Assignment 1: Meet Dodo

– Algorithmic Thinking and Structured Programming (in Greenfoot) –

1 Introduction

In this series of assignments (*Algorithmic thinking and structured programming (in Greenfoot)*) you will learn the basics of Object Oriented programming, and (hopefully will) be having fun in the process. After completion, you will be able to program in Java, in addition to being able to use Greenfoot and Java libraries. With these skills, you will also be able to quickly learn other Object Oriented languages. You will also learn how to use and adapt existing code where necessary. Reuse of existing code is one of Java's strengths because it allows you to make more complex programs in less time.

You will program in the Java language. For the assignments we use the *Greenfoot* programming environment. This is an environment that allows you to write code without first having to learn (the theory of) Java in full-depth. In such a manner you can gradually learn Java while you're actually using it to make stuff.

You will learn the basic building blocks that are used for algorithms and programs. We'll start with simple exercises like making a Dodo walk and turn. Throughout the course, you will elaborate on this. A Dodo (a bird which is now extinct) was never very intelligent, however you will teach it to complete complex tasks, such as independently collecting eggs scattered throughout her world or walking through a maze. One of the final tasks is a competition with your classmates: who can make the smartest Dodo?

If you want to learn how to play chess, you have to know the rules. These rules tell you what you can and what you can't do. But if you want to play chess **well** and **enjoy** playing the game, then merely knowing the rules is not enough. You also have to learn what a good move is. On internet you can find numerous Java tutorials and manuals. Those teach you the rules, but little about how to write a program **well**. In this course you will learn how to systematically deal with a programming task. You will learn to analyze problems and develop an appropriate solution in a structured and organized manner. The goal is to teach you to program well, and let you **enjoy** doing so.

2 Learning objectives

After completing this assignment, you will be able to:

- find your way around the Greenfoot programming environment;
- explain what the Greenfoot *Run* and *Act* buttons do;
- invoke methods and analyze their effect;
- in your own words, describe the difference between a **mutator method** and an **accessor method**;
- identify and list the methods of a **class**;

- use a **class diagram** to identify classes;
- use a **class diagram** to identify **subclasses**;
- identify which methods an object **inherits** from another class;
- explain what an **instance object** is;
- identify properties of an object's **state**;
- apply naming conventions to methods and parameters;
- in your own words, explain what a **type** is;
- describe what the types **`int`**, `String`, **`void`**, and **`boolean`** are;
- given a **signature**, identify a method's **result type** and **parameter(type)s**;
- describe the role of a method's result type and parameter(s);
- describe the relationship between an **algorithm**, a **flowchart** and **program code** in your own words;
- incrementally make code modifications and test these;
- recognize and add **comment** to code;
- modify a method and then compile, run, and test it;
- recognize and interpret **syntax errors**.

3 Instructions

For this assignment you will need:

- Greenfoot: Instructions for installing and opening the Greenfoot environment are given further on;
- scenario **'DodoScenario1'**: to be downloaded from Magister;
- a document (for example, Word) for answering the '(IN) ' questions which are to be handed in.

Throughout the assignment you will also need to answer some questions. Questions with an '(IN) ' must be handed 'IN'. Make sure you have your document open to type in the answers to those questions. For the other questions (those that don't need to be handed in), discuss the answer with your programming partner and jot down a short answer on the assignment paper.

4 Finding your way in Greenfoot

4.1 The world

This is our Dodo. Her name is Mimi and she belongs to the *Dodo* family.

Figure 1: Mimi the Dodo

Mimi lives in a (grid)world that consists of 12 by 12 squares. This world is bounded, she can't get out. Her sole goal in life is to lay and hatch eggs. The picture below shows Mimi in her world, facing to the right. Before her lies an egg. There are also several fences in the world.

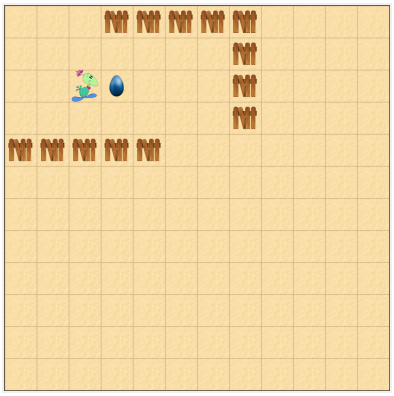


Figure 2: Mimi's world

Mimi is a very well-behaved Dodo. She does exactly what she is told to do, and she always gives honest answers to your questions (she never lies). By calling *methods*, you can tell her to do something or ask her questions.

You can let Mimi do things, such as:

`move`	move one square ahead
`hatchEgg`	hatch an egg
`jump`	jump several squares ahead

You can also ask Mimi a question, such as:

`canMove`	Can you take a step forward?
`getNrOfEggsHatched`	How many eggs have you hatched?

5 Getting started with Greenfoot

5.1 Starting Greenfoot

We're going to play around with Mimi in the Greenfoot environment. To do this, you must download and install Greenfoot (if it has not already been done for you) and open the given scenario.

Choose a folder:

- Decide where you will save all your work. If you wish, you can use a USB-stick for this.

Download:

- Go to Magister and download the scenario files belonging to this assignment: Copy the given *zip*-file with initial scenario to the folder which you have chosen.
- Unzip the *zip*-file to that location (right-click on the file and choose *Extract All...*).

Open the Greenfoot environment:

- Open a file browser and navigate to the USB-stick.
- Double click on the *Greenfoot.exe* file. The programming environment should start now.

Open the scenario:

- Select 'Scenario' in the main menu, and then 'Open'.
- Navigate to the **'DodoScenario1'** scenario (which you downloaden en unzipped) and chose 'Open'.

Rename the scenario:

- 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: `Asgmt1_John`.

Compiling the scenario:

- Click on the 'Compile' button on the bottom right.

You should see the following:

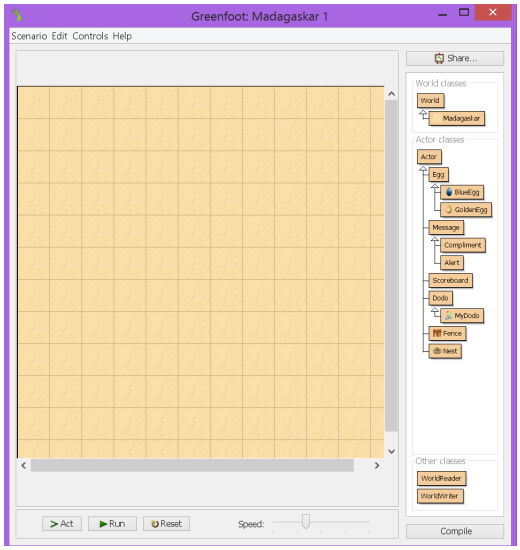


Figure 3: What you see when you open the 'DodoScenario1' scenario

5.2 Creating objects

Objects
A world consists of objects (also known as *class instances*, or simply *instances*).
By selecting **`new`** `MyDodo ()` you can create a new object of `MyDodo`. The world below shows one `MyDodo` object and three `Egg` objects.

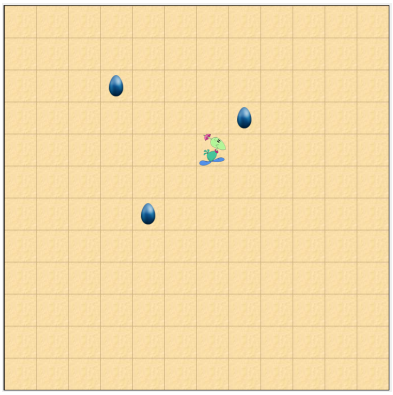


Figure 4: Scenario with one `MyDodo` object and three `Egg` objects

In our assignments we have one lead actress, a `MyDodo` object. She is an instance of `MyDodo`. We will call her Mimi.

We're going to make the scenario as shown in figure 4:

1. Right-click on *MyDodo*, on the far right-hand of the screen.
2. Select **`new`** `MyDodo( )` to create a new `MyDodo` object.
3. Drag the object, our `MyDodo` called Mimi, to a random position in the world.
4. In the same manner, place three `BlueEgg`s in the world. Tip: hold down the *Shift* button on your keyboard to place multiple blue eggs.

5.3 The world in motion

Greenfoot has a few buttons at the bottom of the window.

1. Click on the *Act* button (in the bottom of the window). What happens?
2. What happens when you press *Act* again? And again? And again?
3. Use your mouse to move Mimi and continue to experiment with the *Act*. Make sure you try lots of different situations, for example, with Mimi at the edge of the world facing outwards (with her beak pointing towards the edge).
4. (IN) Explain in your own words, very precisely, what the *Act* button does.
5. Press the *Run* button. What happens?
6. What is the connection between the *Act* and the *Run* button?
7. What does the *Reset* button do? Try it.

5.4 Methods

Methods and their results
There are two types of methods, each with an own purpose:

1. *Mutator methods*: These are commands which make an object do something. It changes the state of an object. You can recognize a mutator method by its result: **`void`**. For example **`void`** `act( )`.

2. *Accessor methods*: These are questions that provide information about (the state of) an object. For example, **`int`** `getNrOfEggsHatched()` which returns an **`int`** (an integer, or whole number) indicating how many eggs Mimi has hatched. Another example is **`boolean`** `canMove( )` which returns a **`boolean`** (either **`true`** or **`false`**), indicating whether Mimi can take a step forwards or not. Access methods only provide information about an object, they don't *do* anything with an object: the object's state remains unchanged (for example, Dodo doesn't also move to another position).

5.4.1 Exploring the mutator method

Our MyDodo has been programmed to do several things. You can let her do things (give her commands) by calling her *mutator methods*.

1. Right-click on Mimi. You will see a list of all the things Mimi can do (in other words, which methods you can call), for example `act( )`, `move( )`, and `hatchEgg( )`.

2. Call the method `act( )`. What happens?

3. Look at your own explanation about what the *Act* button did (see exercise 5.3 part 4). Do you think there is a difference between pressing the *Act* button and calling the `act( )` method?

4. Drag an egg into the world. Drag Mimi on top of the egg. Now right-click on Mimi (be careful not to click on the egg). Call Mimi's method **`void`** `hatchEgg( )`. What happens?

5.4.2 Exploring the accessor method

By using *accessor methods* you can get Mimi to answer questions about herself.

1. As you may or may not know, a Dodo can't fly. But a Dodo can walk. That is, Mimi can't step out of her world, but as long as she stays in her world, she can take a step forward. So, before she takes a step, she has to check if it is possible. Place Mimi somewhere in the middle of the world. If you were to ask Mimi if she can take a step, what answer would you expect her to give?

2. Call the **`boolean`** `canMove( )` method. A dialogue box appears, as in the picture below. It says **`true`**. What does that answer mean? Is that the same answer as you had expected (look back at 5.4.2 part 1)?

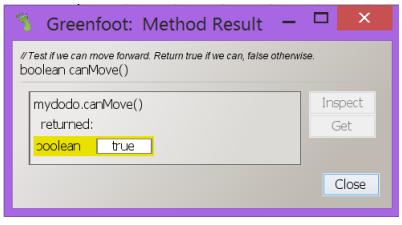


Figure 5: Dialogue box for `canMove( )`

3. Move Mimi to the edge of the world, facing outwards, as shown in figure 6. Call **`boolean`** `canMove( )` again. What result does it give?

4. (IN) What do you think **`boolean`** means?

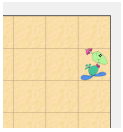


Figure 6: Dodo at the edge of the world, facing outwards

5. Now call the method **`int`** `getNrOfEggsHatched()`. What result does it give? What does **`int`** mean?

6. You can use the **`int`** `getNrOfEggsHatched()` method to see how many eggs MyDodo has hatched. The result is an **`int`** (a whole number). Can you create a situation in which the method **`int`** `getNrOfEggsHatched()` returns the value 3? (In other words, can you make Dodo hatch 3 eggs?). Tip: first use `hatchEgg()`.

5.5 Inheritance

Classes
Every object belongs to a class. As such, Mimi is an *instance* of the `MyDodo` class.

Class diagram:
In the class diagram you can see that `MyDodo` belongs to the Dodo family. In Greenfoot the class diagram is shown on the right-hand side of the screen. Have a look at figure 7. The arrow in the diagram indicates an 'is-a' relationship: `MyDodo` 'is-a' `Dodo`. So, `MyDodo` belongs to the class `Dodo`.

Figure 7: Class diagram of Dodo

MyDodo's methods:
Mimi is a `MyDodo`. Mimi has methods that you can call. By right-clicking on Mimi you can see what a `MyDodo` (and thus Mimi) can do. For example `move()`.

Dodo's methods:
But Mimi (our `MyDodo`) is capable of doing much more. There are things that all Dodos, in general, can do. Mimi is a Dodo, so of course she can also do all the those things too!

Right-click on Mimi (in the world). At the top of the list you see 'inherited from Dodo'. If you click on that, then more methods will appear, such as `layEgg()`. These are methods which belong to the class `Dodo`, things that all Dodos can do. Because `MyDodo` is a `Dodo` (see the class diagram in figure 7), Dodo inherits (or gets) all these Dodo methods too. So, because `MyDodo` is a subclass of `Dodo`, a `MyDodo` so can perform all `Dodo` methods (in addition to its own `MyDodo` methods).

Inheritance:
By using *inheritance*, a new class can be introduced as an extension of an existing class. In such a situation, the existing class is called the *super class*, and the extended class is called the *subclass*. Because `MyDodo` a subclass of `Dodo`, a `MyDodo` can execute methods from both the `Dodo` class as well as its own `MyDodo` class.

Imagine that a new Dodo species were to be born: `IntelligentDodo`. This `IntelligentDodo` would be a subclass of `Dodo`. It can do anything a `Dodo` can do. You

would not have to describe that this new Dodo species can lay an egg, `layEgg( )`, because we already know that all `Dodo`s can lay eggs. With inheritance we only describe the *extra* things (i.e. methods) which that subclass can do (its intelligent behavior). This prevents redundant code and lots of extra work.

Speaking of which... which things can all Dodos do? We will now have a look at the *mutator methods*.

1. Name at least three methods which Mimi inherits from the `Dodo` class.
2. The method **`void`** `turnLeft( )` belongs to the `Dodo` class. Can Mimi execute that method? Try it.
3. (IN) Have a look at the class diagram. Which other 'is-a' relationships do you see? Name at least two.
4. Drag a second `MyDodo` into the world. Compare her methods to those that Mimi has (recall that Mimi is the Dodo which was already standing in the world). Do you see any differences?

5.6 States

The state of an object
You can place different objects in the world, such as a Dodo and an egg. You can also place several objects of the same class in the world at the same time. In the picture below you see three objects of the `Egg` class and one of the `MyDodo` class.

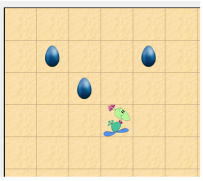


Figure 8: A world with several objects

All objects of the same class have the same methods, and thus have the same behavior. The three objects of `Egg` look identical and can do exactly the same things. Yet, they are different objects, or *instances*.

Just like you're different from the person sitting next to you, even though you are both people. The mere fact that you are each sitting on different chairs or had something different for breakfast, makes you different. The same is true in the Greenfoot world. Every object has its own state. For example, two objects can be standing on different coordinates, making their states different.

You can view the state of an object by right-clicking on the object, and then choosing 'Inspect'.

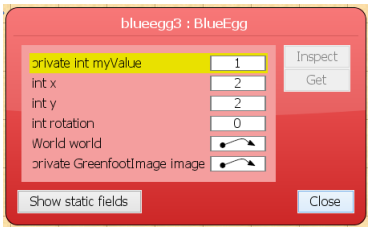


Figure 9: The state of an `Egg`-object with coordinates (2,2)

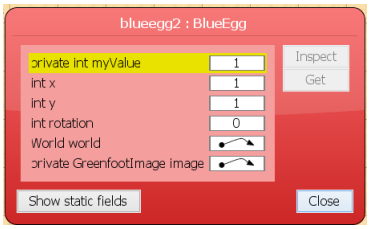


Figure 10: The state of an `Egg`-object with coordinates (1,1)

We will now have a look at the states of objects in more detail.

1. Drag Mimi to the top-left corner of the world.
2. Right-click on Mimi and select 'Inspect'. At which coordinates (**`int`** `x` and **`int`** `y`) is Mimi standing?
3. Drag Mimi to the top-right corner of the world. What are the coordinates of the square in the top-right corner?
4. Drag Mimi to any random position. What do you think the coordinates of that position are?
5. Check your answer using 'Inspect'. Was your answer correct?

5.7 Parameters and results

Results
In section 5.4 we saw that accessor methods give information about an object's state (they answers questions). Such an answer is called the *result* of a method. For example, **`int`** `getNrOfEggsHatched()` has an **`int`** (whole number) as its result.

Parameters
A method can also be given a certain value. This value, called a *parameter*, gives the method more information about a particular task. A method can have 0 or more parameters. Examples:

- **`void`** `jump (int distance)` has one parameter, `distance`, which tells the method how far it must jump. We can also tell that `distance` is a whole number, an **`int`**. By calling `jump (3);` Mimi will move forward 3 squares. By calling `jump (5);` Mimi will move forward 5 squares. The method `jump` needs this additional information in order to work. Otherwise, it would not know how many squares Mimi must move.
- **`void`** `move( )` has no parameters.

A method which needs parameters is more flexible than one that doesn't. Using the method **`void`** `jump( int distance )` Mimi can jump over different distances. On the other hand, using **`void`** `move( )`, Mimi can merely take one step.

Types
Parameters and results have a *type*. Examples of different types are:

Type	Meaning	Example
int	whole number	2
boolean	'true' or 'false'	**true**
String	text	"I lost my pen."
List	list	[1,2,3]

A type can also be a class, such as a List of Eggs.

The type indicates which sorts of values a parameter or result must have. It is not possible to call **void** jump(**int** distance) with no parameters because this method *requires* an **int** as a parameter.

We saw that you can use 'Inspect' to get information about Mimi's state. Accessor methods are used to return this information as a result of their method. Let's have a look at how this works.

1. Right-click on Mimi.
2. The Actor class has a method which returns the coordinates of an object. Can you find this method?

   Tip: MyDodo is a subclass of the Actor class, as you can see in the class diagram. In the list of 'inherited from Actor' methods, search for a method with an X. Do the same for Y.
3. (IN) What do you notice about the methods' names? Explain why some have a 'get' in their name? And others a 'set'? What does his mean? Tip: Have a look back at section 5.4.
4. What is the type of these methods' results? How can you determine that from what you clicked on?

Let's have a look at a few examples of methods with parameters.

1. Place Mimi somewhere in the middle of the world. Right-click on Mimi. Call the method jump(**int** distance). This method requires an **int** (whole number) as a parameter so that it knows how far Mimi must jump ('distance'). Type in a small number and see what happens.
2. Call the method jump(**int** distance) again. This time use a number larger than 11 as the argument. What happens? Can you explain why?
3. What happens when you type something in that is not an **int** (whole number), such as 2.5?
4. Have a look at the error message. It complains about 'incompatible types'. What that means is that you entered a different argument type than was expected. Namely, the method expects an **int** (whole number), but you typed in a **double** (decimal number).
5. What happens if you type a word as an argument, such as "two"?

Signature
By looking at the signature of a method, you can see which parameters, parameter types and result types belong to the method. We will explain what that means here.

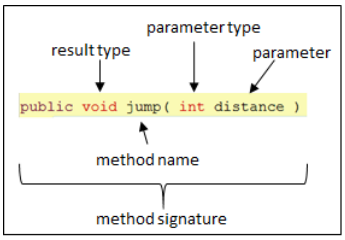


Figure 11: Signature of a method

- *Method name:* the method's name is indicated in front of the first parenthesis. In this example: `jump`.
- *Parameter:* the parameters that the method requires are indicated in between the parentheses '(' and ')'. In this example: `distance`.
- *Parameter type:* the parameter's type (such as **`int`**, **`boolean`**, or `String`) is indicated in front of the parameter's name. In this example: **`int`**.
- *Result type:* the result's type (zoals **`int`**, **`boolean`**, **`void`**) is indicated in front of the method's name. In this example the result type is **`void`**. This method doesn't return anything (it is a mutator method).
- *Method's signature:* together, all of the components above are called the signature. In this example: **`public void`** `jump (` **`int`** `distance`). (the meaning of **`public`** will be explained later)

The types in the signature indicate which types of values a parameter or result must have.

5.8 Describing behavior and reading, modifying, compiling and testing programs

We have now practiced invoking `MyDodo`'s methods. To get Mimi to do more exciting things, you must write your own new methods. Obviously, you must first learn to read code before you can write code. Also, you should be able to describe which changes you want to make, after which you can actually make modifications to existing code.

5.9 Describing behavior

Algorithm
An *algorithm* is a very precisely defined set of instructions. *Program code* is an algorithm written specifically for a computer. It tells the computer exactly, step by step, what it should do. If a set of instructions has been described precisely enough, someone else should be able to follow the steps precisely as you meant it. For the same problem (*initial situation*) the instructions will lead to exactly the same outcome (*final situation*). It is similar to a recipe. The difference with a recipe is that an algorithm is more precise. For example, a recipe might say "stir until smooth". However, each time you (or someone else) makes that dessert it may end up differently, one time being a bit smoother than the next. That is not permitted with an algorithm. For each step it must be precisely clear what it exactly means. A step that can be interpreted in different ways is called *ambiguous*. We try to avoid ambiguity. The result must be exactly the same every time.

Flowchart
An algorithm can be visually represented in a flowchart.
As an example we will look at the algorithm of `MyDodo`'s `act( )` method. We have already had a look at this method in exercise 5.4.1 part 2.

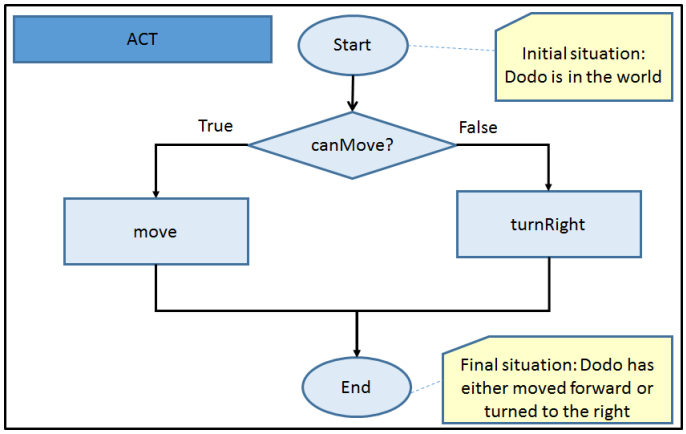


Figure 12: Flowchart of `MyDodo`'s `act( )` method

The flowchart explained:

- The name of the flowchart can be found in the top-left corner, namely *Act*.
- The initial situation is described in the note block in the top-right corner.
- The final situation is described in the note block in the bottom-right corner.
- You begin at 'Start' at the top.
- Follow the arrow to the diamond. A diamond means that a decision (selection) had to be made, depending on the outcome of 'canMove?' another path can be followed, namely:
  - If 'canMove?' is 'True', then the 'True'-arrow to the left is followed.
  - If 'canMove?' is 'False', then the 'False'-arrow to the right is followed.

  This behavior is consistent with that of an **`if`** .. `then` .. **`else`** statement in the code.
- In the case of a rectangle, a method is called. So, depending on which arrow has been followed the 'move' or the 'turnRight' method will be executed.
- When the 'End' is reached, then the *Act* method is finished.

5.9.1 Reading code

Reading code
Once again we review `MyDodo`'s `act( )` method as an example.

```
/**
 * Go to the edge of the world and
 * walk along the border
```

```
 */
public void act( ) {
    if ( canMove( ) ) {
        move( );
    } else {
        turnRight( );
    }
}
```

Explanation of the code:

- *Comment*: the text in between /** and */ (shown in blue in the Greenfoot editor) are comments. Greenfoot/Java doesn't do anything with this. Programmers write comments in their code to help others understand their programs. But it can also be useful for yourself. For example, if you have not looked at your code for a week, you can briefly read the comments to see what the method does without having to extensively study the code itself. If the comments fit on just one line, you can also use //.
- *Signature*: this is **public void** `act( )`.
- *Access modifier*: **public**. What this means and which other possibilities there are, will be explained later.
- *Method name*: `act`.
- *Result type*: **void**. In chapter 5.4 we saw that **void** as a result type means that this is a mutator method which makes the object do something (as opposed to an accessor method which yields information about the object's state).
- *Body*: this is the part between the first curly bracket '{' and its corresponding curly bracket '}'. This is the code that is executed when the `act( )` method is called, in other words, what the method actually does. In this example you see a *conditional expression*: the **if** .. `then` .. **else** in the `act( )`. It works like this:
  - A condition is stated in between the parentheses '(' and ')'. Firstly, the condition is checked.
  - If the condition is **true**, then the code directly following the condition is executed.
  - If the condition is **false**, then the code after the **else** is executed.

  So if you press the *Act* button or select **void** `act( )` by right-clicking, then:
  - first the condition is checked: the method `canMove( )` is called to check whether or not Dodo can take a step.
  - If `canMove( )` is **true**, then the `move( )` method is called and Dodo takes a step forward.
  - If `canMove( )` is **false**, then the `turnRight( )` method is called and Dodo turns (90 degrees) to the right.

We will now look at the corresponding code in Greenfoot:

1. Right-click `MyDodo` in the class diagram and select 'Open editor'.
2. Find the `act( )` method. Tip: use Ctrl+F.
3. Is the code the same as in the theory block above?

4. In exercise 5.4.1 part 2 you analyzed what the `act( )` method did. The theory block above explains precisely what the `act( )` does. Does the explanation above correspond with what you thought?

5.9.2 Code and its corresponding flowchart

We will now look at the `canMove( )` method.

1. Open `MyDodo`'s code in the editor and find the **`boolean`** `canMove( )` method.
2. For each line, explain exactly what the code means. Note: The symbol '!' used in the code is read as 'not' (*negation*).
3. Have a look at the flowchart in figure 13. The flowchart visualizes the algorithm in the **`boolean`** `canMove( )` method. Note:
   - The diamond indicates a decision (*conditional expression*).
   - If the condition in the diamond is 'True', then the left arrow is followed (and those steps are executed).
   - Otherwise (the condition is 'False'), then the right arrow is followed.

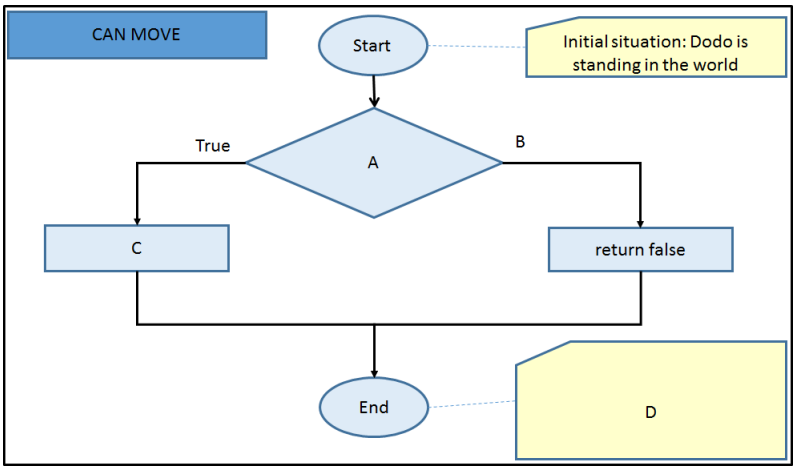


Figure 13: Stroomdiagram `canMove( )` in MyDodo

4. (IN) Fill in the correct text for A, B, C en D in the flowchart.

5.9.3 Adding comments to the code

We will now look at the `move( )` method.

1. In the code, find `MyDodo`'s `move( )` method.
2. (IN) What is this method's signature?
3. (IN) What is this method's result type?
4. Have a look at the method's body, in other words, the code in between the first curly bracket '{' and its corresponding second curly bracket '}'.
5. The method has an **`if`** .. `then` .. **`else`** statement. Depending on whether `canMove( )` is 'True' or 'False' something different will happen. Drag Mimi to different positions in the world and, by right-clicking on Mimi, call `move( )` method. Try to simulate both situations (for 'True' and for 'False'). What exactly happens?

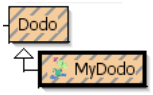
6. Now go back to the code and read the comments above the method (the part between /* and */). This doesn't properly describe what the method does. Namely, much more happens. Change the comments so that it does a better job describing what the method does.

5.9.4 Compiling, Running and Testing

Changes
Always make changes incrementally. A typo is easily made (for example, by forgetting to type a ';'). For this reason it is a good idea to compile, run and test every (small) change to the code.

The class diagram shows when any code changes have been made. When the rectangle of a class is shaded gray it means that the code in that class has been modified and must be compiled (again).

After each code change:



1. *Compile*: press the *Compile* button on the bottom-right.
2. *Run*: press the *Run* button.
3. *Test*: check if the method does what you expect. Call the method by right-clicking on Mimi and then choosing the method, or by pressing the *Act* button. Compare the initial and final situations.

Tip: Make a habit of always following these steps after **every** code change. That way, if you make a mistake, you can quickly find what the problem is. If you make many changes in one go without testing in between, then debugging (detecting errors in your code) can become a time-consuming and very frustrating task!

You have just added comments to the `move( )` method. This shouldn't have changed the behavior of the program. However, it is recognized as an modification.

1. Close the editor.
2. How do you recognize which class has been changed?
3. Press the *Compile* button on the bottom-right to recompile all the classes.
4. If necessary, fix any problems that the compiler reports.
5. *Run* and test the program. Does the `act( )` method still do what you expect?

5.9.5 Adding a new method

Naming conventions
Methods and parameters have names. In Java, there are general conventions (or agreements) on how to choose a name. By following these conventions, your code becomes easier to read and understand for others.

A method's name:

- is meaningful: it is consistent with what the method does
- is in the form of a command: it consists of one or more verbs

- consists of letters and numbers: it does not consist of spaces, commas, or other "strange" characters (with the exception of '_')
- is written in lowerCaseCamel: starts with a lowercase letter, and each subsequent word starts with a capital letter
- for example: `canMove`

A parameter's name:

- is meaningful: it is consistent with what the parameter means
- consists of one or more nouns
- consists of letters and numbers: it does not consist of spaces, commas, or other "strange" characters (with the exception of '_')
- is written in lowerCaseCamel: starts with a lowercase letter, and each subsequent word starts with a capital letter
- for example: `nrOfEggs`

See `http://google-styleguide.googlecode.com/svn/trunk/javaguide.html` for a complete overview of style and naming conventions.

We are now going to add a new method to `MyDodo`.

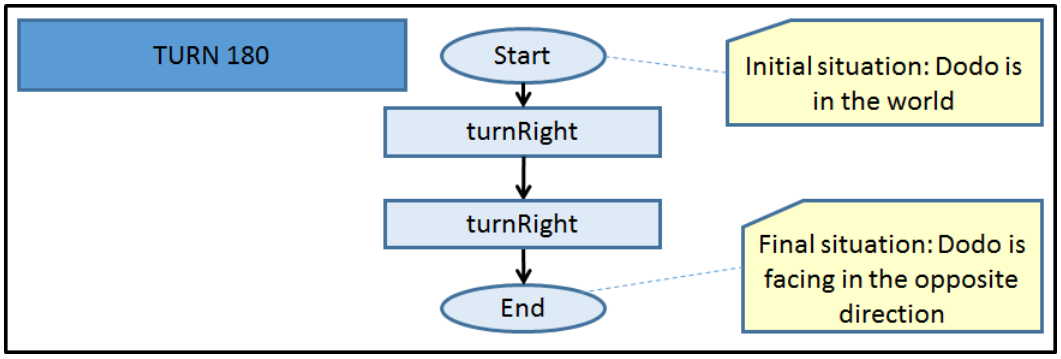


Figure 14: Flowchart TURN180

1. Have a look at the flowchart in figure 14.
2. Open the `MyDodo` class code in the editor.
3. We are going to add a new method to this class. Type the following code in the bottom of the editor screen. Note: do this before the last '}', otherwise the method will fall outside of the class and the compiler will complain.

```
public void turn180( ) {
    turnRight( );
    turnRight( );
}
```

4. What do you expect this method to do?
5. Compile the code. If necessary fix any mistakes.

6. Call your new method by right-clicking on Mimi and selecting **void** `turn180( )`. Test if it works as expected. Also test if the other methods still work correctly. In other words, check if the program as a whole still works as expected.
7. Go back to your code. Add a comment above the `turn180( )` method in which you briefly explain what the method does.
8. Once again you have made a change. Compile again and if necessary fix any mistakes.
9. By right-clicking, test again if the method still works as expected.

5.9.6 Calling a method in `act`

Calling a method in `act`

By calling a method in `act( )` it will be run each time when the *Act* button or the *Run* button is pressed. The difference between clicking on the *Act* button and clicking on the *Run* button is that in the first case the `act( )` method is called only once, while in the latter the `act( )` method is called repeatedly.

To call a method in `act`, you need to make a change to the **void** `act( )` method:

```
public void act( ) {
    methodName( );
}
```

1. Open the `MyDodo` code in the editor and find the **void** `act( )` method.
2. Remove the code in between the curly brackets '{' and '}'.
3. (IN) Call the method **void** `turn180( )`. Have a look at the example above to see precisely how to call the method.
4. (IN) Also change the comments above the `act( )` so that it describes what the method now does.
5. Compile the code. Tip: Do you get an error message which you cannot solve? Then have a look at chapter 5.10.
6. Test your program using the *Act* button. Does the program still work as expected?
7. Run your program using the *Run* button.

5.10 Error messages

The compiler is very picky. Sometimes you may make a mistake in the code or forget something. The compiler will then complain about it. Its useful to be able to recognize some common mistakes and error messages so that you can easily find the problem and fix it. Let's have a look at a few.

1. Open the `MyDodo` class in the editor. Find the `act( )` method. Delete the semi-colon ';' behind `turnRight( );`. Close the editor and compile. Which error message do you get at the bottom of the screen?
2. Restore the semi-colon ';'. Recompile. It should compile without a problem.

3. Test if the program still works as expected.
4. Change the spelling of `turn180( )` and recompile. Which error message do you get?
5. Restore the spelling mistake. Compile and test again.
6. Change `turn180( )` into `turn180`(5).
7. (IN) Compile. Which error message do you get? What does the error message mean?
8. Remove the 5. Press the 'Compile' button at the top of the editor. This compiles only the class shown in the editor. What message do you get at the bottom of the editor screen? What does this mean?
9. Test if the program still works as expected.

Syntax errors
If you type in something which the compiler does not understand, this is called a *syntax error*. Some common mistakes are:

Mistake	**Compiler error message**
missing ';' at the end of the line	';' expected
missing '( )' in the header	'(' expected
missing '{' at beginning of body	';' expected
missing '}' at end of body	illegal start of expression
missing '( )' in method call	not a statement
typo (watch out for capital/lowercase)	cannot find symbol
wrong parameter type used	method cannot be applied to given types
no parameter used	method cannot be applied to given types
wrong return type	incompatible types: unexpected return value

Logical errors
Besides syntax errors, there are also other types of programming errors. The compiler will not complain, but the program does not do what is expected. This is called a *logical error*. Such an error can be very difficult to find, and can be very frustrating. Searching for such an error is called *debugging*. Because prevention is beter than curing, we will discuss good programming practices in the next assignment. By using a structured approach to programming you can significantly reduce the chance of these types of errors slipping into your code.

6 Summary

In this assignment you have become acquainted with Greenfoot. You can now:

- find your way around the Greenfoot environment;
- explore and call an object's methods;
- explain the relationship between an algorithm, its flowchart and its code;
- find the code belonging to methods in the Greenfoot editor;
- incrementally make changes to the code;
- add a method, compile, run and test;
- recognize error messages.

6.1 Diagnostic test

1. Explain in your own words what a result type is. Give an example.
2. Complete the table below.

Method	Result type	Parameter type	Belongs to class	To be used by object
`getNrOfEggsHatched()`		not applicable	MyDodo	MyDodo
`canMove()`	boolean			
`jump()`				
`layEgg()`	void			
`turnLeft()`				Dodo and MyDodo
`canMove()`			MyDodo	
`getX()`				

3. Below are some names for a method. According to the naming conventions for method names, which one is the best?
   - `walk_to_egg`
   - `eggWalker`
   - `WalkToEgg`
   - `walkToEgg`
   - `WALK_TO_EGG`
4. Below are some names for a parameter indicating the number of eggs. According to the naming conventions for parameter names, which one is the best?
   - `nr_of_eggs`
   - `EggCounter`
   - `nrOfEggs`
   - `eggs`
   - `NR_OF_EGGS`
5. What are the coordinates in the bottom-left corner?
6. Here is code for a new method:

```
public A moveDown ( ){
    turnRight( );
    move( );
    B
}
```

The corresponding flowchart is:

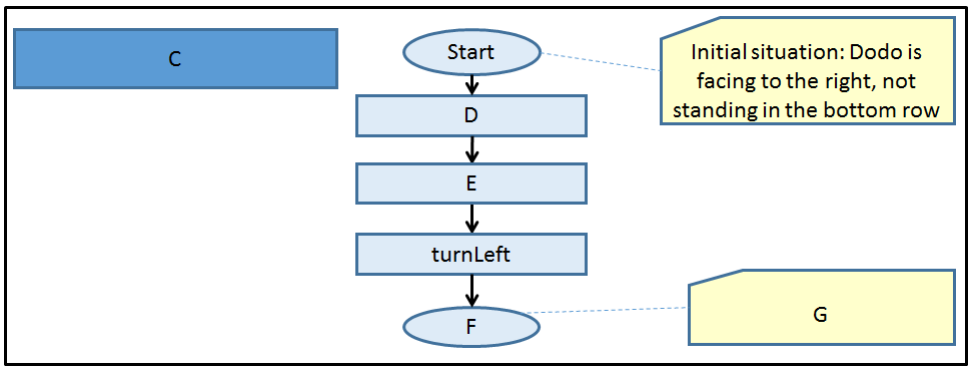


Figure 15: Flowchart for `moveDown()`

Fill in A, B, C, D, E, F and G in the code and the flowchart.

6.2 Answers to the diagnostic test

Here are sample answers to the questions in the diagnostic test:

1. A result type indicates which type of value the method can return. A mutator method which 'does something' to an object has **`void`** as a return type. An accessor method, which returns information about an object, can have, for example a **`boolean`** (which can be true or false), or a **`int`** (integer) as a return type.

2.

Method	Result type	Parameter type	Belongs to class	To be used by object
`getNrOfEggsHatched()`	int	not applicable	MyDodo	MyDodo
`canMove()`	boolean	not applicable	MyDodo	MyDodo
`jump()`	void	int	MyDodo	MyDodo
`layEgg()`	void	not applicable	Dodo	Dodo and MyDodo
`turnLeft()`	void	not applicable	Dodo	Dodo and MyDodo
`canMove()`	boolean	not applicable	MyDodo	MyDodo
`getX()`	int	not applicable	Actor	all objects

3. `walkToEgg`
4. `nrOfEggs` (note: the name `eggs` on its own is not meaningful)
5. The coordinates in the bottom-left corner are: (0,11)
6. See the flowchart in figure 16.

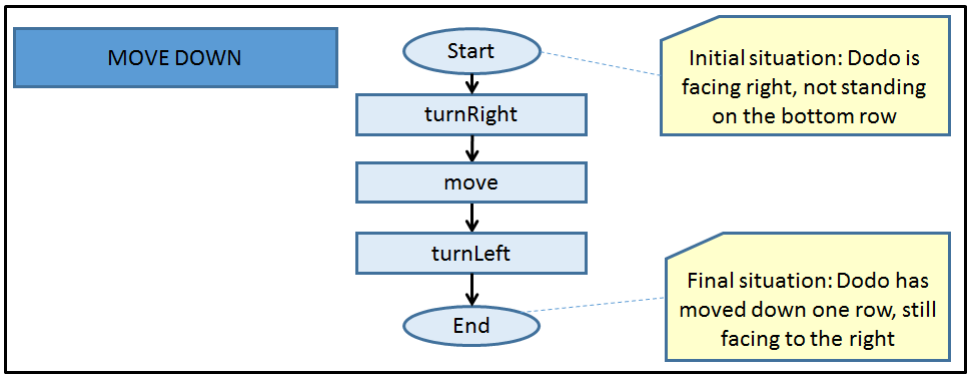


Figure 16: Flowchart for `moveDown( )`

7 Saving your work

You have just finished the first 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 work in on Magister.

1. Go to the folder where you saved your work.
2. With each scenario a 'README.TXT' file is automatically generated. Open the 'README.TXT' file and type your name(s) at the top.
3. Place the (Word) document with your answers to be handed in (answers to the '(IN) ' questions) in the same folder.
4. Compress the entire file into one `.zip` file. In Windows you can do this by right-clicking on the folder and choosing 'Send to' and then 'Compressed (zipped) folder'.
5. Hand the compressed (zipped) folder in via Magister.

Make sure you hand your work in before next Wednesday 8:30 (in the morning).