

# ERSTE DATENSTRUKTUREN UND KOMPLEXE SCHLEIFEN

## Strukturiertes Speichern und Abrufen ähnlicher Daten in Arrays

Struktur und Inhalt des Kurses wurden 2012 von Markus Heckner entwickelt. Im Anschluss haben Alexander Bazo und Christian Wolff Änderungen am Material vorgenommen. Die aktuellen Folien wurden von Alexander Bazo erstellt und können unter der [MIT-Lizenz](#)  verwendet werden.

# AKTUELLER SEMESTERFORTSCHRITT (WOCHE 5)

Kursabschnitt	Themen				
Grundlagen	Einführung	Einfache Programme erstellen	Variablen, Klassen & Objekte		
	Kontrollstrukturen & Methoden	Arrays & komplexe Schleife			
Klassenmodellierung	Grundlagen der Klassenmodellierung				
Interaktive Anwendungen	Event-basierten Programmierung				
Datenstrukturen	Listen, Maps & die Collections				
Software Engineering	Planhaftes Vorgehen bei der Softwareentwicklung				
	Qualitätsaspekte von Quellcode				

# PINGO-QUIZ



<https://pingo.coactum.de/833782> 

# DAS PROGRAMM FÜR HEUTE

- 5-Minuten-Themen: break & continue, switch-Anweisung, Zufallswerte und eine Wiederholung des *Draw Loop*
- Verwaltung von Werten gleichen Typs in einer Variable: *Arrays*
- Verschachtelte Schleifen

# 5 MINUTEN FÜR break UND continue



Schleifen werden über Bedingungen und Zählervariablen im Schleifenkopf gesteuert. Im Rumpf der Schleife können Sie die gesamte Schleife durch den `break`-Befehl abbrechen. Eine einzelne Iteration wird über den `continue`-Befehl übersprungen.

**Beide Befehle werden für unterschiedliche Verwendungszwecke eingesetzt.**

## EINZELNE ITERATIONEN MIT `continue` ÜBERSPRINGEN

Der `continue`-Befehl bricht die aktuelle Iteration der Schleife ab und fährt mit dem nächsten Durchlauf fort. Die Zählervariable wird entsprechend der Anweisungen im Schleifenkopf angepasst. In diesem Beispiel werden alle *ungeraden* Zahlen eines im Schleifenkopf definierten Wertebereichs ausgegeben.

```
1 private void continueOnEven() {  
2     System.out.println("Counting from 1 to 10, skipping even numbers.");  
3     for (int i = 0; i < 10; i++){  
4         if(i % 2 == 0) {  
5             continue;  
6         }  
7         System.out.println(i);  
8     }  
9 }
```

## SCHLEIFEN FRÜHZEITIG MIT break ABBRECHEN

Der break-Befehl bricht die komplette Schleife ab und greift den Code in der darauf folgenden Zeile wieder auf. Im Beispiel wird die Schleife abgebrochen, sobald ein bestimmter Wert, hier 5, erreicht wird (ein eher sinnfreies Beispiel)

```
1 private void breakOnSentinel() {  
2     System.out.println("Break on 5");  
3     for (int i = 0; i < 10; i++) {  
4         if(i == 5) {  
5             break;  
6         }  
7         System.out.println(i);  
8     }  
9 }
```

## SENTINELS UND break

Mit Hilfe des `break`-Befehls können wir bei unklaren Abbruchbedingungen (z.B. bei Benutzereingaben) die Steuerung der Schleife in deren Rumpf verlagern. Werte, die zum Prüfen dieser internen Abbruchbedingung dienen, nennt man auch *Sentinel* bzw. *Wächterwert*.

```
1 private static final int SENTINEL = 0;
2 public void readAndSumNumbers() {
3     int total = 0;
4     int num = readInt("Bitte geben Sie eine Zahl ein (oder brechen mit '0' ab): ");
5     while (num != SENTINEL) {
6         total += num;
7         num = readInt("Wert eingeben: ");
8     }
9     System.out.println("Die Summe der eingegebenen Zahlen ist: " + total);
10 }
11 }
```

**Hinweis:** Die (fiktive) Methode `readInt` im Beispiel gibt den als Parameter übergebenen Text auf dem Bildschirm aus und erlaubt die Eingabe von Ganzzahlwerten durch NutzerInnen. Der eingegebene Wert wird als Rückgabewert der Methode an die aufrufende Stelle zurückgegeben.

# 5 MINUTEN FÜR DIE switch-ANWEISUNG

Mit der `if`-Abfrage wird eine Bedingung genau einmal geprüft. Die Klammern müssen einen booleschen Ausdruck enthalten. Je nach Ergebnis der Auswertung (`false` oder `true`) werden die angegebenen Befehle ausgeführt oder nicht. Die im `else`-Teil definierte Alternative wird ausgeführt, wenn die Bedingung **nicht** zutrifft.

```
1 // Wenn Bouncer sich nach vorne bewegen kann ...
2 if(bouncer.canMoveForward()) {
3     // ... tut er dies.
4     bouncer.move();
5 // Wenn er sich nicht nach vorne bewegen kann ...
6 } else {
7     // ... dreht er sich nach links.
8     bouncer.turnLeft();
9 }
```

**Hinweis:** In einer `if ... else`-Bedingung wird entweder die eine **oder** die andere Anweisungsgruppe (`{ ... }`) ausgeführt. Es spielt keine Rolle, ob durch die Ausführung der `if`-Befehle eine Situation eintritt, die - relativ zur Ausgangssituation - eine Ausführung der `else`-Anweisungsgruppe notwendig machen würde.

## ALTERNATIVEN MIT "CASCADING IF"

Komplexere Zusammenhänge und priorisierte Alternativen können auch durch verschachtelte if-Abfragen abgebildet werden:

```
1 if (score >= 90) {  
2     System.out.println("Note: 1");  
3 } else if (score >= 80) {  
4     System.out.println("Note: 2");  
5 } else if (score >= 70) {  
6     System.out.println("Note: 3");  
7 } else if (score >= 60) {  
8     System.out.println("Note: 4");  
9 } else {  
10    System.out.println("Note: 5");  
11 }
```

Sobald eine der else if-Bedingungen zutrifft, wird deren Anweisungsgruppe ({ ... }) ausgeführt und die restliche Prüfung abgebrochen. Das finale else ohne Bedingungsprüfung wird als *catch all*-Option für den Fall verwendet, dass keine der vorher notierten Bedingungen zutrifft.

## BEISPIEL: "CASCADING IF" UND WOCHENTAGE

```
1 int dayNum = readInt("Gib die Nummer des Wochentags ein: ");
2
3 if (dayNum == 6) {
4     System.out.println("Heute ist Wochenende (Samstag)! ");
5 } else if (dayNum == 7) {
6     System.out.println("Heute ist Wochenende (Sonntag)! ");
7 } else {
8     System.out.println("Heute ist ein Arbeitstag :(");
9 }
```

Solche und ähnliche Konstruktionen lassen sich auch mit der `switch`-Anweisung notieren!

## BEISPIEL: switch UND WOCHENTAGE

```
1 int dayNum = readInt("Gib die Nummer des Wochentags ein: ");
2 switch (dayNum) {
3     case 6:
4         System.out.println("Heute ist Wochenende (Samstag)! ");
5         break;
6     case 7:
7         System.out.println("Heute ist Wochenende (Sonntag)! ");
8         break;
9     default:
10        System.out.println("Heute ist ein Arbeitstag :(");
11        break;
12 }
```

Eine `switch`-Anweisung besteht aus der zu prüfenden Variable (hier: `dayNum`), den verschiedenen möglichen Fällen (`case`), Anweisungen die je nach `case` ausgeführt werden sollen und einem (optionalen) `default`-Fall, der ausgelöst wird, wenn keiner der notierten Fälle zutrifft. Die `break`-Befehle sorgen dafür, dass nach dem Finden und Abarbeiten des richtigen Falles die `switch`-Anweisung verlassen wird. Mit einer `switch`-Anweisung können Variablen vom Typ `byte`, `short`, `int`, `char`, `Enum` (Aufzählung) und `String` (Text) geprüft werden.

## FALLTHROUGH IN switch-ANWEISUNGEN: PROBLEME

Ein besonderes Feature von switch-Anweisungen ist der *Fallthrough*-Mechanismus. Wird ein passender Fall (*case*) gefunden, werden - ausgehend von diesem Fall - **alle** Anweisungen bis zum nächsten *break* ausgeführt. Ein "vergessenes" *break* kann also gravierende Konsequenzen haben.

```
1 int dayNum = readInt("Gib die Nummer des Wochentags ein: ");
2 switch (dayNum) {
3     case 6:
4         System.out.println("Heute ist Wochenende (Samstag)! ");
5         // Hier fehlt das "break"-Statement
6     case 7:
7         System.out.println("Heute ist Wochenende (Sonntag)! ");
8         break;
9     default:
10        System.out.println("Heute ist ein Arbeitstag :(");
11        break;
12 }
```

Das Programm gibt, im Fall von *i* = 6 sowohl den Text "Heute ist Wochenende (Samstag)!" als auch "Heute ist Wochenende (Sonntag)!" aus. **In diesem Fall ist das nicht gewollt!**

# FALLTHROUGH IN switch-ANWEISUNGEN: MÖGLICHKEITEN

Der *fallthrough*-Mechanismus kann auch bewusst genutzt werden:

```
1 int dayNum = readInt("Gib die Nummer des Wochentags ein: ");
2 switch (dayNum) {
3     case 1:
4     case 2:
5     case 3:
6     case 4:
7     case 5:
8         System.out.println("Heute ist ein Arbeitstag");
9         break;
10    case 6:
11    case 7:
12        println("Heute ist Wochenende!");
13        break;
14 }
```

Für die Werte 1 bis 4 werden keine Anweisungen notiert. Tritt einer dieser Fälle auf, fällt die Bearbeitung bis zur Bearbeitung des Falls 5 durch. Hier stehen dann die Anweisungen für alle vorangegangenen Fälle und das notwendige break um die Bearbeitung abzubrechen.

# 5 MINUTEN FÜR ZUFALLSWERTE

Häufig benötigen Sie bzw. werden Sie in Ihren Anwendungen zufällige Werte für numerische Eigenschaften ihrer Objekte benötigen:

- Zufällige Startpositionen für Objekte
- Zufällige Geschwindigkeiten für deren Bewegungen
- Entscheidungen auf Basis bestimmter Wahrscheinlichkeiten
- ...

Wirklichen *Zufall* gibt es in (den meisten) Computern nicht. Wir können aber Werte generieren, die für unsere Zwecke *zufällig* genug sind (Pseudozufallszahlen).

## ZUFALLSWERTE MIT DER Math-KLASSE

Die Methode `Math.random` generiert bei jedem Aufruf einen *zufälligen* double-Wert zwischen 0 (inklusive) und 1 (exklusive). Also einen Wert zwischen 0.0 und 0.99999.... Mit ein bisschen Mathematik können wir damit beliebige *zufällige* Ganzahlen berechnen:

```
1 double seed = Math.random(); // Zufallswert zwischen 0 und 0.99999...
2 int randomInt = (int) (seed * 42); // Zufallswert zwischen 0 und 41
```

Durch die Multiplikation und dem anschließenden *cast* des Ergebnisses entsteht eine zufällige Ganzzahl die irgendwo zwischen 0 und dem bei der Multiplikation verwendeten Wert liegt. Wir können den Prozess in eine Methode auslagern:

```
1 // Gibt einen zufälligen Ganzzahlwert zwischen 0 (inklusive)
2 // und "upperLimit" (exklusive) zurück
3 public int getRandomInt(int upperLimit) {
4     int value = (int) (Math.random() * upperLimit);
5     return value;
6 }
```

## ZUFALLSWERTE MIT DER Random-KLASSE

Für die einfacheren (und besseren) Generierung von Zufallswerten existiert in Java eine separate Klasse: Random. Deren Dokumentation können Sie [hier](#) nachlesen. Einige der möglichen Funktionen sind:

```
1 // Erstellen des Zufallsgenerators
2 Random randomGenerator = new Random();
3
4 // Erzeugen einer zufälligen Ganzzahl
5 int randomInt = randomGenerator.nextInt();
6
7 // Erzeugen einer zufälligen Ganzzahl von 0 bis zu einem bestimmten,
8 // exklusiven Limit (hier: 42)
9 int randomIntWithLimit = randomGenerator.nextInt(42);
10
11 // Erzeugen einer zufälligen Fließkommazahl zwischen 0 und 1.0
12 float randomFloat = randomGenerator.nextFloat();
13
14 // Erzeugen eines zufälligen Wahrheitswertes (true oder false)
15 boolean randomBoolean = randomGenerator.nextBoolean();
```

# 5 MINUTEN FÜR DEN *DRAW LOOP*

Animationen werden in der *GraphicsAp* durch Aktualisierung und Zeichnung von graphischen Objekten in der wiederholt aufgerufenen draw-Methode realisiert.



## BEISPIEL: ANIMATION EINES SICH BEWEGENDEN KREISES

```
1 public class FlyingBall extends GraphicsApp {  
2     // Instanzvariable für den Ball  
3     private Circle ball;  
4  
5     public void initialize() {  
6         setCanvasSize(500,500);  
7         // Erstellen des Balls  
8         ball = new Circle( 250,250, 50, Colors.RED);  
9     }  
10  
11    public void draw() {  
12        drawBackground(Colors.WHITE);  
13        // Bewegen des Balls  
14        ball.move(1, 1);  
15        // Zeichnen des Ball  
16        ball.draw();  
17    }  
18}
```

# ARRAYS

# EIN HÄUFIGES BILD IN UNSEREN ANWENDUNGEN

```
1 public class Animation extends GraphicsApp {  
2     private Circle circle1;  
3     private Circle circle2;  
4     private Circle circle3;  
5     private Circle circle4;  
6  
7     public void initialize() {  
8         circle1 = new Circle(50,50,50,Colors.RED);  
9         circle1 = new Circle(100,100,50,Colors.GREEN);  
10        // ...  
11    }  
12  
13    public void draw() {  
14        circle1.move(1,1);  
15        circle1.draw();  
16        circle2.move(1,1);  
17        circle2.draw();  
18        // ...  
19    }  
20 }
```

## DAS PROBLEM: VIELE OBJEKTE = VIELE VARIABLEN

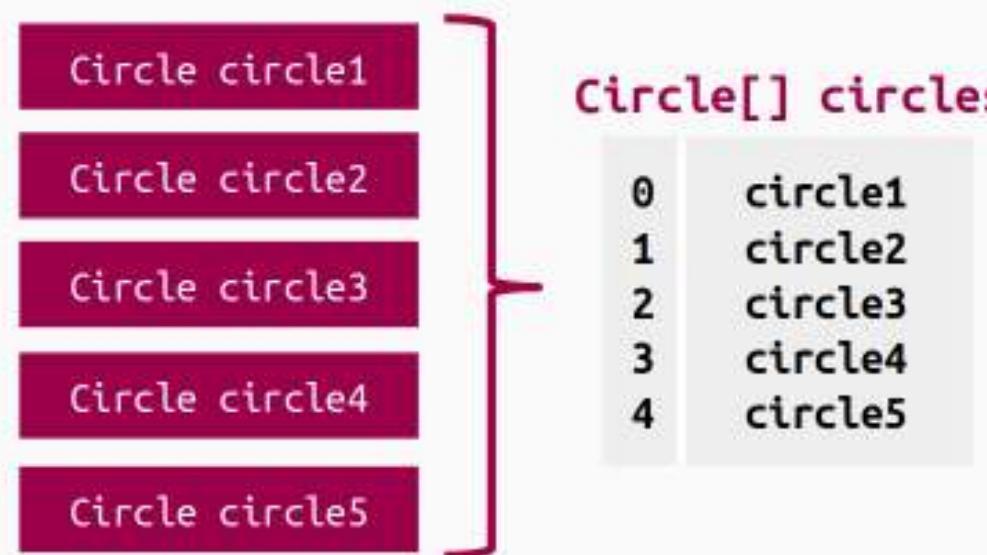
Wir benötigen sehr viele Variablen für das Speichern sehr ähnlicher Objekte:

- Simulation mit mehreren Bällen, die sich bewegen
- Figuren in einem Schachcomputer
- Einlesen mehrerer Werte, die zusammen verarbeitet werden sollen

Alle Objekte/Werte sind vom gleichen Typ. Auch die Variablen werden wahrscheinlich sehr ähnlich heißen. Für solche verwenden wir Arrays (Felder).

# DIE LÖSUNG: ARRAYS

Arrays erlauben die *zentrale* Verwaltung mehrere Objekte vom selben Datentyp über eine Variable.



Arrays gruppieren Objekte gleichen Typs unter einem gemeinsamen Namen. Jedes Element kann dabei über eine fortlaufende Nummer (Index, beginnt bei 0) angesprochen werden.

**Arrays selber werden in einer Variable gespeichert**, wir benötigen nur noch eine Variable für eine Vielzahl an Objekten.

## ARRAYS DEKLARIEREN UND INITIALISIEREN

Die Deklaration ähnelt Array-Variablen ähnelt dem bekannten Modus. Zur Unterscheidung des Feldes von einer *normalen* Variable werden eckige Klammern [] verwendet:

```
DATENTYP[] NAME;
```

Bei der Initialisierung wird das new-Schlüsselwort verwendet. Die Länge (Anzahl der *Plätze* für Objekte) muss an dieser Stelle angegeben werden und kann anschließend nicht verändert werden:

```
DATENTYP[] NAME = new DATENTYP[LENGTH];
```

Der lesende und schreibende Zugriff auf einzelne Elemente erfolgt über den Feldname, die eckigen Klammern und den jeweiligen Index (Position):

```
NAME[INDEX] = VALUE;
```

## BEISPIEL: ARRAY FÜR GANZZAHLEN

```
1 int[] numbers = new int[3]; // Feld für drei Ganzzahlen
2 numbers[0] = 42; // Die Indexierung beginnt bei 0, der erste Wert hat den Index "0"
3 numbers[1] = 1337;
4 numbers[2] = 101;
5
6 int sum = numbers[0] + numbers[1] + numbers[2];
7 System.out.println("Summe aller Werte in numbers: " + sum);
```

Die Kombination aus Variablenname und Index verhält sich bei Zuweisungen und Lesezugriffen genau wie eine einzelne Variable und kann auch so genutzt werden!

## ARRAYS UND SCHLEIFEN

Für den lesenden und schreibenden Zugriff auf Arrays werden häufig Schleifen und deren Zählervariablen (wenn kompatibel) verwendet:

```
1 private static final int MAX_NUMBERS = 3;
2 int[] numbers = new int[MAX_NUMBERS];
3
4 // ...
5
6 int sum = 0;
7 for(int i = 0; i < MAX_NUMBERS; i++) {
8     sum += numbers[i];
9 }
```

## ZUGRIFF AUF DIE ARRAY-LÄNGE

Die (feste) Länge eines Arrays können Sie nach dessen Initialisierung über die Eigenschaft `length` auslesen:

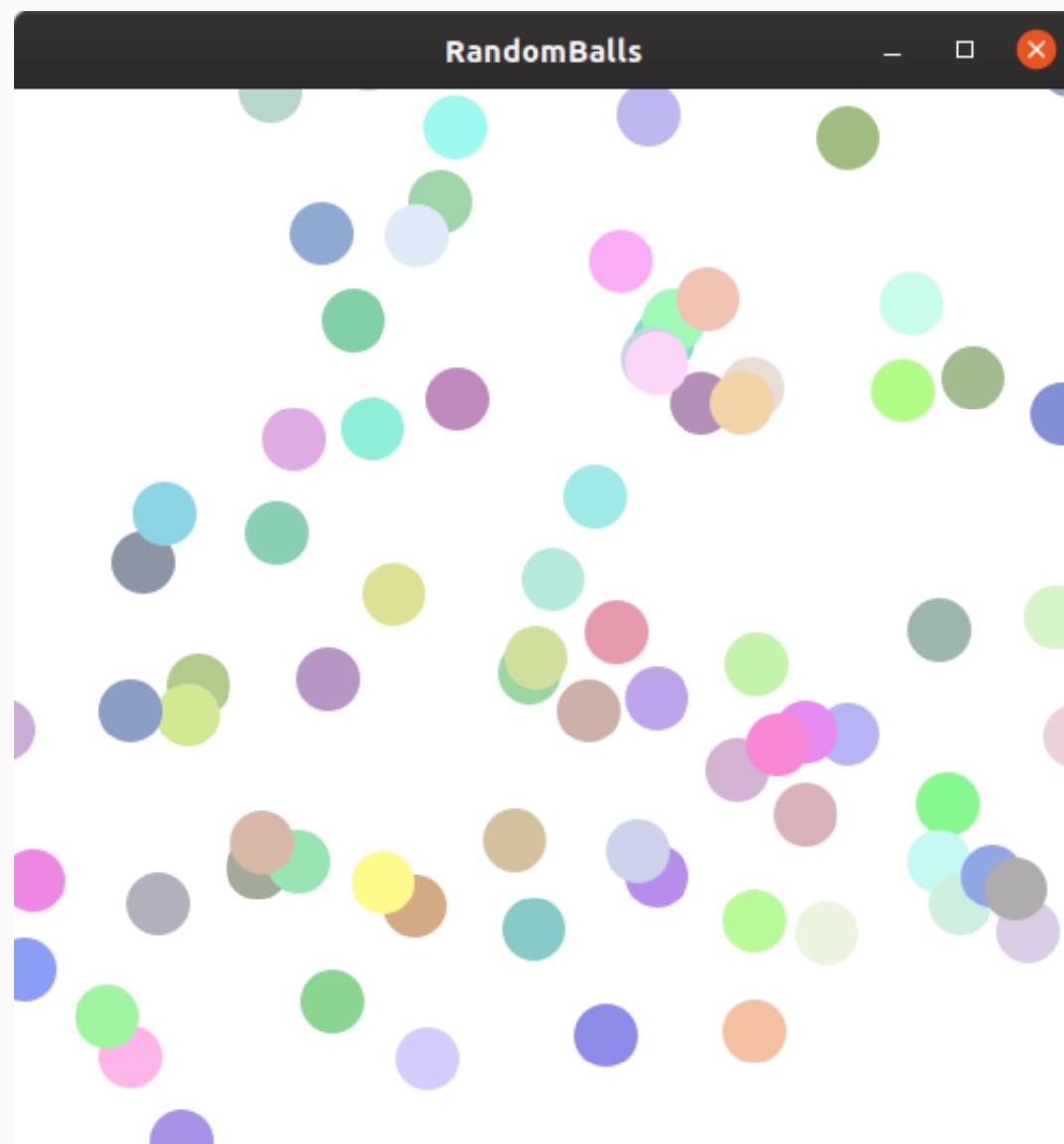
```
1 private static final int MAX_NUMBERS = 3;
2 int[] numbers = new int[MAX_NUMBERS];
3 int arrayLength = numbers.length;
4
5 // Die Eigenschaft "length" erleichtert die Iteration über Arrays
6
7 int sum = 0;
8 for(int i = 0; i < numbers.length; i++) {
9     sum += numbers[i];
10 }
```

**Hinweis:** Die Indizes eines Arrays werden als `int`-Wert abgebildet. Dadurch ergibt sich eine theoretische maximale Länge von 2.147.483.647. In der Praxis ist die maximale Länge wahrscheinlich geringer, da verschiedene *Laufzeitumgebungen* hier unterschiedliche Limits setzen.

## HÄUFIGE PROBLEME BEIM EINSATZ VON ARRAYS

- Halten Sie ihre Indizes im Blick. Java reagiert mit Fehlermeldungen (zur Laufzeit) wenn Sie versuchen, auf ein Element zuzugreifen, das nicht im Array steht (negativer Index oder Index, der über die Länge des Feldes hinausgeht).
- Die Werte im Array müssen dem bei der Deklaration der Array-Variable angegebenen Datentyp entsprechen.
- Bei der Initialisierung werden die Felder des Arrays mit *Default*-Werten belegt (primitive Datentypen) oder sind *leer* (`null`, bei Objekten).

# PRAXIS-BEISPIEL: BÄLLE MIT ZUFÄLLIGER STARTPOSITION

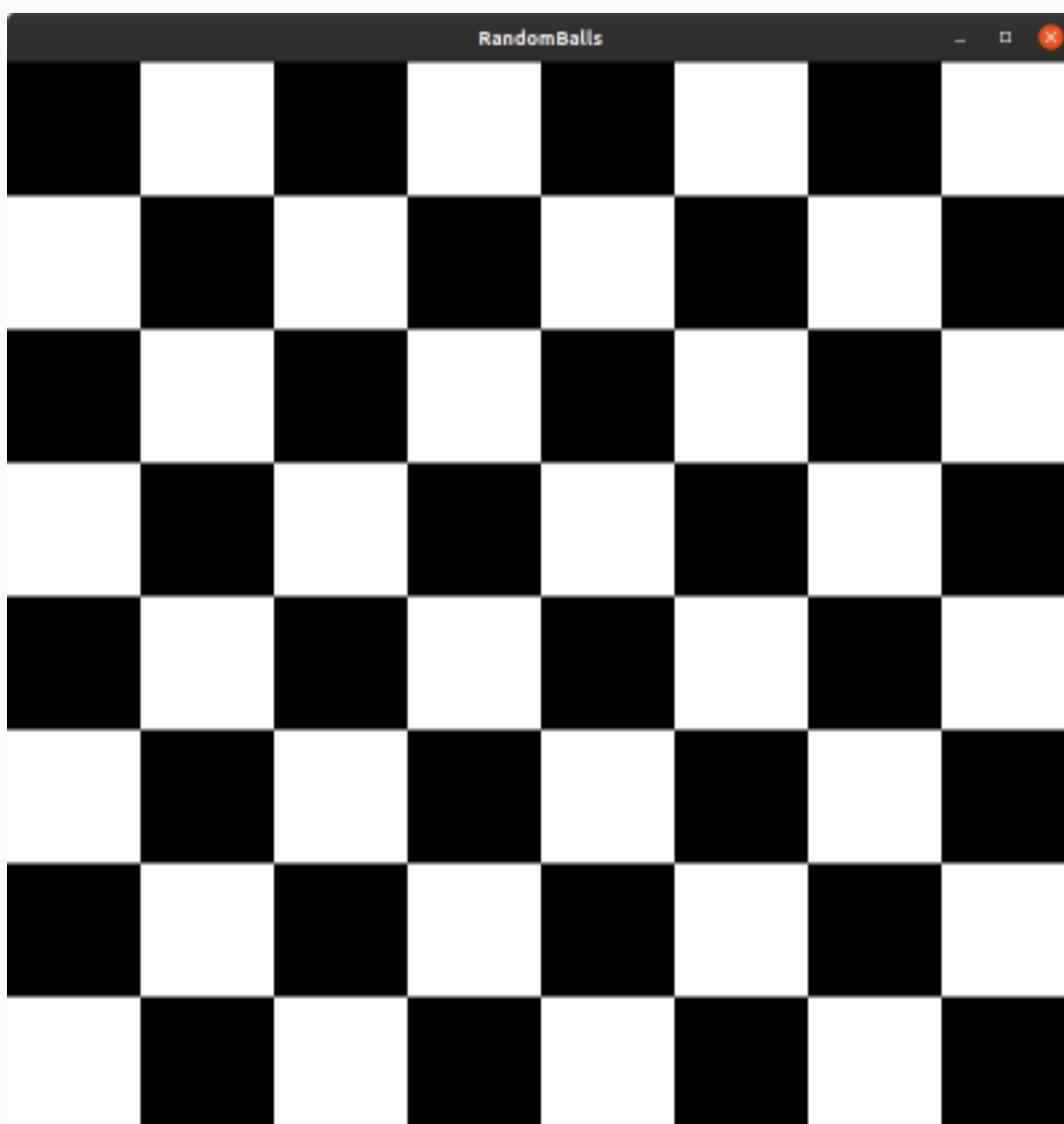


Wir erstellen eine Anwendung mit beliebig vielen, sich bewegenden, Bällen. Die Bälle werden in einem Array gespeichert und bei der Initialisierung zufällig auf der Zeichenfläche positioniert.

# VERSCHACHTELTE SCHLEIFEN

## 2D-DARSTELLUNGEN MIT RASTER

Eine häufige Aufgabe bei der Gestaltung von Anwendungen mit der *GraphicsApp*-Umgebung ist das Ausrichten von Elementen an einem zweidimensionalen Raster, z.B. für ein Schachbrett:



Die einzelnen Elemente dieser und ähnlicher Darstellungen orientieren sich an einem Gitter über die x- und y-Achsen. Andere Beispiele aus dem *GraphicsApp*-Kontext sind das Zeichnen Rasterdarstellungen (Vgl.: Bouncers Welt), das geordnete Positionieren von Elementen oder das Einteilen der Zeichenfläche in einzelne Bereiche.

**Für die Arbeit mit solchen Strukturen können wir verschachtelte Kontrollstrukturen einsetzen!**

## VERSCHACHTELTE SCHLEIFEN: ALLGEMEINES

Kontrollstrukturen können auch innerhalb anderer Kontrollstrukturen genutzt werden, d.h. eine Schleife kann eine andere Schleife enthalten. Generell gilt dabei: Enthält eine Schleife eine andere, so wird bei jeder Iteration der *äußeren Schleife [A]* die *innere Schleife [B]* komplett durchlaufen.

```
1 int i = 0;
2 int j = 0;
3
4 while (i<10) { // Äußere Schleife A
5     while (j<10) { // Innere Schleife B
6         System.out.println("Eine Nachricht aus der inneren Schleife");
7         j++;
8     }
9     // Setzt die Zählervariable für die innere Schleife für die nächste Iteration
10    // der äußeren Schleife zurück.
11    j = 0;
12    i++;
13 }
```

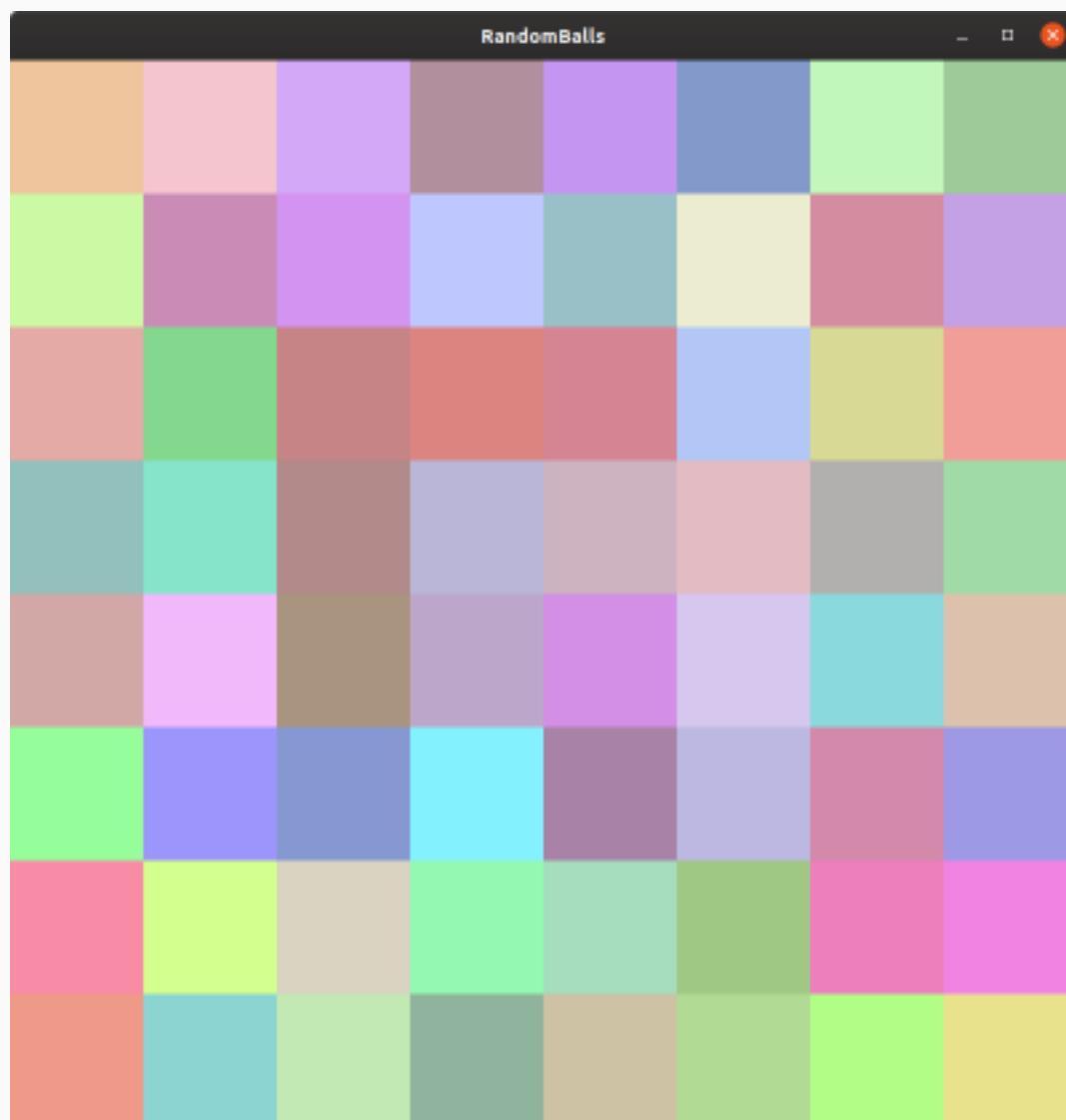
## VERSCHACHTELTE SCHLEIFEN: RASTERDARSTELLUNGEN

for-Schleifen eignen sich sehr gut dafür, Koordinatenpaare für die zweidimensionale Positionierung von Elementen zu generieren:

```
1 private static final GRID_WIDTH = 8;
2 private static final GRID_HEIGHT = 8;
3
4 // Iteration über alle Spalten
5 for(int x = 0; x < GRID_WIDTH; x++) {
6     // Zeilenweise Iteration über die Elemente einer Spalte
7     for(int y = 0; y < GRID_HEIGHT; y++) {
8         System.out.println("Aktuelle Koordinaten {" + x + "," + y + "}");
9     }
10 }
```

**Hinweis:** Die Dimensionen des Gitters (hier 8x8) stimmen in der Regel nicht mit der tatsächlichen Größe der Zeichenfläche in Pixeln überein (z.B. 800px x 800px). Aus den Koordinatenpaaren und der intendierten Größe der einzelnen Felder lassen sich aber die notwendigen Dimensionen berechnen.

# PRAXIS-BEISPIEL: RASTER MIT ZUFALLSFARBEN



Wir erstellen eine Anwendung, die ein Raster aus zufällig eingefärbten Farben darstellt.

# ZUSAMMENFASSUNG

- Der Ablauf von Schleifen lässt sich in deren *Rumpf* durch die Befehle `continue` und `break` steuern
- Einige `if ... else`-Anweisungen lassen sich sehr gut mit der `switch`-Anweisung ersetzen
- (Numerische) Zufallswerte lassen sich über die Funktion `Math.random()` oder Instanzen der Klasse `Random` erzeugen
- Werte gleichen Datentyps können in einem Array verwaltet werden
- Arrays haben eine feste Länge
- Elemente eines Arrays können über einen eindeutigen Index (Position im Feld) angesprochen werden

## VIELEN DANK FÜR IHRE AUFMERKSAMKEIT. WENN SIE MÖCHTEN, SEHEN WIR UNS IM ANSCHLUSS IN DER ZENTRALÜBUNG!

Fragen oder Probleme? In allgemeinen Angelegenheiten und bei Fragen zur Vorlesung wenden Sie sich bitte an Alexander Bazo ([alexander.bazo@ur.de](mailto:alexander.bazo@ur.de)). Bei organisatorischen Fragen zu den Studienleistungen und Übungsgruppen schreiben Sie bitte Florin Schwappach ([florin.schwappach@ur.de](mailto:florin.schwappach@ur.de)). Bei inhaltlichen Fragen zu den Übungen, Beispielen und Studienleistungen schreiben Sie uns unter [mioop@mailman.uni-regensburg.de](mailto:mioop@mailman.uni-regensburg.de).

## QUELLEN

Eric S. Roberts, *The art and science of Java: an introduction to computer science, New international Edition*, 1. Ausgabe, Pearson, Harlow, UK, 2014.