Assignment 6: Dodo keeps getting smarter

– Algorithmic Thinking and Structured Programming (in Greenfoot) –

1 Introduction

In the previous assignment you learned how to use variables. Mimi became smarter by being able to remember things.

More specifically, you used local variables. Local variables are declared and initialized in a method. When the method finishes, the value becomes destroyed and can't be used anymore. And each time that the method is called (directly by right-clicking, or indirectly in the `act`) the variable is re-initialized and loses its 'old' value. If you want to write a more complex program or game you may want to be able to store values throughout the whole program (such as a score). An **instance variable** is used to store information throughout the entire program. In this assignment you will learn how to use these.

In assignment 3 we agreed that we prefer to use the **`while`** loop in the *Run* as opposed to writing our own **`while`** loop in the `act` method. In the previous assignment we made an exception to this agreement. There you called your methods by right-clicking. However, if you want to make a game, then you will have to make use of the `act`. In this assignment we will see that you can use instance variables to write your programs without a **`while`** loop in the `act` method. We define the `act` method so that it executes just one step of our algorithm. Then, when you press the *Run*, our entire program will be executed.

So, throughout this assignment you may not write **`while`** loops in the `act` method.

2 Learning objectives

After completing this assignment, you will be able to:

- explain, in your own words, what **instance variables** are used for;
- explain, in your own words, what the difference is between a local variable and an instance variable;
- explain, in your own words, what a **constructor** does;
- recognize a class's constructor;
- name and follow the steps that are necessary for using an instance variable;
- **initialize** instance variables;
- explain, in your own words, what role **`private`** and **`public`** play in information hiding and modularization;
- reason about the **visibility** of variables and methods;
- explain, in your own words, what **getter and setter methods** are used for.

3 Instructions

In this assignment you will carry on with your code from assignment 5. Therefore you will need a copy of the scenario which you saved after completing assignment 5, `Asgmt5_yourName`. To make a copy follow the next steps:

- Open your scenario from assignment 5, `Asgmt5_yourName`.
- In the Greenfoot menu at the top of the screen, select 'Scenario' and then 'Save As ...'.
- Check that the window opens in the folder where you want to save your work.
- Choose a file name containing your own name(s) and assignment number, for example: `Asgmt6_John`.

You will also need to answer some questions. Questions with an '(IN) ' must be handed 'IN'. For those you will need:

- pen and paper to draw flowcharts which must be handed in ('(IN) '),
- a document (for example, Word) open to type in the answers to the '(IN) ' questions.

The answers to the other questions (those that don't need to be handed in) you must discuss with your programming partner and then jot down a short answer on the assignment paper.

A note in advance: in this assignment you may only make changes to the `MyDodo` class and the `Egg` class.

All relevant material can be found, such as scenarios can be found at:
`https://www.cs.ru.nl/S.Smetsers/Greenfoot/Kandinsky/`

4 Theory

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`** `myNrOfEggsHatched` is private and can't be accessed or modified by another object. No one knows better than Mimi herself, how many eggs she has hatched. 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.

So, objects of other classes can call public accessor or mutator methods. 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. 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 increased by controlling which methods can or can't be called from outside the class.

In order to simplify things, so far we have made all methods public. 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.

Instance variables
Instance variables are an object's 'memory'. They contain information that an object stores or uses. An instance variable is a type of variable: just like any other variable is has a type and a name, such as **`int`** `myNrOfEggsHatched`. 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*). If another object needs to know what the value is, is should ask the object by calling the object's public accessor method.

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`** `myNrOfEggsHatched`. With this variable, MyDodo stores how many eggs she has hatched. 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`** `myNrOfEggsHatched` directly.

Initialization
An instance variable is given an initial value in the class's constructor (this concept 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 hatched any eggs yet. We indicate this in the constructor with: `myNrOfEggsHatched` = 0.

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, the lifetime of an instance variable is equal to that of the object.

Getter accessor method
Another object can use a **`public`** accessor method (a so-called *getter method*) to ask for the value of an object's **`private`** instance variable. Not every instance variable has a **`public`**

getter method. Getter methods are only written if they are useful, that is to say, if the values need to be shared with other objects.

Naming conventions of a getter method
How to name a getter method depends on the result which the method returns.

- If the result is a **boolean**, choose 'is....', for example **boolean** `isHatched( );`
- Otherwise, choose a name as 'get...', for example **int** `getNrOfEggsHatched( )`.

Here, the variable name is written instead of '...'.

Example code for a getter method
Because instance variables are **private**, other objects can't access its value. Another object can ask its value by using a getter method.

Below is an example of a **public** getter method for the **private** `myNrOfEggsHatched` variable. By adding this getter method, another object can call the method to ask how many eggs have been hatched:

```
public int getNrOfEggsHatched( ) {
    return myNrOfEggsHatched( );
}
```

This is a **public** method. Thus, another object can call this `getNrOfEggsHatched( )` method to ask how many eggs have been hatched. Note that another object can't change the value of the variable by calling this method, another object can only ask its value.

Setter mutator method
The value of a **private** instance variable can be changed by another object using a **public** mutator method (a so-called *setter method*).

Tip: Use setter methods sparingly. Only add them to a class if they are really needed. That keeps the variables better protected, and thus it's safer.

Example code for a setter method:
For this example, assume `MyDodo` has an instance variable **private int** `eggsToHatch` with which Mimi can keep track of how many eggs she has to hatch.

Only she knows how many eggs she has to hatch (the variable is **private**). A snake may come along and eat an egg. In that case, Mimi can get a message from the snake telling her there is one less egg to hatch. Mimi then changes the value of `myEggsToHatch` herself (she decreases it by one). We don't trust the snake to change the value of `myEggsToHatch` directly because he would immediately set its value to 0. The snake's intention is to eat all the eggs; he doesn't want Mimi to hatch any eggs at all. So, we don't trust the snake with the variable `myEggsToHatch` and keep it **private**.

The following setter method can be called by another object in order to change one of `MyDodo`'s variables:

```
public void setOneEggLessToHatch( ) {
    myEggsToHatch--; // decrease value by one
}
```

Because the method `setOneEggLessToHatch( )` is **public**, another object can use `setOneEggLessToHatch( )` to decrease the value of the variable `myEggsToHatch` by only one. Another object is now given the 'power' to change the value by one, but not to immediately set it to 0. Obviously, calling `setOneEggLessToHatch()` many times will lead `myEggsToHatch` to equal 0. However, we could keep track of how often another object calls a method and

take adequate measures. In the case of the egg-eating-snake, after he eats 5 eggs we could set a snake trap. Whatever the case, the variable can not be changed without MyDodo knowing about it.

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

- **Declare** the instance variable: indicate the name, type and visibility (as a rule we keep this **`private`**);
- **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;
- (optional) Write a **getter accessor method** so that an object from another class can ask the variable's value;
- (optional) Write a **setter mutator method** so that an object from another class can modify the variable's value.
- **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.

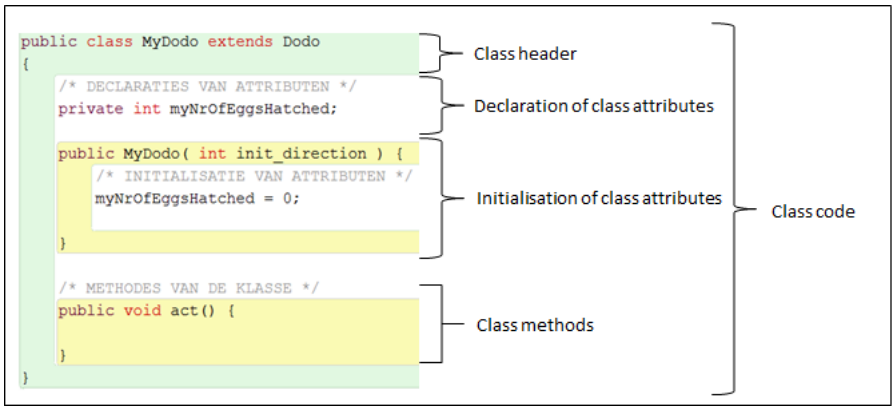
Note: If you don't want other objects to be able to ask or change a variable's value, then don't make getter or setter methods.

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. 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`**.

In Greenfoot 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();`

Example:
In the editor, open the code for the `MyDodo` class.



This class has:

- one instance variable: **`private int`** `myNrOfEggsHatched`;
- one constructor **`public`** `MyDodo (...) ;`
- any other methods **`public void`** `act( )`.

5 Exercises

5.1 (Instance) variables

1. In assignment 5, exercise 5.1.2 'Turn to the East' you filled in a table with values for the variable **`int`** `myDirection` and the result of **`int`** `getDirection()`. Meanwhile you have learnt about visibility and getter methods. Investigate whether **`int`** `myDirection` and **`int`** `getDirection()` are **`private`** or **`public`** and what each is used for (i.e. explain the relationship between the two).
2. How can you recognize a constructor?
3. (IN) Explain in your own words what the difference is between an instance variable and a local variable.
4. What are getter- and setter methods used for?

5.2 Count eggs until the fence

We will make Mimi walk to a fence. Meanwhile she counts the number of eggs she finds. To do this we make use of instance variables. By using instance variables we can keep count of the eggs with each step, and thus we can make use of the **`while`** loop in the *Run*.

1. Open 'world_layEggTrailUntilFence'.
2. Come up with an appropriate name for an instance variable that keeps track of the number of eggs found.
3. Come up with an algorithm which makes Mimi walk to the fence and count how many eggs she finds on the way there. Each time she finds an egg, the number of eggs found should be printed in the console.
4. Draw the corresponding flowchart.

5. We will now write the corresponding method **public void** `walkToFenceAndCountEggs()`.
   Follow these steps:
   - Open `MyDodo`'s code.
   - Compare the code in the editor with that shown below.
   - Add the missing lines to `MyDodo`'s code (i.e. also the comments in capital letters).

```
public class MyDodo extends Dodo {
    // DECLARE THE INSTANCE VARIABLE HERE

    public MyDodo (int init_Direction) {
        // INITIALIZE THE INSTANCE VARIABLE HERE

    }
    public void walkToFenceAndCountEggs() {
        if (!fenceAhead() ) {
            move();
            if( foundEgg() ) {
                // INCREASE THE INSTANCE VARIABLE BY 1


                // show console
                System.out.println("I found " + VARIABLE_NAME + " eggs!");
            }
        }

    }
}
```

   - In front of the comments in capital letters, write real code which does just that.
6. Add your own comments above the method explaining what the method does.
7. Go to the code of the `act` method. Make a call to the **void** `walkToFenceAndCountEggs()` method from within the `act( )` method.
8. Compile and test your method using the *Run* button.
9. What happens? Tip: A console opens, this may be hidden behind another Greenfoot window.
10. Does the program do what you expect?

We have now seen that Mimi can use an instance variable to keep track of how many eggs she has found. The instance variable is initialized (in the constructor) when the Mimi object is created. Because an instance variable exists as long as the object exists, it can be used to store information for a longer period of time (than a local variable).

In assignment 5 we used a **while** loop and a local variable to count eggs. You can't just replace the **while** loop by the **while** in the *Run*; it won't work the same. The `act` method is called over and over again, and each time it is called, the local variable is created and initialized again (in this case, it will be set to 0 every time the `act` method is called). Mimi won't be able to keep count. What **does** work is using instance variables. These are only initialized (in this case, set to 0) when the object is created. After that they keep their value, even if the `act` is called repeatedly.

Thus, by using instance variables instead of local variables, we don't need to use a **while** loop in the `act` anymore, but can merely use the (**while** in the) *Run* instead.

5.3 Keeping track of the number of steps taken

Mimi is not very intelligent. So far, she can keep track of how many eggs she has hatched, but that's it. We are going to make Mimi a little smarter by helping her keep track of how many steps she has taken too.

1. Open `MyDodo`'s code.
2. Find the place in the code where instance variables are declared for this class.
   (a) Find the declaration of the instance variable **`int`** `myNrOfEggsHatched`.
   (b) What type does the instance variable have?
   (c) The variable is **`private`**, what does this mean?
   (d) Where is the first place that the variable gets a value? What value is this?
   (e) When is this value modified?
   (f) Meanwhile, did you find the constructor? How do you recognize the constructor?
3. We will now add a new instance variable `myNrOfStepsTaken`. Follow the next steps:
   (a) **Declare** the instance variable: In an appropriate place, add a **`private`** integer instance variable `myNrOfStepsTaken`.
   (b) **Initialize** the instance variable: What would be an appropriate initial value for the instance variable? In other words, if a new Mimi is placed in the world, what should the value of `myNrOfStepsTaken` be? Give the instance variable the correct initial value. Tip: Do this in the constructor.
   (c) Write a **getter method**: Now that Mimi has a variable to keep track of how steps she has taken, we should add a getter method. Other objects can use this to ask Mimi how many steps she has taken. Come up with an appropriate name for this method. Implement and test the method. Tip: see the theory block above for naming conventions and example code.
   (d) **Modify** its value: Each time Mimi takes a step, the value of `myNrOfStepsTaken` should be raised by one.
      i. Where is the best place to do this? It could be done after each call of `move`, or after each call of the `step` method. Explain why the best option is to increase the value after each call of `step`.
      ii. Find (all) the place(s) in `MyDodo`'s code where `step` is called.
      iii. Modify the code so that the value of `myNrOfStepsTaken` is modified after each call of the `step` method. Tip: To increase a variable's value by one, use `myNrOfStepsTaken++;`.
   (e) Test your changes.
4. Explain why, in this case, we don't write a setter method.

Calling a method from another class
When calling a method, there are two possible cases:

1. The method belongs to an object's **own** class (or a class that is inherited). In this case the object calls the method directly (as you have been doing all along).
2. The method belongs to a **different** class. To be able to call a method from another class, you first need to have an object from that class.

General explanation:
Consider an object from that class called `object`, and a method from `object`'s class called `method`. Assuming that the method has no parameters, use `object.method( );` to call the method. You can read the instruction as follows: `object` is asked to execute his method `method`.
Example of calling a method from a different class:
Imagine there is a `BlueEgg` object called `babyBlueEgg`. The `BlueEgg` class has a method **`int`** `getX( )` (inherited from `Actor`) which returns the egg's x-coordinate.

Mimi wants to know where the egg is (its coordinates). Mimi can ask `babyBlueEgg`'s x-coordinate by using: `babyBlueEgg.getX( );` . This call is done from Mimi's `MyDodo` class.

Note:
Tip: After typing the object's name followed by '.' (in the example above: '`babyBlueEgg.`'), use 'Ctrl+Space' to see a list of all the methods available for that object.

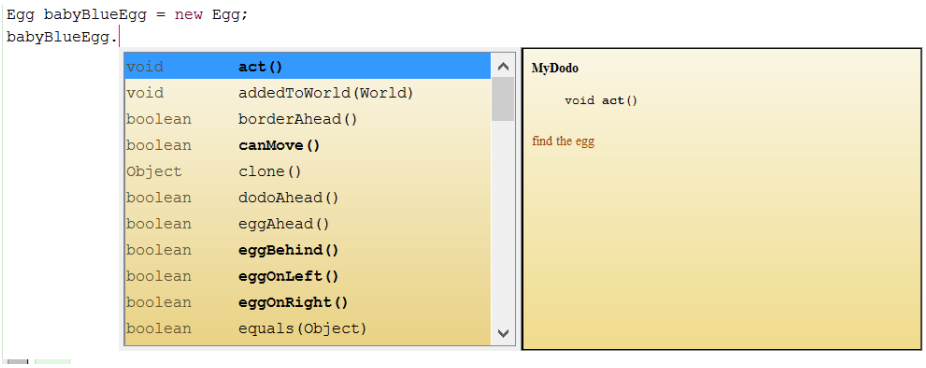


Figure 1: To see a list of all available methods, use: 'Ctrl+space'

5.4 To be hatched or not to be hatched

Once an egg is hatched, it disappears from the world. However, it would be more realistic if the egg wouldn't just disappear, but would be shown as a hatched egg. In this exercise you will learn how to change the way an object looks (depending on its state: hatched or not hatched).

Figure 2: A hatched egg

First we summarize which steps we will have to take. We will then modify the code together, step-by-step, in the following exercise.

- **Declare** an instance variable: The egg has to know whether it has been hatched or not. To do this we add an instance variable to the egg: **`private boolean`** `iAmHatched`.
- **Initialize** the instance variable to the correct initial value. When the egg is placed in the world for the very first time, it will be unhatched.
- Write a **getter method** to ask if the egg has been hatched or not (to get the value of the instance variable **`boolean`** `iAmHatched`): **`public boolean`** `isHatched( )`.

- Write a **setter method** to tell the egg when it has been hatched (to change the value of the instance variable **boolean** `iAmHatched`): **public void** `setHatched( )`.
- **Modify** its value.
  - In the setter method, the next has to be done:
    - change the **value of the instance variable**, so that `iAmHatched` is **true**;
    - change the **picture** of the egg by a picture of a hatched egg.
  - Calling getter and setter methods: Our eggs have become smarter. They know whether they have been hatched or not. But it is Mimi whom actually hatches them; she must tell the egg when it has been hatched. `MyDodo`'s code has to be modified so that Mimi calls the getter and setter methods when she wants to hatch an egg.
    - Before hatching an egg, Mimi needs to check if the egg hasn't already been hatched. Mimi asks the egg its state by calling the egg's getter method **boolean** `isHatched( )`.
    - Once Mimi has hatched the egg, she calls the egg's setter method `setHatched()` to tell the egg that it has been hatched.

    At first using getter and setter methods may seem a little odd. However, this is **the** object oriented way in which objects interact. Objects can do all sorts of things, but they don't do them spontaneously. They are told to do things by other objects.

We now modify the code, step-by-step. We start by making changes to the `Egg` class:

1. **Declare** the instance variable: Each `Egg` object keeps track of whether or not it has been hatched. In order to do this, we use the **boolean** instance variable. `iAmHatched`.
   (a) Open the code for the `Egg` class.
   (b) In an appropriate place, add a **private boolean** instance variable named `iAmHatched`.
2. **Initialize** the instance variable:
   (a) Decide what the initial value of the instance variable should be. So, when an egg is placed in the world (when a Dodo lays the egg), what should the value of **boolean** `iAmHatched` be? Explain.
   (b) Give the instance variable the correct initial value. Tip: do this in the constructor.

   The property "is hatched" has now been added to the class `Egg`. An egg's state can now be hatched or not.
3. We now add methods to either ask for or change the value of the `iAmHatched` instance variable. A getter and a setter method is added to the `Egg` class:
   (a) **Getter method**: In order for another object (for example, Mimi) to know if an egg has been hatched or not, the object (Mimi) must ask the egg what its state is (hatched or not hatched).
      i. Add a new method **public boolean** `isHatched( )` to the `Egg` class.
      ii. This method must return the value of `iAmHatched`.
      iii. Compile and test your method.
   (b) **Setter method**: In order for another object to change the state of the egg from 'is not hatched' to 'is hatched', the object must tell the egg that it has been hatched (if Mimi hatches the egg, then Mimi has to tell the egg to change its state to 'is hatched').
      i. Add a new method **public void** `setHatched( )` to the `Egg` class.
      ii. In this method, change the value of the instance variable **boolean** `iAmHatched`.

iii. After the egg has been hatched, we should also change its picture (instead of the blue egg, we want to show the red cracked egg). To do that, use the following code:

```
GreenfootImage hatched_egg = new GreenfootImage("egg_hatched.png");
setImage(hatched_egg);
```

Note: The pictures used in this scenario can be found in the 'images' folder. One of the pictures is called 'egg_hatched.png'. Using the two instructions above, the image is linked to the Greenfoot class. If you wish, you can find your own image on the internet. Copy the image to the images folder and then link it to the class by modifying the code above.

iv. The `setHatched( )` method is a mutator method. How can you tell?

v. Compile and test your code. By right-clicking you can call the `setHatched( )` method. Use 'Inspect' to view the values of the instance variable. Does the code work as expected?

In step 1 we gave the `Egg` a **`private`** instance variable called `iAmHatched`. In step 2 we gave the variable its initial value. In step 3 we wrote **`public`** getter and setter methods for that variable. Using these methods, other objects (such as Mimi) can ask the value of the variable (i.e. get), or ask the egg to change (i.e. set) it (to also change its picture).

4. **Modify**. If Mimi wants to hatch an egg, she must call the appropriate getter and setter methods. We will now change the code so that Mimi tells the egg that it has been hatched.

   (a) Open the `MyDodo` class. Look for the `hatchEgg( )` method. In your own words, describe what this method does with the egg.

   (b) First, we need to get a hold of the egg that we want to hatch. To get the egg object, use `Egg eggFoundToHatch = getEgg( );` . This instruction creates a local variable which holds an `Egg`. The `getEgg` stores the egg in the local variable. Now, Mimi can call any **`public`** `Egg` method for the Egg variable in `eggFoundToHatch`.
   In the method `hatchEgg( )`, add the statement `Egg eggFoundToHatch = getEgg( );` .

   (c) Before Mimi actually hatches the egg, she must first check if it has not been hatched already. Add this check. Tip: Let her call `eggFoundToHatch.isHatched( )` before she tries to hatch an egg.

   (d) Compile and test your code by right-clicking on either a blue egg or a golden egg and then selecting the method's name. What does `isHatched( )` return if the egg was already hatched? And what does it return if it wasn't yet hatched?

   (e) To change the state of the egg, Mimi must call the **`void`** `setHatched( )` setter method for the egg that she has found. Use `eggFoundToHatch.setHatched( );` to call the setter method.

   (f) Modify the method's comments.

   (g) Compile and test your code. What happens when you call `setHatched( )` (by right-clicking) on an egg that has not yet been hatched? And what happens when you do the same with an egg that has been hatched?

We have now added an instance variable to store the state of an Egg (hatched or not hatched). We also added getter and setter methods so that another object (Mimi) can ask the state or tell the egg it has been hatched. The egg then changes its appearance accordingly (if hatched, to a red cracked egg).

5.5 Go to a location (using instance variables)

In assignment 5 exercise 5.1.5 'Go to a location' you wrote the method

```
public void goToLocation( int coordX, int coordY)
```

which sent Mimi to a certain location with coordinates (coordX, coordY). Your method uses a **while**. As explained in assignment 3, we want to avoid using **while**s in Greenfoot because Greenfoot has its own **while** loop in the `Run`.

Now that you know how to use instance variables, we can use them to get rid of the **while**. In this exercise you will rewrite the method so that it doesn't use a **while**. Instead, the steps will be repeated using the *Run*.

1. To refresh your memory, have a look back at the `goToLocation` method which you wrote in assignment 5 exercise 5.1.5 'Go to a location'.
2. Declare two new instance variables in the `MyDodo` class: **private int** `myDestCoordX` and **private int** `myDestCoordY`.
3. Initially, Mimi must go to a certain location when `goToLocation` is called. Choose which coordinates you would like that to be (any valid coordinates are fine). Initialize the instance variables in the constructor accordingly.
4. Modify the code in the `goToLocation` method as follows:
   (a) Because we're using instance variables, we don't need to pass the coordinates as parameters anymore. Change the `goToLocation` method so that it doesn't take any parameters any more. Tip: Change this in the signature.
   (b) Replace the **while** by an **if** statement.
   (c) The sub-method which you wrote to check if Mimi has reached her destination must also be modified. Because we're using instance variables, we don't need to pass the coordinates as parameters anymore. Change the sub-method's signature so that it doesn't take any parameters any more. Also change this in sub-method's call in `goToLocation`.
   (d) Step through the method and verify that it is correct.
   (e) Write comments above the modified (sub)methods.
5. In the code, call the `goToLocation( )` method from within the `act` method.
6. Compile and test your program using *Run*.
7. Write a setter method:

   ```
   public void setDestination( int locationCoordX, int locationCoordY )
   ```

   which gives the variable `myDestCoordX` the value of the parameter `locationCoordX` and `myDestCoordY` the value of the parameter `locationCoordY`.

   Before setting the `myDestCoordX` and `myDestCoordY` values, it is useful to check if the given coordinates (in the parameters) are valid (i.e. they are inside the world).
8. By right-clicking, call the `setDestination` method. Then press *Run*. What happens?
9. Now download and open the file `ActForAssignment6.txt`. After opening the `Madagaskar` class as well, copy the `act` method to this class. You may choose yourself where to put this method. Directly below the constructor, for example, is an excellent place. As you can see, `act` contains a call to `setDestination`. Compile and test your scenario again using *Run*. What happens now? While the program is still running, click on any location in the world. What happens? Can you explain why this happens?

5.6 (IN) Calculating a fenced area (using instance variables) (A)

In assignment 4 exercise 5.3 'Walk around a fenced area', you taught Mimi how to walk around a fenced area. This time we want her to calculate the area enclosed by the fencing. The method you write must use instance variables and the *Run*.

1. Open the 'world_calculateFencedArea' world.
2. Come up with two instance variables for Mimi to keep track of the width and height of the fenced area.
3. Follow the same steps as in the previous exercises:
   - Declare initialize the instance variable appropriately;
   - Write getter and setter methods where appropriate;
   - Add a new method with which Mimi walks around the fenced area, modifying the value of the instance variables accordingly.
4. Test your code.
5. When she is done walking around the perimeter, show a console with the following text: "The size of the fenced area is xx". Instead of 'xx', print the calculated area.
6. Test your code again.

6 Summary

You can now:

- use instance variables to store values throughout the entire program;
- explain what a constructor is;
- replace a **`while`** loop in the `act`, by an **`if`** statement combined with instance variables.

7 Saving your work

You have just finished the sixth 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 ...'.

8 Handing in

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

1. The (Word) document with your answers to be handed in (answers to the '(IN) ' questions)
2. The MyDodo.java file
3. The Egg.java file