Assignment 2: Designing Dodo's moves

Algorithmic Thinking and Structured Programming (in Greenfoot)

Contents

Introduction **1**

Learning objectives **1**

Instructions **1**

Theory **2**
2.1 Flowcharts 2
2.2 Steps for designing code 4
2.3 Sequence of Instructions 4
2.4 Testing (and debugging) 5
2.5 Comparing 5
2.6 Selections (choices) using `if..then..else` 6
2.7 Accessor method 6
2.8 Loading a world from a file 8
2.9 Repetition is... boring and error-prone: wishing for a generic solution 8
2.10 Repetition using `while` 9
2.11 Nesting 9

Challenges **12**
2.1 Sequence of instructions: `gotoEgg` 12
2.2 A sequence with multiple instructions 12
2.3 Mimi can't walk through fences 13
2.4 Writing your own sequence: `climbOverFence ( )` 14
2.5 Find grain 15
2.6 Repeating using `while`: generic `gotoEgg ( )` 16
2.7 Walk to the edge of the world 17
2.8 Combining sub-methods: `walkToWorldEdgeClimbingOverFences ( )` 18
2.9 Giving compliments 19
2.10 Pick up grains and print coordinates 19

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

Reflection **20**

Diagnostic Test **21**

Saving and Handing in **21**

Introduction

Now that you have become acquainted with the the Greenfoot environment and are able to read some code, make modifications and test them, it's time to start writing your own code using some Java language constructs.

In the following challenges you will develop your own algorithms and try them out in Greenfoot. We will practice working in a structured manner, by first visualising your algorithm in a flowchart, and then translating this into working code, step-by-step.

This assignment's goal is: **Design, implement and test solutions**

Scenario

For this assignment you will use the scenario: **'DodoScenario2'**. From now on, you will continue to use the code you write in this assignment.

Learning objectives

After completing this assignment, you will be able to:

- draw a **flowchart** according to its quality criteria;
- devise an **algorithm** as a solution to a problem by identifying **sequences**, **decisions** (or selections) and **repetitions** required for a solution;
- reason about the **correctness** of an algorithm in terms of the **initial and final situations**;
- visualize an algorithm as a combination of a sequence, decision or a repetition in a flowchart;
- transform a flowchart into code;
- combine **sub-methods** to solve a (complex) problem;
- design **conditional expressions** composed of AND, NOT and `boolean` methods;
- use a **return statement** in both flowchart and code;
- make modifications in a **structured** and **incremental** manner;
- find and analyse errors in code, interpret error messages and use this to fix problems (**debugging**).
- compose and implement a **generic algorithm**.

Instructions

For this assignment you will need a new **'DodoScenario2'**: to be downloaded from the course website[2].

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

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

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

- The reflection sheet: complete and hand it in.

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

There are three types of challenges:

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

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

A note in advance:

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

Theory

Theory 2.1: Flowcharts

An algorithm can be visualized in a flowchart. Subsequently, this can be transformed into code.

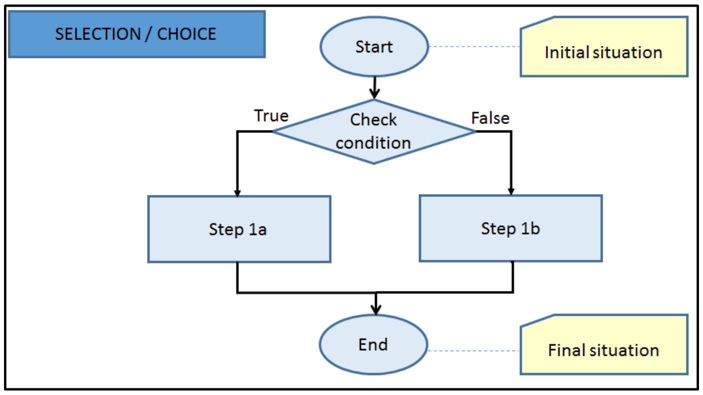


Figure 1: Flowchart

Decomposing a probleem

Sometimes a problem is very complicated and its solution consists of may steps. If you want to solve such a complex problem, you may not know where to start or end up drowning in details. In both cases the problem may seem bigger than it actually is.

A flowchart can help break down (or decompose) a big problem into smaller subproblems. First you have to think up a high-level roadmap (or plan) of what has to be done, and in what order. By placing those steps in a flowchart you can visualize the solution as a whole. Now, each of the subproblems can be tackled one-by-one, without having to worry about the bigger picture. After solving a particular subproblem and testing the solution, you do not have to worry about its details anymore. You can use the solution as a building-block in bigger or other problems. This approach, of breaking down a problem, is called divide-and-conquer. Of course, when you are done solving all the subproblems, you must check whether you have solved the problem as a whole.

What a flowchart looks like

A flowchart has the following components:

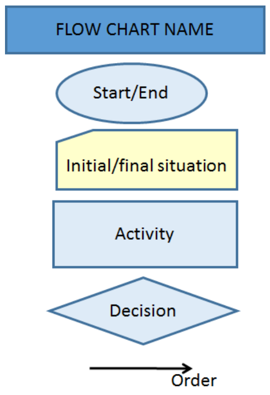


Figure 2: Flowchart components

Flowchart criteria

- Each flowchart has a name.
- There are no loose parts 'floating around' (except the diagram's name). All components are connected by arrows or lines (indicating order).
- There is one initial starting point and one final endpoint, each with a description of its situation.
- An accessor method which yields a result (i.e. gives an answer to a question), has equal initial and final situations.
- An activity (rectangle) always has one incoming and one outgoing arrow.
- An conditional expression (diamond) has at least one incoming and at least two outgoing arrows. The outgoing arrows are labelled with their corresponding values.
- The flowchart has no more than seven activities.

Steps for drawing a flowchart

To solve a problem using a flowchart, follow the next steps:

1. **Initial situation**: Briefly describe (in no more than a few words) what the problem/situation is that is to be solved.
2. **Final situation**: When is your problem solved? How do you know that? Describe that in no more than a few words.
3. **Solution strategy**: Decide on how you want to resolve the issue. Describe it at a high-level: in no more that seven steps.

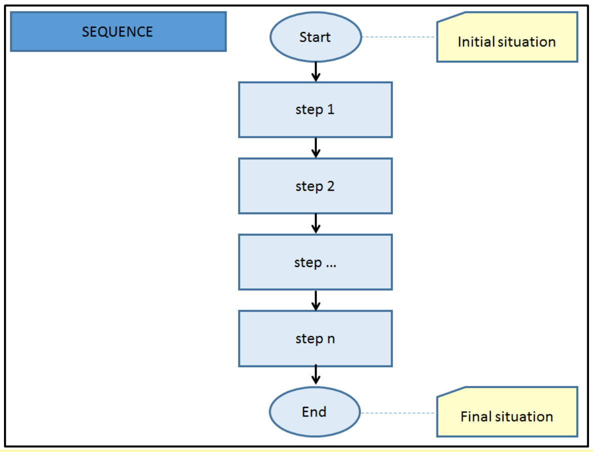
4. **Break down** the problem into sub-problems, each of which can further be broken down into smaller sub-subproblems. For each step:
   - Choose a suitable name (meaningful, consisting of verbs, and formulated at a command) to describe it.
   - Determine if you can re-use by calling the sub-method rather than all its steps. s
   - Determine if steps are repeated. If there is a repetition pattern, use a `while`.
5. Draw the **flowchart**.
6. **Check**:
   - **Rules**: Check if the flowcharts adheres to the 'Flowchart criteria' (listed above).
   - **Quality**: Walk-through the flowchart to ensure it is correct.

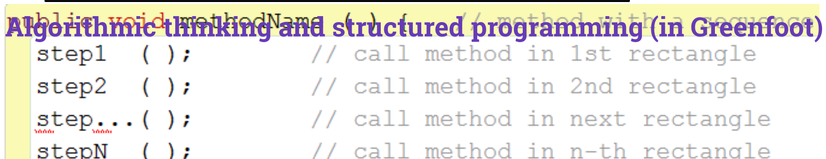
Theory 2.2: Steps for designing code

1. Come up with a global plan (or roadmap) for solving the problem.
2. Draw a **flowchart** for the solution. Have a look back at "Steps for drawing a flowchart" in Theory 2.1.
3. Translate the flowchart into **code**. Pay attention to the naming conventions (discussed in assignment 1).
4. Add **comments** to the code.
5. **Test** the method by dragging an object into the world, right-clicking on the object, and selecting the method. Using several different situations check if the program does what you expect it to.
6. **Debug**. Repair errors. Make sure the code does exactly what the flowchart specifies. If not, adjust your code.
7. **Evaluate the solution and reflect on the process**. Has the problem been solved? Which improvements can you suggest? In the process of reaching the solution, what went right? What could have been done better?

Theory 2.3: Sequence of Instructions

In a sequence, the indicated steps or instructions are performed sequentially, one after the other.





```
public void methodName ( ) {      // method with a sequence
    step1   ( );         // call method in 1st rectangle
    step2   ( );         // call method in 2nd rectangle
    step...( );          // call method in next rectangle
    stepN   ( );         // call method in n-th rectangle
```

Theory 2.4: Testing (and debugging)

Test your program after each minor modification to check if it works as expected.

1. Try to locate exactly where an error occurs.
2. If the program doesn't run at all and you get stuck on compiler errors, have a look back at Theory 1.13).
3. Does the program perform incorrectly? Then you must retrace your steps in reverse order:
   (a) Make sure your code matches your flowchart. You can print values or steps to the console to help trace your code (see Theory 1.14).
   (b) Make sure your flowchart matches the solution you came up with. If the code corresponds exactly with your flowchart, but the program doesn't do what you expect, then you may have a mistake in your flowchart. Analyze your flowchart to determine where you've made any incorrect assumptions. Return to step 2 in Theory 2.2, modify your flowchart, and follow the proceeding steps again.
4. Test different situations. Check that the program deals with unexpected values and boundary situations appropriately.
   - The program such guard for unexpected values: determine what the program should do, like show an error and stop the program. Incorrect values could be negative values or very large values (outside of Mimi's world).
   - Boundary situations are interesting to test, as many mistakes are made in the first and last step. Check for boundary situations such as values 0, 1 or the last step: in front of a border/nest.

   For example, for the method `jump( int distance )`, values for `distance` can't be just anything. Values to guard for are negative values or a large value which would make Mimi step out of the world. Boundary values are 0, 1, and the edge of the world (the width of the world - 1).

Make it a habit to do this immediately after **each** modification. This will help you find any mistakes much faster.

Theory 2.5: Comparing

Is equal to

The comparison operator '==' checks if two values are equal to each other.

An example

Using '`a == 4`' you can compare if '`a`' is equal to '4'. The result is either `true` or `false`.

Note:

The equals-sign '=' (which you use in mathematics) has a different meaning! In code, using '`a = 4`' means 'a becomes 4'.

NOT

The negation operator '`!`' means NOT. The result is either `true` or `false`.

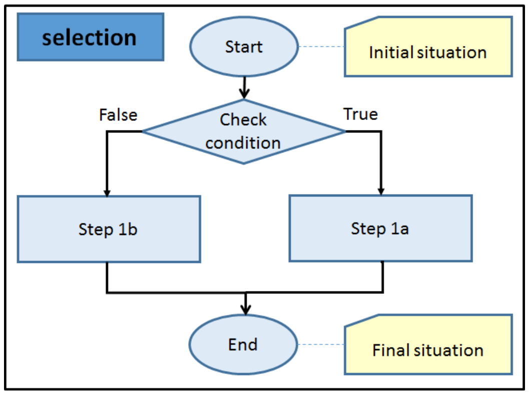
Examples:

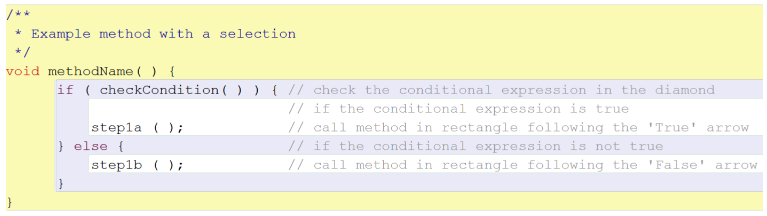
- '! `borderAhead ( )`' checks if there is NO border in front of Mimi.
- '`a != 4`' compares if '`a`' is NOT equal to '4'.

Theory 2.6: Selections (choices) using `if..then..else`

The flowchart and code below show the use of an `if` .. `then` .. `else` construct for:

"If a conditional expression is `true`, then do something. Or else, do something else."





```
/**
 * Example method with a selection
 */
void methodName( ) {
        if ( checkCondition( ) ) { // check the conditional expression in the diamond
                                   // if the conditional expression is true
            step1a ( );            // call method in rectangle following the 'True' arrow
        } else {                   // if the conditional expression is not true
            step1b ( );            // call method in rectangle following the 'False' arrow
        }
}
```

Flowchart and code explained

- If the `condition` is true, then `step1a` is executed.
- Otherwise (the `condition` is false), then `step1b` is executed.

Note:

- If nothing needs to happen when the conditional expression is false, then you can just omit the `else`-branch.
- All branches must come together before continuing with any other statement.

Theory 2.7: Accessor method

An accessor method provides information about an object's state. For example, its result can be an `int` (whole number), `boolean` ('true' or 'false'), or `String` (text).

Flowchart for accessor method

The following is a flowchart corresponding to a `boolean` accessor method:

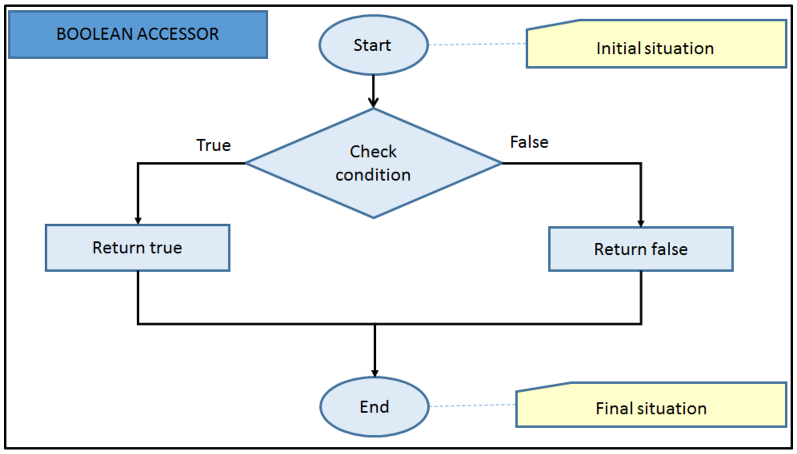


Figure 3: Flowchart for a `boolean` accessor method

The flowchart explained:

- First the conditional expression 'Check condition' in the diamond is checked.
- If the conditional expression is true, then the 'True' arrow on the left will be followed and the result `true` will be returned.
- If the conditional expression is not true, then the 'False' arrow on the right will be followed and the result `false` will be returned.
- After returning a result, a method always terminates (is done). A return is always immediately followed by the End.
- In an accessor method, the initial and final situation are always equal to each other.

Code for accessor method

The corresponding code looks like this:

```
boolean methodName( ) {          // a boolean accessor method
      if( checkCondition( ) ){  // check the conditional expression in the diamond
                                 //if the conditional expression is true
            return true;         // give the result 'true'

      } else {                   // if the conditional expression is not true
            return false;        // give the result 'false'
      }
}
```

The code explained:

- First the value of the conditional expression `checkCondition ( )` is determined.
- If the conditional expression is true, then `true` is returned. After that, the method terminates.
- If the conditional expression is not true, then you jump to `else`. Here `false` is returned. After that, the method terminates.

- After returning a value, nothing more happens. If you try to execute any code after a return, the compiler will complain with the following error message: "unreachable statement";
- The text to the right of
  are comments

Note

An accessor method provides information about an object. It should not change the situation. Therefore, from now on we agree that the initial and final situations of an accessor method are equal to each other.

Theory 2.8: Loading a world from a file

To open a new world within your current scenario follow this example. In the next steps we will open the world 'worldEgg6CellsAhead':

1. Right-click on the world (make sure that you click on an empty cell).
2. Choose `void populateFromFile( )`.

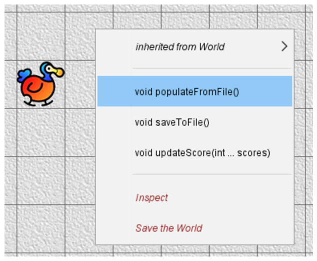


Figure 4: Populating the world

3. Go to the folder 'worlds'.
4. Choose the corresponding file, in this example 'worldEgg6CellsAhead.txt'.

Theory 2.9: Repetition is... boring and error-prone: wishing for a generic solution

Have a look at the scenario below. How would you help Mimi find her egg?

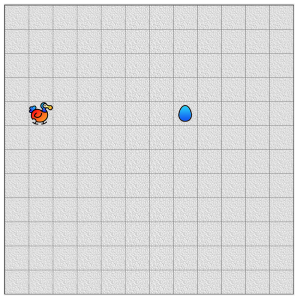


Figure 5: Scenario: many moves needed to reach the egg

Naturally, you could solve this problem by merely calling the method `move( )` a certain number of times. However, imagine that Mimi would have to take 1003 steps to get to her egg? As a programmer, you would be very busy typing (or copy-pasting) the `move( );` statement 1003 times. This strategy has a few drawbacks:

- It's a lot of typing or copy-paste work (which is rather boring).
- You might accidently call the `move( )` method 1004 times instead of 1003 times. The result: your program won't work correctly.
- Your program only works for that one specific situation. It is not flexible or general. For example, it will not work if 42 steps are needed.

To make your program more general and work in several similar situations, you'll have to devise a smarter generic algorithm. What you actually want is a repetition:

"While Mimi has not found her egg, she must take a step."

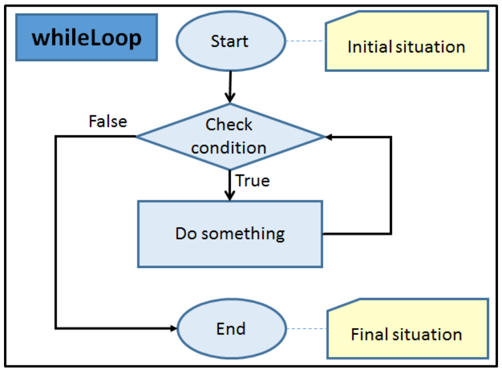
In the final situation, she'll be done when she finds her egg. Note that this description has no ('hard-coded') numbers in it. Such a repetition look is called a `while`-loop.

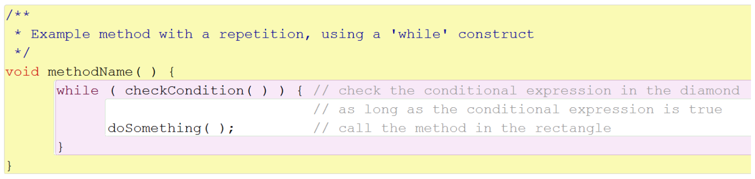
A general algorithm which can be used in multiple (initial) situations is called generic. This does not solve one particular problem, but can be used to solve many similar problems. You will learn more about generic algorithms in later assignments.

Theory 2.10: Repetition using `while`

The flowchart and code below show the use of a `while` construct to repeat a certain block:

"As long as a particular conditional expression is `true`, repeat something."





```
/**
 * Example method with a repetition, using a 'while' construct
 */
void methodName( ) {
      while ( checkCondition( ) ) { // check the conditional expression in the diamond
                                    // as long as the conditional expression is true
            doSomething( );         // call the method in the rectangle
      }
}
```

Flowchart and code explained

- If the `condition` is true, then `doSomething` is executed.
- Otherwise (the `condition` is false) the method is done.

Note:

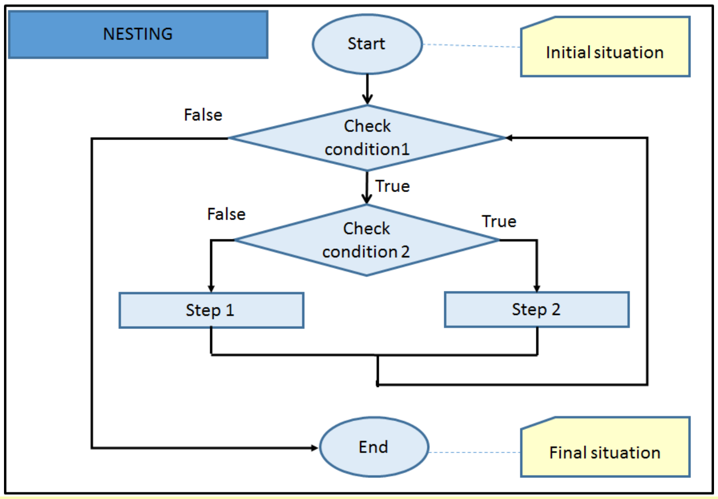
- The condition is tested at the beginning of the loop.
- A "NOT" (negation) is often used in the conditional expression of a `while`, for example, "NOT borderAhead". This is consistent with the way in which you would describe the algorithm in the words: "while something is NOT the case, then ...". In code, '!' means NOT, for example `! borderAhead( )`.
- The method `doSomething( )` must at some point make the conditional expression become 'false'. At that moment, the method (and thus the repetition) stops. A common mistake when using a `while` loop is to forget this, the condition remains 'true' and the loop never ends. This is called an infinite loop.
- Using a repetition until a certain condition is met is called a primed sentinel.

Theory 2.11: Nesting

The language construct for sequence, selection (choice), and repetition can also be used together in various combinations. They can be used sequentially, one after the other, or nested in each other. A sequence, selection or repetition can be used in any order. Any one of them can also be used within any other one.

Example: `if` .. `then` .. `else` nested in a `while` loop

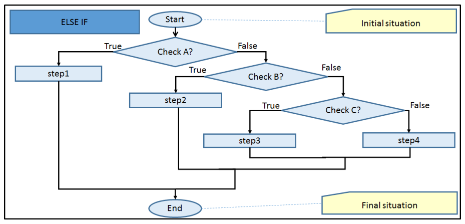
The following is an example an `if` .. `then` .. `else` statement (selection) nested in a `while` loop (repetition).



```
void nestedIfThenElseInWhile( ) {
    while ( checkCondition1( ) ){   // while condition is true, repeat the following
        if ( checkCondition2( ) ){ // if the 2nd cond. exp. is also true, then..
            step2 ( );             // follow true branch
        } else {                   // if the 2nd cond. exp. is not true, then..
            step1 ( );             // follow false branch
        }
    }
}
```

Example: nested `if` .. `then` .. `else` statements

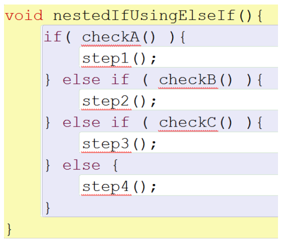
A nested *if* .. *then* .. *else* statement tests several cases simultaneously.



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

Simplified nested `if` .. `then` .. `else`, using `else` .. `if`

This type of `if` .. `then` .. `else` nesting can simplified using an `else` .. `if`. The `else` and the `if` are combined and becomes:



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

Figure 6: A nested `if .. then .. else` using `else if`

- 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.

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.

Challenges

Please read Theory 2.1: Flowcharts.

Please read Theory 2.2: Steps for designing code.

Please read Theory 2.3: Sequence of Instructions.

Please read Theory 2.4: Testing (and debugging).

Challenge 2.1: Sequence of instructions: `gotoEgg`

Our MyDodo, Mimi, has lost her egg. Can you help her find it?

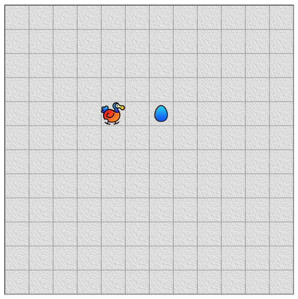
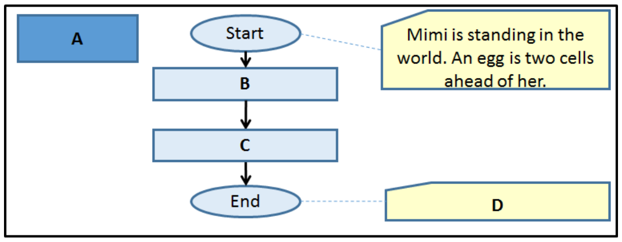
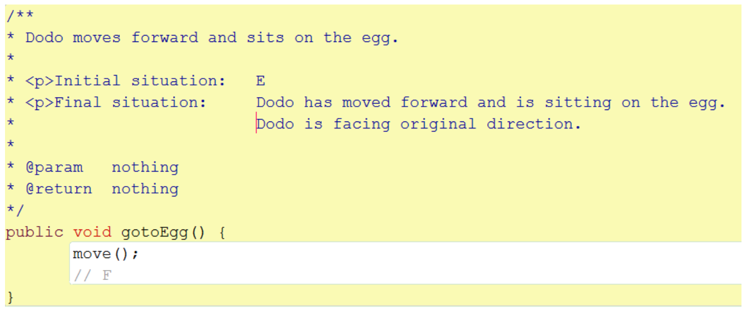


Figure 7: First scenario: Help Mimi find her egg

a) Right-click on Mimi. Using these methods (not the methods in the inherited lists), develop a strategy for Mimi to go to her egg and sit on it.

b) Fill in the missing parts A,B,C,D (in the flowchart) and E and F (in the code):


A
Start
Mimi is standing in the world. An egg is two cells ahead of her.
B
C
End
D



```
/**
* Dodo moves forward and sits on the egg.
*
* <p>Initial situation:   E
* <p>Final situation:     Dodo has moved forward and is sitting on the egg.
*                         Dodo is facing original direction.
*
* @param   nothing
* @return  nothing
*/
public void gotoEgg() {
        move();
        // F
}
```

c) Open the code for MyDodo in the editor. Modify the `gotoEgg( )` method accordingly. Also correctly describe the initial and final situation as (JavaDoc) comments.

d) Test your method. Tip: If it doesn't work correctly or if you get a compiler-error, then follow the steps described in the 'Debugging' theory block 2.4.

Challenge 2.2: A sequence with multiple instructions

In this scenario, Mimi is further away from her egg. Can you help Mimi find her egg this time?

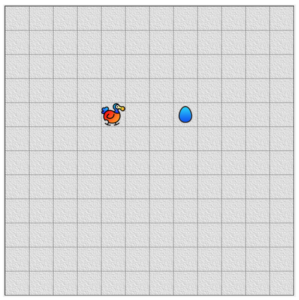


Figure 8: Second scenario: Help Mimi find her egg again

a) Adjust the world so that it matches figure 8.

b) Can you help Mimi find her egg this time? (Again, you may not use methods from the inherited list). Modify the flowchart from Challenge 2.1 and corresponding code in MyDodo's `gotoEgg( )` method.

c) Also change the comments appropriately.

d) Test your method.

Please read Theory 2.5: Comparing.

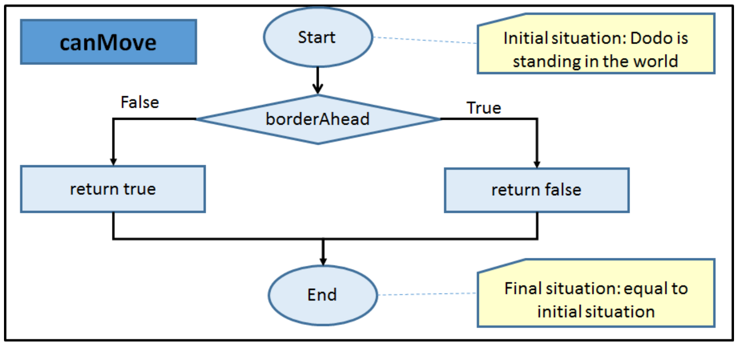
Please read Theory 2.6: Selections (choices) using `if..then..else`.

Please read Theory 2.7: Accessor method.

Challenge 2.3: Mimi can't walk through fences

In assignment 1 we saw that Mimi can't step outside of the world. We will now have a look at the method `boolean canMove( )` again.

a) Have a look at the following flowchart and corresponding code for `canMove( )`:


canMove
Start
Initial situation: Dodo is standing in the world
False
borderAhead
True
return true
return false
End
Final situation: equal to initial situation

```
/**
 * Test if Dodo can move forward,
 * i.e. there are no obstructions or end of world in the cell in front of her.
 *
 * <p> Initial:   Dodo is somewhere in the world
 * <p> Final:     Same as initial situation
 *
 * @return  boolean true if Dodo can move (thus, no obstructions ahead)
 *                  false if Dodo can't move
 *                      there is an obstruction or end of world ahead
 */
public boolean canMove( ) {
    if ( borderAhead( ) ){
        return false;
    } else {
        return true;
    }
}
```

b) The way she is now programmed, Mimi doesn't seem to care about the fences. She walks right through them! Place a fence in the world and see for yourself!

c) Call the `boolean fenceAhead( )` method. What does it do? Call the method with a fence in front of Mimi, and again without a fence in front of Mimi.

d) Change the conditional expression (the diamond) in the flowchart to ensure that Mimi can't walk through fences and can't step out of the world. Tip: In code 'AND' is written as '&&', 'OR' as ' || ', 'NOT' as ' ! '.

e) Modify the code and its comments.

f) Test your method. Tip: If it doesn't work correctly or if you get a compiler-error, then follow the steps described in the 'Debugging' theory block 2.4.

Please read Theory 2.8: Loading a world from a file.

Challenge 2.4: Writing your own sequence: `climbOverFence( )`

We're now going to teach Mimi something new. If she encounters an obstacle (for example a fence), she Mimi should climb over it.

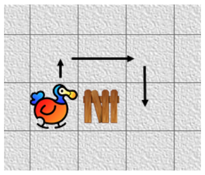


Figure 9: Scenario: Climb over the fence

We do this as follows:

a) Open the "worldFenceObstructing" world. Tip: To load another world, see Theory 2.8.

b) Mimi will stop if she encounters a fence. Check this. If Mimi walks right through the fence, then check if you did Challenge 2.3 correctly before continuing.

c) If Mimi encounters a fence, we want her to climb over it, as in figure 9. Complete the following strategy:

i. turn to the left
ii. take a step
iii. ...

d) Make sure Mimi is facing East again when she is finished.

e) Write a method in `climbOverFence( )` which matches your strategy (not using any methods from the inherited list). Describe the initial and final situation in comments.

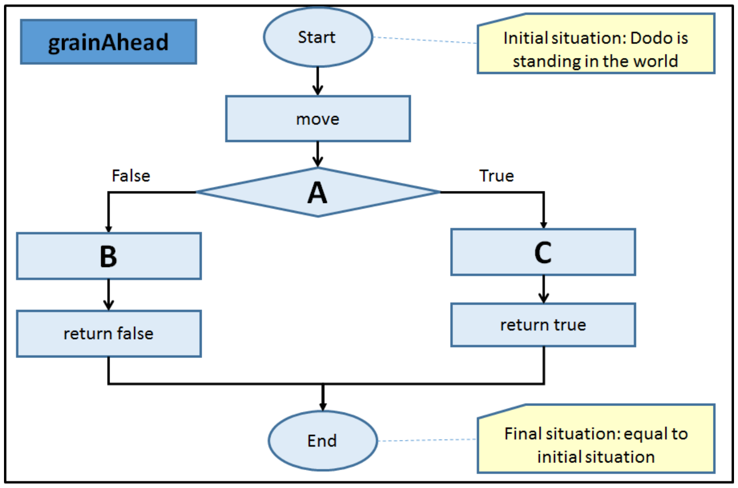
f) Test your method.

Challenge 2.5: Find grain

Mimi is near-sighted. She's hungry and looking for grain. However, she can only spot a grain if she is standing on top of it. Can you help her figure out if there is grain in front of her?

a) Have a look at the flowchart and code for the method `grainAhead( )`. This method returns `true` if there is a grain in the cell in front of Mimi, and `false` otherwise. Note that:

- This accessor method should have a final situation identical to the initial situation. To make it easier to use this method for other tasks, Mimi must return to her original position (and facing the same direction).
- A `return` is the very last thing that can be called in a method (the method stops after that). So, Mimi must go back to her initial situation before the `return`.
- Parts B, C, E and F in the code and flowchart may consist of more than one statement.


grainAhead
Start
Initial situation: Dodo is standing in the world
move
False
A
True
B
C
return false
return true
End
Final situation: equal to initial situation

```
/**
 * Test there is a grain in the cell in front of Dodo
 * <p>
 * Initial situation:   Dodo is somewhere in the world
 * Final situation:     Same as initial situation
 *
 * @param   nothing
 * @return  boolean true if there is a grain in the cell in front of Dodo
 *                  false else otherwise
 */
public boolean grainAhead() {
    D
    if ( onGrain() ){
        // Dodo goes back to initial situation before returning value
        E
        return true;
    } else {
        // Dodo goes back to initial situation before returning value
        F
        return false;
    }
}
```

b) Fill in the blanks A through F (in both the flowchart and the code).

c) Write and test the method `grainAhead( )`.

Please read Theory 2.9: Repetition is... boring and error-prone: wishing for a generic solution.

Please read Theory 2.10: Repetition using `while`.

Challenge 2.6: Repeating using `while`: generic `gotoEgg( )`

Your mission is:

"Help Mimi find her egg, no matter how far away she is from it".

Your solution must be generic. The following initial situation is given:

- Mimi is 0 or more cells away from her egg, but you don't know exactly how many;
- Mimi is facing in the correct direction (she does not have to turn, she only has to take a certain number of steps forward);
- There is nothing in between Mimi and her egg (for example, there is no fence blocking her way);
- Mimi has found her egg if she is standing in the same cell as the egg.

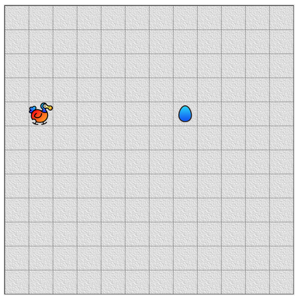


Figure 10: Scenario: a few moves needed to reach the egg

We are going to work on this problem step-by-step:

a) Open the world named "worldEgg6CellsAhead".

b) Argue, in terms of a repetition block and conditional expression, that the following generic algorithm is correct:
"While Mimi has **not** found her egg, she must step forwards."

c) Have a look at the flowchart in figure 11. What must be repeated? Fill in B.

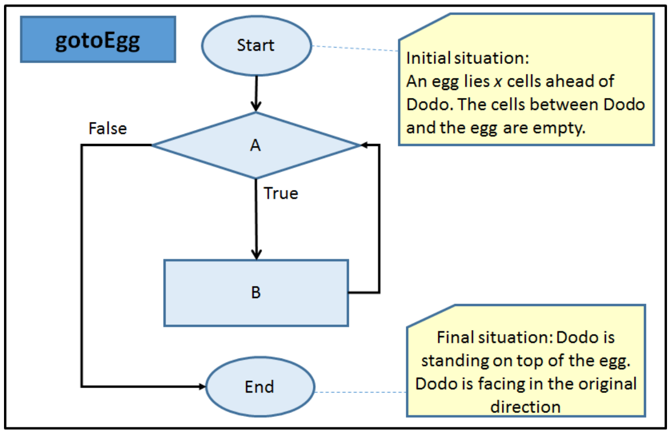
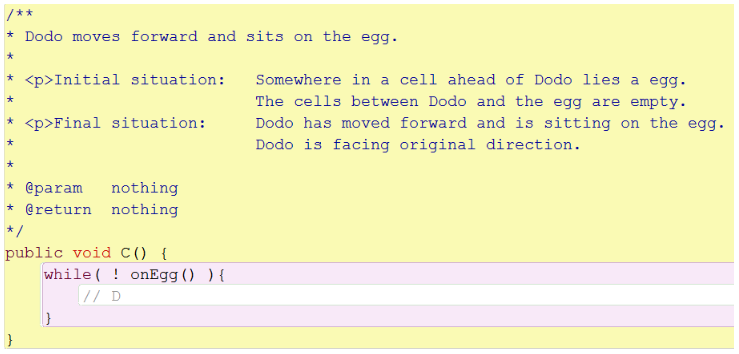

gotoEgg
Start
Initial situation:
An egg lies *x* cells ahead of Dodo. The cells between Dodo and the egg are empty.
False
A
True
B
Final situation: Dodo is standing on top of the egg. Dodo is facing in the original direction
End

Figure 11: Flowchart "While the egg has **not** been found, step forwards."

d) What is the conditional expression in A?

e) Now modify the code for `MyDodo`'s method `gotoEgg( )` so that it is generic. It should look something like figure 12. Note that '!' in code means **not**. Copy the code and fill in C and D.



```
/**
 * Dodo moves forward and sits on the egg.
 *
 * <p>Initial situation:   Somewhere in a cell ahead of Dodo lies a egg.
 *                         The cells between Dodo and the egg are empty.
 * <p>Final situation:     Dodo has moved forward and is sitting on the egg.
 *                         Dodo is facing original direction.
 *
 * @param   nothing
 * @return  nothing
 */
public void C() {
    while( ! onEgg() ){
        // D
    }
}
```

Figure 12: Flowchart "While the egg has **not** been found, step forwards."

f) Modify the comments above the `gotoEgg( )` method accordingly.

g) Test your program.

Challenge 2.7: Walk to the edge of the world

Write a method that makes Mimi walk to any edge of the world. Her initial position is arbitrary: she can be standing anywhere and facing any direction.

a) Load the world: "worldEmpty" (See Theory 2.8).

b) Fill in the blanks in the flowchart.

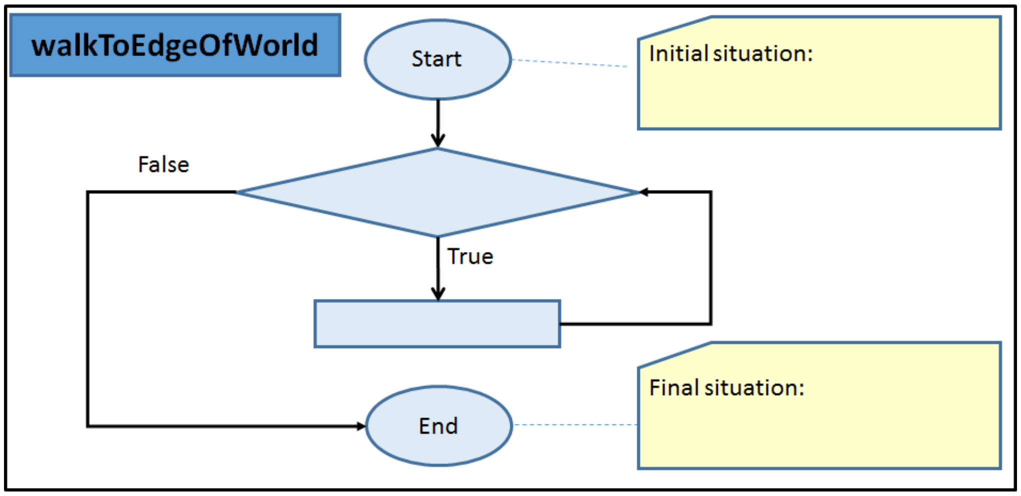


Figure 13: Flowchart for `walkToWorldEdge ( )`

c) Write the corresponding method `void walkToWorldEdge ( )`. Include comments.

d) Test your method by right-clicking on Mimi. Repeat with Mimi in different positions in the world.

e) Does your code also work if Mimi is facing any other direction (West or South)? If necessary, modify the comments to properly explain what Mimi can do.

Please read Theory 2.11: Nesting.

Challenge 2.8: Combining sub-methods: `walkToWorldEdgeClimbingOverFences ( )`

We are now going to combine two of your methods so that Mimi can walk across her world, avoiding any fences she comes across (see figure 14).

- `climbOverFence ( )` (from challenge 2.4)
- `walkToWorldEdge ( )` (from challenge 2.7)

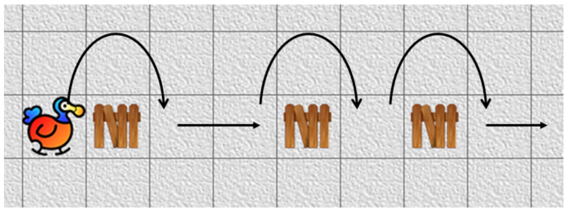


Figure 14: Scenario: Walk to edge of the world, climbing over fences

a) Modify your flowchart for `walkToWorldEdge( )` (from challenge 2.7) so that Mimi climbs over any fences she encounters. Also rename your flowchart.

b) Write a method `walkToWorldEdgeClimbingOverFences( )`. You may assume there is always an empty cell in between two fences. Tip: You can call your `climbOverFence` method using: `climbOverFence( );`

c) Test your method.

Challenge 2.9: Giving compliments

When Mimi completes her task, we want to give her a compliment on her job well-done. Modify the code for `walkToEdgeOfWorldClimbingOverFences( )` to show a compliment. Use the method `showCompliment( String compliment )` as follows:

Code:
```
showCompliment("Congradulations!");
```

Result:

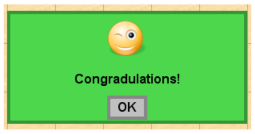

Congradulations!

Figure 15: Example code and its result for giving a compliment

Challenge 2.10: Pick up grains and print coordinates

Your mission is:

"Have Mimi walk across the row, picking up any grain she finds. Print the coordinates of each of the grains."

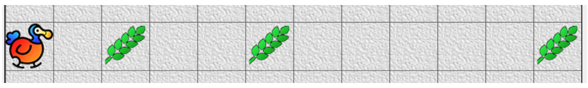


Figure 16: Pick up grains in this row and print their coordinates

- Load the world 'worldGrainsInRow'.
- In the console, print the coordinates of each of the grains in Mimi's row. Tip: you can use `int getX()` and `int getY()` to determine the coordinates of Mimi. Besides, have a look at Mimi's grain-related methods.
- Test your code. Does it also print the coordinates in the last cell?

Reflection

In this assignment you have been introduced to algorithms. In an algorithm you describe how a particular task should be done. You explain each step, choice or repetition very precisely. In the last challenge you came up with your own algorithm, visualised it in a flowchart and wrote and tested the code. One of the most important steps in becoming good at anything is to evaluate and reflect on what you did and how it went:

Result

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

Method

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

Fill the following table with smileys indicating how things went.

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

😶	I can come up with a generic solution to a problem
😶	I can devise an algorithm as a sequence of steps, choices (`if .. then .. else`) or repetitions (`while`)
😶	I can write new code in a structured manner by visualising an algorithm in a flowchart and then translate this into code
😶	I can combine existing solutions (sub-methods) to solve a more complex problem
😶	I can make code changes incrementally by making small changes and testing directly after every minor modification

Diagnostic Test

1. Assume `MyDodo` has the following method: `boolean foundAllEggs( )`, which indicates whether Mimi has found all of her eggs. Which of the following is true?
   (a) This method has a `boolean` parameter.
   (b) The initial and final situations of this method are equal to each other.
   (c) This method has a `boolean` as a result.
   (d) This is a mutator method.
2. Name two advantages for using sub-methods.
3. Give two reasons for testing immediately after each minor code change.
4. Explain, in your own words, what the initial and final situations described in the flowchart are useful in programming.
5. Have a look at the flowchart in figure 17. Write the code for this silly walk. For which initial situations will the program stop? Include a description in your (JavaDoc) comments.

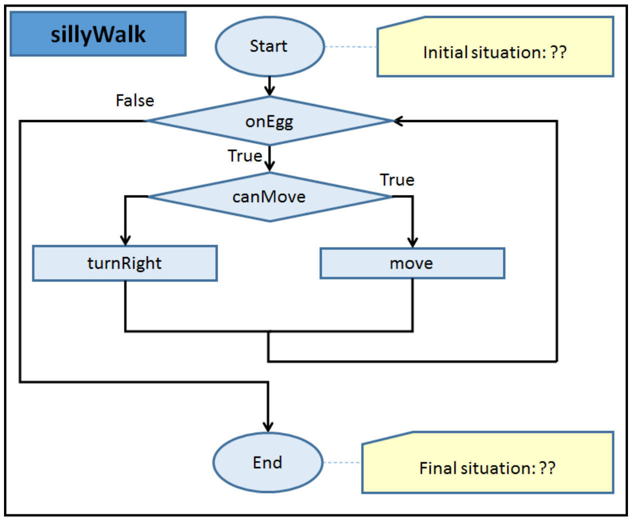


Figure 17: Flowchart for silly walk

Saving and Handing in

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

Handing in

Hand in the following:

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