



Java Intro

Michael Inden

Freiberuflicher Consultant und Trainer

https://github.com/Michaeli71/JAVA_INTRO

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**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@hotmail.ch

Blog: <https://jaxenter.de/author/minden>

Kurse: **Bitte spricht mich an!**

https://github.com/Michaeli71/JAVA_INTRO





Agenda



- **PART 1: Schnelleinstieg Java**
 - Erste Schritte in der JShell
 - Schnelleinstieg
 - Variablen
 - Operatoren
 - Fallunterscheidungen
 - Schleifen
 - Methoden
 - Rekursion



- **PART 2: Strings**
 - Gebräuchliche String-Aktionen
 - Suchen und Ersetzen
 - Formatierte Ausgaben
 - Einstieg Reguläre Ausdrücke
 - Mehrzeilige Strings
- **PART 3: Arrays**
 - Gebräuchliche Array-Aktionen
 - Mehrdimensionale Arrays
 - Beispiel: Flood Fill



- **PART 4: Klassen & Objektorientierung**
 - Basics
 - Textuelle Ausgaben
 - Gleichheit == / equals()
 - Klassen ausführbar machen
 - Imports & Packages
 - Information Hiding
 - Vererbung und Overloading und Overriding
 - Die Basisklasse Object
 - Interfaces & Implementierungen
 - Records



- **PART 5: Collections**
 - Schnelleinstieg Listen, Sets und Maps
 - Iteratoren
 - Generics
 - Basisinterfaces für Container
 - Praxisbeispiel Stack und Queue selbst gebaut
 - Sortierung – `sort()` + Comparator



- **PART 6: Ergänzendes Wissen**
 - Sichtbarkeits- und Gültigkeitsbereiche
 - Primitive Typen und Wrapper
 - Enums
 - ?-Operator
 - switch
 - Besonderheiten in Schleifen break und continue
 - Vererbung und Polymorphie
 - Varianten innerer Klassen



- **PART 7: Einstieg in Lambdas und Streams**

- Syntax von Lambdas
- Lambdas im Einsatz mit `filter()`, `map()` und `reduce()`
- Lambdas im Einsatz mit `Collectors.groupingBy()`
- `takeWhile()` / `dropWhile()`

- **PART 8: Datumsverarbeitung**

- Einführung Datumsverarbeitung
- Zeitpunkte und die Klasse `LocalDateTime`
- Datumswerte und die Klasse `LocalDate`
- Zeit und die Klasse `LocalTime`



- **PART 9: Exception-Handling**
 - Schnelleinstieg
 - Exceptions selbst auslösen
 - Eigene Exception-Typen definieren
 - Propagation von Exceptions
 - Automatic Resource Management
 - Checked / Unchecked Exceptions
- **PART 10: Dateiverarbeitung**
 - Verzeichnisse und Dateien verwalten
 - Daten schreiben / lesen



Part 9

Einstieg in Lambdas & Streams



Warum Lambdas?



Lambdas als ein neues und heiß ersehntes Sprachkonstrukt

Warum Lambdas?



- **Schön**, bereits seit langem in Sprachen wie Groovy und Scala
- Als Krücke auch in Java!
Wirklich, aber wie?



- Lösungen auf sehr elegante Art und Weise formulieren
- **andere Denkweise und neuer Programmierstil (funktional)**
- Hilfe für Parallelverarbeitung und Ausnutzung von Multicores



- Lambda: eine spezielle Art von Methode bzw. ein Stück Code mit einfacher Syntax:

Parameter-Liste -> Ausdruck oder Anweisungen
(String name) -> name.length()

- aber ...
 - ohne Namen (ad-hoc und anonym)
 - ohne Angabe eines Rückgabetyps (wird vom Compiler ermittelt)
 - ohne Deklaration von Exceptions (wird vom Compiler ermittelt)

Beispiele für Lambdas



```
(int x) -> { return x + 1; }           // Typed Param, Statement
(int x) -> x + 1                       // Typed Param, Expression
(x, y)  -> { x = x / 2; return x * y; } // Untyped Param, Multi Statements
it      -> it.startsWith("M")
()      -> System.out.println("no param") // No Param, No Return
```

Ein Lambda ist (NICHT DIREKT) **KEIN** Object zuweisbar



- Lambdas besitzen keinen Obertyp wie in Groovy etwa den Typ **Closure**
- **Können nicht dem Typ **Object** zugewiesen werden**

```
Object lambda = () -> System.out.println("compile-error");
```

"The target type of this expression must be a functional interface"

- Was ist denn nun ein Functional Interface?

Beispiele für Functional Interfaces (SAM-Typen)



Viele bekannt aus Funk und Fernsehen ... äh ... dem JDK

- `Runnable`, `Callable`, `Comparable`, `Comparator`, `FileFilter`, `FilenameFilter`, `ActionListener`, `ChangeListener` usw.

Neue Annotation **@FunctionalInterface** (Angabe optional)

@FunctionalInterface

```
public interface Runnable {  
    public abstract void run();  
}
```

@FunctionalInterface

```
public interface FileFilter {  
    boolean accept(File pathname);  
}
```



- Lambdas als Implementierung eines Functional Interface:

```
new SAMTypeAnonymousClass()  
{  
    public void samMethod(METHOD-PARAMETERS)  
    {  
        METHOD-BODY  
    }  
}
```

<=>

(METHOD-PARAMETERS) -> { METHOD-BODY }



- **Definition einfacher Lambdas**

```
jshell> IntUnaryOperator addOne = x -> x + 1  
addOne ==> $Lambda$21/0x0000000800c09c10@5b6f7412
```

```
jshell> IntUnaryOperator doubleIt = x -> x * 2  
doubleIt ==> $Lambda$22/0x0000000800c0ba08@7530d0a
```

- **Aufruf durch die Methode des Functional Interface**

```
jshell> addOne.applyAsInt(7)  
$10 ==> 8
```

```
jshell> doubleIt.applyAsInt(21)  
$11 ==> 42
```

```
@FunctionalInterface  
public interface IntUnaryOperator {  
  
    int applyAsInt(int operand);  
  
    ...  
}
```



Beispiele

```
Runnable runner = () -> { System.out.println("Hello Lambda"); };
```

```
Predicate<String> isLongWord = (String word) -> { return word.length() > 15; };
```

```
Comparator<String> byLength = (str1, str2) -> Integer.compare(str1.length(),  
                                                                str2.length());
```



- **Definition wie zuvor:**

```
jshell> Predicate<String> isLongWord = (String word) -> { return word.length()  
> 15; };  
isLongWord ==> $Lambda$29/0x0000000800c0c858@34c45dca
```

```
jshell> Comparator<String> byLength = (str1, str2) ->  
Integer.compare(str1.length(), str2.length());  
byLength ==> $Lambda$30/0x0000000800c0ccb0@27973e9b
```

- **Aufruf durch die Methode des Functional Interface**

```
jshell> isLongWord.test("Fahrgastinformation")  
$17 ==> true
```

```
jshell> byLength.compare("MIKE", "Michael")  
$18 ==> -1
```



- Lambdas als Rückgabewerte

```
public static Comparator<String> byLength() {  
    return (str1, str2) -> Integer.compare(str1.length(), str2.length());  
}
```

- Lambdas als **Eingabe** bzw. Parameter

```
List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");  
  
Collections.sort(names, (str1, str2) -> Integer.compare(str1.length(),  
                                                         str2.length()));  
  
Collections.sort(names, byLength());
```

Damals ... Sortierung nach Länge und komma-separierte Ausgabe



- Ohne Lambdas (JDK 7) erfolgte das in etwa so:

```
List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");
Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return Integer.compare(str1.length(), str2.length());
    }
});
```

```
Iterator<String> it = names.iterator();
while (it.hasNext()) {
    System.out.print(it.next().length() + ", ");
}
// => 3, 4, 6, 7,
```

Lambdas im Einsatz: Sortierung und komma-separierte Ausgabe



- Mit JDK 8 und Lambdas schreibt man das kürzer wie folgt:

```
List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");
```

```
names.sort((str1, str2) -> Integer.compare(str1.length(), str2.length()));
```

```
names.forEach( it -> System.out.print(it.length() + ", ") );
```

```
// => 3, 4, 6, 7,
```

- Bei gleicher Ausgabe 12 : 3 Zeilen, Verhältnis 4:1 (alt:neu)
- Aber Moment ...



- `sort()` und `forEach()` ... auf `List`? Wo kommen diese denn her?
- Interfaces können seit Java 8 auch Defaultmethoden enthalten

```
public interface List<E> extends Collection<E> {  
    ...  
    default void sort(Comparator<? super E> c) {  
        Collections.sort(this, c);  
    }  
}
```

```
public interface Iterable<T> {  
    ...  
    default void forEach(Consumer<? super T> action) {  
        for (T t : this) {  
            action.accept(t);  
        }  
    }  
}
```



Beispiel Interface Comparator<T>

```
public static <T extends Comparable<? super T>> Comparator<T> reverseOrder()  
{  
    return Collections.reverseOrder();  
}
```

```
public static <T extends Comparable<? super T>> Comparator<T> naturalOrder()  
{  
    return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;  
}
```



- Methodenreferenz verweist auf ...

- Methoden: a) Instanz-Methoden `System.out::println`, `Person::getName`, ...
`String::compareTo` => `public int compareTo(String anotherString)`

- b) statische Methoden: `System::currentTimeMillis`

- Konstruktor: `ArrayList::new`, `Person[]::new`

-

Methodenreferenz kann anstelle eines Lambda-Ausdrucks genutzt werden

```
List<String> names = Arrays.asList("Max", "Andy", "Michael", "Stefan");
```

```
names.forEach(it -> System.out.println(it));    // Lambda  
names.forEach(System.out::println );           // Methodenreferenz
```



Bulk Data Operations



Externe Iteration vs interne Iteration



- **Extern mit Iterator**

```
Iterator<String> it = names.iterator();  
while (it.hasNext()) {  
    String value = it.next();  
    System.out.println(value);  
}
```

- **Intern mit forEach**

```
names.forEach(System.out::println);
```



- Predicate<T> -- Bedingungen formulieren

```
Predicate<String> isEmpty = String::isEmpty;  
Predicate<String> isShortWord = word -> word.length() <= 3;  
Predicate<String> notIsShortWord = isShortWord.negate();  
Predicate<String> notIsEmptyAndIsShortWord = isEmpty.negate().and(isShortWord);
```

- Collection.removeIf()

```
List<String> names = new ArrayList<>(Arrays.asList("Tim", "Tom", "Andy", "Mike"));  
names.removeIf(isShortWord)  
names.forEach(System.out::println);    => Andy Mike
```



- UnaryOperator<T> -- Aktionen formulieren

```
UnaryOperator<String> nullToEmpty = str -> str == null ? "" : str;  
UnaryOperator<String> trimmer = String::trim;
```

- Collection.replaceAll() -- Aktionen ausführen

```
List<String> names = new ArrayList<>(Arrays.asList("Tim", null, " Tom ",  
                                                    "  Andy", "Mike"));

names.replaceAll(nullToEmpty);
names.replaceAll(trimmer);
names.forEach(s -> System.out.print("'" + s + "'", "));
=> 'Tim', '', 'Tom', 'Andy', 'Mike',
```



Stream API

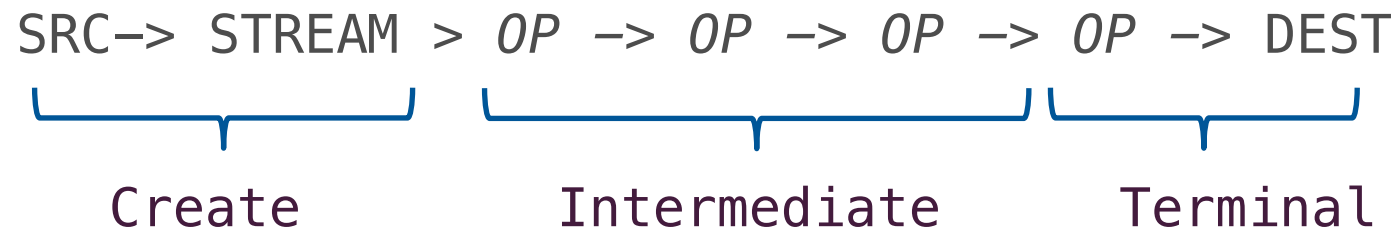


Was sind Streams?



- Streams als neue Abstraktion für Folgen von Verarbeitungsschritten
- Analogie Collection, aber keine Speicherung der Daten
- Analogie Iterator, Traversierung, aber weitere Möglichkeiten zur Verarbeitung

Design der Abarbeitung als Pipeline oder Fließband



```
List<Person> adults = persons.stream() .           // Create  
                        filter(Person::isAdult) .   // Intermediate  
                        collect(Collectors.toList()); // Terminal
```

Streams – Create-Operations



Aus Arrays oder Collections: `stream()`, `parallelStream()`*

```
String[] namesData = { "Karl", "Ralph", "Andi", "Andi«, "Mike" };  
List<String> names = Arrays.asList(namesData);  
Stream<String> streamFromArray = Arrays.stream(namesData);  
Stream<String> streamFromList = names.parallelStream();
```

Für definierte Wertebereiche: `of()`, `range()`

```
Stream<Integer> streamFromValues = Stream.of(17, 23, 3, 11, 7, 5, 14, 9);  
IntStream values = IntStream.range(0, 100);  
IntStream chars = "This is a test".chars();
```

*Umschaltung **sequentiell** <-> **parallel** nach jedem Schritt der Pipeline **möglich**, **aber letzter gewinnt**

Streams – Intermediate- und Terminal-Operations



- Intermediate-Operations

- beschreiben Verarbeitung, sind aber LAZY (führen nichts aus!)
- erlauben es, Verarbeitung auf spezielle Elemente zu beschränken
- geben Streams zurück und erlauben so Stream-Chaining

```
final Stream<Person> allAdultMikes = persons.stream().  
    filter(Person::isAdult).  
    filter(person -> person.getName().equals("Mike")).  
    filter(mike -> mike.livesIn("Zürich"));
```

- Terminal-Operations

- sind EAGER und führen zur Abarbeitung der Pipeline
- produzieren Ergebnis: Ausgabe oder Sammlung in Collection usw.

```
streamFromValues.filter(n -> n < 9).sorted().forEach(System.out::println); // 3 5 7
```



Java-Stream-API an Beispielen



Streams – Intermediate- und Terminal-Operations



```
var countries = List.of("USA", "Schweiz", "Deutschland", "Frankreich", "Italien");
```

```
Function<String, String> asReversed =  
    str -> new StringBuilder(str).reverse().toString();
```

```
countries.stream().  
    filter(str -> str.contains("i")).  
    map(asReversed).  
    forEach(System.out::println);
```

```
ziewhcS  
hcierknarF  
neilatI
```



- **Record**

```
jshell> record Person(String name, int age) {}  
| created record Person
```

```
jshell> var persons = List.of(new Person("Mike", 50), new Person("Tim", 50),  
    ...> new Person("Tom", 7), new Person("Jim", 30))  
persons ==> [Person[name=Mike, age=50], Person[name=Tim, age= ...  
Person[name=Jim, age=30]]
```

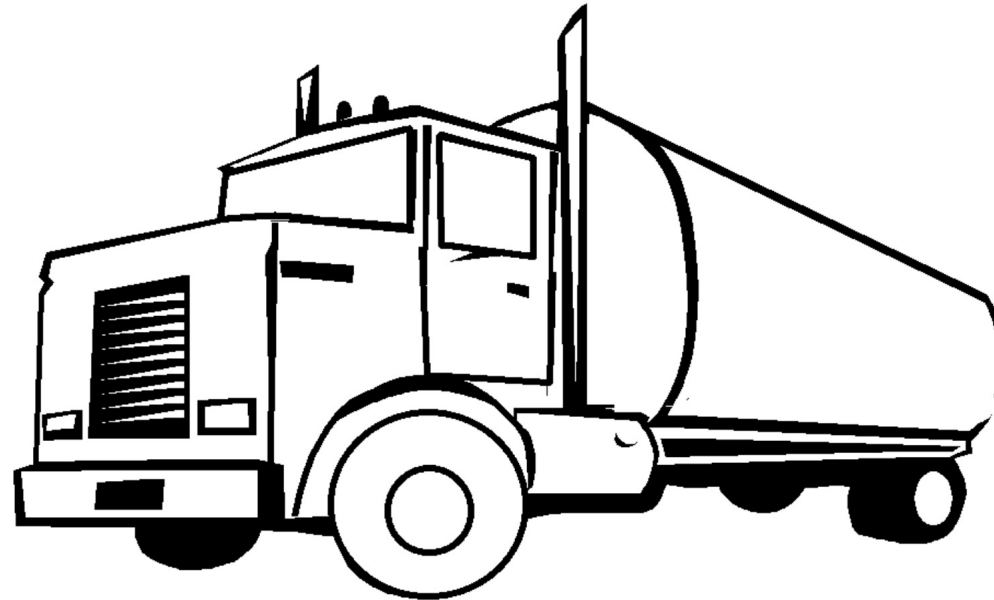
- **Filtering / Mapping**

```
jshell> persons.stream().filter(person -> person.age() > 30).toList()  
$7 ==> [Person[name=Mike, age=50], Person[name=Tim, age=50]]
```

```
jshell> persons.stream().map(person -> person.age() / 2).toList()  
$8 ==> [25, 25, 3, 15]
```



Terminal-Operations an Beispielen





- **forEach(), collect(), toList()**

```
jshell> var cities = List.of("Zürich", "Kiel", "Bremen")  
cities ==> [Zürich, Kiel, Bremen]
```

```
jshell> cities.stream().forEach(System.out::println)  
Zürich  
Kiel  
Bremen
```

```
jshell> cities.stream().collect(Collectors.toList())  
$14 ==> [Zürich, Kiel, Bremen]
```

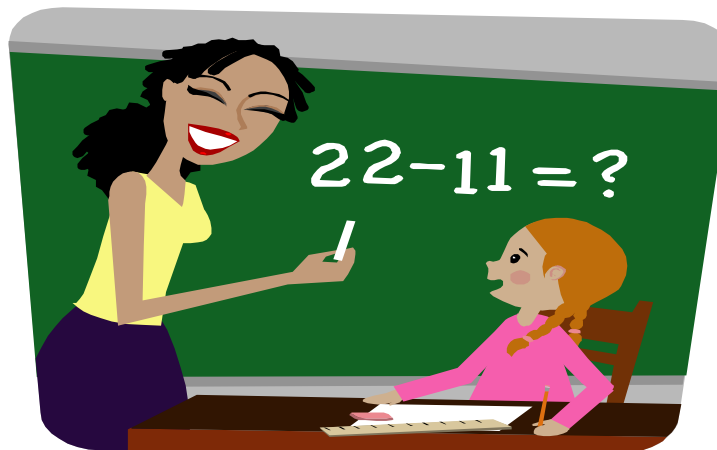
```
jshell> cities.stream().toList()  
$15 ==> [Zürich, Kiel, Bremen]
```

Terminal-Operations – Collectors.joining, groupingBy, partitioningBy



```
List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",  
                                   "Florian", "Michael", "Sebastian");
```

```
String joined = names.stream().sorted().collect(Collectors.joining(", "));  
Object grouped = names.stream().collect(groupingBy(String::length));  
Object grouped2 = names.stream().collect(groupingBy(String::length, counting()));  
Object partition = names.stream().filter(str -> str.contains("i")).  
    collect(partitioningBy(str -> str.length() > 4));
```



Terminal-Operations – Collectors.joining, groupingBy, partitioningBy



```
List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",  
                                   "Florian", "Michael", "Sebastian");
```

```
String joined = names.stream().sorted().collect(Collectors.joining(", "));  
Object grouped = names.stream().collect(groupingBy(String::length));  
Object grouped2 = names.stream().collect(groupingBy(String::length, counting()));  
Object partition = names.stream().filter(str -> str.contains("i")).  
                        collect(partitioningBy(str -> str.length() > 4));
```

joined: Andi, Florian, Michael, Mike, Ralph, Sebastian, Stefan

grouped: {4=[Andi, Mike], 5=[Ralph], 6=[Stefan], 7=[Florian, Michael],
 9=[Sebastian]}

grouped2: {4=2, 5=1, 6=1, 7=2, 9=1}

partition: {false=[Andi, Mike], true=[Florian, Michael, Sebastian]}

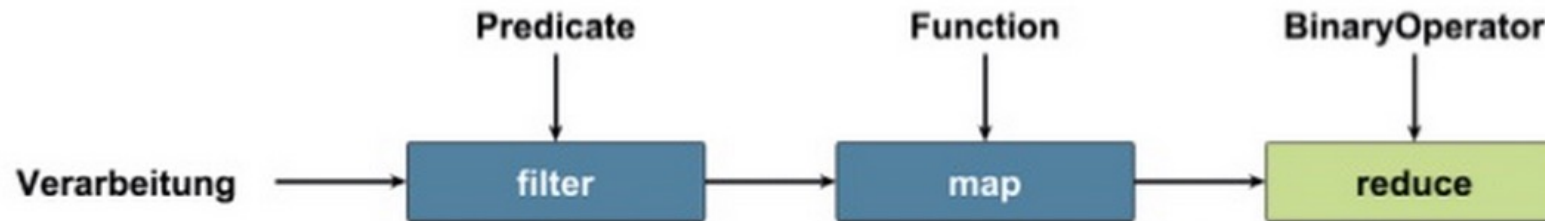
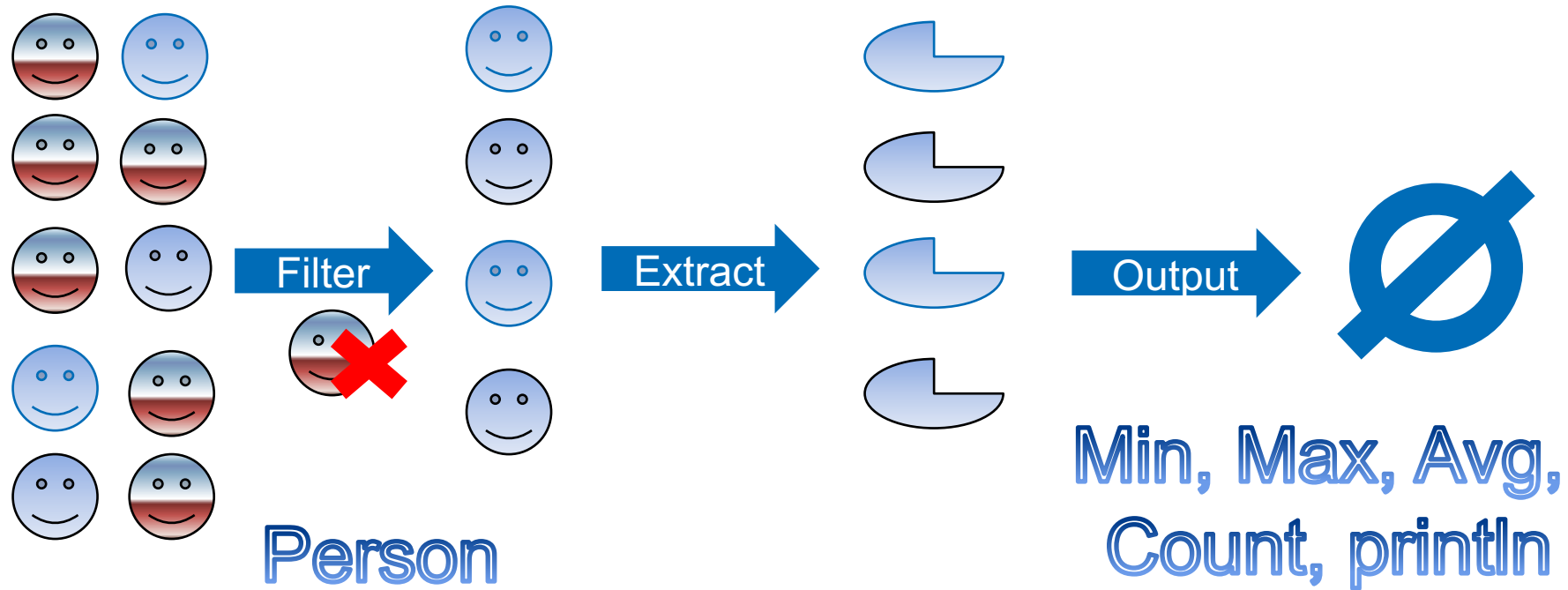


Java-Stream-API

Praxisbeispiel



Filtere eine Liste und extrahiere Daten



Aufgabenstellung: Filtere eine Liste und extrahiere Daten



Gegeben sei folgende List<Person>:

```
List<Person> persons = Arrays.asList(  
    new Person("Stefan", LocalDate.of(1971, Month.MAY, 20)),  
    new Person("Micha", LocalDate.of(1971, Month.FEBRUARY, 7)),  
    new Person("Andi Bubolz", LocalDate.of(1968, Month.JULY, 17)),  
    new Person("Andi Steffen", LocalDate.of(1970, Month.JULY, 17)),  
    new Person("Merten", LocalDate.of(1975, Month.JUNE, 14)));
```

Aufgabe:

1. Filtere auf alle im Juli Geborenen
2. Extrahiere ein Attribut, z.B. den Namen
3. Berechne eine kommaseparierte Liste auf

Herkömmlicher Ansatz: Alles einzeln ausprogrammieren



1. Filtere auf alle im Juli Geborenen

```
List<Person> bornInJuly = new ArrayList<>();  
for (Person person : persons) {  
    if (person.birthday.getMonth() == Month.JULY) {  
        bornInJuly.add(person);  
    }  
}
```

2. Extrahiere ein Attribut, z. B. den Namen

```
List<String> names = new ArrayList<>();  
for (Person person : bornInJuly) {  
    names.add(person.name);  
}
```

Herkömmlicher Ansatz: Alles einzeln ausprogrammieren



3. Bereite eine kommaseparierte Liste auf

```
String result = "";
Iterator<String> it = names.iterator();
while (it.hasNext())
{
    result += it.next();
    if (it.hasNext()) {
        result += ", ";
    }
}
```

=> Andi Bubolz, Andi Steffen



JDK 8-Lösung: Filter-Map-Reduce und Lambdas einsetzen



1. Filter: Filtere auf alle im Juli Geborenen

```
String result = persons.stream().  
                        filter(person -> person.birthday.getMonth() ==  
                                      Month.JULY).
```

2. Map: Extrahiere ein Attribut, z.B. den Namen

```
map(person -> person.name).
```

3. Reduce: Berechne eine kommaseparierte Liste auf

```
reduce("", (str1, str2) -> { if (str1.isEmpty()) {  
                            return str2;  
                            } else {  
                                return str1 + ", " + str2;  
                            }} );
```

JDK 8-Lösung: Filter-Map-Reduce und Lambdas einsetzen



1. Filter: Filtere auf alle im Juli Geborenen
2. Map: Extrahiere ein Attribut, z.B. den Namen
3. Berechne eine kommaseparierte Liste auf: Ersetze reduce() durch collect() und nutze Collectors

```
String result = persons.stream().  
    filter(person -> person.birthday.getMonth() ==  
        Month.JULY).  
    map(person -> person.name).  
    collect(Collectors.joining(", "));
```



Aufgabe: Ermittle alle Zeilen aus einer Log-Dateien die den Text «Error» enthalten, beschränke die Treffermenge auf die ersten 10 Vorkommen

```
List<String> errorLines = new ArrayList<>();
try (BufferedReader reader = new BufferedReader(new FileReader(inputFile))) {
    String currentLine = reader.readLine();
    while (errorLines.size() < maxCount && currentLine != null) {
        if (currentLine.contains("ERROR")) {
            errorLines.add(currentLine);
        }
        currentLine = reader.readLine();
    }
}
return errorLines;
```

- Nutzt externe Iteration
- Vielmehr Code als eigentlich zu erwarten, viel Glue Code
- Zugrundeliegender Algorithmus / Aufgabe kaum ersichtlich



JDK 8-Realisierung deutlich einfacher:

```
final List<String> errorLines = Files.lines(inputFile.toPath())  
    .filter(line -> line.contains("ERROR"))  
    .limit(maxCount)  
    .collect(Collectors.toList());  
  
return errorLines;
```

- Nutzt interne Iteration
- Nahezu kein Glue Code, sondern nur relevanter Code
- Zugrundeliegender Algorithmus / Aufgabe klar ersichtlich und gut lesbar



- **Maps arbeiten nicht mit Streams ;-(**
- Aber ... Das Interface **Map** wurde um eine Vielzahl an Methoden erweitert, die das Leben erleichtern:
 - `forEach()`
 - `putIfAbsent()`
 - `computeIfPresent()`
 - `getOrDefault()`

Map-Abhilfen im Überblick



```
final Map<String, Integer> map = new TreeMap<>();
map.put("c", 3);
map.put("b", 2);
map.put("a", 1);

final StringBuilder result = new StringBuilder();
map.forEach((key, value) -> result.append("(" + key + ", " + value + ") "));
System.out.println(result);

System.out.println(map.getOrDefault("XXX", -4711));
map.putIfAbsent("XXX", 7654321);
map.computeIfPresent("XXX", (key, value) -> value + 123456);
System.out.println(map.getOrDefault("XXX", -4711));
```

=>

```
(a, 1) (b, 2) (c, 3)
-4711
7777777
```



Stream API mit Java 9





Das umfangreiche *Stream-API* war eine *der wesentlichen Neuerungen in Java 8*

takeWhile(...)

dropWhile(...)

ofNullable(...)

iterate(... , ..., ...)



`takeWhile(...)`

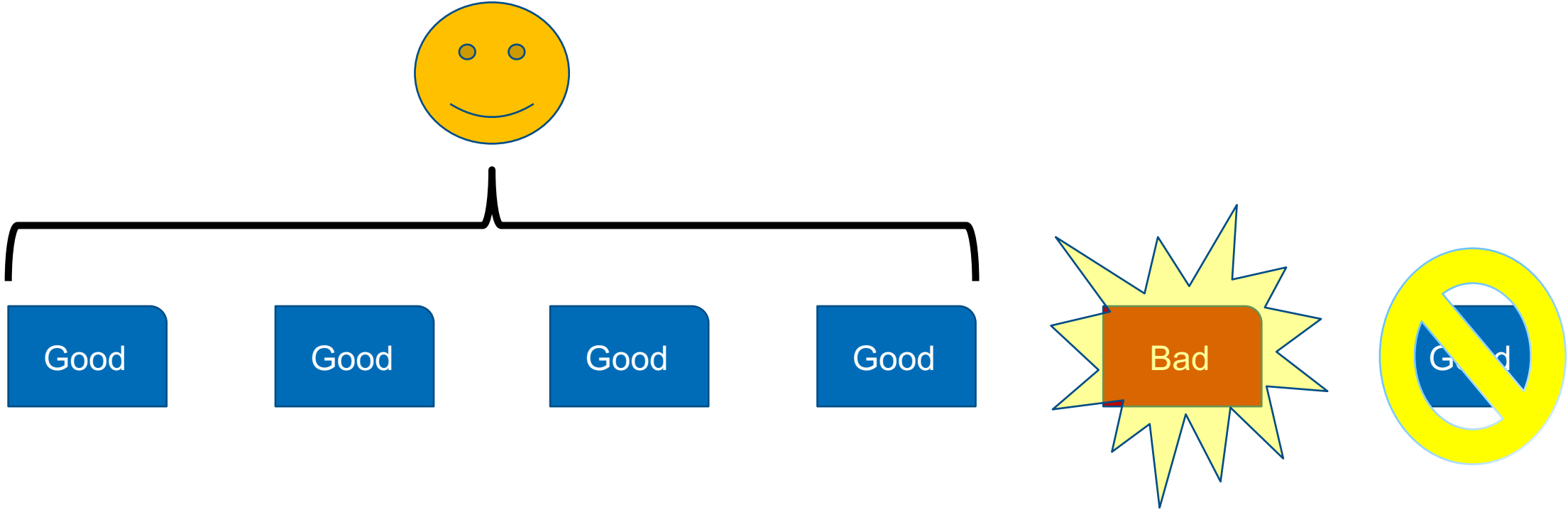
`takeWhile(Predicate<T>)` – Verarbeitet Elemente des Streams, solange die übergebene Bedingung erfüllt ist.

`dropWhile(...)`

`ofNullable(...)`

`iterate(... ,..., ...)`

Stream – Product Scenario



Stream – Filter



JDK8

```
public class StrictCustomer {  
  
    public static void main(String[] args) {  
  
        Stream<String> deliveredProductsQuality = Stream.of("1. Good",  
                                                             "2. Good",  
                                                             "3. Good",  
                                                             "4. Good",  
                                                             "5. Bad",  
                                                             "6. Good");  
  
        deliveredProductsQuality.  
            filter(productQuality -> productQuality.contains("Good")).  
            forEach(System.out::println);  
  
    }  
}
```

```
1. Good  
2. Good  
3. Good  
4. Good  
6. Good|
```

Stream – Filter



JDK8

```
public class StrictCustomer {  
  
    public static void main(String[] args) {  
  
        Stream<String> deliveredProductsQuality = Stream.of("1. Good",  
                                                             "2. Good",  
                                                             "3. Good",  
                                                             "4. Good",  
                                                             "5. Bad",  
                                                             "6. Good");  
  
        deliveredProductsQuality.  
            filter(productQuality -> productQuality.contains("Good")).  
            forEach(System.out::println);  
  
    }  
}
```

1. Good
2. Good
3. Good
4. Good





So ? What do we do ?



Google

Google-Suche

Auf gut Glück!


Google angeboten in: [English](#) [Français](#) [Italiano](#) [Rumantsch](#)



takeWhile(...)



```
public class StrictCustomer {  
    public static void main(String[] args) {  
        Stream<String> deliveredProductsQuality = Stream.of("1. Good",  
                                                             "2. Good",  
                                                             "3. Good",  
                                                             "4. Good",  
                                                             "5. Bad",  
                                                             "6. Good");  
  
        deliveredProductsQuality.  
            takeWhile(productQuality -> productQuality.contains("Good")).  
            forEach(System.out::println);  
    }  
}
```



```
1. Good  
2. Good  
3. Good  
4. Good
```



takeWhile(...)

dropWhile(...)

`dropWhile(Predicate<T>)` – Überspringt Elemente des Streams, solange die übergebene Bedingung erfüllt ist

ofNullable(...)

iterate(..., ..., ...)

dropWhile(...)



```
public class StrictCustomer {  
    public static void main(String[] args) {  
        Stream<String> deliveredProductsQuality = Stream.of("1. Bad",  
                                                             "2. Bad",  
                                                             "3. Bad",  
                                                             "4. Good",  
                                                             "5. Good",  
                                                             "6. Good");  
  
        deliveredProductsQuality.  
            dropWhile(productQuality -> productQuality.contains("Bad")).  
            forEach(System.out::println);  
    }  
}
```

dropWhile(...)



```
public class StrictCustomer {  
    public static void main(String[] args) {  
        Stream<String> deliveredProductsQuality = Stream.of("1. Bad",  
                                                             "2. Bad",  
                                                             "3. Bad",  
                                                             "4. Good",  
                                                             "5. Good",  
                                                             "6. Good");  
  
        deliveredProductsQuality.  
            dropWhile(productQuality -> productQuality.contains("Bad")).  
            forEach(System.out::println);  
    }  
}
```

```
4. Good  
5. Good  
6. Good
```

Kombination der beiden Methoden des Stream APIs



- Kombination der beiden Methoden zur Extraktion von Daten:

```
public static void main(String[] args)
{
    Stream<String> words = Stream.of("ab", "BLA", "Xy", "<START>",
                                    "WELCOME", "TO", "JAX", "Online",
                                    "<END>", "x", "y", "z");

    Stream<String> extracted = words.dropWhile(str -> !str.startsWith("<START>")).
                                    skip(1).
                                    takeWhile(str -> !str.startsWith("<END>"));

    extracted.forEach(System.out::println);
}
```

```
WELCOME
TO
JAX
Online
```



takeWhile(...)

dropWhile(...)

ofNullable(...)

iterate(...,)



- `iterate(T, Predicate<? super T>, UnaryOperator<T>)` – Erzeugt einen `Stream<T>` mit dem übergebenen Startwert. Die folgenden Werte werden durch den `UnaryOperator<T>` berechnet, solange das übergebene `Predicate<T>` erfüllt ist.

```
final IntStream stream = IntStream.iterate(1, n -> n < 10, n -> n + 1);
```

```
System.out.println(stream.mapToObj(num -> "" + num).collect(joining(", ")));
```

- `=> 1, 2, 3, 4, 5, 6, 7, 8, 9`
- Angaben ziemlich analog zur for-Schleife:

```
for(int n = 1; n < 10; n++)  
iterate(1, n -> n < 10, n -> n + 1);
```
- Da es ein Stream ist, aber mit einer Vielzahl weiterer Möglichkeiten

Stream API



Was macht?

```
IntStream.iterate(1, x -> x + 1).filter(n -> n < 10).forEach(System.out::println)
```

Vermutung: 1 2 3 4 5 6 7 8 9?

Und was?

```
IntStream.iterate(1, x -> x + 1).limit(9).forEach(System.out::println)
```

Vermutung: 1 2 3 4 5 6 7 8 9?

Und was?

```
IntStream.iterate(1, x -> x < 10, x -> x + 1).forEach(System.out::println)
```

Vermutung: 1 2 3 4 5 6 7 8 9?

Stream API



Was macht?

```
IntStream.iterate(1, x -> x + 1).filter(n -> n < 10).forEach(System.out::println)
```

Und was?

```
IntStream.iterate(1, x -> x + 1).limit(9).forEach(System.out::println)  
=> 1 2 3 .... 9
```

Und was?

```
IntStream.iterate(1, x -> x < 10, x -> x + 1).forEach(System.out::println)  
=> 1 2 3 .... 9
```



DEMO mit JShell



**Ermittle alle geraden und
durch 3 teilbaren Zahlen
inklusive 30?**



Mit `filter()` passt es wieder nicht ...

```
IntStream.iterate(0, x -> x + 2).  
    filter(n -> n <= 30 && n % 3 == 0).forEach(System.out::println)
```

Erst recht passt das mit dem `limit()` nicht mehr ... wie schreiben wir hier die **Bedingung**?

```
IntStream.iterate(0, x -> x + 2).limit(???.forEach(System.out::println)
```



JDK 9 `iterate()` als Abhilfe:

```
IntStream.iterate(0, n -> n <= 30, n -> n + 2).  
    filter(n -> n % 3 == 0).forEach(System.out::println)
```

=>

0

6

12

18

24

30

Jede Stream-Methode wird gemäss ihres Sinns verwendet ... Kaum macht man es richtig, schon funktioniert es 😊



Java 16



Stream => List ... es war so umständlich ...



```
List<String> namesMi = Stream.of("Tim", "Tom", "Mike", "Michael").  
    filter(str -> str.startsWith("Mi")).  
    collect(Collectors.toList());
```

FINALLY ... toList()



```
List<String> namesMi = Stream.of("Tim", "Tom", "Mike", "Michael").  
    filter(str -> str.startsWith("Mi")).  
    toList();
```



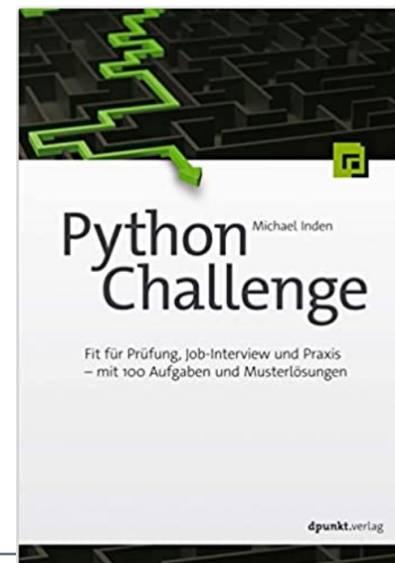
Exercises Part 7

https://github.com/Michaeli71/JAVA_INTRO





Questions?





Thank You
