Funktionales Sortieren von strukturieren Daten

Einleitung:

Eine Zahlenreiche zu sortieren ist einfach, wenn man die Regeln kennt, wie man vergleichen und damit sortieren muss. zBsp. von der Kleinsten bis zur Grössten.

Doch nach was sortiert man eigentlich eine Reihe von Objekten von Personen?

Java stellt dazu zBsp für die List die Methode *sort* zur Verfügung.
Auch hier gilt, dass für das Sortieren, die ***Regel für den Vergleich der Objekte*** bekannt sein muss.

Und je nach Regel wird dann nach Namen, Alter, Schuhgrösse oder … sortiert.

Das Thema Sortieren und Vergleichen in dieser Übung, dient dazu, einen vertieften Einblick in das Anwenden von *Functional-Interface*, *abgeleitete Klassen*, *anonyme Klassen* und *lambda Expressions* zu erlangen und an praktischen Beispielen anzuwenden.

Ziel

- ⇨ Sie können eine *Collection von Objekten* nach verschiedenen Aspekten sortieren.
- ⇨ Sie können das Interface *Comparator* erklären und anwenden.
- ⇨ Sie können das Interface *Comparator* als *abgeleitete Klasse*, als *anonyme Klasse* und als *lambda Expression* implementieren.
- ⇨ Sie können eine *Comparator chain* erklären und anwenden.
- ⇨ Sie können die Class *Comparable* erklären und wenden.
- ⇨ Sie können Objekte nach Ihrer *Natural-Order* und *reverse* sortieren.
- ⇨ Sie können Attribute von Comparatoren für die Sortierung verwenden.
- ⇨ Sie können erklären, wo und wie beim Sortieren das funktionale Programmieren zur Ausprägung kommt.

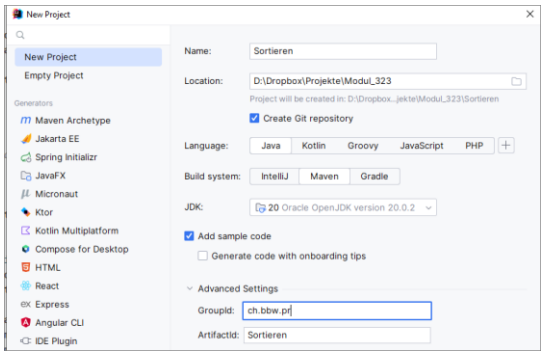
Inhalte

Aufgabe: Vorbereitung der Datenklasse

Um sortieren zu können, brauchen Sie ein Projekt und eine Datenklasse.

Neues Projekt:

Erstellen Sie eine neues auf Maven basiertes Java Projekt:



Klasse Customer übernehmen

Übernehmen Sie die vorbereitet Java Class Customer ***Customer.java*** in Ihr Projekt.

Boiler Plate Code mit Hilfe von lombok oder Record vermeiden

Bei einer Datenklasse kann man mit Hilfe der Library lombok den Standard-Code für *Getter, Setter, Constructoren, toString* … generieren lassen.

In der Klasse Customer wurden dazu diese beiden Teile hinzugefügt.

```
import lombok.Value
…
@Value
```

Fügen Sie im pom.xml die Dependency für lombok hinzu.
Google: *lombok maven repository*

Eine weitere Alternative, die seit Java 14 besteht ist das Java Keyword 'Record'.
Eine Einführung dazu finden Sie hier: https://www.baeldung.com/java-record-keyword

⇨ *Entscheiden Sie sich, ob Sie lombok oder Record verweden.*

Main ergänzen, Daten bereitstellen und ausgeben.

Übernehmen Sie den Code aus *customer.cs* in die *main-Methode*.

Geben Sie die Liste der *customer*s auf die *Console* aus.

Zusatz:

Recherchieren Sie, wie man mit Hilfe von stream nur die ersten 10 Einträge der Liste ausgibt und wenden Sie das zusätzlich an.

Input: Sortieren nach Nach- und Vornamen

In einer ersten Sortierung sortieren Sie die customers nach Nach- und Vornamen.

Hinweis: Sie wenden zunächst keine Stream an, dafür die ***Methode sort der class List***

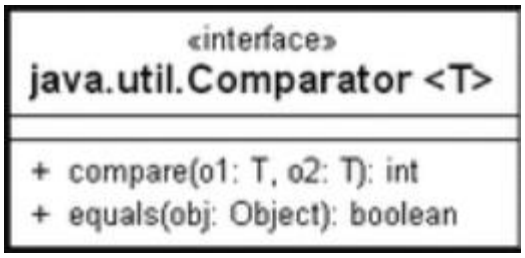
sort verlangt eine Comparator

Wenn Sie *customer.sort* tippen, dann verlangt *sort* als Parameter einen *Comparator*.

Was ist ein Comparator?

```
// first sort, lastname and firstname
System.out.println("lastname, firstname");
customers.sort
custome   sort(Comparator<? super Customer> c)                    void
```

Input: interface java.lang.Comparator<T>



- Comparator ist die Basis-Klasse, um den Vergleich von Objekten einer Klasse zu implementieren.
- Der Comparator ist ein *Functional-Interface.*
- Beim Sortieren kann dann dieser Vergleich für die Reihenfolge der Sortierung benützt werden.
- Für jeden individuellen Vergleich muss eine eigene Ableitung des Comparators erstellt werden.
- Sie müssen für den Vergleich die Methode *int compare(T o1, T o2)* überschreiben
- Weiter gibt es die Methode *equals(Object obj)* für die Prüfung auf Gleichheit.

Die Regel für die Implementierung von *compare(..)* ist:

- Ist *o1* > *o2*, dann wird der return einen Wert grosser 0, als z.B. 1 zurückgeben
- Ist *o1* < *o2*, dann wird der return einen Wert kleiner 0, also z.B. -1 zurückgeben
- Ist *o1* == *o2*, dann wir der return den Wert 0 zurückgeben.

Hinweis aus der Praxis

Die Methode equals muss nicht zwingend implementiert werden.

Sie werden nun Comparatoren als ***eigene Klasse***, als ***anonyme Klasse*** und als ***Lambda Expression*** implementieren.

Und dann werden Sie noch den Schritt zur Implementierung einer ***Comparator-Chain*** machen.

Aufgabe Erste Anwendung eines Comparators für Lastname und Firstname

Für die erste Implementierung verwenden Sie eine eigene Klasse, die von Comparator abgeleitet ist.

CustomerByLastnameFirstname.java

```java
package ch.bbw.pr;

import java.util.Comparator;

/**
 * Customer Comparator
 * @author Peter Rutschmann
 * @version 05.11.2023
 */
1 usage  new *
public class CustomerByLastnameFirstname implements Comparator<Customer> {

    new *
    @Override
    public int compare(Customer o1, Customer o2) {
        int value = o1.getLastname().compareTo(o2.getLastname());
        if (value == 0) value = o1.getFirstname().compareTo(o2.getFirstname());
        return value;
    }
}
```

Angewandt:

```java
// first sort, lastname and firstname
System.out.println("lastname, firstname");
customers.sort(new CustomerByLastnameFirstname());
customers.stream().limit( maxSize: 10).forEach(System.out::println);
```

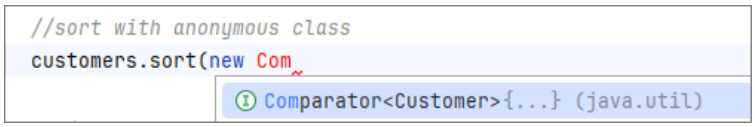
Aufgabe:

- Implementieren Sie das Beispiel

Aufgabe Anwendung eines Comparators als anonyme Klasse

Anstelle einer eigenen Klasse zu schreiben, kann man direkt bei *sort* eine *anonyme Klasse* implementieren.

IntelliJ unterstütz Sie bei der Implementierung. Wenn Sie beginnen zu tippen, dann kann IntelliJ den Rest generieren...



Wird zu

```
customers.sort(new Comparator<Customer>() {
    new *
    @Override
    public int compare(Customer o1, Customer o2) {
        return 0;
    }
});
```

Nun noch den Vergleich ergänzen

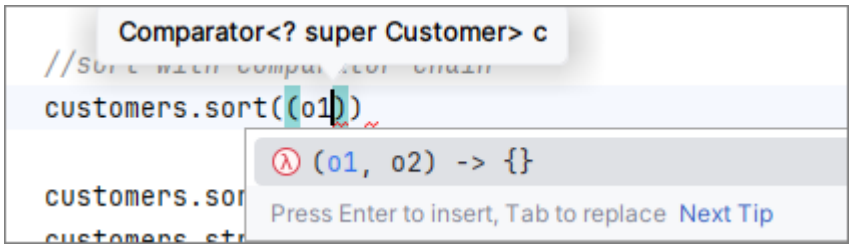
```
customers.sort(new Comparator<Customer>() {
    new *
    @Override
    public int compare(Customer o1, Customer o2) {
        int value = o1.getLastname().compareTo(o2.getLastname());
        if (value == 0) value = o1.getFirstname().compareTo(o2.getFirstname());
        return value;
    }
});
```

IntelliJ schlägt direkt vor, dass man eine lambda Expression anstelle der anonymen Klasse verwenden soll.
Als gleich das nächste Beispiel implementieren

Aufgabe Anwendung eines Comparators als lambda Expression

Anstelle einer anonymen Klasse zu schreiben, kann man direkt bei *sort* eine lambda Expression implementieren.

IntelliJ unterstütz Sie bei der Implementierung. Wenn Sie beginnen zu tippen, dann kann IntelliJ den Rest generieren...



generieren und ergänzen…

```
//sort with lambda expression
customers.sort((o1, o2) -> {
      int value = o1.getLastname().compareTo(o2.getLastname());
      if (value == 0) value = o1.getFirstname().compareTo(o2.getFirstname());
      return value;
});
```

Aufgabe Anwendung einer Comparator-Chain

IntelliJ schlägt als weiter Optimierung vor, dass man die *Lambda Expression* zu einer *Comparator chain* umwandlen kann.

Damit Sie den Unterschied später noch sehen, kopieren Sie zunächst die Sortierung mit einer lambda Expression und wandeln Sie dann die Kopie um in eine Comparator chain.

wird zu

```
//sort with comparator chain
customers.sort(Comparator.comparing(Customer::getLastname).thenComparing(Customer::getFirstname));
```

Bereits die erste Klasse verwendet den Ansatz der funktionalen Programmierung mit Hilfe des functional Interfaces Comparator.

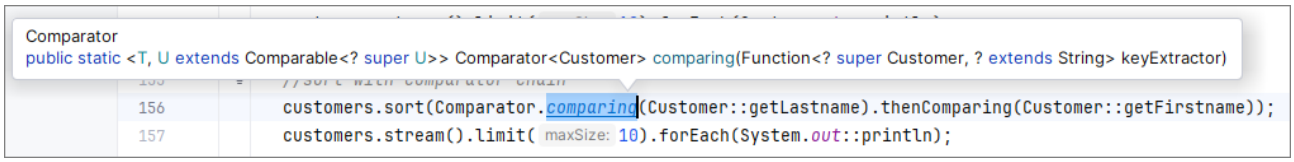
```
public class CustomerByLastnameFirstname implements Comparator<Customer> {
```

Die *Comparator chain* ist sehr kompakt und man kann sehr einfach folgen, was Schritt um Schritt gemacht wird.

Input Wie funktioniert Comparator.comparing(...)?

Implementierung von comparing analysieren

IntelliJ: selektieren Sie *comparing* und rücken Sie die *CTRL-Taste* und *klicken* Sie… so gelangen Sie zur Stelle, wo comparing beschrieben und implementiert ist.



Accepts a function that extracts a `Comparable` sort key from a type `T`, and returns a `Comparator<T>` that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

Params: `keyExtractor` – the function used to extract the `Comparable` sort key

Returns: a comparator that compares by an extracted key

Throws: `NullPointerException` – if the argument is null

API Note: For example, to obtain a `Comparator` that compares `Person` objects by their last name,

```
Comparator<Person> byLastName = Comparator.comparing
(Person::getLastName);
```

Since: 1.8

```
public static <T, U extends Comparable<? super U>> Comparator<T> comparing(
    Function<? super T, ? extends U> keyExtractor)
{
  Objects.requireNonNull(keyExtractor);
  return (Comparator<T> & Serializable)
      (c1, c2) -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

Erklärungen

- Die Implementierung verwendet *Java Generics*<> und ist damit auf verschiedenste Typen anwendbar.
- *comparing* gibt einen *Comparator* zurück.
  Denn sort verlangt ja auch eine Comparator.
- Als keyExtractor wird comparing die Methode übergeben, mir der der zu vergleichende key aus Customer geholt werden kann.
  Deshalb im ersten Schritt getLastname.
- Die beiden keys werden über die Methode compareTo verglichen.

Es gilt:

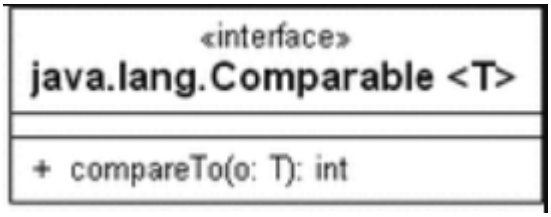
keyExtractor ➔ getLastname ➔ das gibt einen String ➔ Anwenden von compareTo von String

Will man also die Comparator chain anwenden, so müssen die Datentype der verwendeten keys die compareTo Methode implementieren.

Viele Datentypen von Java machen das bereits, da Sie das Interface Comparable implementieren. So auch String.

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
    Constable, ConstantDesc {
```

Input interface java.lang.Comparable<T>



- Dieses Interface verlangt die Implementierung der Methode *compareTo*.
- Eine Datenklasse implementiert das Interface direkt!
  Es werden also *keine extra Klassen von Comparable abgeleitet*.
- In der Methode *compareTo* muss der Vergleich implementiert sein, der der ***natürlichen Ordnung*** (Natural Ordering) der Klasse entspricht.
  Also dem Vergleich, der beim Sortieren die *normal erwartete Sortierung* ergibt.
  Bei einem String geschieht das gemäss der lexikographischen Regel (alphabetisch, Gross vor Klein).

Idee:

Wenn man eine natürliche Ordnung für die Klasse Customer mit Hilfe von Comparable definiert, dann kann man an verschiedensten Stellen im Code, diese verwenden.

Hinweis:

Mit Hilfe von Comparator.naturalOrder kann direkt gemäss der natürlichen Ordnung der Klasse sortiert werden.

```
customers.sort(Comparator.naturalOrder());
```

Aufgabe Natürliche Ordnung für Customer definieren und anwenden.

Ergänzen Sie die Klasse Customer um das *Interface Comparable* und die Implementierung von *compareTo*.

Wenden Sie die Sortierung mit Hilfe von *Comparator.naturalOrder()* an.

```
customers.sort(Comparator.naturalOrder());
```

Input: Reverse Sortierung für Customer anwenden

Und wie geht man vor, wenn man *Customer* nach deren *Size* sortieren will?

Idee

Anwenden einer *Comparator Chain* mit dem key *Customer::Size*

- Superkompakt.

Anmerkung

Hmm es sortiert von *Klein nach Gross*. Kann ich auch von *Gross nach Klein* sortieren?

- Sie können einen eigenen Comparator schreiben.
  - als eigene Klasse
  - als anonyme Klasse
  - als lambda Expression
- Oder Sie verwenden die Methode *reversed* der Klasse Comparator.

```
customers.sort(Comparator.comparing(Customer::getSize).reversed());
```

Erläuterung

```
default Comparator<T> reversed() {
   return Collections.reverseOrder( cmp: this);
}
```

und weiter…

```
public static <T> Comparator<T> reverseOrder( @Nullable Comparator<T> cmp) {
   if (cmp == null) {
      return (Comparator<T>) ReverseComparator.REVERSE_ORDER;
```

```
static final ReverseComparator REVERSE_ORDER
      = new ReverseComparator();
```

```
private static class ReverseComparator
      implements Comparator<Comparable<Object>>, Serializable {

   @java.io.Serial
   private static final long serialVersionUID = 7207038068494060240L;

   static final ReverseComparator REVERSE_ORDER
         = new ReverseComparator();

   public int compare(Comparable<Object> c1, Comparable<Object> c2) { return c2.compareTo(c1); }
```

Im letzten Schritt wird die compareTo Methode der zu vergleichenden Objektklasse mit vertauschten Parametern aufgerufen. 😉

Aufgabe: Weiter Anwendung mit Nachnamen und Geburtsdatum

Sie möchten die Customer nach *Nachnamen* und *Geburtsdatum* sortieren.

- Welche Möglichkeiten gibt es?
- Implementieren Sie zwei unterschiedliche Lösungsansätze.

Aufgabe: Weitere Anwendung mit Nachnamen und Geburtsdatum (umgekehrt)

In der Aufgabe zuvor wird das Geburtsdatum wahrscheinlich von älter nach jünger sortiert.

- Implementieren Sie eine Sortierung, so dass nach Nachnamen und beim Geburtsdatum von jünger nach älter sortiert wird.
- Erklären Sie, wieso diese Lösung nicht zum Ziel führt. [1]

```
customers.sort(Comparator.comparing(Customer::getFirstname).thenComparing(Customer::getBirthdate).reversed());
```

Beispiel des erwarteten Ergebnisses:

```
Customer(lastname=Adrienne, firstname=Mckinney, birthdate=Thu May 23 00:00:00 CET 1974, size=1.84, phone=+41 49 558 95 36)
Customer(lastname=Alden, firstname=Gilmore, birthdate=Sun Dec 28 00:00:00 CET 1980, size=1.63, phone=+41837371621)
Customer(lastname=Alden, firstname=Byrd, birthdate=Wed Jan 01 00:00:00 CET 1947, size=1.72, phone=076 587 72 40)
Customer(lastname=Alea, firstname=Schultz, birthdate=Tue Jan 17 00:00:00 CET 1995, size=1.74, phone=017 665 45 53)
```

Knacknuss

- Finden Sie eine Lösung *ohne* und eine Lösung *mit* einer *Comparator-Chain*.

[1] Reversed kehrt die ganze Collection um und nicht nur die Sortierung von birthdate.

Input: Wiederverwendung an verschiedenen Codestellen

Die natürliche Ordnung lässt sich an beliebigen vielen Stellen im Code immer wieder aufrufen.

```
customers.sort(Comparator.naturalOrder());
```

Diese Implementierung nach Lastname und Birthdate ist direkt beim Aufruf implementiert.

```
customers.sort((o1, o2) -> {
    int value = o1.getLastname().compareTo(o2.getLastname());
    if (value == 0){
        //reversed order
        value = o2.getBirthdate().compareTo(o1.getBirthdate());
    }
    return value;
});
```

Will man diese Sortierung an einer anderen Stelle nochmals verwenden, dann sollte man es dort nicht nochmals implementieren müssen, denn das wäre sehr fehleranfällig.

Idee:

- Eine mögliche Lösung wäre eine abgeleitete Comparator-Klasse zu definieren und jeweils zu verwenden.
  Diese zusätzliche Klasse wirkt neben den anderen Möglichkeiten aber sehr schwerfällig.

Variante:

Da sort als Parameter eine Comparator verlangt

```
default void sort(Comparator<? super E> c) {
```

Kann man eine *anonyme Klasse*, *lambda Expressions oder Comparator-Chain* als Attribute der Klasse Customer definieren und dann bei sort verwenden.

Dieses Attribut ist sinnvollerweise static.

Das Beispiel ist auf der nächsten Seite:

Beispiel mit anonymer Klasse:

```
public class Customer implements Comparable<Customer>{
    private String lastname;
    private String firstname;
    private Date birthdate;
    private double size;
    private String phone;

    new *
    public static Comparator<Customer> comparatorACLastNameBirthdate = new Comparator<Customer>() {
        new *
        @Override
        public int compare(Customer o1, Customer o2) {
            int value = o1.getLastname().compareTo(o2.getLastname());
            if (value == 0){
                //reversed order
                value = o2.getBirthdate().compareTo(o1.getBirthdate());
            }
            return value;
        }
    };
```

Anwendung mit sort

```
//sort with static attribut of Customer
System.out.println("sort with static attribut of Customer");
customers.sort(Customer.comparatorACLastNameBirthdate);
customers.stream().limit( maxSize: 10).forEach(System.out::println);
```

Aufgabe: Anwenden von Comparator Attributen in Customer

- Implementieren Sie das gezeigte Beispiel mit der *anonymen Klasse* und dem Attribut in der Klasse Customer.
- Implementieren Sie eine Variante mit einer *lambda Expression* und einem Attribut der Klasse Customer.
- Implementieren Sie eine Variante mit einer *Comparator-Chain* und einem Attribut der Klasse Customer

Vertiefungsaufgabe zum Sortieren von Listen

Sie haben sich zuvor mit verschiedensten Aspekten zum Sortieren von Listen vertraut gemacht.

Aufgabe:

- Vertiefen Sie Ihr Wissen und Können anhand von diesem Artikel
  Probieren Sie die gezeigten Beispiele aus.

  https://dzone.com/articles/java-8-comparator-how-to-sort-a-list

Zusammenfassend zum obigen Artikel:

Unterschiedliche Beispiele von Vergleichen

- *String.CASE_INSENSITIVE_ORDER)*
- *Comparator.naturalOrder()*
- *Comparator.comparing(<Getter>)*
- *Custom Comparator*

Die Beispiele zeigen auch, das Anwenden des *Custom Comparator.*

- Als *anonyme Klasse*
- Als *lambda* Expression

Und nicht gezeigt, aber möglich, sind eigene *von Comparator abgeleitete Klassen*.

Funktionales Programmieren und Sortieren

- Das Funktionale Interface erlaubt es für einen Comparator eine lambda Expressions zu verwenden.
  Dies ist eine Art in Java funktional zum Implementieren.
- Die Comparator-Chain zeigt den Ansatz des funktionalen Programmierens, wo das Ergebnis im Vordergrund steht, nicht jedoch die eigentliche Implementierung.
  → Sie kümmern sich nicht um die Implementierung des eigentlichen Sortier- oder Vergleichsvorganges.
  Sie verketten Schritte so zusammen, dass Sie das erwartete Ergebnis erhalten.
- Ein Grundsatz des funktionalen Programmierens ist allerdings nicht erfüllt.
  Bei allen Anwendungen wird die Collection selbst verändert.
  Das verletzt das Prinzip der Immutability (Unveränderlichkeit).
  ➔ Wenn man nun einen Stream verwendet, dann kann dieses Prinzip eingehalten werden.

Funktionales Sortieren mit Streams mit einem selbst definierten Comparator

```
List <Customer> sortedCustomer =
      customers.stream().sorted(Customer.comparatorLaLastNameBirthdate).collect(Collectors.toList());
customers.stream().limit( maxSize: 10).forEach(System.out::println);
```

Die Original-List customers wird hier nicht verändert.