SWEN20003
Object Oriented Software Development
Workshop 11

Eleanor McMurtry

Semester 2, 2020

Workshop

This week, we are reviewing **event-driven programming** and a few miscellaneous advanced Java topics.

1. Event-driven programming is a common paradigm used to handle graphical user interfaces, and also used for network communication.
2. An **event** triggers a **callback**—a piece of code that is designed to handle the event. This is analogous to the **notify** method in the **Observer** pattern.
3. An **enum** is a class that has a finite set of possible instances.
4. A **functional interface** has only one method. It represents a C-style **function pointer**.
5. A **lambda function** allows us to create a new implementation of a functional interface in one line. The Java compiler automatically decides which functional interface a lambda function must match.
6. A **stream** allows a clean and simple way to manipulate datasets such as lists. It simplifies code that would otherwise require several loops.

Questions

1. On Canvas, you will find a sample project `week12-events`. It is a skeleton project representing a graphical user interface. There are three possible events:

   - `OnClick`: triggered when a control has been clicked on
   - `OnSubmit`: triggered when the Enter key has been pressed with a control focused (i.e. previously clicked on)
   - `OnInput`: triggered when the user types a letter or presses the spacebar with a control focused

   Callbacks can be added to controls using the `addEventHandler` method. Callbacks are of the functional interface type `Consumer<String>`, meaning they take a single string argument and have no return value.

   Your task is to finish the `TextField` and `SubmitButton` classes using the event-driven paradigm. You may modify the classes however you wish to do this. You must implement the following:

   - when `TextField` receives input, it should concatenate the letter to its string value (which starts empty). It should always display its current value using the `Font` class.
   - when `TextField` receives a submit event, it should print `Input:` followed by its value to the console, and the program should exit.
   - when `SubmitButton` is clicked, it should perform the `TextField` submit action.

2. Create an enumerated type to represent cardinal directions. It should contain values for north, south, east, west, and the four directions between each of these. Add a method `toDegrees()` that returns the bearing of the direction in degrees.

   For example, `NORTH.toDegrees()` should return `0` and `NORTH.toDegrees()` should return `180`.

3. Write Java code using `for` loops to implement the same functionality as the following stream pipeline.

```
List<String> list = Arrays.asList("Avengers: End Game", "Game of Thrones",
    "Jon Snow", "Arya", "SWEN20003", "Suits");
long count = list.stream()
    .filter(IMDB::isTVShow)
    .map(name -> IMDB.getShow(name))
    .filter(show -> show.getRatings() > 4.0)
    .count();
```

What is the purpose of this pipeline?

Extras

This section is intended to give you some extra problems at a more challenging level to help you revise generics and other more advanced concepts.

1. Make the below `CycleList<T>` class implement the interface `Collection<T>`. **Note:** you may need to change some of the method signatures.

```
class CycleList<T> {
    private final List<T> items = new ArrayList<>();
    private int iterator = 0;

    public T next() {
        T item = items.get(iterator++);
        iterator = iterator % items.size();
        return item;
    }

    public void add(T value) {
        items.add(value);
    }

    public boolean contains(T value) {
        return items.contains(value);
    }

    public void addAll(Collection<T> collection) {
        items.addAll(collection);
    }

    public void remove(T item) {
        items.remove(item);
    }
}
```

2. Write a class `SortedCycleList<T extends Comparable<T>>` that acts like a `CycleList<T>`, except the `next()` method cycles through items in sorted order.

3. (a) Write a method
   `static <T> T findFirst(Predicate<T> pred, Collection<T> collection)` that returns the first item in `collection` satisfying `pred`, or `null` if no such item exists.
   (b) Can you find a standard library method that does this for `Streams`?
   (c) The method in question returns `Optional<T>`, a class that represents a value of type `T` that may be *missing* (hence the name "optional"). Experiment with the `ifPresent` and `map` methods of this class. How does an `Optional` value compare to using `null` to represent an unsuccessful result?
   (d) Think about why both `Stream<T>` and `Optional<T>` define the method `map`. Is there a common structure shared between these classes?