This lab introduces programming with functional interfaces in Java. It is due in class Monday, May 4 but earlier handins are welcome and encouraged.

While you may discuss ideas and strategies for problems with other students, you should always make the first attempt on a problem yourself and **you must write up your own solutions in your own words**. *You may not collaboratively write solutions or copy a solution that one person in the group writes up. You also may not look for, copy, or use solutions from other sources, including from generative AI like ChatGPT, even if you make changes. There's no such thing as using someone else's solution for a problem "as an example" for writing your own.*

Preliminaries

This lab involves Java programming. It is recommended that you use Eclipse and that you configure Eclipse as described in the programming with sets lab. (The Eclipse configuration only needs to be done once, so if you did it as part of the previous lab, you don't need to do it again.)

- Create a new Eclipse project named `functions`. Make sure the "Create module-info.java file" box is *not* checked before clicking Finish in the dialog box.
- Import the files from `/classes/cs229/functions` into your project. Make sure they go into the default project under `src` rather than into a package or the top level of the project directory.

Handin

- Make sure you have replaced the `<your name here>` notations in the `@author` tag near the beginning of the files you edited.
- Make sure you have autoformatted all of the files you edited.
- Hand in your work by copying the entire project directory `~/cs229/workspace/functions` to your handin directory `/classes/cs229/handin/`*`username`* (where *username* is replaced by your username).

Log Analyzer

In many real systems, programs produce *log files* that record events such as informational messages, warnings, and errors. System administrators and developers often need tools that can search, filter, sort, and modify log messages.

Functional Interfaces

In Java, an *interface* defines a type by listing the headers for the methods that type will support. A *functional interface* is an interface consisting of a single abstract *functional method.*

For example:

- `Predicate<T>` represents a function that tests whether something is true or false — the functional method is `test`, which takes an object of type `T` and returns `true` or `false`.
- `Comparator<T>` represents a function that compares two objects to determine ordering — the functional method is `compare`, which takes two objects a, b of type `T` and returns a negative number if a comes before b, a positive number if a comes after b, and 0 if a is equivalent to b in the ordering.
- `TextTransform` represents a function that modifies text — the functional method is `transform`, which takes a `String` and returns the transformed `String`.

`TextTransform` is part of the provided code — take a look at it. `Predicate` and `Comparator` are part of the standard Java packages. Look them up by searching the Java 21 API at `https://docs.oracle.com/en/java/javase/21/docs/api/index.html` for `java.util.function.Predicate` and `java.util.Comparator`.

Lambda Expressions

A **lambda expression** is a compact way to write a small function directly in your code. Lambda expressions are commonly used when a method expects a functional interface.

The general form of a lambda expression is:

```
(parameters) -> expression
```

or, if the body has multiple statements:

```
(parameters) -> {
  statements
}
```

When there is one parameter, the parentheses may be omitted:

```
x -> x * x
```

This represents a function that takes a value `x` and returns `x * x`.

Equivalent traditional form:

```
int square(int x) {
  return x * x;
}
```

When there are two or more parameters, parentheses are required:

```
(a, b) -> a + b
```

This represents a function that takes two inputs and returns their sum.

Equivalent traditional form:

```
int add(int a, int b) {
  return a + b;
}
```

Exercises

For this lab you will implement a small log analysis tool as described below. This program demonstrates how *functional interfaces* allow behavior encapsulated in functions to be passed as values to general-purpose methods.

Provided Code

The provided code includes the following components:

- **LogEntry** – represents a single log entry containing a timestamp, a severity level (`DEBUG`, `INFO`, `WARN`, or `ERROR`), and a message
- **Severity** – an enum representing the severity level of a log entry
- **TextTransform** – represents a function that takes a string and returns a modified string
- **LogAnalyzer** – a utility class containing methods that operate on lists of log entries
- **LogAnalyzerMain** – a main program

Only skeletons have been provided for `LogAnalyzer` and `LogAnalyzerMain` — you will be completing those implementations.

Task 1: Implement `filter`

Complete the following method in `LogAnalyzer`:

```
public static List<LogEntry> filter(List<LogEntry> entries,
                                    Predicate<LogEntry> condition)
```

This method should:

- create a new list (`ArrayList`)
- examine each entry in the input list `entries`
- add the entry to the new list if the predicate returns `true`
- return the new list

You should use a loop to process the list — the `foreach` loop syntax is convenient here:

```
  for ( LogEntry item : entries ) { ... }
```

Task 2: Implement `sort`

Complete the following method in `LogAnalyzer`:

```
public static void sort(List<LogEntry> entries,
                        Comparator<LogEntry> comparator)
```

This method should sort the list using the provided comparator. Use the standard library method `Collections.sort(...)`. (Look the `Collections` class up in the Java 21 API by googling `java 21 Collections`.)

Task 3: Implement `transform`

Complete the following method in `LogAnalyzer`:

```
public static void transform(List<LogEntry> entries,
                             TextTransform transformer)
```

This method should modify the messages in `entries` themselves — use `LogEntry`'s `setMessage` method to replace each entry's message with the transformed version.

Task 4: Complete the Main Program

The provided main program `LogAnalyzerMain` creates a list of several sample `LogEntry` objects.

Complete the main program so that it also performs the following steps:

- Use `LogAnalyzer.filter` to select only the entries with severity `ERROR`.
- Use `LogAnalyzer.sort` to order those entries by timestamp (earliest first).

- Use `LogAnalyzer.transform` to convert each message to uppercase.
- Print the resulting log entries.

The output should contain only the error messages, ordered by time, with messages printed in all capital letters.

Use lambda expressions for the behavior passed to the `LogAnalyzer` methods.

To print the log entries in the last step, you can utilize `List`'s `forEach()` method instead of writing a loop. Pass a lambda expression or the *method reference* `System.out::println` as the parameter to `forEach`. (You could have used `List.forEach()` instead of a `foreach` loop for the `filter` method as well, but that operation is longer and the loop is a bit cleaner syntax.)