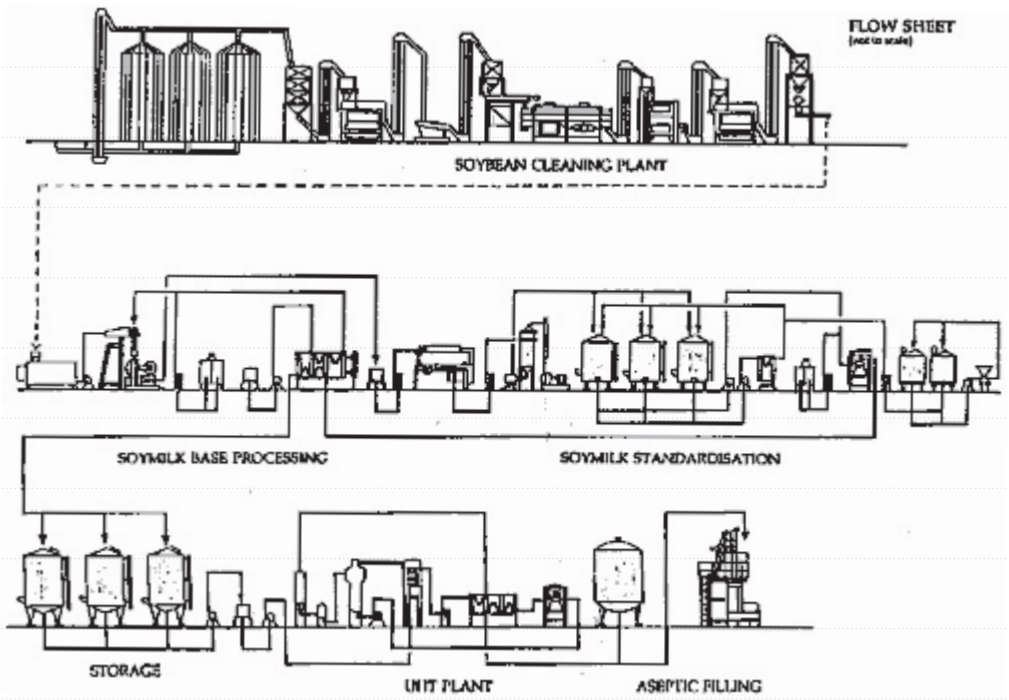
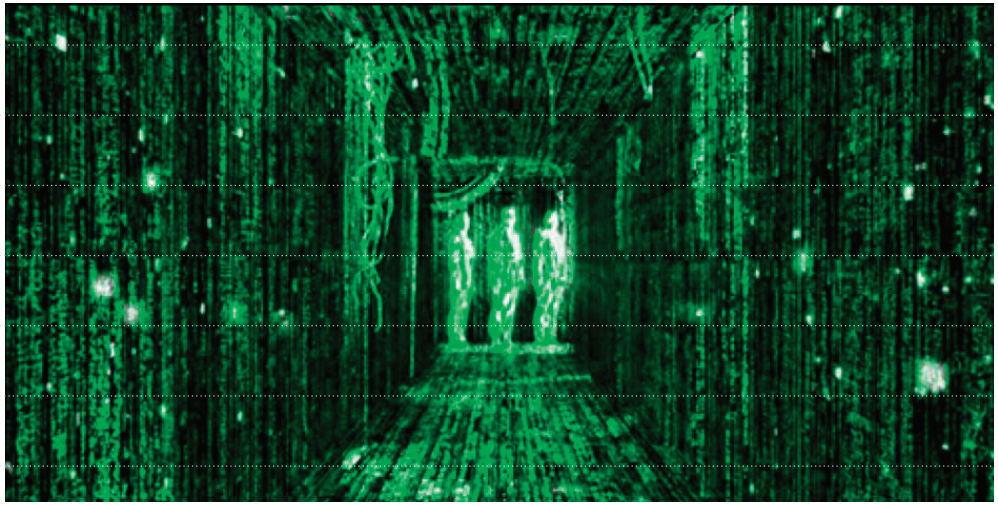


Procedural

VBA is a fundamentally a procedural language as discussed last week. The difference between procedural and object oriented (OOPS) manifests itself through the flow of writing. Procedural is written from top to bottom in one sequence, whereas OOPS doesn't 'look' like a sequence but is dependent on events within objects. Thus, the OOPS code seems non-linear as opposed to procedural.

On the surface, AutoCad is also compiled into objects that one can address. It is the VBA code you write, which interfaces with ACAD ojects that is procedural. Each object has properties, a data-structure, and methods, procedures to manipulate the data. In OOPS these two categories are aggregated into one unit - on object - that can be cloned, like parents and their offspring. In procedural on the other hand, one specifies the data-structure and the methods independently, and rolls them into a calling sequence.

Today we will take a closer look at the syntax of the VBA procedural environment. First explaining what type of variables we generally use in common data-structures and then the different method types - sub-procedures and functions. The essence of procedural lies within the passing of data between the methods - ***from main-procedure to sub-procedures*** - where it gets manipulated. In that context, one talks of the scope of a variable. An obvious analogy to the syntax of programs is language itself, where the data-structure are the scope of words like nouns, adjectives and verbs, and the procedures are the grammatical manipulations of those words.



(Data) Types

A program generally (generally, because as Paul points out in his new book that AutoLisp manages to disolve the distinction between data and procedure) consists of an algorithm or procedure and data. If the concept of the algorithm is clear then the data structure has to be determined.

There are several types of data. A type determines three things: - how much memory to attribute to the data, how to decode it and where it can be applied - its scope (globally or locally, fixed or dynamic).
The most commonly used data types for us are

integer	(2 bytes)	%
double	(2 bytes)	#
long	(4 bytes)	&
string	(63 bytes)	$
variant	(16/ 22 bytes)	

The data types above as you can see have a quantity of memory allocated to them. That means when you use a data type to dimension a variable, you reserve memory for that variable of a certain type. The variant type is a 'token' that is used if a variable has not been clearly defined yet. It can take any type definition for a variable later in the program.
The symbols you see behind the types are shortcuts, meaning that if you want to quickly use a variable that shows up once only as a helper to a calculation for example, you can use the symbol at the moment the variable appears in the program, not having to define the variable at the beginning of the procedure.

Variables

Variables are as the name suggests data that can change over the course of the execution of the program - variable data. They are also named storage locations. As shown above, a program needs to have some data and an algorithm to do any work. A data type contains a certain amount of memory and scope. Now, we can give that memory location a name. The name can be arbitrary, as long as it doesn't coincide with some word that is being used by the program as a predefined structure. The naming and scoping of a variable is done like this:

```
Dim variable-name As type
```

The expression *Dim* and *As* are fixed part of the syntax. The variable-name could be any and the type any of the types that are listed above (and others actually). However, you want to make sure you try to use the right type for the variable, since otherwise one sends the variables around with a load of extra memory or too little memory. There are example codes you will find that state at the beginning of the module *Open Explicit*, which indicates that all the variables have to be dimensioned explicitly and rigorously.

Inside any application that is linked to VBA, like AutoCad or Microstation, there are a lot of application specific types that are construed out of the basic data types. An AutoCad object for expample has its own type as the line-object is of type AcadLine. So you could give a line that is to be drawn a name and its type like this:

```
Dim myline As AcadLine
```

This dimensions not just the memory location but a large field of memory which reserves also space for further definitions of the object/class line. 'myline' is in other words an instance of the AcadLine class.
Variables' life-span last only for the duration of the execution of the program. Afterwards, the memory gets cleared out again.

In addition, you can also create your own types. If you were to use a certain combination of variables (different of the same) a lot that make up a single entity then you could construe that entity yourself made of various basic types. Instead of the normal definition of a 3D point in AutoCad as

```
Dim point(2) As double
```

with its initialization as

```
point(0) = 0#: point(1) = 0#: point(2) = 0#
```

you could make it more elegant by defining the point first as

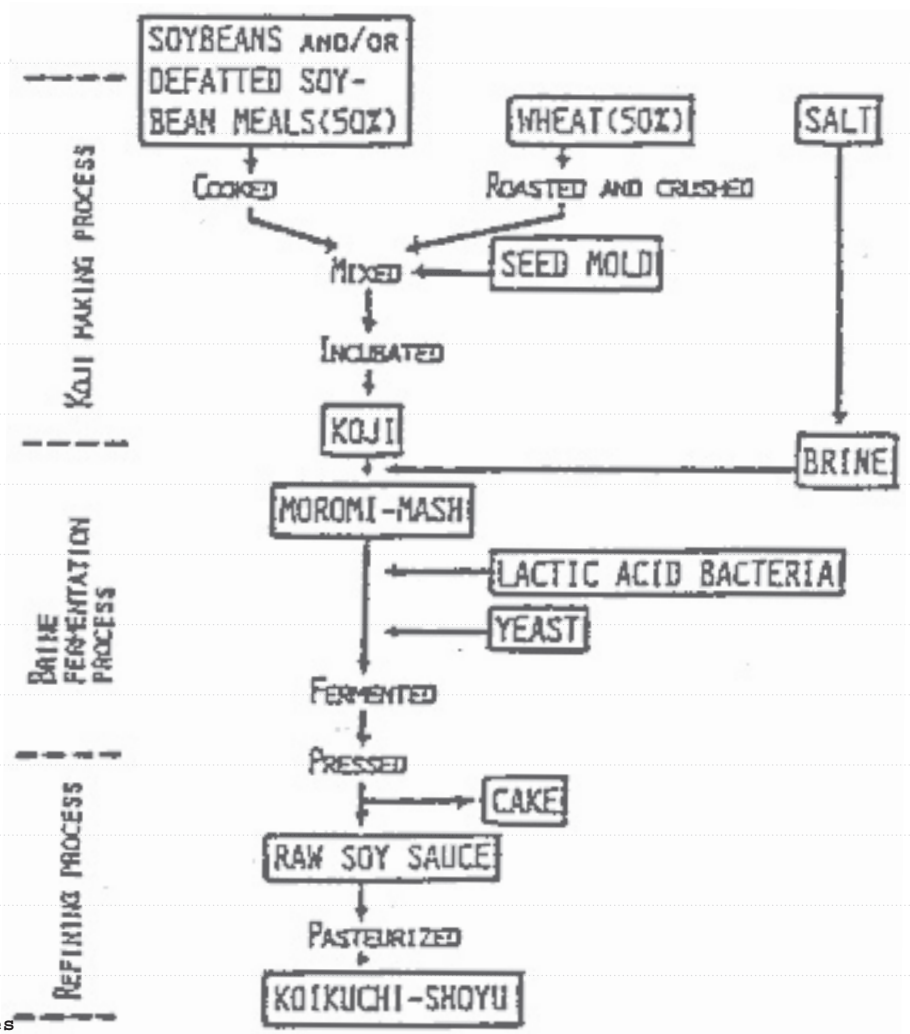
```
Type point
      Dim x As double
      Dim y As double
      Dim z As double
End Type

Dim here As point
```

with its initialization as

```
here.x = 0#
here.y = 0#
here.z = 0#
```

Reads easier and makes it clearer in the program the longer it goes on.



Procedural Methods

As already shown on last week's hand-out, an object consists of Properties, Methods and Events. The Properties define qualities of the object, Methods define manipulation of the objects data and an Event is a special case defined to execute only if a defined change of the object occurs.
Having looked briefly at data definitions that are used to define properties, we will now look at methods.
Methods come in two flavours:

- sub-procedures
- functions

The name sub-procedure hints at its role in the hierarchy within the program: it serves the main-procedure, and is usually called from there as well. The main-procedure is the overarching sequence that decides the linear execution of the program. Sub-procedures are the outsourced 'limbs' of the program. If every computation of the algorithm was written out in one long sequence, it would become less comprehensible. Thus, one reduces the complicatedness by breaking the code up into chunks of computational tasks. When a sub-procedure, or for that matter a function has finished computing, it returns with the result to the procedure it was called from. A nice analogy is to jump to a footnote in a text and returning back to where one left off.
The difference between sub-procedures and functions is that sub-procedures can be send a variety of data and types to be computed and returned, whereas functions return exactly one computed type of data, making them more limited. I will demonstrate that with the startpoint example from last week:

We could either ask a function to return a random number as we did last week:

```
        there(0) = random(0, 25)
        there(1) = random(0, 25)
        there(2) = random(0, 25)
...

Function random(low As Double, up As Double) As Double

    random = (up - low) * Rnd + low

End Function
```

Notice that the single **argument** that is returned needs to be *assigned* to some variable of the same type. That is also why the Function itself is dimensioned as type double. The name of the function itself contains the value!

We can re-write this as a call to a sub-routine:

```
        random there, 0, 25
...

Sub random(ret_pt() as double, low as double, up as double)

        ret_pt(0) = (up - low) * Rnd + low
        ret_pt(1) = (up - low) * Rnd + low
        ret_pt(2) = (up - low) * Rnd + low

end Sub
```

The call to the sub-procedure *random* is effectuated just by typing its name. After the name a space is necessary before ***listing the arguments*** for the sub-procedure to compute with, here the point *there*, the *lower limit* (0) and *upper limit* (25) for a random number. The *order of the argument list* is important and should be repeated exactly when the function or sub-procedure receives them. The function random gets and receives two arguments (0,25), whereas the sub-procedure random gets and receives three arguments (there, 0 ,25)!

One can see that instead of returning just one value, the sub-procedure returns three values.

It might also seem strange that the arguments in the call to the Function random() have changed their name. Variables are only sent as copies of themselves and can therefore change their names between Sub Procedures or Functions. The content remains the same, though as a copy.