

# Programmieren II

## Java 8 – Was ist neu?

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

# Übersicht Sprach-Erweiterungen (1)

## ■ Java 1.1 (1997)

- Innere Klassen
- Reflection
- Beans

## ■ Java 5 (2004)

- Generics
- Enums
- Autoboxing
- Foreach-Schleife
- Varargs
- Annotationen\*
- Statische Imports

Java-Version	Jahr
1.0	1996
1.1	1997
1.2	1998
	1999
1.3	2000
	2001
1.4	2002
	2003
5	2004
	2005
6	2006
	2007
	2008
	2009
	2010
7	2011
	2012
	2013
8	2014
	2015
	2016
9	2017
10+11	2018
12+13	2019
14+15	2020
16+17	2021
18+19	2022
20(+21)	2023

\* nicht im Rahmen der Vorlesung behandelt

# Übersicht Sprach-Erweiterungen (2)

- Java 7 (2011)
  - switch mit String-Literalen
  - Diamond-Operator (<>)
  - try-with-resources
  
- Java 8 (2014)
  - „effectively final“
  - Statische + Default-Methoden in Interfaces
  - Annotation-Erweiterungen
  - **Lambda-Expressions**

Java-Version	Jahr
1.0	1996
1.1	1997
1.2	1998
	1999
1.3	2000
	2001
1.4	2002
	2003
5	2004
	2005
6	2006
	2007
	2008
	2009
	2010
7	2011
	2012
	2013
8	2014
	2015
	2016
9	2017
10+11	2018
12+13	2019
14+15	2020
16+17	2021
18+19	2022
20(+21)	2023

# Übersicht Sprach-Erweiterungen (3)

- Java 10 (2018)
  - Local-Variable Type Inference
- Java 12 (2019)
  - Switch Expressions

Java-Version	Jahr
1.0	1996
1.1	1997
1.2	1998
	1999
1.3	2000
	2001
1.4	2002
	2003
5	2004
	2005
6	2006
	2007
	2008
	2009
	2010
7	2011
	2012
	2013
8	2014
	2015
	2016
9	2017
10+11	2018
12+13	2019
14+15	2020
16+17	2021
18+19	2022
20(+21)	2023

# Was ist noch neu in Java 8?

- Neben Erweiterung der Sprachfeatures bei Annotations und **Lambda-Expressions** auch Ergänzungen bei der API
  - JavaFX nun Teil der Standardbibliothek\*
  - Neue Verschlüsselungsalgorithmen (`java.net.URL` → `https!`)
  - Neue Date/Time-API (Package `java.time`)
  - Nashorn-JavaScript-Engine (ersetzt Rhino)
  - **Functional Interfaces (SAM-Types)**
  - **Default-Methoden bei Interfaces des Collection-Frameworks**
  - **Streaming-API**
  
- Dazu nachher noch ein kleiner „Ausflug“:
  - **NIO.2 / Java 7**

*\* wurde mit Java 12 wieder rückgängig gemacht*

# Effectively Final (1)

- Variablen gelten als „effectively final“, wenn ihnen nur einmal ein Wert zugewiesen wird
  - Variable überall verwendbar, wo normal „final“-Keyword nötig wäre:

```
public void buildMyUi() {  
  
    JButton b = new JButton(); // kein final Keyword!  
  
    b.addActionListener( new ActionListener() {  
        public void actionPerformed((ActionEvent e) {  
            // in anonymer innerer Klasse verwendet  
            System.out.println( b.getText() );  
        }  
    } );  
}
```




*Vor Java 8: Compiler-Error  
Ab Java 8: funktioniert!*

## Effectively Final (2)

- Sobald zweite Zuweisung ergänzt wird → Fehler!

```
public void buildMyUi() {
    JButton b = new JButton(); // erste Zuweisung
    b.addActionListener( new ActionListener() {
        public void actionPerformed((ActionEvent e) {
            // in anonymer innerer Klasse verwendet
            System.out.println( b.getText() );
        }
    } );
    b = new JButton(); // zweite Zuweisung
}
```



- Wenn man `final`-Keyword an der Variable ergänzen könnte, ohne dass es Compiler-Fehler gibt  
→ dann „effectively final“

# Functional Interfaces

- Interfaces mit *genau einer* abstrakten Methode
  - Auch SAM-Type (*single abstract method type*) genannt
  - Beispiele, die es bereits vor Java 8 gab
    - `Runnable (void run())`
    - `Comparator (int compare(T,T))`
    - `ActionListener (void actionPerformed(ActionEvent))`
  - Prominente Erweiterungen mit Java 8 (`java.util.function`)
    - `Consumer (void accept(T))`
    - `BiConsumer (void accept(T,U))`
    - `Function (R apply(T))`
    - `BiFunction (R apply(T,U))`
    - `Supplier (T get())`
    - `Predicate (boolean test(T))`
- Oft verwand als Lambda-Expression



# Lambda-Expressions (1)

- „Verkürzte Schreibweise“ für die Erzeugung von Instanzen eines SAM-Type

```
 JButton b = new JButton();  
 b.addActionListener( new ActionListener() {  
     public void actionPerformed((ActionEvent e) {  
         System.out.println( b.getText() );  
     }  
 } );
```

wird zu

```
 JButton b = new JButton();  
 b.addActionListener( e -> System.out.println( b.getText() ) );
```

## Lambda-Expressions (2)

```
 JButton b = new JButton();  
 b.addActionListener( new ActionListener() {  
     public void actionPerformed( ActionEvent e ) {  
         System.out.println( b.getText() );  
     }  
 } );
```

```
 JButton b = new JButton();  
 b.addActionListener( e -> System.out.println( b.getText() ) );
```

- Woher weiß der Compiler was gemeint ist?
  - addActionListener(...) ist nicht überladen, daher ist der Typ des Arguments eindeutig  
→ muss ActionListener sein!
  - ActionListener ist SAM-Type  
→ „Zu implementierende Methode“ muss actionPerformed sein  
→ Typ des Arguments ergibt sich dadurch ebenfalls eindeutig

## Lambda-Expressions (3)

- Es gibt alternative Schreibweisen:

```
JButton b = new JButton();

// Bekannte Schreibweise von den Folien zuvor
b.addActionListener( e -> System.out.println( b.getText() ) );

// Mit Klammern
b.addActionListener( (e) -> System.out.println( b.getText() ) );

// Mit Klammern + Argument-Typ
b.addActionListener(
    (ActionEvent e) -> System.out.println( b.getText() ) );
```

➔ alle Bedeutungsgleich!

# Lambda-Expressions – mehrere Argumente (1)

## ■ Was ist bei mehreren Argumenten?

```
List<String> names = new ArrayList<>();  
// Namen hinzufügen ...  
  
// Sortieren nach Länge des namen  
Collections.sort( names, new Comparator<String>() {  
    @Override  
    public int compare( String a, String b ) {  
        return Integer.compare( a.length(), b.length() );  
    }  
} );
```

wird zu

```
List<String> names = new ArrayList<>();  
// Namen hinzufügen ...  
  
Collections.sort( names,  
    (a, b) -> Integer.compare( a.length(), b.length() ) );
```

## Lambda-Expressions – mehrere Argumente (2)

- Bei mehreren Argumenten sind die Klammern nicht mehr optional:

```
List<String> names = new ArrayList<>>();  
// Namen hinzufügen ...
```

```
Collections.sort( names,  
    (a, b) -> Integer.compare( a.length(), b.length() ) );
```



```
List<String> names = new ArrayList<>();  
// Namen hinzufügen ...
```

```
Collections.sort( names,  
    a, b -> Integer.compare( a.length(), b.length() ) );
```



# Lambda-Expressions – weiteres Beispiel

## ■ Auslagern des Vergleichs:

```
private int compareByLength( String a, String b ) {  
    return Integer.compare( a.length(), b.length() );  
}
```

```
private void foobar() {  
    List<String> names = new ArrayList<>();  
    // Namen hinzufügen ...  
  
    Collections.sort( names, (a, b) -> this.compareByLength(a, b) );  
}
```

*Argumente der SAM-Type-  
Methode und der aufgerufenen  
Methode identisch!*

→ weitere Schreibweise möglich

```
Collections.sort( names, this::compareByLength );
```

*eine Art „Verweis“ auf die zu nutzende Methode*

# Lambda-Expressions – Code-Blöcke

- Bisher immer nur Aufruf einer anderen Methode

```
JButton b = new JButton();  
b.addActionListener( e -> System.out.println( b.getText() ) );
```

- Mehrere Anweisungen in Form eines Code-Blocks sind möglich:

```
JButton b = new JButton();  
b.addActionListener( e -> {  
    System.out.println( b.getText() );  
    b.setText("Foo");  
} );
```

- Variante ohne geschweifte Klammern ist Vereinfachung des allgemeinen Falls mit komplettem Code-Block

# Lambda-Expressions – Gültigkeitsbereich

- Wichtig: Lambda-Expressions definieren anders als anonyme Instanzen keinen eigenen Gültigkeitsbereich!

```
 JButton b = new JButton();  
 b.addActionListener(new ActionListener() {  
     @Override  
     public void actionPerformed(ActionEvent e) {  
         int b = 12;  
     }  
 });
```



```
 JButton b = new JButton();  
 b.addActionListener( e -> {  
     int b = 12;  
 });
```



**Compiler-Fehler**

*“Lambda expression's local variable b cannot redeclare another local variable defined in an enclosing scope.”*



# Lambda-Expressions – Mehrdeutigkeiten

## ■ Überladene Funktionen ermöglichen Mehrdeutigkeiten

```
public void myFunc( ActionListener al ) {  
    /* ... */  
}  
public void myFunc( Consumer<ActionEvent> con ) {  
    /* ... */  
}
```

- Die zu implementierende Methode von ActionListener und Consumer<ActionEvent> hat „Rückgabotyp“ void und erwartet genau ein Argument.

```
this.myFunc( e -> {} ); // ActionListener oder Consumer!?
```



## ■ Lösung: Type-Cast zur Kennzeichnung!

```
this.myFunc( (ActionListener) e -> {} );
```



# Erweiterungen im Collection-Framework (1)

- Einige Schnittstellen im Collection-Framework wurden um Default-Methoden (insb. zur Verwendung mit Lambda-Expressions) erweitert
- Prominentes Beispiel: Iterieren über Werte
  - Iterable: default void forEach(Consumer<T> action)
  - Map: default void forEach(BiConsumer<K, V> action)

```
List<String> names = new ArrayList<>();  
// Namen hinzufügen ...  
  
// Gibt nacheinander alle Namen aus  
names.forEach( System.out::println );
```

# Erweiterungen im Collection-Framework (2)

## ■ Was steckt dahinter?

- `Iterable.forEach` ist eine Default-Methode, die eine Schleife implementiert und den Consumer für jedes Element aufruft:

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

*Quelle: `java.lang.Iterable`*

- Im Prinzip steht da also

```
List<String> names = new ArrayList<>();  
// Namen hinzufügen ...  
  
// Gibt nacheinander alle Namen aus  
for ( String n : names ) {  
    System.out.println( n );  
}
```

# Streaming-API

- Auch: „Bulk Data Operations for Collections”
- Einführung sog. “Streams” (`java.util.stream.Stream`)
- Welche Operationen gibt es?
  - Stream-Verarbeitung “starten”
    - Sequentiell
    - Parallel
  - Zwischenoperationen (“map/reduce/filter”)
  - Terminalfunktionen
    - Beenden die Stream-Verarbeitung

*Hinweis: es werden nicht alle Operationen vorgestellt*

# Für die Beispiele: Klasse Person

- Für einige der Beispiele wird folgende (Hilfs-)Klasse als bekannt vorausgesetzt:

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person( String name ) {  
        this.name = name;  
    }  
    public Person( String name, int age ) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return this.name; }  
    public void setName( String name ) { this.name = name; }  
    public int getAge() { return this.age; }  
    public void setAge( int age ) { this.age = age; }  
    public String toString() {  
        return "Person[" + this.name + "," + this.age + "]"; }  
}
```

# Streamverarbeitung starten (1)

- Erzeugen eines Streams von Zeichenketten

```
List<String> names = new ArrayList<>();  
// Namen hinzufügen ...  
  
names.stream()... // sequentielle Abarbeitung der Elemente  
  
names.parallelStream()... // parallele Abarbeitung der Elemente
```

- `stream()` ist Erweiterung am Interface `Collection`  
→ für alle `Collection`-Typen verfügbar
- Macht natürlich erst „Spaß“ mit weiteren Operationen

## Streamverarbeitung starten (2)

- Erzeugen eines Streams aus einzelnen/mehreren Objekten oder Arrays:

```
// Erzeugt einen Stream mit 1 Zeichenkette  
Stream.of( "Emma" );  
  
// Erzeugt einen Stream mit 2 Zeichenketten  
Stream.of( "Emma", "Hannah" );  
  
// Erzeugt einen Stream mit 2 Zeichenketten aus Array  
String[] nameArr = { "Emma", "Hannah" };  
Stream.of( nameArr );
```

*Dies funktioniert intern über die Klasse `StreamSupport` – die Details hierzu würden jedoch zu weit führen*

## Streamverarbeitung starten (3)

- Die parallele Streamverarbeitung kann auch jederzeit aktiviert werden:

```
// Erzeugt einen Stream mit 2 Zeichenketten  
Stream.of( "Emma", "Hannah" )  
.parallel()    // ab hier paralleler Stream  
.forEach( this::someCalculation );
```

- Es geht auch explizit sequentiell:

```
// Erzeugt einen Stream mit 2 Zeichenketten  
Stream.of( "Emma", "Hannah" )  
.sequential()  // ab hier sequentieller Stream  
.forEach( this::someCalculation );
```



# Vorschau: Terminalfunktion `forEach`

- Führt Operation auf Elementen des Streams aus und beendet die Stream-Verarbeitung

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia", "Lina",
          "Anna", "Marie", "Mila", "Lea").forEach( names::add );
```

```
// Alle Stream-Elemente ausgeben
names.stream().forEach( System.out::println );
```

*Achtung: `forEach` hier von `Stream`, nicht von `Collection`!*

- Weitere Terminalfunktionen später
  - Warum? → `forEach` praktisch für Beispiele

# Zwischenfunktionen - filter

- **filter**: Elemente anhand eines bestimmten Kriteriums (Predicate) aussortieren  
→ Alle Namen mit weniger als 4 Zeichen bestimmen:

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia", "Lina", "Anna", "Marie", "Mila", "Lea").forEach(names::add);

names.stream()
    .filter( n -> (n.length() < 4) ) // Hier sind für weitere
    .forEach( System.out::println ); // Operationen nur noch
                                     // Mia und Lea im Stream
```

- `n` ist vom Typ `String` (da `List<String>`)
- Rückgabe des Lambda-Ausdrucks ist vom Typ `boolean`:
  - `true` → Element bleibt im Stream
  - `false` → Element wird aus Stream entfernt

Ausgabe
<b>Mia</b>
<b>Lea</b>

# Zwischenfunktionen – map (1)

- map: Elemente mithilfe einer Umwandlungsfunktion (Function) in andere Objekte umwandeln  
→ Alle Namen in Person-Objekte umwandeln

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia", "Lina", "Anna", "Marie", "Mila", "Lea").forEach(names::add);

names.stream()
    .map( n -> new Person( n ) ) // Hier sind ab nächstem
    .forEach( System.out::println ); // Schritt Personen-Objekte im
                                    // Stream, keine Strings mehr!
```

- n ist vom Typ String (da List<String>)
- Rückgabe des Lambda-Ausdrucks ist vom Typ Person

*toString-Methode von Person!*

```
Ausgabe
Person[Emma,0]
Person[Hannah,0]
Person[Mia,0]
Person[Sophia,0]
Person[Emilia,0]
Person[Lina,0]
Person[Anna,0]
Person[Marie,0]
Person[Mila,0]
Person[Lea,0]
```

## Zwischenfunktionen – map (2)

- Das geht aber auch noch kürzer:

```
names.stream().map( n -> new Person( n ) ); // Bekannte Variante
```

```
names.stream().map( Person::new ); // Alternative Schreibweise
```

- Compiler weiß anhand der Argumente (1 String-Argument) welcher der beiden Person-Konstruktoren zu verwenden ist
- map ist der allgemeine Fall für Objekttypen, für primitive Datentypen gibt es spezielle Mapping-Methoden

```
Stream.of( "1","2","3" ).                // Stream mit 3 Zeichenketten.  
    mapToInt( Integer::parseInt ); // Stream aus primitiven int  
                                     // durch Umwandlung mittels  
                                     // Integer.parseInt.
```

- Ergebnis ist ein IntStream, kein normaler Stream (später mehr)

# Zwischenfunktionen – skip

## ■ Überpringen von n Elementen im Stream

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia",
          "Lina", "Anna", "Marie", "Mila", "Lea").forEach( names::add );

names.stream().skip( 4 )           // Emma, Hannah, Mia und
    .forEach( System.out::println ); // Sophia werden übersprungen,
                                     // folgende Operationen
                                     // beginnen beim Emilia
```

 Ausgabe

```
Emilia
Lina
Anna
Marie
Mila
Lea
```

## Zwischenfunktionen – distinct

- Aussortieren von Dubletten
- für Vergleich wird die equals-Methode verwendet

```
// Variante 1: ohne distinct
Stream.of("1", "2", "1", "2", "1", "2")
    .forEach( System.out::println );

// Variante 2: mit distinct
Stream.of("1", "2", "1", "2", "1", "2")
    .distinct()
    .forEach( System.out::println );
```

Ausgabe (1)

1  
2  
1  
2  
1  
2

Ausgabe (2)

1  
2

# Zwischenfunktionen – peek

- Operation auf allen Elementen des Streams durchführen
  - Unterschied zu forEach: Stream-Verarbeitung geht weiter

```
Stream.of("1", "2", "1", "2", "1", "2")  
    .peek( n -> System.out.println( "Peek: " + n ) )  
    .distinct()  
    ...
```

Nach *peek* geht die Stream-Verarbeitung weiter!




```
> Ausgabe  
Peek: 1  
Peek: 2  
Peek: 1  
Peek: 2  
Peek: 1  
Peek: 2
```

## Zwischenfunktionen – flatMap

- Elemente eines Streams in mehrere Objekte für die weitere Verarbeitung mithilfe einer Funktion auftrennen („Flachklopfen“ einer Datenstruktur)  
→ „Ergebnis“ der Funktion muss ein Stream sein!

```
String[][] strings = new String[][]{  
    {"Lorem", "ipsum"}, {"dolor", "sit"}, {"amet", "consectetur"} };  
  
Stream.of( strings )                // Stream aus String[]-Elementen  
  
    .flatMap( Stream::of )           // aus String[]-Eintrag  
                                     // wird neuer String-Stream  
  
    .forEach( System.out::println ); // Stream ab hier  
                                     // besteht dank flatMap  
                                     // aus String-Elementen
```

 **Ausgabe**  
Lorem  
ipsum  
dolor  
sit  
amet  
consectetur



# Zwischenfunktionen – sorted

## ■ Elemente eines Streams sortieren

```
Stream.of("Lorem", "ipsum", "dolor", "sit", "amet", "consectetur")  
    .sorted()  
    .forEach( System.out::println );
```

Ausgabe

```
Lorem  
amet  
consectetur  
dolor  
ipsum  
sit
```

- Elemente müssen vergleichbar sein (Interface Comparable implementieren)

## ■ Alternativ: eigenen Comparator für Sortieren angeben

```
Stream.of("Lorem", "ipsum", "dolor", "sit", "amet", "consectetur")  
    .sorted( (a, b) -> a.compareToIgnoreCase( b ) )  
    .forEach( System.out::println );
```

Ausgabe

```
amet  
consectetur  
dolor  
ipsum  
Lorem  
sit
```

- Vorher: (implizit) String.compareTo (Case-sensitive)
- Nun: String.compareToIgnoreCase (Case-insensitive)

# Terminalfunktionen – Erinnerung: forEach

- Beendet die Stream-Verarbeitung („Rückgabetyp“ void)

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia", "Lina",
          "Anna", "Marie", "Mila", "Lea").forEach( names::add );
```

```
// Alle Stream-Elemente ausgeben
names.stream().forEach( System.out::println );
```

*Achtung: forEach hier von Stream, nicht von Collection!*

# Terminalfunktionen – collect (1)

- „Einsammeln“ der Elemente im Stream, bspw. in eine Liste

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia",
          "Lina", "Anna", "Marie", "Mila", "Lea").forEach( names::add );

List<String> resNames = names.stream()
    .skip( 4 )           // 4 Elemente überspringen
    .collect( Collectors.toList() ); // Elemente als Liste einsammeln

System.out.println( resNames );
```



Ausgabe

[Emilia, Lina, Anna, Marie, Mila, Lea]

Ausgabe von toString für List

## Terminalfunktionen – collect (2)

- Es gibt verschiedene, mitgelieferte Kollektoren
  - Werden von `java.util.stream.Collectors` bereitgestellt
    - Als Liste: `Collectors.toList`
    - Als Set: `Collectors.toSet`
    - Zusammenfügen als String: `Collectors.joining( CharSequence )`

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia",
          "Lina", "Anna", "Marie", "Mila", "Lea").forEach( names::add );

String namesCsv = names.stream()
    .skip( 4 )
    .collect( Collectors.joining( ", " ) );

// 4 Elemente überspringen
// Elemente als Gesamtstring
// einsammeln und jedes Element
// durch ", " vom andern trennen

System.out.println( namesCsv );
```

 Ausgabe

Emilia, Lina, Anna, Marie, Mila, Lea


# Terminalfunktionen – Reduzierer (1)

## ■ count: Zählen der Anzahl der Elemente im Stream

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia",
          "Lina", "Anna", "Marie", "Mila", "Lea").forEach( names::add );

long count = names.stream()
    .skip( 4 ) // 4 Elemente überspringen
    .count(); // verbleibende Elemente zählen

System.out.println( "count = " + count );
```

 **Ausgabe**  
count = 6


## Terminalfunktionen – Reduzierer (2)

- `findFirst`: Das erste Element finden
- `findAny`: „Irgendein“ Element finden

```
// Befüllen einer Liste mit Namen - mittels Stream
List<String> names = new ArrayList<>();
Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia",
          "Lina", "Anna", "Marie", "Mila", "Lea").forEach( names::add );

Optional<String> first = names.stream()
    .skip( 4 )           // 4 Elemente überspringen
    .findFirst();       // erstes Element finden

if ( first.isPresent() ) { // isPresent von Optional
    System.out.println( "first = " + first.get() );
} else {
    System.out.println( "no element found" );
}
```

 **Ausgabe**  
first = Emilia

# Terminalfunktionen – IntStream

- Bei typisierten Streams sind weitere, typenspezifische Terminalfunktionen verfügbar

- Bspw. bei IntStream arithmetische Operationen:

```
// Größte Zahl im IntStream
OptionalInt max = IntStream.of( 1, 2, 3, 4, 5 ).max();

// Kleinste Zahl im IntStream
OptionalInt min = IntStream.of( 1, 2, 3, 4, 5 ).min();

// Durchschnitt der Zahlen im IntStream
OptionalDouble avg = IntStream.of( 1, 2, 3, 4, 5 ).average();

// Summe der Zahlen im IntStream
int sum = IntStream.of( 1, 2, 3, 4, 5 ).sum();

System.out.printf( "%s, %s, %s, %s\n", max, min, avg, sum );
```



Ausgabe

```
OptionalInt[5], OptionalInt[1], OptionalDouble[3.0], 15
```

# Beispiel 1: Parallele Verarbeitung

```


public class Parallel {
    public static void main( String[] args ) {
        // Befüllen einer Liste mit Namen - mittels Stream
        List<String> names = new ArrayList<>();
        Stream.of("Emma", "Hannah", "Mia", "Sophia", "Emilia",
                  "Lina", "Anna", "Marie", "Mila", "Lea").forEach( names::add );

        long start = System.currentTimeMillis();
        names.parallelStream().forEach( Parallel::timeConsumingCalculation );
        System.out.println( "Parallel took " +
                           ( System.currentTimeMillis() - start ) + "ms" );

        start = System.currentTimeMillis();
        names.stream().forEach( Parallel::timeConsumingCalculation );
        System.out.println( "Sequential took "
                           + ( System.currentTimeMillis() - start ) + "ms" );
    }

    public static void timeConsumingCalculation( String name ) {
        try { Thread.sleep( 1000L ); } catch ( Exception e ) { }
    }
}

```

 Ausgabe (8-Kern CPU)

Parallel took 2010ms  
Sequential took 10001ms



## Beispiel 2: Mit Stream vs. ohne Stream

- Alle Zeichenketten mit ungerade Länge ausfiltern

// Mit Streams

```
String[] data =  
    { "Abc", "Defg", "Hi", "Jkl" };  
  
List<String> filtered = Stream.of( data )  
    .filter( e -> (e.length()%2==0) )  
    .collect( Collectors.toList() );
```

// Klassisch, ohne Streams

```
String[] data =  
    { "Abc", "Defg", "Hi", "Jkl" };  
  
List<String> filtered = new ArrayList<>();  
for ( String s : data ) {  
    if ( s.length()%2 == 0 ) {  
        filtered.add( s );  
    }  
}
```

→ Code wird kürzer & übersichtlicher

→ Es ist vom Wording im Code meist klarer was inhaltlich passiert

# NIO.2

- Eingeführt mit JDK7
  - Neue API für I/O-Operationen
  - Keine Spracherweiterung → Ergänzung der Klassenbibliothek
  
- Aspekte die wir betrachten
  - File vs. Path
  - (vereinfachter) Zugriff auf Dateien
  - Nutzung der Java 8-Features mit NIO.2

## NIO.2 – File vs. Path (1)

- Pfad ist (wie File) allgemeine Abbildung einer abstrakten „Adresse“ einer Datei
- Diese „Adressen“ können absolut oder relativ sein

	absolut	relativ
Windows	<code>C:\temp\abc\def.zip</code>	<code>temp\abc\def.zip</code>
Unixide Systeme	<code>/tmp/abc/def.zip</code>	<code>abc/def.zip</code>

## NIO.2 – File vs. Path (2)

- `java.util.File` hat diverse Nachteile
  - Fehlende Exceptions bei Fehlern die genauer spezifizieren was schiefgegangen ist
  - `File.rename` funktionierte nicht auf allen Plattformen gleich
  - Symbolische Links kaum unterstützt
  - Attribute wie bspw. Zugriffs- & Eigentumsrechte nicht abgebildet
  - Probleme mit großer Anzahl von Dateien in einem Verzeichnis
  - Probleme mit zirkulären symbolischen Links



- NIO.2 korrigiert diese Defizite

Quelle: <https://docs.oracle.com/javase/tutorial/essential/io/legacy.html>

## NIO.2 - Path

- Instanzen von `java.nio.file.Path` werden nicht mit dem `new`-Operator erzeugt
  - Hierfür ist die Verwendung der einer Factory-Methode vorgesehen:  
`java.nio.file.Paths.get(..)`

```
// alt - mit File
File f = new File( "/tmp/abc/def.txt" );

// neu - mit Path
Path p = Paths.get( "/tmp/abc/def.txt" );
```

## NIO.2 – Files

- Zur Interaktion mit dem Dateisystem bietet die Klasse `java.nio.file.Files` statische Methoden an
- Die bekannte Funktionalität kann meist 1:1 wieder gefunden werden, Auswahl:
  - `File.createNewFile` → `Files.createFile`
  - `File.mkdir` → `Files.createDirectory`
  - `File.renameTo` → `Files.move`
  - `File.canRead` → `Files.isReadable`
- Ausführliche Übersicht Alt/Neu siehe <https://docs.oracle.com/javase/tutorial/essential/io/legacy.html#mapping>
- Es kam jedoch auch neue Funktionalität hinzu

## NIO.2 – Files.newBufferedReader

- Erzeugen eines BufferedReader-Objekts für eine Datei

```
// alt - mit File
File f = new File( "/tmp/abc/def.txt" );
try( BufferedReader br = new BufferedReader( new FileReader( f ) ) ){
    // ...
} catch (IOException e) { }
```

```
//neu - mit Path
Path p = Paths.get( "/tmp/abc/def.txt" );
try( BufferedReader br = Files.newBufferedReader( p ) ){
    // ...
} catch (IOException e) { }
```

```
// Encoding Angabe möglich - vorher nur über "Umweg" InputStream
try( BufferedReader br = Files.newBufferedReader( p,
                                                    StandardCharsets.UTF_8 ) ){
    // ...
} catch (IOException e) { }
```

## NIO.2 – Files.newBufferedWriter

- Erzeugen eines BufferedWriter-Objekts für eine Datei

```
// alt - mit File
File f = new File( "/tmp/abc/def.txt" );
try( BufferedWriter bw = new BufferedWriter( new FileWriter( f ) ) ){
    // ...
} catch (IOException e) { }
```

```
//neu - mit Path
Path p = Paths.get( "/tmp/abc/def.txt" );
try( BufferedWriter bw = Files.newBufferedWriter( p ) ){
    // ...
} catch (IOException e) { }
```

```
// Encoding Angabe möglich - vorher nur über "Umweg" OutputStream
try( BufferedWriter bw = Files.newBufferedWriter( p,
                                                    StandardCharsets.UTF_8 ) ){
    // ...
} catch (IOException e) { }
```



## NIO.2 – Files.readAllBytes

- Einlesen einer kompletten Binärdatei

```
Path p = Paths.get( "/tmp/abc/ghi.zip" );  
try {  
    byte[] data = Files.readAllBytes( p );  
} catch (IOException e) {  
}
```

- Kein *try-with-resources* → Komplette Ressourcen-Behandlung innerhalb der aufgerufenen Methode
- Inhalt der Datei steht im Anschluss als byte-Array zur Verfügung

## NIO.2 – Files.readAllLines

- Einlesen aller Zeilen einer Textdatei in 1 Befehl

```
Path p = Paths.get( "/tmp/abc/def.txt" );  
try {  
    List<String> lines1 = Files.readAllLines( p );  
    List<String> lines2 = Files.readAllLines( p,  
                                              StandardCharsets.UTF_8 );  
} catch (IOException e) {  
}
```

- Kein *try-with-resources* → Komplette Ressourcen-Behandlung innerhalb der aufgerufenen Methode
- Inhalt der Datei steht zeilenweise in der String-Liste
  - Direkte Weiterverarbeitung mit Streams möglich!

# NIO.2 und Streams (1)

- Auszug aus der Musterlösung zu Bücherei
  - Noch keine Nutzung von NIO.2 + Streams

```
public void loadBooks() {  
    try ( BufferedReader br = new BufferedReader(  
        new FileReader( new File( this.filename ) ) ); ) {  
        while ( br.ready() ) {  
            String[] parts = br.readLine().split( ";" );  
            if ( parts.length == 4 ) {  
                this.books.add( new Book( parts[0], parts[1],  
                    new Integer( parts[2] ), parts[3] ) );  
            }  
        }  
    } catch ( Exception ex ) {  
        System.err.println( "Read error: " + ex.getLocalizedMessage() );  
    }  
}
```

# NIO.2 und Streams (2)

## ■ Mögliche Anpassung:

```
public void loadBooks() {
    Path p = Paths.get( this.filename );
    try {
        Files.readAllLines( p )    // alle Zeilen lesen
            .stream()              // als Stream weiterverarbeiten

            .map(l->l.split( ";" )) // aufsplitten an ";" zu String-Array

            .filter( l->l.length == 4 ) // alle Arrays rausfiltern, die nicht
                                        // genau 4 Einträge haben

            .map( l->new Book( l[0], l[1],    // Array in ein Buch-Objekt umwandeln
                             new Integer( l[2] ), l[3] ) )

            .forEach( this.books::add );    // jedes Buch in die Liste hinzufügen
    } catch (IOException ex) {
        System.err.println( "Read error: " + ex.getMessage() );
    }
}
```

## NIO.2 und Streams (3)

### ■ Weitere Möglichkeit (Mix aus „klassisch“ und Streams):

```
public void loadBooks() {
    Path p = Paths.get( this.filename );
    try {
        // Ergebnis der Stream-Operationen wird this.books zugewiesen!
        this.books = Files.readAllLines( p ) // alle Zeilen lesen
            .stream()                        // als Stream weiterverarbeiten
            .map( this::parseBook )          // Zeile in Buch umwandeln
            .filter( Objects::nonNull )     // null-Werte aussortieren
            .collect( Collectors.toList() ); // Stream in Liste umwandeln
    } catch (IOException ex) {
        System.err.println( "Read error: " + ex.getLocalizedMessage() );
    }
}

private Book parseBook(String line) {
    String[] p = line.split( ";" );
    if ( p.length == 4 ) {
        return new Book( p[0], p[1], new Integer( p[2] ), p[3] );
    } else {
        return null;
    }
}
```

# Literatur

- Processing Data with Java SE 8 Streams, Part 1+2  
<http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html>  
<http://www.oracle.com/technetwork/articles/java/architect-streams-pt2-2227132.html>
  
- **The Java™ Tutorials / Lesson: Aggregate Operations**  
<https://docs.oracle.com/javase/tutorial/collections/streams/index.html>