Assignment Snake

Object Orientation

Spring 2022

1 Snake

In this assignment you implement a JavaFX snake game. The snake can be turned left or right using the keyboard while it also moves forward at a quick pace. The user has to direct the snake towards the food. When the snake eats the food, its length increases by one and the food item is moved to a random location. Clicking the screen moves the food to the mouse position. The game ends when the snake collides with its own body or the playing field boundaries.

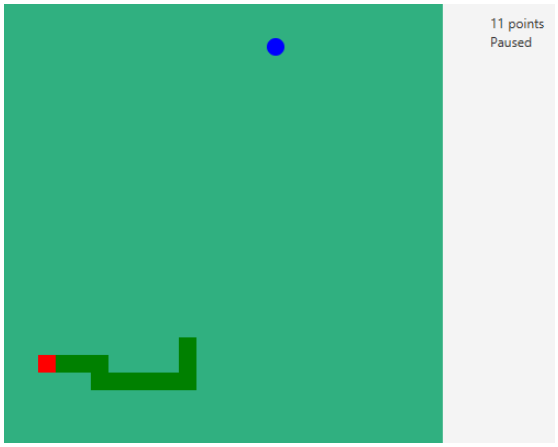

Figure 1: Example of a finished product. The snake is made up of green rectangles and a red head. The food item is a blue circle.

2 Learning Goals

After doing this exercise you should be able to:

- Use a `Timeline` to periodically execute code.
- Use keyboard handlers to process input.
- Know how and why to separate the game data, user interface, and input handler in GUI applications

3 Problem Sketch

The core functionality of this exercise consists of three major parts: the game data, the user interface and the input handler. The game data consists of all game logic, while the user interface handles all the visual representations on the screen and the input handler handles key presses and mouse clicks. For this assignment it is important that these three concepts are separated well. This implies that the game logic, the JavaFX objects and the EventHandlers are in separate modules/classes, using property binding to communicate between them. The project template on Brightspace should help achieving this.

While it is quite common for games to make game objects directly drawable, this practice is considered unidiomatic for user interface design, which is why, for the purposes of this assignment, you should try to keep them separated. There are other reasons for this, but one very practical reason is that many programming languages can run on multiple different platforms, but the libraries that handle visuals or input are often platform-specific (for example, the JavaFX library handles both visuals and input for us, but it doesn't work on Android, even though Android runs Java). When you port software to a different platform, or want to switch certain libraries for another reason, you only have to change the modules that use those libraries, so keeping them well-contained reduces the amount of work and increases the flexibility of your software.

4 Classes

The project template on Brightspace contains a total of 8 classes:

- `Main` creates a new JavaFX application
- `World` represents the complete state of one snake game.
- `Segment` represents the location of one segment of the snake.
- `Snake` represents the head of the snake and keeps track of all body segments.
- `Food` represents the location of the food item.
- `InputHandler` contains all functionality regarding keyboard- and mouse input and adapts the given world instance accordingly.
- `SnakeGame` is a JavaFX Pane that handles displaying the given world instance
- `Direction` is a simple enumeration you should use to indicate the direction in which the snake is currently moving. It contains two helper functions to rotate any direction 90 degrees to the left or right.

5 Assignment

For this assignment, you should implement the following functionality:

5.1 Timeline

The movement of the snake should be triggered by a `Timeline` every `DELAY` milliseconds with an infinite cycle count. The timeline should play or pause depending on the game's `running` state.

5.2 Movement

In every step, you should calculate the potential new position of the head of the snake. There are three different possibilities for this:

- If the snake head would collide with its own body or the field boundaries, it's game over and the animation should stop. You don't have to implement restarting the game.
- If the snake head would collide with the food, the snake should grow in length and the food should be moved to a random location.
- Otherwise, the snake moves to the new position, dragging its body with it.

5.3 Graphics

There is no explicit function for drawing the game elements, all visible elements should be JavaFX objects, such as `Circle` or `Rectangle`, located in `SnakeGame`. When the game data changes, the user interface should be updated automatically by binding the locations of the JavaFX objects to the location properties of the objects. Keep in mind that the coordinates of the visual representations change by a larger amount than the game coordinates of objects, due to them being scaled.

Keeping positions of shapes in sync with the game data can be accomplished using property bindings. But if the snake eats the food, it grows an additional segment, for which an additional square must be drawn on the screen. Growing the snake is implemented in the game logic. How should the graphical representation know that it has to create a new square for the new segment? We do not want to implement this functionality in `Snake`, because only `SnakeGame` may contian JavaFX code.

The solution is to implement a callback mechanism in `Snake`. Whenever the snake grows, it calls a registered function with the new segment as argument. `SnakeGame` has to register a function there on program startup, and this function can have JavaFX code. This function must create a new square and bind its location to the segment position.

In particular, this means:

1. In `SnakeGame`, implement `Snake.SnakeSegmentListener`. You can do this in a number of ways, for example by letting SnakeGame itself implement SnakeSegmentListener, or by creating a private inner class that implements SnakeSegmentListener, or by using a lambda expression.
2. In the program initialization code, register it to `Snake`'s list of listeners.
3. In `Snake`, call `onNewSegment(segment)` every time a new segment is created.

This pattern, called the *observer pattern*, is a simplified version of how property binding works behind the scenes. It allows us to accurately represent every object in `SnakeGame` without breaking the barrier between user interface and game data. In this exercise there will only be one listener, but in general there could be many. That's why `Snake` has a list of listeners, even though only `SnakeGame` will be listening.

5.4 Controls

Keyboard input should be handled in `InputHandler`. When the user presses `'s'`, the program should pause if it wasn't paused already and resume otherwise. You can use the `pause()` and `play()` functions from `Timeline` for this. When the user presses `'a'` or `'d'`, the snake should turn left or right, respectively.

5.5 Mouse

Mouse input should be also handled in `InputHandler`. When the user clicks the screen, the food should be teleported to its location, provided that it is on the field.

5.6 User Interface

There should be two lines of text on the screen. One should display "*x* points", where *x* should be the current score. The other should indicate whether or not the game is paused. If the game hasn't started yet, it should read "Press 's' to start" instead. The user interface should update automatically when the game state changes using property binding.

6 Finished Product

After implementing the features above, verify the following:

1. The snake moves forward every `DELAY` milliseconds.
2. Pressing `'a'` or `'d'` makes the snake turn left or right, respectively, while pressing `'s'` toggles between pause and running.
3. Clicking the screen teleports the food to the mouse location.
4. Eating the food increases the length of the snake by one and teleports the food to a random new location.
5. Biting your own tail or the wall pauses the game.
6. The user interface displays up-to-date information about the score and whether or not the game is paused.

If you get an error message about the missing JavaFX library, create a new global library of that name, and add all JavaFX jar files to it. Detailed instructions on how to do that can be found in the pie charts assignment text.

7 Submit Your Project

To submit your project, follow these steps.

1. Use the **project export** feature of NetBeans to create a zip file of your entire project: File → Export Project → To ZIP.
2. **Submit this zip file on Brightspace**. Do not submit individual Java files. Do not submit any other archive format. Only one person in your group has to submit it. Submit your project before the deadline, which can be found on Brightspace.