



# Lambdas

## Programmiermethodik 2

- Typen
- Typebounds
- Kompatibilität
- Generische Methoden



# Ausblick



- Ich möchte Verhalten/Funktionalität als Parameter übergeben.
- Ich möchte Standardverhalten (Implementierung) bereits in einem Interface vorgeben.

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



# Einstieg

- 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

- 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



- 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

- 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 Rückgabebetyp hat
  - keine Deklaration von Exceptions erlaubt/erfordert

- Weitere Beispiele:

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

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

- 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)
  - Beispiele:
    - `Comparator<T>`
    - `Runnable` (siehe Threads)
    - `EventHandler` (siehe Grafische Benutzerschnittstellen)

- 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

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

    boolean equals(Object obj);
}
```

- Anwendungsbeispiel

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

System.out.println(vergleichDerLaenge.compare("Hallo", "Welt"));
```

- Was ist der Rückgabebetyp 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<String> vergleichDerLaenge = (str1, str2) -> {  
    return Integer.compare(str1.length(), str2.length());  
};
```

- 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

- 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

- 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



```
/**
 * Ein Lambda-Interface zur Verrechnung zweier Int-Zahl mit einer weiteren
 * Int-Zahl als Ergebnis.
 *
 * @author Philipp Jenke
 */
@FunctionalInterface
public interface VerrechnungZweiInts {
    /**
     * Berechnung.
     *
     * @param zahl1
     *     Erste Eingabe.
     * @param zahl2
     *     Zweite Eingabe.
     * @return Ergebnis.
     */
    public int verrechne(int zahl1, int zahl2);
}
```

zwei  
Lambda-  
Ausdrücke

Ergebnis:

65

88

65

```
public class ZugriffAufObjekt {

    /**
     * Offset als Objektvariable.
     */
    private int offset = 23;

    /**
     * Lambda zur Addition zweier Zahlen.
     */
    private VerrechnungZweiInts addition = (zahl1, zahl2) -> {
        return zahl1 + zahl2;
    };

    /**
     * Lambda zur Addition zweier Zahlen mit Offset.
     */
    private VerrechnungZweiInts additionMitOffset = (zahl1, zahl2) -> {
        return zahl1 + zahl2 + offset;
    };

    /**
     * Test-Methode in der die Lambdas angewendet werden.
     */
    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));
    }

    /**
     * Programmeinstieg.
     *
     * @param args
     *     Kommandozeilenparameter
     */
    public static void main(String[] args) {
        ZugriffAufObjekt zao = new ZugriffAufObjekt();
        zao.berechne();
    }
}
```

- 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

- 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`

```
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()));
```

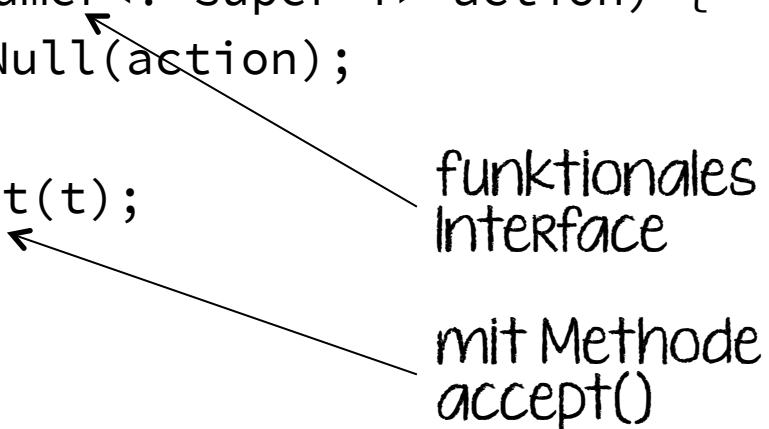
# Beispiel: Interface Iterable

- 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

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

<code>Consumer&lt;T&gt;</code>	Beschreibt eine Aktion auf einem Element vom Typ <code>T</code> . Dazu ist eine Methode <code>void accept(T)</code> definiert.
<code>Predicate&lt;T&gt;</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&lt;T,R&gt;</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&lt;T&gt;</code>	Stellt ein Ergebnis vom Typ <code>T</code> bereit. Im Gegensatz zu <code>Function&lt;T,R&gt;</code> erhält ein <code>Supplier&lt;T&gt;</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*



- 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");  
}
```

- 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?

```
public interface TextVerarbeitung1 {  
  
    /**  
     * Verarbeitet den Text  
     *  
     * @param text  
     *       Eingabetext.  
     * @return Ausgabertext.  
     */  
    public default String werteAus(String text) {  
        return TextVerarbeitung1.class.getName() + ": " + text;  
    }  
}
```

```
public interface TextVerarbeitung2 {  
  
    /**  
     * Verarbeitet den Text  
     *  
     * @param text  
     *       Eingabetext.  
     * @return Ausgabertext.  
     */  
    public default String werteAus(String text) {  
        return TextVerarbeitung1.class.getName() + ": " + text;  
    }  
}
```

- Klasse, die beide Interfaces implementiert

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

- Verwenden der Default-Implementierungen: Compiler-Fehler

- entweder *eigene Implementierung*

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

- oder *verwenden einer der beiden Default-Methoden*

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

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

- 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

- 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

- 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 );
```

- 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 -&gt; String.valueOf(obj)</code>
Instanzmethode eines Typs	<code>Object::toString</code>	<code>obj -&gt; obj.toString()</code>
	<code>String::compareTo</code>	<code>(str1, str2) -&gt; str1.compareTo(str2)</code>
Instanzmethode eines Objekts	<code>person::getName</code>	<code>() -&gt; person.getName()</code>
Konstruktor	<code>ArrayList::new</code>	<code>() -&gt; new ArrayList&lt;&gt;()</code>



- Gegeben ist das folgende Interface:

```
public interface StringVerarbeitung {  
    /**  
     * Verarbeitung eines Strings  
     *  
     * @param funktion  
     *       Verarbeitungsfunktion.  
     * @param text  
     *       Text, der verarbeitet werden soll.  
     * @return Bearbeiteter Text.  
     */  
    public default String wendeFunktionAufStringAn(  
        Function<String, String> funktion, String text) {  
        return funktion.apply(text);  
    }  
  
    /**  
     * Verarbeite eine String zu einem anderen String  
     *  
     * @param text  
     *       Zu verarbeitender Text.  
     * @return Bearbeiteter Text.  
     */  
    public String verarbeite(String text);  
}
```

- 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

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

## – Beispiele

```
List<String> namen = Arrays.asList("Jan", "Hein", "Klasse",  
    "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);  
}
```

- 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);
```

- 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); } );
}
```

- 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*

- 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

- 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;
```

- 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 = Arrays.asList("Gandalf", "Aragorn", "Frodo", "Gimli");  
filter(namen, name -> name.toUpperCase().startsWith("G"));
```

- liefert:

*Gandalf*

*Gimli*

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

```
List<String> namen =  
    new ArrayList<String>(Arrays.asList("Gandalf", "Aragorn", "Frodo",  
        "Gimli"));  
Predicate<String> beginntMitG = name -> name.toUpperCase().startsWith("G");  
  
namen.removeIf(beginntMitG);  
namen.forEach(System.out::println);
```

- 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'*

- 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*

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



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