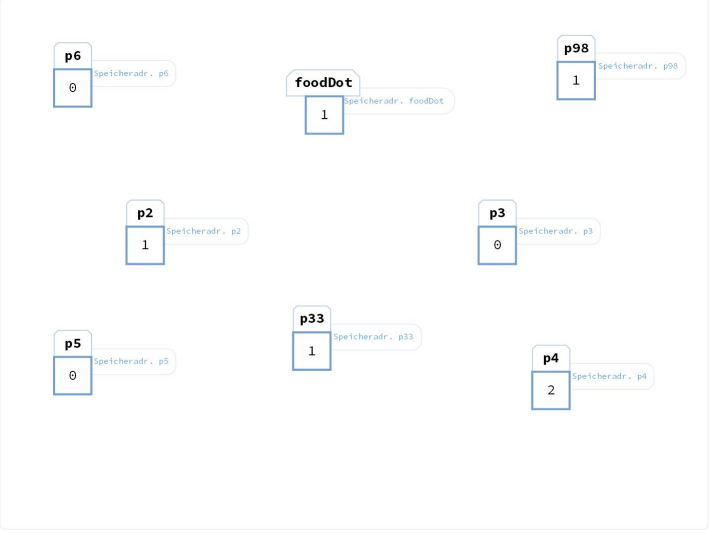
8. Arrays

8.1 Informationen gruppieren und strukturieren

In unserem Alltag haben wir es häufig eben nicht mit einzelnen Informationen zu tun, sondern mit großen Informationsmengen. Diese werden oft in Listen und Tabellen abgelegt bzw. dargestellt und verarbeitet. So werden bei Sportveranstaltungen wie Marathons oder Schirennen die Teilnehmenden in Listen mit Startnummern geordnet. Kundinnen und Kunden eines Unternehmens wird eine Kundennummer zugeordnet und diese sowie weitere Informationen werden in Tabellen oder Datenbanken gespeichert. Tabellen bzw. Listen finden sich beispielsweise aber auch bei Bestellungen und Rechnungen. Es können auch Einkaufs- oder To-Do-Listen sein oder Tabellen zur Verwaltung von Adressen, Büchern in Bibliotheken oder Zeitaufzeichnungen uvm.
Informationen in Listen oder Tabellen zu strukturieren, erleichtert es uns, den Überblick über größere Informationsmengen zu behalten. Insbesondere dann, wenn Listen bzw. Tabellen nach bestimmten Kriterien oder Werten sortiert oder gefiltert werden. Entsprechend der Reihenfolge der Anordnung werden oft die Einträge der Liste dann abgearbeitet.

In der Programmierung konnten wir mit Variablen bisher jeweils einzelne Werte speichern. Ausschließlich mit Variablen ist es aber nicht praktikabel größere Datenmengen, z.B. Alter oder Namen von 1000 Menschen, effizient zu verarbeiten. Denn dazu müssten wir 1000 Variablen anlegen und wären dadurch auch auf 1000 Einträge beschränkt. Auch das “Durchlaufen” dieser Variablen, um bestimmte Werte aufzufinden, gestaltet sich als sehr aufwändig. Kurzum, wir wären in der Programmierung ohne das Konzept einer Gruppierung wie Listen und Tabellen, wie wir das auch vom Alltag her kennen, ziemlich eingeschränkt. Eine Möglichkeit, wie Informationen in der Programmierung bzw. mit Processing gruppiert werden können, sind sogenannte Arrays (deutsch: Felder bzw. Datenfelder). Wie diese umgesetzt und eingesetzt werden, ist Inhalt dieses Kapitels.



Variablen im Speicher (Schema)

Beispiele für die Motivation von Arrays wären etwa

- das Speichern, Auslesen und Bearbeiten von 100 Spielfeld-Elementen beim Pacman (vgl. untenstehende Grafik: “board”)
- das Speichern des Alters von 50 Personen, um nach bestimmten Werten zu suchen (vgl. untenstehende Grafik: “ages”)
- das Speichern, Auslesen und Bearbeiten der Namen von 1500 Marathonläuferinnen und -läufern mit deren Startnummer (vgl. untenstehende Grafik: “runners”)

Idee von Listen im Speicher (Schema)

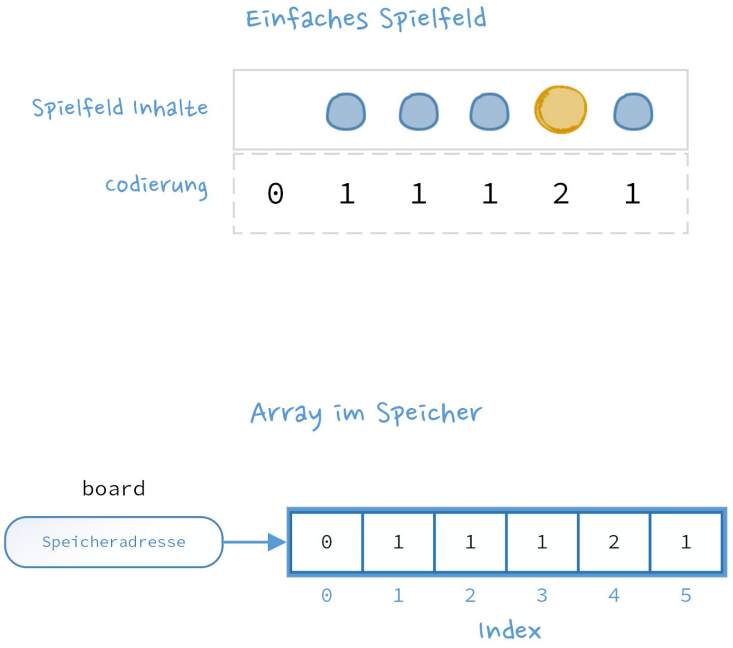
8.2 Was sind Arrays?

In Kapitel 3 wurden Variablen mit Gefäßen verglichen, welche je nach Typ Inhalte einer bestimmten Art fassen können. Nun könnte man auch Regale bauen, auf welche eine bestimmte Anzahl gleicher solcher Gefäße gestellt werden können. Das Regal wird mit einem Aufkleber mit dessen Namen versehen und das zweite Gefäß auf dem Regal könnte dann mit “im Regal mit dem Namen ‘beispielregal’, das zweite Gefäß” angesprochen werden. Arrays kann man sich als solche Regale vorstellen. Beim Erstellen wird festgelegt, welche Art von Gefäßen darauf gestellt werden können (der Regaltyp) und wie viele Gefäße darauf Platz haben (die Länge eines solchen Regals).

Ein Array kann man sich zudem vereinfacht auch als nummerierte Liste vorstellen. Bei vielen Sportbewerben, etwa Marathons, Schirennen, Eiskunstlauf oder Radrennen, usw., werden Startnummernlisten der Athletinnen und Athleten erstellt. Für jede Nummer der Startliste wird dann der Name einer Sportlerin bzw. des Sportlers zugelost oder in der Reihenfolge der Anmeldung zugewiesen. Auch das Endergebnis eines Bewerbes wird in einer Liste bzw. Tabelle dargestellt - jede Sportlerin bzw. jeder Sportler erreicht einen bestimmten Platz im Ranking (1. Platz, 2. Platz, etc.) - je nach erreichter Zeit, Punkteanzahl, Distanz bzw. weiterer Kriterien. Weitere bekannte Beispiele sind die Tabellen von Fußball-, Basketball- oder Hockey-Ligen.

Wie bildet man solche Strukturen - Arrays - in der Programmierung ab? Wie können in Processing nun mehrere zusammengehörige Werte geordnet gespeichert und verarbeitet werden?
In der Programmierung ist ein Array ein weiterer Datentyp. Das Besondere an diesem Datentyp ist, dass Array-Variablen nicht nur einen einzelnen Wert eines bestimmten Typs speichern können (wie Variablen das machen), sondern mehrere (wie bei einer Liste) Werte des gleichen Datentyps: zum Beispiel 2.000 Teilnehmerinnen und Teilnehmer eines Marathons, den Inhalt (Futterpunkt, Kraftpille, leeres Feld,...) von 99 Spielfeldelementen im Pacman-Spiel oder die Ergebnisse einer täglichen Blutdruckmessung eines Jahres.

Bei einem einfachen Pacman-Spielfeld mit 6 Elementen könnte das dann folgendermaßen aussehen:



Beispiel für ein Pacman-Spielfeld mit sechs Elementen: Grafische Spielfeldinhalte und deren Codierung (oben) sowie schematische Darstellung als Array mit Namen *board* im Speicher (unten)

Die einzelnen Werte in einem Array sind fortlaufend nummeriert. Dadurch können die

einzelnen Werte über diese fortlaufende Nummer, dem sogenannten **Index**, angesprochen werden. Da die Nummerierung der einzelnen Array-Elemente mit dem Index fortlaufend ist, können Arrays wunderbar mit Schleifen kombiniert werden, um zum Beispiel alle Elemente der Reihe nach abzufragen.

Achtung: Wie das in der Programmierung oft der Fall ist, beginnt der Index bei 0!

8.3 Arrays erstellen und verwenden

Deklaration von Arrays

Das Deklarieren von Arrays ähnelt dem der bereits bekannten Variablen:

```
float[] temperatures;
String[] startingList;
int[] ageList;
```

allgemein:
`datentyp[] arrayName;`

An erster Stelle steht auch hier der gewünschte Datentyp. Dieser Datentyp bestimmt bei Arrays, welche Arten von Information in den einzelnen Elementen des Arrays abgelegt werden können. Alle bisher vermittelten Datentypen können als Array-Datentyp eingesetzt werden (`int`, `float`, `boolean`, `String`, etc.).
Um zu kennzeichnen, dass es sich nun eben nicht um eine "herkömmliche" Variable, sondern um ein Array handelt, wird an den Datentyp ein eckiges Klammernpaar angehängt.

Dem folgen, wie gewohnt, der gewünschte Name des Arrays und der Strichpunkt als Abschluss der Anweisung.

Hinweis: Ein Array speichert eine bestimmte Anzahl von Werten **eines einzelnen Datentyps**!

Das Array

- `float[] temperatures;` etwa speichert Temperaturwerte, von denen jeder einzelne eine Gleitkommazahl ist.
- `int[] ageList;` wiederum beinhaltet Altersangaben in ganzen Jahren.

Achtung: Es ist **nicht** möglich, unterschiedliche Arten von Daten in einem einzigen Array zu kombinieren! So können in einem Array etwa nicht ein String als auch ein Integer gespeichert werden.

Bei der Deklaration eines Arrays mit zB. `int[] ageList;` ist nur bekannt, dass das Array `ageList` heißt, aber die Länge des Arrays ist noch nicht bekannt und auch das Array mit den einzelnen Arrayelementen ist im Speicher noch nicht angelegt. Die Speicheradresse des Arrays beinhaltet zu diesem Zeitpunkt den besonderen Wert `null`. Das heißt, dass die Speicheradresse nicht bekannt ist und somit Zugriffe auf die Arrayelemente nicht möglich sind.

Deklaration eines Arrays (Speicherschema)

Instanziieren und Initialisieren

Aber anders als bei Variablen handelt es sich bei dem Datentyp Array nun um eine Erweiterung der bereits bekannten Datentypen für Variablen - um ein Objekt, das aus mehreren einzelnen Elementen desselben Datentyps, zum Beispiel Integer, Float, String oder Boolean, besteht.

Daher kommt nach der Deklaration noch ein Schritt hinzu - das Array muss (im Speicherbereich) zusätzlich erzeugt, also erstellt, werden. Dieses Erzeugen nennt man beim Programmieren oft auch “Instanziieren”. Ist das Array instanziiert, erfolgt auch bei Arrays eine erste Wertzuweisung - die Initialisierung. Für die Instanziierung und Initialisierung von Arrays gibt es verschiedene Möglichkeiten:

Instanziierung mit fixer Länge:
Ein Array kann mit der gewünschten Anzahl von Elementen instanziiert werden. Die einzelnen Elemente des Arrays werden dabei automatisch mit Standardwerten initialisiert. Die Anzahl der Elemente eines Arrays wird die “Länge des Arrays” genannt.

```
int[] ageList;                  Deklaration des Arrays ageList
ageList = new int[5];           Instanziierung und implizite Initialisierung mit Nullen
```

Schemadarstellung im Speicher: Deklaration des Arrays ageList (links), Instanziierung und implizite Initialisierung des Arrays ageList mit Nullen (rechts) .

Es können Deklaration, Instanziierung und implizite Initialisierung auch zu einer einzigen Anweisung zusammengefasst werden:

```
int[] arrayName = new int[5];
```

Bei dieser Variante erfolgt auf der rechten Seite der Anweisung die Instanziierung durch Angabe, dass ein neues Integer-Array (`new int[5]`) angelegt werden soll. Innerhalb der eckigen Klammern wird die Größe des Arrays in Anzahl von Elementen angegeben. Das hier angelegte Array kann beispielsweise genau 5 Elemente speichern.
Die Größe des Arrays kann auch über eine Variable bestimmt werden:

```
int arraySize = 10;
int[] arrayName = new int[arraySize];
```

Wird für ein Array eine Größe bzw. Länge festgelegt, egal ob direkt mit einem Wert oder über eine Variable, so ist diese Länge nicht mehr veränderbar. Wird das Array über die Größe bzw. Länge instanziiert, dann wird jedem Element im Array automatisch ein Standardwert (engl.: default value) zugewiesen. Das bedeutet, ohne explizite Zuweisungen wird jedes Element des Arrays automatisch mit dem Standardwert initialisiert. In der Tabelle ist zusammengefasst, welche Standardwerte den Array-Elementen je nach Datentyp zugewiesen werden:

Datentyp	Standardwert
`int`	0
`float`	0.0
`boolean`	false
`char`	'\u0000'
`String`	null

Den einzelnen Elementen eines Arrays können natürlich nach dieser anfänglichen Initialisierung jederzeit später neue Werte zugewiesen werden.

Initialisierung durch Angabe der konkreten Werte

Eine andere Möglichkeit ein Array zu initialisieren ist es, direkt die gewünschten Werte anzugeben. Hier erfolgt die Instanziierung mittels `new int[]` und unmittelbar nachfolgend die Initialisierung durch Angabe der Werte in geschwungenen Klammern:

```
int[] array2 = new int[]{5, 3, 4, 5, 1};
```

Zwischen den eckigen Klammern bei der Instanziierung wird nun **nicht** die Länge des Arrays angegeben. Stattdessen werden in den nachfolgenden geschwungenen Klammern die gewünschten Werte gelistet. Die Anzahl der Werte in den geschwungenen Klammern bestimmt die Länge des Arrays. Auch hier kann man entweder alles in einer Zeile schreiben, wie oben, oder auf 2 Schritte aufteilen:

```
int[] array2;
array2 = new int[]{5, 3, 4, 5, 1};
```

In diesem Beispiel hat das Array 5 Elemente. Diese Länge ist später nicht mehr veränderbar. Die Werte der einzelnen Elemente selbst können hingegen jederzeit verändert werden. Zu beachten ist, dass die Reihenfolge der angegebenen Werte genau wie

angegeben übernommen werden. Im obigen Beispiel bedeutet dies, dass der Wert 5 dem ersten Array-Element zugewiesen wird, der Wert 3 dem zweiten, usw.

Eine einfachere bzw. noch kürzere Schreibvariante ist:

```
int[] array2 = {5, 3, 4, 5, 1};
```

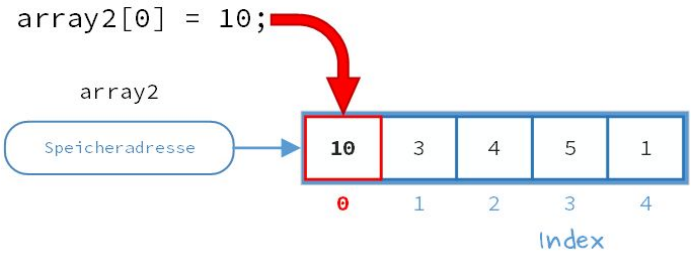
Diese Form der Initialisierung (mit impliziter Instanziierung) ist jedoch nur erlaubt, wenn Deklaration und Initialisierung in einer Zeile geschrieben werden, denn Processing führt im Hintergrund automatisch die Instanziierung durch. Wird die Deklaration getrennt, zeigt Processing eine Fehlermeldung an.

Auf Array Elemente zugreifen

Ist ein Array deklariert, instanziiert und initialisiert, kann der Wert jedes Elements im Array ausgelesen werden und es können natürlich auch jedem einzelnen Element des Arrays neue Werte zugewiesen werden. Um ein einzelnes Element in einem Array anzusprechen, werden der Name des Arrays und der Index des entsprechenden Elements benötigt. Dabei ist zu beachten, dass das erste Element eines Arrays in Processing immer den Index 0 hat.

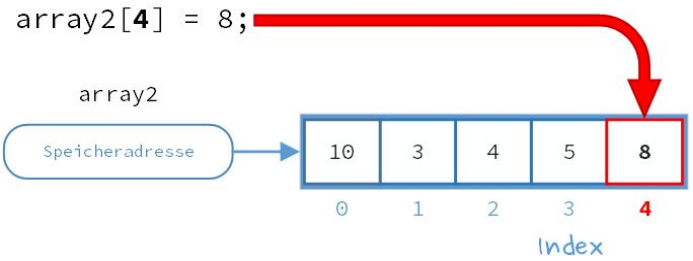
Soll dem ersten Element des Arrays `array2` der Wert 10 zugewiesen werden, schreibt man:

```
array2[0] = 10;
```



Innerhalb der eckigen Klammern nach dem Arraynamen wird nun also der Index jenes Elements geschrieben, das "angesprochen" werden soll (siehe nachfolgende Visualisierung). Die rechte Seite der Zuweisung wird genau wie bei Zuweisungen an Variablen programmiert.

Da der Index bei 0 beginnt, ergibt sich, dass das **letzte Element von insgesamt 5 Elementen den Index 4** hat:

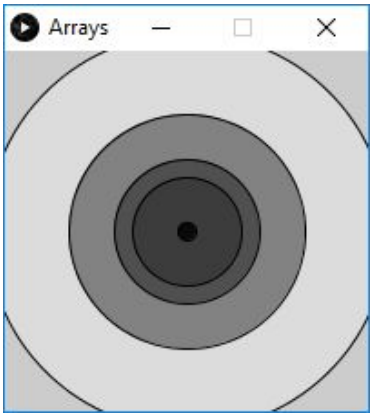


Die gespeicherten Werte können auch für Berechnungen oder einfache Zuweisungen verwendet werden:

```
int[] array3 = {5, 3, 4, 5, 1};
```

```
int min = array3[4];      // weist den 5. Wert von array3 der
                          // Variable min zu
int sum = array3[0] + array3[1];
                          // addiert den 1. und 2. Wert von array3
                          // und weist die Summe der Variable sum zu
```

Beispiel: Das folgende Code-Beispiel illustriert die Verwendung von Arrays anhand einer grafischen Anwendung. Der Programmcode zeichnet konzentrische Kreise. Die Größen der Kreise sind in diesem Beispiel im Array `circleSizes` gespeichert.




```
void setup() {
  size(200, 200);

  int[] circleSizes = {220, 130, 80, 60, 10};
  int x = width/2;
  int y = height/2;

  for (int i = 0; i < circleSizes.length; i++) {
    fill(circleSizes[i]);
    ellipse(x, y, circleSizes[i], circleSizes[i]);
  }

}
```

Der Ausdruck `circleSizes.length` liefert in Processing die Länge bzw. Größe des Arrays `circleSizes` - das heißt, die Anzahl der Elemente im Array `circleSizes` - als `int`-Wert. Daher kann mit der Abbruchbedingung `i < circleSizes.length` die Schleife genau so oft ausgeführt werden, wie es Elemente im Array `circleSizes` gibt. In jedem Schleifendurchlauf wird schließlich das Array `circleSizes` über den Index i ausgelesen und zuerst als Grauwert bei der Füllfarbe und danach auch für die Größe des Kreises verwendet.

Um also die Elemente eines Array (hier: `arrayName`) der Reihe nach zu “durchlaufen”, kann eine `for`-Schleife mit folgendem Aufbau verwendet werden (der `arrayName` ist dem tatsächlichen Namen des Arrays anzupassen):

```
for (int i = 0; i < arrayName.length; i++){
      print(arrayName[i]);    // gibt den Wert des “aktuellen”
                              // Elements (am Index i) im Array aus
}
```

Zuweisung eines Arrays an ein anderes Array

Ein Array kann auch einem anderen Array zugewiesen werden:

```
int[] arrayName = new int[]{5, 3, 4, 5, 1};
int[] array2 = arrayName;
```

Allerdings verhalten sich Arrays hier anders als Variablen!

Lesen Sie dazu folgenden Code durch und schreiben Sie sich die von Ihnen erwartete Ausgabe dieses Programmcodes auf:

```
void setup() {
  int[] firstArray = {5, 3};
  int[] secondArray = firstArray;
  firstArray[0] = 10;

  println("Erste Ausgabe");
  println(firstArray[0]);
  println(secondArray[0]);

  println(" ------------------- ");

  firstArray = new int[]{5, 3};
  secondArray = new int[2];
  secondArray[0] = firstArray[0];
  firstArray[0] = 15;

  println("Zweite Ausgabe");
  println(firstArray[0]);
  println(secondArray[0]);

}
```

Möglicherweise haben Sie erwartet, dass im ersten Block der `println()` - Ausgabe, in der Konsole folgendes ausgegeben wird:

```
Erste Ausgabe
10
5
 -------------------
```

Allerdings ist die tatsächliche Ausgabe:

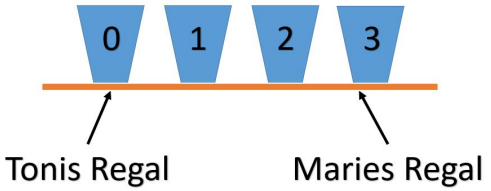
```
Erste Ausgabe
10
10
 -------------------
```

Was ist hier passiert? Arrays verhalten sich bei der Zuweisung anders als etwa die bisher bekannten einfachen Typen!

Im Gegensatz zu den bisher bekannten Datentypen wird bei einer Zuweisung mit Array nicht der ganze Inhalt kopiert. Vielmehr wird bei dieser Zuweisung nur die Speicheradresse des Arrays kopiert, der dahinter liegende Speicher hat dann zwei unterschiedliche Namen. Also zwei Variablennamen verweisen auf das gleiche Array, auf "denselben Inhalt".

Stellen Sie sich das ganze anhand der Analogie mit Gefäßen für Variablen vor. Ein Array könnte dann abgebildet werden als ein Regal mit Unterteilungen, auf welchem eine bestimmte Anzahl derselben Gefäße Platz hat, so wie in der folgenden Abbildung dargestellt.
Die Anzahl der Gefäße wird durch die Größe des Arrays angegeben, die Art der Gefäße durch den Datentyp der Array-Elemente.

Bei der Zuweisung wird nun nicht ein zweites Regal mit denselben Gefäßen und Inhalten erstellt! Sondern, es wird ein zweites Etikett mit dem "neuen" Array-Namen an das vorhandene Regal dazu geklebt. Wenn das Regal zuerst Toni gehört hat, könnten Sie zum Beispiel noch "Maries Regal" hinzufügen, dann würden das Regal dann beiden Personen, sowohl Toni als auch Marie "gehören" - sie würden es quasi teilen.



Zuweisung: Maries Regal = Tonis Regal (Schema)

Das bedeutet aber auch: Wenn der Inhalt eines Gefäßes (d.h. Array-Elements) für "Tonis Regal" bzw. der gesamte Inhalt des Arrays geändert wird, dann wird auch Maries Regal von der Änderung betroffen sein. Wird zum Beispiel das Gefäß 0 auf "Maries Regal" mit Wasser angefüllt, dann ist auch Tonis Gefäß 0 mit Wasser gefüllt, denn - beide Personen beziehen sich auf ein und dasselbe Regal und damit auch auf das identische Gefäß.

Anders ist es, wenn Sie das Array mit `new int[arraySize]` initialisieren. Hier legen Sie tatsächlich ein neues Regal an, bzw. neuen Speicherplatz.
Bei der Zuweisung `secondArray[0] = firstArray[0];` wird der Inhalt des Elements (hier mit dem Index 0) kopiert. Daher hat `firstArray[0] = 15;` keine Auswirkung am Array `secondArray`. Die Ausgabe vom zweiten `println()`-Block ist daher:

```
Zweite Ausgabe
15
5
```

Wann immer Sie ein Array einem anderen Array zuweisen, müssen Sie im Hinterkopf behalten, dass die Änderung eines Elements in einem Array auch das zweite Array automatisch betrifft, da hier KEINE Kopie des Arrays entstanden ist.
Dies gilt auch, wenn Sie Arrays als Parameter einer Funktion verwenden.

```
void setup() {
  int[] firstArray = {5, 3};
  println("before change: " + firstArray[0]);
  changeArray(firstArray);
  println("after change: " + firstArray[0]);
}

void changeArray(int[] a) {
  a[0] = 15;
}
```

Beim Ausführen dieses Programms wird die folgende Ausgabe erzeugt:

```
before change: 5
after change: 15
```

Beim Funktionsaufruf `changeArray(firstArray);` wird nicht das gesamte Array bei der Parameterübergabe kopiert, sondern nur die “Speicheradresse” übergeben. Die Zuweisung `a[0] = 15;` in der Funktion `changeArray()` findet daher nicht an einer Kopie statt sondern am Original, weswegen der Inhalt von `firstArray[0]` verändert wird.

Arrays kombinieren mit Schleifen

Der große Vorteil eines Arrays ist, dass es mehrere “zusammengehörige” Elemente geordnet speichert. Jedes einzelne Element kann über seinen Index angesprochen werden. Der Index ist eine fortlaufende Nummerierung der Elemente im Array. Das kann dazu genutzt werden, um mit Schleifen elegant und effizient auf alle Werte im Array der Reihe nach zuzugreifen. Im Gegensatz zu den bisher bekannten Datentypen, welche jeweils nur einen einzelnen Wert speichern konnten, ermöglichen Arrays eine Gruppierung von Werten. Dies soll das folgende Beispiel illustrieren:

Beispiel: Fünf Personen wurden nach ihrem Alter gefragt und dieses wurde jeweils notiert. Nun soll das größte Alter gefunden werden. Ohne Arrays, d.h. mit “einfachen” Variablen, würde das Beispiel folgendermaßen aussehen:

```
void setup() {
  int age1 = 18;
  int age2 = 30;
  int age3 = 24;
  int age4 = 19;
  int age5 = 18;
```

```
  int maxAge = age1;


 if (maxAge < age2) {
    maxAge = age2;
  }

  if (maxAge < age3) {
    maxAge = age3;
  }

  if (maxAge < age4) {
    maxAge = age4;
  }

  if (maxAge < age5) {
    maxAge = age5;
  }

  println(maxAge);
}
```

Zur Ermittlung der ältesten von fünf Personen sind vier `if`-Anweisung notwendig. Sind mehr Altersangaben zu vergleichen, dann sind die Fallunterscheidungen entsprechend zu erweitern. Dabei sind diese Abfragen immer ähnlich: “Falls das momentane maximale Alter kleiner ist als …, dann setze das maximale Alter auf …”.

Daher könnte diese Abfrage effizienter mit einer Schleife automatisiert wiederholt umgesetzt werden. Mit Variablen ist dies jedoch so nicht umsetzbar, da für den Computer zwischen den angelegten Variablen kein Bezug steht - auch wenn die Variablen für uns Menschen fortlaufend nummeriert sind.

Folgendes Codebeispiel ist von der Grundidee her richtig, würde aufgrund der fehlenden Gruppierung der Werte aber nicht funktionieren:

```
void setup() {
  int age0 = 18;
  int age1 = 30;
  int age2 = 24;
  int age3 = 19;
  int age4 = 18;

  int maxAge = age0;

  for (int i = 1; i < 5; i++) {
    if (maxAge < age[i]) {
      maxAge = age[i];
    }
  }
```

```
  println(maxAge);
}
```

Um eben diese notwendige Gruppierung zu realisieren, werden Arrays verwendet.

```
void setup() {
  int[] age = {18, 30, 24, 19, 18};

  int maxAge = age[0];

  for (int i = 1; i < 5; i++) {
    if (maxAge < age[i]) {
      maxAge = age[i];
    }
  }

  println(maxAge);
}
```

Dank des Arrays sind nun die einzelnen Elemente intern mit einem Index, also einer Zahl, versehen. Diesen Zusammenhang versteht auch der Computer. Damit lässt sich das Array sehr gut mit Schleifen kombinieren. In diesem Beispiel wird nun tatsächlich jedes einzelne Element des Arrays durchgegangen und mit dem momentan größten Alter verglichen.

Die Länge des Arrays, also die Anzahl seiner Elemente, wird meist direkt oder indirekt als Abbruchbedingung verwendet, um das gesamte Array elementweise zu durchlaufen. Die Länge eines Arrays kann ganz einfach mit folgender `int` Variable abgefragt werden:

```
arrayName.length
```

Das heißt, an den Array-Namen wird die Endung "`.length`" angefügt. In diesem Fall ist `length` ausnahmsweise kein reserviertes Wort, trotz der farblichen Unterlegung.

Mit dieser Variablen lässt sich nun ein Unterprogramm für die Maximumsuche schreiben, das für beliebige int-Arrays mit Mindestlänge 1 funktioniert.

```
void setup() {
  int[] age = {18, 30, 24, 19, 18};
  println(arrMax(age));
}

int arrMax (int[] a) {
  int max = a[0];
  for (int i = 1; i < a.length; i++) {
    if (max < a[i]) {
      max = a[i];
    }
  }
```

```
    return max;
  }
```

Die integrierte Variable `length` wird bei der Instanziierung eines Arrays automatisch angelegt und beinhaltet die Arraygröße.
Im obigen Beispiel wird etwa das Array `age` mit 5 Elementen, d.h. der Länge 5 angelegt. Wird das Programm gestartet, speichert Processing im Hintergrund als Länge den Wert 5 ab und wir können diesen Wert über `age.length` abfragen.
Stoppen wir das Programm, verändern das Array zB auf `int[] age = new int[3]` und starten es erneut, dann speichert Processing wieder im Hintergrund die veränderte Länge in `length`, also den Wert 3 für das erzeugte Array.

8.4 Typische Fehlermeldungen

Da in Zusammenhang mit der Verwendung von Arrays typische Fehlermeldungen auftreten können, wird im Folgenden als Unterstützung für die systematische Fehlersuche auf zwei solcher "klassischer" Processing Fehlermeldungen eingegangen.

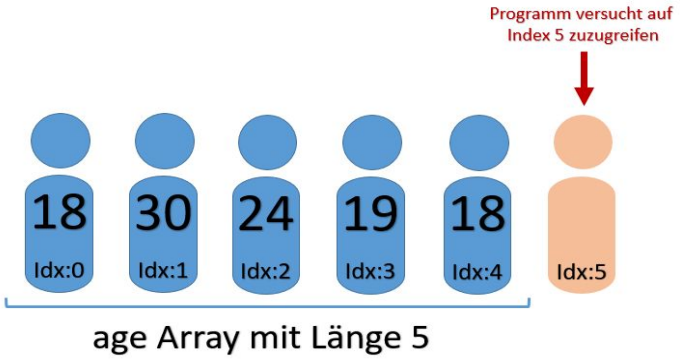
ArrayIndexOutOfBoundsException

Der Fehler "ArrayIndexOutOfBoundsException" wird angezeigt, wenn versucht wird, auf einen Index zuzugreifen, der nicht existiert. Dieser Fehler tritt typischerweise im Zusammenhang mit Schleifen auf, wenn die Abbruchbedingung über den Arraybereich hinausgeht.

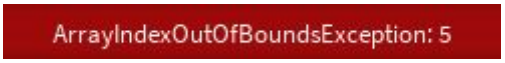
Beispiel: Hier sehen Sie noch einmal den Code für die Maximumsuche, leicht verändert.

```
void setup() {
  int[] age = {18, 30, 24, 19, 18};

  int maxAge = age[0];

  for (int i = 1; i <= age.length; i++) {
    if (maxAge < age[i]) {
      maxAge = age[i];
    }
  }

  println(maxAge);
}
```

Statt einem `<` (kleiner) wurde nun ein `<=` (kleiner gleich) verwendet. Die Konsequenz ist, dass der Index um eins zu weit gezählt wird. Der Wert von `i` erhält im letzten Schleifendurchlauf den Wert 5, da das Array `age` fünf Elemente hat. Im Schleifenrumpf wird daher versucht, auf `age[5]` zuzugreifen. Dieser Index existiert jedoch nicht, da das letzte Element des Arrays nur den Index 4 hat. Im unteren Bild wurde der Vorgang auch noch einmal grafisch dargestellt:



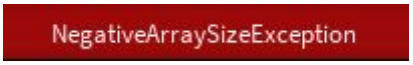
Führen Sie den oberen Code aus, wird Processing folgende Fehlermeldung anzeigen:



An diesem Fehler sehen Sie, dass ein ungültiger Indexzugriff stattgefunden hat und welcher Index ungültig war (hier: 5). Dieser Fehler wird allerdings erst zur Laufzeit angezeigt. Das bedeutet, das Programm muss zuerst gestartet und die entsprechende Zeile ausgeführt werden, damit dieser Fehler entdeckt werden kann.

NegativeArraySizeException

Der "NegativeArraySizeException" Fehler tritt dann auf, falls für die Array-Länge eine negative Zahl übergeben wird und versucht wird, auf irgendeine Weise mit dem Array zu arbeiten. Dieser Fehler tritt meist implizit auf, wenn etwa eine Arraygröße mit einer Variable angegeben wird, die durch fehlerhafte Berechnung einen negativen Wert annimmt.



Auch dieser Fehler taucht erst dann auf, wenn das Programm tatsächlich ausgeführt wird.

Beispiel für eine NegativeArraySizeException Ursache:

```
void setup() {
  int a = 5;
  int b = -1;

  int[] array = new int[a*b];

}
```

Beispiel:
Achtung, im folgenden Beispiel wirft Processing trotz des negativen Indexes keine NegativeArraySizeException, denn diese wird nur beim Erstellen eines Arrays mit negativem Index hervorgerufen. Stattdessen wird eine ArrayIndexOutOfBoundsException hervorgerufen. Bei letzterer liegt der Index außerhalb des gültigen Bereichs, was sowohl oberhalb als auch unterhalb sein kann.

```
void setup() {
  int[] ages = {18, 30, 24, 19, 18};
  int a = 3;
  int b = -1;

  int oneAge = ages[a*b];

}
```

8.5 Zweidimensionale Arrays

Bisher haben sie sogenannte 1-dimensionale Arrays kennen gelernt, die Sie sich wie Listen vorstellen können. 1-dimensionale Arrays enthalten nur eine bestimmte Art von Information, d.h. jeder Eintrag bezieht sich beispielsweise auf ein Alter, oder ein Gewicht usw.

Da als Typ der Elemente eines Arrays jeder Datentyp genommen werden kann, ist es auch möglich, als Datentyp der Elemente eines Arrays wieder ein Array zu nehmen. Dadurch kann in einem Element mehr als nur eine Information gespeichert werden. Sie können sich das wie eine Tabelle vorstellen: Beispielsweise haben Sie eine Tabelle von allen Teilnehmern eines Marathons. Jede Teilnehmerin bzw. jeder Teilnehmer hat eine eindeutige Nummer (den Index), das ist die erste Dimension. Von jedem Teilnehmer haben Sie aber noch weitere Informationen wie Alter, Größe, Gewicht, usw., die Sie speichern wollen - das wäre die zweite Dimension.

	0	1	2	3
0 (Alter)	30	53	24	37
1 (Größe in cm)	176	185	169	173
2 (Gewicht in kg)	70	78	58	68

Um das Alter von Teilnehmer/in mit der Nummer 2 herauszufinden, müssen Sie zuerst zur Spalte Nr. 2 gehen und dann in die Zeile 0 schauen. Hier können Sie ablesen, dass das Alter 24 Jahre ist.

Ein anderes Beispiel, das besonders wichtig für graphische Information ist, sind Koordinaten. Im Koordinatensystem haben Sie ebenfalls zwei Schritte auszuführen, um einen gewissen Punkt im Koordinatensystem zu finden. Sie müssen zuerst auf der x-Achse, die x-Koordinate finden und dann senkrecht von der x-Koordinate y-Schritte nach oben oder unten gehen. Erst dann haben Sie einen bestimmten Punkt im Koordinatensystem gefunden.

Deklarieren

Das Deklarieren von 2-dimensionalen Arrays ist den von 1-dimensionalen Arrays sehr ähnlich. Um festzulegen, dass es sich bei einem Array um ein 2-dimensionales Array handelt, verwenden Sie zwei eckige Klammernpaare:

```
int[][] arrayName;
```

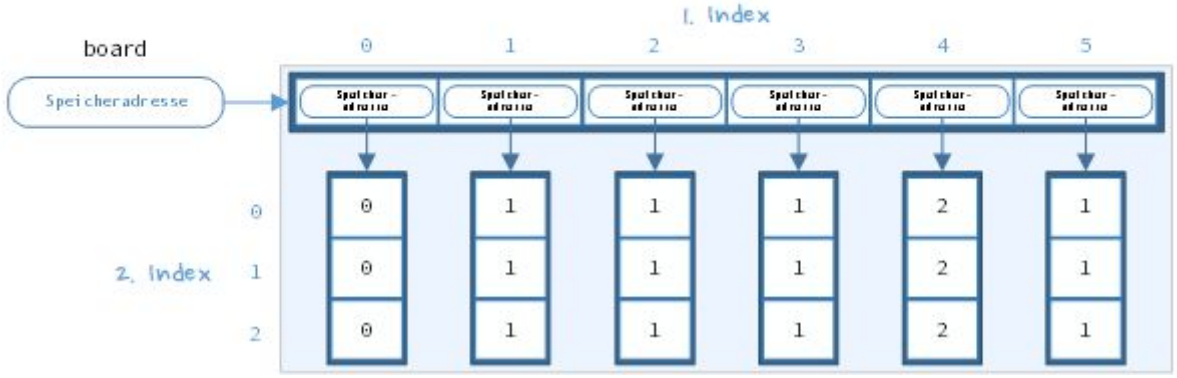
Diese Deklaration sagt aus, dass es sich bei `arrayName` um ein Array handelt, welches Arrays in seinen Elementen speichert. Die einzelnen Arrays wiederum speichern Werte vom Typ `int`.

Initialisierung und Zuweisung

Auch das Instanziieren ist dem vom 1-dimensionalen Array ähnlich. Will man beim Instanziieren vorerst nur die Größe des Arrays definieren, sieht das wie folgt aus:

```
int[][] board = new int[6][3];
```

In diesem Fall hat das Array sechs Arrays und die sechs Arrays können jeweils drei ganze Zahlen speichern.

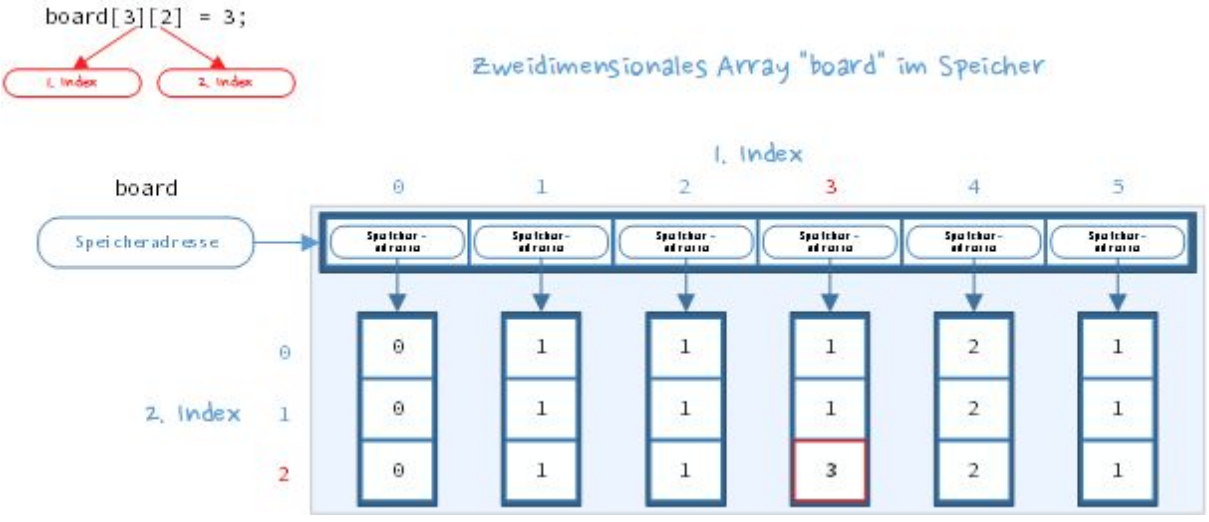


zweidimensionales Array (Speicherschema)

Mit

```
board[3][2] = 3;
```

kann man dem Array mit Index 3 seiner letzten Stelle (Index 2) den Wert 3 zuweisen:

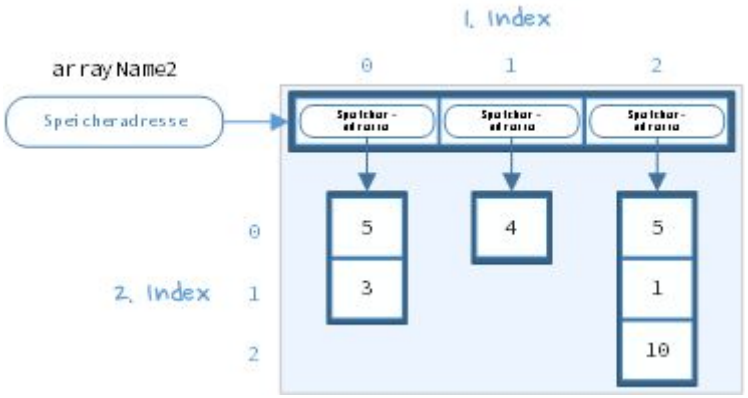


Wertzuweisung in einem zweidimensionalen Array (Speicherschema)

Werden Arrays auf diesem Weg initialisiert, dann haben alle “inneren” Arrays immer gleich viele Elemente, so wie eine einfache Tabelle in der Regel in jeder Zeile gleich viele Spaltenelemente beinhaltet. Das muss aber grundsätzlich nicht sein. Durch die Initialisierung mit direkter Wertübergabe ist es auch möglich, dass in einem Array unterschiedlich lange Arrays gespeichert werden können.

```
int[][] arrayName2 = new int[]{{5, 3}, {4}, {5, 1, 10}};
```

Das Array `arrayName2` speichert drei Arrays: Das erste innere Array besitzt zwei Elemente (`{5, 3}`), das zweite Array hat nur ein Element (`{4}`) und das dritte Array hat drei Elemente (`{5, 1, 10}`). Die geschwungenen Klammernpaare repräsentieren dabei jeweils ein Array.



Die Tabelle von Marathonläufern mit Alter, Größe und Gewicht wäre in einem Array beispielsweise so realisiert:

```
void setup() {
  int[][] marathon = {
    {30, 176, 70},
    {53, 185, 78},
    {24, 169, 69},
    {37, 173, 68}
  };

  println("Alter des dritten Teilnehmers: " + marathon[2][0]);

}
```

Dass für jedes innere Array eine separate Spalte verwendet wird, hat keinerlei Auswirkung auf den Inhalt des Arrays. Es dient lediglich der einfacheren Lesbarkeit und Vermeidung von Fehlern.

Betrachtet man die inneren Arrays als Elemente eines eindimensionalen Arrays, dann ist es verständlich, dass auch die inneren Arrays als Ganzes angesprochen und verändert werden können.

```
int[][] arrayName2 = new int[][]{{5, 3}, {4}, {5, 1, 10}};
arrayName2[0] = new int[]{8, 6, 10, 15};
```

Mit `arrayName2[0]` wird das erste Array angesprochen. Durch die Zuweisung (in der zweiten Zeile des Codes) wird das gesamte erste innere Array überschrieben. Es hat daher nicht mehr die Werte `{5, 3}` sondern die neuen Werte `{8, 6, 10, 15}`. Genauso ist es auch möglich, ein Element des zweidimensionalen Arrays, das ja ein eindimensionales Array ist, einer Array-Variablen zuzuweisen.

```
int[][] arrayName2 = new int[][]{{5, 3}, {4}, {5, 1, 10}};
int[] oneDimArray = arrayName[2];
```

In diesem Codeausschnitt wird der Variable `oneDimArray` das Array mit Index 2 aus `arrayName2` zugewiesen. Ein `println(oneDimArray[2])` würde hier beispielsweise die Zahl `10` auf der Konsole ausgeben.

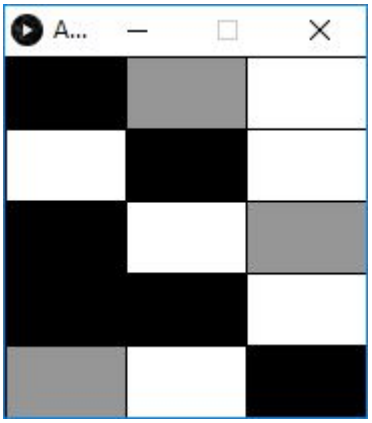
Beachten Sie, dass die Einteilung der Informationen in Zeilen und Spalten eigentlich nur eine Frage des verwendeten “Tabellen-”Denkmodells ist. Das bedeutet: Es kann prinzipiell genauso - etwa im Beispiel der Marathonläufer/innen - für jeden Läufer bzw. jede Läuferin eine Zeile verwendet werden und in Spalten werden dann die Angaben zu Alter, Größe und Gewicht gespeichert. Für das Array selbst ändert sich nichts, nur wie Sie es sich selbst vorstellen.

Zweidimensionale Arrays und Schleifen

Zweidimensionale Arrays sind besonders für die graphische Programmierung in 2D interessant. Denn Sie können sich das Zeichenfenster als ein zweidimensionales Array vorstellen, das x Pixel groß ist für die erste Dimension und y Pixel groß in der zweiten Dimension. Jedes dieser Pixel hat einen Grauwert, den man in einem 2D-Array abspeichern kann.

Um ein zweidimensionales Array mit Schleifen durchzugehen, benötigen Sie verschachtelte Schleifen. Die innere Schleife durchläuft die jeweils inneren Arrays, die äußere Schleife das äußere Array.

Beispiel: In einem zweidimensionalen Array sind Grauwerte gespeichert. Das Sketch-Fenster soll entsprechend der Größe des Arrays in Rechtecke unterteilt werden. Die Länge der ersten Dimension (d.h. die Anzahl der Arrays im "äußeren" Array) gibt an, wie viele Rechtecke es horizontal (x-Richtung) gibt. Die Länge der zweiten Dimension (Anzahl der Elemente in den inneren Arrays) gibt an, wie viele Rechtecke es vertikal (y-Richtung) gibt. Die Rechtecke sollen dann entsprechend der Graustufen im Array eingefärbt werden.




```
void setup() {
  size(180, 180);
  int[][] colors = {
    {0,   255, 0,   0,  150},
    {150, 0,   255, 0,  255},
    {255, 255, 150, 255, 0 }
  };
  int w = width / colors.length;
  int h = height / colors[0].length;

  for (int x = 0; x < colors.length; x++) {
    for (int y = 0; y < colors[x].length; y++) {
      fill(colors[x][y]);
      rect(x * w, y * h, w, h);
    }
  }
}
```

Das Lesen von Programmcode kann herausfordernd sein, wenn verschachtelte Schleifen kombiniert mit zweidimensionalen Arrays vorkommen. Oft hilft es dann - zusätzlich zur guten Dokumentation von Programmen - die Schleifen nicht "von oben nach unten" zu lesen sondern "von innen nach außen":

Betrachten wir dazu zuerst die innere Schleife und versuchen wir sie zu vereinfachen und zu erklären.

```
for (int y = 0; y < colors[x].length; y++) {
  fill(colors[x][y]);
  rect(x * w, y * h, w, h);
}
```

Diese Schleife verwendet die Variable `y` als Zählvariable und die Variable `x`, die sich in der Schleife nicht verändert. Sie können daher für die Variable `x` einen konstanten Wert einsetzen, z.B. 0.

```
for (int y = 0; y < colors[0].length; y++) {
  fill(colors[0][y]);
  rect(0 * w, y * h, w, h);
}
```

Diese Schleife ist jetzt leichter zu lesen. Sie wird so oft wiederholt, bis man am Ende des ersten inneren Arrays (`colors[0]`) angekommen ist. In diesem konkreten Beispiel ist die Länge vom ersten inneren Array gleich fünf. Danach wird die Farbe gesetzt. Diese wird aus dem ersten inneren Array an der Stelle `y` gelesen. Die Rechtecke werden anschließend von oben nach unten gezeichnet, da sich `y` erhöht. Zusammengefasst zeichnet also die innere Schleife eine Spalte an Rechtecken.

Das, was die innere Schleife macht, lässt sich auch in einer Funktion schreiben.

```
/**
 * Zeichnet von oben nach unten eine Spalte von Rechtecken
 *
 * @param colors  Array das Grauwerte der Rechtecke beinhaltet
 * @param xOffset Abstand vom linken Rand des Sketch-Fensters
 * @param w       Breite des Rechtecks
 * @param h       Höhe des Rechtecks
 */
void drawColumn(int[] colors, int xOffset, int w, int h){
    for (int y = 0; y < colors.length; y++) {
      fill(colors[y]);
      rect(xOffset, y * h, w, h);
    }
}
```

Die Funktion `drawColumn` nimmt als Parameter ein eindimensionales Array, eine x-Koordinate, und die Breite und Höhe des Kachels und zeichnet damit eine Spalte aus Rechtecken. Damit lässt sich die verschachtelte Schleife vereinfachen.

```
void setup() {
  size(180, 180);
  int[][] colors = {
    {0,   255, 0,   0,  150},
    {150, 0,   255, 0,  255},
    {255, 255, 150, 255, 0 }
  };
  int w = width / colors.length;
  int h = height / colors[0].length;

  for (int x = 0; x < colors.length; x++) {
    drawColumn(colors[x], x * w, w, h);
  }
}
```

Für jedes Array in `colors` wird eine Spalte gezeichnet, das um eine Rechtecksbreite verschoben ist.