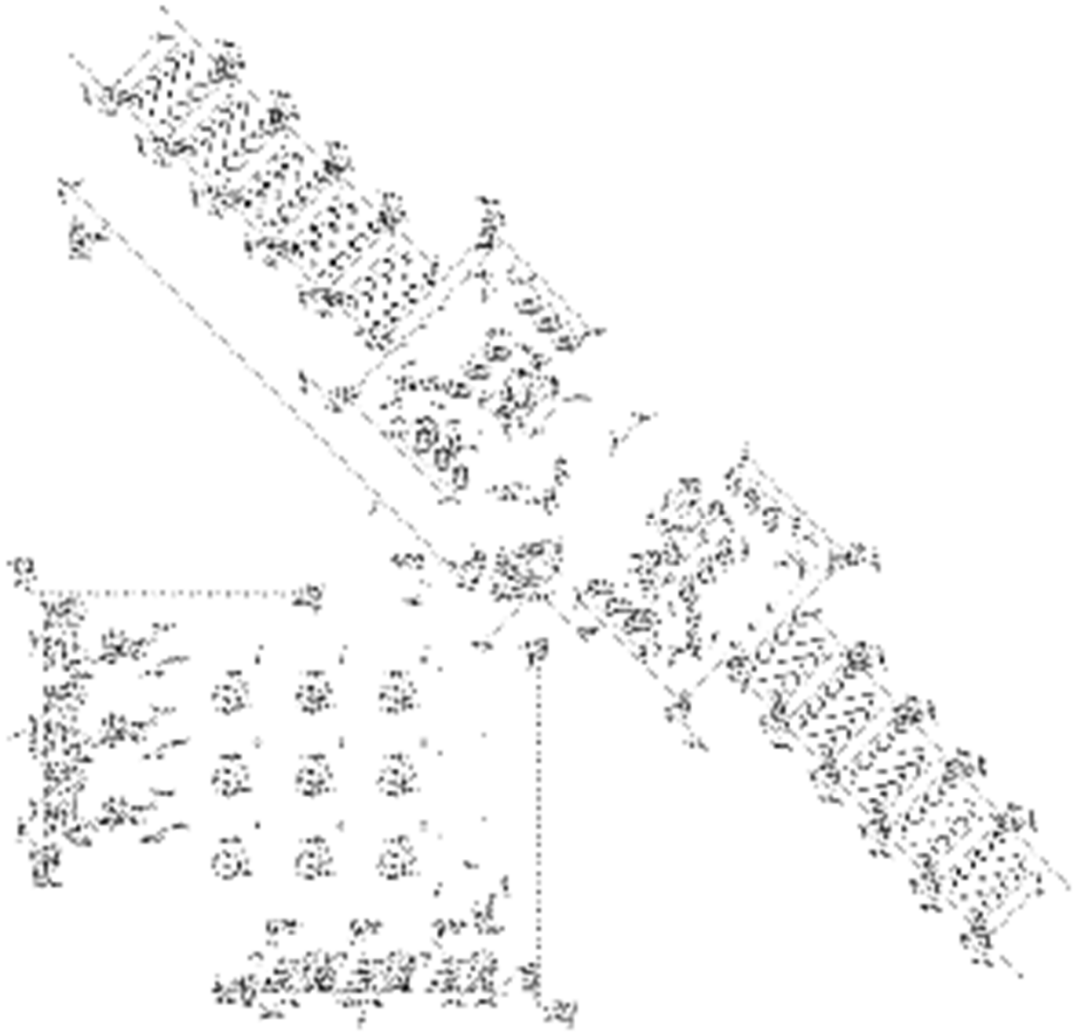
An dieser Stelle muss kurz erwähnt werden, dass dieses Modell aus Sicht der Berechenbarkeitstheorie ebenso mächtig ist, wie es beispielsweise die Turing-maschine und das Lambda-Kalkül sind. Informatikstudenten, die sich im Grundstudium mit dieser Thematik intensiv auseinandersetzen müssen, wissen welche Konsequenz das hat.

Für den interessierten Leser gibt es zahlreiche Webseiten und Artikel zu diesem Thema, eine Webseite[1)] möchte ich allerdings besonders empfehlen.

Aber Vorsicht: Conway's Game of Life kann süchtig machen!

1 http://www.conwaylife.com/

GoI Turing Maschine [Rendell]



http://www.youtube.com/watch?v=DD0B-4KNna8&feature=player_embedded#at=43

Das stets hungrige JavaGotchi

Im folgenden Abschnitt werden wir ein Mini-Tamagotchi implementieren, das in der zweiten Hälfte der 90er Jahren so populär gewesen war.

Der Einfachheit halber verzichten wir auf die Interaktion.

Zustands- und Aktivitätsdiagramme

Zustands- und Aktivitätsdiagrammen sehen sehr ähnlich aus. Wir haben es aber nicht mehr mit Anweisungen bzw. Aktivitäten zu tun, sondern mit Zuständen.

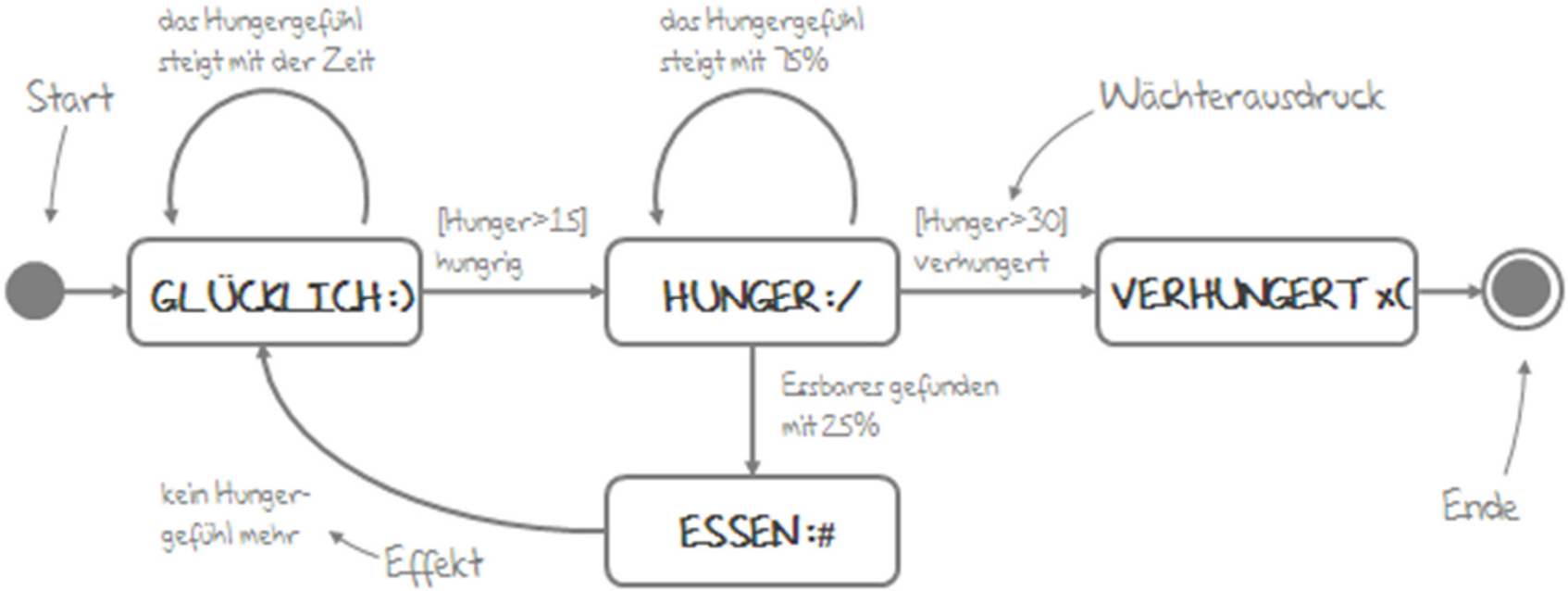
Zunächst definieren wir die vier unterschiedlichen Zustände GLÜCKLICH, HUNGER, ESSEN und VERHUNGERT für unser einfaches JavaGotchi. Folgende Szenarien wollen wir dafür realisieren:

Wenn das JavaGotchi keinen Hunger verspürt, dann ist es GLÜCKLICH. Ist etwas Zeit vergangen, wird sich ein Hungergefühl langsam entwickeln. Sollten mehr als 15 Einheiten Hunger vorhanden sein, dann wechseln wir in den Zustand HUNGER. Das JavaGotchi beginnt mit der Suche nach Futter und hat damit in 25% der Fälle Erfolg. Bei Erfolg wechselt es in den Zustand ESSEN und empfindet kein Hungergefühl mehr - anschließend wechselt es in den Zustand GLÜCKLICH.

Sollte es bei der Futtersuche über eine längere Zeit Pech haben und mehr als 30 Einheiten Hunger verspüren verhungert es und wechselt damit in den Zustand VERHUNGERT.

Zustandsdiagramme entwerfen

Hier sehen wir die Darstellung der Zustände für unser JavaGotchi in dem sogenannten Zustandsdiagramm:



Zustände werden dabei durch abgerundete Kästchen und der entsprechenden Bezeichnung symbolisiert. Wir können zwischen Zuständen wechseln. Die Pfeile (Transitionen) zeigen die Überführungsrichtungen der Zustände und eine Beschreibung liefert die notwendigen Voraussetzungen dafür. Eine solche Verbindung kann einen Effekt beschreiben.

Wenn wir eine Bedingung mit einem Zustandsübergang verknüpfen wollen, können wir wie in der Abbildung gezeigt, einen Wächterausdruck angeben.

Enumeratoren für Zustände einsetzen

Eine einfache Möglichkeit die unterschiedlichen Zustände zu definieren ist der Einsatz von Enumeratoren.

Ein Enumerator repräsentiert für die verschiedenen, konstanten Werte intern einfach Zahlen, ist aber wesentlich besser lesbar:

enum <Bezeichner> {<Wert1>, <Wert2>, ...}

Für unsere vier unterschiedlichen Zustände können wir den Enumerator Zustand wie folgt angeben:

```
enum Zustand {HUNGER, VERHUNGERT, ESSEN, GLÜCKLICH};
```

JavaGotchi

```
public class JavaGotchi {
   enum Zustand {HUNGER, VERHUNGERT, ESSEN, GLÜCKLICH};

   public static void main(String[] args) {
      Zustand zustand     = Zustand.GLÜCKLICH;
      int hunger          = 0;
      boolean spielLaeuft = true;       // Kleiner Gameloop für unser JavaGotchi
      while (spielLaeuft) {
         switch (zustand) {
            case HUNGER:
               System.out.println(":/ ... hunger");
               if (Math.random()<0.25)  // in 25% der Fälle finden wir etwas Essbares
                  zustand = Zustand.ESSEN;
               break;
            case VERHUNGERT:
               System.out.println("x(");
               spielLaeuft = false;
               break;
            case ESSEN:
               System.out.println(":# ... essen");
               hunger  = 0;
               zustand = Zustand.GLÜCKLICH;
               break;
            case GLÜCKLICH:
               System.out.println(":)");
         }

         if (zustand != Zustand.ESSEN) {
            hunger += (int)(Math.random()*5);
            if (hunger>15) zustand = Zustand.HUNGER;       // :/
            if (hunger>30) zustand = Zustand.VERHUNGERT;   // x(
         }
      }
   }
}
```

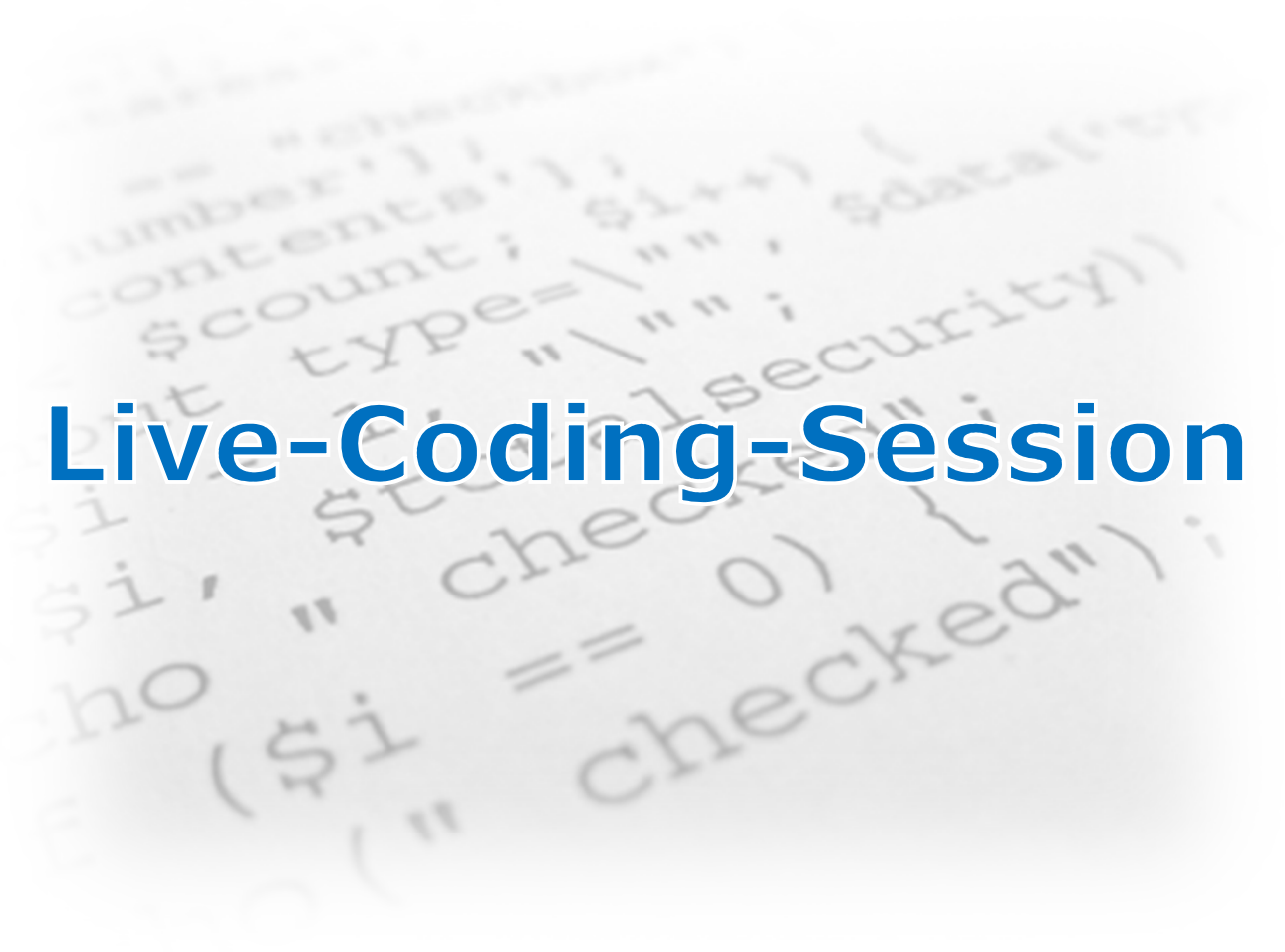
Starten wir das JavaGotchi

Schauen wir uns einen kleinen Testlauf an, bei dem die vielen glücklichen Smilies etwas abgekürzt dargestellt wurden:

```
:)
:)
...
:)
:/ ... hunger
:# ... essen
:)
:)
...
:)
:/ ... hunger
:/ ... hunger
:# ... essen
:)
...
```

Das Programm war allerdings so schnell an uns vorbeigelaufen, dass wir die einzelnen Zustände und Übergänge nicht nachvollziehen konnten. Wir können am Ende der while-Schleife nach der Anpassung der Parameter nach Zeile 40 eine kleine Pause von 400 ms einfügen:

```
try {
   Thread.sleep(400);
} catch (InterruptedException e) {
   e.printStackTrace();
}
```



Live-Coding-Session

Debuggen und Fehlerbehandlungen

Übersicht zum Vorlesungsinhalt

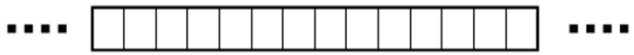
zeitliche Abfolge und Inhalte können variieren

Debuggen und Fehlerbehandlungen

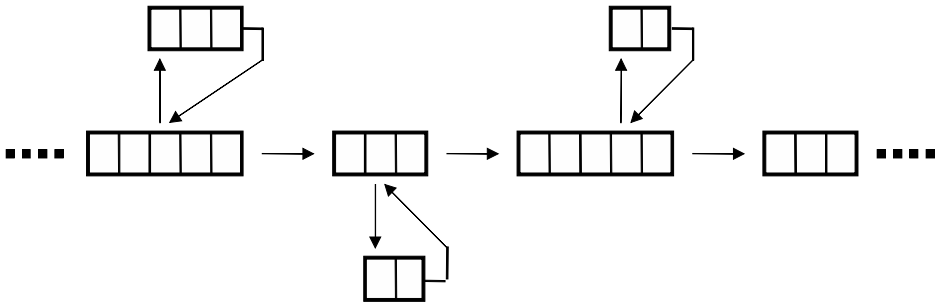
- Typische Fehlerklassen
- Vor- und Nachbedingungen
- Zusicherungen (assertions)
- Exceptions fangen
- Fehlerort lokalisieren
- Exceptions an Aufrufer weiterleiten
- Projekt: PI
- Textausgaben in der Konsole
- Zeilenweises Debuggen und Breakpoints
- Entwicklungsumgebung Eclipse
- Testgetriebene Softwareentwicklung
- Arbeiten mit JUnit
- Annotationen und Vergleichsmethoden
- Refactoring
- Checklist für TDD

Das richtige Konzept

Es sollte bei der Entwicklung darauf geachtet werden, dass nicht allzu viele Programmzeilen zu einer Funktion gehören. Das Problem sind oft wiederverwendete Variablen oder Seiteneffekte:



Besser ist es, das Programm zu gliedern und thematisch in Programmabschnitte zu unterteilen. Zum Einen wird damit die Übersicht gefördert und zum Anderen verspricht die Modularisierung den Vorteil, Fehler in kleineren Programmabschnitten besser aufzuspüren und vermeiden zu können:



Beim Conway-Projekt haben wir es bereits richtig vorgemacht.

Typisch auftretende Fehlerklassen

Wir unterscheiden hauptsächlich zwischen diesen drei Fehlerklassen:

1. Kompilierfehler: Die Syntaxregeln der Sprache wurden nicht eingehalten, der Java-compiler beschwert sich.

2. Laufzeitfehler: Im laufenden Programm werden Regeln verletzt, die zum Absturz führen können.

3. Inhaltliche Fehler: Das Programm tut nicht das Richtige, es gibt logische Fehler.

Am Anfang scheint es manchmal ausweglos zu sein, **Kompilierfehler** ohne Hilfe zu beheben. Aber mit einigen Beispielen und genügend Erfahrung sind das eigentlich die Fehler, die am schnellsten zu lösen sind, denn Java unterstützt uns mit den Fehlermeldungen enorm.

Laufzeitfehler sind schon unangenehmer, da sie oft nur unter bestimmten Voraussetzungen entstehen und teilweise nicht reproduzierbar sind. Solche Fehler müssen wir sehr Ernst nehmen und bereits beim Entwurf die typischen Fallstricke erkennen.

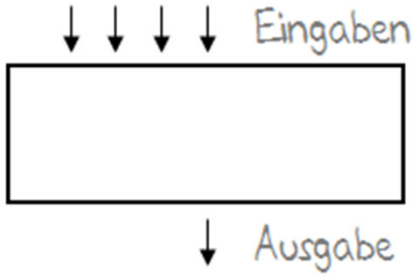
Logische Fehler sind am schwierigsten zu identifizieren. Was ist also, wenn zulässige Eingaben nicht zu gewünschten Ergebnissen führen? Nach einem Abschnitt über Fehlerbehandlungen in Java, in dem wir uns den Laufzeitfehlern widmen wollen, werden wir exemplarisch einen logischen Fehler Schritt für Schritt aufspüren.

Vor- und Nachbedingungen

Ein wichtiges Werkzeug der Modularisierung stellen die sogenannten **Constraints** (Einschränkungen) dar.

Beim Eintritt in einen Programmabschnitt (z.B. eine Funktion) müssen die Eingabeparameter überprüft werden, um klarzustellen, dass der Programmteil mit den Eingabedaten arbeiten kann.

Wir nennen das auch die notwendigen **Vorbedingungen** (preconditions). Wenn die Vorbedingungen erfüllt sind, muss der Programmabschnitt das Gewünscht liefern, also zugesicherte **Nachbedingungen** (postconditions) erfüllen:

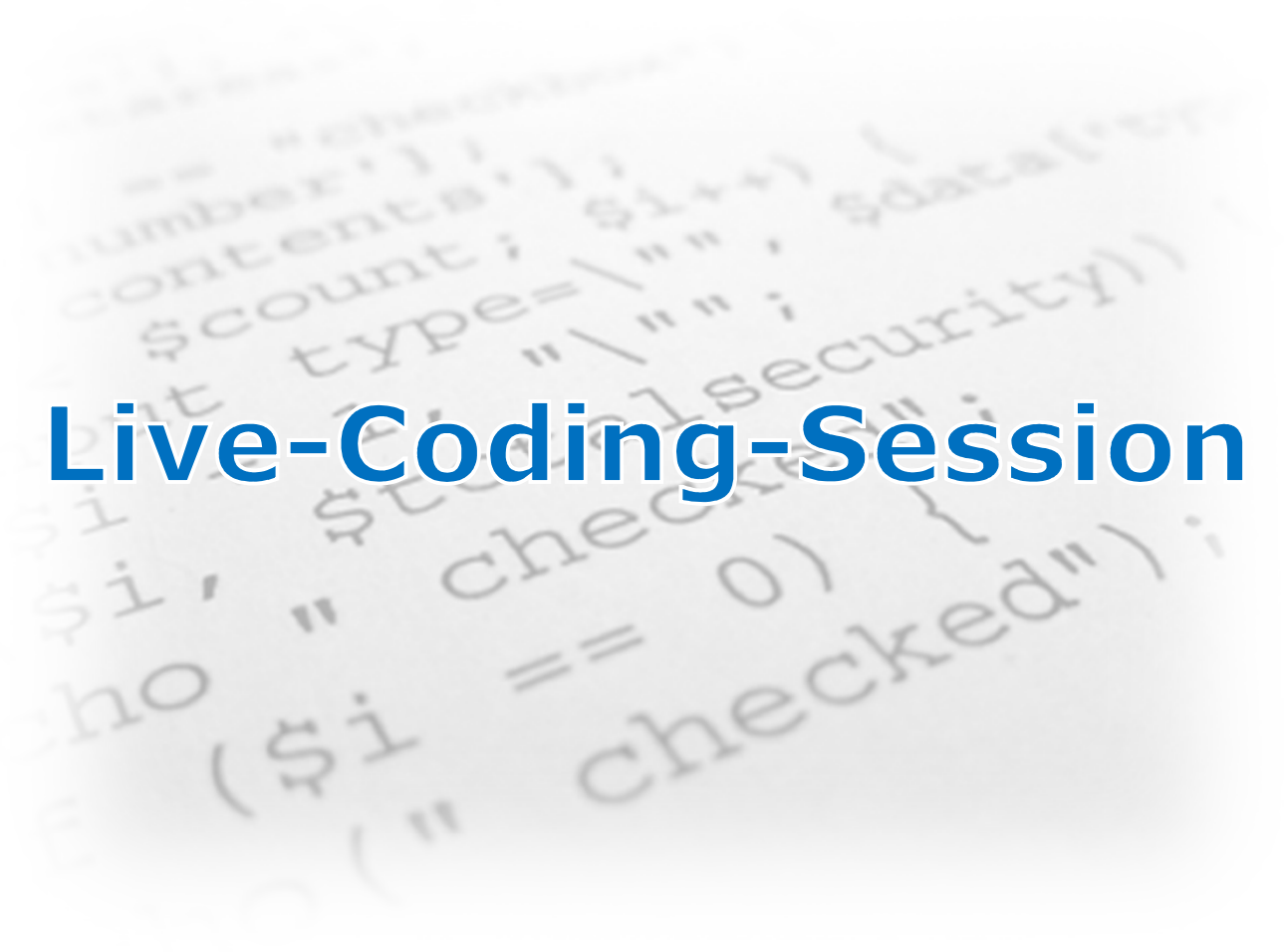


Der Entwickler einer Funktion ist dafür verantwortlich, dass auch die richtigen Berechnungen durchgeführt und zurückgeliefert werden. Dies stellt sicher, dass ein innerhalb eines Moduls auftretender Fehler, auch dort gesucht und behandelt werden muss und nicht im aufrufenden Modul, welches eventuell fehlerhafte Daten geliefert hat.

Beispiel zur Berechnung der Fakultät

Als Beispiel schauen wir uns mal die Fakultätsfunktion an:

```
public class Fakultaet {
   /*
    * Fakultätsfunktion liefert für n=0 ... 20 die entsprechenden
    * Funktionswerte n! = n * (n-1) * (n-2) * ... * 1
    * (0! ist per Definition 1)
    *
    * Der Rückgabewert liegt im Bereich 1 .. 2432902008176640000
    *
    * Sollte eine falsche Eingabe vorliegen, so liefert das Programm
    * als Ergebnis -1.
    */
   public static long fakultaet(int n) {
      // Ist der Wert außerhalb des erlaubten Bereichs?
      if ((n < 0) || (n > 20))
         return -1;

      long erg = 1;
      for (int i = n; i > 1; i--)
         erg *= i;
      return erg;
   }

   public static void main(String[] args) {
      for (int i = -2; i < 22; i++)
         System.out.println("Fakultaet von "+i+" ist "+fakultaet(i));
   }
}
```



Live-Coding-Session

Ausgabe der Fakultätsfunktion

Damit haben wir ersteinmal die Ein- und Ausgaben überprüft und sichergestellt, dass wir mit den gewünschten Daten arbeiten, nun müssen wir dafür Sorge tragen, dass die Module fehlerfrei arbeiten.

Wenn wir die Funktion fakultaet, wie in der main-Funktion gezeigt, mit verschiedenen Parametern testen, erhalten wir die folgende Ausgabe:

```
C:\Java>java Fakultaet
Fakultaet von -2 ist -1
Fakultaet von -1 ist -1
Fakultaet von 0 ist 1
Fakultaet von 1 ist 1
Fakultaet von 2 ist 2
Fakultaet von 3 ist 6
Fakultaet von 4 ist 24
...
Fakultaet von 18 ist 6402373705728000
Fakultaet von 19 ist 121645100408832000
Fakultaet von 20 ist 2432902008176640000
Fakultaet von 21 ist -1
```

Wir sehen, dass das Programm für unzulässige Eingaben wie erwartet eine -1 zurückliefert. Bei diesem Beispiel ist das machbar, denn die Fakultätsfunktion liefert nach ihrer Definition nur positive Werte.

Aber was machen wir, wenn wir beispielsweise eine Funktion erstellen wollen, die sowohl positive als auch negative Werte zurückliefern kann? Zu dieser Frage kommen wir zu einem späteren Zeitpunkt nocheinmal zurück.

Zusicherungen in Java

Gerade haben wir im Kommentarteil zu einer Funktion die Bedingungen für die Ein- und Ausgaben schriftlich festgehalten. Anstatt nun mit vielen if-Anweisungen, die Inhalte zu prüfen, können wir ein in Java vorhandenes Konzept verwenden.

So können wir Zusicherungen (assertions) zur Laufzeit eines Programms mit der folgenden Anweisung erzwingen:

```
assert <Bedingung>;
```

Sollte die Bedingung nicht erfüllt sein, bricht das Programm mit einer Fehlermeldung ab. Damit lassen sich auch logische Fehler in der Entwicklungsphase besser vermeiden und identifizieren.

Damit Java die Überprüfung der Zusicherungen durchführt, müssen wir das Programm mit einem kleinen Zusatz starten:

```
java -ea <Programmname>
```

Fakultätsfunktion und Zusicherungen

Um unsere Bedingungen für die Eingabe (n aus dem Intervall [0, 20]) und die Ausgabe (Ergebnis ist positiv), könnten wir beispielsweise wie folgt als Zusicherungen in die Funktion einbauen:

```
public static long fakultaet(int n) {
   // Ist der Wert außerhalb des erlaubten Bereichs?
   assert ((n >= 0) && (n <= 20)); // Vorbedingung (precondition)

   long erg = 1;
   for (int i = n; i > 1; i--)
      erg *= i;

   assert erg>0;                    // Nachbedingung (postcondition)
   return erg;
}
```

Wenn wir das veränderte Programm jetzt mit der Option -ea (enable assertions) starten, dann erhalten wir:

```
C:\Java>java -ea Fakultaet
Exception in thread "main" java.lang.AssertionError
        at FakultaetAssertion.fakultaet(FakultaetAssertion.java:15)
        at FakultaetAssertion.main(FakultaetAssertion.java:27)
```

Wir erkennen zwar, dass die Zusicherung in Zeile 15 zum Abbruch geführt hat, es wäre aber sicherlich hilfreicher, wenn die Fehlermeldung einen aussagekräftigen Hinweis enthält.

Fakultätsfunktion und "sprechende" Zusicherungen

Wir können folgende Syntax verwenden, die um eine Fehlermeldung erweitert ist:

assert <Bedingung>: <Fehlermeldung>;

Wir könnten die Vorbedingung beispielsweise so abändern:

```
assert ((n >= 0) && (n <= 20)): "Eingabe n nicht zulaessig";
```

Die Ausgabe der Fehlermeldung ist jetzt sehr viel lesbarer.

Wir haben an dieser Stelle das Konzept der Zusicherungen sowohl für die Vor- als auch die Nachbedingungen verwendet. Es gehört aber zum guten Programmierstil, nur die Nachbedingungen mit Hilfe von assert zu prüfen.

Jetzt werden wir Exceptions kennenlernen, die sich für die Einhaltung der Vorbedingungen besser eignen.

Konzept der Exceptions

Wenn ein Fehler während der Ausführung eines Programms auftritt, wird ein Objekt einer Fehlerklasse (Exception) erzeugt.

Da der Begriff Objekt erst später erläutert wird, stellen wir uns einfach vor, dass ein Programm gestartet wird, welches den Fehler analysiert und wenn der Fehler identifizierbar ist, können wir dieses Programm nach dem Fehler fragen und erhalten einen Hinweis, der Aufschluss über die FehlerquelleFehlerquelle gibt

Schauen wir uns ein Beispiel mit einer Problematik an, die uns schon einmal begegnet ist:

```
public class ExceptionTest{
   public static void main(String[] args){
      int d = Integer.parseInt(args[0]);
      int k = 10/d;
      System.out.println("Ergebnis ist "+k);
   }
}
```

Der Aufruf Integer.parseInt wandelt eine Zeichenkette in die entsprechende Zahl um.

Auf den ersten Blick ist kein Fehler erkennbar, aber ein Test zeigt schon die Fehleranfälligkeit des Programms.

Fehler über Fehler

Wir erhalten teilweise Laufzeitfehler bei den folgenden Eingaben:

```
C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionTest.main(ExceptionTest:5)

C:\>java ExceptionTest d
Exception in thread "main" java.lang.NumberFormatException: For input
string: "d"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at ExceptionTest.main(ExceptionTest:4)
```

Zwei Fehlerquellen sind hier erkennbar, die Eingabe eines falschen Datentyps und die Eingabe einer 0, die bei der Division durch 0 einen Fehler verursacht. Beides sind für Java wohlbekannte Fehler, daher gibt es auch in beiden Fällen entsprechende Fehlerbezeichnungen NumberFormatException und Arithmetic-Exception.

Wir Fragen uns: In welchen Situationen bzw. unter welchen Bedingungen stürzt das Programm ab?

Einfache try-catch-Behandlung

Die Syntax dieser Klausel sieht wie folgt aus:

```
try {
    <Anweisung>;
    ...
    <Anweisung>;
} catch(Exception e){
    <Anweisung>;
}
```

Die try-catch-Behandlung lässt sich lesen als: Versuche dies, wenn ein Fehler dabei auftritt, mache jenes.

Einsatz von try-catch (einfach)

Um unser Programm vor einem Absturz zu bewahren, wenden wir diese Klausel an:

```
try {
   int d = Integer.parseInt(args[0]);
   int k = 10/d;
   System.out.println("Ergebnis ist "+k);
} catch(Exception e){
   System.out.println("Fehler ist aufgetreten...");
}
```

Wie testen nun die gleichen Eingaben, die vorhin zu Fehlern geführt haben:

```
C:\>java ExceptionTest 2
Ergebnis ist 5

C:\>java ExceptionTest 0
Fehler ist aufgetreten...

C:\>java ExceptionTest d
Fehler ist aufgetreten...
```

Einen Teilerfolg haben wir schon zu verbuchen, da das Programm mit den fehlerhaften Eingaben zur Laufzeit nicht mehr abstürzt.

Mehrfache try-catch-Behandlung

Dummerweise können wir die auftretenden Fehler nicht mehr eindeutig identifizieren, da sie jeweils die gleichen Fehlermeldungen produzieren.

Um die Fehler aber eindeutig zu unterscheiden und behandeln zu können, lassen sich einfach mehrere catch-Blöcke mit verschiedenen Fehlertypen angeben:

```
try {
    <Anweisung>;
    ...
    <Anweisung>;
} catch(Exceptiontyp1 e1){
    <Anweisung>;
} catch(Exceptiontyp2 e2){
    <Anweisung>;
} catch(Exceptiontyp3 e3){
    <Anweisung>;
} finally {
    <Anweisung>;
}
```

Der Vollständigkeit halber wollen wir noch erwähnen, dass am Ende ein optionaler finally-Block folgen kann, der Anweisungen enthält, die in jedem Fall am Ende ausgeführt werden.

Einsatz von try-catch (mehrfach)

Wenden wir die neue Erkenntnis auf unser Programm an und erweitern es:

```
try {
   int d = Integer.parseInt(args[0]);
   int k = 10/d;
   System.out.println("Ergebnis ist "+k);
} catch(NumberFormatException nfe){
   System.out.println("Falscher Typ! Gib eine Zahl ein ...");
} catch(ArithmeticException ae){
   System.out.println("Division durch 0! ...");
} catch(Exception e){
   System.out.println("Unbekannter Fehler aufgetreten ...");
}
```

Wie testen nun die gleichen Eingaben, die vorhin zu Fehlern geführt haben:

```
C:\>java ExceptionTest3 2
Ergebnis ist 5

C:\>java ExceptionTest3 0
Division durch 0! ...

C:\>java ExceptionTest3 d
Falscher Typ! Gib eine Zahl ein ...
```

Den Fehlerort lokalisieren

Wenn wir in einem Programm zur Laufzeit einen Fehler erzeugen, dann liefert uns Java den genauen Ort. Hier haben wir beispielsweise eine Methode erzeugeFehler, die ganz offensichtlich zu einem Fehler führt:

```
public class ExceptionLokalisieren {
   public static void erzeugeFehler() {
      int erg = 3/0;
   }

   public static void rufeFehlerAuf() {
      erzeugeFehler();
   }

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

Das führt zur Laufzeit den folgenden Fehler:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at ExceptionLokalisieren.erzeugeFehler(ExceptionLokalisieren.java:5)
at ExceptionLokalisieren.rufeFehlerAuf(ExceptionLokalisieren.java:9)
at ExceptionLokalisieren.main(ExceptionLokalisieren.java:13)
```

Den Fehlerort lokalisieren

Sollten wir den Aufruf nun in der main-Methode durch einen try-catch-Block einschließen, geht diese Fehlermeldung zunächst verloren, kann aber durch die Methode printStackTrace wieder ausgegeben werden:

```
public class ExceptionLokalisieren {
   public static void erzeugeFehler() {
      int erg = 3/0;
   }

   public static void rufeFehlerAuf() {
      erzeugeFehler();
   }

   public static void main(String[] args) {
      try {
        rufeFehlerAuf();
       } catch (Exception e) {
         e.printStackTrace();
       }
    }
}
```

Fehler an Aufrufer weiterreichen

Eine Funktion liefert immer genau ein Ergebnis. Wir können allerdings zusätzlich noch erlauben, dass konkrete Fehler zurückgeliefert werden können. Wenn wir eine Funktion derart ausstatten wollen, erweitern wir die Funktionssignatur (im Beispiel aufgrund des Platzmangels ohne Parameterliste) wie folgt:

```
public static <Datentyp> name() throws Exception {
    // ...
    throw new Exception(<Fehlermeldung>);
}
```

Wenn uns jetzt beispielweise ein Eingabefehler auffällt, können wir die Funktion jederzeit durch throw mit einer Fehlermeldung beenden.

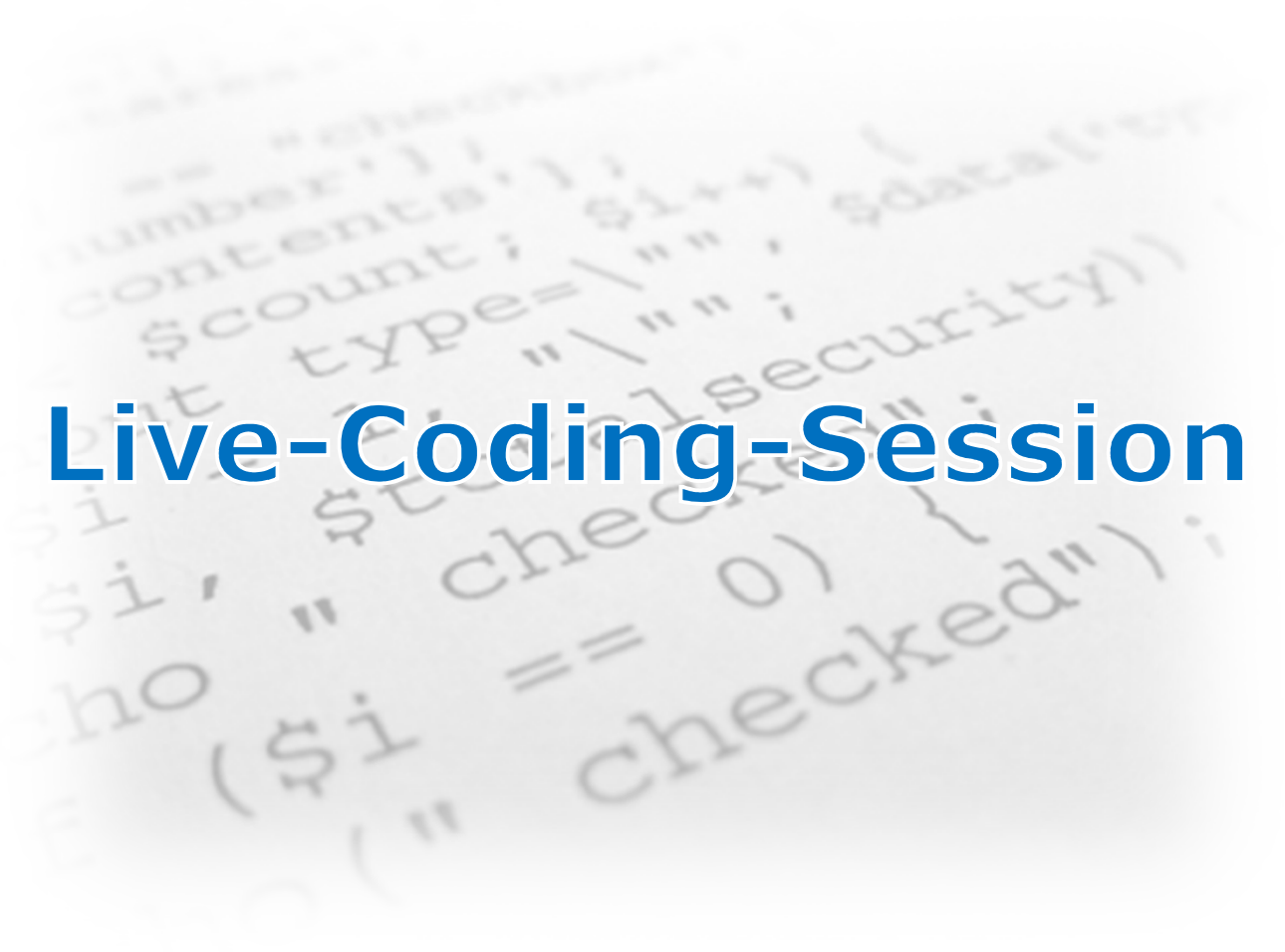
Fakultätsfunktion mit Exception

```
public class FakultaetException {
   /*
    * Fakultätsfunktion liefert für n=0 ... 20 die entsprechenden
    * Funktionswerte n! = n * (n-1) * (n-2) * ... * 1
    * (0! ist per Definition 1)
    *
    * Der Rückgabewert liegt im Bereich 1 .. 2432902008176640000
    *
    * Sollte eine falsche Eingabe vorliegen, so wirft das Programm
    * an den Aufrufer einen Fehler zurück.
    */
   public static long fakultaet(int n) throws Exception {
      // Ist der Wert außerhalb des erlaubten Bereichs?
      if ((n < 0) || (n > 20))
         throw new Exception("AUSSERHALB DES ZULAESSIGEN BEREICHS");

      long erg = 1;
      for (int i = n; i > 1; i--)
         erg *= i;
      return erg;
   }

   public static void main(String[] args) {
      for (int i = -2; i < 22; i++) {
         try {
            System.out.println("Fakultaet von "+i+" ist "+fakultaet(i));
         } catch(Exception e){
            System.out.println(e.getMessage());
         }
      }
   }
}
```

Testen der Fakultätsfunktion

Wir erhalten die hier etwas gekürzte Konsolenausgabe:

```
AUSSERHALB DES ZULAESSIGEN BEREICHS
AUSSERHALB DES ZULAESSIGEN BEREICHS
Fakultaet von 0 ist 1
Fakultaet von 1 ist 1
...
Fakultaet von 20 ist 2432902008176640000
AUSSERHALB DES ZULAESSIGEN BEREICHS
```



Live-Coding-Session

Fehlerhafte Berechnungen aufspüren

Beim Programmieren wird leider ein wesentlicher Teil der Zeit mit dem Aufsuchen von Fehlern verbracht.

Das ist selbst bei sehr erfahrenen Programmierern so und gerade, wenn die Projekte größer und unübersichtlicher werden, sind effiziente Programmiertechniken, die Fehler vermeiden, unabdingbar.

Sollte sich aber doch ein Fehler eingeschlichen haben, gibt es einige Vorgehensweisen, die die zum Auffinden benötige Zeit auf ein Mindestmaß reduzieren.

Textausgaben auf der Konsole

An der einen und anderen Stelle, konnten wir bereits erfolgreich Informationen auf der Konsole ausgeben. Jetzt wollen wir das etwas genauer spezifizieren. Es gibt prinzipiell drei Möglichkeiten, Daten auszugeben.

Der Unterschied zwischen der ersten und zweiten Variante ist lediglich, dass die Ausgabe mit einem Zeilenumbruch abgeschlossen wird:

System.out.print(<String>);

System.out.println(<String>);

Für die ersten beiden Varianten gilt: Ein <String> kann dabei aus verschiedenen Elementen von Zeichenketten und Ausdücken bestehen, die durch ein + verknüpft (konkateniert) werden:

```
boolean b = true;
int i = 10, j = -5;
double d = 41.6229;

System.out.println(d);
System.out.println("Text " + b);
System.out.println("Text " + i + " Text" + j);
```

Wenn wir als Ausdruck nicht nur eine Variable zu stehen haben, müssen wir darauf achten, dass der Ausdruck geklammert ist:

```
System.out.println("Text " + (i + 1));
```

Besondere Variante aus C/C++

Bei der dritten Variante handelt es sich um eine formatierte Ausgabe:

```
System.out.printf(<String>, <Datentyp1>, <Datentyp2>, ...);
```

Wir konstruieren eine Zeichenkette <String> und vergeben dort Platzhalter, beginnend mit dem Symbol %. Anschließend werden diese in der Reihenfolge eingesetzt, wie sie nach der Zeichenkette angegeben werden. Sollten diese nicht übereinstimmen, wird eine Fehlermeldung geliefert.

Die Platzhalter spezifizieren gleich den Datentyp, so stehen beispielsweise

- b für Boolean,
- d für Integer,
- o für Oktalzahl,
- x für Hexadezimalzahl,
- f für Gleitkommazahl,
- e für Exponentenschreibweise,
- s für Zeichenkette und
- n für Zeilenumbruch.

Beispiele

Schauen wir uns gleich Beispiele an:

```
System.out.printf("Text %d, %d, %b \n", i, j, b);
System.out.printf("Text >%10.3f< \n", d);
System.out.printf("Text >0123456789<", d);
```

Dass es sich um eine formatierte Ausgabe handelt, sehen wir im zweiten Beispiel. Hier reservieren wir für die Ausgabe der Zahl 10 Plätze und geben eine Genauigkeit von 3 Stellen nach dem Komma an.

Als Ausgabe erhalten wir:

```
Text 10, -5, true
Text >    41,623<
Text >0123456789<
```

Wir sehen in der zweiten Zeile, wie die 10 Plätze rechtsbündig aufgefüllt wurden.

In den Zeichenketten können auch Steuerzeichen (Escape-Sequenzen) vorkommen. Wir kennen bereits \n für einen Zeilenumbruch und \t für einen Tabulator.