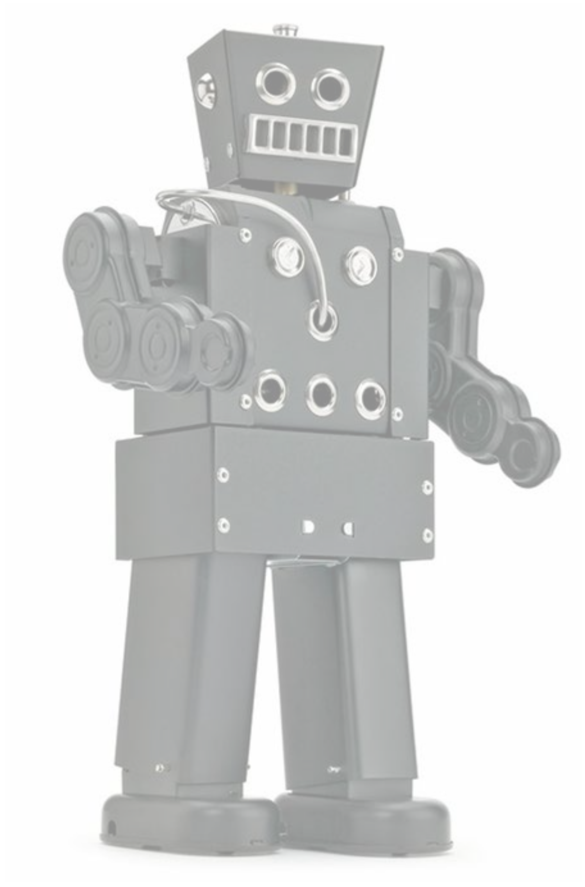
Getting started with Excel Visual Basic Programming



Contents

1 Introduction

Learning to program means overcoming various challenges; learning to program Visual Basic for Applications (VBA) in Excel even more, since there's not only the programming language to learn, but also the Excel object model. For many years, Stanford University has utilized a model where programming concepts are introduced through a model regarding a simple programmable robot named "Karel."

The "robot" approach has a couple of advantages. Mostly, it starts with a disconnect between the complexity of a full-blown language and the tasks that can be performed with it. Karel's world is simple, and his commands are simple, and there is a direct connection between those commands and the actions the result in.

Second of all, programming is mostly about *problem solving*. The student gets bogged down quickly in language issues and loses track of the reason to write programs in the first place; to solve certain problems. The world Karel lives in provides simple problems (with not always simple solutions!) and allows us to focus on solving a given problem. Dealing with language specific issues will come eventually, do not be afraid; but we have to learn to walk before we can run.

Finally, starting with Karel has a big advantage over dry programs that output meaningless lines of numbers and text; it's fun and purposeful. Most of those who code did not start with it because it's good business (it can be if you're *really* good at it), but because it's fun. And bossing Karel around is definitely fun!

1.1 Who is Karel?

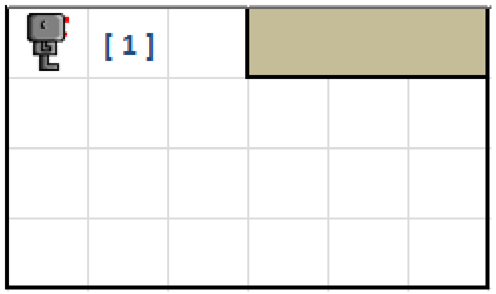
Karel is named to honor Karel Čapek, a Czech writer who coined the word "robot" in his 1920's play "*Rossum's Universal Robots.*" Computer scientist Robbert E. Pattis introduced an educational language called "Karel" in his 1981 book *Karel The Robot: A Gentle Introduction to the Art of Programming.*

Ever since, various implementations of *Karel the Robot* have been created; most notably the university of Stanford uses Karel for their Computer Science program with a Java implementation, but Karel has been developed for other languages as well; and now he's available in Excel, too!

Karel's world exists of a plain rectangle of finite size. Pattis' original implementation featured a world organized in *streets* and *avenues*, with the (1,1) origin at the bottom, but for Excel it makes more sense to work in *rows* and *columns*, and count rows *downward*, just as Excel does.

To make life more exciting, Karel's world can have *walls* (which you can draw yourself using the borders in Excel) and *beepers*. A beeper is described by Pattis as a "small cone emitting a soft beeping sound," but the reality is that beepers can represent *anything* depending on the scenario we're programming. In Excel, beepers are identified by putting a numeric value in a cell; the number indicates the number of beepers in that position.

A very simple *Karel* world. The robot is in position (1, 1)—row 1, column 1. There a single beeper at position (1, 2)
At the right top there's a walled-off area. Karel doesn't care about colored cells—in fact, he's blind—but he cannot move from one cell to another when there is a "wall" in between them; in Excel, the walls are simply created by drawing borders between cells.



1.1.1 Karel 1000

You will start with the "Karel 1000," the basic entry model. This is the simplest model from the factory, and quite frankly, it's very limited. In fact, you will likely not believe how simple it is! Karel 1000 can perform five commands, and that's it:

- Move – make one step in the direction Karel is facing. If Karel crashes into a wall when moving he will tell you so, remain in his current position and terminate your program.
- Turn Right – make a 90° turn. (Note that there's no left turn or "turn around")
- Pick up a beeper – Karel has a bag of nearly[*] unlimited capacity and can put any beeper in there he finds. However, tell Karel to pick up a beeper when there's no beeper in the cell he resides in will cause him to panic, and result in your program crashing and terminating.

[*] To be precisely, Karel's bag will fit 2,147,483,647 beepers.

- Put down a beeper – Similarly, Karel can drop beepers from his bag. One at a time, but each cell can hold a lot of beepers! Again, asking Karel to put down a beeper when his bag is empty will cause panic; your program will crash and terminate.
- Power off – nobody wants a robot on the loose! We all remember the 2008 Massacre when the PowerWeld 9000 escaped its Ford factory in Detroit and went on a rampage, killing 12 customers in a Mercedes showroom. Although Karel is rather limited and powerless, it's best to take no risks. Don't forget to power off Karel at the end of your program—your life might be at risk otherwise!

1.1.2 Karel 2000

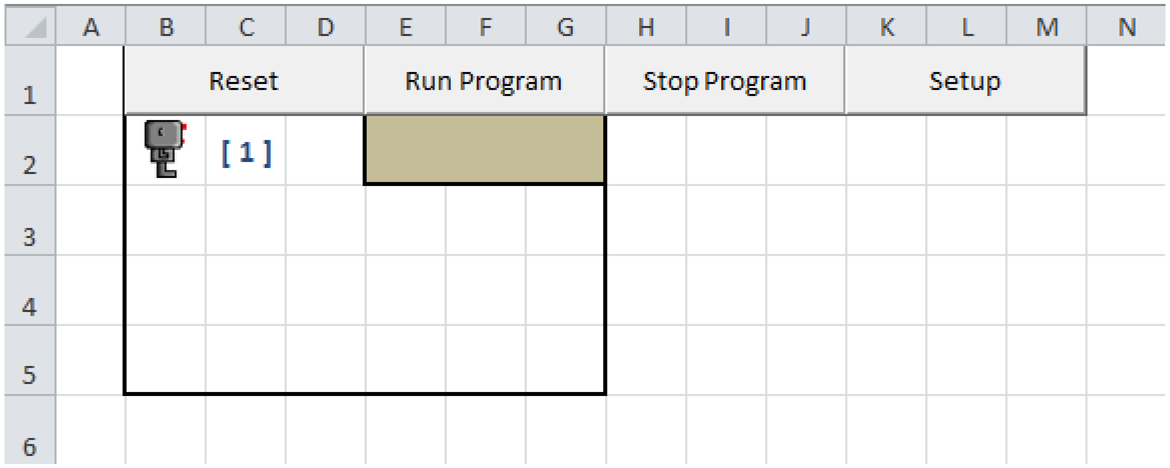
It will not take long before realizing how limited the economy model of the Karel line really is. Some limitations like having only a right–turn available can easily be dealt with, but other shortcomings seriously limit what Karel can do. For that reason, we will eventually upgrade to a more advanced model, the "Karel 2000." On top of the Karel 1000 functionality, this model offers:

- Left Turn and Turn Around.
- Move Fast (multiple steps at a time)
- Tell if he's in front of a wall or not
- Tell if there's a beeper in the current cell (yes|no – not the actual number of beepers)
- Tell if there are beepers in the bag (yes|no – not the actual number of beepers)

There's still a lot to be desired—obviously, the Karel marketing team wants to keep open the option to upsell you a future "Karel 3000," but the "SuperKarel" robot allows for plenty of interesting problems to be tackled.

1.2 Programming Karel

To start with Karel, open the `karel the robot vs.xlsm` file in Excel. The "m" in the `xlsm` extension indicates macro's, and you might get a warning that this workbook contains macro's; activate them to continue.



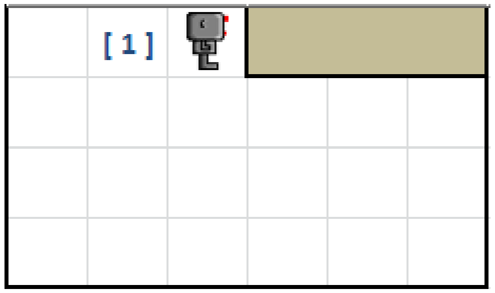
Karel's little world, fresh from the factory.

There are a couple of buttons available, and it's good to know what they do:

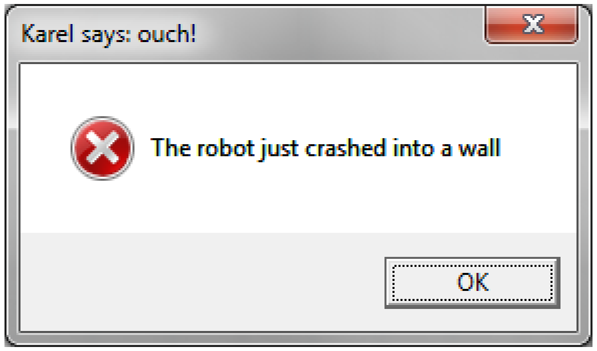
- RESET—this will reset Karel to the (1,1) position, with an empty bag, facing east.
- RUN PROGRAM—this will run whatever program is defined for Karel
- STOP PROGRAM—this will immediately power off Karel, regardless of what program stepps he's executing. Consider this the big red "stop" button in case your program goes wrong (Karel has escape an is on his way to column XFD, or keeps running around in circles, etc)
- SETUP—here you can set Karel's starting position, direction, number of beepers in his bag. You can also load a predefined "assignment" for various exercises; the picture above ("factory condition") shows the first assignment.

1.2.1 Time to get to work!

- Click on the RUN PROGRAM button. Karel should take two steps and power off.



Click on the RUN PROGRAM button again. Karel should take *another* two steps. But he's no longer where he was from the factory settings, and will crash into a wall instead! Don't do this too often; it will void the warranty (Karel comes with a built-in crash counter that factory mechanics can check).


Karel says: ouch!

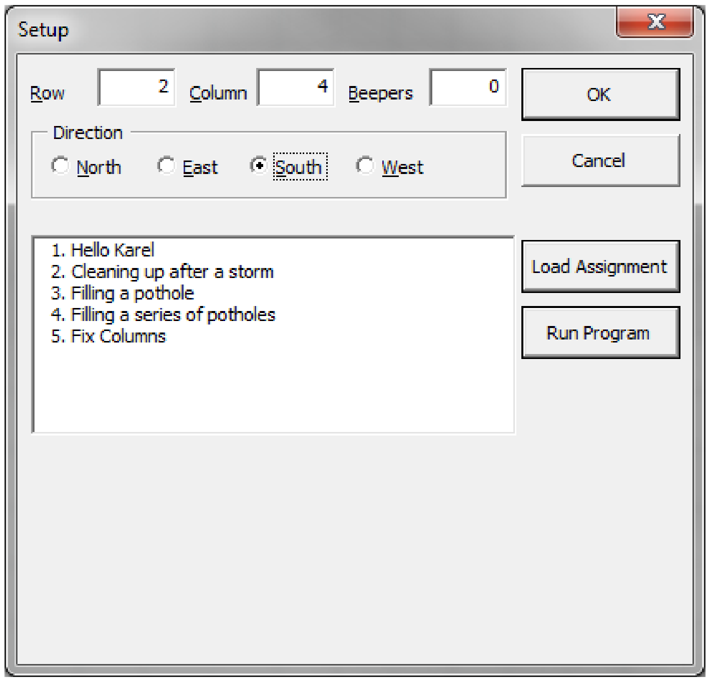
The robot just crashed into a wall

To run the program again from the starting position you will have to reset Karel to his initial position.

- Click on the RESET button.
- Click on the RUN PROGRAM button.

Another way of setting the start position of Karel is by clicking the Setup button.

- Click the SETUP button.
- Specify row 2, column 4, and have Karel facing south.

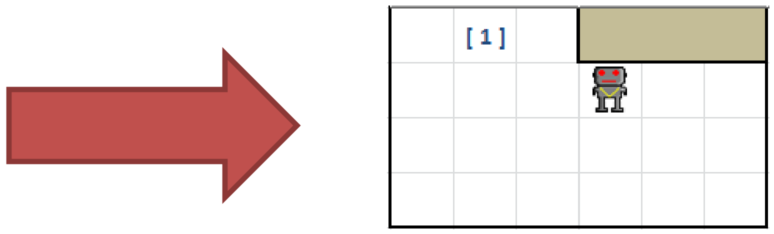


Setup

Row 2 Column 4 Beepers 0

Direction: ( ) North ( ) East [✓ South] ( ) West

1. Hello Karel
2. Cleaning up after a storm
3. Filling a pothole
4. Filling a series of potholes
5. Fix Columns



- Run the program again.

1.2.2 Working with Assignments

You can also load assignments through the SETUP window.

- Open the setup window by clicking on the SETUP button.
- Click on assignment 3, "FILLING A POTHOLE."
- Click LOAD ASSIGNMENT.

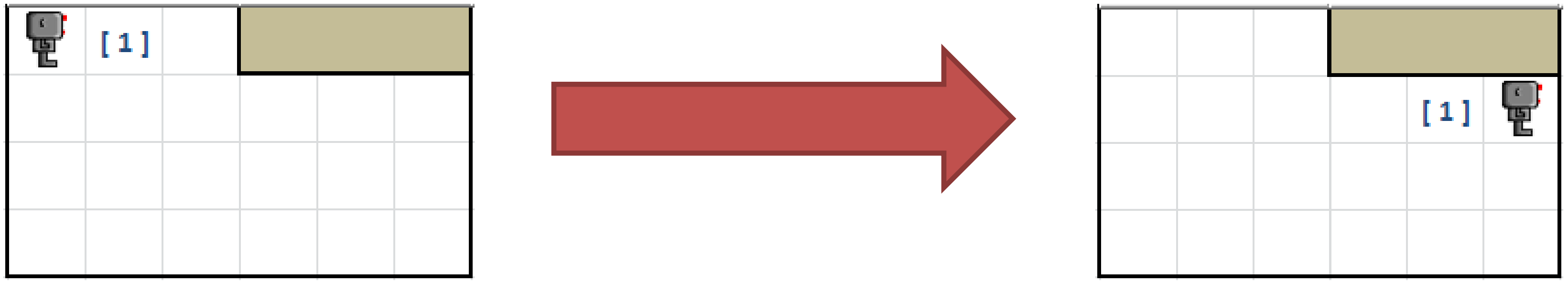
This will return you to Excel, where you can review the "world" and decide what to do with it; from here you can write your program and test it with RUN PROGRAM.

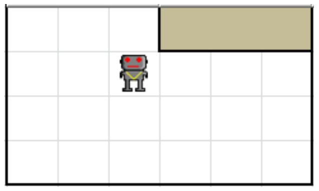
Once you think your program works well you can also test it by returning to the setup window, choose the assignment and click RUN PROGRAM. This will reset Karel to the correct starting position for the assignment and immediately start his program.

Note that in later phases it is very possible to get *multiple* environments with the same assignment; the environment you've made your program for might not be identical to the one you started out with. This is normal in programming; things change! Whenever you see "make sure your program works with any number of beepers," or something along those lines, be aware that this might happen.

1.3 Writing your first program

Now that you're familiar with Karel and his little world it's time to get to action. Load assignment #1 again, so that your Excel screen looks like the one below. The goal of this section is to end up with Karel moving the beeper to the middle of the grassy area in his garden, as displayed here:





Conceptually, the program needs to do the following:

- Move forward
- Pick up beeper
- Move forward again (you're now in front of the green area)
- Make right turn
- Move forward

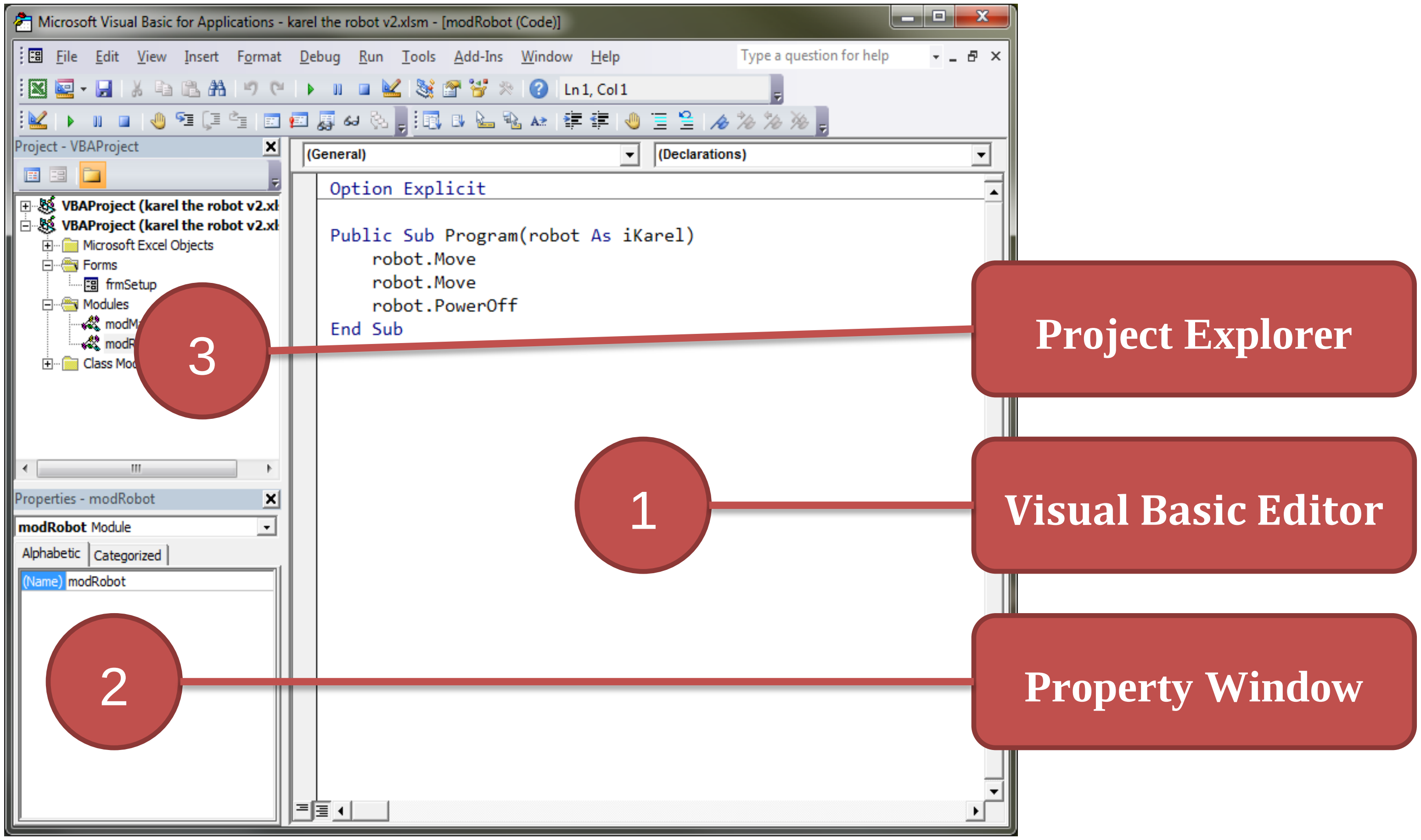
At this point we run into the problem that Karel can only make right turns, which wasn't a problem when turning south, but now he has to turn east again. Given that the Karel 1000 cannot turn left, the only solution is to turn right three times:

- Make right turn, and again, and again
- Move forward
- Move forward
- Put down beeper

- Move forward

And that would leave Karel at the other side of his backyard, with his beeper moved to the middle of the grassy area. How to actually *program* this?
For that, we need to enter the VBE, the *Visual Basic Editor* inside Excel, and that is done with keystroke ALT + F11.

- Press ALT+F11 to get into the Visual Basic Editor



```
Option Explicit

Public Sub Program(robot As iKarel)
    robot.Move
    robot.Move
    robot.PowerOff
End Sub
```

The Visual Basic environment usually shows multiple windows that can be docked; the most important ones are the PROJECT EXPLORER, the actual VISUAL BASIC EDITOR window and the PROPERTY WINDOW. You can use the View menu to enable or disable windows as desired. For now, the Properties window isn't really needed, but it doesn't hurt to have it around.

Code is organized in modules, and the PROJECT EXPLORER offers a convenient view of all the modules associated with a file ("Project"). The module that is open in the editor *should* be a module called `modRobot` (as the title bar indicates). If not, or just to be on the safe side, double-click on the `modRobot` icon under "MODULES" in the PROJECT EXPLORER. Whenever you encounter the icon TRY IT OUT you're encouraged to enter the code into your own robot project. You cannot train for a marathon by laying on a couch and watching marathons onTV; in a similar way you cannot learn to program merely by looking at code. Trying it out, experimenting with, coming up with better solutions, is the only way to learn and gain experience. Experiment! Try it out!

1.3.1 How code is organized

Visual Basic doesn't allow code to run unchecked inside a module; it needs to be enclosed inside a `procedure`. In this case, the procedure our code lives in a conveniently named procedure "Program":

```
Public Sub Program(robot As iKarel)
    robot.Move
    robot.Move
    robot.PowerOff
End Sub
```

The very first line defines the procedure. Procedures that return a value are called FUNCTIONS; more about those later. If we just need our procedure to "do" something without returning a value we use a "SUB" procedure; more about that later as well. PUBLIC means that it can be accessed from outside the module; more about that later as well.

Usually, a procedure needs to be fed with *arguments* to work. In this case, our sub procedure `Program` uses an argument `robot`. The overall code that creates our virtual Karl delivers him to us through this argument; inside our program we now have a variable `robot` that we can use to do things with. As `iKarel` defines the type of the variable; to ensure that `Program` works with a KAREL type variable, and not a number or text.

Visual Basic is an *object oriented* language. Objects are like little mini programs with their own functionality and the existing program shows how that works. If we want to call the `Move` procedure of an `iKarel`–type object we call `.Move`; if we want it to make a turn we call `.TurnRight`, etc.

- Remove the existing three lines and replace them with the following code:

```
Public Sub Program(robot As iKarel)
    ' Pick up the beeper and walk to the grass
    robot.Move
    robot.PickBeeper
    robot.Move
    ' Negotiate the corner
    robot.TurnRight
    robot.Move
    robot.TurnRight
    robot.TurnRight
    robot.TurnRight
    ' Drop the beeper at the right spot and
    ' walk to the end
    robot.Move
    robot.Move
    robot.PutBeeper
    robot.Move
    ' Don't forget to turn off the robot
    robot.PowerOff
End Sub
```

The lines that start with a single quote are comments and are intended to document the code. It's very good practice to put reasonable comments inside the code, to help you understand what it does if you ever need to fix bugs are have it adjusted to changed situations ("maintenance").

Comments should be short, but not cryptic. Don't comment every line and don't describe what the code factually does; instead document what the conceptual meaning of the code is.

In the following example, the code on the left has comments that are short and indicate what the *intention* of the code is. Maybe you made a mistake when writing the code, and the code doesn't actually do what it's intended to do; you will find out quickly when the descriptive comments don't match up with the code.

The code on the right, on the other hand, has useless comments. Each line is commented, making it a lot harder to read the code in the first place. Worse, the comments don't add anything to what we already know from reading the code. Yes, variable I loops. But why? We don't know what the *intention* of the code is, so we can only guess if this code is performing its job properly.

Most coders are the ones who maintain their code, so who cares? You do! When you're trying to figure out why your program is not doing what it's supposed to be doing six months after you wrote it you'll be wondering why you wrote that particular code in the first place. Therefore, "write your code as if it is maintained by a violent psychopath who knows where you live."

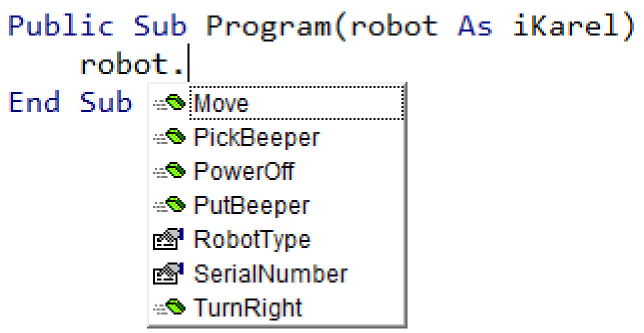
Comment like this

```
'   Loop through the list of cells to find the maximum,
'   and show the maximum value afterwards
max = 0
for i = 1 to 10
    if cells(i) > max then max = cells(i)
next
debug.print max
```

Not like this

```
'    Set max to zero
max = 0
'   Run a loop for i ranging from 1 to 10
for i = 1 to 10
    '   check if value in cell(i) larger than max
    if cells(i) > max then
        '   make max equal to cell(i)
    end if
'   next i
next
'   print max
debug.print max
```

Finally you will notice that Visual Basic will give you help when you're typing your code. This is called *autocomplete*. The autocomplete function will go through the list of available object methods as you type them.



```
Public Sub Program(robot As iKarel)
    robot.
End Sub
```

For instance, when you start typing ".P" it will jump to the first "P" in the list, PickBeeper. If you then continue with a "U", it will jump to PutBeeper. At that point you can hit the TAB key and have autocomplete finish the word for you.

You can also invoke autocomplete by CTRL+SPACE, for instance when typing variable names. For our current program it doesn't matter that much, but when programs get more complex and you have a long list of variables (with long names) to choose from it makes life a lot easier!

1.3.2 Running your code

- Once you've entered your program you can return to Excel (Alt+F11 again) and click the RUN PROGRAM button.

If you did everything correct you will be rewarded with seeing how little Karel picks up his beeper, goes around the corner, drops it off in front of the grass, and continues to the end of the yard. Congratulations! You just wrote your first program!

1.3.3 Coding Style

Visual Basic applies some basic formatting to code, but leaves indentation to the programmer. For instance, these two programs are, as far as VBA is concerned, identical:

```
Public Sub Program(robot As iKarel)
    ' Pick up the beeper
    robot.Move
    robot.PickBeeper
    robot.Move
    ' Don't forget to turn off the robot
    robot.PowerOff
End Sub
```

```
Public Sub Program(robot As iKarel)
' Pick up the beeper
robot.Move
robot.PickBeeper
robot.Move
' Don't forget to turn off the robot
robot.PowerOff
End Sub
```

The code on the left is indented; it's easy to see where the procedure starts and end. The code on the right is not. For a short program like this one it doesn't matter, but once code gets more complicated it's a horrible idea to omit indentation.

As a rule, any code between two lines that identify “begin” and “end” (as with Sub/End Sub) should be indented. For assignments, code that runs correctly but is not indented will result in a failed assignment. The code works; but it is not maintainable.

REMEMBER TO CODE AS IF A VIOLENT PSYCHOPATH WHO KNOWS WHERE YOU LIVE WILL MAINTAIN IT

2 Bossing Karel around

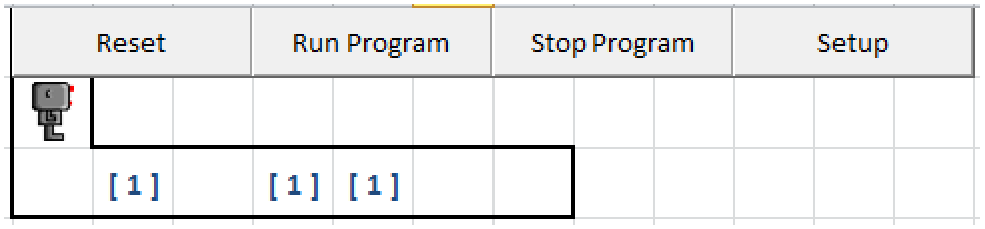
In this chapter you will practice with writing simple programs for Karel. The goal is to gain practice in writing code, as well as learning to make the first steps in *decomposition*. Let's start with a couple of exercises to warm up.

2.1 Cleaning up the back yard

After a big storm a lot of debris has collected in Karel's back yard. To clean up, he will simply have to walk to the locations where a beeper is located and pick it up (this indicates cleaning the designated spot).

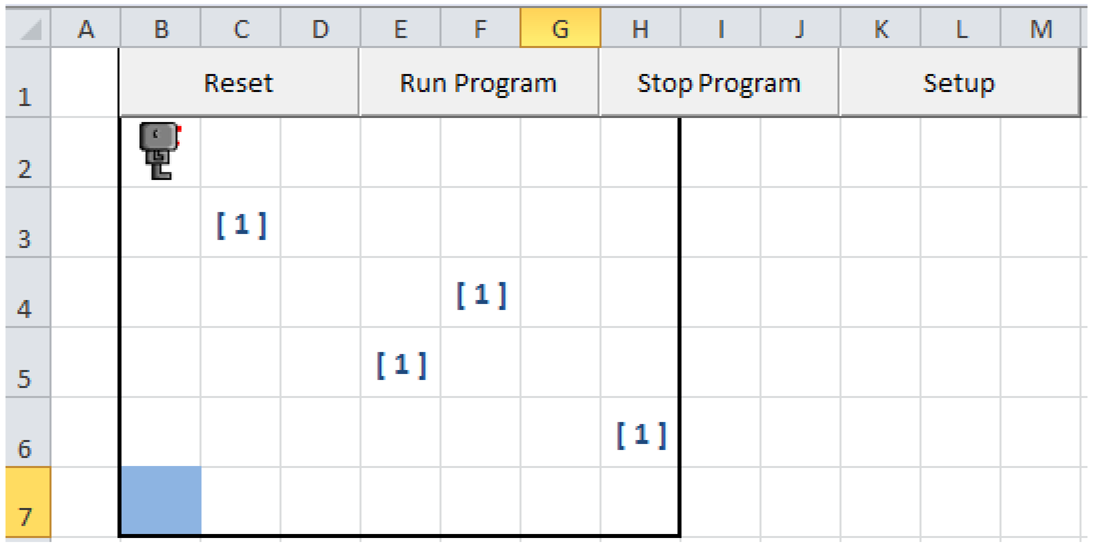
2.1.1 Cleaning up a strip in the yard

ASSIGNMENT #2: Debris has cluttered up Karel's back yard after a storm. The debris is symbolized by beepers; you task is to have Karel walk through the yard, pick up the garbage, and stop at the end of the yard.



2.1.2 Cleaning up the entire yard

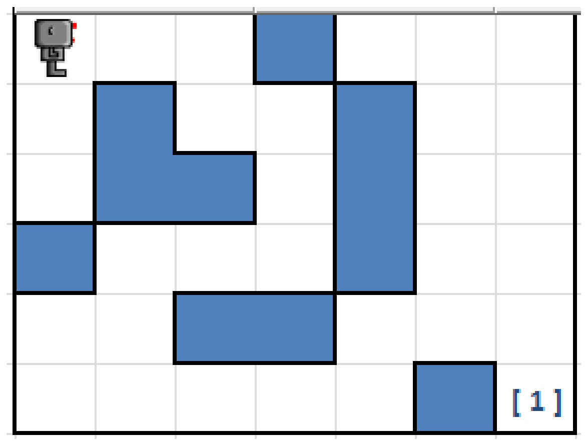
Assignment #3: Karel's parents are very proud of his work, and rewarded him with more work (robots *love* to work, so more work is truly a reward, not a punishment). This time, Karel has to clean up not just a small strip, but the entire back yard:



This problem is more interesting than it seems at first, due to Karel's inability to make left turns. How many lines of code (including begin/end sub) does it take for Karel to pick up all the debris? Taking different routes might make a big difference in how many operations Karel has to perform, when you can reduce the number of left turns taken.

2.1.3 The watery garden

ASSIGNMENT #4: Karel left his toy at the other side of the back yard. There are a lot of water puddles in the back yard because of the bad weather, and Karel cannot enter those as his circuits would short out. Your task is to write a program to navigate Karel to his toy, pick it up, and bring it back.



2.2 Decomposition

When you study the Karel code you've written so far, you will encounter a lot of lines where Karel has to turn around (twice a `TurnRight` operation) or turn left (three times a `TurnRight` operation). Karel doesn't mind taking those steps; the little bugger *loves* it. But as a programmer, writing those repetitive lines of code is not only boring but also dangerous; it increases the chances of errors.

Whenever code repeats itself it is a good practice to *decompose* a procedure into smaller ones. This process can be repeated until a level of desired simplicity is reached.

2.2.1 A real world example of decomposition

Image you have to make a business trip by car, from west coast to east coast, stopping at a couple of cities. On top of it, a Karel robot is the driver (he'll have an interface hooked up to the car. And this model *can* make left turns). Would you *start* writing your instructions with "make a left when you leave the parking lot," or would you start with a highlevel approach:

- Across the country: Los Angeles to New York, stopping at three cities

And from there cut it up into smaller pieces:

- Los Angeles to Denver
- Denver to Dallas
- Dallas to Chicago

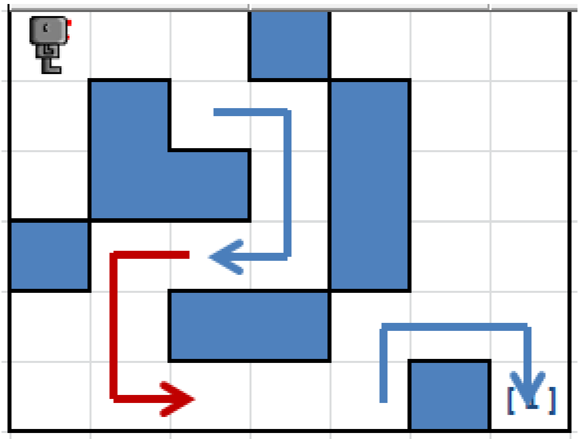
- Chicago to New York

And from there work on each of the legs, probably on an interstate-exit to interstate-exit level, before getting down to street level.

The same applies to programming. For any problem it's best to start at a fairly high level and have an overall game plan ready. Usually this will reveal what you're going to need in various steps. Then, were possible, decompose those high level steps in smaller steps.

2.2.2 Decomposing the Watery Garden

Let's take another look at Karel's route to get his toy back through the Watery Garden (ASSIGNMENT #4). Looking carefully at the map we can see that Karel has to make multiple "U-turns," either left or right-handed:



Then there's the dreaded left turn, and it would be *really* convenient if Karel could make a "double move". How would our program look like if all that were possible?

```
DoubleMove
TurnRight
Move
TurnLeft
UTurnRight  ' (following the "blue path")
UTurnLeft   ' (following the "red path")
DoubleMove
TurnLeft
UturnRight
```

```
10    PickBeeper
```

Compare that to the code written in the original assignment! The trick with decomposition is to find decompositions that are as generic as possible, so they can be reused many times. Sure, we could incorporate a decomposition “two steps forward, a left turn, one step forward, a right turn, another step forward” but that doesn’t sound a particular common maneuver. “Turn right,” “Turn Around,” “Double Move” on the other hand do, and for this particular problem it would be foolish to not make a fairly generic “U-Turn” maneuver set, especially when you need to write code to get Karel home as well.

2.2.3 Creating a new sub procedure

Let’s start at the very top of what we want to achieve: have Karel pick up his toy, and return to the (1,1) position. We could write a `RetrieveToy` procedure as follows:

Try it out!
VBA Code

```
1     Public Sub Program(robot As iKarel)
2         RetrieveToy robot
3         robot.PowerOff
4     End Sub
5
6
7     Sub RetrieveToy(robot As iKarel)
8         ' code goes here now
9     End Sub
```

Notice that `RetrieveToy`, just like the original `Program` procedure, is going to need an `iKarel` type argument–that’s what we manipulate to make Karel move around after all. There’s no *need* to call the argument in the `RetrieveToy` procedure robot; we might as well call it `SpongeBobSquarePants`, or `Bob`. But it makes sense to choose argument names that indicate what we’re passing around, and `robot` seems a good choice (that’s what Program uses, after all).

It is also important to notice that while the definition of RetrieveToy uses parenthesis (always!), the call to it in line 2 does not. This is one of those little peculiarities in Visual Basic (and different from practically any other language): sub procedure calls do not use parenthesis.

But why would you move your RetrieveToy code to a different procedure? What’s the point of that?

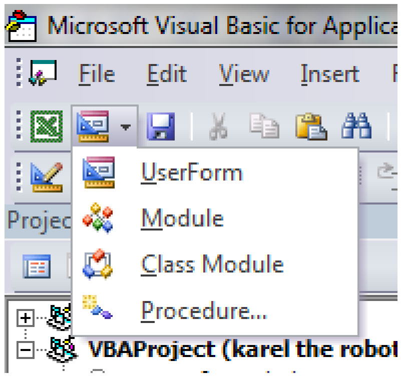
That is a good questions, and there are two good reasons to do it:

- Up to now, any assignment meant throwing out your old code; or at the very least storing it somewhere else (inside a word document, for instance). By putting the code for an assignment in a specific procedure for that assignment you can effectively keep your old code, and return to it, simply by swapping out the code in line 2 of the `Program` procedure.
- Did you see how we kept `robot.PowerOff` in the Program procedure? Surely you forgot to include that line a couple of lines! The best way of dealing with "forgetting to always write a certain line of code" is by *no longer needing to write it*. The reality is that *Good Programming Practice* is in most cases a euphemism for *mitigating your own stupidity.* And I'm comfortable in stating that I've picked up a lot of good programming practices over the years!

2.2.4 Organizing your code

While we're reorganizing our code into a new procedure, let's take the process a step further *and create a new module* for the code. Module are like sheets of paper that hold the code; there can be many modules in a project. Why? Pretty much for the same reason you'd be using multiple worksheets in Excel: to organize your code.

For instance, we can use the current module, modRobot, as the entry point for our code, and as the location for generic *helper functions*, like `TurnRight` and `TurnAround`. Code that is more specific to the assignments in this chapter can be stored in a module called `modChapter2`, and code for chapter 3 can be stored in `modChapter3`, etc. It is good practice to give module names a `mod` prefix; this will prevent name clashes later on.



There are various ways to insert a new module inside a VBA project. The INSERT, MODULE menu is probably the most obvious choice. But you can also use the insert button on the toolbar (second button in the toolbar). And you can right-click on the MODULES folder icon in the project explorer, and choose INSERT, MODULE in the context menu that subsequently appears.

Whatever the method is, once the new module is created it will appear.

- Change the name by clicking on the NAME tag in the PROPERTIES window, and type modChapter2 followed by ENTER.
- Double-click on MODROBOT in the PROJECT EXPLORER to return to the `modRobot` module.
- Highlight the SUB/END SUB lines of the `RetrieveToy` procedure, and cut (CTRL+X) them onto the clipboard
- Double-click on MODCHAPTER2 in the PROJECT EXPLORER to return to the `modChapter2` module.
- Paste (CTRL+V) the procedure onto the module.

Now we're ready to code!

2.2.5 Decomposing RetrieveToy

The problem is clearly defined as “getting Karel to his toy *and back*.” It doesn’t take rocket science to see that this divides our program into two sub problems:

- `WalkToToy`
- `ReturnHomeFromToy`

Whenever such a clear case arises, decomposition shouldn’t even be a *consideration*; it should be the *only* choice. Thus, our `RetrieveToy` procedure becomes:

Try it out! VBA Code

```
Sub RetrieveToy(robot As iKarel)
    walk_to_toy robot
    return_home_from_toy robot
End Sub

Private Sub walk_to_toy(robot As iKarel)

End Sub

Private Sub return_home_from_toy(robot As iKarel)

End Sub
```

The procedures `walk_to_toy` and `return_from_toy` are marked `private`, which means they can *only* be called from within the CHAPTER2 module. To make it easier to see what procedures are marked `Public` and what procedures are marked `Private`, a convention is used. Public procedures have names with `CapitalsButNoSpaces`; Private procedures have names with `no_caps_and_all_underscore`. Again, this a *convention*; not something enforced by Visual Basic; but it makes it easier in the long run to keep track of everything.

So what about RetrieveToy? Is that procedure Public or Private? Visual Basic will consider any procedure on a module Public unless explicitly marked Private. But it’s better to be explicit than to be implicit about those things, so the first line of code should be changed into **`Public`** `Sub RetrieveToy(robot as iKarel)`—another good practice.

2.2.6 Implementing walk_to_toy

Simply by saying “we split RetrieveToy into two sub procedures” merely delegates the solution to those procedures, it doesn’t solve the problem; we still have to *write* the code ourselves! So let’s take a stab at solving walk_to_toy, happily delegating further problems to yet more sub programs:

Try it out! VBA Code

```
Private Sub walk_to_toy(robot As iKarel)
    DoubleMove robot
    TurnRight robot
    robot.Move
    TurnLeft robot
    u_turn_right robot
    u_turn_left robot
    DoubleMove robot
    TurnLeft robot
    u_turn_right robot
    robot.PickBeeper
    TurnAround robot
End Sub
```

Some of the new sub procedures are extremely generic, as DoubleMove and TurnLeft. Those are put as Public procedures on the modRobot module, as we expect to use them many times in other procedures as well.

On the other hand, the u_turn procedures are *extremely* specific for this particular navigation problem. Instead of littering the modRobot module with non-generic code, it is better to define these as private procedures on the modChapter2 module.

The attentive student will have noticed that there is a call to a TurnRight procedure as well. What’s the point, isn’t TurnRight a method of the KAREL 1000 model? The answer is: why, of course! But… *it is a known problem with KAREL 1000 robots that their wiring gets switched during maintenance overhauls; now they only have a TURNLEFT method instead of a TURNRIGHT method.*

Ok, this may be a bit silly in this particular course, but it illustrates a problem that often occurs in reality: Things Change. Imagine a CSV text file, where the delimiter all of a sudden switches from a comma to a tab or semi-colon. One way to mitigate such problems is *layering*—we add a layer between our program and the external data. When the external data (or robot) changes, all we need to do is adapt our interface layer; a much easier solution than making changes all over your program.

Solid Programming Practice: anticipate changes, and develop an architecture that allows dealing with that change in only one particular location

We now have a list of helper functions to implement:

- `TurnRight`, `TurnLeft` and `TurnAround` on `modRobot`;
- `DoubleMove` on `modRobot`;
- `u_turn_left` and `u_turn_right` on `modChapter2.`

That begs the question: where to start?

2.2.7 Implementing the u-turns

Our best bet is to start with the high–level code first. Why? Because it keeps to flow of code going! It is not very likely, but writing the `u_turn` procedures might turn up the need for *even more* generic helper functions. Instead of finishing the code on `modRobot`, only to discover that we need to return there because `u_turn_left` uncovered the need for `TripleMove` or something like that would not be productive. Best to finalize the more complex functions first; they will tell us what else we need (or not). Remember to enter this code on `modChapter2`:

Try it out! VBA Code

```
' ------------------------------------------------------------
' Let the robot make a full u-turn on the outside of a 2x3 matrix
' This procedure is used by:
' - walk_to_toy
' - return_home_from_toy
' ------------------------------------------------------------
Private Sub u_turn_right(robot As iKarel)
    robot.Move
    TurnRight robot
    DoubleMove robot
    TurnRight robot
    robot.Move
End Sub

' ------------------------------------------------------------
' Let the robot make a full u-turn on the outside of a 2x3 matrix
' This procedure is used by:
' - walk_to_toy
' - return_home_from_toy
' ------------------------------------------------------------
```

```
Private Sub u_turn_left(robot As iKarel)
    robot.Move
    TurnLeft robot
    DoubleMove robot
    TurnLeft robot
    robot.Move
End Sub
```

As it turns out we're lucky—no extra generic move procedures are needed as a result of the u_turn procedures. But the only way to find that out was by writing them!

2.2.8 Implementing the generic move procedures

We're now getting down to the lowest level of decomposing our code; the humble generic move routines. It is likely that these are going to be used many, many times over, which is why they belong on the `modRobot` module as public procedures (as opposed to the `u_turn` procedures). Let's start with the easiest of them all, `TurnRight`:

Try it out! VBA Code

```
Public Sub TurnRight(robot As iKarel)
    robot.TurnRight
End Sub
```

Again, this procedure may seem like a waste, but layering the code like this prevents a ton of maintenance work if our Karel happens to get rewired the wrong way around (and now only makes *left* turns instead of *right* turns) during an overhaul.

Next up are the other two turn procedures, hardly any more complex:

Try it out! VBA Code

```
Public Sub TurnAround(robot As iKarel)
    robot.TurnRight
    robot.TurnRight
End Sub

Public Sub TurnLeft(robot As iKarel)
    robot.TurnRight
    robot.TurnRight
    robot.TurnRight
End Sub
```

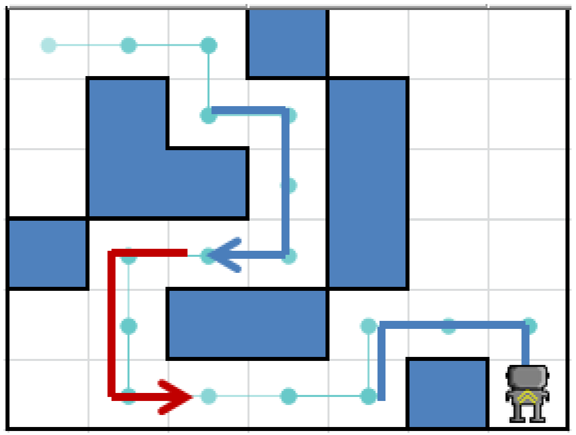
And finally, the DoubleMove needs to be implemented (and is hardly the stuff to worry over at night as well):

Try it out! VBA Code

```
Public Sub DoubleMove(robot As iKarel)
    robot.Move
    robot.Move
End Sub
```

2.2.9 Time to Test!

Now that all the helper functions are in place it's a good moment to test our code. The `return_home_from_toy` code is still empty, but the `walk_to_toy` code will reveal quickly any bugs or typos in our code, if there are any.



Success! The `return_home_from_toy` code still needs implementing, but that's a trivial exercise left to the student.

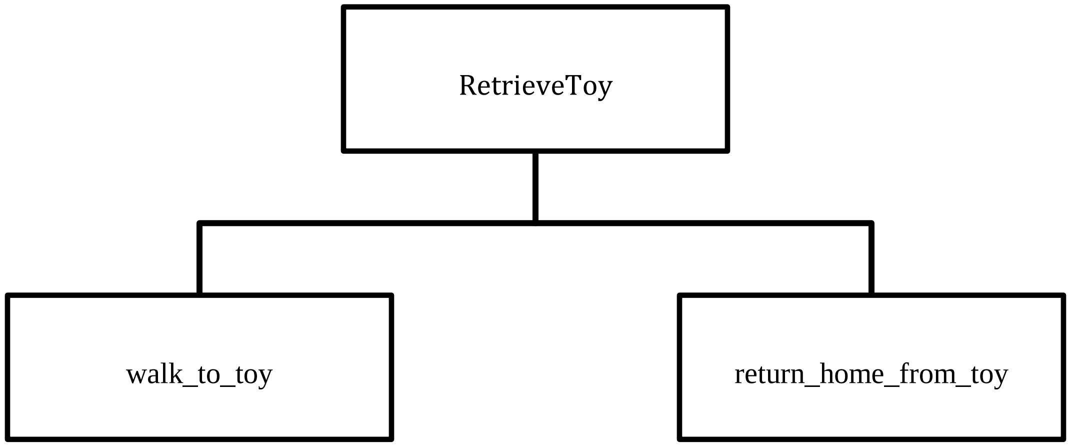
Good practice: test often. If possible, test after each single change of your code base; don't implement multiple changes at once, test after each change. Do not add features until the current bugs are fixed.

2.3 A systematic approach to decomposition

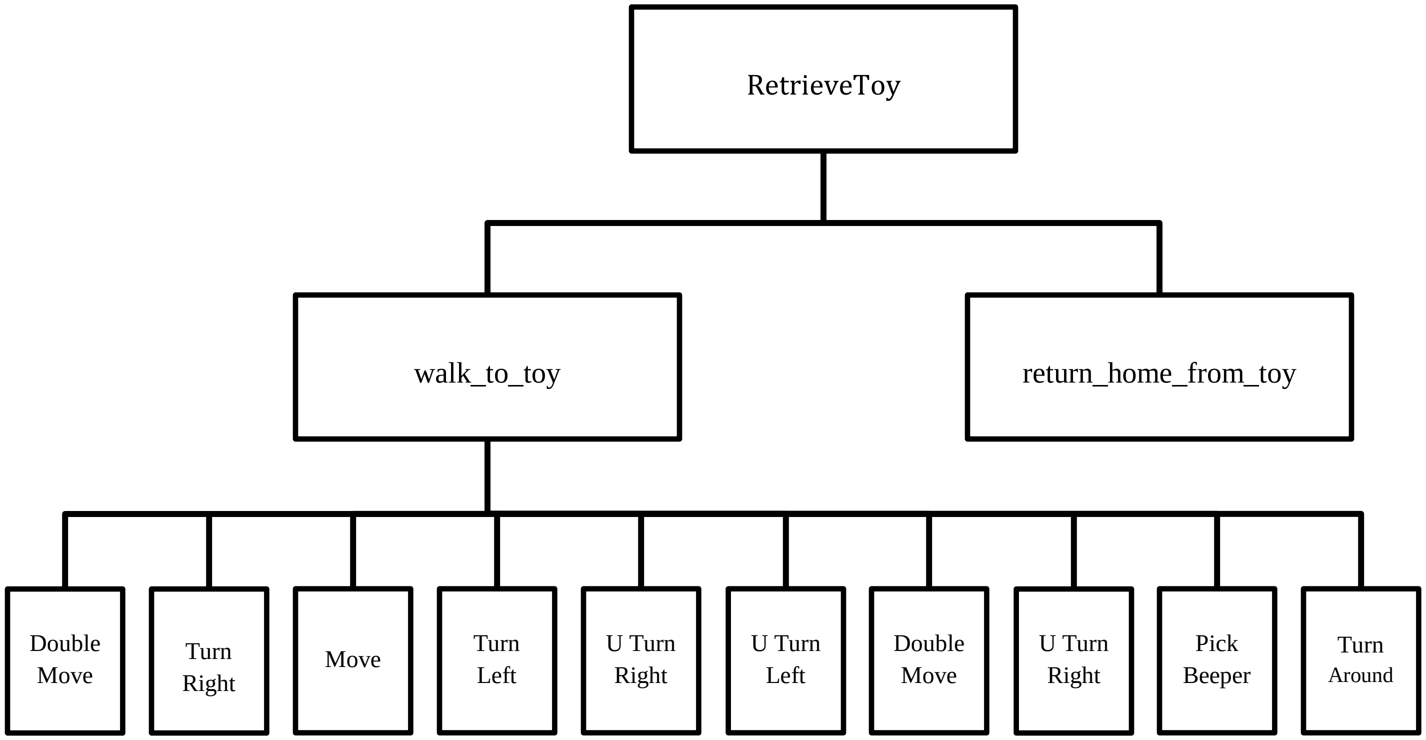
It’s easy to lose track when decomposing complex problems into a set of simpler ones. There are many techniques developed in regards to designing programs; some work better in some situations than others, and sometimes the best approach is to mix and match various methods.

On such method is JSP or *Jackson Structured Programming*. JSP is especially effective for processes that deal with data streams. You don’t want to design a video game with JSP, but a process where two sets of data are merged into a single one lends itself exclusively for JSP.

A JSP process is diagrammed as a box, which might be decomposed into other boxes. As such, initially our `RetrieveToy` process would look like this:



When a process is split up in sub-processes, they are displayed in the diagram left-to-right. This diagram could be extended for `walk_to_toy` as follows:



This approach may seem trivial for a straightforward project like `RetrieveToy`, but can be quite useful for bigger projects.

2.4 Towards a better Karel

One thing that should become clear through the exercises is that it requires *a lot of code* to make Karel succeed in the world. That's because we have to tell him *every little step* he has to take. Right now, even with all the helper functions, it still takes a stunning *nine* steps for Karel to find his way to his toy.

With just some rudimentary intelligence—the ability to sense if he's in front of a wall, and the ability to sense if he's near a sensor—we could replace `RetrieveToy` by something like this:

- Put Beeper (Karel has to find his way back)
- Move

- Find Beeper (this wouldn't work if Karel were on the beeper he just dropped)
- Pick Beeper
- Find Beeper (that would be the beeper left at "home")

Well yes, but now we have that find_beeper procedure to implement. I'm sure that's a lot of code

And here's the kicker: `find_beeper` would, including SUB/END SUB lines, be only *nine* lines long *and it covers Karel's travel in both directions!* What's more, it would work regardless of the size of the yard. *That* is the power of true programming!

2.4.1 Meet SuperKarel

As it turns out, the robot factory sells an advanced KAREL model named SUPERKAREL. We're not so heartless that we sell of poor little Karel—he'll be delegated to cleaning up the back yard, and he seems happy to do so—but from now on, the robot used will be SuperKarel instead.

What is it that SuperKarel offers?

- `TurnRight`
- `TurnAround`
- `MultiMove`
- `OnBeeper`
- `HasBeeper`
- `FrontIsClear`

The first two methods might be a bit annoying. The paint on our `TurnRight` and `TurnAround` code is hardly dry, and now we don't need it anymore! They served mainly an educational purpose though; now that we've learned the trick of writing sub procedures there's no reason to wait for Karel to make his ¾ turns in slow motion. We know we can do that trick ourselves; let's move on and concentrate on more important things!

The `MultiMove` method is interesting; it's like `DoubleMove` but on steroids. `MultiMove` takes a `steps` argument that tells SuperKarel how many moves it should make; 2, 3, 15, 1000… whatever your heart desires!

The last three methods are *functions*; the return a value. For all three methods the values returned are `True|False` values (so called Boolean types). They respectively indicate if there are any (at least 1) beepers in the cell where SuperKarel is; if SuperKarel has any (at least 1) beepers in his bag, and finally whether or not there is a wall right in front of SuperKarel.

It would be nice if SuperKarel could tell us *how many* beepers there are on the ground or in his cell, or what the *distance* to the nearest wall in front of SuperKarel is, but alas. Perhaps there will be a MEGAKAREL in the future who can do that? In the next chapter we'll start with familiarizing ourselves with SuperKarel's new functionality, and learn new language features that can take advantage of it.

3 Letting SuperKarel do the work

The very first step in using our new SuperKarel robot is by adjusting the `Program` procedure in `modRobot`:

```
Public Sub Program(robot As iSuperKarel)
    ' call your program here
    RetrieveToy robot
    ' turn off the robot!
    robot.PowerOff
End Sub
```

The only thing that has changed is the *type* of the Robot; instead of using an `iKarel` class robot, we're now using an `iSuperKarel` class robot. And SuperKarel is backward compatible with the programs we wrote so far; if you try to run `RetrieveToy` you will see that it will run (at least it should) without any problems.

For testing purposes, let's load assignment #1 (HELLO KAREL) and have Program call a procedure `TestSuperKarel` that we'll create in a new (menu INSERT, MODULE) `modChapter3` module:

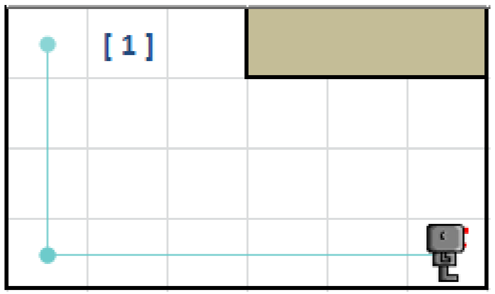
Module `modRobot`:

```
Public Sub Program(robot As iSuperKarel)
    ' call your program here
    TestSuperKarel robot
    ' turn off the robot!
    robot.PowerOff
End Sub
```

Module `modChapter3`:

```
Public Sub TestSuperKarel(robot As iSuperKarel)
    ' let's zip to the other side of the back yard!
    robot.TurnRight
    robot.MultiMove 3
    robot.TurnLeft
    robot.MultiMove 5
End Sub
```

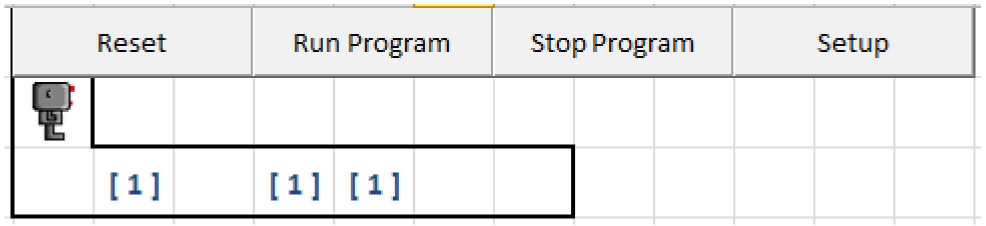
Testing reveals that SuperKarel, indeed, zips to the other side of the back yard:



BAM! That was indeed fast! What's even better is that `MultiMove` is, in fact, the *least* spectacular new function SuperKarel can show us!

3.1 Repetition using the While statement

Let's revisit Assignment #2 from the previous chapter in which little Karel has to clean up a strip in his back yard:



Karel did exactly what we told him to do, but the problem is of course that we had to tell him *exactly* what to do. Why[*] let Karel pick up debris if you have to find out by *yourself* where the debris is in the first place?

3.1.1 Introducing While

In programming, repeating code *under certain conditions* is one of the most common constructs and it is generally referred to as a *While Loop*. Here's how it works in VBA:

```
Do While <Boolean Expression>
    (code that is repeated again and again)
Loop
```

[*] It's actual a rhetorical question. But the answer would of course be: "because it makes the little fellow so happy!"

Visual Basic has to know somehow what code needs to be repeated, and what not (it is likely that there's more code to follow once the repetitive part is over, after all). This is why the repeating code is *enclosed* in the `Do While` statement at the beginning, and `Loop` at the end. In fact, VBA offers no less than *four* different versions of the "Do Loop":

```
Do While <Condition>
    …
Loop
```

```
Do Until <Condition>
    …
Loop
```

```
Do
    …
Loop While <Condition>
```

```
Do
    ….
Loop Until <Condition>
```

`While` and `Until` are each other opposites; `While` will loop as long as a certain condition is met, `Until` will loop as long as a certain condition is *not* met.

Similarly, the condition checks `While` or `Until` can be placed at either end of the loop. Placing it at the end will guarantee that the code will run at least once. For a task like "*keep walking until you hit a wall*" that is obviously not a good idea; if SuperKarel starts right in front of a wall he'd crash right into it.

But another task could be "*process rows until you hit the bottom*." Since our robot *always* is on "a" row, you always want that process to run *at least* once—and that's when you would place `While` or `Until` at the bottom of the loop.

The condition check is a "Boolean Expression." A Boolean type value is either True or False (notated as `True|False`) and a Boolean expression is nothing else but an expression that yields a Boolean value. That could be:

- A function returning a Boolean value, like SuperKarel's `OnBeeper` or `FrontIsClear` functions;
- A comparison of two values, `like BeeperCount > 5` or `Name = "Karel"`;
- The result of a Boolean operation, like `OnBeeper` **`And`** `FrontIsClear`.

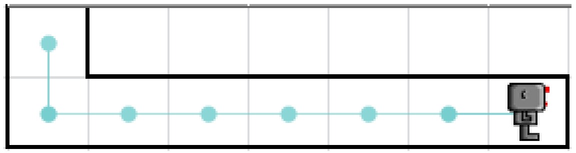
3.1.2 Letting SuperKarel do the work

This is where SuperKarel's enhanced sensor capability comes into play. Instead of guiding Karel step by step through the back yard, we can let SuperKarel do a lot of the work by himself. Create the following program in `modChapter3` and let `Program` in `modRobot` call it:

Try it out! VBA Code

Try it out! VBA Code

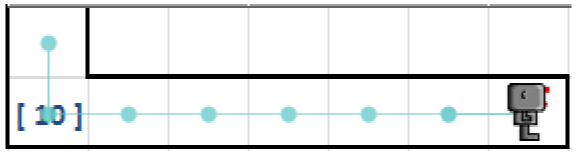
```
1   Public Sub CleanYardStrip(robot As iSuperKarel)
2       ' move into the yard
3       robot.TurnRight
4       robot.Move
1       robot.TurnLeft
2       ' keep advancing until you hit the end
3       Do While robot.FrontIsClear()
4           robot.Move
5           Do While robot.OnBeeper()
6               robot.PickBeeper
7           Loop ' repeat picking up beepers
8       Loop ' repeat advancing to the right
9   End Sub
```



All the code “inside” the while loop (lines 8-9-10-11) will be repeated over and over until the test on line 7 (`FrontIsClear`) fails. And there’s *another* `while` loop *inside* the first one, that instructs SUPERKARL to continue picking up beepers until they’re all gone.

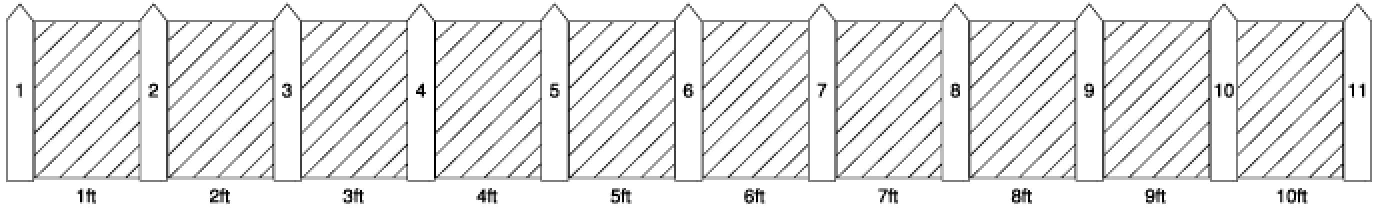
Success! Or not? Clearly our more powerful program is a lot more flexible. But with power comes responsibility; will our program clean up the yard regardless of where the debris is, or how much?

- Reload the assignment.
- Adjust the beepers to 10 in every spot (highlight cells B3:H3, enter 10, press CTRL + ENTER)
- Run the program



The problem is that, in this particular case, the code will end up telling SuperKarel to move east *six* times, since that will cover the entire width of the back yard. However, it is inside the “move” loop that we tell SuperKarel to pick up the beepers, and that is a problem, *since we want to cover all seven cells* in the back yard.

In computer science this is referred to as the *fence post problem*: to cover 10 yards with fence, using a fence pole every foot, you will need *eleven*, not ten, fenceposts:



There's no elegant solution around this. You're letting SuperKarel move one time less than there are cells (otherwise he'd crash into a wall). Which means that when you remove all beepers while moving, you're going to miss a spot.

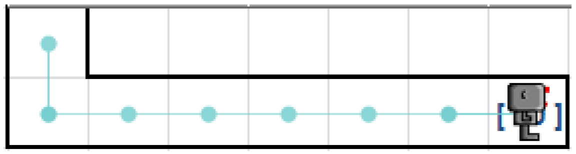
But what if I clear the beepers and *then* move?

Well, yes, that surely solves the problem of "not clearing the first cell"…

Try it out!
VBA Code

```
Public Sub CleanYardStrip(robot As iSuperKarel)
    ' move into the yard
    robot.TurnRight
    robot.Move
    robot.TurnLeft
    ' keep advancing until you hit the end
    Do While robot.FrontIsClear
        Do While robot.OnBeeper
            robot.PickBeeper
        Loop ' repeat picking up beepers
        robot.Move
    Loop ' repeat advancing to the right
End Sub
```

But it replaces it with the problem of "not clearing the last cell"



The only true solution is cleaning up the beepers before you start moving (or after you stop moving, depending on the implementation).

Try it out! VBA Code

```
Public Sub CleanYardStrip(robot As iSuperKarel)
    ' move into the yard
    robot.TurnRight
    robot.Move
    robot.TurnLeft
    ' keep advancing until you hit the end
    Do While robot.FrontIsClear()
        Do While robot.OnBeeper()
            robot.PickBeeper
        Loop ' repeat picking up beepers
        robot.Move
    Loop ' repeat advancing to the right
    Do While robot.OnBeeper()
        robot.PickBeeper
    Loop ' repeat picking up beepers
End Sub
```

And this time SuperKarel *will* clean up all debris, regardless where it is, or how much there is. By now the codebase looks rather large (given the problem) and repeating lines of code in multiple places means only one thing: it's time to decompose.

3.1.3 Decomposition

There are two places in the code where we basically say "clean up all beepers," so it makes sense to create a separate procedure for that. Picking up all beepers is likely something that will happen more often, so it's probably best to regard it as a fairly generic procedure that should be placed in `modRobot`. And while we're at it, we might as well write the reverse function to put all beepers in the bag in a cell.

Try it out! VBA Code

```
Public Sub CleanYardStrip(robot As iSuperKarel)
    ' move into the yard
    robot.TurnRight
    robot.Move
    robot.TurnLeft
    ' keep advancing until you hit the end
    Do While robot.FrontIsClear()
        PickAllBeepers robot
        robot.Move
    Loop ' repeat advancing to the right
    PickAllBeepers robot
End Sub
```

And here are the generic helper functions to be placed in `modRobot`:

```
Public Sub PickAllBeepers(robot As iSuperKarel)
    Do While robot.OnBeeper()
        robot.PickBeeper
    Loop
End Sub

Public Sub PutAllBeepers(robot As iSuperKarel)
    Do While robot.HasBeeper()
        robot.PutBeeper
    Loop
End Sub
```

But that is not the only decomposition we can make. Why not start with taking out some of the complexity of the code, and delegate it to a helper function called `clean_up_row`? This would not be as generic as the `PickAllBeepers` and `PutAllBeepers` function, so it's better placed as a `Private` procedure in the `modChapter3` module:

```
Private Sub clean_up_row(robot As iSuperKarel)
    ' -----------------------------------------------------
    ' clean up the row Karel is in
    ' assume he's facing east
    ' -----------------------------------------------------

    ' walk to east wall
    Do While robot.FrontIsClear
        robot.Move
    Loop

    ' go to west wall, picking up all beepers
    ' start with cleaning current position
    robot.TurnAround
    PickAllBeepers robot

    ' process remaining cells
    Do While robot.FrontIsClear
        robot.Move
        PickAllBeepers robot
    Loop

    ' return to facing east
    robot.TurnAround
End Sub
```

Instead of cleaning up at once, Karel is first sent to the far east side of the yard. *Then* he walks back west, picking up beepers as he goes. The advantage of this method is that it covers situations where he enters a new row not on the far left side of the map (in case of a non-rectangular garden).

We can now simplify our `CleanYardStrip` procedure:

Try it out! VBA Code

```
Public Sub CleanYardStrip(robot As iSuperKarel)
    ' move into the yard
    robot.TurnRight
    robot.Move
    robot.TurnLeft
    ' keep advancing until you hit the end
    clean_up_row robot
End Sub
```

It's important to test the code, to make sure it works. But once testing is complete, why stop with improving our code right now? It's a small step to rewrite to code to clean up *any* row in the garden. Note that we changed the name of the procedure; the call in `Program` has to be updated accordingly!

Try it out! VBA Code

```
Public Sub CleanYard(robot As iSuperKarel)
    ' Clean row, check if there's more
    clean_up_row robot
    robot.TurnRight
    Do While robot.FrontIsClear
        robot.Move
        robot.TurnLeft
        clean_up_row robot
        robot.TurnRight
    Loop
    ' move back to top
    robot.TurnAround
    Do While robot.FrontIsClear
        robot.Move
    Loop
    ' face back east
    robot.TurnRight
End Sub
```

Once the code has been tested successfully, why not load ASSIGNMENT #3 and see how the code deals with *that* scenario? This is truly programming; we now have a flexible solution that works under multiple conditions!

And this completes the exercise?

Of course not. Because there's more optimizing to be done. Careful study of the code reveals yet another decomposition to be made; on multiple occasions the action "keep walking until you hit a wall" is performed. The most obvious thing to do is to make a generic `WalkToWall` procedure (and it's generic enough to be placed in the `modRobot` module):

Try it out!
VBA Code

```
Public Sub WalkToWall(robot As iSuperKarel)
    Do While robot.FrontIsClear
        robot.Move
    Loop
End Sub
```

Subsequently, we can clean up our `clean_up_row` code:

Try it out!
VBA Code

```
Private Sub clean_up_row(robot As iSuperKarel)
    ' -----------------------------------------------------
    ' clean up the row Karel is in
    ' assume he's facing east
    ' -----------------------------------------------------

    ' walk to east wall
    WalkToWall robot
~~Do While robot.FrontIsClear~~
        ~~robot.Move~~
    ~~Loop~~

    (… rest of code …)
End Sub
```

And in addition, our CleanYard procedure can now be rewritten too:

Try it out! VBA Code

```
Public Sub CleanYard(robot As iSuperKarel)
    ' Clean row, check if there's more
    clean_up_row robot
    robot.TurnRight
    Do While robot.FrontIsClear
        robot.Move
        robot.TurnLeft
        clean_up_row robot
        robot.TurnRight
    Loop
    ' move back to top
    robot.TurnAround
    WalkToWall robot
    ' face back east
    robot.TurnRight
End Sub
```

And *now* our yard cleaning project is pretty much done!

But all those turns… can't we fix that?

When looking at the main loop in the code we can see that, in order to move to a new row, Karel has to turn, and in order to clean up the current row, Karel has to turn back. Wouldn't it make more sense to simply rewrite the code so that he starts cleaning up a row facing south, and put those turns in there? It would make the `CleanYard` procedure a lot simpler, right?

```
    clean_up_row robot
    Do While robot.FrontIsClear
        robot.Move
        clean_up_row robot
    Loop
```

That is certainly a consideration, and the student is encouraged to experiment with that solution. The downside of it is that the relative generic `clean_up_row` procedure now starts with Karel facing south, which makes less sense then Karel facing east *in the direction he's going to clean*. Besides that, *other* code that uses clean_up_row might break now, because *that* code assumes that Karl is facing east! Now we have to hunt down where `clean_up_row` is being used in the code, that doesn't make things simpler!

A better approach would be to make a `clean_row_facing_south` procedure:

```
Private Sub clean_up_row_facing_south(robot As iSuperKarel)
    robot.TurnLeft
    clean_up_row robot
    robot.TurnRight
End Sub
```

And then we can update `CleanYard` accordingly:

```
Public Sub CleanYard(robot As iSuperKarel)
    ' Clean row, check if there's more
    clean_up_row robot
    robot.TurnRight
    Do While robot.FrontIsClear
        robot.Move
        clean_up_row_facing_south robot
    Loop
    ' move back to top
    robot.TurnAround
    WalkToWall robot
```

```
        ' face back east
        robot.TurnRight
    End Sub
```

Note that, since Karel is starting facing east, there's no reason to make a turn south and call `clean_up_row_facing_south` for the initial cell instead; it's simpler to stick to the initial `clean_up_row` there. And that truly concludes our garden project!

3.2 Practice assignments

3.3 Future Assignments

3.3.1 Ballot Validator

After all the problems in the 2000 presidential election in Florida, most states eliminated the punch-card ballot problems that made *hanging chad* a household phrase at the end of 2000. The most common replacement is electronic voting machines—which have problems of their own, as we saw in several districts in the 2006 elections—largely because they do not generate any sort voter–verifiable paper trail, making it impossible to conduct recounts or determine whether fraud has occurred.

Another possible (if fanciful) strategy for securing elections would be to retain the punch cards but to have a miniaturized robot—Karel, of course—check each of the ballots before it is counted to ensure that no unwanted chad remains. Karel's job is to move across the punch card ballot and make sure that no stray bits of the card remain in any of the ballot spaces in which the user that no stray bits of the card remain in any of the ballot spaces in which the user has attempted to cast a vote.

To make this task more concrete, imagine that Karel is sitting at the extreme left edge of a punch–card ballot that looks like this:

The partially enclosed rectangles in the interior of the card represent the areas of the ballet that voters must punch out to record their preferences. On the original ballot, these rectangles are completely filled with beepers, as shown in the ballot rectangles on the 2nd and 8th avenues. Ideally, the voter will punch out all the beepers when casting a vote, leading to an empty rectangle of the sort shown in the 4th column.

Unfortunately, some bits of the card—the chad—sometimes end up remaining in the hole, as shown on the 6th column, where the beeper at the top of the rectangle is still attached to the ballet. The situation on the 10th row, however, is even worse in that the punched out beeper from the middle of the rectangle has ended up in the bottom of the rectangle, leaving *two* beepers on that square.

- Ballot validator
- Repair columns
- Find center
- ???

3.4 Repetition with for/next