SWEN20003
Object Oriented Software Development
Workshop 11 (Solutions)

Eleanor McMurtry

Semester 2, 2020

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.

   Solution:

```
public class TextField extends Eventable {
    private static final Image image = new Image("res/text.png");
    private static final Font font = new Font("res/VeraMono.ttf", 36);

    private String contents = "";

    public void onSubmit() {
        System.out.println("Input: " + contents);
        System.exit(0);
    }

    public TextField(Point topLeft) {
        super(topLeft, image.getWidth(), image.getHeight());
        this.addEventHandler(Event.OnInput, text -> contents += text);
        this.addEventHandler(Event.OnSubmit, __  -> onSubmit());
    }

    @Override
    public void draw() {
        image.drawFromTopLeft(this.topLeft.x, this.topLeft.y);

        DrawOptions opts = new DrawOptions().setBlendColour(Colour.BLACK);
        font.drawString(contents, this.topLeft.x, this.topLeft.y + image.getHeight() / 2, opts);
```

```
    }
}

public class SubmitButton extends Eventable {
    private static final Image image = new Image("res/button.png");
    private static final Font font = new Font("res/VeraMono.ttf", 24);
    private static final int TEXT_OFFSET = 50;

    public SubmitButton(Point topLeft, TextField text) {
        super(topLeft, image.getWidth(), image.getHeight());
        this.addEventHandler(Event.OnClick, __ -> text.onSubmit());
    }


    @Override
    public void draw() {
        image.drawFromTopLeft(this.topLeft.x, this.topLeft.y);

        DrawOptions opts = new DrawOptions().setBlendColour(Colour.BLACK);
        font.drawString("Submit",
                        this.topLeft.x + TEXT_OFFSET,
                        this.topLeft.y + image.getHeight() / 2,
                        opts);
    }
}

public class Program extends AbstractGame {
    private final List<Eventable> controls = new ArrayList<>();

    public Program() {
        TextField text = new TextField(new Point(300, 300));
        controls.add(text);
        controls.add(new SubmitButton(new Point(350, 400), text));
    }

    @Override
    protected void update(Input input) {
        controls.forEach(control -> control.update(input));
    }

    public static void main(String[] args) {
        new Program().run();
    }
}
```

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`.

   Solution:

```
public enum Direction {
    NORTH(0),
    NORTHEAST(45),
    EAST(90),
    SOUTHEAST(135),
    SOUTH(180),
    SOUTHWEST(225),
    WEST(270),
    NORTHWEST(315);
```

```
    private final int degrees;
    public int toDegrees() {
        return degrees;
    }

    private Direction(int degrees) {
        this.degrees = degrees;
    }
}
```

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?

Solution: the pipeline returns all strings that represent TV shows, and that are rated over 4.0.

```
List<String> list = Arrays.asList("Avengers: End Game", "Game of Thrones",
    "Jon Snow", "Arya", "SWEN20003", "Suits");

int count;
for (String str : list) {
    if (IMDB.isTVShow(str)) {
        Show show = IMDB.getShow(str);
        if (show.getRatings() > 4.0) {
            ++count;
        }
    }
}
```

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?