Learning goals

Before the next day, you should have achieved the following learning goals:

- To work with the `Arrays` class.
- To understand the flexibility of the new lambda forms available in Java 8.
- To write simple lambda expressions using the Java 8 syntax.
- To use *method references*
- To gain experience using `Predicate`.

You should be able to finish most of non-star exercises in the lab session. Remember that star exercises are more difficult. Do not attempt star-exercises unless the other exercises are clear to you.

Preamble

The `Arrays` class provides a number of static utility methods for manipulating arrays. For example, to print out an array, consider using `Arrays.asList`. The point of this is that if you just print an array directly, you do not see anything useful (just the type and memory address), but if you print a `List`, it shows the individual elements separated by commas (it is simpler than creating a loop to traverse the array and print out the elements).

The following exercises presume that you have a main class containing an array which you then pass to the `Arrays.sort` method. For example, initially the class might look something like the following:

```
import java.util.Arrays;

public class Outline {
  public static void main(String... args) { // varargs alternative to String[]
    Integer[] intArray = {1,7,3,4,8,2};
    System.out.println(Arrays.asList(intArray));

    // Arrays.sort(intArray,.......)
  }
}
```

The exercises

All of the exercises should be answered using Java 8 *lambda expressions* unless specified otherwise.

1. Create an array containing some `String`s. Sort the array by

   - length (i.e., shortest to longest)
   - reverse length (i.e., longest to shortest)
   - first character
   - Strings that contain `"e"` first, everything else second.

   Remember that the `compare` method of Comparator should return a negative number if the first entry is *less* than the second, a positive number if the first entry is *greater* than the second, and 0 if they are the same. See the JavaDoc API for details.

2. For the last sorting example (strings with `"e"` first), move the logic that computes the number to a separate `static` method. For example,

   ```
   StringUtils.eChecker(s1, s2)
   ```

   will return

   - `-1` if `s1` is *less* (i.e., it contains `"e"` but `s2` doesn't),
   - `1` if s1 is *greater*, and
   - 0 otherwise.

   Now, rewrite the final lambda sorting example, but use a method reference in place of an explicit lambda.

3. Create a class with a `static` method called `betterString`. This method should take two `String`s and a lambda as its arguments. This lambda states whether the first of the two strings is *better*.

   The method should return the *better* string; i.e., if the lambda returns `true` the method should return the first string, otherwise it should return the second string.

   For the lambda, define an interface called `TwoStringPredicate` with a method that takes two `String`s and returns `true` if the first is *better* than the second, `false` otherwise.

   Here are two examples:

   - returns whichever of `test1` and `test2` is longer,

     ```
     StringUtils.betterString(test1, test2, (s1, s2)
                                  -> s1.length() > s2.length())
     ```

   - always returns `test1`,

     ```
     StringUtils.betterString(test1, test2, (s1, s2) -> true)
     ```

4. Use generics to replace `betterString` with `betterEntry` and `TwoStringPredicate` with `TwoElementPredicate`. Make sure your previous examples still work when you only change `betterString` to `betterElement`.

5. (*) Create a `static` method called `allMatches`. It should take a `List` of `String`s and a `Predicate<String>`, and return a new `List` of all the values that passed the test. Test it with several examples. For example:

```
List<String> shortWords = StringUtils.allMatches(words,
                                    s -> s.length() < 4);
List<String> wordsWithB = StringUtils.allMatches(words,
                                    s -> s.contains("b"));
List<String> evenLengthWords = StringUtils.allMatches(words,
                                    s -> (s.length() % 2) == 0);
```

6. (**) Rewrite `allMatches` so it works on any `List` and associated `Predicate`, not just on `String`s. Verify that your examples from the previous question still work.

7. (*) Create a `static` method called `transformedList`. It should take a `List` of `String`s and a `Function<String,String>` and return a new `List` that contains the results of applying the function to each element of the original list. For example:

```
List<String> excitingWords =
            StringUtils.transformedList(words, s -> s + "!");
List<String> eyeWords =
            StringUtils.transformedList(words, s -> s.replace("i", "eye"));
List<String> upperCaseWords =
            StringUtils.transformedList(words, String::toUpperCase);
```

8. (**) Rewrite `transformedList` so it works with generic types. Verify that your examples from the previous question still work.