Assignment 7: Dodo's Race

– Algorithmic Thinking and Structured Programming (in Greenfoot) –

1 Introduction

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

Who can make the smartest Dodo? In this assignment you are going to compete against your classmates. One golden egg (worth 5 points) and 15 blue eggs (each worth one point) will be placed in the world at random. The goal of the assignment is to program MyDodo so that she obtains as many points as possible by finding and hatching eggs. However, the number of steps she can take is limited. So, Mimi has to gain as many points as possible in the least number of steps. 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?

The idea is that you come up with the best possible algorithm and implement it. You can then use different practice scenarios to test your algorithm. 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. For the final race, a new unknown scenario will be used. The goal is thus to devise an algorithm that is generic enough to be the 'smartest' in any unknown scenario.

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 hatching them;
- ... and Mimi must become much smarter!

2 Learning objectives

After completing this assignment, you will be able to:

- apply an **`else if`** statement;
- use *random* numbers;
- explain what a **class constant** is used for;
- use class constants;
- give examples of when a **`List`** could be useful;
- declare and use `List` variables;
- explain, in your own words, what **object types** are;

- explain, in your own words, what the difference is between object types and **primitive types** when assigning a value;
- explain what **`null`** means and what it is used for;
- explain that a list object can hold primitive types or other object types;
- apply existing `List` methods, such as getting and deleting elements;
- use Java Library Documentation to look for and use existing Java methods;
- use a *for-each-loop* for traversing the elements of a list;
- swap elements in a list;
- split a complex algorithm into subalgorithms;
- implement a sequence of subalgorithms in Greenfoot.

3 Instructions

In the previous exercises you have wrote quite a lot of code. The goal for writing that code was just for practicing. 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. If you realise that you want to use a method that you've already written and tested in a previous assignment, you can simply copy-paste it into the new scenario (and retest it).

In this assignment you will need scenario **'DodoScenario7'**. As usual, we start by making a copy in which we will work:

- Download **'DodoScenario7'** from the course website[1].
- Open 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: `Asgmt7_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) must be discussed with your programming partner. Jot down a short answer on the assignment paper.

A note in advance: Except for a few minor changes to `Madagaskar`, in this assignment you may only make changes to the `MyDodo` class.

[1] `http://www.cs.ru.nl/S.Smetsers/Greenfoot/Kandinsky/`

4 Theory and exercises

We first start with two theory blocks, each followed by exercises to practice the given theory. After that you will start programming for Dodo's race.

Nested `if .. then .. else` statements

A *nested* ***`if`*** *..* `then` *..* ***`else`*** statement tests several cases simultaneously.

Flowchart:
The following is an example of a flowchart of a nested **`if`** .. `then` .. **`else`** statement:

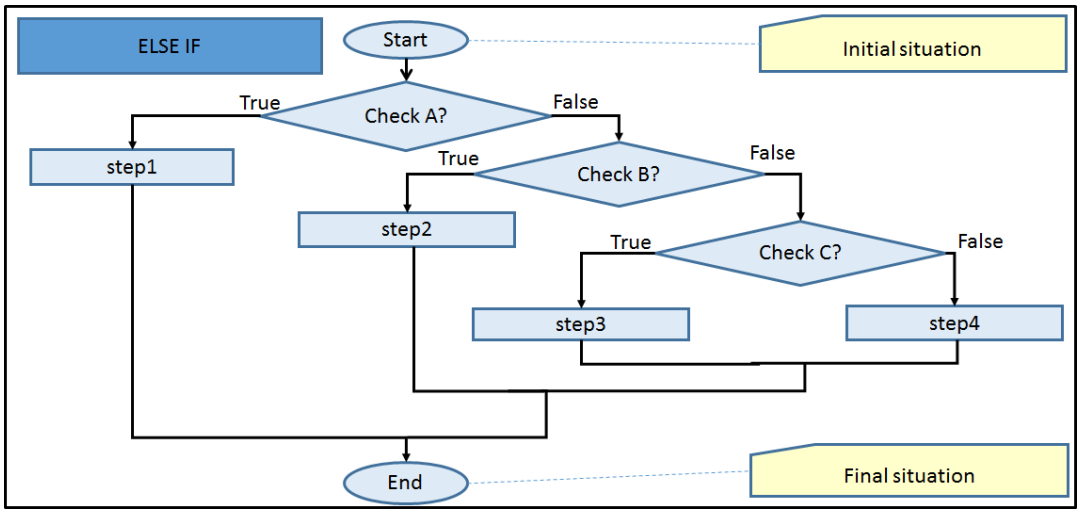

Figure 1: Flowchart of a nested **`if`** .. `then` .. **`else`**

The flowchart explained:

- First the conditional expression 'Check A?' in the first diamond is evaluated.
- If the conditional expression is true, then the 'True' arrow will be followed and 'step1' will be executed.
- If the conditional expression is not true, then the 'False' arrow on the right will be followed and 'Check B?' will be evaluated
  - If the conditional expression 'Check B' is true, then 'step2' will be executed.
  - If the conditional expression 'Check B' is false, then the 'False' arrow on the right will be followed to 'Check C?'.
- ...

Code:

```
if( checkA() ){
    step1();
} else {
    if( checkB() ){
        step2();
    } else {
        if( checkC() ){
            step3();
        } else {
            step4();
        }
    }
}
```

Simplified:
This type of nesting can simplified:

- If 'Check A?' is 'True' then 'step1' is executed;
- Else (so A is 'False'), if 'Check B?' is 'True' then 'step2' will be executed.
- Else (so A and B are both 'False'), if 'Check C?' is 'True' then 'step3' will be executed.
- Else (so A, B and C are all 'False'), then 'step4' will be executed.

Code:
The **else** and the **if** are combined. The method above becomes:

```
if( checkA() ){
    step1();
} else if ( checkB() ){
    step2();
} else if ( checkC() ){
    step3();
} else {
    step4();
}
```

We now need fewer lines and fewer curly brackets. This makes the code a lot less messy, easier to read and less prone to errors.

Note:
The last branch can use either an **else** or an **else if** (with another condition, say `checkD`). However, these do have another meaning! The code after an **else if** will only be executed if `checkD` returns true.

Object types
We have learned that Java has primitive types such as **int** and **boolean**. Primitive types are built-in Java. That means you get primitive types as a present.

In addition to primitive types, Java also has `object types` (sometimes called *reference types*). Object types are types that belong to a class. You can make a new object type by writing (or importing) a new class. Examples of object types are `MyDodo` and `Egg`.

Use of object types:
You can use object types in the same way as you use primitive types. As a reminder we summarise the places in code where you have already used types:

Location	Example	Type
result	`int methodName( )`	`int`
parameter	`void methodName( String text )`	`String`
local variable	`int value = 4`	`int`
instance variable	`private int myNrEggsFound = 0`	`int`

However, you have also already seen object types. For example, in assignment 5, we gently skipped over the following:

```
World myWorld = getWorld( );
```

Here the variable with the name `myWorld` and type `World` is declared (made) and immediately initialised (given a value) with the result of `getWorld( )`. The result type is `World`. You can see this by looking at the signature `public World getWorld( )`. Because `World` is a class, its type is an object type (rather than a primitive type).

Values:
The type determines which sorts of values can be used. For example, a variable with type `int` (such as `int nrOfEggs`) can only store integers. It can't store a `boolean` or `String`. The same applies to object types. A variable whose type is a class, such as `Egg`, can only store `Egg` objects. It can't store a `Fence` object or an `int` value.

However, we can store a `BlueEgg` or a `GoldenEgg` object in an `Egg` variable. This is because each `BlueEgg` or `GoldenEgg` object is also an `Egg` (see the chapter on 'Inheritance' in assignment 1). An example of a variable declaration and initialization with an object type is:
`Egg firstEgg = new BlueEgg( );`

Note:

- It may seem as though that primitive types and object types are more or less the same. However, there are some differences. One of the differences is the way in which the values are stored. For a variable `nrOfEggs` with a primitive type such as `int`, the value (for example 4) is stored directly in the variable.

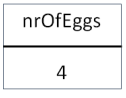

Figure 2: Values of primitive types are stored directly in the variable

For objects this works differently. Objects are not stored directly in a variable. Rather, a *reference* to the object is stored. You can compare it to the way in which a program like Facebook stores your *friends*. Obviously, your friends aren't physically stored in Facebook. But their Facebook login name is stored. You can see this login as a reference to the person itself. Besides that, other Facebook-users can also use this login to reference to the same person. In the example below you can see how both Peter and Jane have a reference to Jack.

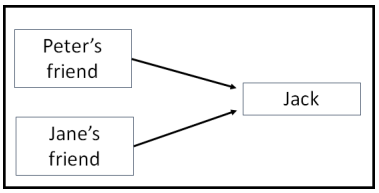

Figure 3: Values of object types refer to objects

In a Java program the same happens. You can have several variables that all point to the same object, and thus all have the same reference as a value.

- A common programming error is using the wrong type. For example, the compiler will complain if you try to give an `Egg` variable the **`int`** value 3:

```
// incorrect assignment
Egg thirdEgg = 3;
```

4.1 Instance variables that point to objects

To get a better understanding of object types, we will practice using them in this exercise. We will first add two new instance variables with an object type. Then, using setter methods we will assign them a value (a pointer as a value). We will then investigate what the effect is of changing that pointer. By "playing around" with pointers, you will be able to get a better grasp on what they are and how they work. Follow the next steps:

1. Make sure the scenario (automatically) starts with the 'world_mimi_2_eggs.txt' world.
2. Add two instance variables to the `MyDodo` class, both of the type `Egg`. Name the first instance variable `myFirstEgg`, and the second `mySecondEgg`. Tip: use **`private`** `Egg myFirstEgg;`.
3. For each variable add a setter method which gives these variables a value (a certain egg). The setter method's signature for `myFirstEgg` is:

   ```
   public void setFirstEgg( Egg newEgg )
   ```

   Complete the code of so that `myFirstEgg` is assigned the value in the argument (`newEgg`). Then do the same for `mySecondEgg`.
4. Compile (but don't run) your code to check that you haven't made any typo's.
5. We will now write methods for hatching the eggs. For `myFirstEgg` we write the method **`public void`** `hatchFirstEgg( )` which changes the state of the `myFirstEgg` to 'hatched'. To do that, use `myFirstEgg.setHatched( )`.
6. Do the same for the other instance variable.
7. Compile your code.
8. Right-click on Mimi and call the **`void`** `hatchFirstEgg( )` method. What happens? If you don't see anything, check the console. Which message do you get? This is what you get when you don't immediately give an initial value to an instance variable. You don't have to do anything with this message right now. We will get back to this later. To solve this we must first assign an egg value to `myFirstEgg`(see the next step).

9. We will now investigate what the effects are of assigning values to the pointers:
   (a) We start by assigning a value to the instance variable `myFirstEgg`. Do that by right-clicking on the Mimi and then calling the the **`void`** `setFirstEgg( Egg newEgg )` method.
   (b) A window appears asking for a parameter. We want the parameter to *point* to the topmost egg. Do this by clicking on the topmost egg. Now, the instance variable `myFirstEgg` is pointing to the topmost egg.
   (c) (IN) Now call the **`void`** `hatchFirstEgg( )` method. What happens? Explain why.
   (d) Repeat the previous steps for the second instance variable `mySecondEgg`, however, this time, point to the bottommost egg (just as you did with `myFirstEgg` in part 9b). Check if what you see corresponds to what you expected.
10. We will now investigate what the effects are of changing the pointers:
   (a) Press *Reset* to restart the scenario.
   (b) Call the **`void`** `setFirstEgg( Egg newEgg )` method and point to the topmost egg. Call the **`void`** `setSecondEgg( Egg newEgg )` method and **also** point to the topmost egg.
   (c) Now call **`void`** `hatchFirstEgg( )`. Does what you expect actually happen?
   (d) Now call **`void`** `hatchSecondEgg( )`.
   (e) (IN) Sketch the new situation with its pointers. Give an explanation for what happens. Use the information about *reference types* from the theory block above.

`null`
An instance variable with an object type, which has been declared but not initialized, doesn't point to an object. It has not yet been given an initial value. For example:

```
private Egg myFirstEgg;
```

Here, the variable `myFirstEgg` is declared but is not yet initialized, and thus `myFirstEgg` isn't pointing to anything. Java uses **`null`** to indicate this situation. **`null`** is a special value which means 'no object'. In Java, instance variables with an object type which are not explicitly assigned a value, are given the value **`null`**.

If a variable has the value **`null`**, then you can't call its methods. Methods belong to objects. They change the state of the object (mutator methods) or give information about an object (accessor methods). So, if a variable has the value **`null`**, the object is missing, and its impossible to call its methods.

A common programming error is to try to call the method of a variable which has the value **`null`**. Trying this anyway will result in a `NullPointerException`. We saw this in the previous exercise 4.1 part 8. There the method **`void`** `hatchFirstEgg( )` was called before the variable `myFirstEgg` was initialized. At that moment `myFirstEgg` didn't have a value yet, and was thus **`null`**. The method call resulted in a `NullPointerException`. (The variable `myFirstEgg` was later initialized with the value `setFirstEgg`).

Agreement:
It's a good habit to explicitly initialize every instance variable in the constructor. If you are not yet sure which value it should get, use **`null`**.

4.2 Dodo scores

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? Mimi is allowed to take 40 steps. With who's algorithm will she hatch the most eggs?

We are now going to try out several different algorithms. After that, you will come up and try out your own algorithms.

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.

4.2.1 (IN) Random movements

We will implement the first simple algorithm together. In this exercise we let Mimi walk through the world, taking steps in *random* directions (almost as if she were drunk).

1. You can use the following `Dodo` method to choose a random direction:

   ```
   public int randomDirection( )
   ```

   Open the `Dodo` class and find this method.

2. This method makes use of a standard method from the Greenfoot library, namely `getRandomNumber`. The 'Greenfoot Class Documentation' describes all Greenfoot methods. Find this method in the Greenfoot library. Tip: In the Greenfoot menu at the top of the screen, go to 'Help' and then 'Greenfoot Class Documentation'. You can easily search using 'Index'.

3. The method is given the parameter '4'. What does this mean?

4. 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?

5. Come up with an algorithm which makes Mimi choose a random direction and then take a step in that direction. Make sure that Mimi doesn't step out of the world or walk into a fence. Tip: before taking a step, check whether or not she can actually make that move.

6. In `MyDodo`, write the corresponding method `moveRandomly( )`.

7. Compile and test your method by right-clicking.

8. Does Mimi do what you expect her to do?

9. Modify the code in `MyDodo`'s `act` method so that it calls the `moveRandomly( )` method.

10. Compile, run and test your program. Tip: if it doesn't work properly, follow the steps described at the end of chapter 'Debugging' in assignment 2.

11. Add code to `moveRandomly( )` so that Mimi hatches an egg as soon as she finds one.

Class constant
A class constant is a variable whose value can't change throughout the program. 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`.

Public constants
Class constants that are **`public`** can be referred to from other classes. For example, in the `Dodo` class, four **`public`** class constants are declared. The first declaration is:

```
public static final int NORTH = 0;
```

This constant is used in different places in the `Dodo` class. Because it is **`public`**, it can also be used in other classes. If another class, for example the `Egg` class, wants to use the constant then the you must indicate which class it comes from. This is done by writing the class name in front of the constant, separated by a full-stop '.'. To refer to `Dodo`'s constant `NORTH` from within the `Egg` class, use: `Dodo.NORTH`.

To see how this is used in the code, have a look at the method

```
public void push( int direction )
```

in the `Egg` class. There, the condition `direction == Dodo.NORTH` is used to test if `Dodo` is facing in the `NORTH direction`.

4.2.2 Scoreboard

The game ends as soon as the maximum number of steps has been taken. We will now modify the program so that it stops as soon as this maximum has been reached. 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).

1. It's useful to make a variable for the maximum number of steps. By using a variable it's easier to change the value if we would wish to do so, so instead of using the value '40' we choose to create a variable which we initialize with the value '40'. Because this value will not change throughout a program, we use a class constant, named `MAXSTEPS`. The class constant has already been added to the class `Madagaskar` for you. Adjust it to the appropriate value.
2. We will now add a scoreboard to the scenario. We do this as follows:
   (a) Open the `Madagaskar` class and look for the method call of `addScoreboard()` in the constructor. This has already been typed in for you, but has been placed in comments.

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

(c) Compile.

(d) As you can see in the scenario, the scoreboard has been added at the bottom of the world. How many values does the scoreboard hold?

3. `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. 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'.

4. No one can keep track of how many steps Mimi takes better than Mimi herself. To do this, an instance variable `myNrOfStepsTaken` has been added to the `MyDodo` class. The `myNrOfStepsTaken` variable keeps track of the number of steps taken. Similarly, the instance variable `myEggScore` has been added to the `MyDodo` class to keep track of the points scored (score). We will now adjust the code in order to be able to use these variables. Follow the next steps:

   (a) Give both instance variables their correct initial values. Tip: Do this in the constructor.

   (b) Make sure that `myNrOfStepsTaken` is modified as soon as Mimi takes a step. Tip: Do this in `move( )`.

   (c) Make sure that `myEggScore` is modified as soon as Mimi hatches an egg. Tips:

   - Do this in the `hatchEgg( )` method;
   - You can ask the value of a particular egg (the number of points you get by hatching it) using `getValue( )`.
   - If you don't know how to use `MAXSTEPS`, then reread the theory block above.

   (d) Also call **`void`** `updateScores` to make sure 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.

5. Test the program by calling the `moveRandomly( )` method in `act( )`. Make your test complete: also test hatching a golden egg (5 points) or an egg that has already been hatched (0 points).

6. What happens after Mimi takes 40 steps?

7. Adjust the program so that, before Mimi takes a step, it checks if this is possible. Once Mimi has taken the maximum number of steps, show her a compliment (in a dialogue) with her score and then stop the program.

4.3 Walking through a list of eggs

Adding the `moveRandomly( )` method to Mimi's repertoire (in the previous exercise) hasn't really made her any smarter. In previous exercises we have done better! We have already come up with a smarter strategy to find eggs, namely the method `walkThroughWorldAndCountEggs( )` in exercise 5.3.1 'Counting eggs in the world' (assignment 5). In that exercise, Mimi traversed each row systematically as shown in the following picture:

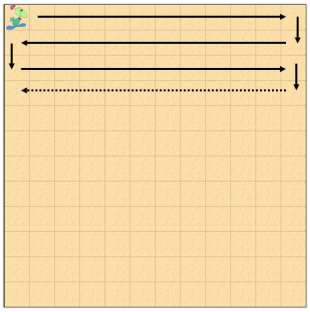

Figure 4: Mimi walks through the world systematically

However, there's an even smarter way to look for and hatch eggs. For this, we will use Java's `List`s. Imagine that you would have a list of all the `Egg` objects in the world. Thus, using getter methods, you would also be able to get a hold of their coordinates (and their values). Having this list, would you be able to come up with a smarter algorithm to hatch all the eggs using a minimal amount of steps? In this exercise you will learn how to use `List`s.

Lists
The object variables that we used in the previous assignments could hold exactly one object. However, sometimes it can be useful to store a whole series (or row) of objects, rather than just one object. Obviously you could make a separate variable for each object, but if you have lots of objects that's not practical. What you need is a way to group objects. There are several ways to do this in Java, the easiest is using a `List`.

A `List` is a series of variables. These variables are the *elements* of a list. These are stored in sequence. Because they are stored in sequence, you can number them. And that is exactly the way in which you can select an element. Namely, you can select (or find) an element by using its position or *index*. We start counting at 0. This is done as follows: the first element has index (is at position) 0, the second element has index 1, ... , the tenth element has index 9. So, you can select the 10th element out of a list by asking for the element with index 9.
Elements in lists:
All elements in a list are of the same type. You can't store `Egg`s and `Dodo`s in the same list: it is either a list of `Egg`s or a list of `Dodo`s.

Operations on lists:
The number of elements in a list is called its *size*. You can add elements to a list. You can remove elements out of a list. You can add or remove anywhere in a list: the front, back, or somewhere in the middle (using the index to indicate exactly where). A list with no elements is *empty*.

Lists in Java:
The Java class for lists is called `List`. Before using lists you must import a special library. This sounds more complicated than it is: just add the following line of code to the top of the class document where you want to use lists: `import java.util.List;`.

The Java `List` library has lots of useful methods which you can use, such as:

- Length: `size( )`. Returns the number of elements in the list.
- Select: `get( int index )`. Returns the element at position 'index'.
- Add an element: `add( int index, E element )`. Adds 'element' to the list at position 'index'.

- Remove element: `remove( E element )`. Removes 'element' from the list.
- Remove element at position:] `remove( int index )`. Removes element from list at position 'index'.
- Is list empty?: `isEmpty( )`. Returns whether or not the list is empty. Because you can't remove elements from an empty list, this is a very useful check before trying to remove any elements.
- Contains: `contains( Object o )`. Returns whether or not object 'o' is in the list.

Knowing which methods are available to you, and understanding how to use them can save you a lot of time. Why reinvent the wheel when these methods have been written and tested for you!

Have a look at the complete Java library `List`. To get an overview of all the methods available to you, you can do one of the following two steps:

- Go to `https://docs.oracle.com/javase/7/docs/api/java/util/List.html`.
- In the Greenfoot menu choose 'Help', and then 'Java Library Documentation'. Then search for 'List'.

A list can contain objects. But it's important to realize that lists are objects too! A cookbook has a list of recipes and each recipe has a list of ingredients. So, a list can contain a list of other objects.

The same rules which apply to any other objects also apply to list objects. If you want to use a list, you first declare a variable in which you can store a reference to a list object.

For example, a list of compliments for Mimi can be declared as follows:

```
List<String> dodoCompliments;
```

where `dodoCompliments` is the variable's name and `String` is the type of the objects in the list.

The `MyDodo` class has an example of lists:

```
public List<Egg> getListOfEggsInWorld()
```

This method returns a list of all the eggs in the world. This can be useful when programming Mimi to find the eggs in the world which she has to hatch.

To use this list of eggs, a variable to store the list must first be declared:

```
List<Egg>  eggsInTheWorld = getListOfEggsInWorld( );
```

In this example:

- A variable with the name `eggsInTheWorld` is declared;
- The type of `eggsInTheWorld` is `List<Egg>` (a list of eggs);
- The initial value of `eggsInTheWorld` is the result of the `getListOfEggsInWorld`: the `eggsInTheWorld` is a list of all the eggs in the world at that moment.

Now that there is a variable `eggsInTheWorld` with the list of eggs, things can be done with that list, such as searching through it (e.g., to find the nearest egg), or removing eggs from it (once it has been hatched).

The for-each-loop:
To traverse a list (i.e. to go through each of the elements one-by-one), in Java a *for-each-loop* is used.

```
void methodDoSomethingWithList ( ) {
    for ( ElementType elemVariable: listOfElements ) {
        doSomethingWithElement ( elemVariable );
    }
}
```

Here a list of elements `listOfElements` is traversed. For each element in the list, the code in the loop is executed. So for a list with 5 elements, the code in the loop is executed 5 times. During each loop the variable `elemVariable` attains the next object in the list. The object's type is `ElementType`. In the method call `doSomethingWithElement`, something is done with that particular object.

Example:
The following code traverses a list of `Egg` elements and hatches each of them:

```
public void hatchEachEggInWorld ( ) {
    List<Egg> eggList = getListOfEggsInWorld( );
    for ( Egg egg: eggList ) {       // get the egg in the list
        egg.setHatched ( true );      // and hatch that egg
    }
}
```

In this example, the method `getListOfEggsInWorld( )` returns a list of eggs. The list variable `eggList` is declared and initialized, it stores the list of eggs. In the for-each-loop, one-by-one, we take a look at each `Egg` in the `eggList`. For this egg the method `setHatched( true );` is called. So, at the end of the for-each-loop, each of the eggs in the world will be hatched.

In the following exercises you will work with lists. You will be using methods that return lists. You won't have to make any list objects yourself. However, you will need to use a list variable to store the list which a method returns (just like the list variable `eggList` stores the result of the method `getListOfEggsInWorld( )` in the example above). Once you have stored this list, you can do things with it, like walk through its elements.

4.3.1 Make a list of eggs and print their coordinates

In this exercise we will practice using lists. Help Mimi make a list of all the eggs in the world and then print the coordinates of each egg.

1. Open the 'world_eggs' world.
2. The `getListOfEggsInWorld( )` method in the theory block above is used to make a list of all the eggs in the world. Find the method declaration in the code and add comments to it, explaining in your own words what it does.
3. We will now write a method that traverses a list of eggs and prints the coordinates of each egg in the console:
   (a) Open `MyDodo`'s code in the editor. If it has not been done already, import the Java `List` library by adding the following just below `import greenfoot.*;`:

   ```
   import java.util.List;
   ```

   (b) Choose a method name for printing the coordinates of the eggs.

(c) Declare an `Egg` list and initialize it so that it stores all the `Egg` objects in the world. Tip: Use part 2.

(d) Use a *for-each-loop* to traverse the list, egg-by-egg.

(e) For each egg, get its coordinates and print these. Tips:

- Use `egg.getX( )` and `egg.getY( )` to get the coordinates of an egg (with the name `egg`);
- Use the Java `println` method to print to the console.

4. By right-clicking, call and test your method. Add a few extra eggs to the world and re-test. Does your method also work if there are no eggs in the world?

4.3.2 (IN) Java methods for lists

We will now have a look at `List` in the Java library so that you become acquainted with some predefined methods that you can use, as well as how to read the Java library documentation (`https://docs.oracle.com/javase/7/docs/api/java/util/List.html`). Suppose you would have a list of eggs called `myEggList`. Which method call:

1. removes an element from a list at a particular `index`.
2. removes the second egg in the list? Choose the correct answer from the following:
   (a) myEggList.remove(0);
   (b) myEggList.remove(1);
   (c) myEggList.remove(2);
   (d) myEggList.remove(3);
3. removes an object from the front of the list? Choose the correct answer from the following:
   (a) egg.remove(0);
   (b) myEggList.remove(0);
   (c) List.remove(0);
4. returns the third egg in the list?
5. adds an egg called `newEgg` to the front of the list?
6. returns how many objects a list has (i.e. how long it is)?
7. removes the last object in the list `myEggList`?
8. checks if the list is empty?
9. checks if the list exists (that is to say, if `myEggList` actually points to a list)?

4.3.3 Turning lists around

New in this scenario are surprise eggs. The class `SurpriseEgg` is a subclass of `Egg` (just like `BlueEgg` and `GoldenEgg`). When hatched, Blue eggs are worth one point, golden eggs are worth five points, and how many points a surprise egg is worth, well, that's a surprise! In the constructor of `SurpriseEgg` its value is set randomly. This is done using the Greenfoot method `getRandomNumber`. So, every surprise egg has a (different) random value. Hatching a surprise eggs results in a surprise (random) number of points.

You can make a list of surprise eggs using the following method in the `SurpriseEgg`

```
public static List<SurpriseEgg> generateListOfSurpriseEggs( int size )
```

This method expects a number as a parameter, namely the number of eggs you want to have in the list. For example, the following code makes a list of 10 surprise eggs:

```
List<SurpriseEgg> mySurpriseEggList = SurpriseEgg.generateListOfSurpriseEggs( 10 );
```

As usual, a variable (`mySurpriseEggList`) is declared to hold the list that is produced by `generateListOfSurpriseEggs`. The type of that variable is the same as the result type of the method: `List<SurpriseEgg>`. As you can see, the method is called by giving the class name `SurpriseEgg` first, followed by the name of the method `generateListOfSurpriseEggs`.

In the following exercises you will write several list methods. This time you will not only 'look' at the elements in the list. You will sort the lists (put elements in a particular order). Using sorting, we can give Mimi a strategy for hatching eggs.

1. **Make and print a list:** We start by making a list of surprise eggs and then printing the values of each of the surprise eggs.
   (a) Come up with an appropriate name for this method.
   (b) Use the `generateListOfSurpriseEggs` method to make a list of 10 surprise eggs.
   (c) A method for printing the values can be very useful in other parts of the program (as you will see later) or while debugging, so we'll make a sub-method out of it. As a sub-method, it can easily be called (and thus re-used). Write a method which prints the value of each surprise egg in the console:
      - Come up with a suitable name for this method. Pass the list to be printed as a parameter.
      - Using a *for-each-loop* traverse the eggs in the list;
      - For each egg, get its value and print it in the console using `println`.
2. **Most valuable egg:** The first strategy (or algorithm) is rather straightforward: finding the most valuable egg.
   (a) Come up with an algorithm to find the egg in the list that has the highest value. Tip: While traversing the list, have a (local) variable keep track of the highest value you have found so far. You have done something similar in assignment 5 exercise 5.3.2 'Find the row with the most eggs'.
   (b) Implement this algorithm.
   (c) Compile and test your code by printing the highest value in the console. Tip: You may find it useful to call your sub-method for printing the list of eggs (part 1c) to check if your method works properly.
   (d) Is the result correct? Execute the code several more times and check if the correct value is found each time. Fix any errors before you continue to the next exercise.
3. **Turning a list around (optional and challenging exercise):** Write a method that turns the list of surprise eggs around. That is to say, reorganizes the list so that the last element is put in the front, followed by the second to last, etc.
   (a) Come up with a name for your method.
   (b) Generate a new list of 10 surprise eggs.
   (c) Print the values in the list. Tip: use the sub-method you wrote in the previous exercise 1c.
   (d) Come up with an algorithm to reorder the elements in the list accordingly. Tips:
      i. You already learned how to switch two values in assignment 5 5.1.1 'Tracing code' part 10.

ii. Have a look at the different methods available to you in the Java Library Documentation (such as `add`, `get`, `set`, `remove`). Choose which one(s) you want to use in your algorithm.

iii. First implement a small part of the algorithm and test this:

A. Start by swapping the first and last element.

B. Print the list again.

C. Compile and test this. Also check if, in the end, the list is just as big as it was initially.

(e) Now, implement the algorithm to turn the whole list around. Tips:

- Use a **`while`**`–loop` (and not a *for-each-loop*).
- Use a local variable called `index` to keep track of where you are in the list. Don't forget to increment this variable in the **`while`** loop.

(f) Compile and test your code by printing the values of each element.

(g) Does your program work as you expect it to? If not, think of why it doesn't work properly. To help debugging, print the list of values throughout the code to figure out what happens. Fix your program.

(h) Does your program also work if there are 11 eggs in your list? And also if there are 0?

4.3.4 Hatching the first egg in the list

Help Mimi get the first egg out of the list, walk to it, and then hatch it. The algorithm should be made up of the following steps:

Task 1: Get list of eggs. Get the list of eggs (and print it);

Task 2: Determine destination. Get the first element out of the list and use this to determine Mimi's destination;

Task 3: Walk to the egg. Have Mimi walk to the egg;

Task 4: Hatch the egg. Have Mimi hatch the egg.

The corresponding flowchart looks like this:

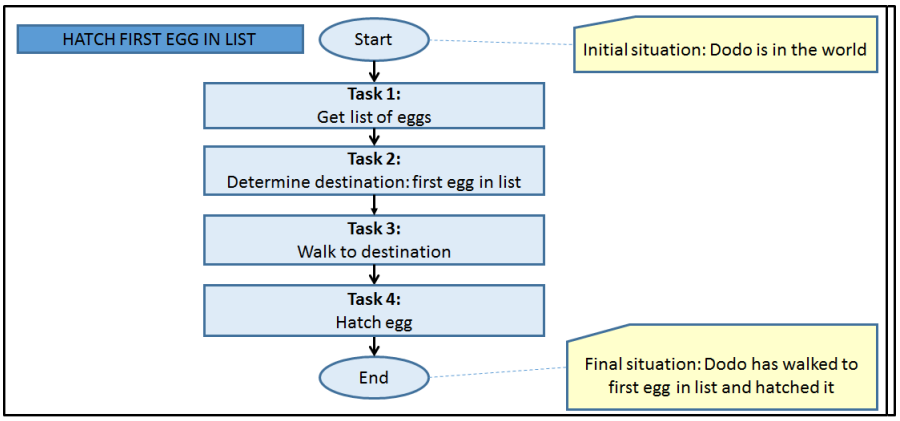

Figure 5: Flowchart for 'Hatching the first egg in the list'

The algorithm has now been split into several subtasks. Solve these and execute them consecutively (one after the other) to solve the problem as a whole.

1. Come up with a method name (for all the subtasks together).

2. The four subtasks will be carried out in the body of this method. For each task, write and test a separate sub-method, and then call it in your main method.

   Task 1: Get list of eggs. (a) Make a list of the `Egg` objects in the world. To do this, use a local variable, for example, with name `eggsInTheWorld`. Tip: Just as in exercise 4.3.1 part 2, use `getListOfEggsInWorld()` again to get a list of all the eggs in the world.
   (b) Print the list.
   (c) Test this task.

   Task 2: Determine destination. (a) First check if the list is not empty. This is important because you're not allowed to try to get something out of an empty list.
   (b) Get the first egg out of the list.
   (c) Print the coordinates of that egg using `println`.
   (d) Test this task.

   Task 3: Walk to the egg. Go to the egg. Tip: Use the following method:

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

   which you wrote in assignment 5. Test this task.

   Task 4: Hatch the egg. Let Mimi hatch the egg. Test this task.

3. Make sure you call all four subtasks in your method. Now compile and test your code by right-clicking on the main method of your algorithm.

4. Have a look at the console. Explain what happens.

4.3.5 Finding the nearest egg

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

1. Given the following (partial) flowchart:

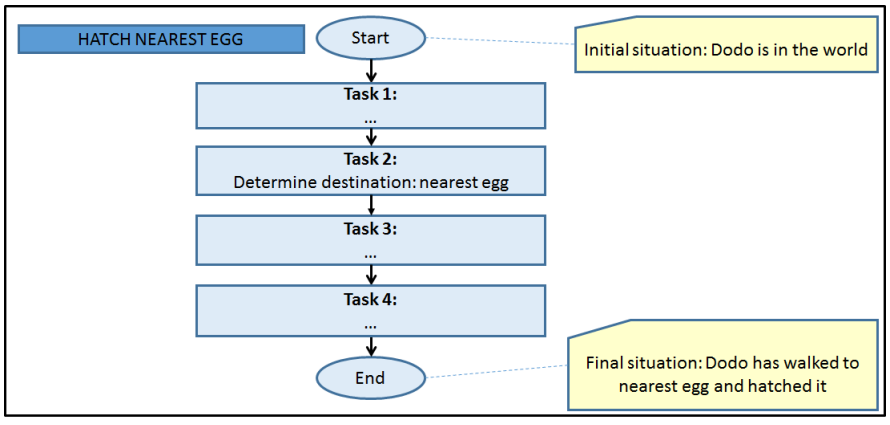

Figure 6: Flowchart for 'Finding and hatching the nearest egg'

Complete the following list of tasks:

Task 1:

Task 2:

Task 3:

Task 4:

2. Come up with an appropriate method name for the entire method (all the subtasks together).
3. Just as in the previous exercise, write sub-methods for each task and call these in your method:

   Task 1: Create a list of eggs in the world, and use a local variable to store this list. Tip: Make use of an existing method.

   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. You don't have to use Pythagoras to calculate the distance, just count the number of steps (or moves) she will need to take. Tip: If you need to turn a negative number into a positive number, the method `Math.abs( number )` will return the absolute value of any given number.

   (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: Implement this task.

   Task 4: Implement this task.

4. Compile and test your method as a whole.

4.3.6 Hatch all eggs in the list

As an extension to exercise 4.3.4 'Hatching the first egg in the list' we now let Mimi hatch all the eggs in the list, one by one. The solution to this problem is similar to that in exercise 4.3.4. However, now Mimi doesn't stop after hatching the first egg, but goes on to hatch the second one, the third, and so on.

We can describe the algorithm as follows:

Task 1: Get a list of all the eggs in the world.

Task 2: Check if the list is empty.

If 'True' (the list is empty), then you're done.

If 'False' go on to the next task (task 2a).

Task 2a: Choose the next egg as your destination.

Task 2b: Go to that destination.

Task 2c: Hatch the egg and go back to the start of task 2.

The corresponding flowchart is as follows:

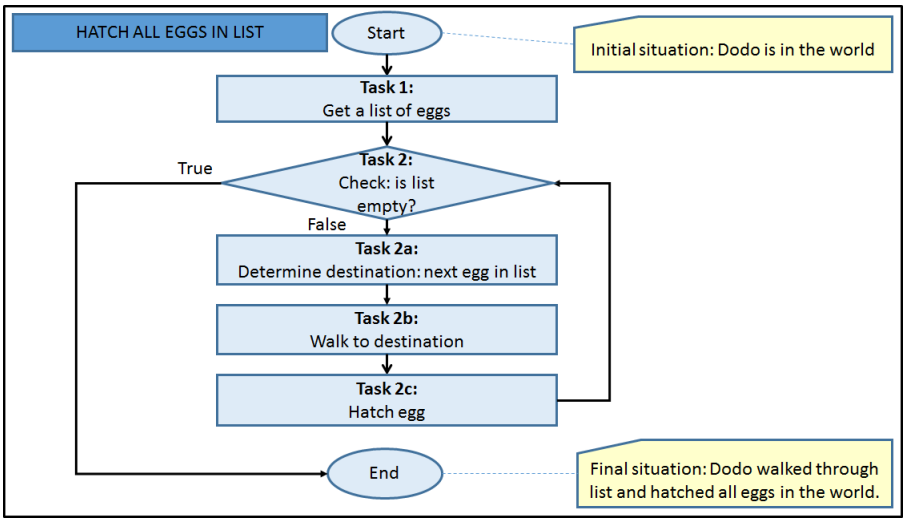

Figure 7: Flowchart for 'Hatch all eggs in the list'

You will now implement this algorithm yourself. This time, however, with less directives. If you get stuck, have a look at the previous exercises.

1. Come up with an appropriate name for your method.
2. For each of the subtasks, write code and test them individually.
3. Call the subtasks in the body of your method.
4. Compile and test your method as a whole.

Tip: use the list method `E remove( int index )` to select and remove the first egg from the egg list.

4.3.7 Continuously hatch the nearest egg

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

4.4 Dodo's Race

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

4.4.1 Competition assumptions and rules

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.
- Mimi can take no more than 40 steps.

- 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:

- world_dodoRace1
- world_dodoRace2
- world_dodoRace3
- world_dodoRace4

Obviously, you may also create your own worlds for testing.

4.4.2 (IN) Design and code

After you finish implementing your solution, make sure that your code includes comments. Take a picture of any design or preparation you did. This could be pseudo-code, flowcharts, or sketches on (scrap) paper. Hand these preparations in together with your code.

4.4.3 (IN) Reflection

After you finish implementing your solution, 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 quality of your *solution*? Have a look back at assignment 4 chapter 'Quality' and give a short analysis.
3. How satisfied are you with the quality of your *code*?
4. 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?
5. 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?
6. Why is your algorithm better than the algorithms that you developed in the other exercises of this assignment?
7. What did you learn by doing this assignment?
8. What would you do differently next time?

5 Summary

You can now:

- explain what object variables and object types are;
- explain what `null` means and what this has to do with calling some methods;
- use an `else if` statement;
- use class constants to store values;
- use lists;
- use *for-each-loop*s;
- search for and use existing methods in the Java Library Documentation;
- break a complex algorithm down into subtasks and implement and test these separately.

6 Saving your work

You have just finished the seventh 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 ...'.

7 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 Madagaskar.java file