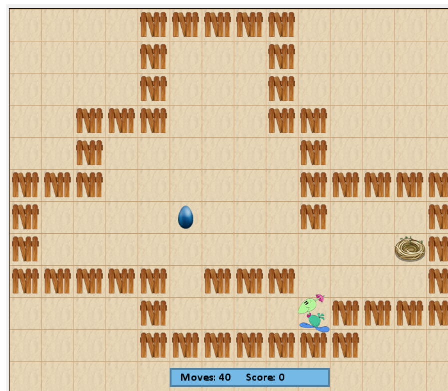
Assignment 8: Sokoban

– Algorithmic Thinking and Structured Programming (in Greenfoot) –

©2015 Renske Smetsers-Weeda & Sjaak Smetsers
Licensed under the Creative Commons Attribution 4.0 license,
https://creativecommons.org/licenses/by/4.0/

1 Introduction

In this assignment you will write a game in Greenfoot known as *Sokoban*. See http://sokoban.info/ for an example. Our variation is called 'Mimi the eggs-hoarder'. The original game takes place in a warehouse. The player is a warehouse employee whose task is to push crates to the correct location.



This game is played in the Madagaskar world; the world in which Mimi lives. The goal is for Mimi to push all the eggs into the nests. The following rules apply:

- Mimi can only push eggs forward, she can't pull an egg;
- Mimi cannot sit on an egg or step over it;
- Mimi can only push one egg at a time. As a consequence, Mimi cannot push two (consecutive or adjacent) eggs simultaneously;
- No two eggs can occupy one cell (not even in a nest);
- A nest can hold only one egg;
- An egg can be pushed out of a nest;
- Neither Mimi nor an egg can be pushed through a fence;
- Mimi can step over a nest (of course, very carefully so that she doesn't destroy it). If the nest holds an egg, she will push the egg out;
- There are as many nests in the world as eggs;
- The level is completed when each nest is filled with an egg.

2 Learning objectives

After completing this assignment, you will be able to:

- handle user interaction in the code;
- apply nested **if** .. then .. **else** statements;
- apply the knowledge from previous assignments to implement a game on your own.

3 Instructions

For this assignment you will need the **'DodoScenarioSokoban'** scenario.

4 Theory

Contrary to the previous assignments, you will not implement an algorithm to make Mimi do something. This time the user directs Mimi by pressing a key, and in the code we must explain what Mimi must depending on which key is pressed.

We start by explaining a particular use of **if** .. then .. **else** statements, this time in the context of Sokoban. This is needed to incorporate user-interaction for the different keys.

Nested if .. then .. else .. statements
You can test multiple cases simultaneously using a nested **if** .. then .. **else** statement. This was explained in assignment 7, using a general example. We will now give a Greenfoot example that will also be used in this assignment.

Example:
Suppose we want to check whether the user has pressed one of the arrow keys on the keyboard, and if so, have Mimi face that direction.

Code using nested if .. then .. else .. statements
The Greenfoot method **boolean** `isKeyDown(String key)` tests whether the user has pressed a particular key. For example, to check whether the "arrow-up" button has pressed, the following method is called:

```
Greenfoot.isKeyDown ("up");
```

Similarly, the string `"left"` is associated with left-arrow-key, `"right"` with the right-arrow-key, and `"down"` with the down-arrow key.

In the `handleKeyPress` method this test is used to determine which key has been pressed and changes the direction in which Mimi is facing accordingly:

```
public void handleKeyPress() {
    if ( Greenfoot.isKeyDown( "left ") ) {
        setDirection ( WEST );
    } else {
        if ( Greenfoot.isKeyDown( "right" ) ) {
            setDirection ( EAST );
        } else {
            if ( Greenfoot.isKeyDown( "up" ) ) {
                setDirection ( NORTH );
```

```
                } else {
                    if ( Greenfoot.isKeyDown( "down" ) ) {
                        setDirection ( SOUTH );
                    }
                }
            }
        }
    }
```

Flowchart using nested conditional statements:

The corresponding flowchart looks like this:

Figure 1: Flowchart for `handleKeyPress`

Note:

As a result of the numerous accolades and the fact that the code stretches further and further to the right, the code becomes hard to oversee and furthermore being error-prone.

Code using `else .. if ..` statements

In Java, there is an alternative way for writing these types of nested conditional statements. Most programmers combine the **`else`** and the **`if`** into one code-line using an **`else .. if ..`** statement. The method `handleKeyPress` then becomes:

```
public void handleKeyPress() {
    if ( Greenfoot.isKeyDown( "left" ) ) {
        setDirection ( WEST );
    } else if ( Greenfoot.isKeyDown( "right" ) ) {
        setDirection ( EAST );
    } else if ( Greenfoot.isKeyDown( "up" ) ) {
        setDirection ( NORTH );
    } else if ( Greenfoot.isKeyDown( "down" ) ) {
        setDirection ( DOWN );
    }
}
```

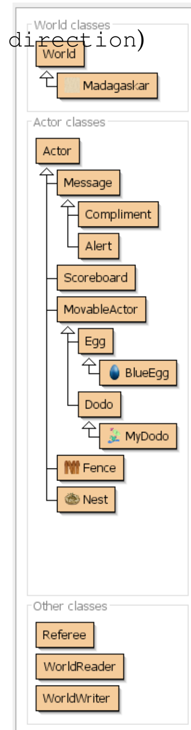
Note:

This results in less lines of code, and furthermore, the code-structure becomes much clearer.

5 Exercises (implementation of the game)

For this assignment you will start with a brand new scenario. Compared to the previous scenarios, some minor modifications have been made. The most important addition is the `MovableActor` class.

Each `Dodo` can be moved (and is thus movable), and now belongs to the `MovableActor` class. Contrary to the previous assignments, `Egg`s can now be moved too. As a consequence, `Egg`s now belong to the `MovableActor` class too. The class diagram shows both `Dodo` and `Egg` as sub-classes of `MovableActor`.



The `MovableActor` class basically has no new functionality. The main difference is that several 'move'-related methods such as **void** `step (`**int** `direction)` and **boolean** `borderAhead (`**int** `direction)` have been moved from the `Dodo` class to the `MovableActor` class. Mimi now inherits these methods from `MovableActor`. See figure 2:

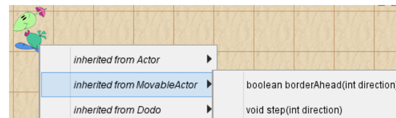


Figure 2: Methods which Dodo inherits from `MovableActor`

Because `Egg`s now also belong to the `MovableActor` class, they too can use the `step` and `borderAhead` methods. In addition, several new methods have been added to the `Egg` class. The most important are:

void `push (` **int** `direction)`: which allows an egg to be pushed in a particular direction. This direction is passed as a parameter.

boolean `canBePushed(` **int** `direction )`: which can be used to test if an egg can be moved in the specified direction.

5.1 Getting started

1. Download and open the `DodoScenarioSokoban` scenario;
2. Compile the scenario and press *Run*. Mimi should run along the inside of the fence that encloses the world.
3. Using the right-mouse-button, test the new `Egg` methods `push` and `canBePushed` several times, each time with the egg in a different position in the world. Do these methods work as expected?
4. Replace the code in `MyDodo`'s `act` method by a call of the method `handleKeyPress`.
5. Compile and run the scenario (by pressing the *Run* button). Desbribe what happens. If nothing happens, then press one of the arrow keys on your keyboard. As you can see the `handleKeyPress` method is not complete yet. Which key(s) does Mimi respond to? Which doesn't Mimi respond to.

5.2 Mimi responds to arrow keys

As you may see, lots of things don't work properly yet. Mimi walks through fences and over eggs, and does not adequately respond to the arrow keys pressed by the user. In this task you will create order in Mimi's world.

We start by fixing Mimi's behavior to the user's instructions:

1. Open the `MyDodo` class in the editor and find `handleKeyPress`.
2. Have a look at the body of the `handleKeyPress` method. Here, a submethod called `getNewDirection` is used.
3. Consider where you have to change which code so that Mimi adequately responds to all the arrow keys. Tip: Have a look at the theory block above about 'Nested **if** .. `then` .. **else** .. statements' in which the Greenfoot method `isKeyDown` is discussed.
4. Adjust and compile `MyDodo`.
5. Run the scenario and make sure Mimi indeed responds to each arrow key adequately. If not, adjust your code.

5.3 Mimi pushes eggs forward (A)

Mimi takes a step when the user presses an arrow button. However, if Mimi is in front of an egg, she shouldn't merely take a step. More must happen! We will now ensure that Mimi doesn't just step over an egg, but that she pushes it forward. We will now adjust `handleKeyPress` to work as expected. To do so, follow the next tips:

- First decide which specific cases should be distinguished and adjusted;
- The `Dodo` class contains all the methods which you need to distinguish these cases;
- Obviously, the new `Egg` methods will be useful;
- Mimi can check whether there is an egg laying directly in front of by calling `eggAhead`;
- Mimi can get a hold of the egg using `getEggAhead`;
- Using the existing methods, she can ask the egg whether or not it can be pushed, and if so, push it forwards and then (in the same direction), take a step herself.

Compile and test your changes. Systematically try a few cases. Fix any errors you encounter. Make sure that Mimi indeed complies to all the rules described in chapter 1.

5.4 Scoreboard

We need a scoreboard in order to track how many steps Mimi has taken and how many eggs she has already placed in a nest. We will let Mimi keep track of these scores herself and have a scoreboard display the values. Do this as follows:

1. First, add two instance variables to `MyDodo` for storing this information (one for number of steps taken, the other for the number of eggs in a nest). Consider meaningful names, appropriate types and a suitable initial value.
2. As soon as Mimi takes a step, change the value of the instance variable for the number of steps taken.
3. After each step, call `updateScores` to ensure that the changed situation is actually displayed on the scoreboard.
4. To ensure you haven't made any mistakes thus far, compile and test your changes before proceeding.
5. Now add the second variable to keep track of how many eggs have been placed in a nest. Consider when and how you can determine this. Tip: The `Dodo` method **`boolean`** `nestAhead()` may be useful.

6. Again, call `updateScore`.
7. Compile and test your program.

5.5 Egg in its nest

At the time-being, when an egg is pushed into a nest, you can't see it anymore. This is because the egg and the nest are in the same cell, and the egg is hidden behind the nest. That's too bad! Let's change that. So, if an egg is pushed into a nest, we want to show a nest with an egg in it. To do this, we first adjust the `Nest` class.

1. Open the `Nest` class in the editor.
2. Add an instance variable to this class which indicates whether the nest is empty or not. Ensure that this variable gets an appropriate initial value (i.e. when a new nest is brought into the world, what should its value be?).
3. Add a method to fill the nest when appropriate. The effect of this is not only to change the value of the instance variable, but also to adjust the picture which is shown. Tip: Look back to assignment 6 "To be hatched or not to be hatched" in which you did something similar. There, in the class `Egg`, you also changed the picture of an egg into a hatched egg.
4. Compile and test your program using your right-mouse-button.
5. Similarly, add a method to empty the nest when an egg is pushed out.
6. Compile and test your program using your right-mouse-button.
7. Also adjust your `MyDodo` class. First consider what exactly must happen and where your code needs to be adjusted. Recall that the nest itself does not know whether or not it contains an egg; Mimi, however, does! To change the state of the nest, you need (a reference to) the nest object. Which method can you use to do this? Keep in mind that you can also push an egg out of a nest.
8. Compile and test your program, this time also run the scenario.

5.6 Level achieved (A)

The program must now decide whether the level has been completed or not. We introduce a referee to make this decision.

1. Have a look at the code of the `Referee` class.
2. At the top of the class an instance variable `myWorld` is declared which references to the world. Further down, you can find a method `updateScoreboard`. This method is used in the `Dodo` class and wille called as soon as `updateScores` adjusts the score.

   If you wish, you can check this yourself by opening the `Dodo` class in the editor and having a look at the `updateScores` method.
3. The second score, called `score2` indicates how many eggs are already in a nest. If this is equal to the number of eggs in the world, the level has been completed. Add a method that checks whether the game is over. If this is **`true`**, call the method `levelFinished`. Tip: Use the `World` method `getObjects` to determine how many eggs the world contains. Have a look at the Java documentation about what the method does and consider how you can use the result of this method to determine if the level has been completed or not.

5.7 Optional: Oops .. undo ?

Finally, it is still a bit unsatisfying that when the user accidentally presses the wrong key he may not be able to complete the level anymore (he gets into a situation in which the level cannot longer be solved). An *undo* function would be helpful.

In order to achieve this, some code adjustments must be made in the `myDodo` class. We will only give some general indications on how to do this; its up to you to come up with and implement the details.

1. Determine a suitable key for an *undo*. Add a method to handle the user-interaction.
2. Determine which information must be remembered/stored in order to undo a move.
3. Besides having Mimi take a step back, sometimes an egg's position must be restored (or undone) too.
4. Consider what should happen if the user wants to undo multiple steps.
5. Does the correct value appear on the scoreboard after an *undo*?

5.8 Optional: New levels

Add your own new challenging levels to the scenario. Let a fellow student test whether they are solvable or not. Also have them decide what the level's difficulty is.

6 Summary

You can now:

- handle user-interaction (keyboard input);
- make use of **`else if`** statements;
- use what you have learned in previous assignments to write a game on your own.

7 Saving your work

You have just finished the bonus assignment. Save your work!

8 Handing in

Hand your (digital) work in via email to renske.weeda@gmail.com: