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 gern 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, Übungsaufgaben und Folien

Kursleitung und Kontakt

Michael Inden
Lead Software Architect & Trainer @ Zühlke Engineering AG
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 Default-Methoden) 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 2: Schreibe Functional Interfaces für die folgenden Abbildungen

a) Objekte vom Typ `String` auf `int`: `StringToIntConverter`
b) Objekte vom Typ `String` auf `String`: `StringToStringMapper`

Implementiere diese mit mindestens einem Lambda.

Prüfe die Funktionalität mit folgenden Zeilen:

```
System.out.println("stringToInt('Java') => " +
                   stringToInt.convert("Java"));
System.out.println("stringToString('  Michael  ') => '" +
                   stringToStringTrimmer.convert("  Michael  ") + "'");
System.out.println("stringToUpperCase('Hands On') => " +
                   stringToUpperCaseString.convert("Hands On"));
```

Übung 3: Gegeben sei ein rudimentäres Fragment eines Functional Interfaces als Ergänzung der obigen String-Mapper:

```
public interface Mapper<S, T>
{
       T map(S elem);

       default public List<T> mapAll(final List<S> sourceElements)
       // ...
}
```

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 <<]
```

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 `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) -> …
```

KÜR:

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

```
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? Schaue 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: Sortiere eine Liste von Namen gemäß ihrer natürlichen Ordnung basierend auf `Comparable<String>` und gib diese auf der Konsole aus. Die nachfolgende klassische Variante soll mit JDK 8-Mitteln vereinfacht werden – zudem sollen die externen Iterationen in interne gewandelt werden:

```
final List<String> names = Arrays.asList("Tim", "Peter", "Mike");

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

Collections.sort(names, naturalOrder);
for (final String name : names)
{
	System.out.println(name);
}
```

Tipp: Schreibe einen dazu passenden `Comparator<String>`. Nutze

a) einen Lambda
b) eine Methodenreferenz auf `String::compareTo`

als Eingabe für die Default-Methoden `sort()` aus dem Interface `List<E>`.

Übung 3: Lösche aus einer Liste von Namen alle mit < 4 Zeichen. Wandle dabei eine externe in eine interne Iteration um. Gegeben ist folgende Implementierung mit JDK 7, die mithilfe von JDK-8-Mitteln einfacher gestaltet werden soll:

```
private static List<String> removeIf_External_Iteration(
                                        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("Flo"); names.add("Clemens");
	return names;
}
```

Tipp: Nutze das in Aufgabe 1 erstellte `Predicate<T> isShortWord`.

Übung 4: Modifiziere eine Liste mit Namen: Verändere dort all diejenigen Namen, die mit einem „M“ starten: Aus „`Michael`“ wird dann „ `>>MICHAEL<<`“. Ergänze eine weitere Aktion: Alle Wörter mit „i“ sollen umgedreht werden. 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;
}
```

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

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 modellieret werden.

Übung 1: Geben sei eine Menge von Namen:

```
final String[] namesArray = {"Tim", "Tom", "Andy", "Mike", "Merten"};
```

Es sollen alle mit einem T im Namen ermittelt und in UpperCase gewandelt auf der Konsole ausgegeben werden. Dabei sollen die drei typischen Operationen Create, Intermediate und Terminal nachvollzogen werden.

Die erwartete Ausgabe ist: `TIM, TOM, MERTEN,`

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 3a: Gruppiere die Namen nach dem Anfangsbuchstaben, sodass wir folgendes Ergebnis erhalten: `{A=[Andy], T=[Tim, Tom], M=[Mike, Merten]}`

```
final Map<Character, List<String>> grouped = // …
System.out.println(grouped);
```

Tipp: Nutze `collect()` und die `Collectors.groupingBy()`.

Übung 3b: Teile den Inhalt gemäß der Länge (<= 3 und > 3), sodass wir folgende Ausgabe erhalten: `{false=[Andy, Mike, Merten], true=[Tim, Tom]}`

```
final Map<Boolean, List<String>> partioned = // …
System.out.println(partioned);
```

Tipp: Nutze `collect()` und die `Collectors.partitioningBy()`.

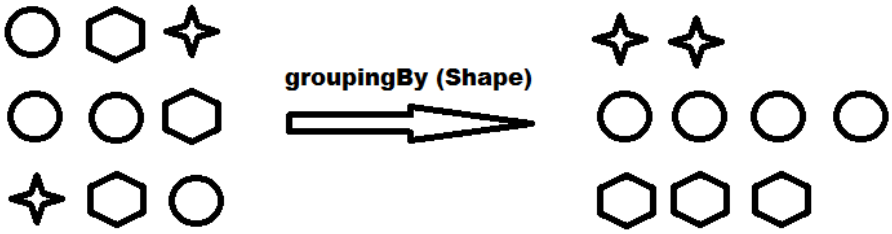
Übung 3c: Gruppiere den Inhalt nach dem Anfangsbuchstaben und summiere dann die Anzahl der Vorkommen, sodass wir folgende Ausgabe erhalten:
`{A=1, T=2, M=2}`

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

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

Übung 4: Finde eine grafische Notation für die Operationen `filter()`, `map()` und `Collectors.groupingBy()` sowie `Collectors.partitioningBy()`.

Tipp:



Übung 5: Gegeben seien die Zahlen von 1 bis 10. Quadriere alle ungeraden Zahlen und ermittle deren Summe.

```
final IntStream ints = IntStream.of(1,2,3,4,5,6,7,8,9,10);
```

Übung 6: 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? Schaue einmal auf die Klasse `IntSummaryStatistics`.

KÜR:

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

```
Stream.of("Andi", "Barbara", "Carsten", "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));
```

Welche Erkenntnisse gewinnst du anhand der Ausgaben? In welcher Reihenfolge sollte man die Operationen ausführen? Tausche einmal die Aufrufe von `filter()` und `map()`.

„Schnüffel“:

Verbessere das Hineinschauen in den Stream durch Aufruf von `peek(Consumer)`.

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 bzw. Geburtstag und heute?

```
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 und 2 Monaten und 14 Tagen wie folgt und schaue auf die Konsolenausgabe (`P1Y2M14D`):

```
final Period period1 = Period.of(1, 2, 14);
```

Übung 2b: Wie lautet die Ausgabe für `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 2d: Addiere die obige Zeitdauer auf den 10.10.2013. Wo landet man?

Übung 3: 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 4: 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 5a: Welcher Wochentag war der Heiligabend 24.12.2013 sowie der erste und letzte Tag im Dezember? Starte mit folgenden Programmzeilen:

```
final LocalDate christmasEve = LocalDate.of(2013, 12, 24);
System.out.println(DayOfWeek.from(christmasEve));
```

Übung 5b: 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 5c: Der wievielte Tag im März war das jeweils? Und der wievielte im Jahr?

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

Hinweis: Hier gibt es noch folgende alternative Lösung mit der Methode `adjustInto()` aus dem Interface `TemporalAdjuster`:

```
*   temporal = thisAdjuster.adjustInto(temporal);
*   temporal = temporal.with(thisAdjuster);
```

Welche Variante ist zu bevorzugen bzw. besser zu verstehen?

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 FirstDayOfQuarterOrig implements TemporalAdjuster
{
    @Override
    public Temporal adjustInto(final 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.println(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 eigene Realisierung des Interface `TemporalAdjuster`, die auf den n-ten Wochentag springt, z. B. den 3 Freitag im Monat:

```
    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)));
    }
```

⇨

```
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 im Interface `Comparator<T>`. Diese Neuerungen werden hier mithilfe einer Übungsaufgabe erforscht.

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()`