Assignment 7: Dodo's Race

Algorithmic Thinking and Structured Programming (in Greenfoot)

© 2017 Renske Smetsers-Weeda & Sjaak Smetsers[1]

Contents

Introduction 1

Learning objectives 1

Instructions 1

Theory 2
- 7.1 Visibility . . . 2
- 7.2 Class constant . . . 3
- 7.3 Instance variables . . . 3
- 7.4 Class constructor . . . 4
- 7.5 Steps for using instance variables . . . 5

Challenges 6
- 7.1 Random movements . . . 6
- 7.2 Dodo scores . . . 6
- 7.3 Finding the nearest egg . . . 8
- 7.4 Continuously hatch the nearest egg in a list . . . 8
- 7.5 Dodo's Race . . . 9

Reflection 11

Saving and Handing in 12

[1]Licensed under the Creative Commons Attribution 4.0 license: https://creativecommons.org/licenses/by/4.0/

Introduction

Who can make the smartest Dodo? In this assignment you are going to compete against your classmates. The idea is to come up with the most optimal (not brute force) algorithm and implement it.

In the previous assignments you have written and executed quite some (complex) code. Now you are going to combine all the things you have learned to work out and implement an (Artificial Intelligence) algorithm.

We first start with some theory followed by exercises to practice the given theory. To get you started, we will first implement a few simple algorithms together. Along the way you may get your own ideas on how to improve these. The goal is to devise an algorithm which is generic enough to be the 'smartest' in any unknown scenario. After that you will start programming for Dodo's race.

The goal for writing the code in the previous assignments code was to practice and learn. In the process, the code might have become cluttered and messy. Besides, you won't need all that code for this assignment. Therefore, we will begin this new assignment with a 'clean' scenario. Hopefully you will realise that you may want to use a method that you've already written and tested in a previous assignment. You can then simply copy-paste it into the new scenario (and retest it).

Learning objectives

After completing this assignment, you will be able to:

- use random numbers;
- use `private` visibility to protect data from being visible or accessible by other objects;
- use class constants for variables that never change throughout a program;
- use instance variables as an object's memory;
- recognize a class's constructor and understand that it initialises an object (and its variables);
- split a complex algorithm into subalgorithms;
- implement a sequence of subalgorithms.

Instructions

For this assignment you will need:

- scenario **'DodoScenario7'**: to be downloaded from the course website[2].

Throughout the assignment you will also need to answer some questions. The following must be handed in:

- All flowcharts: use pencil and paper, or go to `https://www.draw.io/`;
- Your code: the file `MyDodo.jav` contains all your code and must be handed in;
- The reflection sheet: complete and hand it in.

[2] `http://course.cs.ru.nl/greenfoot/`

You must discuss all other answers with a programming partner. Jot down a short answer on (the assignment) paper.

There are three types of challenges:

★	**Recommended**. Students who need more practice or with limited programming experience should complete all of these.
★★	**Mandatory**. Everyone must complete these.
★★★	**Excelling**. More inquisitive tasks, designed for students who completed 2 star tasks and are ready for a bigger challenge.

Students who skip 1-star challenges should complete all 3-star challenges.

A note in advance:

- In this assignment you may only make changes to the `MyDodo` class;
- You may use methods from the `MyDodo` or `Dodo` class, not from the `Actor` class;
- Teleportation is not permitted: if Mimi needs to get somewhere, she must walk there!

Theory

Theory 7.1: Visibility

You can protect your data by making these less visible or accessible to other objects. There are different levels of visibility or accessibility of methods or instance variables (the concept of an instance variable will be explained later). The possibilities for the visibility are:

Visibility	Explanation
public	accessible from outside the class
private	only accessible from within the class itself
protected	only accessible from within the class or its subclasses

Private elements are not visible to other classes. An element that has been defined as `public` can be reached (and can thus possibly be modified) by other code/objects in the program. In object-oriented programming, each class can have `public` methods. Objects in other classes can request 'public' information or let the object do something which is 'public'. An example is `public boolean canMove()` which you can use to ask Mimi if she can move or not. Objects of other classes can see and call any public accessor or mutator method.

On the other hand, the instance variable `private int myNrOfStepsTaken` is private and can't be accessed or modified by another object. No one knows better than Mimi herself, how many steps she has taken. No one else should be allowed to change that value. So it is, and should be, Mimi's responsibility to do keep track of this herself.

However, **how** Mimi does what she does is up to her. The implementation of the method is private. How Mimi determines if she `canMove` is up to her, it is nobody else's business, and is hidden from all other objects. So, objects of other classes can call public accessor or mutator methods.

From a security or privacy point of view, you may not want others to know the details of precisely how something is done (think about encrypting messages). This principle is called information hiding. It says that the specific details about how a class implements something should be hidden from other classes.

The modularity of your program is improved by controlling which methods can or can't be called from outside the class.

Until know we have made all methods public, just to simplify things. That's actually not good practice. One of the fundamental ideas of object orientation is to let an object maintain control and knowledge over itself. So, by default, all methods and variables should be `private`. This is safer. In such a

manner you force another object to ask if Mimi will give an egg away, instead of just taking it away from her.

Theory 7.2: Class constant

A class constant is a variable whose value can't change throughout the program. Moreover, it has the same value for all instances. You can recognize a constant by `static final`. Class constants can be `public` or `private`.

Example

An example of a constant declaration (at the top of the `Madagaskar` class) is:

```
private static final int MAXWIDTH = 12;
```

Because this constant is `private` it can only be used in the `Madagaskar` class itself. The `final` ensures that:

- it is impossible to change its value somewhere in the program;
- you must immediately give the class constant an initial value (at the same time as the declaration).

The class constant `MAXWIDTH` has the value 12 because we want a world which is 12 cells wide. In the code we can then refer to `MAXWIDTH` instead of using (a hard-coded) 12. Now, if we want to write a game with a world which is 40 by 40 cells, we can easily change the size throughout the program by changing only one value (by replacing 12 in the declaration by 40). So, using class constants makes it easier for us to modify our program.

Naming conventions for a constant

The name of a constant:

- is meaningful: it corresponds to what the constant is used for;
- consists of one or more nouns;
- consists of letters and numbers: it does not contain spaces, commas, or other 'strange' characters (with exception to '_');
- is written in capital letters: words are separated by a '_';
- for example: `WORLD_FILE`.

Using public constants in other classes

Class constants that are `public` can be referred to from other classes.

For example, `Madagaskar` has a `public int MAXSTEPS`. To refer to to this constant, use: `Madagaskar.MAXSTEPS`.

Theory 7.3: Instance variables

Instance variables are an object's 'memory'. They contain information that an object stores or uses. Until know we used local variables which are destroyed as soon as the method ends. In contrast, instance variables exist as long as the object itself exists.

An instance variable is a type of variable: just like any other variable is has a type and a name, such as `int myNrOfStepsTaken`. They also adhere to the same naming conventions. (Instance variables are sometimes called attributes or fields.) Note: by adding `my` to its name you can easily recognize that it is an instance variable (and not a local variable).

Instance variables belong to a particular object. For this reason, the visibility of an instance variable should always be `private` (principle of information hiding, see Theory 7.1). If another object needs to know what the value is, is should ask the object by calling the object's public accessor method.

```
public class MyDodo extends Dodo
{
    /* INSTANCE VARIABLE DECLARATION: */
    private int myNrOfStepsTaken;

    /* CONSTRUCTOR: */
    public MyDodo() {
        super( EAST );
        /* INSTANCE VARIABLE INITIALISATION: */
        myNrOfStepsTaken = 0;
    }
```

Figure 1: Code for declaring and initialising an instance variable `int myNrOfStepsTaken`

Declaration

An instance variable is declared in a class. It should always be `private`. We make instance variables `private` so that other objects don't have the power to change values without permission. During the declaration the variable is given a type and a name.

For example: `private int myNrOfStepsTaken` (see the top part of figure 2). With this variable, MyDodo stores how many steps she has taken. Because this value is `private`, only she knows what its value is and only she can change it. Another object can't access or change the value of the variable `int myNrOfStepsTaken` directly.

Initialization

An instance variable is given an initial value in the class's constructor (the constructor is explained in the next theory block). When the object is created, the variable automatically gets this value. For example, when a new MyDodo is placed in the world, then it has not taken any steps yet. We indicate this in the constructor with: `myNrOfStepsTaken = 0`. See the bottom part of figure 2.

Lifetime

An instance variable has the same lifetime as the object to which it belongs. When an object is created (with `new`) the constructor-method is automatically called. All of the object's instance variables are created and initialized. The instance variables exist as long as the object exists. So, as long as you see Mimi in the world, the variable `myNrOfStepsTaken` exists and is not reset to '0'.

Theory 7.4: Class constructor

The constructor is a special method which is called as soon as an object is created (with `new`, either in the code or right clicking in the class diagram). A constructor is used to initialize an object and give its instance variables their initial values.

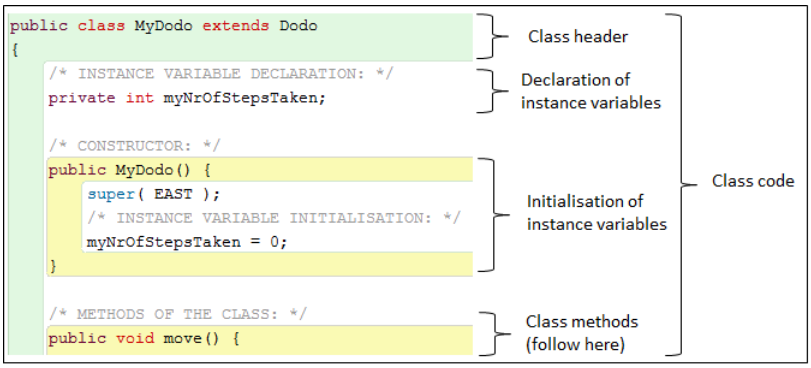
Recognizing the constructor

The constructor looks similar to a normal method, except for two things:

- a constructor's name is always the same as the class' name;

- and the constructor never has a return type, not even `void`.

The constructor for `MyDodo` can be found in the `MyDodo` code:



This class has:

- one instance variable: `private int myNrOfStepsTaken;`
- one constructor `public MyDodo (...) ;`
- any other methods, such as `public void move( )`.

Calling the constructor

You can call `MyDodo`'s constructor using by right-clicking in the class diagram and selecting `new MyDodo ()`. In code it looks like this: `MyDodo mimi = new MyDodo ();`.

Theory 7.5: Steps for using instance variables

Instance variables are used to give an object 'memory'. In order to use instance variables, take the following steps:

1. **Declare** the instance variable: indicate the name, type and visibility (as a rule we keep this `private`). An example is: `private int myNrOfStepsTaken ;`.
2. **Initialize** the instance variable (give it an initial value) in the constructor (the next theory block explains how to do this): as soon as the object is created, the variable is given this initial value. An example is: `myNrOfStepsTaken = 0;`.
3. (optional) Write a **getter accessor method** so that an object from another class can ask the variable's value.
4. (optional) Write a **setter mutator method** so that an object from another class can modify the variable's value.
5. **Modify** the instance variable: in the object's code, decide where the instance variable should be changed. For example: if it's used to keep track of the number of eggs hatched, then it should be increased each time an egg is hatched. An example is `myNrOfStepsTaken ++;`.

Note

If you don't want other objects to be able to ask or change a variable's value, then don't make getter or setter methods.

Challenges

Please read Theory 7.1: Visibility.
Please read Theory 7.2: Class constant.

Challenge 7.1: Random movements

In this challenge we let Mimi walk through the world, taking steps in random directions (almost as if she were drunk).

a) To choose a random direction, use the following `Dodo` method: `public int randomDirection ( )` Open the `Dodo` class and find this method.

b) Right-click on Mimi and choose the `Dodo` method `randomDirection`. Call the method several times, each time jot down the result. What do you notice? What does this have to do with the parameter '4' that `getRandomNumber` is given?

c) Draw a high-level flowchart describing how Mimi takes 40 steps, before each step she chooses a new random direction and then takes a step in that direction. Make sure that Mimi doesn't try to step out of the world or walk into a fence so that she actually takes 40 steps without getting "stuck" (we don't want the error message popping-up).

d) In `MyDodo`, write the corresponding method `moveRandomly ( )`, including (JavaDoc) comments.

e) Test your program.

f) You have now hard-coded the value '40', making the code harder to maintain. The maximum number of steps has been defined in the `Madagaskar` class with the class constant `MAXSTEPS`. Find it and make sure it is set to the appropriate value. Replace '40' with `MAXSTEPS`. Theory 7.2 explains that you must use: `Madagaskar . MAXSTEPS`.

g) Test your program again.

Please read Theory 7.3: Instance variables.
Please read Theory 7.4: Class constructor.

Please read Theory 7.5: Steps for using instance variables.

Challenge 7.2: Dodo scores

The game ends as soon as the maximum number of steps has been taken. We will also use a scoreboard to indicate how many steps can still be taken and what the current score is (depending on which eggs have been hatched so far). No one can keep track of how many steps Mimi takes better than Mimi herself. In this challenge you will create a scoreboard and give Mimi a special type of (private) memory so that she can keep track of her score. This memory is private so that no one else can tamper with it.

a) **Adding the scoreboard:** We will add a scoreboard to the scenario. We do this as follows:

i) Open the `Madagaskar` class and look for the method call of `addScoreboard ( )` in the constructor. The call has already been typed, but has been placed in comments.

ii) Remove the comment marks so that the scoreboard is added as soon as the world is created.

iii) In the scenario, check that the scoreboard has been added at the bottom of the world. How many values does the scoreboard hold?

b) **Keeping track of Mimi's score:** No one can keep track of how many steps Mimi takes better than Mimi herself. To do this, add an instance variable `myNrOfStepsTaken` to the `MyDodo` class. We first set the variables for keeping track of the steps taken and the current score, in the next step we will adjust the scoreboard accordingly.

i) The `myNrOfStepsTaken` variable keeps track of the number of steps taken.

ii) Similarly, add an instance variable `myEggScore` to the `MyDodo` class to keep track of the egg-points scored (score).

iii) We will now adjust the code in order to be able to use these variables. Follow the next steps:

1) Give both instance variables (`myNrOfStepsTaken` and `myEggScore`) their correct initial values. Tip: Do this in the constructor of `MyDodo`.

```
public class MyDodo extends Dodo
{
    /* INSTANCE VARIABLE DECLARATION: */
    private int myNrOfStepsTaken;

    /* CONSTRUCTOR: */
    public MyDodo() {
        super( EAST );
        /* INSTANCE VARIABLE INITIALISATION: */
        myNrOfStepsTaken = 0;
    }
```

Figure 2: Code for declaring an instance variable. It is initialized in the constructor.

2) Make sure that `myNrOfStepsTaken` is modified as soon as Mimi takes a step. Tip: Do this directly after the `move( )` call.

3) Add code to your method `moveRandomly( )` (from Challenge 7.1) so that:
   - Mimi picks up an egg when she finds one.
   - You may want to change the name of the method to a more meaningful name.

4) Make sure that `myEggScore` is modified as soon as Mimi picks up an egg. Tips:
   - With each call of the `pickUpEgg( )` method, store the egg in an `Egg` variable (for example `Egg currentEgg`).
   - Find out how much that egg is worth by using `currentEgg.getValue( )`.
   - Modify the `myEggScore` according to the number of points the egg picked up is worth.

c) **Updating the scoreboard:** `Dodo` also has an `updateScores` method with which Mimi can update the scoreboard. As soon as Mimi has taken a step or if she has found an egg, the scoreboard must be updated.

(a) Right-click on Mimi and select `void updateScores( int score1, int score2)` (inherited from Dodo). Fill two numbers in as parameters. What does the scoreboard show? Tip: The method may be hidden behind 'more methods'.

(b) Call `void updateScores` in your method so that the new values are shown on the scoreboard. Tips:
   - Make sure that the first value shows how many steps Mimi is still allowed to take (not how many steps she has taken).

- Adjust the score as soon as any of the two values (score or the number of steps taken) change.

d) Test the program by calling your method which makes Mimi move randomly. Make your test complete: also test picking up a golden egg (5 points) or a rainbow egg (random number of points).

e) When the game is over, give Mimi a compliment (in a dialogue) with her score.

Challenge 7.3: Finding the nearest egg

Mimi is rather lazy. She only wants to pick up the nearest egg, regardless of how many points its worth. Can you help her find and hatch it?

a) Given the following (partial) flowchart:

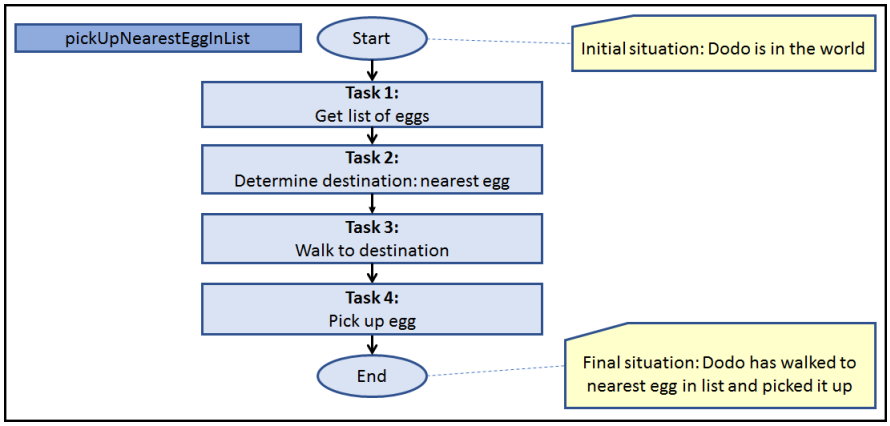


Figure 3: Flowchart for 'Finding and picking up the nearest egg'

b) Write sub-methods for each task. Tips:

Task 1: Make use of an existing method to get a list of eggs in the world (note: NOT Surprise eggs). Use a local variable to store this list.

Task 2: Write a method which returns the nearest `Egg`:

(a) Determine which variables you need for your solution.

(b) Write a sub-method which determines the number of steps it will take Mimi to get to her egg. Tip: you may be able to re-use code from Challenge 6.9.

(c) For each of the eggs, print its coordinates and its distance. This will make it easier to test if your method works properly.

(d) Compile and test your method by right-clicking.

Task 3: Re-use code from Challenge 4.5.

Task 4: Make use of an existing (`Dodo`) method.

c) Come up with an appropriate method name for the entire method (all the subtasks together).

d) Compile and test your method as a whole.

Challenge 7.4: Continuously hatch the nearest egg in a list

Write a method that makes Mimi walk to the nearest egg in a list and hatch it, until all the eggs in the world have been hatched.

Some tips:

- Write a sub-method `void pickUpAndRemoveNearestEggInList ( List<Egg> eggList )` (similar to Challenge 7.3). This method is given an eggList, finds the nearest egg and removes it from the list.
- Write a method `void repeatedlyPickUpNearestEggInList ( )` which gets the list of eggs in the world and calls your sub-method appropriately.

Challenge 7.5: Dodo's Race

Who can you make the smartest Dodo and win the competition? To win you will probably need to add some more 'intelligence' to Dodo. Can you make Dodo smarter?

Help Mimi hatch as many eggs as possible, in as few steps as possible. This will be a competition between you and your classmates. Who can come up with the best algorithm?

One golden egg (worth 5 points) and 15 blue eggs (each worth one point) will be placed in the world at random. Mimi is only allowed to take 40 steps (MAXSTEPS). The goal is to get the highest score by picking up the eggs? During the final competition, your program and that of your classmates, will be run in a new world. Which scenario you will be given will be a surprise. Who can make the smartest Dodo?

Work in a structured manner! Come up with a **simple** algorithm first, draw a flowchart, and then work this out into code and test it. After that, incrementally make improvements, testing each minor adjustment you make.

Competition

Assumptions:

- There are no fences or other objects in the world, only one `MyDodo` object and several `Egg` objects.
- There are 15 blue eggs and one golden egg.
- The golden egg is worth five points.
- A blue egg is worth one point.
- A maximum number of steps may be taken.
- During the final competition an unknown world will be used. Right now, you don't know where the eggs will be laying.

Rules for the race: To make the competition as fair as possible there are a few rules:

- Mimi can only be moved using `MyDodo`'s `void move( )` method.
- The team with the highest score wins the competition.
- Mimi can take no more than 40 steps. Make sure the constant `int MAXSTEPS` in `Madagaskar` is set to 40.

Example worlds: You can use the following example worlds to test your algorithm:

- worldDodoRace1
- worldDodoRace2

- worldDodoRace3
- worldDodoRace4

It may be useful to create your own worlds for testing.
New components: To make the race possible (and fair), we have to add some things to our program:

- Mimi has to keep track of how many steps she has taken;
- Mimi has to keep track of her score;
- a scoreboard, which shows the number of steps taken and the score;
- the game must stop when the maximum number of steps has been taken;
- Mimi has to collect as many points as possible by finding eggs and picking them up;
- ... and Mimi must become much smarter!

Deliverables

After you finish implementing your solution, make sure that your code includes (JavaDoc) comments. Have a look back on how it went and answer the following questions:

1. How did it go? What did you find hard to do?
2. How satisfied are you with the your solution?
3. In which situations will your algorithm not work very well? What is the 'worst-case' scenario? In other words: which scenario do you really hope to **not** get during the race?
4. In which situations does your algorithm work well? What is the 'best-case' scenario for your algorithm? How many steps will Mimi need to take?
5. Why is your algorithm better than the algorithms that you developed in the other exercises of this assignment?
6. What did you learn by doing this assignment?
7. What would you do differently next time?
8. Take a picture of any design preparation you did. This could be pseudo-code, flowcharts, or sketches on (scrap) paper. Hand these preparations in together with your code.

Reflection

In this assignment you practiced using list to store values. You also learned about the for-each-loop to traverse lists and how to use a list to store (Egg) objects. One of the most important steps in becoming good at anything is to evaluate and reflect on what you did and how it went:

Result

I know my solution works because …
I am proud of my solution because …
I could improve my solution by …

Method

My approach was good because …
What I could do better next time is …

Fill the following table with smileys indicating how things went.

	I can do it
	I did it a bit but didn't fully get it
	I didn't get it at all

	I can use `private` visibility to protect data.
	I can use class constants for variables that never change throughout a program.
	I can use instance variables as an object's memory.
	I can use a constructor to initialize an object (and its variables).
	I can break a complex algorithm down into subtasks and implement and test these separately.

Saving and Handing in

You have just finished the assignment. Save your work! You will need this for future assignments. In the Greenfoot menu at the top of the screen, select 'Scenario' and then 'Save'. You now have all the scenario components in one folder. The folder has the name you chose when you selected 'Save As ...'.

Handing in

Hand in the following:

- Your code: The java file `MyDodo.jav`;
- Flowcharts: paste (photo's of) your flowcharts in a Word document;
- The reflection sheet: complete and hand it in.