Hendrik PA Lensch
Lukas Ruppert
Raphael Braun
Hassan Shahmohammadi

Eberhard Karls
Universität
Tübingen

Praktische Informatik 2

Sheet 9

Submission date: 07.07.2022

Learning Goals

In this exercise, you will learn to implement and use *linked lists* and *dynamic arrays*. We will again use Object-Oriented Programming to transfer the "real-world" notion of objects into abstract code. For distinguishing between very similar objects with similar behavior, we will use `enum`s to distinguish between them without declaring individual classes.

About the Tests

In this task, you're relatively free to implement the required functionality in whatever way you see fit. Make use of helper functions, where necessary or convenient. We will inspect your data structures to judge whether you have implemented things correctly. See the Appendix for a coarse overview of the given code and the `interface`s your `class`es need to comply with. Make sure that your program does not crash when encountering edge cases, such as empty lists or arrays. When encountering invalid inputs, your functions should not alter the state of your objects. As always, the public tests you get to see are not enough to tell whether your implementation is correct.

9.1 Snake

This week's task is to use dynamic structures to implement a snake game. To make things more interesting, we make our own rules for the game.

9.1.1 The `Snake` (15 + 5 P)

In our case, the `Snake` is defined as a *linked list* of `SnakePiece`s. Each `SnakePiece` describes the 2D `Position` on the playing field and the `Food` that has been eaten to gain this piece of the snake. The playing field is a simple 2D grid, which is not explicitly represented in memory. The top-left corner on the screen has the `Position(0, 0)` and grows down (`y`) and to the right (`x`). Each `SnakePiece` directly follows its predecessor either horizontally or vertically. Diagonal movement or gaps between the pieces are not allowed (this comes naturally and does not need to be enforced). The first element of the `Snake` list conveniently represents its `head`, while the last element represents its `tail` (snakes don't have feet).

a) Implement the `Snake`, such that one can access its `head` and `tail` and add new `SnakePiece`s at the end. Each `SnakePiece` needs to hold a piece of `Food` and its `Position`.

b) Allow the game to check if a particular `Position` is occupied by the `Snake`.

9.1.2 Basic Movement (3 + 3 + 9 + 15 P)

a) The game can ask the `Snake` for its current `direction` of travel, or request a change in `direction`. However, the `Snake` will refuse to reverse its `direction` if there is more than one `SnakePiece`.

b) Based on the `direction`, the `Snake` can predict its next `Position`, which may be outside of the playing field. (This case will be checked by the `SnakeGame`.)

c) Every iteration of the SnakeGame, the Snake moves forward. The head moves towards the predicted next Position, while all following pieces each inherit the previous piece's Position.

There is no need to validate the given next Position. For now, we can ignore the Food parameter.

d) When running into the wall, the Snake does *not* die, but instead reverses, such that its head becomes its tail and vice versa. All elements in between are also rearranged accordingly. The Positions of the individual SnakePieces do not change, only their order in the list changes.

Since reversing itself puts a lot of stress on the Snake, it stays put and only continues to move forward in the next iteration.

After reversal, the Snake's direction should follow the orientation of the first two SnakePieces after reversal, or be the reversed direction, if there is just one SnakePiece.

9.1.3 The Food (10 + 10 + 5 + 5 P)

Of course, the Snake also needs some Food to eat. Each Food item has a FoodType assigned to it and a bestBefore date determining when the Food spoils and eventually disappears again.

a) To keep track of all the Food, implement a *dynamic array* that keeps track of these FoodItems. Both the Food and the Position will have to be memorized for each item.

The order of the individual Food items will not be important for us, but the game needs to be able to add and remove arbitrary Food items. Also, there should *never* be any gaps in the FoodItems.

b) The game needs to be able to check if there is a Food item at a given Position and remove it when it is eaten by the Snake.

c) All Food that has not yet been eaten by the Snake will eventually spoil. Each iteration, all uneaten Food whose bestBefore date lies past the SnakeGame's iterationCount becomes SPOILED.

d) All uneaten Food that is 100 iterations past its bestBefore date decomposes and should thus be removed from the FoodItems.

9.1.4 Eating Food (5 + 5 + 15 P)

a) When the Snake moves into a piece of Food, it gains a new SnakePiece containing that Food at the Position its tail had before moving.

The SnakeGame will pass the Food to the Snake, if there is any. As before, you also do not have to check the Position you're moving towards.

b) Eating SPOILED Food is unhealthy for the Snake. Instead of gaining a new piece, the Snake looses its head when eating SPOILED Food.

c) The Snake can also eat itself. In that case, all SnakePieces at the intersection are cut off and become regular Food pieces again, which are added to the FoodItems at their current Position.

9.1.5 Game Over

When there are no SnakePieces left, the game ends.

Appendix: Class Structure and Interfaces

To get you started, this is the rough structure of the classes we expect you to implement during this exercise. You can find these interfaces in SnakeInterface.java. Please create your classes such that they implement these interfaces to make sure that the tests run without errors.

```
interface ISnake {
    // 9.1.1 a)
    ISnakePiece getHead();
    ISnakePiece getTail();
    void addPieceAtTail(Position pos, Food food);

    // 9.1.1 b)
    ISnakePiece getPieceAtPos(Position pos);

    // 9.1.2 a)
    TravelDirection getDirection();
    void setDirection(TravelDirection newDirection);
    // 9.1.2 b)
    Position computeNextPosition();

    // 9.1.2 c) + 9.1.4 a) + 9.1.4 b)
    void moveTowards(Position pos, Food food);
    // 9.1.2 d)
    void reverse();

    // 9.1.4 c)
    ISnake cutTailAt(Position pos);
};

interface ISnakePiece {
    // 9.1.1 a)
    Food getFood();
    Position getPosition();
    void setPosition(Position newPos);
    ISnakePiece getNext();
    void setNext(ISnakePiece next);
};

interface IFoodItems {
    // 9.1.3 a)
    int getNumFoodItems();
    Food getFoodAt(int i);
    Position getPositionAt(int i);
    void addFood(Food newFood, Position newFoodPos);

    // 9.1.3 b)
    Food getFoodAtPos(Position pos);
    void removeFood(Food food);

    // 9.1.3 c) + 9.1.3 d)
    void spoilAndRemoveOldFood(int iterationCount);

    // 9.1.4 c)
    void addFoodFromCutSnake(ISnake cutSnake);
};
```

You will also be given the following code containing some basic structures in `SnakeUtils.java`:

```java
enum FoodType {
    STRAWBERRY, BANANA, BLUEBERRY, CABBAGE, SPOILED
};

class Food {
    private FoodType type;
    private int bestBefore;

    Food(FoodType type, int bestBefore) {
        this.type = type;
        this.bestBefore = bestBefore;
    }

    FoodType getType() { return type; }
    int getBestBefore() { return bestBefore; }

    void spoil() { this.type = FoodType.SPOILED; }

    boolean equals(Food other) {
        return other != null && type == other.type && bestBefore == other.bestBefore;
    }

    public String toString() {
        return "["+type.toString()+", best before "+bestBefore+"]";
    }
};

enum TravelDirection {
    UP, DOWN, LEFT, RIGHT
}

class Position {
    int x;
    int y;

    Position(int x, int y) {
        this.x = x;
        this.y = y;
    }

    boolean equals(Position other) {
        return other != null && x == other.x && y == other.y;
    }

    public String toString() {
        return "["+x+", "+y+"]";
    }
};
```

The main game and GUI is implemented in the `SnakeGame.java`. You will have to instantiate your `Snake` and `FoodItems` `class`es in the `SnakeGame` constructor to be able to play the game.