Tech Ramp Up

Michael Inden

Freiberuflicher Consultant und Trainer

https://github.com/Michaeli71/Tech-Ramp-Up-1-Java-Law-Of-Big-3

Speaker Intro

- **Michael Inden**, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- Freiberuflicher **Consultant**, **Trainer** und **Konferenz-Speaker**
- **Seit Januar 2022 Head of Development bei Adcubum in Zürich**
- Autor und Gutachter beim dpunkt.verlag / APress

E-Mail: michael_inden@hotmail.com
Blog: https://jaxenter.de/author/minden

Kurse: Bitte sprecht mich an!

Agenda

Workshop Contents

- **PART 1: Intro Identität & Inhaltliche Gleicheit**
- **PART 2: Suchen mit `equals()`, Fallstricke und der Kontrakt**
- **PART 3: Suchen im `HashSet` und `hashCode()`**
- **PART 4: Collections Recap**
- **PART 5: Iteratoren**
  - RandomIterator ??
  - Ergänzungen aus Design Patterns?

https://github.com/Michaeli71/Tech-Ramp-Up-1-Java-Law-Of-Big-3

PART 1:
Intro Identität & Inhaltliche Gleicheit

Beispielklasse definieren

```
jshell> class Car
...> {
...>     public Car()
...>     {
...>         this("Unbekannt", "Unlackiert", 0);
...>     }
...>
...>     public Car(String brand, String color, int horsePower)
...>     {
...>         this.brand = brand;
...>         this.color = color;
...>         this.horsePower = horsePower;
...>     }
...>
...>     public String toString()
...>     {
...>         return "Marke: %s / Farbe: %s / PS: %d".
...>                formatted(brand, color, horsePower);
...>     }
```

Klassen definieren

- **Versuchen wir es mal:**

```
jshell> Car myFirstCar = new Car()
myFirstCar ==> Marke: Unbekannt / Farbe: Unlackiert / PS: 0

jshell> Car mySecondCar = new Car("Audi", "BLUE", 220)
mySecondCar ==> Marke: Audi / Farbe: BLUE / PS: 220

jshell> Car myThirdCar = new Car("Renault", "PETROL", 120)
myThirdCar ==> Marke: Renault / Farbe: PETROL / PS: 120
```

Verhalten definieren – Methoden

- **Wäre es nicht schön, sprechende Namen zum Verändern zu nutzen?**

```
...>     public Car(String brand, String color, int horsePower)
...>     {
...>         this.brand = brand;
...>         this.color = color;
...>         this.horsePower = horsePower;
...>     }
...>
...>     public void paintWith(String newColor)
...>     {
...>         this.color = newColor;
...>     }
...>
...>     public void applyTuningKit()
...>     {
...>         this.horsePower += 150;
...>     }
```

Klassen definieren

- **Versuchen wir es mal:**

```
jshell> Car emptyCar = new Car()
emptyCar ==> Marke: Unbekannt / Farbe: Unlackiert / PS: 0

jshell> emptyCar.paintWith("PURPLE")

jshell> emptyCar.applyTuningKit()

jshell> emptyCar
emptyCar ==> Marke: Unbekannt / Farbe: PURPLE / PS: 150
```

- **Soweit so gut, aber wie ändern wir die Marke und sollten wir das überhaupt nachträglich können?**

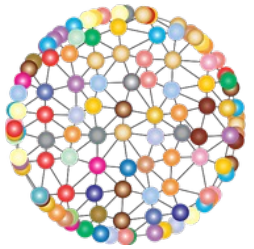
Kurze Diskussion

Objekte vergleichen

Objekte vergleichen

```
Car tomsCar = new Car("Audi", "BLUE", 275)
Car jimsCar = new Car("Audi", "BLUE", 275)

tomsCar == jimsCar
```

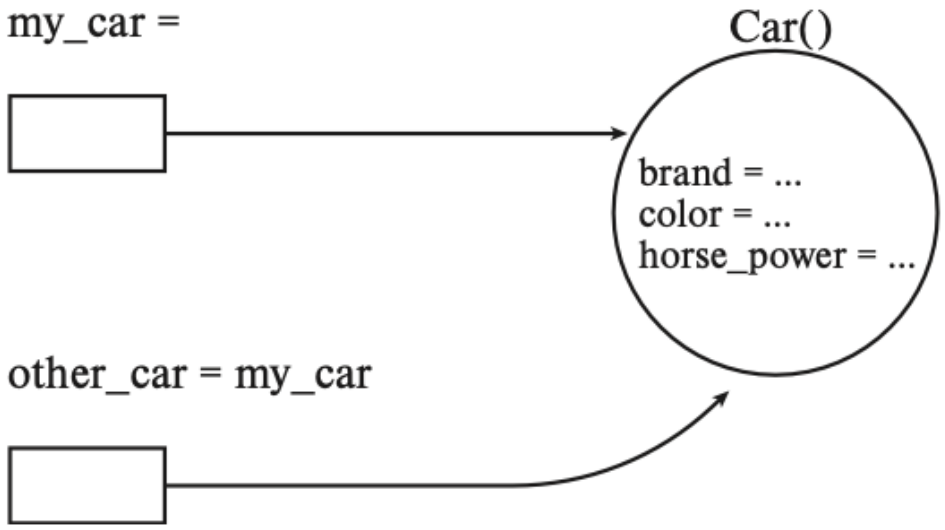
Was ist das Ergebnis?



Objekte vergleichen

```
jshell> Car tomsCar = new Car("Audi", "BLUE", 275)
tomsCar ==> Marke: Audi / Farbe: BLUE / PS: 275

jshell> Car jimsCar = new Car("Audi", "BLUE", 275)
jimsCar ==> Marke: Audi / Farbe: BLUE / PS: 275

jshell> tomsCar == jimsCar
$107 ==> false
```

- **Referenzgleichheit / Identität!**
- **Inhaltliche Gleichheit / semantische Gleichheit => gleiche Werte der Attribute**



Wie prüft man diese beiden Arten der Gleichheit in Java?

Objekte vergleichen

- Operator ‘==' – Mit dem Operator ‘==' werden Referenzen verglichen. Somit wird auf *Identität* geprüft, also, ob es sich um dieselben Objekte handelt.

- ‘`equals()`' – Mit `equals()` werden zwei Objekte bezüglich ihres *Zustands* (d. h. der Wertebelegung der Attribute) verglichen.

```
jshell> tomsCar.equals(jimsCar)
$108 ==> false
```

Wieso geht das denn auch nicht? Was braucht es für semantische Gleichheit in Java?

Objekte vergleichen

- ‘`equals()`' – Mit `equals()` werden zwei Objekte bezüglich ihres *Zustands* (d. h. der Wertebelegung der Attribute) verglichen.
- Zum inhaltlichen Vergleich eigener Klassen ist dort die Methode `equals()` selbst zu implementieren.
  - Standardmäßig erfolgt sonst lediglich ein Vergleich der beiden Referenzen mit dem Operator ‘==’.
  - In eigenen Realisierungen muss der Teil des Objektzustands verglichen werden, der für die semantische Gleichheit, also die inhaltliche Gleichheit, relevant ist.

Implementierung von `equals()`

1. **Typprüfung – Um nicht Äpfel mit Birnen zu vergleichen, sichern wir vor dem inhaltlichen Vergleich ab, dass nur Objekte des gewünschten Typs verglichen werden.**
2. **Objektvergleich – Anschließend werden diejenigen Attributwerte verglichen, die für die Aussage »Gleichheit« relevant sind.**

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                              // Null-Akzeptanz
        return false;
    if (this == other)                              // Reflexivität
        return true;
    if (this.getClass() != other.getClass())  // Typgleichheit sicherstellen
        return false;

    Car otherCar = (Car) other;
    return Objects.equals(brand, otherCar.brand) &&
           Objects.equals(color, otherCar.color) &&
           horsePower == otherCar.horsePower;
}
```

PART 2:
Suchen mit `equals()`, Fallstricke und der Kontrakt

Beispieldaten

```
public enum Farbe
{
    KARO, HERZ, PIK, KREUZ
}

public class Spielkarte
{
    private final Farbe farbe;
    private final int   wert;

    public Spielkarte(final Farbe farbe, final int wert)
    {
        this.farbe = farbe;
        this.wert = wert;
    }

    …

}
```

Beispieldaten

```
public static void main(final String[] args)
{
    // JDK 7: Diamond Operator: ArrayList<> statt ArrayList<Spielkarte>
    final Collection<Spielkarte> spielkarten = new ArrayList<>();

    spielkarten.add(new Spielkarte(Farbe.HERZ, 7));

    // PIK 8 einfügen
    spielkarten.add(new Spielkarte(Farbe.PIK, 8));
    spielkarten.add(new Spielkarte(Farbe.KARO, 9));

    // Finden wir eine PIK 8?
    final boolean gefunden = spielkarten.contains(new Spielkarte(Farbe.PIK, 8));
    System.out.println("gefunden=" + gefunden);
}
```

Gefunden oder nicht gefunden?

Abhilfe

```java
@Override
public boolean equals(Object other)
{
    if (other == null) // Null-Akzeptanz
        return false;

    if (this == other) // Reflexivittät
        return true;

    if (this.getClass() != other.getClass()) // Typgleichheit sicherstellen
        return false;

    //  Enum mit equals, int mit Wertevergleich
    final Spielkarte karte = (Spielkarte) other;
    return this.farbe.equals(karte.farbe) && this.wert == karte.wert;
}
```

Der `equals()`-Kontrakt (aus `Object.java`)

`public boolean equals(Object obj)` muss alles mit jedem vergleichen können

Äquivalenz-Relation mit folgenden Eigenschaften:

Null-Akzeptanz: Für jede Referenz x, die nicht null ist, liefert x.equals(null) den Wert false

Reflexiv: Für jede Referenz x, die nicht null ist, muss x.equals(x) den Wert true zurückliefern.

Symmetrisch: Für alle Referenzen x und y darf x.equals(y) nur den Wert true zurückgeben, wenn y.equals(x) dies auch tut.

Transitiv: Für alle Referenzen x, y und z gilt: Wenn sowohl x.equals(y) und y.equals(z) den Wert true ergeben, dann muss x.equals(z) auch true zurückliefern.

Person-Klasse

```
public final class Person
{
    private final String name;
    private final String city;
    private final Date birthday;

    public Person(final String name, final String city, final Date birthday)
    {
        if ( name == null || birthday == null || city == null)
            throw new IllegalArgumentException("parameters 'name', 'birthday' and 'city'
                                               must not be null!");

        this.name = name;
        this.city = city;
        this.birthday = birthday;
    }

    public final String getName()       { return name; }
    public final String getCity()       { return city; }
    public final Date   getBirthDay()   { return birthday; }
```

Der equals()-Kontrakt am Beispiel

```
public boolean equals(final Person other)
{
      return this.getName().equals(other.getName());
 }
```

Abprüfen

- Signatur: public boolean equals(Object other)
- Null-safe: for any non-null reference value x, x.equals(null) should return false.
- reflexive: for any reference value x, x.equals(x) should return true.
- symmetric: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- transitive: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

Der `equals()`-Kontrakt am Beispiel

Wieso ist das ungünstig???

- Fehler fällt nicht auf, wenn man nur Objekte gleichen Typs explizit über `equals()` vergleicht: `person1.equals(person2)`
- **Probleme beim Aufruf von `equals(null)`**
- **Allerdings betreibt man hier ungewolltes Overloading statt Overriding.**
- **Probleme wenn die Objekte in Container-Klassen des JDK gespeichert werden. Hier kommt die nicht überschriebene `equals()`-Methode der Klasse Object zum Einsatz.**
- `equals()` **vergleicht momentan nur die Namen, aber nicht die anderen Attribute**
  `equals()` **sollte alle Attribute eines Objekts vergleichen, die dieses eindeutig machen**

equals() korrekt laut Kontrakt

```
public boolean equals(Object other)
{
   if (other == null)     // Null-Akzeptanz
      return false;

   if (this == other)     // reflexiv
      return true;

   if (this.getClass() != other.getClass())    // compare only objects of same type
      return false;

   final Person otherPerson = (Person)other;
   return this.getName().equals(otherPerson.getName()) &&
          this.getCity().equals(otherPerson.getCity()) &&
          this.getBirthDay().equals(otherPerson.getBirthDay());
}
```

PART 3:
Suchen in `HashSet` und `hashCode()`

Suche im `HashSet`

```
public static void main(final String[] args)
{
    // JDK 7: Diamond Operator
    final Collection<Spielkarte> spielkarten = new HashSet<>();

    spielkarten.add(new Spielkarte(Farbe.HERZ, 7));
    // PIK 8 einfügen
    spielkarten.add(new Spielkarte(Farbe.PIK, 8));
    spielkarten.add(new Spielkarte(Farbe.KARO, 9));

    // Finden wir eine PIK 8?
    final boolean gefunden = spielkarten.contains(new Spielkarte(Farbe.PIK, 8));
    System.out.println("gefunden = " + gefunden);
}
```

Gefunden oder nicht gefunden?

Suchen mit `HashSet`

Gefunden = FALSE !!!

Wie kann das nach den zuvor durchgeführten Korrekturen an `equals()` sein?

Erinnern wir uns an die Verwaltung von Objekten in Hash-basierten Containern

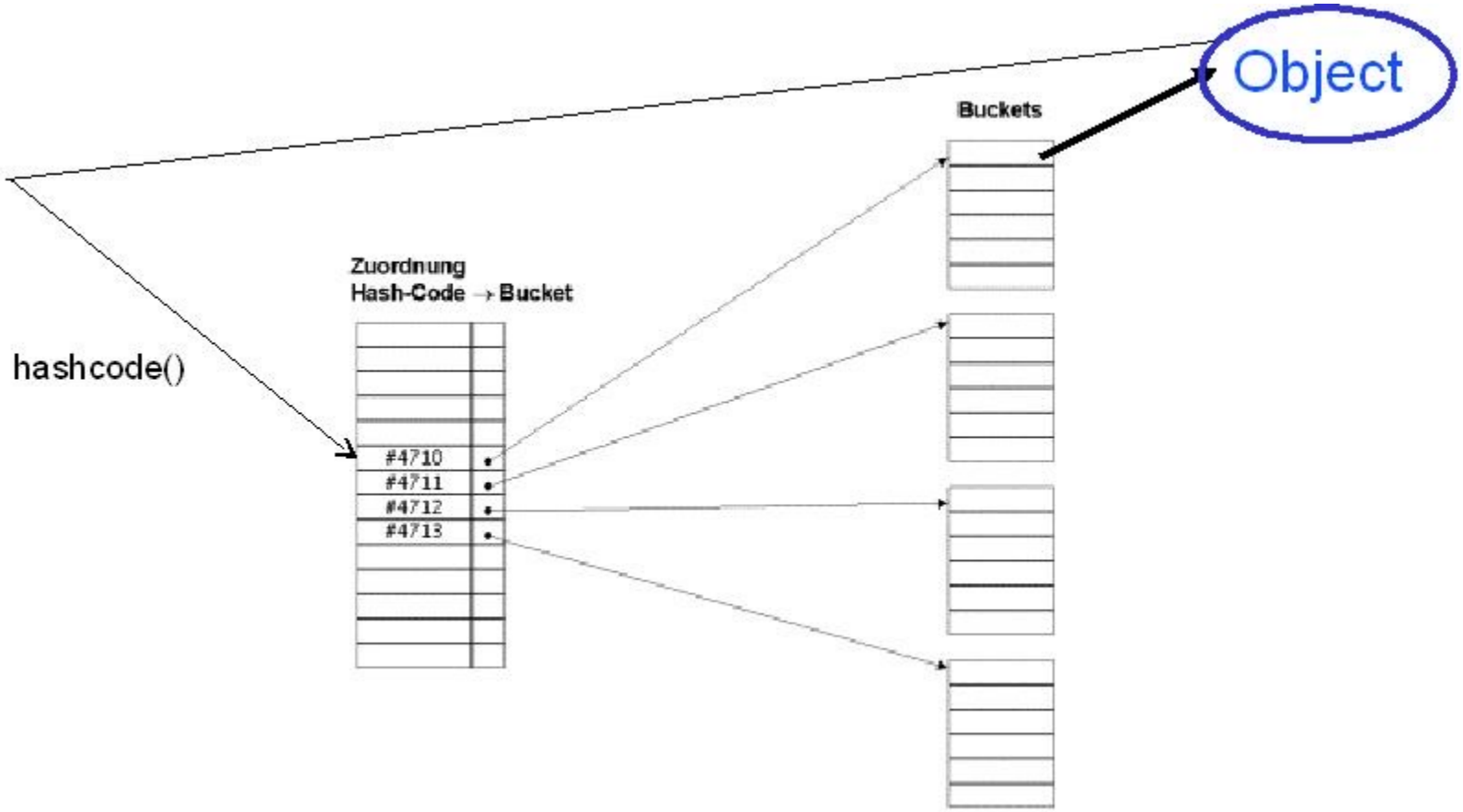
Verwaltung von Objekten in Hash-basierten Containern

Hash-basierte Container == riesige Schrankwände mit nummerierten Schubladen

Schubladen bieten wiederum Platz für beliebig viele Sachen.

Diese speziellen Schubladen werden im Informatik-Sprachgebrauch auch als „*Bucket*" bezeichnet.

Verwaltung von Objekten in Hash-basierten Containern



`hashCode()` berechnet die „Schublade”, in der mit `equals()` nach Objekten gesucht wird.

Verwaltung von Objekten in Hash-basierten Containern

Verwendet man diese Analogie zur Schrankwand, so wird die Benutzung von Hash-basierten Containern relativ einfach:

1. **Immer nur eine Schublade** => diese **quillt über** und man **findet** seine Sachen nur **mühselig** wieder, da immer der **komplette Inhalt** von vorne bis hinten **durchsucht** werden muss.

2. Verteilt man die Sachen **relativ gleichmäßig**, so kann man **schnell** die **richtige Schublade** finden und dort auch das **gewünschte Objekt**.

3. **Verliert man die Nummer**, **verändert** sich diese oder **vertut** sich, so **findet** man seine Sachen **nicht** dort, wo man diese erwartet. Im Extremfall müsste man dann alle Schubladen durchsuchen, um verlorene Sachen zu finden.

Verwaltung von Objekten in Hash-basierten Containern

Wir können folgende Anforderungen an die Methode `hashCode()` stellen:

1. Der Wert der Methode `hashCode()` **berechnet** die **Nummer** der **Schublade**. Dort wird das gewünschte **Objekt** per `equals()` **gesucht**. ***Daher müssen bei der Speicherung von Objekten in Hash-basierten Containern immer sowohl die Methode `equals()` als auch `hashCode()` implementiert werden.***

2. Eine **gleichmäßige Verteilung** erreicht man, indem die `hashCode()`–Methoden verschiedener Objekte **möglichst verschiedene Werte** liefert

3. Der letzte Punkt der Analogie besagt, dass man die Nummern nicht verlieren oder verwechseln darf. Für ein Objekt muss der über die `hashCode()`–Methode berechnete Hash-Wert **während** einer **Programmausführung** immer **gleich** bleiben.

Suche im HashSet: Erklärung

```
final Collection<Spielkarte> spielkarten = new HashSet<Spielkarte>();

spielkarten.add(new Spielkarte(Spielkarte.Farbe.HERZ, 7));

// gibt's ne Herz 7 ???
final boolean gefunden = spielkarten.contains(new Spielkarte(Spielkarte.Farbe.HERZ, 7));
System.out.println("gefunden="+ gefunden);
```

Für die Klasse Spielkarte wurde `hashCode()` nicht überschrieben.

=> Standard-Implementierung aus der Klasse Object liefert typischerweise als `hashCode()` die Objektreferenz zurück

=> Suche verwendet ein neu erzeugtes Spielkarten-Objekt, dadurch sind die zwei durch `hashCode()` berechneten Hash-Werte unterschiedlich. Demnach wird in zwei verschiedenen Schubladen gesucht.

Zur Korrektur von `hashCode()` schauen wir auf den `hashCode()`-Kontrakt.

Der `hashCode()`-Kontrakt

Der `hashCode()`-Kontrakt (aus Object.java)

```
public int hashCode()
```

Eindeutigkeit: Während der Ausführung einer Java-Applikation muss der Aufruf der Methode `hashCode()` für ein Objekt immer den selben Wert zurückliefern.Bei verschiedenen Ausführungen des Programms darf der Wert unterschiedlich sein.

Verträglichkeit mit `equals()`: Wenn zwei Objekte von equals() als gleich angesehen werden, dann muss die Methode hashCode() für beide Objekte den selben Wert liefern. Umgekehrt gilt dies nicht! Bei gleichem `hashCode()` können zwei Objekte per `equals()` verschieden sein.

Kontraktkonfrome Implemntierung von `hashCode()`

1. **Der Wert für ein Objekt wird aus einigen unveränderlichen Attributen berechnet. Nur unveränderliche Attribute zur Berechnung verwenden, da sich der Wert zur Programmlaufzeit nicht ändern darf.**

2. **Automatisch konsistent zu `equals()` da nur ein Subset der Attribute zur Berechnung verwendet wird.**

```
@Override
public int hashCode()
{
    return Objects.hash(farbe, wert);
}
```

Law Of The Big 3 Live Demo

LawOfBig3Example.java

Übungen 1 - 3

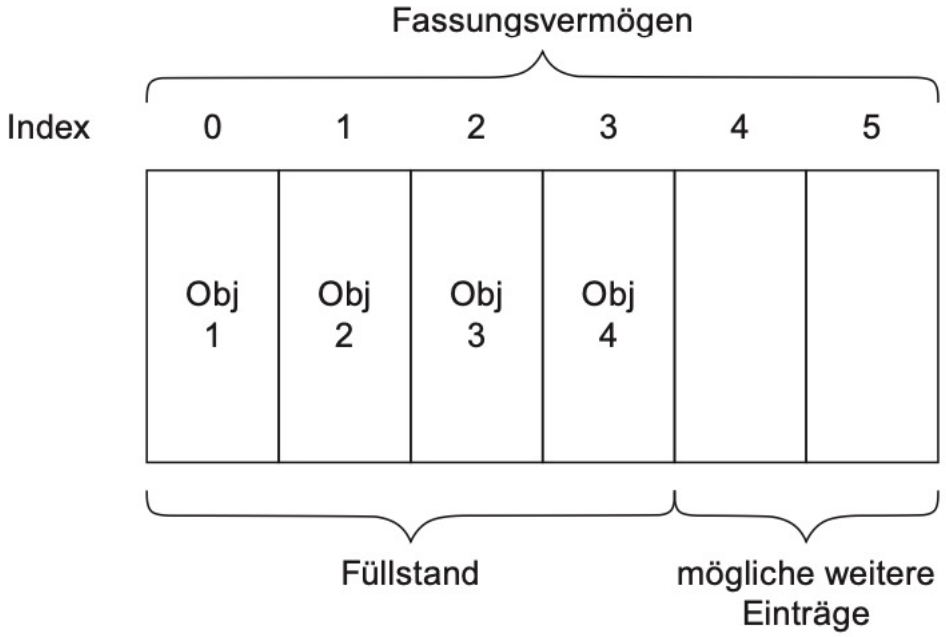
https://github.com/Michaeli71/Tech-Ramp-Up-1-Java-Law-Of-Big-3

PART 4:
Collections Recap

Listen

Listen

- Unter einer Liste versteht man eine über ihre Position geordnete Folge von Elementen – dabei können auch identische Elemente mehrfach vorkommen.
- Die `ArrayList` kann man sich wie ein größenveränderliches Array vorstellen.
- Sie ermöglicht einen indizierten Zugriff und erlaubt das Hinzufügen und Entfernen von Elementen



Listen – Teile extrahieren / Sublisten

- **Teilbereich als Liste liefern (`subList()`) => ACHTUNG: VIEW**

```
jshell> var numbers = new ArrayList<>(List.of(0,1,2,3,4,5,6,7,8,9))
numbers ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

jshell> var first3 = numbers.subList(0, 3)
first3 ==> [0, 1, 2]

jshell> var last4 = numbers.subList(numbers.size() - 4, numbers.size())
last4 ==> [6, 7, 8, 9]

jshell> last4.addAll(List.of(11, 13, 17))
$124 ==> true

jshell> last4
last4 ==> [6, 7, 8, 9, 11, 13, 17]

jshell> numbers
numbers ==> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 17]
```

SubListExample.java

Listen

- **Basisfunktionalitäten in Utility-Klasse `Collections`: `min()` / `max()`**

```
var numbers = new ArrayList<>(List.of(7, 5, 3, 1, 6, 4, 2));

Collections.min(numbers)                    1
Collections.max(numbers)                    7
```

- **Nettigkeiten: `reverse()` / `sort()`**

```
Collections.reverse(numbers);
System.out.println(numbers);                [2, 4, 6, 1, 3, 5, 7]

Collections.sort(numbers);
System.out.println(numbers);                [1, 2, 3, 4, 5, 6, 7]

Collections.shuffle(numbers);
System.out.println(numbers);                [7, 1, 6, 4, 5, 2, 3]
```

Woher weiss Java, wie man Zahlen oder Texte sortiert?

Sortierung

Grundlagen zur Sortierung

- Für Arrays und Listen gibt es **keine Automatik**. Um diese sortiert zu halten, wird ein **manueller Schritt** notwendig => die Methode **`sort()`**.

- **Natürliche Ordnung und `Comparable<T>`** – Objekte können darüber ihre Ordnung, d. h. ihre Reihenfolge untereinander, beschreiben. Diese Reihenfolge wird auch als natürliche Ordnung bezeichnet, da sie durch die Objekte selbst bestimmt wird.

- Teilweise benötigt man zusätzlich zur natürlichen Ordnung weitere oder **alternative Sortierungen**. Diese können mithilfe von Implementierungen des Interface **`Comparator<T>`** festgelegt werden.

Java Comparable

Sortierungen und das Interface `Comparable<T>`

Oftmals besitzen Werte oder Objekte eine natürliche Ordnung: Das gilt etwa für Zahlen und Strings. Für komplexe Typen ist die Aussage »kleiner« bzw. »größer« nicht immer sofort ersichtlich, lässt sich aber bei Bedarf selbst definieren.

Dazu erlaubt das Interface `Comparable<T>` typsichere Vergleiche und deklariert die Methode `compareTo(T)` folgendermaßen:

```
public interface Comparable<T>
{
    public int compareTo(T o);
}
```

Das Vorzeichen des Rückgabewerts bestimmt die Reihenfolge der Elemente:

- $= 0$: Der Wert 0 bedeutet Gleichheit des aktuellen und des übergebenen Objekts.
- < 0: Das aktuelle Objekt ist kleiner als das übergebene Objekt.
- > 0: Das aktuelle Objekt ist größer als das übergebene Objekt.

Diverse Klassen im JDK (alle Wrapper-Klassen, `String`, `Date` usw.) implementieren das Interface `Comparable<T>` und sind damit automatisch sortierbar.

Weitere Ordnungen / Sortierungen und Comparator<T>

- **Ergänzenden Sortierungen können mithilfe von Implementierungen des Interface Comparator<T> festgelegt werden.**

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

- **Vorteile:**
  - Anwendungsklassen nicht überfrachtet, sondern Sortierungen in eigenständigen Vergleichsklassen definiert
  - Es lassen sich von der natürlichen Ordnung abweichende Sortierungen realisieren und auch Objekte von Klassen sortieren, für die keine natürliche Ordnung existiert, weil Comparable<T> nicht implementiert ist.

Vordefinierte Sortierungen in Comparator<T>

- naturalOrder()
- reverseOrder()
- nullsFirst() / nullsLast()

```
jshell> var names = Arrays.asList("A", null, "B", "C", null, "D")
names ==> [A, null, B, C, null, D]

jshell> Comparator<String> naturalOrder = Comparator.naturalOrder();
naturalOrder ==> INSTANCE

jshell> var nullsFirst = Comparator.nullsFirst(naturalOrder);
nullsFirst ==> java.util.Comparators$NullComparator@18e8568

jshell> names.sort(nullsFirst)

jshell> names
names ==> [null, null, A, B, C, D]
```

Java Comparator Fallstrick bei `List.of()`

ACHTUNG: JAVA Fallstrick ...

```
jshell> var names = List.of("Mike", "Andy", "Peter", "Jim", "Tim")
names ==> [Mike, Andy, Peter, Jim, Tim]

names.sort(Comparator.naturalOrder())
|  Exception java.lang.UnsupportedOperationException



jshell> var modifiable = new ArrayList<>(names)
modifiable ==> [Mike, Andy, Peter, Jim, Tim]

jshell> modifiable.sort(Comparator.naturalOrder())

jshell> modifiable
modifiable ==> [Andy, Jim, Mike, Peter, Tim]
```

Natürliche Ordnung / Umgedrehte Sortierung

```
jshell> var modifiable = new ArrayList<>(names)
modifiable ==> [Mike, Andy, Peter, Jim, Tim]

jshell> modifiable.sort(Comparator.naturalOrder())

jshell> modifiable
modifiable ==> [Andy, Jim, Mike, Peter, Tim]


jshell> modifiable.sort(Comparator.reverseOrder())

jshell> modifiable
modifiable ==> [Tim, Peter, Mike, Jim, Andy]
```

Erweiterung in Comparator<T> mit Java 8

- `comparing()` – Definiert einen Komparator basierend auf der Extraktion zweier Werte, die sich mit `Comparable<T>` vergleichen lassen.

- `thenComparing()`, `thenComparingInt()/-Long()` und `-Double()` – Hintereinanderschaltung von Komparatoren

```
Comparator<Person> byName = Comparator.comparing(Person::getName);
Comparator<Person> byAge = Comparator.comparing(Person::getAge);

// Kombination von Komparatoren
Comparator<Person> byNameAndFirstname = byName.
                                        thenComparing(byFirstname);
Comparator<Person> byNameAndAge = byName.thenComparing(byAge);
```

Spezielle Sortierung nach letztem Buchstaben / nach Länge

- **byLastChar**

```
jshell> modifiable.sort(Comparator.comparing(name -> name.charAt(name.length() - 1)))

jshell> modifiable
modifiable ==> [Mike, Tim, Jim, Peter, Andy]
```

- **Nach Länge**

```
jshell> modifiable.sort(Comparator.comparing(String::length))

jshell> modifiable
modifiable ==> [Jim, Tim, Mike, Andy, Peter]
```

Case-Insensitive Sortierung

```
jshell> var names = new ArrayList<>(List.of("tim", "TOM", "MIKE", "Michael",
"andreas", "STEFAN"))
names ==> [tim, TOM, MIKE, Michael, andreas, STEFAN]

jshell> names.sort(Comparator.comparing(String::toLowerCase))

jshell> names
names ==> [andreas, Michael, MIKE, STEFAN, tim, TOM]
```

Spezielle Sortierung nach Länge sowie auch absteigend

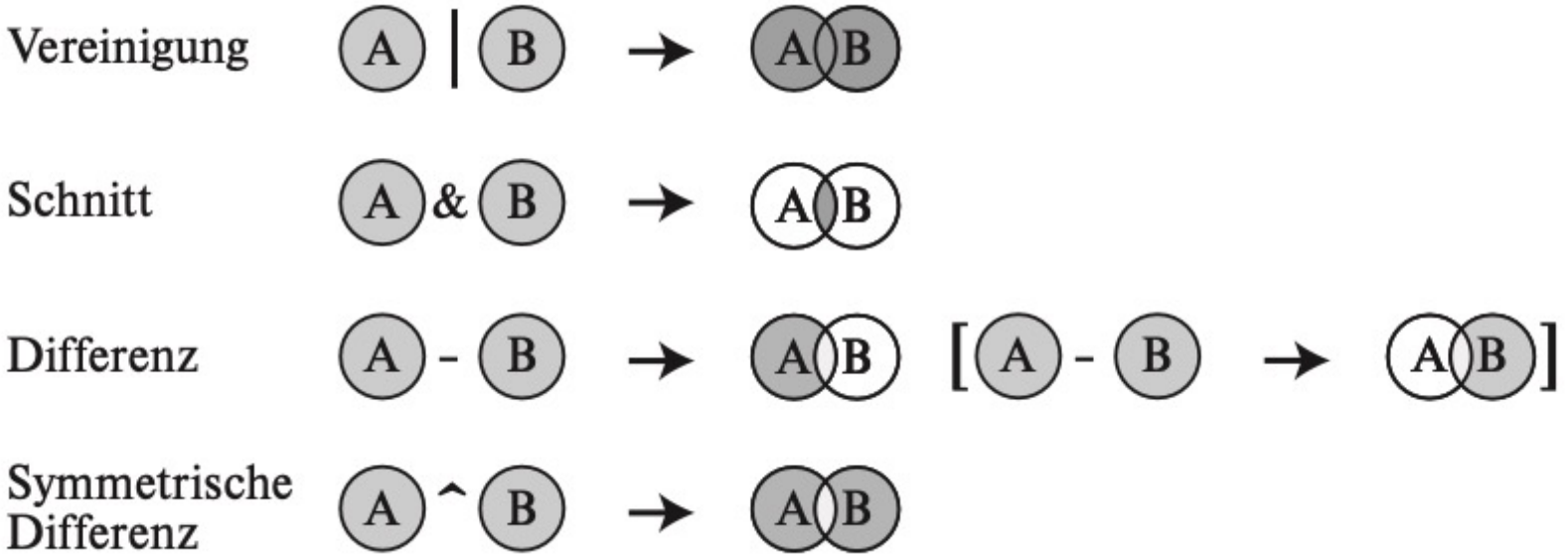
- **Nach Länge auf- und absteigend**

```
jshell> var names = List.of("Tim", "Tom", "Michael", "Andy", "James")
names ==> [Tim, Tom, Michael, Andy, James]

jshell> var modifiable = new ArrayList<>(names)
modifiable ==> [Tim, Tom, Michael, Andy, James]

jshell> modifiable.sort(Comparator.comparing(String::length))

jshell> modifiable
modifiable ==> [Tim, Tom, Andy, James, Michael]

jshell> modifiable.sort(Comparator.comparing(String::length).reversed())


jshell> modifiable
modifiable ==> [Michael, James, Andy, Tim, Tom]
```

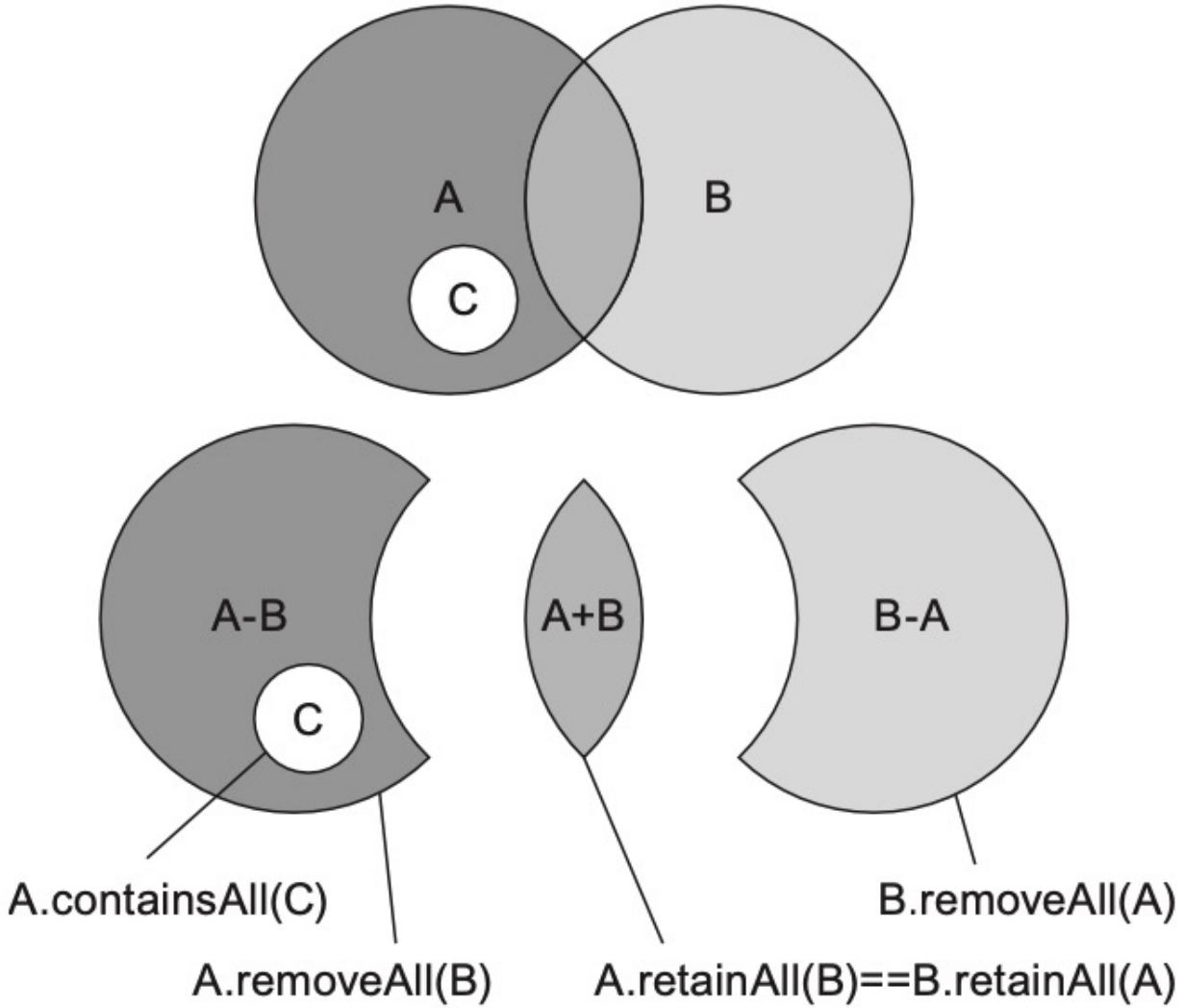
Mengen
(HashSet / TreeSet)

Sets – Basisfunktionalitäten

- Ein `HashSet` ist eine Sammlung (Menge) von Elementen.
- Mathematisches Konzept => keine Duplikate
- Somit bilden Sets eine ungeordnete, duplikatfreie Datenstruktur, bieten **aber keinen** indizierten Zugriff.
- Stattdessen einige Mengenoperationen insbesondere die Berechnung von Vereinigungs-, Schnitt-, Differenz- und symmetrischen Differenzmengen:



Vordefinierte Mengenoperationen



Was ist denn mit Sortierung?

Sortierte Mengen => Klasse `TreeSet`

- Mitunter möchte man die Werte einer Menge sortiert aufbereiten
- Dabei hilft die Klasse `TreeSet`, die automatisch für eine Sortierung sorgt
- Das funktioniert für Zahlen und Texte (für eigene Klassen -> `Comparable`)

```
jshell> var randomNumbers = Set.of(7, 4, 8, 2, 1, 11, 17, 5, 21)
randomNumbers ==> [17, 11, 8, 7, 5, 4, 21, 2, 1]

jshell> var sortedNumbers = new TreeSet<>(randomNumbers)
sortedNumbers ==> [1, 2, 4, 5, 7, 8, 11, 17, 21]


jshell> var cities = Set.of("Kiel", "Aachen", "Bremen", "Oldenburg", "Zürich")
cities ==> [Kiel, Zürich, Aachen, Oldenburg, Bremen]

jshell> var sortedCities = new TreeSet<>(cities)
sortedCities ==> [Aachen, Bremen, Kiel, Oldenburg, Zürich]
```

Schlüssel-Wert-Abbildungen (HashMap / TreeMap)

Maps – Basisfunktionalitäten

- In der `ArrayList` sind Elemente als geordnete Sammlung gespeichert und man kann indiziert (Typ `int`) darauf zugreifen.
- Eine `HashMap` hingegen speichert Abbildungen von Schlüsseln auf Werte, sogenannte »Schlüssel-Wert«-Paare.
- zugrunde liegende Idee, jedem gespeicherten Wert einen eindeutigen Schlüssel zuzuordnen
- Als Beispiel bilden Telefonbücher Namen auf Telefonnummern ab. Es gibt keinen indizierten Zugriff, sondern dieser erfolgt über den Schlüssel.
- Maps werden auch als Dictionary oder als Lookup-Tabelle bezeichnet

Die Klasse HashMap

- **Erzeugen mit new**

```
jshell> var cityInhabitantsMap = new HashMap<String, Integer>()
cityInhabitantsMap ==> {}
```

- **Erzeugen mit Collection-Factory-Methoden**

```
jshell> var unmodifiableMap = Map.of("Bern", 170_000, "Berlin", 3_500_000)
unmodifiableMap ==> {Berlin=3500000, Bern=170000}
```

HashMaps -- Durch die Elemente iterieren

- **Bei `HashMaps` existiert kein indizierter Zugriff**
- **Allerdings lassen sich die Abbildungen mit folgenden Methoden auslesen:**

1. `entrySet()` – Erzeugt eine Menge mit allen Schlüssel-Wert-Paaren.
2. `keySet()` – Liefert eine Menge hinterlegten Schlüsseln.
3. `values()` – Liefert eine Liste hinterlegten Werten.

```
cityInhabitantsMap ==> {Zürich=400000, Kiel=265000, Bremen=550000}

jshell> cityInhabitantsMap.keySet()
$198 ==> [Zürich, Kiel, Bremen]

jshell> cityInhabitantsMap.values()
$199 ==> [400000, 265000, 550000]

jshell> cityInhabitantsMap.entrySet()
$200 ==> [Zürich=400000, Kiel=265000, Bremen=550000]
```

Was ist denn mit Sortierung?

Sortierte Maps => Klasse `TreeMap`

- Mitunter möchte man die Schlüssel einer Map sortiert aufbereiten
- Dabei hilft die Klasse `TreeMap`, die automatisch für eine Sortierung sorgt
- Das funktioniert für Zahlen und Texte (für eigene Klassen -> `Comparable`)

```
jshell> var cityInhabitants = Map.of("Kiel", 265000, "Zürich", 400000,
"Berlin", 3500000)
cityInhabitants ==> {Kiel=265000, Zürich=400000, Berlin=3500000}

jshell> var sortedMap = new TreeMap<>(cityInhabitants)
sortedMap ==> {Berlin=3500000, Kiel=265000, Zürich=400000}
```

Maps als Lookup-Map: Römische Zahlen Buchstaben auf Wert

- **Wert ergibt sich normalerweise aus der Addition der Werte der einzelnen Ziffern von links nach rechts beispielsweise XVI für den Wert 16.**

```
var valueMap = Map.of('I', 1, 'V', 5, 'X', 10, 'L', 50, 'C', 100, 'D', 500, 'M', 1000);
```

1. **Additionsregel**: Gleiche Ziffern nebeneinander werden addiert, etwa XXX = 30 Ebenso gilt dies für kleinere Ziffern nach größeren, z. B. XII = 12.
2. **Wiederholungsregel**: Es dürfen maximal drei gleiche Ziffern aufeinanderfolgen. Nach Regel 1 könnte man die Zahl 4 als IIII schreiben, was diese Regel 2 verbietet. Hier kommt die Subtraktionsregel ins Spiel.
3. **Subtraktionsregel**: Steht ein kleineres Zahlzeichen vor einem größeren, so wird der entsprechende Wert subtrahiert. Schauen wir nochmals auf die 4: Diese kann man als Subtraktion $5 - 1$ realisieren. Das wird im römischen Zahlensystem als IV notiert. Für die Subtraktion gelten folgende Regeln:
   - I steht nur vor V und X
   - X steht nur vor L und C
   - C steht nur vor D und M

Maps als Lookup-Map

```
public static int fromRomanNumber(final String romanNumber)
{
    int value = 0;
    int lastDigitValue = 0;

    for (int i = romanNumber.length() - 1; i >= 0; i--)
    {
        final char romanDigit = romanNumber.charAt(i);
        final int digitValue = valueMap.getOrDefault(romanDigit, 0);

        final boolean addMode = digitValue >= lastDigitValue;
        if (addMode)
        {
            value += digitValue;
            lastDigitValue = digitValue;
        }
        else
        {
            value -= digitValue;
        }
    }
    return value;
}
```

RomanNumbers.java

Übungen 4 – 5 (ggf. Magic Triangle)

https://github.com/Michaeli71/Tech-Ramp-Up-1-Java-Law-Of-Big-3

PART 5:
Iteratoren Recap

Iteratoren – Grundlagen

- **Möglichkeit verschiedene Datenstrukturen auf gleiche Art zu durchlaufen**
- **Abstraktion von der zu traversierenden Datenstruktur:**
  - Keine Kenntnis des internen Aufbaus nötig
  - Bäume lassen sich durchlaufen wie Listen oder Sets
- **Ein Iterator ist eine Art Zeiger, der auf das aktuelle Element verweist und das nächste Element eines Datencontainers liefern kann.**
- **Alle Datenstrukturen, die das Interface `Collection<E>` erfüllen, bieten über die Methode `iterator()` Zugriff auf das Interface `java.util.Iterator<E>`, dass zwei Methoden bietet:**
  - `hasNext()` und `next()`.

Iteratoren im korrekten Einsatz / typisches EInsatzmuster

```
jshell> var names = List.of("Tim", "Tom", "Mike", "Andy")
names ==> [Tim, Tom, Mike, Andy]

jshell> Iterator<String> it = names.iterator()
it ==> java.util.ImmutableCollections$ListItr@17d10166

jshell> while (it.hasNext())
   ...> {
   ...>     String nextValue = it.next();
   ...>     System.out.println(nextValue);
   ...> }
Tim
Tom
Mike
Andy
```

Anwendungsbeispiel

```
public final class NullIterator<E> implements Iterator<E>
{
    public boolean hasNext()
    {
        return false;
    }


    public E next()
    {
        throw new NoSuchElementException("NullIterator provides no elements!");
    }

    public void remove()
    {
        throw new UnsupportedOperationException("NullIterator does not " +
                                                "implement remove()!");
    }
}
```

ArrayIterator selbstgestrickt


```
public class ArrayIteratorAdapter implements Iterator<T>
{
   int currentPosition;
   final T[] adaptee;

   public ArrayIteratorAdapter(final T[] adaptee)
   {
       this.adaptee = adaptee;
       this.currentPosition = 0;
   }

   // TODO
   public boolean hasNext()
   public T next()
}
```

ArrayIterator – Beispiellösung

```
public class ArrayIteratorAdapter<T> implements Iterator<T>
{
    int currentPosition;
    final T[] adaptee;

    public ArrayIteratorAdapter(final T[] adaptee) {
        this.adaptee = adaptee;
        this.currentPosition = 0;
    }

    @Override
    public boolean hasNext() {
        return currentPosition < adaptee.length;
    }

    @Override
    public T next() {
        final T next = adaptee[currentPosition];
        currentPosition++;
        return next;
    }
}
```

Random-Iterator

```
public class RandomListIterator<E> implements Iterator<E>
{
    private int     currentPos  = 0;
    private List<E> suffledList = List.of();

    public RandomListIterator(List<E> original)
    {
        this.suffledList = new ArrayList<>(original);
        Collections.shuffle(suffledList);
    }

    @Override
    public boolean hasNext()
    {
        return currentPos < suffledList.size();
    }

    @Override
    public E next()
    {
        E currentElement = suffledList.get(currentPos);

        currentPos++;

        return currentElement;
    }
}
```

RandomListIterator.java

Eigene Iteratoren (zur Fakultätsberechnung)

```
jshell> class FactorialIterator implements Iterator<Integer>
   ...> {
   ...>     private int n;
   ...>     private int result = 1;
   ...>     private int iteration = 1;
   ...>
   ...>     public FactorialIterator(int n) {
   ...>         this.n = n;
   ...>     }
   ...>
   ...>     public boolean hasNext() {
   ...>         return iteration <= n;
   ...>     }
   ...>
   ...>     public Integer next() {
   ...>         result *= iteration;
   ...>         iteration++;
   ...>         return result;
   ...>     }
   ...> }
|  created class FactorialIterator
```

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

$$\begin{aligned} 0! &= 1 &&= 1 \\ 1! &= 1 &&= 1 \\ 2! &= 1 \cdot 2 &&= 2 \\ 3! &= 1 \cdot 2 \cdot 3 &&= 6 \\ 4! &= 1 \cdot 2 \cdot 3 \cdot 4 &&= 24 \\ 5! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 &&= 120 \end{aligned}$$

Eigener Iterator im Einsatz

- **Handgestrickt**

```
jshell> Iterator<Integer> fac = new FactorialIterator(5)
fac ==> FactorialIterator@504bae78

jshell> fac.next()
$13 ==> 1

jshell> fac.next()
$14 ==> 2

jshell> fac.next()
$15 ==> 6

jshell> fac.next()
$16 ==> 24

jshell> fac.next()
$17 ==> 120

jshell> fac.next()
$18 ==> 720
```

Mit Schleife

```
jshell> var fac = new FactorialIterator(5)
fac ==> FactorialIterator@5eb5c224

jshell> while (fac.hasNext())
   ...>     System.out.println(fac.next())
1
2
6
24
120
```

Übungen 6 - 7

https://github.com/Michaeli71/Tech-Ramp-Up-1-Java-Law-Of-Big-3

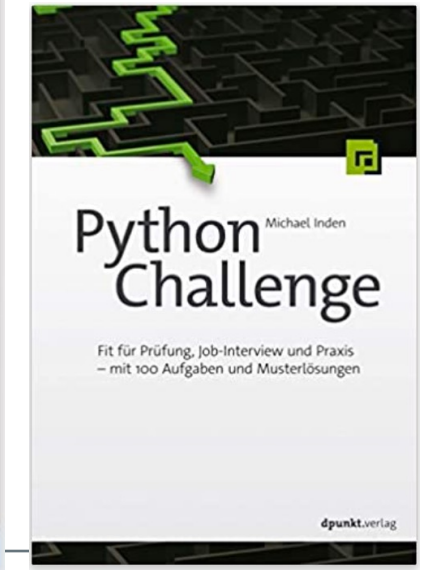
Thank You


HELP
IS ON THE
WAY


Michael Inden
Einfach Java
Gleich richtig programmieren lernen
dpunkt.verlag


5. Auflage
Michael Inden
Der Weg zum Java-Profi
Konzepte und Techniken für die professionelle Java-Entwicklung
dpunkt.verlag


Michael Inden
Java Challenge
Fit für das Job-Interview und die Praxis – mit mehr als 100 Aufgaben und Musterlösungen
dpunkt.verlag


Michael Inden
Python Challenge
Fit für Prüfung, Job-Interview und Praxis – mit 100 Aufgaben und Musterlösungen
dpunkt.verlag