The Comparable Interface

Learning Objectives

- You're aware of Java's Comparable class and now how to implement it in your own classes
- You know how to use Java's tools for sorting lists and stream elements.
- You know how to sort list elements using multiple criteria (e.g., you know how to sort a person based on name and age).

In the previous section, we looked at interfaces in more general terms - let's now familiarize ourselves with one of Java's ready-made interfaces. The Comparable interface defines the `compareTo` method used to compare objects. If a class implements the Comparable interface, objects created from that class can be sorted using Java's sorting algorithms.

The `compareTo` method required by the Comparable interface receives as its parameter the object to which the "this" object is compared. If the "this" object comes before the object received as a parameter in terms of sorting order, the method should return a negative number. If, on the other hand, the "this" object comes after the object received as a parameter, the method should return a positive number. Otherwise, 0 is returned. The sorting resulting from the `compareTo` method is called *natural ordering*.

Let's take a look at this with the help of a Member class that represents a child or youth who belongs to a club. Each club member has a name and height. The club members should go to eat in order of height, so the Member class should implement the `Comparable` interface. The Comparable interface takes as its type parameter the class that is the subject of the comparison. We'll use the same `Member` class as the typ parameter.

```java
public class Member implements Comparable<Member> {
    private String name;
    private int height;

    public Member(String name, int height) {
        this.name = name;
        this.height = height;
    }

    public String getName() {
        return this.name;
    }

    public int getHeight() {
        return this.height;
    }

    @Override
    public String toString() {
        return this.getName() + " (" + this.getHeight() + ")";
    }

    @Override
    public int compareTo(Member member) {
        if (this.height == member.getHeight()) {
            return 0;
        } else if (this.height > member.getHeight()) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

The `compareTo` method required by the interface returns an integer that informs us of the order of comparison.

As returning a negative number from `compareTo()` is enough if the `this` object is smaller than the parameter object, and returning zero is sufficient when the lengths are the same, the `compareTo` method described above can also be implemented as follows.

```java
@Override
public int compareTo(Member member) {
    return this.height - member.getHeight();
}
```

Since the Member class implements the Comparable interface, it is possible to sort the list by using the `sorted` method. In fact, objects of any class that implement the `Comparable` interface can be sorted using the `sorted` method. Be aware, however, that a stream does not sort the original list - *only the items in the stream are sorted*.

If a programmer wants to organize the original list, the `sort` method of the `Collections` class should be used. This, of course, assumes that the objects on the list implement the `Comparable` interface.

Sorting club members is straightforward now.

```
List<Member> member = new ArrayList<>();
member.add(new Member("mikael", 182));
member.add(new Member("matti", 187));
member.add(new Member("ada", 184));

member.stream().forEach(m -> System.out.println(m));
System.out.println();
// sorting the stream that is to be printed using the sorted method
member.stream().sorted().forEach(m -> System.out.println(m));
member.stream().forEach(m -> System.out.println(m));
// sorting a list with the sort-method of the Collections class
Collections.sort(member);
member.stream().forEach(m -> System.out.println(m));
```

Sample output

```
mikael (182)
matti (187)
ada (184)

mikael (182)
ada (184)
matti (187)

mikael (182)
matti (187)
ada (184)
```

mikael (182)
ada (184)
matti (187)

Quiz:

Ordering by the number of pages

Points:
1/1

Which piece of code orders the books by the number of pages?

```
public class Book implements Comparable<Book> {

    public int numberOfPages;

    public Book(int numberOfPages) {
        this.numberOfPages = numberOfPages;
    }

    @Override
    public int compareTo(Book book) {
        return /* code comes here */
    }
}
```

Select the correct answer

[✓ this.numberOfPages - book.numberOfPages]

book.numberOfPages - this.numberOfPages

this.numberOfPages < book.numberOfPages

```
book.numberOfPages +
this.numberOfPages
```

The answer is correct

Answered Tries remaining: 2

Programming exercise:
Wage order

Points
1/1

You are provided with the class Human. A human has a name and wage information. Implement the interface `Comparable` in a way, such that the overridden `compareTo` method sorts the humans according to wage from largest to smallest salary.

Exercise submission instructions

How to see the solution

Programming exercise:
Students on alphabetical order

Points
1/1

The exercise template includes the class `Student`, which has a name. Implement the `Comparable` interface in the Student class in a way, such that the `compareTo` method sorts the students in alphabetical order based on their names.

The name of the `Student` is a String, which implements `Comparable` itself. You may use its `compareTo` method when implementing the method for the `Student` class. Note that `String.compareTo()` also treats letters according to their size, while the `compareToIgnoreCase` method of the same class ignores the capitalization completely. You may use either of these methods in the exercise.

Exercise submission instructions

How to see the solution

Implementing Multiple Interfaces

A class can implement multiple interfaces. Multiple interfaces are implemented by separating the implemented interfaces with commas (`public class ... implements *FirstInterface*, *SecondInterface* ...`). Implementing multiple interfaces requires us to implement all of the methods for which implementations are required by the interfaces.

Say we have the following `Identifiable` interface.

```
public interface Identifiable {
    String getId();
}
```

We want to create a Person who is both identifiable and sortable. This can be achieved by implementing both of the interfaces. An example is provided below.

```
public class Person implements Identifiable, Comparable<Person> {
    private String name;
    private String socialSecurityNumber;

    public Person(String name, String socialSecurityNumber) {
        this.name = name;
```

```java
            this.socialSecurityNumber = socialSecurityNumber;
        }

        public String getName() {
            return this.name;
        }

        public String getSocialSecurityNumber() {
            return this.socialSecurityNumber;
        }

        @Override
        public String getId() {
            return getSocialSecurityNumber();
        }

        @Override
        public int compareTo(Person another) {
            return this.getId().compareTo(another.getId());
        }
    }
```

Sorting Method as a Lambda Expression

Let's assume that we have the following Person class for use.

```java
public class Person {

    private String name;
    private int birthYear;

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

    public String getName() {
        return name;
    }

    public int getBirthYear() {
        return birthYear;
    }
}
```

And person objects on a list.

```
ArrayList<Person> persons = new ArrayList<>();
persons.add(new Person("Ada Lovelace", 1815));
persons.add(new Person("Irma Wyman", 1928));
persons.add(new Person("Grace Hopper", 1906));
persons.add(new Person("Mary Coombs", 1929));
```

We want to sort the list without having to implement the `Comparable` interface.

Both the `sort` method of the `Collections` class and the stream's `sorted` method accept a lambda expression as a parameter that defines the sorting criteria. More specifically, both methods can be provided with an object that implements the Comparator interface, which defines the desired order - the lambda expression is used to create this object.

```
ArrayList<Person> persons = new ArrayList<>();
persons.add(new Person("Ada Lovelace", 1815));
persons.add(new Person("Irma Wyman", 1928));
persons.add(new Person("Grace Hopper", 1906));
persons.add(new Person("Mary Coombs", 1929));

persons.stream().sorted((p1, p2) -> {
    return p1.getBirthYear() - p2.getBirthYear();
}).forEach(p -> System.out.println(p.getName()));

System.out.println();

persons.stream().forEach(p -> System.out.println(p.getName()));

System.out.println();

Collections.sort(persons, (p1, p2) -> p1.getBirthYear() - p2.getBirthYear());

persons.stream().forEach(p -> System.out.println(p.getName()));
```

Sample output

```
Ada Lovelace
Grace Hopper
Irma Wyman
Mary Coombs
```

```
Ada Lovelace
Irma Wyman
Grace Hopper
Mary Coombs

Ada Lovelace
Grace Hopper
Irma Wyman
Mary Coombs
```

When comparing strings, we can use the `compareTo` method provided by the String class. The method returns an integer that describes the order of both the string given to it as a parameter and the string that's calling it.

```
ArrayList<Person> persons = new ArrayList<>();
persons.add(new Person("Ada Lovelace", 1815));
persons.add(new Person("Irma Wyman", 1928));
persons.add(new Person("Grace Hopper", 1906));
persons.add(new Person("Mary Coombs", 1929));

persons.stream().sorted((p1, p2) -> {
    return p1.getName().compareTo(p2.getName());
}).forEach(p -> System.out.println(p.getName()));
```

Sample output

```
Ada Lovelace
Grace Hopper
Irma Wyman
Mary Coombs
```

Programming exercise:

Literacy comparison (2 parts)

Points
2/2

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on

NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

This exercise uses open data about literacy levels, provided by UNESCO. The data includes statistics of estimated or reported levels of literacy for various countries for the past two years. File `literacy.csv`, included with the exercise template, includes the literacy estimates for women and men over 15 years of age. Each line in the file `literacy.csv` is as follows: "theme, age group, gender, country, year, literacy percent. Below are the first five lines as an example.

```
Adult literacy rate, population 15+ years, female (%),United Republic of Tanz
Adult literacy rate, population 15+ years, female (%),Zimbabwe,2015,85.28513
Adult literacy rate, population 15+ years, male (%),Honduras,2014,87.39595
Adult literacy rate, population 15+ years, male (%),Honduras,2015,88.32135
Adult literacy rate, population 15+ years, male (%),Angola,2014,82.15105
```

Create a program that first reads the file `literacy.csv` and then prints the contents from the lowest to the highest ranked on the literacy rate. The output must be exactly as in the following example. The example shows the first five of the expected rows.

Sample output

```
Niger (2015), female, 11.01572
Mali (2015), female, 22.19578
Guinea (2015), female, 22.87104
Afghanistan (2015), female, 23.87385
Central African Republic (2015), female, 24.35549
```

This exercise is worth two points.

Tip! Here's how to split a string into an array by each comma.

```java
String string = "Adult literacy rate, population 15+ years, female (%),Zimb
String[] pieces = string.split(",");
// now pieces[0] equals "Adult literacy rate"
// pieces[1] equals " population 15+ years"
// etc.
// to remove whitespace, use the trim() method:
pieces[1] = pieces[1].trim();
```

Exercise submission instructions

How to see the solution

Sorting By Multiple Criteria

We sometimes want to sort items based on a number of things. Let's look at an example in which films are listed in order of their name and year of release. We'll make use of Java's Comparator class here, which offers us the functionality required for sorting. Let's assume that we have the class `Film` at our disposal.

```
public class Film {
    private String name;
    private int releaseYear;

    public Film(String name, int releaseYear) {
        this.name = name;
        this.releaseYear = releaseYear;
    }

    public String getName() {
        return this.name;
    }

    public int getReleaseYear() {
        return this.releaseYear;
    }

    public String toString() {
        return this.name + " (" + this.releaseYear + ")";
    }
}
```

The `Comparator` class provides two essential methods for sorting: `comparing` and `thenComparing`. The `comparing` method is passed the value to be compared first, and the `thenComparing` method is the next value to be compared. The `thenComparing` method can be used many times by chaining methods, which allows virtually unlimited values to be used for comparison.

When we sort objects, the `comparing` and `thenComparing` methods are given a reference to the object's type - the method is called in order and the values returned by the method are compared. The method reference is given as `Class::method`. In the example below, we print movies by year and title in order.

```
List<Film> films = new ArrayList<>();
films.add(new Film("A", 2000));
films.add(new Film("B", 1999));
films.add(new Film("C", 2001));
films.add(new Film("D", 2000));

for (Film e: films) {
    System.out.println(e);
}

Comparator<Film> comparator = Comparator
              .comparing(Film::getReleaseYear)
              .thenComparing(Film::getName);

Collections.sort(films, comparator);

for (Film e: films) {
    System.out.println(e);
}
```

Sample output

```
A (2000)
B (1999)
C (2001)
D (2000)

B (1999)
A (2000)
```

D (2000)
C (2001)

Programming exercise:

Literature (3 parts)

Points
3/3

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

Write a program that reads user input for books and their age recommendations.

The program asks for new books until the user gives an empty String (only presses enter). After this, the program will print the number of books, their names, and their recommended ages.

Part 1: Reading and printing the books

Implement the reading and printing the books first, the ordering of them doesn't matter yet.

Sample output

```
Input the name of the book, empty stops: The Ringing Lullaby Book
Input the age recommendation: 0

Input the name of the book, empty stops: The Exiting Transpotation Vehicles
Input the age recommendation: 0

Input the name of the book, empty stops: The Snowy Forest Calls
Input the age recommendation: 12

Input the name of the book, empty stops: Litmanen 10
Input the age recommendation: 10

Input the name of the book, empty stops:
```

```
4 books in total.

Books:
The Ringing Lullaby Book (recommended for 0 year-olds or older)
The Exiting Transpotation Vehicles (recommended for 0 year-olds
or older)
The Snowy Forest Calls (recommended for 12 year-olds or older)
Litmanen 10 (recommended for 10 year-olds or older)
```

Part 2: Ordering books based on their age recommendation

Expand your program so, that the books are sorted based on their age recommendations when they are printed. If two (or more) books share the same age recommendations the order between them does not matter.

Sample output

```
Input the name of the book, empty stops: The Ringing Lullaby
Book
Input the age recommendation: 0

Input the name of the book, empty stops: The Exiting
Transpotation Vehicles
Input the age recommendation: 0

Input the name of the book, empty stops: The Snowy Forest Calls
Input the age recommendation: 12

Input the name of the book, empty stops: Litmanen 10
Input the age recommendation: 10

Input the name of the book, empty stops:

4 books in total.

Books:
The Ringing Lullaby Book (recommended for 0 year-olds or older)
The Exiting Transpotation Vehicles (recommended for 0 year-olds
or older)
```

```
Litmanen 10 (recommended for 10 year-olds or older)
The Snowy Forest Calls (recommended for 12 year-olds or older)
```

Part 3: Ordering books based on their age recommendation and name

Expand your program, so that it sorts the books with the same age recommendation based on their name alphabetically.

Sample output

```
Input the name of the book, empty stops: The Ringing Lullaby Book
Input the age recommendation: 0

Input the name of the book, empty stops: The Exiting Transpotation Vehicles
Input the age recommendation: 0

Input the name of the book, empty stops: The Snowy Forest Calls
Input the age recommendation: 12

Input the name of the book, empty stops: Litmanen 10
Input the age recommendation: 10

Input the name of the book, empty stops:

4 books in total.

Books:
The Exiting Transpotation Vehicles (recommended for 0 year-olds or older)
The Ringing Lullaby Book (recommended for 0 year-olds or older)
Litmanen 10 (recommended for 10 year-olds or older)
The Snowy Forest Calls (recommended for 12 year-olds or older)
```

Exercise submission instructions

How to see the solution

You have reached the end of this section! Continue to the next section:

→ 3. Other useful techniques

Remember to check your points from the ball on the bottom-right corner of the material!

In this part: