

# Workshop Java 8

## Ablauf

Der Workshop Java 8 gliedert sich jeweils in einen Vortragsteil, der den Teilnehmern einen spezifischen Bereich von Java 8 und die dortigen Neuerungen näherbringt. Im Anschluss daran sind jeweils einige Übungsaufgaben von den Teilnehmern einzeln oder als Gruppenarbeit am PC zu lösen.

## Voraussetzungen

- 1) Aktuelles JDK 8 installiert
- 2) Aktuelles Eclipse installiert  
(Alternative: NetBeans 8 oder IntelliJ IDEA)

## Teilnehmer

- SW-Entwickler mit Java-Erfahrung sowie
- SW-Architekten, die Java 8 kennenlernen/evaluieren möchten

## Begleitmaterial

- USB-Stick mit JDK 8, Eclipse-Projekt, Übungsaufgaben und Folien

## Kursleitung und Kontakt

Michael Inden

E-Mail: michael\_inden@hotmail.com

## Part 1: Lambdas, Methodenreferenzen, Default-Methoden

Lernziel: Kennenlernen der Basisbausteine der funktionalen Programmierung (Functional Interfaces, SAM-Typen, Lambdas, Methodenreferenzen und Defaultmethoden) anhand von einfachen Beispielen.

**Übung 1a:** Schaue auf das folgende Interface LongBinaryOperator.

```
public interface LongBinaryOperator
{
    long applyAsLong(final long left, final long right);
}
```

Was sind gültige Lambdas und wieso? Wofür erhält man Kompilierfehler?

```
final LongBinaryOperator v1 = (long x, Long y) -> { return x + y; };
final LongBinaryOperator v2 = (long x, long y) -> { return x + y; };
final LongBinaryOperator v3 = (long x, long y) -> x + y;
final LongBinaryOperator v4 = (long x, y) -> x + y;
final LongBinaryOperator v5 = (x, y) -> x + y;
final LongBinaryOperator v6 = x, y -> x + y;
```

**Tipp:** Überlege kurz und prüfe deine Vermutungen dann mithilfe der IDE.

**Übung 1b:** Welche Varianten kompilieren, wenn man das Interface wie folgt verändert?

```
public interface LongBinaryOperator
{
    long applyAsLong(final long left, final Long right);
}
```

**Übung 2a:** Implementiere ein Runnable mit einem Lambda, der die Konsolenausgabe „I am the first lambda and say hello“ vornimmt. Führe das Runnable wie folgt mit einem Thread aus.

```
final Runnable runner = ... // Lambda
new Thread(runner).start();
```

**Übung 2b:** Die zuvor erstellte Konsolenausgabe soll einen variablen Anteil bekommen. Wie kann man Werte oder Informationen an den Lambda übergeben?

**Tipp 1:** Greife auf eine außerhalb des Lambdas definierte Variable messageText zu, um statt des Worts „world“ einen beliebigen anderen Text auszugeben. Was ist für lokale Variablen zu bedenken? (Stichwort: effectively final)

**Tipp 2:** Nutze eine Methode provideTextOutput(String message), die einen Lambda vom Typ Runnable zurückgibt, um eine Nachricht konfigurierbar zu gestalten.

**Übung 3:** Überlege dir eine einfache grafische Notation (Kästchen und Pfeile) für folgende Functional Interfaces mit Eingaben und Ausgaben:

```
public interface Predicate<T> {
    boolean test(T t)
}


public interface BinaryOperator<T> {
    T apply(T t1, T t2);
}

public interface Comparator<T> {
    int compare(T t1, T t2);
}

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

Tipp: 

```
public interface Function<T,R> {
    R apply(T t)
}
```

 $\Rightarrow$   $T \rightarrow$    $\rightarrow R$

**Übung 4:** Konvertiere anonyme Klassen vom Typ `FileFilter`. Gegeben sind die beiden folgenden Implementierungen `directoryFilter` und `pdfFileFilter`, die a) Verzeichnisse und b) PDF-Dateien filtern.

```
final FileFilter directoryFilter = new FileFilter()
{
    @Override
    public boolean accept(final File pathname)
    {
        return pathname.isDirectory();
    }
};

final FileFilter pdfFileFilter = new FileFilter()
{
    @Override
    public boolean accept(final File pathname)
    {
        return (pathname.isFile() &&
            pathname.getName().toLowerCase().endsWith(".pdf"));
    }
};
```

Tipp: Verwende für den ersten `FileFilter` eine Methodenreferenz auf die Methode `isDirectory` und im zweiten einen Lambda.

**Übung 5:** Vereinfache zwei `Comparator<String>`, die a) nach Länge und b) case-insensitive sortieren durch den Einsatz von Lambdas. Nutze zudem die besser lesbare Sortierung basierend auf der Defaultmethode `List.sort()`:

```
final List<String> names = Arrays.asList("Josef", "Jörg", "Jürgen");

final Comparator<String> byLength = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return Integer.compare(str1.length(), str2.length());
    }
};

final Comparator<String> caseInsensitive = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return str1.compareToIgnoreCase(str2);
    }
};

Collections.sort(names, byLength);
System.out.println(names);

Collections.sort(names, caseInsensitive);
System.out.println(names);
```

**Tipp:**

```
final Comparator<String> byLengthJDK8 = (str1, str2) -> ...
final Comparator<String> caseInsensitiveJDK8 = (str1, str2) -> ...
```

### Übung 6: Eigene Functional Interfaces

Schreibe jeweils ein Functional Interfaces `StringToIntConverter` und `StringToStringConverter` mit einer Methode `convert()` für die folgenden Abbildungen

- a) `String` auf `int`
- b) `String` auf `String`

Prüfe die Funktionalität mit folgenden Zeilen und implementiere diese mit Lambdas:

```
StringToIntConverter stringToLength = ...;
StringToStringConverter stringToTrimmedString = ...;
StringToStringConverter stringToUpperCaseString = ...;

System.out.println("stringToLength('Java') => " +
    stringToLength.convert("Java"));
System.out.println("stringToTrimmedString(' Michael ') => '" +
    stringToTrimmedString.convert(" Michael ") + "'");
System.out.println("stringToUpperCaseString('Hands On') => " +
    stringToUpperCaseString.convert("Hands On"));
```

**Übung 7a:** Gegeben sei ein rudimentäres Fragment eines Functional Interfaces als Ergänzung den obigen String-Mappern:

```
public interface Mapper<S, T>
{
    // TODO: map S -> T
    // TODO: mapAll List<S> -> List<T>
    // ...
}
```

Vervollständige den Mapper, sodass folgendes Programm kompiliert:

```
public static void main(final String[] args)
{
    List<String> names = Arrays.asList("Tim", "Andi", "Michael");

    final Mapper<String, Integer> intMapper = String::length;
    System.out.println(intMapper.mapAll(names));

    final Mapper<String, String> stringMapper = str -> ">> " +
                                                    str.toUpperCase() + " <<";
    final List<String> upperCaseNames = stringMapper.mapAll(names);
    System.out.println(upperCaseNames);
}
```

Das erwartete Ergebnis ist

```
[3, 4, 7]
[>> TIM <<, >> ANDI <<, >> MICHAEL <<]
```

**Tipp:** Stelle die `mapAll()`-Funktionalität mithilfe einer Defaultmethode bereit.

**Übung 7b:** Welche Alternativen bietet das JDK 8? Schaue ins Package `java.util.function`. Dazu benötigt man aber dann auch eine Methode `applyAll()` analog zu `mapAll()` mit folgender Signatur:

```
public static <T,R> List<R> applyAll(final List<T> sourceElements,
                                     final Function<T, R> func)
```

**Übung 7c:** Was ist von Defaultmethoden in Applikationsklassen zu halten? Welche Vorteile bringen diese, welche Fallstricke sollte man bedenken? Diskutiere mit deinem Nachbarn!

## Workshop Java 8

**Übung 8a:** Was ist bei Lambdas und dort ausgelösten Checked Exceptions zu bedenken?  
Wie kann man folgendes `Runnable` so korrigieren, dass es kompiliert?

```
final Runnable runner = () -> { System.out.println("Throwing");  
                                throw new IOException();  
};
```

**Übung 8b:** Realisiere ein Functional Interface als Erweiterung zu `Runnable`, dass eine Checked Exception in den Typ `RuntimeException` wandelt.

```
@FunctionalInterface  
public interface RunnableThatThrows extends Runnable  
{  
    @Override  
    default void run()  
    {  
        // TODO  
    }  
  
    public void runThrows() throws Exception;  
}
```

Der Aufruf soll dann wie folgt möglich sein:

```
public class Exercise8_RunnableThatThrowsExample  
{  
    public static void main(String[] args)  
    {  
        final RunnableThatThrows runner2 = () ->  
            { System.out.println("RunnableThatThrows"); throw new IOException(); };  
        runner2.run();  
    }  
}
```

**Übung 8c:** Realisiere ein Functional Interface als Erweiterung zu `Comparator`, dass eine Checked Exception in den Typ `RuntimeException` wandelt basierend auf folgenden Zeilen:

```
@FunctionalInterface  
public interface ComparatorThatThrows<T> extends Comparator<T>  
{  
    public int compareThrows(final T t1, final T t2) throws Exception;  
}
```

## Workshop Java 8

### KÜR:

Beschäftige dich mit der Frage: Wie kann man mit Lambdas mit etwas tricksen rekursive Aufrufe formulieren? Widerlege also die Aussage von Angelika Langer, dass Rekursion in Lambdas nicht möglich ist. Starte mit den folgenden, *nicht kompilierfähigen* Ausdrücken:

```
Function<Integer, Integer> myFactorialInt =  
    i -> i == 0 ? 1 : i * myFactorialInt.apply( i - 1 );  
Function<Integer, Long> myFactorialLong =  
    i -> i == 0L ? 1L : i * myFactorialLong.apply( i - 1 );
```

**Tipp:** Versuche es mit einem statischen Attribut oder einem Instanzattribut,

Prüfe deine Lösung mit folgenden Aufrufen:

```
myFactorialInt.apply(5); myFactorialLong.apply(5)  
myFactorialInt.apply(20); myFactorialLong.apply(20)
```

Die Ausgaben sind vermutlich wie folgt:

```
120  
120  
-2102132736  
2432902008176640000
```

Man erkennt den Überlauf für den Typ `Integer` bzw. `int`. Was gibt es für Lösungsmöglichkeiten? Schau mal in die Klasse `Math`. Seit Java 8 findet man dort Abhilfe.

## Part 2: Collections und Bulk-Operations sowie Grundlagen dazu

Lernziel: Die Stärke von Lambdas kann insbesondere im Zusammenhang mit Bulk Operations für Collections ausgespielt werden. Nachfolgend vertiefen wir das Wissen zu Prädikaten, interner und externer Iteration sowie zu neuen Hilfsmethoden in den Containerklassen bzw. deren Interfaces wie z. B. `List<E>`.

**Übung 1a:** Formuliere die Bedingungen „Zahl ist gerade“, „Zahl ist Null“, „Zahl ist positiv“ als `java.util.function.Predicate<Integer>` und/oder als `java.util.function.IntPredicate` und prüfe diese mit Test-Eingaben.

```
final Predicate<Integer> isEven = // ... TODO ...
final Predicate<Integer> isZero = // ... TODO ...
final Predicate<Integer> isPositive = // ... TODO ...
```

**Übung 1b:** Kombiniere die Prädikate wie folgt und prüfe wieder:

- Zahl ist positiv und Zahl ist gerade (Nutze `and()`)
- Zahl ist positiv und ungerade (Nutze `negate()`)

**Übung 1c:** Formuliere die Bedingung „Wort kürzer als 4 Buchstaben“ und prüfe das damit realisierte `Predicate<String> isShortWord`.

**Übung 2:** Gegeben sei folgende Klasse:

```
class Person
{
    final String name;
    final int age;

    public Person(final String name, final int age)
    {
        this.name = name;
        this.age = age;
    }

    public int getAge()
    {
        return age;
    }

    public boolean isAdult()
    {
        return getAge() >= 18;
    }
    // ...
}
```

Formuliere die Bedingung „18 oder älter“ mit einem `Predicate<Person>`

- a) durch Aufruf von `getAge()` (als Lambda)
- b) durch Aufruf von `isAdult()` (mit Methodenreferenz)



**Übung 3:** Wandle eine externe (for-Schleife) in eine interne Iteration um:

```
final List<String> names = Arrays.asList("Tim", "Peter", "Mike");
for (final String name : names)
{
    System.out.println(name);
}
```

**Tipp:** Nutze die Default-Methode `forEach()` aus dem Interface `Iterable<E>`. Verwende hierbei die Methodenreferenz `System.out::println`.

**Übung 4a:** Lösche aus einer Liste von Namen all diejenigen mit kurzem Namen und gib das Ergebnis auf der Konsole aus. Wandle auch hier eine externe in eine interne Iteration um. Gegeben ist dazu folgende JDK-7-Implementierung, die mithilfe von JDK-8-Mitteln einfacher gestaltet werden soll.

```
public List<String> removeIf_External(final List<String> names)
{
    final Iterator<String> it = names.iterator();
    while (it.hasNext())
    {
        final String currentName = it.next();
        if (currentName.length() < 4)
        {
            it.remove();
        }
    }

    return names;
}

private static List<String> createNamesList()
{
    final List<String> names = new ArrayList<>();
    names.add("Michael");
    names.add("Tim");
    names.add("Andy");
    names.add("Flo");
    names.add("Yannis");
    names.add("Clemens");
    return names;
}
```

**Tipp:** Nutze das in Aufgabe 1 erstellte `Predicate<T> isShortWord` und die Methode `removeIf(Predicate<T>)` aus dem Interface `Collection<E>`.

**Übung 4b:** Lösche aus der Liste von Namen all diejenigen mit ungerader Länge und gib das Ergebnis auf der Konsole aus.

**Übung 4c:** Lösche aus der Liste von Namen all diejenigen an ungerader Position und gib das Ergebnis auf der Konsole aus. Was stellst du dabei fest?

**Übung 5a:** Modifiziere eine Liste mit Namen: Verändere dort all diejenigen Namen, die mit einem „M“ starten: Aus „Michael“ wird dann „ >>MICHAEL<<“. Als Ausgangsbasis dient folgender JDK-7-Sourcecode:

```
private static List<String> replaceAll_External_Iteration(
    final List<String> names)
{
    final ListIterator<String> it = names.listIterator();
    while (it.hasNext())
    {
        final String currentName = it.next();
        if (currentName.startsWith("M"))
        {
            it.set(">>" + currentName.toUpperCase() + "<<");
        }
    }

    return names;
}

private static List<String> createNamesList()
{
    final List<String> names = new ArrayList<>();
    names.add("Michael");
    names.add("Tim");
    names.add("Flo");
    names.add("Merten");
    return names;
}
```

**Tipp:** Nutze eine Implementierung eines `UnaryOperator<T>` und die Methode `replaceAll(UnaryOperator<T>)` aus dem Interface `Collection<E>`.

**Übung 5b:** Ergänze eine weitere Aktion: Alle Wörter mit „i“ oder „l“ sollen umgedreht werden. Also wird aus Michael => leahciM und aus Tim => miT. Und in Kombination mit dem vorherigen: <<LEAHCIM>> und miT

**Übung 6:** Lösche aus einer Liste von Namen all diejenigen mit kurzem Namen (3 oder weniger Zeichen). Filtere zuvor alle leeren Einträge oder null-Werte heraus. Zudem sollen alle Leerzeichen um die Namen entfernt werden. Nutze die Neuerungen aus Collections.

```
final List<String> names = new ArrayList<>(Arrays.asList("Jan",
    "Michael ", "", " Tim ", null, " Tom ", " Marius "));
```

=> [Michael, Marius]

**Tipp:** Versuche es mit `removeIf()` sowie `replaceAll()`. Denke daran, komplexere Lambdas als Hilfsvariablen zu definieren. Das erhöht die Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit

### Part 3: Streams und Filter-Map-Reduce

**Lernziel:** In diesem Abschnitt wollen wir die Arbeit mit den in JDK 8 neu eingeführten Streams kennenlernen und dabei insbesondere auf das Filter-Map-Reduce-Framework eingehen. Damit können auch komplexere Berechnungen in Form kleiner Verarbeitungsschritte modelliert werden.

**Übung 1a:** Welche drei Arten von Operationen gibt es? Was sind typische Vertreter?

**Übung 1b:** *Create:* Wie erzeugt man einen `Stream<String>`?

```
final String[] namesArray = {"Tim", "Tom", "Andy", "Mike", "Merten"};
final List<String> names = Arrays.asList(namesArray);

final Stream<String> streamFromArray = // ...
final Stream<String> streamFromList = // ...
final Stream<String> streamFromValues = // ...
```

**Tipp:** Nutze `Arrays.stream()`, `List.stream()`, `Stream.of()`.

**Übung 1c:** *Intermediate:* Filtere eine Liste von Namen (alle startend mit „T“) und führe ein Mapping aus, z. B. auf `UPPERCASE` aus.

```
final Stream<String> filtered = // ...
final Stream<String> mapped = // ...
```

**Tipp:** Nutze `Stream.filter()` bzw. `Stream.map()`.

**Übung 1d:** *Terminal:* Gib den Inhalt des Streams auf der Konsole aus.

**Tipp:** Nutze `Stream.forEach()`.

**Übung 2:** Wenn eine Ausgabe kommaseparierte gewünscht ist, so kann man das wie folgt realisieren – es wird jedoch abschließend ein Komma zu viel ausgegeben.

```
final String[] namesArray = {"Tim", "Tom", "Andy", "Mike", "Merten"};
final List<String> names = Arrays.asList(namesArray);

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

Statt zur Korrektur eine Spezialbehandlung des letzten Zeichens (wie weiß man es bei einem Stream und bei einem parallelen Stream?) zu nutzen, bietet es sich an, vordefinierte Methoden einzusetzen.

Die erwartete Ausgabe ist: Tim, Tom, Andy, Mike, Merten

**Tipp:** Nutze `collect()` und `Collectors.joining(", ")` zur Ausgabe.

**Übung 3:** Wandle den Inhalt eines Streams in ein Array mit `toArray()` bzw. in eine Liste mit `collect(Collectors.toList())`:

```
final Object[] contentsAsArray = streamFromArray...  
final List<String> contentsAsList = streamFromValues...
```

**Tipp:** Verwende die Vorarbeiten aus Aufgabe 1.

**Übung 4:** Zuvor (S. 10 Übung 6) haben wir die Defaultmethoden aus `Collection` genutzt, um aus einer Liste von Namen all diejenigen mit kurzem Namen zu entfernen. Filtere wieder alle leeren Einträge oder `null`-Werte heraus und entferne Leerzeichen um die Namen. Verwende hierzu nun das Stream-API. Vergleiche die Lösungen. Was gilt es zu beachten oder bedenken?

```
final List<String> names = new ArrayList<>(Arrays.asList("Jan", "Michael ", "", " Tim ",  
                                                         null, " Tom ", " Marius "));
```

=> `[Michael, Marius]`

**Übung 5:** Quadriere alle ungeraden Zahlen von 1 bis 10 und ermittle deren Summe:

```
final int result = IntStream.of(1,2,3,4,5,6,7,8,9,10).filter(...).  
                                                         map(...).  
                                                         reduce(0, ...);
```

**Tipp:** Verwende lokale Hilfsvariablen, um die Verarbeitungsabfolgen besser lesbar zu gestalten. Möglicherweise ist `Integer::sum` beim Aufruf von `reduce()` nützlich.

**Übung 6a:** Finde mindestens drei unterschiedliche Arten, die Werte von 1 bis 10 als Stream zu erzeugen.

**Übung 6b:** Generiere die Ziffernfolge der natürlichen Zahlen als unendlichen Stream. Starte die Ausgabe bei 11 und begrenze diese auf 10 Elemente.

**Tipp:** Nutze die Methoden `skip()`, `limit()`, `generate(Supplier)` und `iterate()`. Zur Umwandlung eines `IntStream` in einen `Stream<Integer>` dient die Methode `boxed()`. `collect()` und `Collectors.joining()` hilft bei der Ausgabe.

**Übung 6c:** Bedenke, dass unendliche Streams problematisch sein können — insbesondere beim Einsatz der Methode `sorted()`. Führe folgende Anweisungen zunächst ohne `sorted()` und danach mit aus. Schau, was passiert:

```
public class Exercise6_InfiniteStreamSurprises
{
    public static void main(String[] args)
    {
        IntStream.iterate(0, i -> i + 1).
                    boxed().
                    //      sorted().
                    limit(10).
                    map(e -> "" + e).
                    forEach(System.out::println);
    }
}
```

**Übung 7:** Erstelle ein Programm, das den Ablauf der Verarbeitungs-Pipeline mithilfe von Konsolenausgaben visualisiert. Einen Startpunkt liefern diese Zeilen:

```
Stream.of("Andi", "Barbara", "Marius", "Merten", "Micha", "Tim").
    map(name -> { System.out.println("map: " + name);
                  return name.toUpperCase(); }).
    filter(name -> { System.out.println("filter: " + name);
                     return name.startsWith("M"); }).
    forEach(name -> System.out.println("forEach: " + name));
```

**Tipp:** Verbessere das Hineinschauen in den Stream durch Aufruf von `peek(Consumer)`. Welche Erkenntnisse gewinnst du anhand der Ausgaben? In welcher Reihenfolge sollte man die Operationen ausführen? Tausche einmal die Aufrufe von `filter()` und `map()`.

**Übung 8a:** Gruppier die Namen nach dem Anfangsbuchstaben, sodass wir folgendes Ergebnis erhalten: {A=[Andy], T=[Tim, Tom], M=[Mike, Merten]}

```
final String[] namesArray = {"Tim", "Tom", "Andy", "Mike", "Merten"};
...
final Map<Character, List<String>> grouped = // ...
System.out.println(grouped);
```

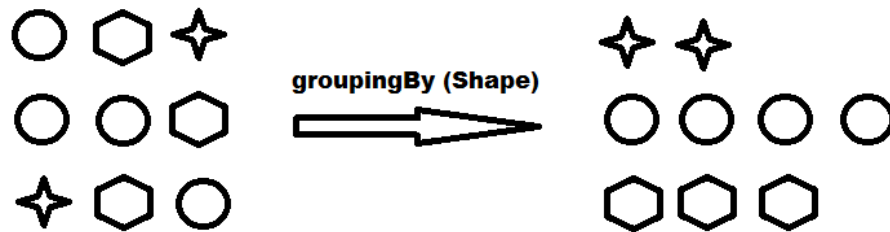
**Übung 8b:** Gruppier den Inhalt nach dem Anfangsbuchstaben und summiere dann die Anzahl der Vorkommen, sodass wir folgende Ausgabe erhalten: {A=1, T=2, M=2}

```
final Map<Character, Long> groupedAndCounted = // ...
System.out.println(groupedAndCounted);
```

**Tipp:** Nutze `collect()` sowie die Methoden `Collectors.groupingBy()` und `Collectors.counting()`.

**Übung 9:** Finde eine grafische Notation für die Operationen `filter()`, `map()` und `Collectors.groupingBy()` sowie `Collectors.partitioningBy()`.

Tipp:



**Übung 10:** Gegeben seien die Zahlen von 1 bis 10 als Eingabe. Berechne das Minimum, Maximum, die Summe und den Durchschnitt dieser Zahlenfolge.

```
final IntStream ints1 = IntStream.of(1,2,3,4,5,6,7,8,9,10);  
final IntStream ints2 = IntStream.range(0, 11);  
final IntStream ints3 = IntStream.rangeClosed(0, 10);
```

Tipp: Verwende die Stream-Methoden `min()`, `max()`, `sum()` usw.  
Was ist daran unschön? Schau einmal auf die Klasse `IntSummaryStatistics`.

**Übung 11:** Realisiere die mit JDK 9 bereitgestellte Funktionalität von `takeWhile()` und `dropWhile()` basierend auf JDK-8-Bordmitteln:

- `takeWhile(Predicate<T>)` verarbeitet Elemente des Streams, solange die als `Predicate<T>` übergebene Bedingung erfüllt ist.
- `dropWhile(Predicate<T>)` überspringt Elemente des Streams, solange die als `Predicate<T>` übergebene Bedingung erfüllt ist.

```
final Stream<Integer> ints1 = Stream.of(1,2,3,4,5,6,7,8,9,10,11,12,13,14);  
final Stream<Integer> ints2 = Stream.of(1,2,3,4,5,6,7,8,9,10,11,12,13,14);  
final Predicate<Integer> isBelow10 = n -> n < 10;
```

```
takeWhile  
1, 2, 3, 4, 5, 6, 7, 8, 9  
  
dropWhile 1  
10, 11, 12, 13, 14
```

Tipp: Probiere verschiedene Realisierungsvarianten von `dropWhile()` ... schau doch mal ins Interface `Iterator<T>` und dessen Methode `forEachRemaining()`.

**Übung 12:** Experimentiere ein wenig mit folgendem Sourcecode. Je nach Variante kann man ein Worthäufigkeitshistogramm erstellen, oder aber nur selektiv die Vorkommen einzelner Wörter. Was passiert, wenn man alternativ den Lambda

(key2,value) -> value++  
bzw. (key2,value) -> ++value nutzt?

```
public class MapComputeIfPresentExample
{
    public static void main(String[] args)
    {
        final Map<String, Integer> wordCounts = new LinkedHashMap<>();

        final String shortStory =
            "Diese Geschichte IST kurz und handelt von TIM. TIM wohnt in Kiel. TIM arbeitet in " +
            "Bremen. TIM IST während der Arbeitswoche dort. TIM fährt donnerstags nach Hause.";

        // Einkommentieren, um spezielle Worte zu zählen
        // wordCounts.put("TIM", 0);
        // wordCounts.put("IST", 0);

        // for (final String word : shortStory.split(" "))
        for (final String word : shortStory.split("( |\\.)"))
        {
            // Histogramm, putIfAbsent auskommentieren, um spezielle Worte zu zählen
            wordCounts.putIfAbsent(word, 0);
            wordCounts.computeIfPresent(word, (key,value) -> value +1);
        }
        System.out.println(wordCounts);
    }
}
```

## Part 4: Date And Time API

Lernziel: Die Verarbeitung von Datumswerten wurde mit Java 8 komplett überarbeitet und deutlich vereinfacht. Die Übungen liefern einen Einstieg in die neuen Klassen, in Berechnungen und weitere Hilfsfunktionalitäten.

**Übung 1:** Berechne die Zeit zwischen heute und deinem Geburtstag und vice versa?

```
final LocalDate now = LocalDate.now();  
final LocalDate birthday = LocalDate.of(1971, 2, 7);
```

Tipp: Nutze die Klasse `LocalDate` und die Methode `until()`. Nutze als Variante auch den Aufruf von `Period.between()`.

**Übung 2a:** Erzeuge eine Zeitdauer von 1 Jahr, 2 Monaten und 14 Tagen mit der Klasse `Period` und prüfe die Konsolenausgabe (`P1Y2M14D`).

**Übung 2b:** Wie lautet die Ausgabe für eine Hintereinanderausführung von `ofYears(1)`, `ofMonths(2)` und `ofDays(14)`?

**Übung 2c:** Kombiniere die Methode `ofYears()` und `withXYZ()`-Methoden zum Erzeugen. Welche Ausgabe erhält man dann? Und warum?

**Übung 3:** Beantworte folgende Fragen und führe dazu die Aktionen aus:

- a) Welcher Wochentag war der Heiligabend 2013 (24.12.2013)?
- b) Welchen Wochentag haben wir heute?
- c) springe ausgehend von Heiligabend 7 Tage in die Vergangenheit und die zuvor erzeugte `Period` (`P1Y2M14D`) in die Zukunft
- d) der wievielte Tag im Monat bzw. Jahr ist das jeweils?

Tipp: Die Klasse `LocalDate` bietet die Methoden `getDayOfMonth()` sowie `getDayOfYear()`.

**Übung 4:** Berechne das Datum des ersten und letzten Freitags und Sonntags im März 2014. Starte mit folgendem Sourcecode-Schnipsel:

```
final LocalDate midOfMarch = LocalDate.of(2014, 3, 15);  
final TemporalAdjuster toFirstSunday =  
    TemporalAdjusters.firstInMonth(DayOfWeek.SUNDAY);
```

Tipp: Nutze dabei die Klasse `TemporalAdjusters` und ihre Hilfsmethoden und die Methode `with()` aus der Klasse `LocalDate`.



**Übung 5:** Ermittle alle Zeitzonen, die mit den Präfixen „America/L“ oder „Europe/S“ starten. Gib diese sortiert auf der Konsole aus.

**Tipp:** Nutze die Klasse `ZoneId` sowie deren Methode `getAvailableZoneIds()`. Setze Streams und das Filter-Map-Reduce-Framework ein, um die gewünschten Zeitzonen-Ids herauszufiltern.

**Übung 6:** Wenn ein Flug von Zürich nach San Francisco 11,5 Stunden dauert und am 7.7.2014, um 14:30 h startet, wann ist man dann in San Francisco? Welcher Zeit entspräche das am Abflugsort? Beginne die Berechnungen wie folgt:

```
final LocalDate departureDate = LocalDate.of(2014, 7, 7);
final LocalTime departureTime = LocalTime.of(14, 30);
final ZoneId zoneEurope = ZoneId.of("Europe/Zurich");
final ZonedDateTime departure =
    ZonedDateTime.of(departureDate, departureTime, zoneEurope);
```

**Tipp:** Nutze die Klasse `Duration` und die Methoden `ofHours()` sowie `plusMinutes()`, um die Flugzeit zu modellieren. Die Zeitzone in San Francisco ist „America/Los\_Angeles“. Verwende die Methoden `plus()` sowie `withZoneSameInstant()` aus der Klasse `ZonedDateTime`.

**Übung 7:** Erzeuge ein `LocalDateTime`-Objekt und gib es in verschiedenen Formaten aus. Formatiere und parse es aus diesen Formaten: `dd.MM.yyyy HH`, `dd.MM.yy HH:mm`, `ISO_LOCAL_DATE_TIME`, `SHORT ENGLISH`.

```
public static void main(final String[] args)
{
    final LocalDateTime jdk9Release =
        LocalDateTime.of(2017, 7, 27, 13, 14, 15);

    final DateTimeFormatter formatter1 = // ..
    final DateTimeFormatter formatter2 = // ..

    // ..
}
```

Die Ausgabe sollte wie folgt sein:

```
Formatted: 27 07 2017 13 / Parsed: 2017-07-27T13:00
Formatted: 27.07.17 13:14 / Parsed: 2017-07-27T13:14
Formatted: 2017-07-27T13:14:15 / Parsed: 2017-07-27T13:14:15
Formatted: 27 Jul 2017, 13:14:15 / Parsed: 2017-07-27T13:14:15
```

**Tipp:** Nutze die Klasse `DateTimeFormatter` sowie ihre Konstanten und Methoden wie `ofPattern()` und `ofLocalizedDateTime()`.

**Übung 8:** Gerade bei der Auswertung von Benutzereingaben kommt es oft auf Fehlertoleranz an. Erstelle eine Utility-Methode `faulttolerantparse()`, die es erlaubt, folgende Datumsformate zu parsen: `dd.MM.yy`, `dd.MM.yyyy`, `MM/dd/yyyy` sowie `yyyy-MM-dd`.

```
public static void main(final String[] args)
{
    final List<String> exampleDates = Arrays.asList("07.02.71",
                                                    "07.02.1971", "02/07/1971", "1971-02-07");

    for (final String dateAsString : exampleDates)
    {
        final LocalDate parsedLocalDate =
            faulttolerantparse(dateAsString);
        System.out.println("Parsed '" + dateAsString + "' into: " +
                           parsedLocalDate);
    }
}
```

Die Ausgabe sollte wie folgt sein:

```
Parsed '07.02.71' into: 2071-02-07
Parsed '07.02.1971' into: 1971-02-07
Parsed '02/07/1971' into: 1971-02-07
Parsed '1971-02-07' into: 1971-02-07
```

## KÜR 1:

Gegeben ist folgender selbstdefinierter `TemporalAdjuster`, der einen Datumswert auf den Anfang des jeweiligen Quartals setzt: Demnach springt man am 18. August auf den 1. Juli usw. Eine Implementierung für diese Datumsarithmetik findet man im Internet unter <http://www.leveluplunch.com/java/examples/first-day-of-quarter-java8-adjuster/>.

Betrachte den Sourcecode und überlege, was ungünstig an dieser Implementierung ist:

```
public class Exercise04_FirstDayOfQuarterOrig implements TemporalAdjuster
{
    @Override
    public Temporal adjustInto(Temporal temporal)
    {
        int currentQuarter = YearMonth.from(temporal).
            get(IsoFields.QUARTER_OF_YEAR);

        if (currentQuarter == 1)
        {
            return LocalDate.from(temporal).
                with(TemporalAdjusters.firstDayOfYear());
        }
        else if (currentQuarter == 2)
        {
            return LocalDate.from(temporal).withMonth(Month.APRIL.getValue()).
                with(firstDayOfMonth());
        }
        else if (currentQuarter == 3)
        {
            return LocalDate.from(temporal).withMonth(Month.JULY.getValue()).
                with(firstDayOfMonth());
        }
        else
        {
            return LocalDate.from(temporal).withMonth(Month.OCTOBER.getValue()).
                with(firstDayOfMonth());
        }
    }
}
```

**Tipp:** Was kann man vereinfachen? Entferne Spezialbehandlungen und Inkonsistenzen! Wenn du eine auf Monaten basierende Lösung hast, schaue nochmals in die Aufzählung `IsoFields`. Dort findet man auch `DAY_OF_QUARTER`.

Ein nutzendes Programm sieht wie folgt aus:

```
public static void main(final String[] args)
{
    final LocalDate midOfMarch = LocalDate.of(2014, 3, 15);
    final LocalDate midOfJune = LocalDate.of(2014, 6, 15);
    final LocalDate midOfSep = LocalDate.of(2014, 9, 15);
    final LocalDate midOfNov = LocalDate.of(2014, 11, 15);

    final TemporalAdjuster toFirstDay = new FirstDayOfQuarterOrig();

    final LocalDate[] dates = {midOfMarch, midOfJune, midOfSep, midOfNov};
    for (final LocalDate date : dates)
    {
        System.out.print(date.with(toFirstDay) + " / ");
    }
}
```

Die erwarteten Ausgaben sind folgende: 2014-01-01 / 2014-04-01 / 2014-07-01 / 2014-10-01

## KÜR 2: NthWeekdayAdjuster

Schreibe eine Klasse `NthWeekdayAdjuster` als eigene Realisierung des Interface `TemporalAdjuster`, die auf den n-ten Wochentag springt, z. B. den 3. Freitag im Monat. Test diese Klasse mit dem folgenden Programm:

```
public static void main(final String[] args)
{
    final LocalDate aug18 = LocalDate.of(2015, Month.AUGUST, 18);
    System.out.println("Starting at " + aug18);

    final LocalDate nextFriday =
        aug18.with(TemporalAdjusters.next(DayOfWeek.FRIDAY));
    System.out.println("Next friday: " + nextFriday);

    System.out.println("2nd friday: " + aug18.with(
        new NthWeekdayAdjuster(DayOfWeek.FRIDAY, 2)));
    System.out.println("3rd sunday: " + aug18.with(
        new NthWeekdayAdjuster(DayOfWeek.SUNDAY, 3)));
    System.out.println("4th tuesday: " + aug18.with(
        new NthWeekdayAdjuster(DayOfWeek.TUESDAY, 4)));
}
```

Die erwarteten Ausgaben sind folgende:

```
Starting at 2015-08-18
Next friday: 2015-08-21
2nd friday: 2015-08-14
3rd sunday: 2015-08-16
4th tuesday: 2015-08-25
```

## PART 5: Diverses

Lernziel: Java 8 enthält eine Vielzahl an nützlichen Funktionalitäten unter anderem in den Klassen und Interfaces `Comparator<T>`, `Optional<T>` und `CompletableFuture<T>`. Diese Neuerungen werden hier mithilfe von Übungsaufgabe erkundet.

**Übung 1a:** Sortiere eine Menge von Strings nach deren Länge, nutze dabei die Erweiterungen im Interface `java.util.Comparator<T>`. Als Ausgangsbasis schauen wir auf eine Realisierung mit JDK 7 und inneren Klassen:

```
public static void main(final String[] args)
{
    final List<String> names = createNamesList();

    final Comparator<String> byLength = new Comparator<String>()
    {
        @Override
        public int compare(final String str1, final String str2)
        {
            return Integer.compare(str1.length(), str2.length());
        }
    };

    Collections.sort(names, byLength);
    System.out.println(names);
    // [Tim, Flo, Jörg, Andy, Michael, Clemens].
}

private static List<String> createNamesList()
{
    final List<String> names = new ArrayList<>();
    names.add("Michael");
    names.add("Tim");
    names.add("Jörg");
    names.add("Flo");
    names.add("Andy");
    names.add("Clemens");
    return names;
}
```

Tipp: Nutze `Comparator<T>.comparing()`

**Übung 1b:** In der Ausgabe sind die Namen nach Länge sortiert, aber innerhalb der Länge nicht alphabetisch. Entdecke weitere Möglichkeiten aus `Comparator<T>`, um das zu korrigieren. Ändere danach die Sortierung nach Länge wie folgt:

```
[Flo, Tim, Andy, Jörg, Clemens, Michael]
[Clemens, Michael, Andy, Jörg, Flo, Tim]
```

Tipp: `Comparator<T>.thenComparing()/naturalOrder()/reversed()`

**Übung 2:** Ermittle die älteste Person, vereinfache die nachfolgende JDK-7-Implementierung. Bilde insbesondere auch die Optionalität des Ergebnisses sauber ab.

```
public static Person findOldestPersonJDK7(final List<Person> persons)
{
    Person oldestPerson = null;
    for (final Person person : persons)
    {
        if (oldestPerson == null || person.getAge() > oldestPerson.getAge())
        {
            oldestPerson = person;
        }
    }
    return oldestPerson;
}
```

```
final List<Person> persons = Arrays.asList(new Person("Tim", 44), new Person("Tom", 6),
                                           new Person("Mike", 28));
```

=>

Person [name=Tim, age=44]

**Übung 3:** Gestalte das folgende Programmfragment mit Optional<T> um. Die Methode findPersonByName(String) soll nun ein Optional<T> zurückgeben. Auch Teile aus processPerson(Person) können modifiziert und aus der Methode herausgezogen werden.

```
public static void main(final String[] args)
{
    final Person person1 = findPersonByName("unknown");
    processPerson(person1);
    final Person person2 = findPersonByName("Micha");
    processPerson(person2);
}

private static void processPerson(final Person person)
{
    if (person != null)
    {
        System.out.println(person);
    }
    else
    {
        System.out.println("No person found!");
    }
}
```

**Übung 4:** Mit Java 8 kann man für `Optional<T>` mit `ifPresent()` im Erfolgsfall eine Aktion ausführen. Oftmals wünscht man sich dies auch für den Negativfall. Das kommt erst mit Java 9. Realisiere die mit JDK 9 bereitgestellte Funktionalität von `ifPresentOrElse()`, die dazu dient, nicht nur im Positivfall, sondern auch im Negativfall eine Aktion ausführen zu können. Gewünscht ist es folgende Funktionalität in eine Utility-Methode umzuwandeln:

```
final String desiredName = "Unknown";

final Optional<Customer> optCustomer = findCustomerByName(desiredName);

if (optCustomer.isPresent())
{
    printCustomerDetails(optCustomer.get());
}
else
{
    showWarnMessage("No such customer with name '" + desiredName + "'");
}
```

Das Ganze soll sich wie folgt nutzen lassen:

```
public static void main(final String[] args)
{
    final Optional<String> optCustomer1 = findCustomer("Tim");

    ifPresentOrElse(optCustomer1, customer -> System.out.println("found: " + customer),
        () -> System.out.println("not found"));
}
```

Die Suchmethode sei folgendermaßen implementiert:

```
private static Optional<String> findCustomer(final String customerId)
{
    System.out.println("findCustomer(" + customerId + ")");

    final Stream<String> customers = Stream.of("Tim", "Tom", "Mike", "Andy");
    if (customers.anyMatch(name -> name.contains(customerId))
    {
        return Optional.of(customerId);
    }

    return Optional.empty();
}
```

**Übung 5:** Wir wollen `Optional<T>` nutzen und so potenziell unsichere Aufrufketten, wie folgende

```
final String version = computer.getGraphicscard().getFirmware().getVersion();
```

abmildern. Diese wirken harmlos, bergen allerdings immer auch die Gefahr von `NullPointerExceptions`. Deshalb wollen wir uns eine Hilfsmethode `safeResolve()` basierend auf dem Interface `Supplier<T>` schreiben, um dann Folgendes ausführen zu können:

```
final Optional<T> optValue =  
    safeResolve( () -> computer.getGraphicscard().getFirmware().getVersion());  
  
optValue.ifPresent(System.out::println);
```

**Übung 6:** Normalerweise nutzt man `CompletableFuture<T>` zur Ausführung länger dauernder Aktionen. Diese werden hier durch einfache Berechnungen stilisiert, etwa das Berechnen eines Ergebnisses. Es sollen folgende Schritte realisiert werden:

- a) Schritt 1: Aufwendige Berechnung, hier nur Rückgabe eines Strings
- b) Schritt 2: Konvertierung, hier nur Abbildung von `String` auf `Integer`
- c) Schritt 3: Konvertierung in `Double` durch Multiplikation mit 0.75

```
final Supplier<String> longRunningAction = () ->  
{  
    System.out.println("Current thread: " + Thread.currentThread());  
    return "101";  
};  
  
final CompletableFuture<String> step1 = // ...  
  
// Schritt 2: Konvertierung, hier nur Abbildung von String auf Integer  
final Function<String, Integer> complexConverter = // ...  
final CompletableFuture<Integer> step2 = // ..  
  
// Schritt 3: Konvertierung, hier nur Multiplikation mit .75  
final Function<Integer, Double> complexCalculation = value -> .75 * value;  
final CompletableFuture<Double> step3 = // ..  
  
// Explizites Auslesen per get() löst die Verarbeitung aus  
System.out.println(step3.get());
```

**Übung 7:** Als praxisnahes Beispiel für `CompletableFuture<T>` soll der Inhalt einer Datei eingelesen und analysiert werden. Die notwendige Funktionalität ist in Form von Methoden mit folgenden Aktionen realisiert: Das Einlesen der Datei geschieht zeilenweise. Die Zeilen werden dann als einzelne Worte aufbereitet. Diese Funktionalität ist Schritt 1 und wird durch die Methode `extractWordsFromFile(Path)` realisiert. Im Anschluss daran sollen in einem zweiten Schritt zwei



Filterungen parallel auf den bereitgestellten Daten durchgeführt werden: Zum einen werden zu ignorierende Wörter herausgefiltert. Das geschieht durch die Methode `removeIgnorableWords()`. Zum anderen werden Wörter mit weniger als vier Buchstaben aus dem Ergebnis entfernt (`removeShortWords()`). Beide Verarbeitungen sollen parallel zu einander ablaufen. In Schritt 3 sollen die beiden parallel berechneten Ergebnisse miteinander verbunden werden. Damit ergibt sich folgendes Grundgerüst:

```
final Path exampleFile = Paths.get("Example.txt");
// Schritt 1: Möglicherweise längerdauernde Aktion final
CompletableFuture<List<String>> contents = // ..
contents.thenAccept(text -> System.out.println("Initial: " + text));

// Schritt 2: Filterungen parallel ausführen final
final CompletableFuture<List<String>> filtered1 = // ..
final CompletableFuture<List<String>> filtered2 = // ..

// Schritt 3: Verbinde die Ergebnisse
final CompletableFuture<List<String>> result = // ..
System.out.println("result: " + result.get());
```

**Übung 8:** Befülle ein `int[200]` wiederholend mit den Werten `0..99 0..99`. Erzeuge also folgende Füllung: `0 1 2 3 4 ... 98 99 0 1 2 ... 97 98 99`. Mit JDK 7 könnte man das Ganze wie folgt realisieren. Schreibe das eleganter und kürzer mit JDK 8:

```
public static void main(final String[] args)
{
    final int[] original = new int[200];

    specialFill(original);

    printRange(original, 0, 20);
    printRange(original, 100, 120);
    printRange(original, 180, 200);
}

private static void specialFill(final int[] original)
{
    for (int j=0; j < original.length / 100; j++)
    {
        for (int i=0; i < 100; i++)
        {
            original[j * 100 + i] = i;
        }
    }
}
```

**Tipp:** Nutze `Arrays.parallelSet()` und einen `IntUnaryOperator`, etwa `idx -> idx % 100`.

**Übung 9:** Berechne inplace die Summe für ein `int`-Array für 1.000 Elemente gefüllt als aufsteigende Folge 1, 2, 3, 4, 5, 6, 7...

```
public static void main(final String[] args)
{
    final int[] original = new int[1_000];
    for (int i=0; i < original.length ; i++)
    {
        original[i] = i + 1;
    }

    sum(original);
    System.out.println(original[999]);
}

private static void sum(final int[] values)
{
    int sum = 0;
    for (int i = 0; i < values.length; i++)
    {
        values[i] += sum;
        sum = values[i];
    }
}
```

Vereinfache die Berechnung mit JDK 8-Bordmitteln. Zur Überprüfung der Funktionalität: Die Summe sollte 500500 betragen.

**Tipp:** Fülle das Array mit `parallelSetAll()` und `IntUnaryOperator` und nutze für die Berechnung `parallelPrefix()` und `IntBinaryOperator`. Dieser kombiniert zwei `int`-Werte zu einem neuen. Versuche es mal mit diesem Lambda `(val1, val2) -> val1 + val2`.

## Part 6: JavaFX 8

Lernziel: Kennenlernen einiger Neuerungen in Java FX 8, im Speziellen des DatePickers.

**Übung 1:** Mache dich mit den Grundlagen einer JavaFX-Applikation durch Lektüre des Buchs „Java 8 – Die Neuerungen“ vertraut. Erweitere das folgende Programm FirstJavaFX um eine Zählvariable, die Button-Klicks zählt und diese in einem `javafx.scene.control.TextField` anzeigt.

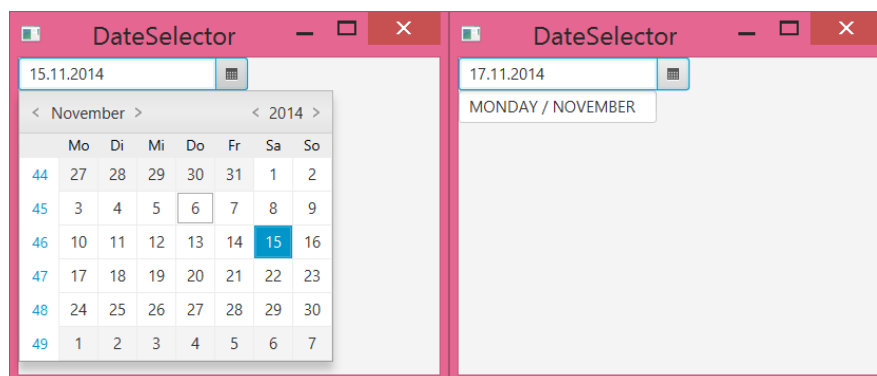
```
import javafx.application.Application;
// ...

public class FirstJavaFX extends Application
{
    @Override
    public void start(final Stage primaryStage) throws Exception
    {
        final FlowPane pane = new FlowPane(5,5);
        pane.setPadding(new Insets(7));
        final Button button = new Button("Press Me");
        pane.getChildren().add(button);

        primaryStage.setScene(new Scene(pane, 350, 50));
        primaryStage.setTitle("FirstJavaFX");
        primaryStage.show();
    }

    public static void main(final String[] args)
    {
        Launch(args);
    }
}
```

**Übung 2:** Schreibe eine Applikation DateSelector zur Datumsauswahl, die dazu einen `javafx.scene.control.DatePicker` nutzt. Dabei soll das aktuelle Datum als Vorauswahl dienen. Das vom Benutzer gewählte Datum soll in einem Textfeld dargestellt werden.



**Tipp:** Setze das neue Date and Time API ein, nutze Lambdas für das Event Handling.