Collections of Objects

Buch, Kapitel 4

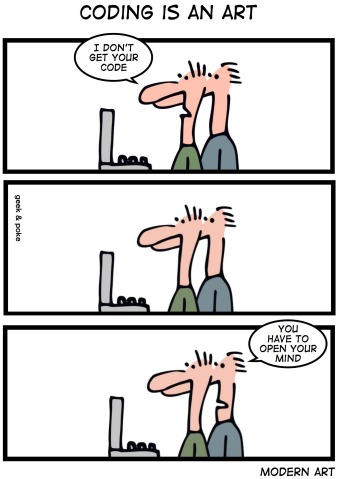
J. Kleimann, H.-W. Sehring, D. Versick, F. Zimmermann

Studiengang Wirtschaftsinformatik

NORDAKADEMIE
HOCHSCHULE DER WIRTSCHAFT

Vorlesungsinhalt

1. Lerninhalte
2. MusicOrganizer
3. Iterieren von Collections
4. Iterieren mit `while`
5. Suchen in Sammlungen unter Verwendung von Schleifen
6. Zeichenketten
7. Iteratoren



Lernziele

Nach Beenden dieser Lektion werden Sie

- mit einer `ArrayList` mehrere gleichartige Objekte gemeinsam verwalten und bearbeiten können.
- einen ersten Einstieg in das Java Collection Framework erhalten haben.
- drei verschiedene Möglichkeiten (for-each, while und Iterator) kennen, mit den Elementen einer Collection zu arbeiten.
- Programmkonstrukte erstellen können, die es erlauben Anweisungen mehrfach auszuführen.

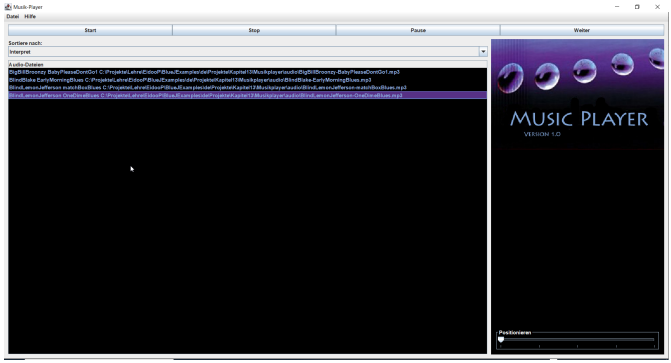
Lerninhalte

- Referenztypen
- Methodenaufrufe auf Objekten

Die Musiksammlung

Das zu entwickelnde System soll

- in der Lage sein, einzelne Dateien hinzuzufügen.
- keine Grenzen bezüglich der Anzahl von Dateien haben.
- die Anzahl der gespeicherten Dateien angeben.
- eine Dateiliste erstellen können.



Collections

- Die Besonderheit des MusicOrganizer, die uns hier interessiert, ist dass eine **unbegrenzte** Zahl von Musiktiteln verwaltet werden muss.
- Die Arrays aus den letzten Foliensätzen können immer nur eine vorher angegebene Maximalzahl von Objekten speichern. Eine solche Maximalzahl lässt sich aber häufig nicht sinnvoll angeben.
- “Wachsende” Arrays sind mühsam und nicht objektorientiert:

```
int[] tooSmall = new int[] { 1, 2, 3 }; // I want to add 4 :-(
int[] oneMore = new int[tooSmall.length + 1];
System.arraycopy(tooSmall, 0, oneMore, 0, tooSmall.length);
oneMore[3] = 4;
tooSmall = oneMore;
```

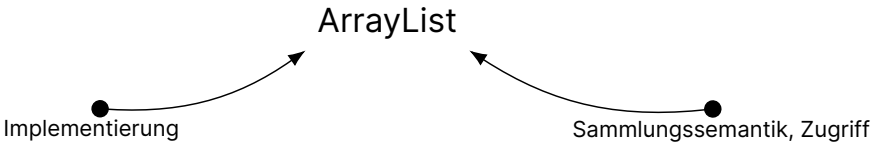
- Java stellt dafür eine Reihe von Klassen zur Verfügung, die (nahezu) unbegrenzt viele Objekte aufnehmen können: die Collection Klassen.
  Collections: Sammlungen, Mengendatentypen
- Die Collection Klassen können in beliebigen Anwendungskontexten wiederverwendet werden.
- Die Collection Klassen sind Teil der Java Runtime Bibliothek.

Pakete und Bibliotheken

- Die Collection Klassen werden in Paketen und Klassenbibliotheken (packages und libraries) zur Verfügung gestellt.
- Das Paket `java.util` enthält sehr viele Klassen zur Verwaltung Sammlungen von Objekten. Es ist in allen Java Distributionen enthalten.
- Die Klassen des Packages java.util unterscheiden sich
  - in der Organisationsform (Stack, Queue, Liste, Set, Map, ...)
  - in Ihren Laufzeiteigenschaften. Je nach Einsatzprofil sind andere Klassen vorteilhaft.
  - in Zusatzeigenschaften, wie Sortierbarkeit, der Verwaltung von Doppelten, etc.
  - in Ihren Concurrency Eigenschaften (parallele Verarbeitung erfordert mehr Koordinationsaufwand)

Arten von Collections

Die Klassennamen geben typischerweise Aufschluss über die Collection.



Sammlungssemantik	z.B. Liste (sortiert, doppelte Einträge), Menge (unsortiert, keine doppelten Einträge), Stack
Zugriff	lesend und schreibend (nur Hinzufügen, Lesen wahlfrei über Index, Lesen wahlfrei über Name, Lesen des ersten Elements, …)
Implementierung	z.B. durch Array, als verkettete Liste, als Bucket-Struktur, …; Einfluss auf Kosten von Operationen!

Übung

Welche Vorteile hat die Nutzung der Collection class library?

- Aufwand?
- Performance?
- Lernaufwand?
- Wiederverwendung?

Beispiel

MusicOrganizer

```
public class MusicOrganizer {
  // Storage for an arbitrary number of file names.
  private java.util.ArrayList files;

  // Perform initialization for the organizer.
  public MusicOrganizer() {
    files = new java.util.ArrayList();
  }
  ...
}
```

Nutzung von Klassen aus Klassenbibliotheken

- Möchte man eine Klasse eines fremden Pakets nutzen, ist der **voll qualifizierte Klassenname** anzugeben:

```
java.util.ArrayList files =
        new java.util.ArrayList();
```

- Um dies zu vermeiden, kann man einmal am Anfang der Klasse ein import Statement angeben:

```
import java.util.ArrayList;
```

Dann ist die Klasse `java.util.ArrayList` im gesamten Code der Klasse `MusicOrganizer` bekannt und kann überall unter ihrem einfachen Klassennamen `ArrayList` angesprochen werden:

```
ArrayList files = new ArrayList();
```

Beispiel (vollqualifiziert vs. import)

`MusicOrganizer` (vollqualifiziert)

```
public class MusicOrganizer {
  // Storage for an arbitrary number
      of file names.
  private java.util.ArrayList files;

  // Perform initialization for the
      organizer.
  public MusicOrganizer() {
    files = new java.util.ArrayList();
  }
  ...
}
```

`MusicOrganizer` (import)

```
import java.util.ArrayList;

public class MusicOrganizer {
  // Storage for an arbitrary number
      of file names.
  private ArrayList files;

  // Perform initialization for the
      organizer.
  public MusicOrganizer() {
    files = new ArrayList();
  }
  ...
}
```

Übung

Erstellen Sie ein neues Projekt MusicOrganizer und fügen die Klasse `MusicOrganizer` hinzu.

Parametrisierte Collections 1/2

- In einer Collection können beliebig viele beliebige Objekte gespeichert werden.
- Zur Fehlervermeidung benötigen wir sortenreine Collections, die nur String- oder nur Person- oder nur Music-Objekte aufnehmen kann.
- Java soll in der Lage sein, die Typen von Objekten zur Übersetzungszeit zu überprüfen.

→ Collections benötigen einen **Typparameter**.

Nicht parametrisiert

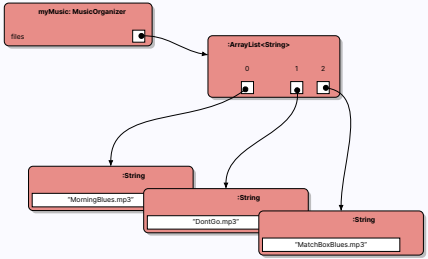
```
ArrayList list = new ArrayList();
list.add ("name");
list.add (new NumberDisplay(12));
list.add (new TicketMachine());

Object value = list.get(1);
```

Hier nur `Object` als Typ möglich - wir wissen nicht welchen konkreten Typ wir aus der Liste bekommen!

Parametrisierte Collections 2/2

Wir legen fest:

- den Typ der Collection: `ArrayList` - eine **List**e
- den Typ der Objekte, den die Collection enthalten wird: **String** — dieser wird mittels <Typ> am Typ der Collection angegeben



- Wir haben dann eine `ArrayList` von Strings (`ArrayList<String>`).
- Wir können auch ArrayList-en mit anderen Objekten anderer Referenztypen erzeugen, zum Beispiel mit Personen, TicketMachines, ...
- Es ist nicht möglich ArrayList-en aus primitiven Datentypen zu bauen.

Beispiel

Festgelegter Typ der Objekte in Collection

MusicOrganizer

```
public class MusicOrganizer {
  // Storage for an arbitrary number of file names.
  private java.util.ArrayList<String> files;

  // Perform initialization for the organizer.
  public MusicOrganizer() {
    files = new java.util.ArrayList<>();
  }

  ...
}
```

Generic Classes (Generische Klassen)

- Collections sind ein Beispiel für generische Klassen.
- Manchmal werden generische Klassen auch parametrisierte Klassen genannt, da sie einen Typparameter haben.

- `ArrayList` stellt die Methoden
  - `add (Typ item)`, `Typ get(...)`, `remove (Typ item)`, ...
  um Objekte, die den Typ des Typparameters haben hinzuzufügen, zu holen oder zu löschen.

Beispiele

```
private ArrayList<String> strings =
    new ArrayList<>();
private ArrayList<NumberDisplay>
    displays = new ArrayList<>();
private ArrayList<TicketMachine>
    machines = new ArrayList<>();

...

String string = strings.get(7);
displays.add (new NumberDisplay(52))
    ;
TicketMachine machine = machines.get
    (7);
String string2 = displays.get(42);
    // COMPILE ERROR

...
```

Übung

Welchen Rückgabetyp und welche Parametertypen haben die Methoden `add`, `get` und `remove` für eine Collection vom Typ `ArrayList<String>` bzw. `ArrayList<MusicTrack>`?

Erzeugen eines `ArrayList` Objekts

- Bei eine Variablendeklaration wird der aktuelle Typparameter in spitzen Klammern festgelegt:

```
ArrayList<String> files;
```

- Bei der Erzeugung eines Objekts braucht der aktuelle Typparameter nicht noch einmal wiederholt werden. Der Compiler leitet den Typparameter aus der Deklaration ab. Es muss allerdings durch Angabe von <> angegeben werden, dass es sich um einen generischen Typ handelt:

```
files = new ArrayList<>();
```

- Auch möglich aber veraltet:

```
files = new ArrayList<String>();
```

- Der Verzicht auf generischen Typ ist (je nach Compiler-Einstellung) auch möglich, es wird jedoch stark davon **abgeraten**:

```
files = new ArrayList();
```

Übung

Deklarieren Sie eine Variable, die Objekte der Klasse MusicTrack aufnehmen kann. Wie kann man diese initialisieren?

Was macht eine `ArrayList` aus?

- ArrayLists haben eine festgelegte Kapazität, die erweitert wird, wenn es nötig ist.
- Führt intern über seine Größe Buch.
- Erhält die Ordnung der Objekte.
- Wie `ArrayList` intern aufgebaut ist, ist unerheblich. Wenn überhaupt, interessieren uns die Performance Charakteristiken einer `ArrayList`. Entsprechende Eigenschaften werden in der Veranstaltung „Algorithmen und Datenstrukturen“ diskutiert.

Collection Methoden

Mit der Methode `size` kann man herausfinden, wie groß die Collection ist.

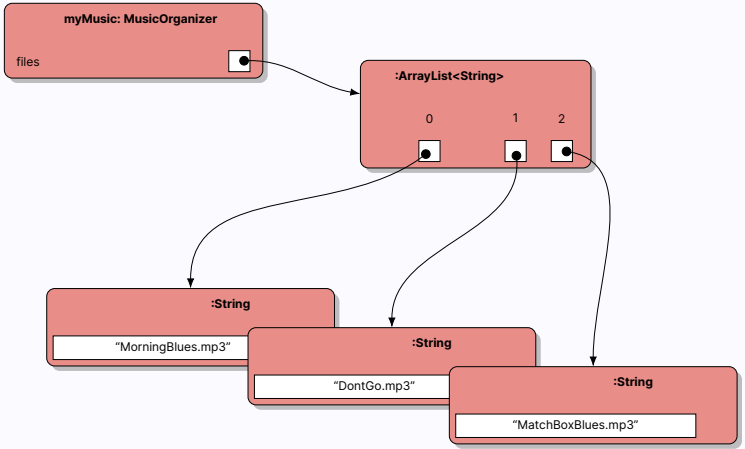
add und size

```
public class MusicOrganizer {
  private ArrayList<String> files;
  ...
  public void addFile(String filename) {
    files.add(filename);
  }

  public int getNumberOfFiles() {
    return files.size();
  }
  ...
}
```

Übung

Fügen sie Ihrem Projekt die beiden Methoden `addFile` und `getNumberOfFiles` hinzu.

Nummerierung der Elemente



Analog zu Arrays erfolgt der Zugriff auf

- das erste Element mit dem Index `0`
- das letzte Element mit dem Index `size()-1`

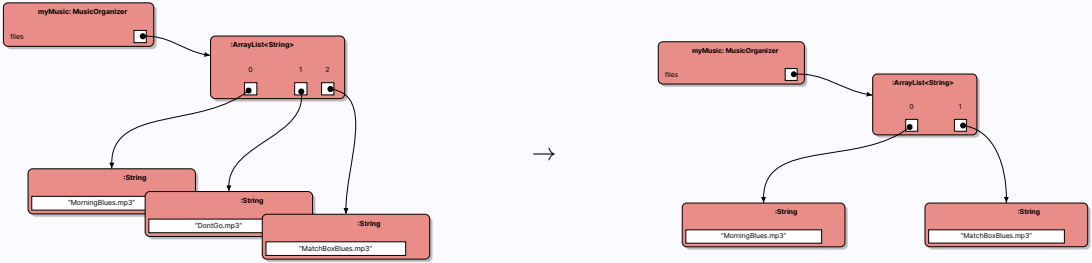
Zugriff auf Objekte mit Index

get Methode

```
public void listFile(int index) {
  if (index >= 0 && index < files.size()) {
    String filename = files.get(index);
    System.out.println(filename);
  } else {
    System.out.println("This is not a valid index.");
  }
}
```

Löschen mit Index: Methode remove(int index)

Löschen des Dateinamens mit dem Index 1: `files.remove(1);`



- Löscht das Element „DontGo.mp3“ mit Index 1 aus der Liste.
- Führt zu einer neuen Nummerierung aller folgenden Elemente innerhalb der `ArrayList`. Der „MatchBoxBlues.mp3“ bekommt den neuen Index 1.
- remove(1) hat als Rückgabewert das gelöschte Element „DontGo.mp3“ selbst.

Wiederholung

- In Collections kann man eine beliebige Anzahl von Objekten speichern.
- Wir haben die `ArrayList` Collection verwendet.
- `ArrayList` kommt aus dem Paket `java.util`.
- `ArrayList` ist ein parametrisierter (generischer) *Typ*.
- Objekte einer festgelegten Klasse (der *Typ*) können hinzugefügt und gelöscht werden.
- Jedes Objekt erhält einen Index.
- Löschen eines Objekts verändert den Index aller nachfolgenden Objekte.
- ArrayListen ändern die Ordnung der Elemente nicht.
- Wichtige Methoden: `add`, `get`, `remove` und `size`.

Motivation

- Sehr häufig will man eine Aktion für alle Objekte einer Collection durchführen.
- Zum Beispiel alle Dateinamen einer Musiksammlung ausdrucken.
- Java verfolgt einen anderen Ansatz als funktionale Programmiersprachen: Schleifenanweisungen.
- Java kennt mehrere Arten von Schleifen.
- Wiederholung: Die einfachste ist die for-each Schleife.

for-each Schleife

... bei Collections

Allgemeine Struktur

```
for (ElementType element : collection) {
  Schleifenrumpf;
}
```

- **`for`** Schlüsselwort
- In runden Klammern mit einem Doppelpunkt getrennt:
  - Eine **Variable**, die nacheinander jedes Element der Collection aufnimmt.
  - Die **Collection**, deren Elemente durchlaufen werden sollen.
- der **Schleifenrumpf**, in dem das Element der Collection verarbeitet wird.

Übung

Wie könnte eine `listAllFiles` Methode aussehen?

Beispiel
Organizer

listAllFiles

```
ArrayList<String> files;

/**
* List all file names in the organizer.
*/
public void listAllFiles() {
  for(String filename : files) {
    System.out.println(filename);
  }
}
```

- Der Typ der Variable `filename` stimmt mit dem generischen Typparameter überein.

Filtern von Elementen

- Es ist eine häufige Anforderung, dass man nicht alle Elemente durchläuft, sondern nur gewisse.

findFiles

```
public void findFiles(String searchString) {
  for (String filename : files) {
    if (filename.contains(searchString)) {
      System.out.println(filename);
    }
  }
}
```

- Nb.: Filter können auch mit λ-Ausdrücken formuliert werden. Diese ermöglichen Syntax ähnlich zu funktionalen Programmiersprachen.

Diskussion der for-each Schleife

- Leicht zu schreiben und zu verstehen.
- Beginn und Ende Bedingungen der Schleife automatisch gesetzt.
- Die Collection kann und darf nicht verändert werden.
- Index eines Elements ist nicht verfügbar. Die for-each Schleife kann auch für Collections verwendet werden, die nicht Index basiert sind.
- Keine Endlos-Schleifen möglich, da die Anzahl der Schleifendurchgänge vorher bekannt ist.

Die `while` Schleife

Wiederholung

- Eine **`while`** Schleife bietet sich an, wenn das Durchlaufen der Schleife von einer Abbruchbedingung, z.B. dem Finden eines bestimmten Elements in einer Collection abhängig gemacht werden kann.
- Die Abbruchbedingung kann sehr flexibel formuliert werden und ist nicht auf das Durchlaufen von Collections beschränkt.
- Es sind Endlosschleifen möglich, wenn die Abbruchbedingung niemals wahr wird.
- Generell fehleranfälliger aufwändiger zu programmieren als eine for-each Schleife, aber bei weitem flexibler.

`while` Schleife

`while` Schleife

```
while (Bedingung) {
  Rumpf;
}
```

- Die **`while`** Schleife hat eine Abbruchbedingung, die in runden Klammern angegeben werden muss. Es ist ein Ausdruck, der ein boolesches Ergebnis, also wahr oder falsch zurück gibt.
- Anders als in der Programmiersprache C sind **`int`** Ausdrücke nicht erlaubt.
- Der Schleifenrumpf wird nur ausgeführt wenn die Bedingung wahr ist.
- Ist die Bedingung falsch, so ist die Schleife beendet.
- Nach Ausführen des Rumpfes, wird die Bedingung erneut geprüft.

`while` und Collections

For-each Äquivalent

listAllFiles Version 2

```
/**
* List all file names in the organizer.
*/
public void listAllFiles() {
  int index = 0;
  while (index < files.size()) {
    String filename = files.get(index);
    System.out.println(filename);
    index++;
  }
}
```

Fünf Punkte, die zu beachten sind

1. Eine Indexvariable wird vor der `while` Schleife deklariert und initialisiert.
2. Die Abbruchbedingung muss richtig formuliert werden.
3. Man muss in der Schleife eine Variable für Elemente deklarieren.
4. Jedes Element muss mit `get(index)` aus der Collection geholt werden und der Elementvariablen zugewiesen werden.
5. Die Indexvariable muss hochgezählt werden.
6. Wird die Collection durch Hinzufügen oder Löschen von Objekten verändert, so kann der Index aus der Synchronisation geraten[1].

[1] Eine parallele Änderung vorausgesetzt.

Vergleich for-each vs while

- for-each
  - leichter zu schreiben
  - weniger anfällig für Programmierfehler
- while
  - Man muss nicht die ganze Collection durchlaufen
  - auch ohne Collection nutzbar
  - Sorgfalt erforderlich: Endlosschleifen möglich

Suchen in Collections

Abbruchbedingung formulieren

- Eine häufig auftretende Aufgabe.
- Abbruchbedingung: Schleife wird weiter ausgeführt wenn
  - noch kein Element gefunden wurde **und**
  - es noch weitere Suchmöglichkeiten gibt
- Collection kann leer sein. Es ist immer eine wichtiger Check ob die Schleife mit leeren Collections umgehen kann.

Abbruchbedingung Teil 1

- Wenn ein Element in der Liste gefunden wurde, wird eine lokale boolesche Variable gesetzt.
  - Diese Variable wird vor der Schleife angelegt:

    ```
    boolean found=false;
    ```

    oder

    ```
    boolean searching=true;
    ```

  - Findet man ein Element, das den Filterbedingungen entspricht, ändert man den Wahrheitswert:

    ```
    found=true;
    ```

    oder

    ```
    searching=false;
    ```

Abbruchbedingung Teil 2

- Das Durchlaufen der `ArrayList` wird über einen Index gemacht.
- Eine Indexvariable wird vor der Schleife mit 0 initialisiert.

```
int index=0;
```

- Der Zugriff auf das Element der `ArrayList` mit dem Index.

```
String file = files.get(index);
```

- Am Ende jeder Schleife wird der Index um 1 erhöht.

```
index = index + 1;
```

- Weitere Elemente sind vorhanden, solange

```
index < files.size()
```

Die Suchschleife

findFile

```
int index = 0;
boolean searching = true;
while (index < files.size() && searching) {
  String file = files.get(index);
  if (file.equals(searchString)) {
    // Do something with the file found
    // We don't need to keep looking.
    searching = false;
  } else {
    index++;
  }
}
// We found it at index, or searched whole collection.
```

Eine Alternative

found statt searching

```
int index = 0;
boolean found = false;
while (index < files.size() && !found) {
  String file = files.get(index);
  if (file.equals(searchString)) {
    // Do something with the file found
    // We don't need to keep looking.
    found = true;
  } else {
    index++;
  }
}
// We found it at index, or searched whole collection.
```

Übung

Was passiert, wenn die Collection leer ist?

Die Klasse String

- Strings repräsentieren Zeichenketten.
- String ist eine vordefinierte Klasse. Strings haben einige besondere Eigenschaften, die nicht alle Klassen haben.
- String liegt im Paket `java.lang` und braucht nicht importiert werden.
- Der Operator + kann zur Verkettung zweier `String` Objekte genutzt werden.
- Aufpassen beim Vergleich von zwei `String` Objekten:
  - Testet auf Identität

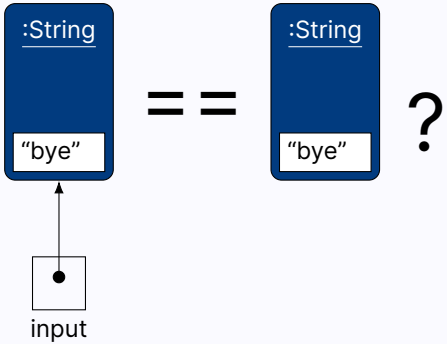
    ```
    if (file==searchString){...}
    ```

    Nicht verwenden.
  - Testet auf inhaltliche Gleichheit.

    ```
    if (file.equals(searchString)){...}
    ```

Übung: Identität und Gleichheit

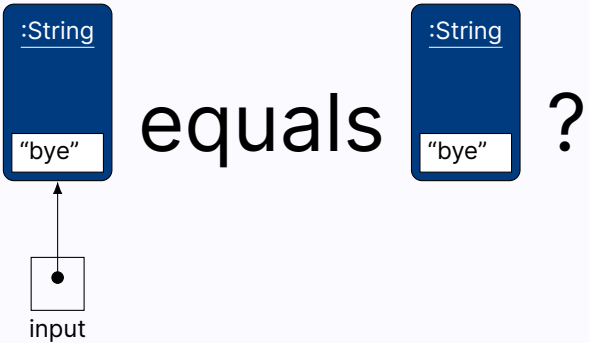
```
String input = reader.getInput();
if(input == “bye“) {...}
```



false!

Übung: Identität und Gleichheit

```
String input = reader.getInput();
if(input == “bye“) {...}
```

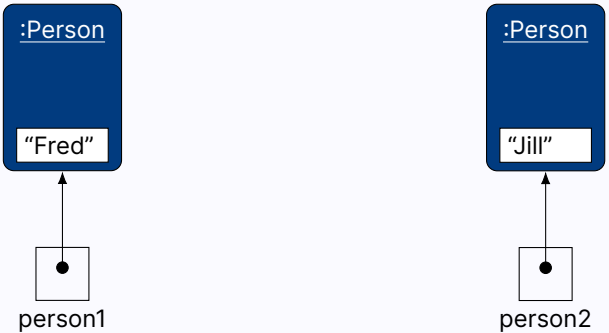


true!

Übung: Identität und Gleichheit

Was ist das Ergebnis von

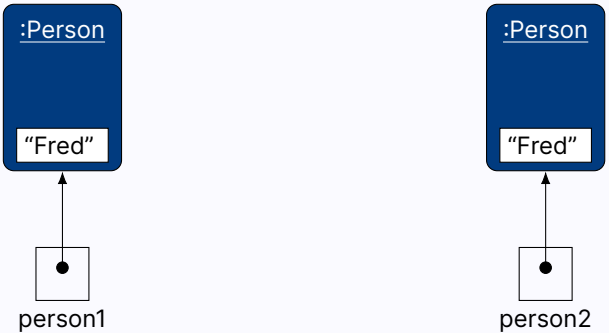
```
person1 == person2
```



Übung: Identität und Gleichheit

Was ist das Ergebnis von

```
person1 == person2
```



Übung: Identität und Gleichheit

Was ist das Ergebnis von

```
person1 == person2
```

Weg von den Strings

- Collections können beliebige Objekttypen speichern.
- Angaben zu Geschäftsobjekten wie z.B. Musikstücke sind üblicherweise nicht in einem einfachen `String` gut abbildbar.
- Konkrete Musikstücke haben z.B. verschiedene Eigenschaften, wie
  - Künstler
  - Titel
  - Dateiname
  - Länge des Stückes
  - ...

Der objektorientierte Weg

- Arbeiten mit Collections kann auf eine „besonders objektorientierte“ Art realisiert werden: Mit **Iterator**-Objekten.
- mit der `iterator()` Methode kann man sich von einer beliebigen Collection ein `Iterator`-Objekt geben lassen.
- `Iterator`-Objekte sind in der Lage die Objekte eine Collection zu durchlaufen.
- `Iterator`-Objekte sind in der Lage beim Durchlaufen der Collection Objekte zu löschen, ohne dass sie aus der Synchronisation kommen.

Methoden von Iterator<E>

```
boolean hasNext();
E next();
void remove();
```

Nutzung eines `Iterator` Objekts

Allgemeines Schema

```
import java.util.Iterator;
...
Iterator<ElementType> it =
  myCollection.iterator();
while (it.hasNext()) {
  // get the next object
  ElementType element = it.next();
  // do something with that object
}
```

Konkrete Anwendung

```
public void listAllFiles() {
  Iterator<MusicTrack> it =
    files.iterator();
  while (it.hasNext()) {
    MusicTrack mt = it.next();
    System.out.println(mt.getDetails());
  }
}
```

Vergleich aller Zugriffsmethoden

Wir haben drei Wege kennengelernt, wie man die Elemente einer Collection bearbeiten kann.

- for-each-loop
  - Geeignet, wenn man jedes Element einmal bearbeiten möchte.
- while-loop
  - Geeignet, wenn man die Verarbeitung gerne frühzeitig beenden möchte.
  - Kann auch für Wiederholungen ohne Collection benutzt werden.
- `Iterator` Objekt
  - Geeignet, wenn man die Verarbeitung gerne frühzeitig beenden möchte.
  - Sinnvoll, wenn der Indexzugriff auf eine Collection nicht möglich ist (z.B. `HashSet`) oder sehr ineffektiv ist (z.B. `LinkedList`).
  - Wird verwendet, um gewisse Elemente einer Collection zu löschen.

Das Arbeiten mit Collections ist eine grundlegende Fertigkeit, die jeder Programmierer beherrschen muss.

Fragen?

Für weitere Fragen im Nachgang können Sie mich gerne über Moodle oder via E-Mail kontaktieren!

Konzepte: Zusammenfassung I

- **Collections** Collectionobjekte (Sammlungsobjekte) sind Objekte, die eine beliebige Anzahl anderer Objekte enthalten können.
- **Pakete und Bibliotheken** Java Bibliotheken enthalten wiederverwendbare Klassen, die zum Bau von komplexeren Anwendungen genutzt werden können. Zusammengehörige Klassen sind in Paketen (syn. Verzeichnisse) zusammengefasst. Eine Bibliothek enthält in der Regel mehrere Pakete.
- **Typparameter** Ein Typparameter legt den Parameter- oder Rückgabetyp von Methoden fest. Wichtigstes Beispiel sind die Collection Klassen.
- **Parametrisierte Collections** Eine parametrisierte Collection kann nur Objekte vom Typ eines Typparameters verarbeiten. Parametrisierte Klassen werden auch als generische Klasse bezeichnet.
- **List** Eine Liste (List) ist eine Sammlung, die Elemente als Einträge enthält.
- **Iterator** Ein Iterator ist ein Objekt, mit dessen Hilfe über alle Elemente einer Sammlung iteriert werden kann.

Collections

Collectionobjekte (Sammlungsobjekte) sind Objekte, die eine beliebige Anzahl anderer Objekte enthalten können.

Pakete und Bibliotheken

Java Bibliotheken enthalten wiederverwendbare Klassen, die zum Bau von komplexeren Anwendungen genutzt werden können. Zusammengehörige Klassen sind in Paketen (syn. Verzeichnisse) zusammengefasst. Eine Bibliothek enthält in der Regel mehrere Pakete.

Typparameter

Ein Typparameter legt den Parameter- oder Rückgabetyp von Methoden fest. Wichtigstes Beispiel sind die Collection Klassen.

Parametrisierte Collections

Eine parametrisierte Collection kann nur Objekte vom Typ eines Typparameters verarbeiten. Parametrisierte Klassen werden auch als generische Klasse bezeichnet.

List

Eine Liste (List) ist eine Sammlung, die Elemente als Einträge enthält.

Iterator

Ein Iterator ist ein Objekt, mit dessen Hilfe über alle Elemente einer Sammlung iteriert werden kann.