

Lambdas

Programmiermethodik 2

Zum Nachlesen:

Michael Inden: Java 8 - Die Neuerungen: Lambdas, Streams, Date And Time API und JavaFX 8 im Überblick, dpunkt Verlag

Wiederholung

- Typen
- Typebounds
- Kompatibilität
- Generische Methoden

Ausblick



Agenda

- Lambdas
- Default-Methoden
- Iteration
- Collections-Erweiterungen



Lambdas

Imperative und Deklarative Programmierung

- Imperative Programmierung
 - Programmierung = Folge von Kommandos (Anweisungen, Methodenaufrufe)
 - Objektorientierte Programmierung (Objekte, Zustände)
 - prozedurale Programmierung
- Deklarative Programmierung
 - adressiert technisch versierte Anwender
 - z.B. SQL
 - nicht das Wie? sondern das Was? beschreiben

Funktionale Programmierung

- Programm = Verkettung von Funktionsaufrufen
- oft Analogie zur mathematischen Beschreibung
 - Beispiel: Fibonacci-Zahlen

Mathematische Beschreibung

$$fib(0) = 0$$

$$fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2)$$

rekursiv-funktionales Java-Programm

```
public static int berechneFibonacciZahl(int zahl) {  
    switch (zahl) {  
        case 0:  
            return 0;  
        case 1:  
            return 1;  
        default:  
            return berechneFibonacciZahl(zahl - 1)  
                + berechneFibonacciZahl(zahl - 2);  
    }  
}
```

- Unterscheidung: rein funktional vs. kann-auch-funktional
 - Java gehört (spätestens mit Java 8) in zweite Kategorie

Funktionale Programmierung in Java

- bereits bisher möglich, aber nicht immer elegant
- ab Java 8: neues Konstrukt: Lambdas (Funktionen)
- ähnlich wie Methoden, aber nicht an Klassen gebunden
- können ...
 - an beliebiger Stelle deklariert werden
 - an Variablen gebunden werden
 - als Argumente an Methoden übergeben werden
 - u.v.m

Überblick: Lambda

- **Parameterliste**

- **Liste von Anweisungen**

```
(int x, int y) -> { return x + y; }
```

- nur ein Parameter: "()" kann entfallen
- Typ aus Kontext klar: Typ kann weggelassen werden (Type Inference)

- Rückgabewert = Wert des letzten Ausdrucks.
- nur ein Ausdruck: "{}" kann entfallen
- nur ein Ausdruck: return kann weggelassen werden

Lambda-Ausdrücke

- Beispiel: Lambda-Ausdruck, der zwei int-Argumente bekommt und dessen Summe berechnet:

```
(int x, int y) -> { return x + y; }
```

- ein Lambda-Ausdruck ist Code, der
 - keinen Namen hat
 - keinen expliziten Rückgabetyp hat
 - keine Deklaration von Exceptions erlaubt/erfordert
- Weitere Beispiele:

```
(long x) -> { return x * 2; }
```

```
() -> { String name = "Lambda"; System.out.println("Hallo"  
    + name); }
```

Funktionale Interfaces und SAMs

- Lambda-Ausdrücke haben als Typen Funktionale Interfaces
 - engl. *functional interface*
 - neuer Typ in Java 8
 - Interface mit genau einer (abstrakten) Methode
 - wird auch SAM (*Single Abstract Method*) genannt
- werden durch Annotation `@FunctionalInterface` deklariert

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

- SAMs gab es auch vorher schon (hießen aber nicht so)
 - `Comparator<T>`, `Runnable` (siehe Threads), `EventHandler` (siehe Grafische Benutzerschnittstellen)

Überblick: Funktionales Interface

- neuer Typ für Lambda-Ausdrücke

@FunctionalInterface

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object obj);  
}
```

Schlüsselwort



- hat genau eine Methode (repräsentiert Signatur der kompatiblen Lambda-Ausdrücke)
- zusätzlich alle Methoden von `Object` erlaubt
- wird auch SAM (*Single Abstract Method*) genannt

Funktionale Interfaces

- Besonderheiten
 - alle Methoden aus Basisklasse `Object` sind zusätzlich erlaubt
 - Methoden müssen nicht explizit als `public abstract` deklariert werden
 - weil: sind sie automatisch in einem Interface

- Anwendungsbeispiel

```
Comparator<String> vergleichDerLaenge = (String str1,  
    String str2) -> {  
    return Integer.compare(str1.length(), str2.length());  
};  
System.out.println(vergleichDerLaenge.compare("Hallo", "Welt"));
```

Type Inference

- Was ist der Rückgabetypp eines Lambda-Ausdrucks?
- Wird automatisch vom Compiler bestimmt
 - dies nennt man *Type Inference*
- auch für die Parameter darf auf den Typ verzichtet werden
 - wird beim Aufruf bestimmt

```
Comparator<? super String> vergleichDerLaenge =  
    (str1, str2) -> {  
        return Integer.compare(str1.length(), str2.length());  
    };
```

Kurzformen der Syntax

- weitere Verkürzungsmöglichkeiten
 - falls auszuführender Code = nur ein Ausdruck → geschweifte Klammern können weggelassen werden
 - falls auszuführender Code = nur ein Ausdruck → `return` kann weggelassen werden
 - falls nur ein Parameter → runde Klammern können weggelassen werden
- Beispiel: Methode zum Verdoppeln einer Zahl
 - vollständige Version:

```
(long x) -> { return x * 2; }
```
 - verkürzte Version:

```
x -> x * 2
```
- weiterer Vorteil:
 - Einsatz des Lambda-Ausdrucks für alle Typen, die der Operator `*` unterstützt

Lambdas als Parameter

- Erinnerung Sortieren von Listen

```
List<String> namen = Arrays.asList("Andy", "Michael", "Max",  
    "Stefan");  
Collections.sort(namen, <Comparator-Objekt>);
```

- mit Lambda:

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

oder:

```
Collections.sort(namen, vergleichDerLaenge);
```

- Vorteil: keine eigene Klasse notwendig

this und Objektvariablen

- Lambdas dürfen auf das aktuelle Objekt (this) und auf Objektvariablen zugreifen
- Sichtbarkeit wie gehabt
- aber: nach Möglichkeit vermeiden
 - undurchsichtiger Code
 - Performance-Einbußen bei Parallelisierung
- Zugriff auf Objektvariablen
 - keine Einschränkung
- Zugriff auf lokale Variablen
 - Veränderung nicht erlaubt
 - früher (bis Java 7): Zugriff nur auf final-Variablen
 - ab Java 8: Compiler prüft, final nicht notwendig (effectively final)

this und Objektvariablen

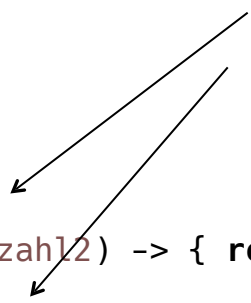
```
@FunctionalInterface
public interface VerrechnungZweiInts {
    public int verrechne(int zahl1, int zahl2);
}

public class ZugriffAufObjekt {
    private int offset = 23;
    private VerrechnungZweiInts addition = (zahl1, zahl2) -> { return zahl1 + zahl2; };
    private VerrechnungZweiInts additionMitOffset =
        (zahl1, zahl2) -> { offset += 2; return zahl1 + zahl2 + offset; };

    public void berechne() {
        System.out.println(addition.verrechne(23, 42));
        System.out.println(additionMitOffset.verrechne(23, 42));
        offset = 0;
        System.out.println(additionMitOffset.verrechne(23, 42));
    }

    public static void main(String[] args) {
        ZugriffAufObjekt zao = new ZugriffAufObjekt();
        zao.berechne();
    }
}
```

zwei
Lambda-
Ausdrücke



Ergebnis:
65
88
65

Übung: String-Verarbeitung

- Gesucht ist ein Lambda-Ausdruck, der von zwei Strings denjenigen zurückgibt, bei dem als erstes der Buchstabe 'A' (egal ob 'a' oder 'A') vorkommt.
 - Schreiben Sie den Lambda-Ausdruck.
 - Schreiben Sie ein passendes funktionales Interface.
- Beispiel
 - "Welt", "Hallo" → "Hallo"



Default-Methoden

Interface-Erweiterungen

- bisher: Interfaces bieten nur Schnittstelle (abstrakt, keine Implementierung)
- bei Umstellung auf Java 8
 - Wunsch, Interfaces zu erweitern
 - Konsequenz: alle implementierenden Klassen mussten angepasst werden
 - Idee: Implementierungen zulassen
 - neue Methode im Interface mit Implementierung
- Default-Methoden
 - Methoden in Interfaces mit Implementierung
 - Kennzeichnung durch Schlüsselwort default

Beispiel: Interface List

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

- damit ist jetzt möglich:

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

Beispiel: Interface Iterable

- Iterable steht über Collection und damit List

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

funktionales
Interface



mit Methode
accept()

- Was kann man damit machen?
 - Auf jedes Element eines Iterables eine Aktion ausführen

Funktionale Interfaces

- viele funktionale Interfaces (SAMs) in `java.util.function`, u.a.:

<code>Consumer<T></code>	Beschreibt eine Aktion auf einem Element vom Typ <code>T</code> . Dazu ist eine Methode <code>void accept(T)</code> definiert.
<code>Predicate<T></code>	Definiert eine Methode <code>boolean test(T)</code> . Diese berechnet für eine Eingabe vom Typ <code>T</code> einen booleschen Rückgabewert (z. B. <code>olderThan()</code>). Damit lassen sich sehr gut Filterbedingungen ausdrücken.
<code>Function<T,R></code>	Definiert eine Abbildungsfunktion in Form der Methode <code>R apply(T)</code> . Damit wird ein allgemeines Konzept von Transformationen beschrieben. Recht gebräuchlich ist beispielsweise die Extraktion eines Attributs aus einem komplexeren Typ.
<code>Supplier<T></code>	Stellt ein Ergebnis vom Typ <code>T</code> bereit. Im Gegensatz zu <code>Function<T,R></code> erhält ein <code>Supplier<T></code> keine Eingabe. Im Interface ist die Methode <code>T get()</code> deklariert. Damit lassen sich Objekterzeugungen auf verschiedene Weise nachbilden.

Beispiel: Ausgabe aller Collection-Elemente

```
List<String> namen = Arrays.asList("Andy", "Michael",  
    "Max", "Stefan");  
namen.forEach(name -> System.out.println("Name: " + name));
```

- Ausgabe:

Name: Andy

Name: Michael

Name: Max

Name: Stefan

Vorgabe von Standardverhalten

- Weiter Anwendung von Default-Methoden
 - Vorgabe von Standardverhalten
- Beispiel: eigene Datenstruktur mit Iterator
 - Spezialisierung von `Iterator<E>`

```
boolean hasNext();
```

```
E next();
```

```
void remove() // diese Methode wurde sehr häufig nicht verwendet:  
UnsupportedOperationException
```

- daher jetzt Default-Methode:

```
default void remove() {  
    throw new UnsupportedOperationException("remove");  
}
```

Konflikte?

- Gedankenspiel
 - Klasse implementiert zwei Interfaces
 - beide Interfaces haben Methoden mit identischer Signatur
- Auswahl der Implementierung (bis Java 7)
 - Klasse liefert (gemeinsame) Implementierung
 - VM wählt diese Implementierung → kein Problem
- mit Default-Methoden ab Java 8
 - beide Interfaces liefern Implementierung in Default-Methoden
 - Welche Implementierung soll die VM verwenden?

Beispiel: Konflikte

```
public interface TextVerarbeitung1 {  
    public default String werteAus(String text) {  
        return TextVerarbeitung1.class.getName() + ": " + text;  
    }  
}
```

```
public interface TextVerarbeitung2 {  
    public default String werteAus(String text) {  
        return TextVerarbeitung2.class.getName() + ": " + text;  
    }  
}
```

- Klasse, die beide Interfaces implementiert:

```
public class Umwandlung implements TextVerarbeitung1,  
    TextVerarbeitung2 { ...
```

Fehler!

```
@Override  
public String werteAus(String text) {  
    return Umwandlung.class.getName() + ": " + text;  
}
```

```
@Override  
public String werteAus(String text) {  
    return TextVerarbeitung1.super.werteAus(text);  
}
```

Vorteile von Default-Methoden

- ermöglichen API-Erweiterungen unter der Beibehaltung von Rückwärts-Kompatibilität
- erlauben es, ein gewünschtes Standardverhalten vorzugeben
- kann man überschreiben
 - Basisinterface mit Default-Verhalten
 - Spezialbehandlung bei Bedarf

Nachteile von Default-Methoden

- klare Trennung zwischen Schnittstelle und Implementierung verloren
- jetzt Mehrfachvererbung möglich
- man könnte, auf die Idee kommen, auch Variablen/Getter/Setter in Interfaces anzubieten
 - geht nicht, da Variablen in Interfaces final sein müssen
 - Alternative: abstrakte Basisklasse

Statische Methoden in Interfaces

- auch statische Methoden dürfen nun (Java 8) in Interfaces deklariert und implementiert! werden
- sinnvoll, wenn "Objekt"-unabhängige Hilfsmethoden umgesetzt werden sollen
 - bisher: Hilfsklassen mit nur statischen Methoden
- aber
 - weitere Verwässerung zwischen Schnittstelle und Implementierung
 - mit Bedacht und Vorsicht einsetzen

Methodenreferenzen

- Referenzen auf Methoden
- Syntax: Klasse::Methodenname
- Beispiele
 - Methoden: `System.out::println`, `Person::getName` , ...
 - Konstruktoren: `ArrayList::new`, `Person[]::new`, ...
- Anwendungsbeispiel:

```
List<String> namen = Arrays.asList("Max", "Andy", "Michael",  
    "Stefan");
```
- mit Lambda:

```
namen.forEach( name -> System.out.println(name) );
```
- mit Methodenreferenz

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


Methodenreferenzen mit Parametern

- Was passiert mit Parametern?
 - Compiler findet das passende und übergibt Argumente bei Aufruf

Referenz auf ...	Als Methodenreferenz	Als Lambda
Statische Methode	<code>String::valueOf</code>	<code>obj -> String.valueOf(obj)</code>
Instanzmethode eines Typs	<code>Object::toString</code> <code>String::compareTo</code>	<code>obj -> obj.toString()</code> <code>(str1, str2) -> str1.compareTo(str2)</code>
Instanzmethode eines Objekts	<code>person::getName</code>	<code>() -> person.getName()</code>
Konstruktor	<code>ArrayList::new</code>	<code>() -> new ArrayList<>()</code>

Übung: Default-Methode

- Gegeben ist das folgende Interface:

```
public interface StringVerarbeitung {  
    public default String  
        wendeFunktionAufStringAn(  
            Function<String, String> funktion,  
            String text) {  
        return funktion.apply(text);  
    }  
  
    public default String verarbeite(  
        String text) {  
        // TODO  
    }  
}
```

- Setzen Sie die Methode `verarbeite()` so als default um, dass Sie `wendeFunktionAn()` verwendet und den String unverändert zurückgibt

- Gegeben ist folgende Klasse, die das Interface implementiert

```
public class StringVerarbeitungLowerCase  
    implements StringVerarbeitung {  
    private String funktion(String text) {  
        return text.toLowerCase();  
    }  
}
```

- Überschreiben Sie die Methode `verarbeite()`, sodass die Methode `funktion()` als Methodenreferenz verwendet wird.



Iteration

Iteration

- Erinnerung: Iteration = Durchlaufen einer Collection
- Umsetzung
 - Schleifen (for, for-each, while, do-while)
 - Iterator: `java.util.Iterator<T>`

Externe Iteration

- Beispiele

```
List<String> namen = Arrays.asList("Jan", "Hein", "Klaas",  
    "Pit" );  
// Iterator  
final Iterator<String> it = namen.iterator();  
while (it.hasNext()){  
    System.out.println(it.next());  
}  
// for-Schleife  
for (int i = 0; i < namen.size(); i++){  
    System.out.println(namen.get(i));  
}  
// for-each-Schleife  
for (final String name : namen){  
    System.out.println(name);  
}
```

Interne Iteration

- neu mit Java 8: interne Iteration
- Collection selber übernimmt Iteration
- Von außen mitgegeben:
 - Was wird mit jedem Element gemacht?
- Beispiele:

```
// drei Varianten, gleiches Ergebnis  
namen.forEach((String name) ->  
    { System.out.println(name); });  
namen.forEach(name -> System.out.println(name) );  
namen.forEach(System.out::println);
```

Beispiel: Aufhellen von Grafiken

- Aufhellen aller ausgewählten Grafiken (externe Iteration):

```
public static void aufhellenExtern(List<GraphicsFigure>
    auswahl){
    for (GraphicsFigure grafik : auswahl){
        aufhellen(grafik);
    }
}
```

- mit interner Iteration:

```
public static void aufhellenIntern(List<GraphicsFigure>
    auswahl){
    auswahl.forEach( grafik -> { aufhellen(grafik); } );
}
```

Vorteile

- Möglichkeit zur Parallelisierung
- Möglichkeit zur Verallgemeinerung

```
public static void verarbeite(Collection<GraphicsFigure>  
    grafiken, Consumer<GraphicsFigure> verarbeitung){  
    grafiken.forEach(verarbeitung);  
}
```

- Aufruf:

```
Consumer<GraphicsFigure> operation = grafik ->  
    aufhellen(grafik);  
verarbeite(grafiken, operation);
```

- Verallgemeinerung:
 - Funktionalität als Parameter übergeben
 - Fachbegriff: Code As Data

Übung: Interne Iteration

- Schreiben Sie eine Variante der unten stehenden Codeblocks, die interne anstelle der verwendeten externen Iteration verwendet:

```
public static void verarbeiteExtern(List<Integer> zahlen) {  
    for (int zahl : zahlen) {  
        System.out.println("zahl: " + (zahl + 7) % 3);  
    }  
}
```



Collections-Erweiterung

Interface Predicate<T>

- Prädikate = Funktionen, die einen Wahrheitswert berechnen
- Verwendung:
 - Formulierung von Fallunterscheidungen
 - Umsetzung von Filtern (Auswahl von Elemente, die eine geforderte Eigenschaft haben)
- Funktionales Interface: `java.util.function.Predicate<T>`
- Methode: `boolean test(T)`
- Beispiele

```
Predicate<String> istNull = str -> str == null;
```

```
Predicate<String> istLeer = String::isEmpty;
```

```
Predicate<Person> istErwachsen =  
    person -> person.getAlter() >= 18;
```

Beispiel

- Filter für Namen, die mit "G" beginnen:

```
public static void filter(List<String> namen,  
    Predicate<String> filter) {  
    namen.forEach(name -> {  
        if (filter.test(name)) { System.out.println(name); }  
    });  
}
```

```
List<String> namen = new ArrayList<String>(Arrays.asList("Gandalf",  
    "Aragorn", „Frodo", "Gimli"));  
Predicate<String> beginntMitG = name ->  
    name.toUpperCase().startsWith("G");  
filter(namen, beginntMitG);
```

- liefert:

Gandalf

Gimli

Methode Collection.removeIf()

- Bedingtes Löschen von Elementen aus einer Collection
- Syntax: void removeIf(Predicate<T>)
- Beispiel:

```
namen.removeIf(beginntMitG);  
namen.forEach(System.out::println);
```

Interface: Unärer Operator

- Veränderung von Elementen
- funktionales Interface: `UnaryOperator<T>`
- Methode: `T apply(T)` (erbt von `Function<T,R>`)
- Beispiel:
 - Ersetzen eines null-Strings einen leeren Strings:

```
UnaryOperator<String> ausNullMachLeer =  
    str -> str == null ? "" : str;  
ausNullMachLeer.apply(null) → ""  
ausNullMachLeer.apply("Mike") → "Mike"
```

Methode List.replaceAll()

- Ersetzen aller Einträge einer Liste durch einen unären Operator
- Neue Methode im Interface List<T>:
 void replaceAll(UnaryOperator<T>);
- Beispiel: Ersetzen aller null-Strings in einer Liste durch leere Strings

```
UnaryOperator<String> ausNullMachLeer = str -> str == null ? "" : str;
List<String> namen = Arrays.asList("Jan", null, "", "Pit");
namen.forEach(name -> System.out.print("'" + name + "' "));
System.out.println();
namen.replaceAll(ausNullMachLeer);
namen.forEach(name -> System.out.print("'" + name + "' "));
System.out.println();
```

- Ausgabe:

'Jan' 'null' " 'Pit'

'Jan' " " 'Pit'

Übung: Bedingte Großbuchstaben

- Schreiben Sie eine Methode *bedingteGrossBuchstaben()*.
- die Methode hat einen Parameter: ein Prädikat für Strings
- die Methode erzeugt einen Unären Operator, der einen String verarbeiten kann
- liefert das Prädikat wahr, dann wird der String in Großbuchstaben umgewandelt
 - ansonsten bleibt der String bestehen
- Beispiel:

```
List<String> namen = Arrays.asList("Aragorn", "Gimli", "Gandalf", "Frodo");
namen.replaceAll(bedingteGrossBuchstaben(name -> name.length() <= 5));
namen.forEach( name -> System.out.print(name + " "));
```

- liefert:

Aragorn GIMLI Gandalf FRODO

Zusammenfassung

- Einstieg
- Default-Methoden
- Iteration
- Collections-Erweiterungen

Quellen

- Dieser Satz Folien basiert zu großen Teilen auf folgender Literatur: Michael Inden: Java 8 – Die Neuerungen, dpunkt Verlag, 2014