TEKO
SCHWEIZERISCHE FACHSCHULE

# Objektorientiertes Programmieren 2

## Dynamische Datenstrukturen

Stephan Kessler, BSc in Systems Engineering

Kontakt: stephan.kessler@edu.teko.ch

# Vorbereitung

Lesen Sie folgende Abschnitte im Buch ***Sprechen Sie Java?*** als Vorbereitung für dieses Thema. Beachten Sie, dass bei Angaben von Unterkapiteln nur diese auch gelesen werden müssen:

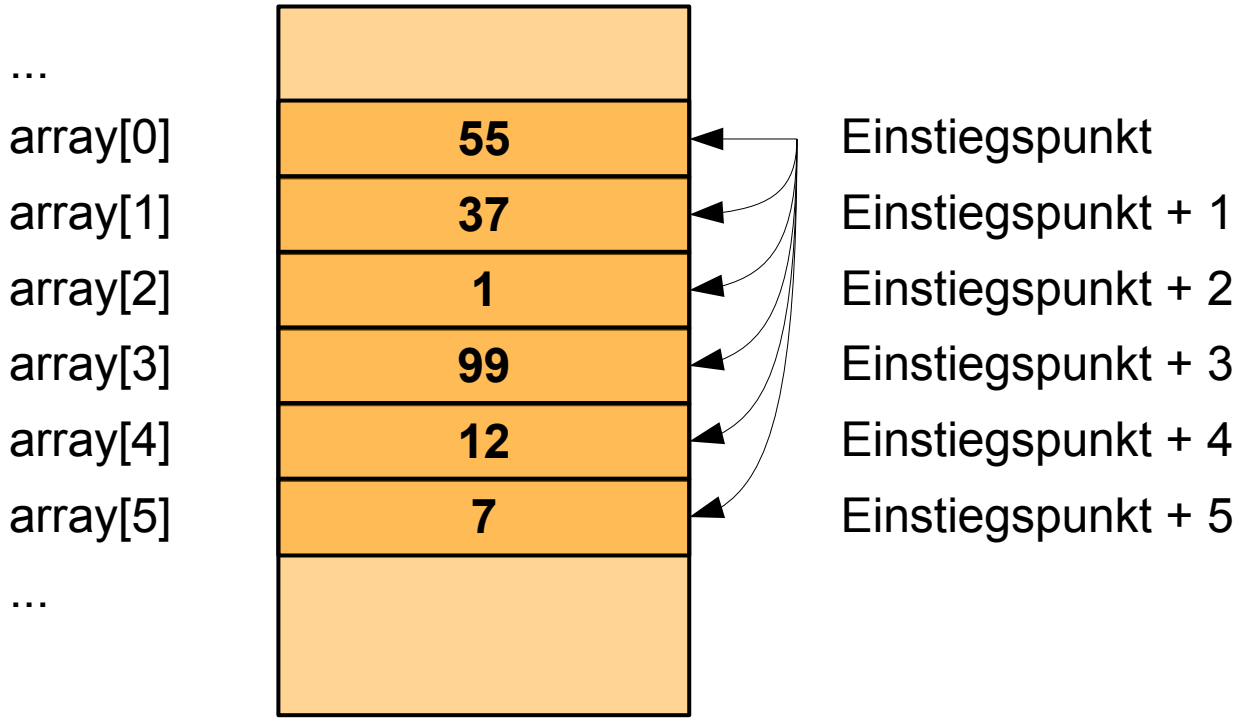
- Dynamische Datenstrukturen
  (S. 165-178)

# statische Arrays

```
int[] array = new int[100];
```

Bis anhin haben Sie statische Arrays verwendet um Werte und Referenzen von Objekten des gleichen Typs abzuspeichern. Dabei geben Sie wie gewohnt beim Erstellen des Arrays zugleich die Grösse des Arrays an.

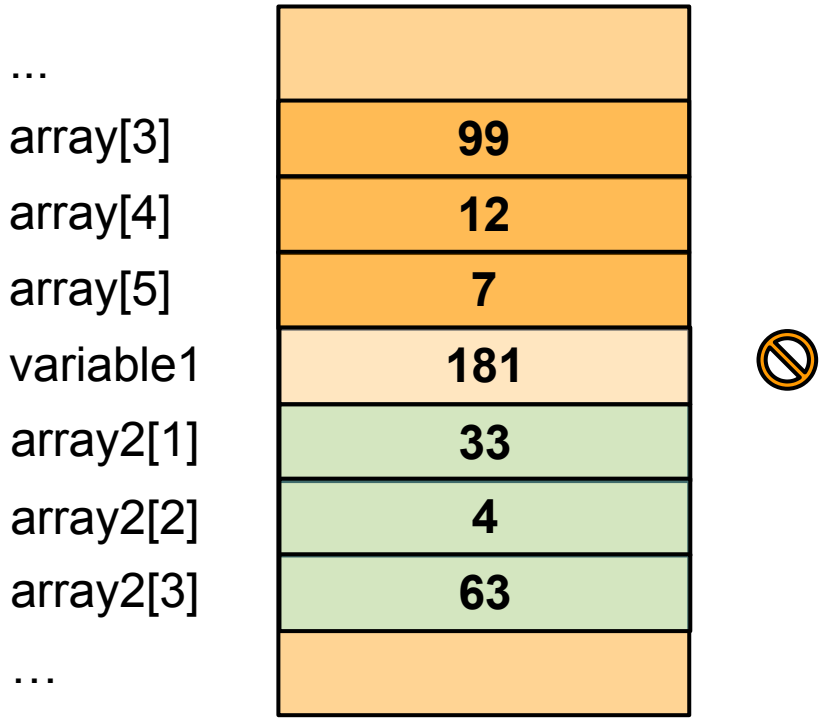
# statische Arrays




Statische Arrays weisen eine einfache Struktur im Speicher auf. Die Elemente des Arrays liegen dabei direkt aufeinander.

Ein Zugriff auf den Speicher geschieht dabei im Hintergrund über eine Adresse (Speicheradresse). Da das Array die erwähnte Struktur aufweist, kann auf ein gewünschtes Element **direkt**, und ohne zusätzliches «Nachschauen», zugegriffen werden.

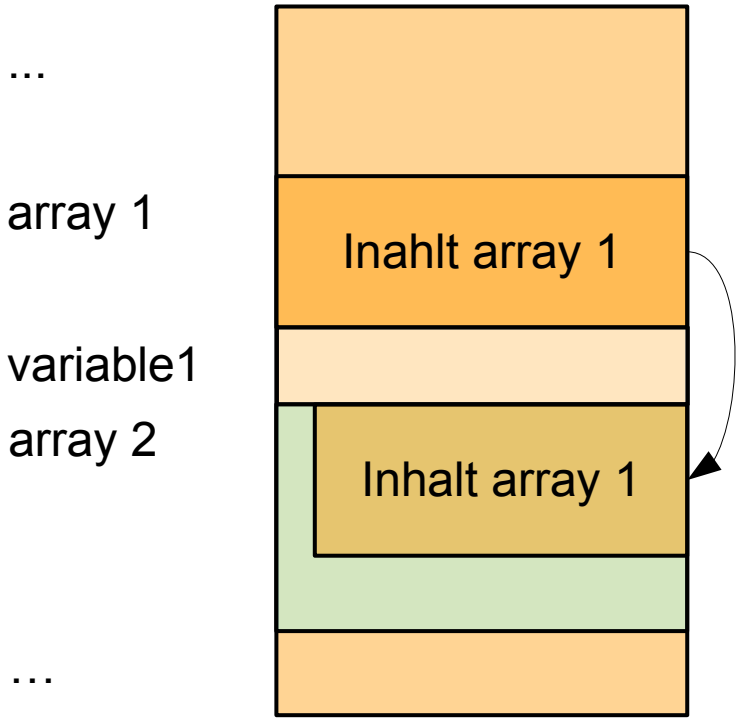
# statische Arrays




Um jedoch einen Speicherbereich verwenden zu können, muss dieser Bereich reserviert werden. Dies geschieht in Java automatisch durch das Erstellen einer Variable oder eines Arrays.

Da nicht garantiert ist, dass direkt nach dem Array ein freier Speicherbereich vorhanden ist, kann ein statisches Array nicht erweitert werden, ohne die Eigenschaft des direkten Zugriffes zu verlieren. (Es müsste im Speicher vermerkt werden, dass dort ein «überspringen» durchgeführt werden muss. Diese Information müsste aber zuerst gelesen werden, was ein weiterer Schritt bedeutet und somit keinen direkten Zugriff darstellt.)

# statische Arrays




Ein statisches Array stellt eine optimale Variante da, um auf eine bestimmte Position (Index) zuzugreifen.

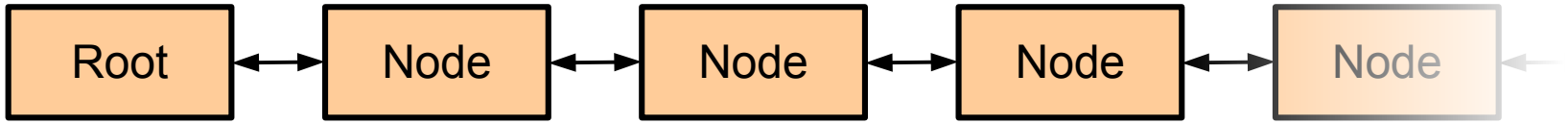
Möchte man jedoch über mehrere Zyklen weitere Elemente dem statischen Array hinzufügen, muss dazu ein neues Array mit mehr Speicherinhalt erstellt und das Ursprungs-Array in das neue Array kopiert werden.

# dynamische Datenstrukturen

```
int[] array = new int[100];
int[] array2 = new int[101];
for(int i = 0; i < array.size; i++){
array2[i] = array[i];
...
```

Um diesen Vorgang programmiertechnisch zu vereinfachen, kann eine Klasse dazu erstellt werden, welche den beschriebenen Prozess automatisiert. Diese Klasse kann als **dynamische Datenstruktur** betrachtet werden.

# Aufgabe




Es gibt jedoch auch weitere Varianten um dynamische Datenstrukturen zu erstellen, ohne dabei statische Arrays zu verwenden. Lösen Sie dazu die Praktikumsaufgaben 1 und 2.

# generische Typen

```
public class Box {
      private Object object;

      public void set(Object object) {
          this.object = object;
      }

      public Object get(){
          return object;
      }
}
```

Bis anhin haben Sie in Java den Typ einer höheren Klasse (Elternklasse) verwendet um den Typ des zu referenzierenden Objektes flexibel zu halten. Im oberen Code können Sie z.B. über die Methode *set()* jeglichen Typ von Objekt übergeben, da als Objekttyp *Object* erwartet wird.

Über die Methode *get()* können Sie das entsprechende Objekt wieder aufrufen. An dieser Stelle müssen Sie jedoch stets wissen, welcher Typ von Objekt Sie erhalten, um dieses korrekt zu verwenden. Eine falsche Interpretation des Typs würde erst zur Laufzeit durch eine Exception aufgedeckt. Solche Fehler können daher bei komplexeren Strukturen erst viel später entdeckt werden.

# generische Typen

```java
public class Box {
	private Object object;

	public void set(Object object) {
	    this.object = object;
	}

	public Object get(){
	    return object;
	}
}
```

```java
public class Box<T> {
	private T object;

	public void set(T object) {
	    this.object = object;
	}

	public T get(){
	    return object;
	}
}
```

Um diese Fehler frühzeitig zu erkenn (zur Compile-Zeit), stehen Ihnen in Java sogenannte generische Typen zur Verfügung. Dadurch geben Sie jeweils an, welchen Typ Sie übergeben wollen.

Im oberen Code wird anhand des Box-Beispiels die Implementierung von generischen Typen dargestellt.

# generische Typen

```
public class Box<T> {
      private T object;

      public void set(T object) {
          this.object = object;
      }

      public T get(){
          return object;
      }
}
```

Dazu schreiben Sie nach dem Klassennamen zwischen < > den Namen des generischen Typs. Hierbei wird üblicherweise ein Buchstabe verwendet.

# generische Typen

```
public class Box<T> {
      private T object;

      public void set(T object) {
          this.object = object;
      }

      public T get(){
          return object;
      }
}
```

Nun können Sie diesen Typ in der gesamten Klasse wie ein normaler Typ entsprechend verwenden.

# generische Typen

```
…
Box<String> box1 = new Box<String>();
Box<Integer> box2 = new Box<Integer>();
Box<Box<String>> box3 = new Box<Box<String>>();

box1.set("Hallo Welt!");
box2.set(4);
box3.set(box1);

String text = box1.get();
Integer zahl = box2.get();
String text2 = box3.get().get();
…
```

Bei der Instanziierung folgt nun der zu erwartende Typ nach dem Klassenname, zwischen <>.
Dadurch nimmt die Klasse den Typ auf und verwendet ihn, wie im vorherigen Code beschrieben.

Verschachtlungen sind dabei auch möglich.

# generische Typen

```
public class Box<T, V> {
      private T object;
      private V value;

      public void set(T object, V value) {
          this.object = object;
          this.value = value;
      }

      public T get(){
          return object;
      }
}
```

Ihnen steht auch die Möglichkeit zur Verfügung, mehrere generische Typen in einer Klasse zu verwenden. Dabei werden diese durch ein Komma getrennt. Bei der Instanziierung werden die Typen dann auch durch ein Komma getrennt.

# Aufgabe

```java
public class Box<T, V> {
	private T object;
	private V value;

	public void set(T object, V value) {
	    this.object = object;
	    this.value = value;
	}

	public T get(){
	    return object;
	}
}
```

- Implementieren Sie in Ihrer *NodeList*- und *Node*-Klasse (aus dem Praktikum) generische Typen, so, dass der Benutzer den Typ der Listen-Elemente angeben kann und diese nicht mehr von «*Object*»-Referenzen gehalten werden.
- Testen Sie Ihre Implementierung mit dem Praktikum 2.

# dynamische Datentypen

- ArrayList — Benutzt Array um Daten zu halten.
- Vector — Ähnlich wie ArrayList, jedoch ThreadSafe.
- LinkedList — Verkette Elemente (wie *NodeList*).
- HashMap — Verbindet Elemente durch «Schlüssel».
- Hashtable — Ähnlich wie HashMap, jedoch ThreadSafe.

Java bietet Ihnen durch die Standardbibliothek eine Sammlung an dynamischen Datentypen. Diese besitzen jedoch verschiedene Eigenschaften und werden daher in unterschiedlichen Situationen verwendet.

# Aufgabe

## Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

**All Implemented Interfaces:**

`Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

**Direct Known Subclasses:**

`AttributeList, RoleList, RoleUnresolvedList`

- Lesen Sie sich in die Java-Doc-Seite von *ArrayList* ein.
- Führen Sie das Praktikum 3 aus.

# ForEach

```
…
ArrayList<String> list = new ArrayList<String>();
list.add("Hallo");
list.add("Welt");
list.add("!");

for(String eintrag : list){
     System.out.println(eintrag);
}
…
```

In Java gibt es eine weitere Form wie Sie die For-Schleife nutzen können. Diese Variante wird ForEach genannt. Dabei sollen bei einer Liste alle Einträge direkt schrittweise aufgerufen werden. Dies kann in manchen Situationen Schreibarbeit einsparen, da in dieser Variante Laufvariable, Anfangs- und Endbedingung nicht beschrieben werden müssen.

# ForEach

```
…
ArrayList<String> list = new ArrayList<String>();
list.add("Hallo");
list.add("Welt");
list.add("!");

for(String eintrag : list){
     System.out.println(eintrag);
}
…
```

Diese Art der For-Schleife wird durch den Doppelpunkt gekennzeichnet

Dabei geben Sie den Typ des Eintrages und einen Variable-Name an. Nach dem Doppelpunkt geben Sie das Objekt, welches die Einträge besitzt, an. Dieses Objekt **muss** die Schnittstelle *Iterable* implementiert haben, damit dieses für eine ForEach-Schleife verwendet werden kann.

Auch können Sie statische Arrays für eine ForEach-Schleife verwenden.

# assoziative Datenfelder

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|---|---|
| prime | 2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 | 23 | 29 |

Stellen Sie sich vor, dass Sie ein Programm schreiben wollen, welches mit Primzahlen operiert. Dabei soll Ihr Programm überprüfen, ob eine gewisse Zahl eine Primzahl ist. Um nicht bei jeder Abfrage eine Berechnung durchzuführen, speichern Sie die Primzahlen in ein Array. Für den Aufbau des Arrays stehen Ihnen jedoch mehrere Möglichkeiten zur Verfügung.

Oben dargestellte Grafik zeigt eine Variante auf. Die Primzahlen sind in einem Integer-Array gespeichert. Für die Abfrage müssen Sie jedes Element durchgehen und überprüfen, ob die abzufragende Zahl vorkommt.

# assoziative Datenfelder

| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|---|---|
| prime | false | false | true | true | false | true | false | true | false | false |

Eine weitere Möglichkeit wäre ein Boolean-Array zu verwenden. Dazu wird der Index als den zu überprüfenden Wert aufgefasst und der Inhalt mit *true* (ist eine Primzahl) und *false* (ist keine Primzahl) versehen.

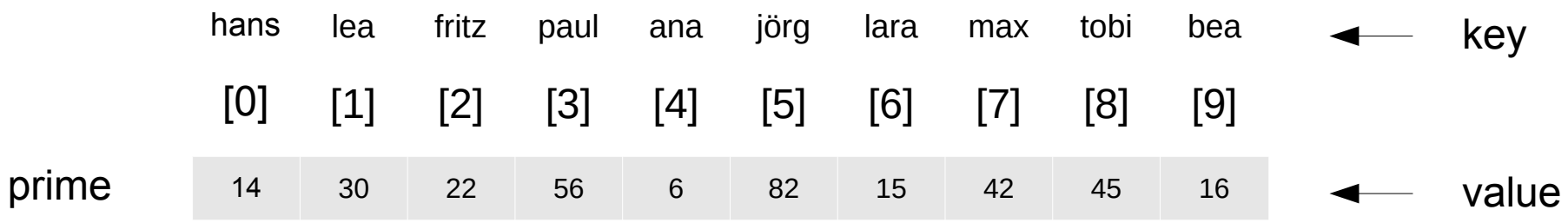
Dies hat den Vorteil, dass für eine Überprüfung nicht das ganze Array durchsucht werden muss. Ist z.B. die Zahl 7 gefragt, kann direkt auf den Index 7 zugegriffen werden und man erhält direkt das Ergebnis, dass die Zahl 7 eine Primzahl ist.

# assoziative Datenfelder

| | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 |
|---|---|---|---|---|---|---|---|---|---|---|
| | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| prime | false | true | false | true | false | false | false | true | false | true |

Sind jedoch bei Ihrem Programm nicht die Zahlen von 0 bis 9, sondern von 100 bis 109 gefragt, könnten Sie den Index durch Subtraktion mit der Zahl 100 erreichen. Sie haben also die Möglichkeit in der Programmumgebung dem Nachschlagewert (key) mit einer **Berechnung** zum Eintrag (value) zu gelangen.

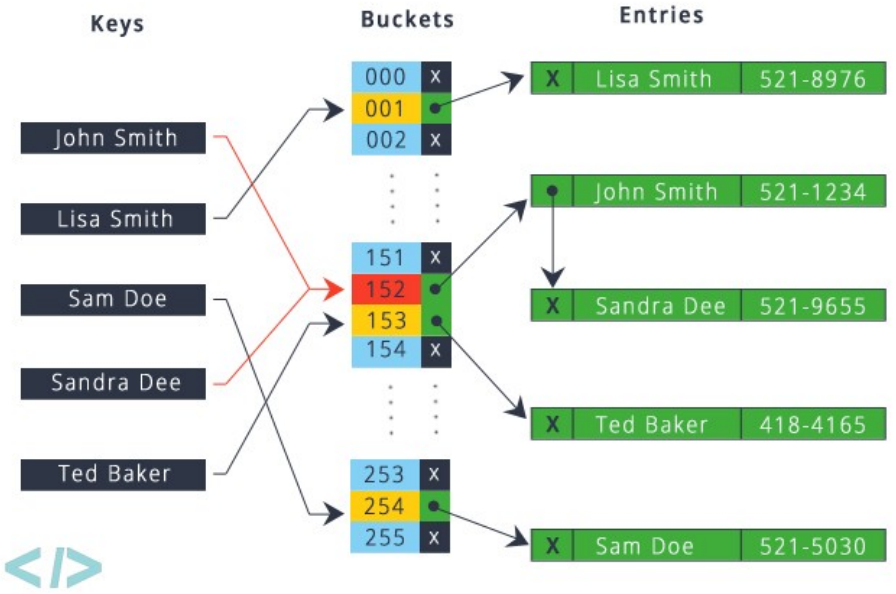
# HashMap




Mit einer Berechnung könnten Sie auch jeden Wert wie Strings als Nachschlagewert (key) verwenden, um den Eintrag zu erhalten (value). Die Berechnung würde diesen Wert in den Index umwandeln.

Solch ein Verfahren wird meistens mit Hash-Funktionen realisiert. In Java ist dies z.B. in den *HashMap*- oder *Hashtable*-Klasse umgesetzt. Die Klassen nehmen zwei generische Typen auf. Den Ersten für den Key-Typ und den Zweiten für den Value-Typ. Mit der Methode *put* können Sie einen Eintrag mit entsprechendem Schlüssel (key) setzten. Mit der Methode *get* wird durch Angabe des Schlüssels der Wert (value) geladen. Dabei geschieht der Zugriff direkt (mit entsprechender Berechnung).

# Aufgabe




- Führen Sie nun die Praktikumsaufgabe 4 aus.