Assignment 5: A smarter Dodo

– Algorithmic Thinking and Structured Programming (in Greenfoot) –

1 Introduction

Sometimes it can be useful for an object to have 'memory'. For example, if Mimi could remember how many eggs she has laid. In this assignment you will learn how to use variables in order to store information. We're going to make Mimi smarter!

By making Mimi smarter, we can also get her do more complex (composite) tasks. It's difficult to use the repetition in the *Run* for complex tasks. Therefore, in this assignment you will write each algorithm in a separate method. In this method you can make use of **`while`**-loops where needed.

2 Learning objectives

After completing this assignment, you will be able to:

- explain what **variables** are used for;
- apply naming conventions to variables;
- identify and follow the steps (such as declaration and initialization) that are necessary when using a variable;
- **initialize** variables;
- use the **assignment operator** '=';
- use the **comparison operators** '==', '!=', '<', '<=', '>' and '>=';
- use the **arithmetic operators** '+, '-', '∗', '/' and '%';
- use the **incrementing operator** '++' (and decrementing operator '−−');
- explain in your own words what the role of a **counter** variable is in a **`while`** loop;
- determine the values which a variable acquires throughout the execution a (part of a) program (**tracing**).

3 Instructions

In this assignment you will carry on with your code from assignment 4. Therefore you will need a copy of the scenario which you saved after completing assignment 4, `Asgmt4_yourName`. To make a copy follow the next steps:

- Open your scenario from assignment 4, `Asgmt4_yourName`.
- In the Greenfoot menu at the top of the screen, select 'Scenario' and then 'Save As ...'.
- Check that the window opens in the folder where you want to save your work.

- Choose a file name containing your own name(s) and assignment number, for example: `Asgmt5_John`.

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

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

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

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

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

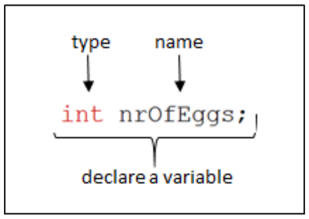
4 Theory

Variables
Variables are used to store information.

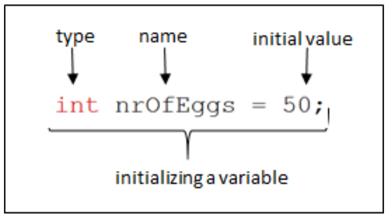
A variable has a:

- Type: such as **`int`**, **`boolean`**, `String`, or, as we shall see later on, a class or another object. The variable type describes what type of a value a variable can store.
- Name: for example `Mimi`, or `nrOfEggs`.
- Initial value: optional (you can also assign a value later).

Declaring a variable:
Before you can use a variable, it must be created. This is called a *declaration*:



Usually, at the same time, you will assign it a value. This is called *initialization*:



Assigning a value:
To assign a value to a variable you must use '='. The '=' sign is pronounced as 'becomes'.

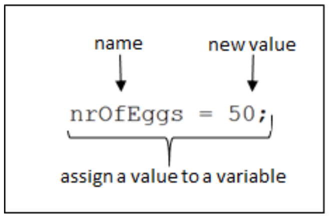


Figure 1: The variable `nrOfEggs` becomes 50

You can only assign a value to a variable if they both have the same **type**. This means that the type on the left-hand-side of the '=' should be the same as the type on the right-hand-side. In the example above, the type of:

- the left-hand-side is **`int`**, because the type of `nrOfEggs` is an **`int`**,
- the right-hand-side is also **`int`**, because "50" is an **`int`**.

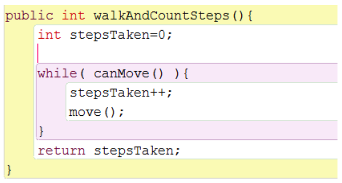
Using a variable:
When using (the value of) a variable, you don't need to mention the type anymore. (This is just like when you call a method, where you don't have to repeat the entire signature of the method. Moreover, Java does not even allow you to do so.) What you can do with a variable (which *operations* you can use) depends on its type. For example, variables which store a text (or `String`) can be 'stuck together' (or concatenated). And numerical values (such as **`int`**) can be compared, multiplied, added, incremented, etc.
Examples are:

- `nrOfDozens = nrOfEggs / 12;`
  the variable `nrOfDozens` is assigned the value of `nrOfEggs` divided by 12. Note: the division that is used here is called an *integer division*, that is to say that the result is an integer (whole number) and any remainder will be thrown away.
- `nrOfEggs = nrOfEggs + 1;`
  the variable `nrOfEggs` is incremented by 1 (for example, when Mimi lays another egg).
- `nrOfEggs = nrOfEggs ++;`
  the variable `nrOfEggs` is increased by 1 ('++' is the same as '+1').

A variable can also be passed to a method as a parameter. For example in the following method call:
`layNrOfEggs( nrOfEggs );`
Here, in the method call of `layNrOfEggs`, the variable `nrOfEggs` is passed as a parameter. Based on this parameter, Mimi knows how many eggs she must lay.

Life span:
A variable that is created inside a method declaration exists only for that particular method. After the method is executed, the variable is destroyed. Thus, these variables have a limited life span (also called *scope*). A variable's life begins where it is declared and ends when the stops. [1]

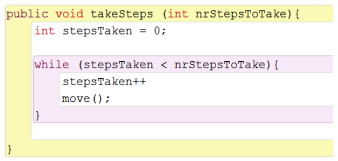
The Greenfoot editor helps you recognize the life span by using different background colors. For example, in the picture below you can see that the scope of the **`int`** `stepsTaken` variable is limited to the white area and all areas that are enclosed by the white area in which is declared, in this case the pink area.



```
public int walkAndCountSteps(){
    int stepsTaken=0;

    while( canMove() ){
        stepsTaken++;
        move();
    }
    return stepsTaken;
}
```

Parameters also have a life span: this is limited to the body of a method (or constructor). For example, in the picture below you can see that the scope of the **int** `nrStepsToTake` parameter is limited to the white area of the method (and the enclosed pink area).



```
public void takeSteps (int nrStepsToTake){
    int stepsTaken = 0;

    while (stepsTaken < nrStepsToTake){
        stepsTaken++
        move();
    }

}
```

Variables that are declared in a class (outside of a method and thus not within a particular method) exist as long as the object exists (you will learn more about this in the next assignment).

Naming conventions for a variable
The name of a variable:

- is meaningful: it corresponds to what the variable means (one exception: the name of a counter variable in a loop may be one letter, such as `i`);
- consists of one or more nouns;
- is written in lowerCaseCamel: it starts with a lowercase letter, an each subsequent 'word' starts with a capital letter;
- consists of letters and numbers: it does not contain spaces, commas, or other 'strange' characters (one exception: '_' may be used);
- for example: `nrEggsFound`.

Example:
We write a method which squares a number:

```
/**
* This method returns the square of a given number
*/
public int square (int number){
        // declare and initialize the result, the number squared
    int result = number*number;

        // the method returns the result after squaring
    return result;
}
```

Note:
A variable that is declared in a method (and thus can only be used within that method) is called a *local variable*. In this example, **int** `result` is a local variable because it has been declared in the method `square`. The parameter **int** `number` can only be used within this method.

Comparison operators:
The following operators compare numerical values:

Operator	Meaning	Example
==	is equal to	`number` == 4
!=	is NOT equal to	`number1` != `number2`
>	is larger than	`number` > 3
>=	is larger or equal to	`number1` >= `number2`
<	is smaller than	`number` < 5
<=	is smaller or equal to	`number` <= 5

The result of a comparison is always a **`boolean`** (thus, **`true`** or **`false`**).

Arithmetic operators:
The following operators perform arithmetic operations on numerical values:

Operator	Meaning	Example
+	addition	`result = number + 4`
−	subtraction	`result = number − 4`
∗	multiplication	`result = number ∗ 5`
/	division	`result = number / 5`

Increment and decrement operators:
The following operators increase or decrease the value of a variable by one:

Operator	Meaning	Example
++	increase by one	`result ++`
−−	decrease by one	`result −−`

Example:

- Initialization: Assume Mimi has found 4 eggs, then the following statement declares and initializes the variable: **`int`** `nrEggsFound` = 4;
- Addition operator: If she then finds another two eggs, then the following statement increases the value of `nrEggsFound` by two: `nrEggsFound` = `nrEggsFound` + 2;
- Decrement: And if she loses an egg, the following statement decreases the value by one: `nrEggsFound−−`;

The above operations change the value of a variable. To check whether Mimi now has five eggs, we use the following boolean expression: `nrEggsFound` == 5. This is indeed **`true`**.

Console
To make a window appear with text you can use a standard Java method: `System.out.println (String);`.

The method has a `String` (text) parameter. The window that appears is called a console. In assignment 3, exercise "Giving compliments" (exercise 5.5), you already practiced passing a `String` as a parameter. When using the `println` method:

- Place text in between quotation marks ' ' and ' '.
- To show the value of a variable (such as an integer), the quotation marks are not needed, just use the variable name.
- Combine texts and variables using ' + '.

Example:
To show text and the value of a variable (for example, `nrOfEggs`), call the following method:

```
System.out.println ("there are "+ nrOfEggs + " eggs in the world.");
```

In this example, if `nrOfEggs` is equal to 5, then the following will be shown in the console:
`There are 5 eggs in the world.`
Note:
For more information about the Java `println` method, have a look at: `https://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html#println(java.lang.String)`.

5 Exercises

5.1 Variables

5.1.1 Tracing code

Tracing
Keeping track of variables and reviewing their values at certain points in a program is called *tracing*. Tracing is a useful skill for detecting errors in code.
Example:
We will now trace the variables `number1`, `number2` and `number3` using the following code:

```
int number1 = 2;
int number2 = 3;
int number3 = number1 * number2;
number2 = number3 - number1;
number1 = number1 + number2 + number3;
number3 = number2 * number1;
```

We use a table to keep track of each variable's value after executing each line of code:

Statement	Value after executing the statement		
	number1	**number2**	**number3**
`int number1 = 2;`	**2**	-	-
`int number2 = 3;`	2	**3**	-
`int number3 = number1 * number2;`	2	3	**6**
`number2 = number3 - number1;`	2	**4**	6
`number1 = number1 + number2 + number3;`	**12**	4	6
`number3 = number2 * number1;`	12	4	**48**

Note: '-' indicates that the variable has not yet been declared or initialized. At the end of the program, the value of `number1` is equal to 12, the value of `number2` is equal to 4, and the value of `number3` is equal to 48.

In the following exercises we will practice using variables and operators. Have a look at the following pieces of code. Indicate what the value of each variable is after executing the code:

- First, write down what you think the value of the variable will become after executing the code.
- Use Greenfoot to check your answer:

1. Write a **`void`** `practiceTracingCode()` method in which you can copy-paste the code (You can reuse the same method for all of the questions below).
2. In several places in the code, print the value of each of the variables. For example, you can print the value of a variable `nrOfEggsFound` in the console using:
   `System.out.println("Value of nrOfEggsFound is: "+ nrOfEggsFound);`

- If the answer is different than you expected, use the printed values to figure out how the code actually works.

You should be able to complete the following exercises without a problem. If you do have problems, then apparently you have not understood the topic well enough. Go back and make sure you understand this before moving on. This will prevent you from making unnecessary mistakes later on. Taking the time to understand this well now will be paid back (double)!

1. What does the value of **`int`** `nrOfEggsFound` become? Match the code on the left with the correct result on the right:

Code
`int` `nrOfEggsFound = 3;` `nrOfEggsFound ++;`
`int` `nrOfEggsFound = 2;` `nrOfEggsFound = nrOfEggsFound + 4;`
`int` `nrOfEggsFound = 1;` `nrOfEggsFound −−;`

Result of **`int nrOfEggsFound`**
0
2
3
4
6

2. (IN) What do the values of **`int`** `number1` and **`int`** `number2` become?

```
int number1 = 2;
int number2 = 4;

number1 = number1 + number2;
number2 = number1 + number2;
```

3. (IN) What do the values of **`int`** `number1` and **`int`** `number2` become?

```
int number1 = 3;
int number2 = 4;

number1 ++;
if( number1 != number2 ){
     number1++;
 } else {
     number2 ++;
 }
```

4. What do the values of **`int`** `number1` and **`int`** `number2` become?

```
int number1 = 2;
int number2 = 4;

number1 = number2;
number2 = number1 * number1;
```

5. (IN) What do the values of **`int`** `number1` and **`int`** `number2` become?

```
int number1 = 2;
int number2 = 4;

number1 = number2;
number2 = number1;
```

6. What do the values of **int** `number1` and **int** `number2` become?

```
int number1 = 6;
int number2 = 3;

number1 = number2;
number2 = number1;
if( number1 == number2 ){
    number1 = number1 + number2;
}
```

7. What do the values of **int** `number1` and **int** `number2` become?

```
int number1 = 6;
int number2 = 3;

number1 = number2;
number2 = number1 * number1;

if( number1 < number2 ){
    number1 = number1 + number2;
} else {
    number2 ++;
}
```

8. (IN) What does the value of **int** `number` become?

```
int number = 3;
while( number < 10 ) {
    number = number + 2;
}
```

9. (IN) What does the value of **int** `number3` number?

```
int number1 = 10;
int number2 = 8;
int number3 = 4;

number3 = doSomethingWithTwoNrs( number1, number2 );
```

where:

```
public int doSomethingWithTwoNrs ( int a, int b ) {
    int result = (a + b)/2;
    return result;
}
```

10. What do the values of **int** `number1` and **int** `number2` become? In your own words, describe what this code does with `number1` and `number2`. What is the role of **int** `numberTemp`?

```
int number1 = 6;
int number2 = 3;
int numberTemp = 0;

numberTemp = number1;
number1 = number2;
number2 = numberTemp;
```

5.1.2 Turn to the East

We are going to write a method which turns Mimi so that she faces East.

1. Right-click on `Dodo`'s method **`int`** `getDirection( )`. Which value (result) belongs to the direction in which she is currently facing.
2. Have a look at the value of the variable **`int`** `myDirection`. Do this by right-clicking on Mimi and then selecting 'Inspect'.

   Turn Mimi a few times and use 'Inspect' to complete the table:

Facing	Result of `int getDirection()`	Value of `int myDirection` using 'Inspect'
North		
East		
South		
West		

3. Determine an algorithm to turn Mimi until she is facing East.
4. Draw the corresponding flowchart.
5. Write the corresponding method **`void`** `faceEast( )`.

   Tip: Have a look at the class `Dodo` in the editor. At the top of the class, so-called **class constants** have been declared for the four directions:

   ```
   public static final int NORTH   = 0;
   public static final int EAST    = 1;
   public static final int SOUTH   = 2;
   public static final int WEST    = 3;
   ```

   Because of these class constants, you can use `Dodo.NORTH` instead of '0'. For example, you can use **`if`** `( getDirection( ) == Dodo.NORTH )` to check if Mimi is facing North. This makes the code easier to read and less prone to errors. What these class constants are, will be explained in assignment 6. But for now, you can use them in your method in this manner.

6. Add comments to your code.
7. Compile. Test your method by right-clicking on Mimi and then selecting method.

You have now learned how to compare a variable and a number.

5.1.3 Turn how many degrees?

Write a method **`int`** `degreesNeededToTurnToFaceNorth( )` which returns how many degrees Mimi must turn (counterclockwise) to face North. Can you do think of an algorithm which does this without using an **`if`** .. `then` .. **`else`** statement?

In this exercise you have done (arithmetic) calculations using variables.

5.1.4 Turn and step

We are now going to write a generic method which first turns Mimi so that she is facing a particular direction and then lets her take a step in that direction. The direction is given as a parameter. This is a generic method because it works for any (initial or final) direction.

1. Come up with an algorithm which makes Mimi turn so that she faces a given direction and then moves forward (in that direction). Tips:

   (a) What is the `type` of the parameter?
   (b) Your method will get the direction Mimi has to face as an argument. First, think about how you can determine whether or not Mimi is facing in the correct direction.
   (c) Try to come up with an algorithm which doesn't separately specify what needs to happen for each particular direction. Tips:
      - This can be done using a `while` loop.
      - Use what you came up with in the previous exercise 1b.

2. (IN) Draw the corresponding flowchart.

3. Translate the flowchart into code for the method `void turnToDirectionAndMove( int direction )`.
   - First write a separate sub-method `boolean facingCorrectDirection( int direction )` which determines whether or not Mimi is already facing in the correct direction.

4. Add comments to your code.

5. Compile and test you method by right-clicking on it.

5.1.5 Go to a location

We will now write a method `void goToLocation( int coordX, int coordY )` that sends Mimi to a location with the following coordinates (coordX, coordY).

1. Come up with a suitable algorithm. Tip: It may be useful to consider different cases individually, depending on which direction she needs to head.

2. (IN) Draw the corresponding flowchart. Tip: Use a separate sub-method (and thus flowchart) to check if Mimi has reached the desired location.

3. Write the corresponding code. Don't forget to add comments.

4. Compile. Test your method by right-clicking and filling in several different values. Does it work properly if you fill in (0,0)? And (2,3)? And (11,11)? And how about (14,14)?

5. Adjust your program so that it deals with invalid coordinates correctly:
   (a) Write a separate sub-method `boolean validCoordinates(int coordX, int coordY)` which checks if the given coordinates (the arguments) are valid (in other words: if they exist in Mimi's world).
   (b) If invalid coordinates are given, show an error message using:
   `showError( "Invalid coordinates");`
   (c) If the input is invalid, Mimi should stay in her place (do nothing);
   (d) Your program should work for any world size, not only a world with 12 by 12 cells. Ask the World what its (current) width and height are. Tip: To get the world's width, first call: `World world = getWorld( );` and then ask its width: `world.getWidth( )`.

6. Test your code modifications.

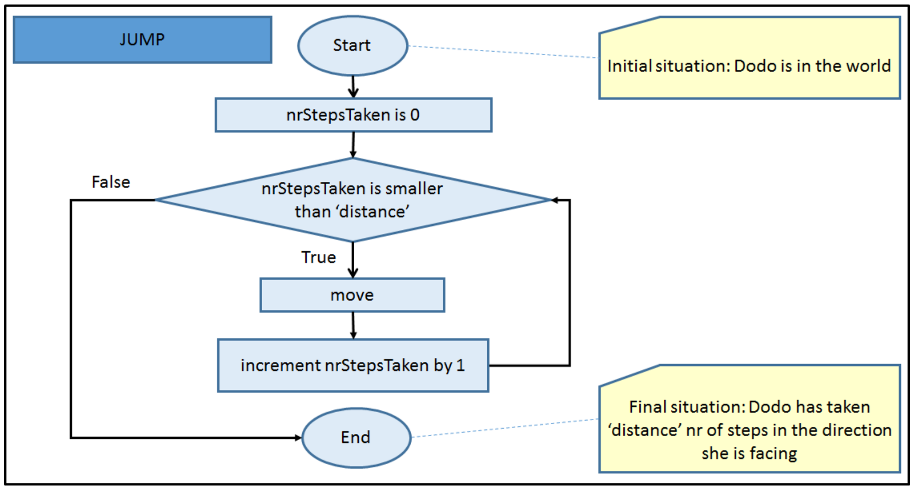
We have now seen how you can use and compare variables. We also saw how to use a combination of operators with different parameters, variables, and numbers to build more complex conditional expressions.

5.2 Count repetitions

Counter
A *counter* is a variable that is used to keep track of how often something happens. For example, a variable can be used to count how often the code in a **while** loop is executed. That way, you know when you're done.

As a concrete example, let's have a look at `MyDodo`'s code for **void** `jump (`**int** `distance)` which we came across in scenario 1. This method makes Mimi take steps repeatedly until the the required distance has been reached. A counter is used to keep track of how many steps Mimi has taken (which is also equal to the number of times the **while** loop has been executed). When the counter equals the required distance, Mimi is done.

Flowchart:


JUMP
Start
Initial situation: Dodo is in the world
nrStepsTaken is 0
False
nrStepsTaken is smaller than 'distance'
True
move
increment nrStepsTaken by 1
End
Final situation: Dodo has taken 'distance' nr of steps in the direction she is facing

The flowchart explained::

- First we need a counter 'nrStepsTaken' to store how many steps Mimi has taken.
  - While 'nrStepsTaken' is smaller than the required 'distance', then Mimi is not done yet and more steps need to be taken. Therefor, the conditional expression is 'True'. Mimi takes a step ('move') and the counter 'nrStepsTaken' is increased by one.

    After that, the method jumps back to the diamond and checks the conditional expression again. If the condition 'nrStepsTaken is smaller than distance' is still 'True' then the 'True' path is followed again (this is a loop). Otherwise the method ends.
  - If 'nrStepsTaken' is larger or equal to the required 'distance', then no more steps have to be taken and Mimi is done. The conditional expression is 'False'. The method ends.

Code:

```
public void jump (int distance){
    int nrStepsTaken = 0;            //counter 'nrStepsTaken' is initialized
    while( nrStepsTaken < distance){ // Not enough steps taken? Then continue
        move();                      // take a step
        nrStepsTaken++;              // increase counter by 1
    }
}
```

The code explained:

- A counter is declared and initialized with the value 0: `int nrStepsTaken = 0;`.
- First the conditional expression `nrStepsTaken < distance` is checked.
  - While the conditional expression is **`true`** (thus if `nrStepsTaken < distance == true`), the code between the curly brackets { and } is executed. In this case `move( )` is called, followed by incrementing the counter `nrStepsTaken++`.
    After that the method jumps back to the conditional expression `nrStepsTaken < distance`. If the conditional expression is still **`true`**, then the code between the curly brackets is executed again (loop). Otherwise the method ends.
  - If the conditional expression is **`false`** (thus if `nrStepsTaken < distance == false`), in other words `nrStepsTaken` is larger or equal to `distance`, then the method ends.

Note:
The number of times that the **`while`** loop still has to be executed depends on the value of **`int`** `nrStepsTaken`. The variable `nrStepsTaken` is incremented by one each time the code in the **`while`** is run. At a certain point, the value of `nrStepsTaken` will be equal to `distance`. In that case, Mimi is done. The **`while`** will not be executed anymore and the method ends.

A common mistake is to forget to increment the counter in the loop. Then the program will never come out of the loop and will repeat for ever.

5.2.1 Tracing the While

It's time to practice using the **`while`** loop. Just like in the first exercise (ex. 5.1.1), we're going to review code and trace how the variables change as the code runs. This is a good way too really understand how a **`while`** loop works. That way, you'll make less (hard to find) mistakes while coding. Have a look at the following pieces of code and try to figure out what happens to the variables. Use Greenfoot to check your answers. Use your **`void`** `practiceTracingCode()` method from ex. 5.1.1 to run the code.

1. What do the values of `nrOfStepsAlreadyTaken` and `nrOfStepsToTake` become after running the following code? How often does the method `move ();` get called?

```
int nrOfStepsToTake = 8;
int nrOfStepsAlreadyTaken = 0;

while( nrOfStepsAlreadyTaken < nrOfStepsToTake){
    move();
    nrOfStepsAlreadyTaken++;
}
```

2. After running the following code, what are the values of `counter` and `nrOfStepsToTake`? How often does the method `move ();` get called? Does the code work correctly?

```
int nrOfStepsToTake = 6;
int counter = 0;

while( counter <= nrOfStepsToTake){
    move();
    counter++;
}
```

3. (IN) After running the following code, what are the values of `counter` and `nrOfStepsToTake`? Explain why the code is correct this time (in comparison to the last question).

```
int nrOfStepsToTake = 4;
int counter = 0;

while( counter <= nrOfStepsToTake-1){
    move();
    counter ++;
}
```

4. Complete the following table for each time that the code in the **`while`** is executed.

```
int number1 = 2;
int number2 = 5;
int counter = 0;

while( number1 <= number2){
     move();
     number1 ++;
     counter ++;
 }
```

Nr of **`while`**-loops executed	Value of `number1` after **`while`**	Value of `number2` after **`while`**
0		
1		
2		

5. (IN) Modify the following **conditional expression** in the **`while`** so that `move( )` is called three times.

```
int number1 = 10;
int number2 = 8;

while( number1 > number2 ){
    move();
    number1--;
}
```

6. Have a look at the method `jump` as described in the theory block above.
   (a) Add (i.e. copy-paste) the method **`void`** `jump(`**`int`** `distance)` to the `MyDodo` class.
   (b) For each of the following, indicate how often the code in the **`while`** (of **`void`** `jump(`**`int`** `distance)`) is executed:
      i. if **`int`** `distance` is equal to 5,
      ii. if **`int`** `distance` is equal to 1?
      iii. if **`int`** `distance` is equal to 0?
      iv. if `nrStepsTaken` is equal to 3 and `distance` is equal to 5?

   Tip: You may want to print some values into the console.

We have now seen how to use a counter variable to determine how often code (a **`while`** loop) should be executed in order to get correct results.

5.2.2 Number of steps until the edge of the world

We're now actually going to practice with using a **`while`** and a counter, in addition to having a method return a value when it's done.

The counter belongs to the method. Recall from the theory block about a variable's life-span that the counter's scope is restricted to that of the method. When the method terminates, then the counter will no longer exist. By having the method return the counter's value, it can be used in other parts of your program.

In this exercise we write a method which returns how many steps Mimi has to take to get to the edge of the world. In order to do this, we let Mimi walk forward (until she reaches the end of the world) and count the number of steps she takes to get there. This value is returned by the method.

1. Come up with an algorithm for counting the number of steps which Mimi takes until the edge of the world has been reached. Make use of a **`while`** and a counter variable. Tip: Have a look at the method **`void`** `jump(` **`int`** `distance )` for inspiration.
2. Choose an appropriate variable name for the counter.
3. Draw the corresponding flowchart.
4. Write the corresponding method **`int`** `walkToWorldEdgeAndCountSteps( )` which counts how many steps Mimi takes and returns that result.
5. Add comments to your code.
6. Compile and test your method by right-clicking on Mimi and choosing the method. What is the smallest value that your method can return as a result? And the largest?

We have now used a counter variable to keep track of how often a **`while`** loop has been executed. We also wrote a method that returns the variable so that it can be used in other parts of the code.

5.2.3 Counting number of eggs is a row

Mimi will walk to the edge of the world and count how many eggs she finds in that row (or column).

Write a method that returns how many eggs there are in a row or a column.

1. Come up with an algorithm for counting the number of eggs between Mimi and the edge of the world. Use a **`while`**-loop. Tips:
   - In order to do this, let Mimi walk forwards (until the edge of the world is reached) and count the number of eggs she finds.
   - Have a look at the method **`void`** `walkToEdgeOfWorld( )` which you wrote in assignment 2 for inspiration.
2. Choose an appropriate variable name to store how many eggs have been found.
3. Draw the corresponding flowchart. Return the number of eggs found as a result.
4. Write the code for the corresponding method **`int`** `walkToWorldEdgeAndCountEggs( )`.
5. Don't forget to add comments.
6. Compile and test your method by right-clicking. Tip: Use `println` to print the variable. That way you can check whether or not your method works correctly.

We have now used a variable to store information which was gathered throughout the execution of a **`while`** loop.

5.2.4 Trail of eggs

We want to teach Mimi to leave a trail of (a certain number of) eggs. The initial situation is as follows:

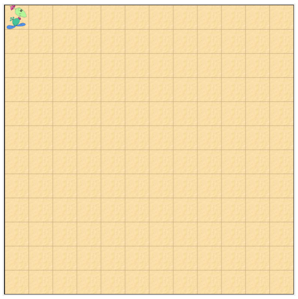


Figure 2: Initial situation

We write a method called **`void`** `layTrailOfEggs(` **`int`** `nrOfEggsToLay )` with which Mimi leaves a trail of `nrOfEggsToLay` eggs behind. Thus, `layTrailOfEggs(6)` will result in the following final situation:

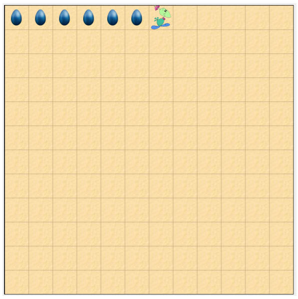


Figure 3: Final situation after Mimi has left a trail of 6 eggs

1. Come up with an algorithm for leaving a trail of a certain number of eggs. Tip: Consider using a **`while`** and counter variable in your solution.
2. Draw the corresponding flowchart.
3. Write the method **`void`** `layTrailOfEggs(` **`int`** `nrOfEggsToLay )` so that Mimi lays a trail of `nrOfEggsToLay` eggs.
4. Don't forget to include comments in your code.
5. Compile and (by right-clicking on the method) test your code.
6. (IN) If **`int`** `nrOfEggsToLay` is equal to 6, how often will the **`while`** be run?

We have now used a counter variable to determine how often a **`while`** loop is executed.

5.3 Eggs in the world

Now that you have written a few basic, but very important types of methods, it's time for a more challenging exercise. In the following exercise we will teach Mimi how to count and remember how many eggs she finds. Then, in the exercises that follow, she can use this to do some 'more intellectual' work.

5.3.1 Counting eggs in the world

We want Mimi to count the number of eggs in the world. To do this we want her to walk through the entire world. We want her to visit every cell, but no cell twice. The following picture shows a way for her to walk through the world systematically:

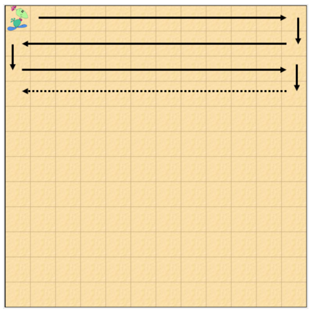


Figure 4: Mimi walks through the world systematically

As you can see, Mimi walks across each row, and the end of the row goes to the next row, and finally stops when she reaches the very last cell. To make programming and especially testing, easier, we divide the problem down into a few subproblems:

1. Walk to the starting point;
2. Walk to the end of the row while counting each egg found;
3. Determine if the final cell has been reached:
   (a) While the final cell has not yet been reached, go to the next row, turn, and repeat step 2;
   (b) If the final cell has been reached, then stop.

You can then combine the (sub)solution to each subproblem into one complete solution.

We will now deal with each subproblem, one-by-one, only focussing on that particular problem at a time. Always try to make use of methods you have already written (in previous exercises). 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! You don't even need to make a new flowchart for those methods, because even that you already did! You do, however, need to make one high-level flowchart in which you show how to solve the 'big' problem and which existing sub-methods you want to use.

1. (IN) Sketch a flowchart for the complete solution (using the steps 1, 2 and 3 above). Tip: Use a **`while`** loop.
2. Walk to the starting point:
   Try to make use of a method which you have already written in a previous exercise. Tip: Have a look at the method `goToLocation` from exercise 5.1.5. Determine if you can reuse this method.
3. Walk to the end of the row and count each egg found:
   In exercise 5.2.3 you wrote and tested the method **`int`** `walkToWorldEdgeAndCountEggs()`. Determine if you can reuse this method.
4. Determine if the final cell has been reached:
   Write a sub-method to determine if the final cell has been reached. You can do that as follows:

(a) The `World` methods **`int`** `getWidth( )` and **`int`** `getHeight( )` return how wide and high the world is. By right-clicking, call both methods. Write down the coordinates of the four corners of the world.

(b) In which corner will Mimi be when she is done? Is that always the same corner? If the world would be made one row larger, would it still be the same corner? If Mimi knows how high and wide the world is, how can she figure out if she is done (i.e. has reached the last cell)? Describe this using the terms 'height' and 'width' instead of actual numbers like '12'.

(c) Come up with an algorithm to determine the coordinates at whcih Mimi is done. This always has to work, no matter what the size of the world is.

(d) Now write (and test) the corresponding method to determines if Mimi has reached the final cell. Tips:

- Use the `World` methods **`int`** `getWidth( )` and **`int`** `getHeight( )`. Don't use the number '12', because if the world is made larger, then your program won't work anymore.
- Write a method **`boolean`** `isEven(`**`int`** `number)` that determines if a number is even or odd. You can use modulo '%' (remainder after division). A number for which 'number%2' is equal to 0 is even. If it is not equal to 0, then the number is odd.

5. Go to the next row:
Write and test a sub-method which makes Mimi step to the next row and turn so that she is facing the correct direction. Your method should work on both sides, whether she reaches the left-hand-side of the world or the right-hand-side of the world.

6. Combine your solution to each subproblem above into one complete solution:
Follow the next steps to write a method so that Mimi walks through the whole world and counts eggs:

(a) First, come up with a variable name for remembering how many eggs Mimi has found.

(b) (IN) Refine your flowchart for the complete solution (from step 1), making use of the sub-solutions you have just found. Include the variables. Also determine when the variable for the number of eggs must be incremented.

(c) Translate your flowchart into code by writing the method **`void`** `walkThroughWorldAndCountEggs( )`. Check if the initial and final situations of any reused methods match what you expect them to be.

(d) When Mimi is done, print the number of eggs that she has found in the console, using `println( )`.

(e) Don't forget to comment your code.

(f) Compile and test your method. Determine if it is a complete solution to the problem.

You have now found a solution to a complex problem by breaking it down into sub-problems (abstraction). Then you designed, implemented, and tested the sub-problems individually (modularization) or re-used existing methods. Then you combined the sub-solutions in order to solve the problem as a whole.

5.3.2 Find the row with the most eggs

We're going to teach Mimi how to find which row has the most eggs in it. Again, we're going to let her walk through the world systematically. When she's finished, a popup dialog box (console) shows which row number has the most eggs, and how many eggs there are in that particular row.

1. Come up with an appropriate algorithm. Tips: Make use of a **`while`**. For each row, let Mimi determine if that particular row contains more eggs than the rows she has already visited.

2. How many variables do you think you will need to store all the information that Mimi has to remember? Choose names and types for these variables. Tip: You probably need 4 **`int`** variables.
3. (IN) Draw a flowchart, showing your algorithm on a high-level, Tips:
   - Don't go into details.
   - Break the problem down into smaller problems.
   - Use methods you have already written in previous exercises.
   - For each subproblem, draw a sub-flowchart. (Each sub-flowchart will become a sub-method in the code, which you can write and test separately from the other methods.) Note: you don't have to draw flowcharts if you re-use an existing method!
   - Make your solution generic by asking the world its height using `getHeight( )`. Always try to avoid using concrete numbers like '12' in your code.
   - After Mimi has walked through all the rows, show a popup dialog box (console) with which row has the most eggs, and how many eggs there are in that row. After that, the program stops.
4. Write a separate sub-method for each subproblem.
5. One-by-one, compile and test each sub-method separately (by right-clicking on the method's name).
6. Write the main method `findRowWithMostEggs()`. In this method, call your sub-methods according to your high-level flowchart.
7. Add comments to the code.
8. Compile and test your `findRowWithMostEggs()` method (by right-clicking).
9. Call your method `findRowWithMostEggs()` in the `act( )`.
10. Compile, run and test your program.
11. (IN) 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?

Type casting
In some cases an object's type can be converted to another type. This is called *type casting*. Using brackets, the new type is written in front of the object.

When would you use this? For example, when dividing two numbers (**`int`**s), the answer may be a decimal (**`double`**) value. Without casting, dividing two **`int`**s automatically results in an **`int`**, not a **`double`**.

For example, the following code will result in '1' being printed:

```
int number1 = 5;
int number2 = 3;

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

Why? Because `number1` and `number2` are **`int`**s, and thus so is its answer. 5 is divided by 3 which is then rounded down to the nearest whole number (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 1.6666666666666667 in the console.

Example:
Here is an example of casting using MyDodo's code. If for example, we want to cast the value of **`int`** `nrEggsFound` to a **`double`**(a decimal value), then we write: `(double) nrEggsFound`).

5.3.3 Average number of eggs

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*.

1. Initialize a variable with the type **`double`** to store the average value.
2. Type cast the variable **`int`** `nrEggsFound` into a **`double`**.
3. Calculate the average number of eggs.
4. Show the result in the console.

6 Summary

You can now:

- use variables to store information;
- calculate using variables;
- use variables in conditional expressions;
- use a counter in a **`while`** loop;
- re-use your (own) methods in other parts of your code;
- break a complex problem down into sub-problems (abstraction);
- design, implement, and test sub-problems individually (modularization);
- call sub-methods from other methods (re-use).

7 Saving your work

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

8 Handing in

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

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

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