

Assignment 5: Dodo Plans

Algorithmic Thinking and Structured Programming (in Greenfoot)

© 2017 Renske Smetsers-Weeda & Sjaak Smetsers¹

Contents

Introduction	1
Learning objectives	1
Instructions	1
Theory	2
5.1 The Sum Plan	2
5.2 The Min/Max plan	4
5.3 Type casting	5
Challenges	7
5.1 Counting eggs in the world	7
5.2 Find the row with the most eggs	7
5.3 Monument of eggs	8
5.4 Strong monument of eggs	8
5.5 Pyramid of eggs	8
5.6 Average number of eggs per row	9
5.7 Bit Parity	10
5.8 Generic Bit Parity	12
Reflection	13
Saving and Handing in	14

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

Introduction

In the previous assignments you came up with your own solutions and implemented those. You also learned how to apply plans and come up with generic solutions. In this assignment you will teach Mimi to do more complex tasks based on more complex plans.

In this assignment you are going to learn which typical plans computer scientists use and use those plans to solve your own problems. In order to do this, you will have to integrate the things that you have learned in the previous assignments.

This assignment's goal is: **Learn to apply plans.**

Learning objectives

After completing this assignment, you will be able to:

- recognize and apply **plans** for determining max/min values, sums and averages;
- apply **type-casting**;
- decompose a complex problem down into subproblems which can be dealt with separately;
- design and implement **generic algorithms**.

Instructions

In this assignment you will carry on with your code from the previous assignment. Make a copy of that scenario to continue working with. To make a copy follow the next steps:

- Open your scenario from the previous assignment.
- 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 the current assignment number, for example:
Asgmt5_John.

Note: We recommend that you to continue working with your own code. If it is **absolutely** impossible to carry on working with your own code from the previous assignment, then you may download a new scenario from the course website².




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.java` contains all your code and must be handed in;
- The reflection sheet: complete and hand it in.

²<http://course.cs.ru.nl/greenfoot/>

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

There are three types of challenges:

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

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

A note in advance:

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

Theory

Theory 5.1: The Sum Plan

Summing values is a frequently practiced programming activity. It isn't very much work to implement, but can easily be messed up. Its plan is similar to the *Counting Plan* in Theory 4.7. A *sum plan* has the following structure:

- Initialize a variable for keeping track of the sum to zero;
- Determine when done, i.e. all elements have been traversed;
- Inside the loop, determine what value needs to be added to the sum (the running total) and modify the sum accordingly;
- At the end of the method, return the sum value.

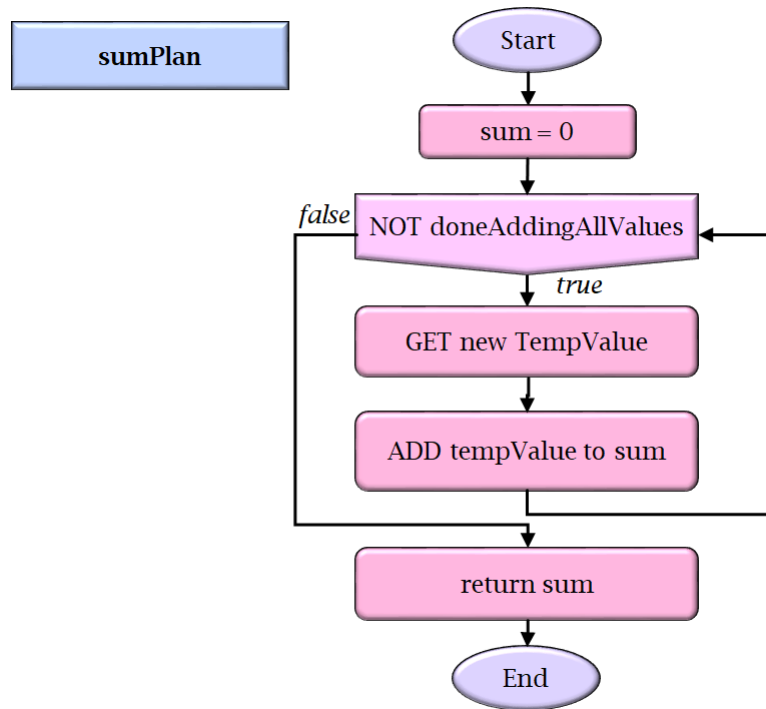


Figure 1: Flowchart for finding the sum of several values

```

/**
 * Framework code for finding the sum of several values.
 */
public int sumPlan( ) {
    int sum = 0;           // set sum to 0
    while ( elements left? ) { // check if more values must be considered
        int currentValue = ... // get the value of the current element
        sum += currentValue;    // add the current value to the sum (running total)
        ...
    }
    return sum;
}
  
```

Example:

As a concrete example, let's have a look at MyDodo's code for sumEggValues().

```

/**
 * Example of a sum plan: summing all the egg values in a row
 */

public int sumEggValues( ) {
    int sumOfEggValues = 0;    // set counter to 0

    while ( !borderAhead( ) ) { // check if more steps can be taken
        Egg currentEgg = getEgg(); // get the Egg which Dodo is standing on
        int currentEggValue = currentEgg.getValue(); // get the value of the current egg
    }
  
```

```
        sumOfEggValues += currentEggValue;           // add the egg value to the sum (
            running total)
        move();                                       // move to the next cell
    }

    return sumOfEggValues;
}
```

Mimi gets the values of any eggs she finds and sums their values together. The variable `sumOfEggValue` is used to keep track of the sum (or running total). For each egg she finds, she determines its value (`currentEggValue`) and adds it to the current sum, using: `sumOfEggValues += currentEggValue;` (which means the same as `sumOfEggValues = sumOfEggValues + currentEggValue;`).

Theory 5.2: The Min/Max plan

To find the minimum or maximum of some items, it is not necessary to keep track of all items, just the current min/max at any stage. For this plan, follow these steps:

- a) Declare the variables needed:
 - i) The value of the current item.
 - ii) The minimum/maximum thus far.
- b) Use the first value encountered as the initial value for the min/max;
- c) Traverse all items using a `while`-loop (or `for`-each-loop)
- d) Compare the current item to the current min/max.
- e) If searching for a maximum and the current item is greater than the current maximum, then the current value is to be assigned as the new current-maximum.

The max plan is depicted in the following flowchart:

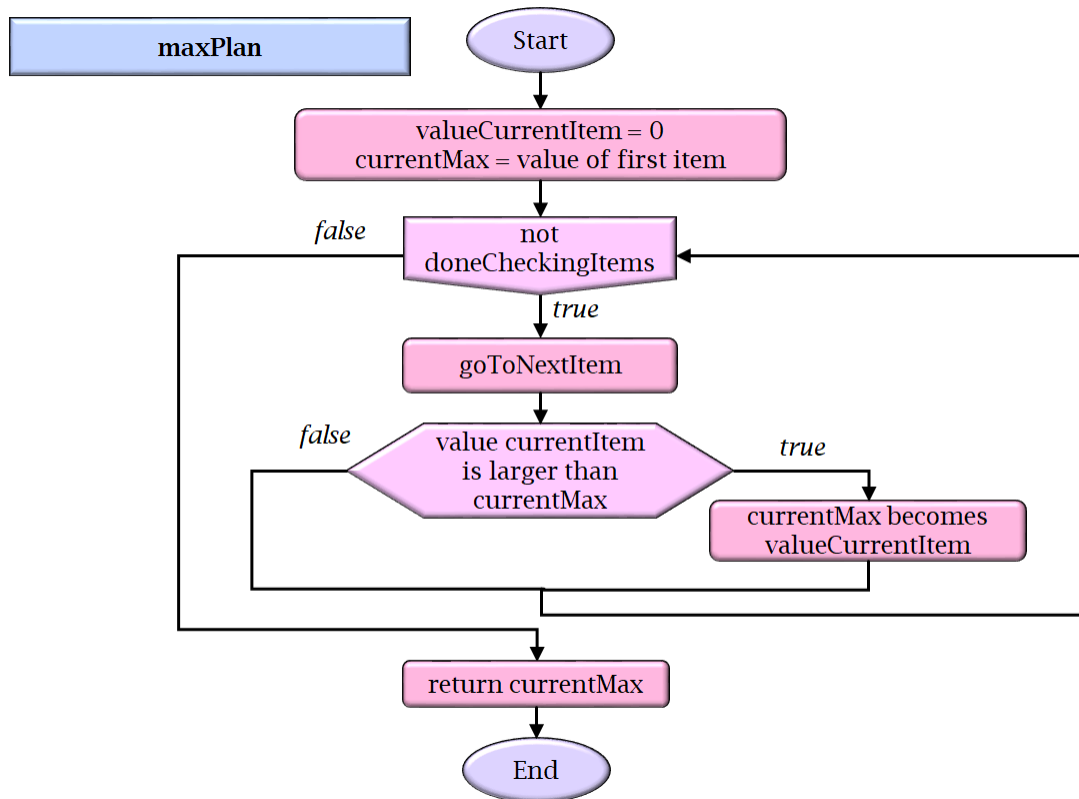


Figure 2: Flowchart for a Max Plan

The following example shows how to find the most number of eggs in a row. The rows in the world are traversed using a sentinel controlled loop (using boolean `doneCheckingRows`).

```

public int findMaxEggsInARow(){
    //declare variables needed
    int nrOfEggsInCurrentRow = 0; // value of current row
    int currentMaxEggs = 0; // the maximum thus far

    // use the first value encountered as the initial value for the max
    currentMaxEggs = countEggsInRow();

    while( !doneCheckingRows() ){ // check if there are rows still to be checked
        stepDownToNextRow(); // go to next row

        nrOfEggsInCurrentRow = countEggsInRow(); // get value of current row
        if( nrOfEggsInCurrentRow > currentMaxEggs ){ //compare current to max
            currentMaxEggs = nrOfEggsInCurrentRow; // if current is the max, store the
            new value
        }
    }
    return currentMaxEggs;
}
  
```

The `currentMaxEggs` variable is used to store the current maximum and it is initialised to 0 which is the smallest value possible. Note that each value is compared with the current maximum. Where a value is found to be greater than the current maximum it replaces the current maximum and is used for future comparisons.

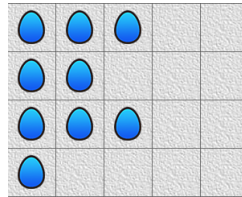


Figure 3: Multiple occurrences with the same max value

In some situations, multiple occurrences of the same minimum or maximum value may be present. For example, figure 3 shows two rows with the same max (3 eggs in a row)...so which row has the most eggs in it? There is no absolute answer to this question. Depending on the problem at hand, you must choose an appropriate solution. Often this means choosing either the first or the last maximum occurrence.

Theory 5.3: Type casting

In some cases a value can be converted from one type to another type. This is called *type casting*. Using brackets, the new type is written in front of the object.

When dividing two numbers (ints), the answer may be a decimal (double) value. For example $7:4=1.75$. Without casting, dividing two ints automatically results in an int, not a double. For example:

```
int number1 = 7;
int number2 = 4;

System.out.println( number1 / number2 );
```

will result in '1' being printed. Why? Because number1 and number2 are ints, and thus so is its answer. 7 is divided by 4 which is then rounded down to the nearest whole number, so actually, the decimal part is just 'chopped off'.

However, dividing a double by an int does result in a double. So, when dividing, by type casting the numerator to a double, the result will be a double. So, `System.out.println((double)number1 / number2);` will print 1.75 to the console.

```
int number1 = 7;
int number2 = 4;

System.out.println( (double) number1 / number2 );
```

Example

To cast the value of `int nrEggsFound` to a double (a decimal value), we write: `(double)nrEggsFound`.

Challenges



Please read **Theory 5.1: The Sum Plan.**



Challenge 5.1: Counting eggs in the world

Now that you have written a few basic, but very important types of methods, it's time to combine that knowledge in a more challenging exercise.

Your challenge: "We want Mimi to count the number of eggs in the world and return this value."

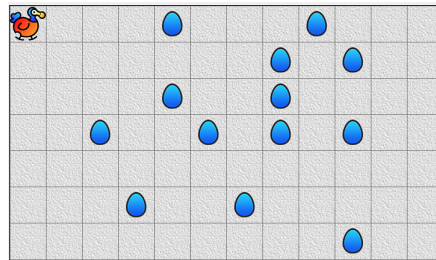


Figure 4: Initial situation: Mimi starts in top-left corner to count eggs in the world

To make programming and especially testing, easier, we divide the problem down into a few sub-problems. Then deal with each subproblem, one-by-one, only focussing on that particular problem. The solutions are then combined into one complete solution.

- a) Determine which subproblems you have:
 - Count the eggs in each row;
 - Add the number of eggs in a row to the total number of eggs found in the entire world;
 - Go to the next row;
 - Walk back to initial position;
 - ...
- b) Sketch a high-level flowchart for the complete solution. Determine the initial and final situations. Are they equal? They should be.
- c) Write and test code for each sub-problem. Tips:
 - **Re-use:** Always try to make use of methods you have already written (in previous assignments). All you have to do is call the method. That will save you a lot of time, and besides, you have even already (debugged and) tested those methods!
 - **Generic:** Make your solution generic (see 4.5). This way, your code will work in a different worlds and initial situations too.
 - **Print:** Throughout the code, for testing purposes, print values to the console such as the (total) number of eggs or the eggs in the row.
- d) Combine the sub-methods to come up with a complete solution. Test this.

You have now found a solution to a larger problem by breaking it down into sub-problems (decomposition). You designed, implemented, and tested each individually (modularization) and combined these and existing methods to solve the problem as a whole.



Please read **Theory 5.2: The Min/Max plan.**



Challenge 5.2: Find the row with the most eggs

We're going to teach Mimi how to find which row has the most eggs in it. When she's finished, print the row number with the most and how many eggs there are in that row to the console. Mimi should also return back to the top-left corner again. The method must return the row number with the most eggs.

- Come up with an appropriate algorithm. Decide which variables you will need to keep track of everything.
- Draw a flowchart, showing your algorithm on a high-level.

Tips:

- Mimi should start and end in the top-left corner.
 - Re-use methods which you have already written and tested.
 - Make your solution generic (see 4.5).
 - Consider an appropriate solution if there is more than one row with the same maximum value.
- Test each sub-method separately, then test your solution as a whole.
 - Take a moment to look back and reflect. Did you make a lot of mistakes while programming your solution? What kind of mistakes did you make? What could you have done better?

You have now learned how to use the Min/Max Plan.



Challenge 5.3: Monument of eggs

Teach Mimi to build the following monument of eggs:

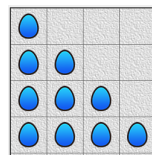


Figure 5: Monument of eggs

- Fill as much of the world with this pattern as possible.
- Come-up with a generic solution. You may not use hard-coded values, such as '3' or '4'.



Challenge 5.4: Strong monument of eggs

Teach Mimi to build the following monument of eggs:

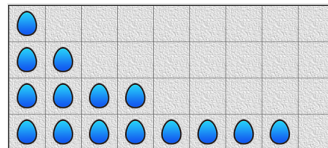


Figure 6: Sturdy monument of eggs

- Fill as much of the world with this pattern as possible.
- Come-up with a generic solution. You may not use hard-coded values, such as '3' or '4'.



Challenge 5.5: Pyramid of eggs

Teach Mimi to build a pyramid of eggs:

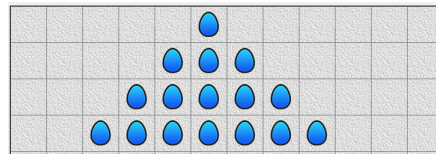


Figure 7: Pyramid of eggs

- Fill as much of the world with this pattern as possible.
- Come-up with a generic solution. You may not use hard-coded values, such as '3' or '4'.



Please read **Theory 5.3: Type casting**.



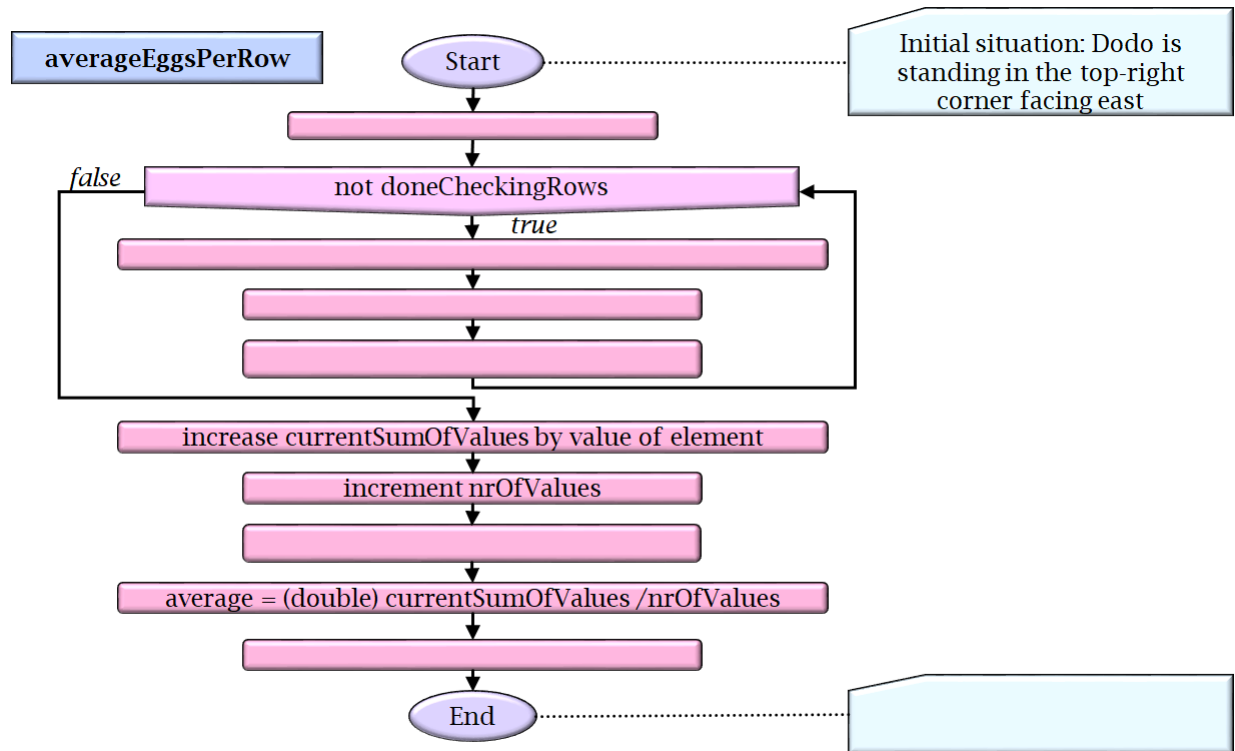
Challenge 5.6: Average number of eggs per row

Mimi knows how to count eggs, and the number of eggs in each row. But how many eggs are there on average in each row? This exercise is similar to the previous one. However, because average is (probably) a decimal value, you must use *type casting*.

"Mimi walks through the world and returns the average number of eggs per row."

Use the steps and flowchart below to help you get started:

- Assume Mimi starts in the top-left corner, facing East.
- Write the formula for calculating an average.
- Determine which variables you need to keep track of in order to calculate the average number of eggs per row.
- When calculating the average, type cast the variable for the total number of eggs found to a double (a decimal value).
- Test your method. Pay careful attention to eggs in the first and the last row.
- Ensure that the initial and final situations are the same.



Challenge 5.7: Bit Parity

Have a look at this video: <https://www.youtube.com/embed/0Xz64qCjZ6k?rel=0>. The magician can figure out which card, out of dozens, has been flipped over while they weren't looking. The magic in the trick is actually computer science. Computers use the same kind of technique to detect and correct errors in data.

When messages are transmitted from one computer to another, errors can occur. In some cases, data is lost. In other cases, data is corrupted. *Bit Parity* is an error detection mechanism. By adding several extra bits it can (in some cases) determine whether a bit has been lost or damaged, and even correct it. Your task is to teach Mimi the Bit Parity algorithm.

Bit Parity algorithm explained

We want to make sure that each column and each row has an **even** number of eggs (or bits). That way, if later on we find a row or column with an odd number of eggs (or bits), we know that the world has been damaged. So, if a row has an odd number of blue eggs, we add a golden egg in the last column. We do the same for the rows.



Figure 8: Original world (before any damage)



Figure 9: Golden parity eggs added to original world (before any damage)

Error detection

When checking for an error we figure out if a row or kolom has an even or odd number of eggs. We count both golden eggs and blue eggs.

Here are the possibilities:

- **Missing egg:** We can detect where the egg went missing by checking which row and column have an odd number of (blue and golden eggs) eggs. In this case, the third row and fourth column. The egg has been stolen from coordinates (4,3).
- **Extra egg:** An extra egg being placed in the world 'illegally' also 'damages' the world. Error detection is the same as with a missing egg.
- **No error:** If all rows and columns have an even number of (golden and blue) eggs, than no error has occurred (or our algorithm merely isn't smart enough to detect it).

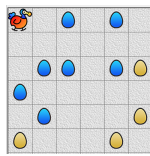


Figure 10: Damaged world (stolen egg)

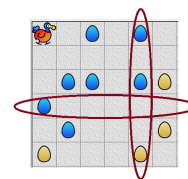


Figure 11: Damaged cell located

Error correction

In this case the error can be fixed by placing another egg at (4,3). In other cases, an egg may have to be removed.

Error-fixing in Mimi's world

Your mission is to teach Mimi how to perform the Bit Parity algorithm to check whether the world has been 'damaged' and fix it. The world can be 'damaged' by an egg being either stolen or an egg being placed in the world 'illegally'.

- Draw a high-level flowchart describing the algorithm. Tip: consider using sub-methods such as:
 - `countEggsInRow` (you already have this one)
 - `getIncorrectRowNr` (when facing South, this can also be used to find an incorrect column number)
 - `gotoIncorrectBit`
 - `fixIncorrectBit`
 - `goToLocation` (you already have this one)
 - ...
- Write and test each sub-method individually.
- Test your method as a whole.
- Evaluate your solution. Does it also work when:
 - there world has not been damaged (i.e. there are no incorrect bits)?
 - an extra egg has been placed in the world?

- Mimi is not initially standing in the top-left corner?

Is there a smarter algorithm?

- e) Improve your solution.



Challenge 5.8: Generic Bit Parity

This challenge is about generic algorithms. In Challenge 5.7 you implemented the Bit Parity algorithm. However, Mimi has suddenly lost her feeling for direction. She no longer knows where *North* is. This also means you can't use any compass-related methods such as `facingEast()` or `goToLocation`.

Challenge: *Teach Mimi a generic Bit Parity algorithm, without using direction.*

- a) Adjust your solution from the previous Challenge accordingly.
- b) Explain what the advantages are of your new algorithm. What makes this one better than using a compass-related methods?
- c) Can you re-use the method for counting eggs in rows to count the eggs in columns? Implement this improvement.
- d) Now that you have implemented this improved, more generic algorithm, do you see any other improvements? Implement these.
- e) In which cases (initial situations) will your solution not work?

Reflection

In this assignment you practiced applying plans. You also practiced describing an algorithm on a high level so that it becomes clear which subtasks you can tackle separately.

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 apply plans for determining min/max values, sums and averages.
	I can explain what type-casting is.
	I can describe an algorithm by drawing a high-level flowchart.
	I can identify sub-methods in an algorithm and design, write and test these separately.
	I can design and implement generic algorithms.

Saving and Handing in

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

Handing in

Hand in the following:

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