

~~maXbox~~



maXbox Starter 31

Start with Closures

1.1 A Function Block in a Box

Today we step through a topic of closures.

One of the questions that comes up when learning closures is how they can be useful when they're no less verbose as traditional methods of callbacks, IoC and delegation. The answer is that closures or methods don't have to be so verbose; they are reusable. What are closures:

👉 They are a block of code plus the bindings to the environment they came from (Ragusa Idiom / function object).

First we will start with an anonymous method:

```
procedure  
begin  
    writeln('Anonymous method has executed');  
end
```

This isn't very different from a normal procedure. You'll notice that the procedure has no name. You'll also notice the missing semicolon after the final end like others in maXbox (whitespacing).

Without a name, we have to call them by a reference. Maybe you know a procedure type as long ago used in many languages before object oriented coding was up:

```
Type  
TMath_Func = PROCEDURE(var x: single);
```

Such a procedure type can be called or passed by reference.

Procedure type objects have the addresses of procedures as values. You can specify procedure types like above: `Tmath_func = Procedure.`

P-types are often used as parameters to other procedures like also a closure does. This is useful when the procedure specifies an algorithm that involves calling another procedure or function, and the called procedure or function varies from one call to the next.

var

```
fct1x, fct2x, fct3x: TMath_Func;
```

A good example of the use of a P-type is a numeric integration procedure or a simple function plot. The procedure implements the algorithm of a numeric math function but the function being plotted or integrated is passed in as a parameter:

```
PROCEDURE nth_power(var x: single);  
BEGIN  
  x:= Power(x,4);  
END;
```

Just like a reference to a procedure or procedure of object, a method reference must have a type. The type must be a matching procedure or function declaration, means of the same signature:

type

```
TMath_Func = PROCEDURE(var x: single);  
             PROCEDURE nth_power(var x: single); //same match
```


So how do we call that function type?

Variables of procedure types can be assigned the value of a procedure that has the same types of parameters and return value, and the same attributes. We call it:

```
fct3x:= @nth_power;
```

or pass it to another function:

```
fct_table(2,100,0.1, @nth_power,'Power of(x) Closure scheme');
```

 You can now make further assignments by assigning the p-type to another reference. This reference can be another variable, a method parameter, a class field and so on. As long as they are all of the same type, assignments can be made as needed.

Now we go back to our anonymous method on first page. How do call that method if it has no name? Right it must be used with a reference.

That reference can be a variable to which the method is assigned.

```
var
  fct3x: TMath_Func;
var
  myAnonymousMethod: TProc;
```

☞ Only procedures that are not contained in other procedures can be assigned as the values of procedure variables. Standard procedures cannot be assigned as values of procedure variables.

Now we step from anonymous methods and p-types to a closure.

You remember the definition:

A block of code plus the bindings to the environment they came from.

This is the formal thing that sets closures apart from function pointers, procedure types and similar techniques.

As more "things" on planet Earth are converted to the inventory set of digitally connected Internet devices, the roles and responsibilities of web developers and technology managers will need to evolve with closures.

Hope you did already work with the Starter 1 till 30:

<http://sourceforge.net/apps/mediawiki/maxbox/>

We start a closure with simplification, the image below is my sight of a closure: the cards are functions with red variables of game state which are enclosed by a black box and can be called by ref name behind.



☞ Simplification: Your classes are small and your methods too; you've said everything and you've removed the last piece of unnecessary code.

1.1.1 Get the Closure Code

Closures are reusable blocks of code that capture the environment and can be passed around as method arguments for immediate or deferred execution.

The term closure is very often treated as a synonym for the term anonymous methods. Though closures are specific to anonymous methods, they are not the same concept. Why?

The **scope** is missing as I want to show you now.

You see that our p-type has a parameter x through which I pass the current base value to the const with exponent 4.

```
PROCEDURE nth_power(var x: single);
  BEGIN
    x:= Power(x,4);
  END;
```

So the value **4** isn't a passed parameter to that procedure, lets define now the closure in Python (Pascal available in mX4):

```
def generate_power_func(n):
    print "id(n): %X" % id(n)
    def nth_power(x):
        return x**n
    print "id(nth_power): %X" % id(nth_power)
    return nth_power
```

Here the inner function `nth_power(x)` is the code-block of the closure. The trick that you want to notice in what's going on is what happens to the value of x like x^n . The argument x is now a local variable in outer() and the behavior of local variables isn't normally very exciting. But in this case, x is global to the function inner() schema.

```
>>> def outer(n):
...     def inner(x):
...         return x^n
...     return inner
...
>>> customInner=outer(4)
>>> customInner(2)
16
```

And since inner() uses the name, it doesn't go away when outer() exits. Instead inner() captures it or „closes over“ it. You can now call outer() as many times as you like and each value of x will be captured separately!

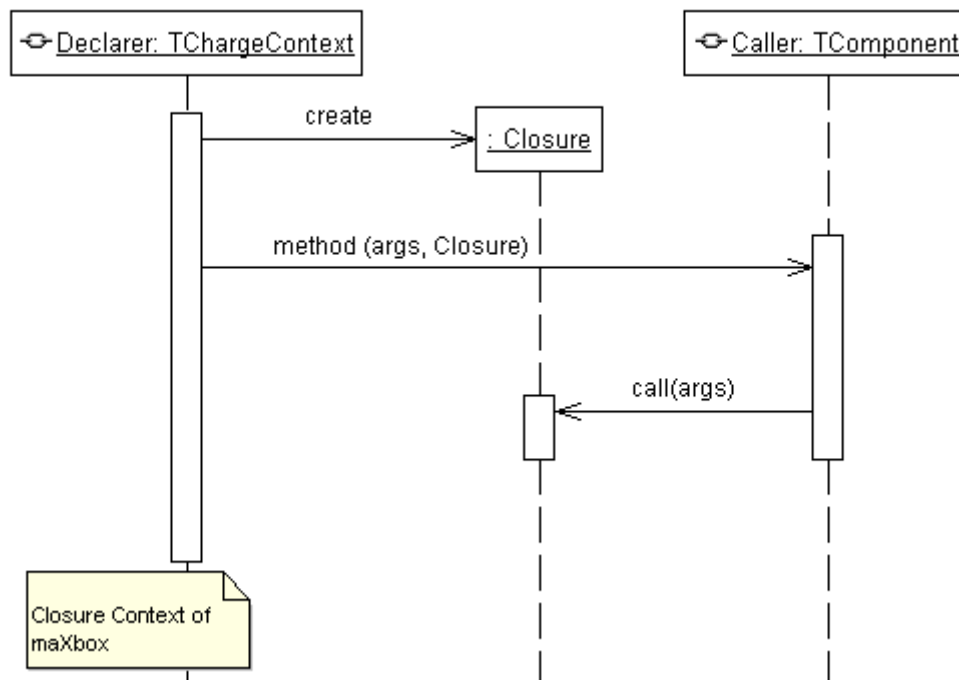
```
>>> refpow_to_4 = generate_power_func(4)    //ref instance
>>> refpow_to_4(2)                          //call
16
>>> refpow_to_3 = generate_power_func(3)
>>> refpow_to_3(10)
1000
```

The function that's returned is called a closure.

When a function is written enclosed in another function, it has full access to local variables from the enclosing function; this feature is also called lexical scoping.

Closures create another scope which includes any symbols used by the anonymous method but which are not declared inside the method itself, like the above parameter **n**. This **scope** is then bound to the anonymous method and follows it everywhere it goes.

The image below declares this context with arguments which a caller can follow in another context.



So now you're wondering why I just spent this time explaining closures when it works just like a nested method anyway and you don't have to take any steps to make it work. Without closures you can think of:

- In C or Pascal you think as a function pointer or procedure type
- In Java as an anonymous inner class
- In Delphi or C# you would consider a delegate.

The reason is because a closure is not really a direct access to the symbols in the outer scope (its sort of compiler magic - see appendix) so you need the old basics; on the other hand a closure is more flexible and reusable than all the others.

If you have ever written a function that returned another function, you probably may have used the closure concept even without knowing about them.

We should keep 3 things in mind:

1. The ability to bind to local variables is part of that, but I think the big reason is that the notation to use them is simple and clear.
2. Java's anonymous inner classes can access locals - but only if they are declared final.
3. With closures we can avoid:
 - Bad Naming (no naming convention)
 - Duplicated Code (side effects)
 - Long Methods (too much code) and temporary Fields (confusion)
 - Long Parameter List (Object is missing)
 - Data Classes (no methods)
 - Large Class and Class with too many delegating methods
 - Coupled classes

Without closures, you would use more upper variables using parameters and you would repeat yourself because you can't pass your function to another function like the following counter ex. stresses:

```
function newCounter ()
  local i := 0
  return function ()  //-- anonymous function
    i := i + 1
    return i
  end
end

c1 := newCounter()
print(c1())  --> 1
print(c1())  --> 2
```

You can pass the counter function to print as many times you like and the scope as the local var goes with the result of the call¹, otherwise it would be initialised by 0 over and over again!



¹ Note that this means that result cannot live only in CLOSURES

Simply put, a closure is a function plus all it needs to access its upvalues correctly (local vars in most of the time). If we call `newCounter` again, it will create a new local variable `i`, so we will get a new closure, acting over that new variable (and you avoid globals too!):

```
c2 = newCounter()  
  print(c2()) --> 1  
  print(c1()) --> 3  
  print(c2()) --> 2
```

👉 Why is that so interesting and important? Because functions are in a closure sense first-class values with memory of scope.

1.1.2 Some Practise

As you already know the tool is split up into the toolbar across the top, the editor or code part in the centre and the output window at the bottom. Change that in the menu `/view` at our own style.

👉 In `maXbox` you will **not call closures only callbacks/p-types in a script**, so the part above is also an introduction to other languages but closures are in the pipe of `mX4`.

👉 Before this starter code will work you will need to download `maXbox` from the website. It can be from <http://www.softwareschule.ch/maxbox.htm> (you'll find the download `maxbox3.zip` on the top left of the page). Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click `maXbox3.exe` the box opens a default demo program. Test it with `F9 / F2` or press **Compile** and you should hear a sound. So far so good now we'll open the examples:

271_closures_study_workingset2.txt
271_closures_study2.txt

http://www.softwareschule.ch/examples/271_closures_study2.txt

👉 One way in which anonymous methods are useful is by simplifying code for common coding patterns. For instance, the following code pattern is probably familiar or common:

```
procedure TestClosureTask;  
var OldCursor: TCursor;  
  begin  
    OldCursor:= Screen.Cursor;  
    try  
      Screen.Cursor:= crHourglass;
```

```

    // Do some lengthy process to show hourglass cursor
finally
    Screen.Cursor:= OldCursor;
end;
end;

```

So, this kind of boilerplate construct calls for an anonymous method or callback method:

```

type TProc = procedure;

procedure repeatProc;
var i: integer;
begin
    for i:= 1 to 9999999 do
end;

procedure ShowHourGlass (Proc: TProc);
var
    OldCursor: TCursor;
begin
    OldCursor:= Screen.Cursor;
    Screen.Cursor:= crHourGlass;
    try
        Proc(); //calls back!
    finally
        Screen.Cursor:= OldCursor
    end;
end;

    //lambda expression!
    //def adder = {x,y -> return x+y}

begin
    ShowHourGlass(@repeatProc); //call

```

The result is fewer lines of code and a general or common way to display the hour glass cursor in all instances (dry - don't repeat yourself).

This example requires two types of units: TCursor and Screen the second one is a global package value of the VCL.

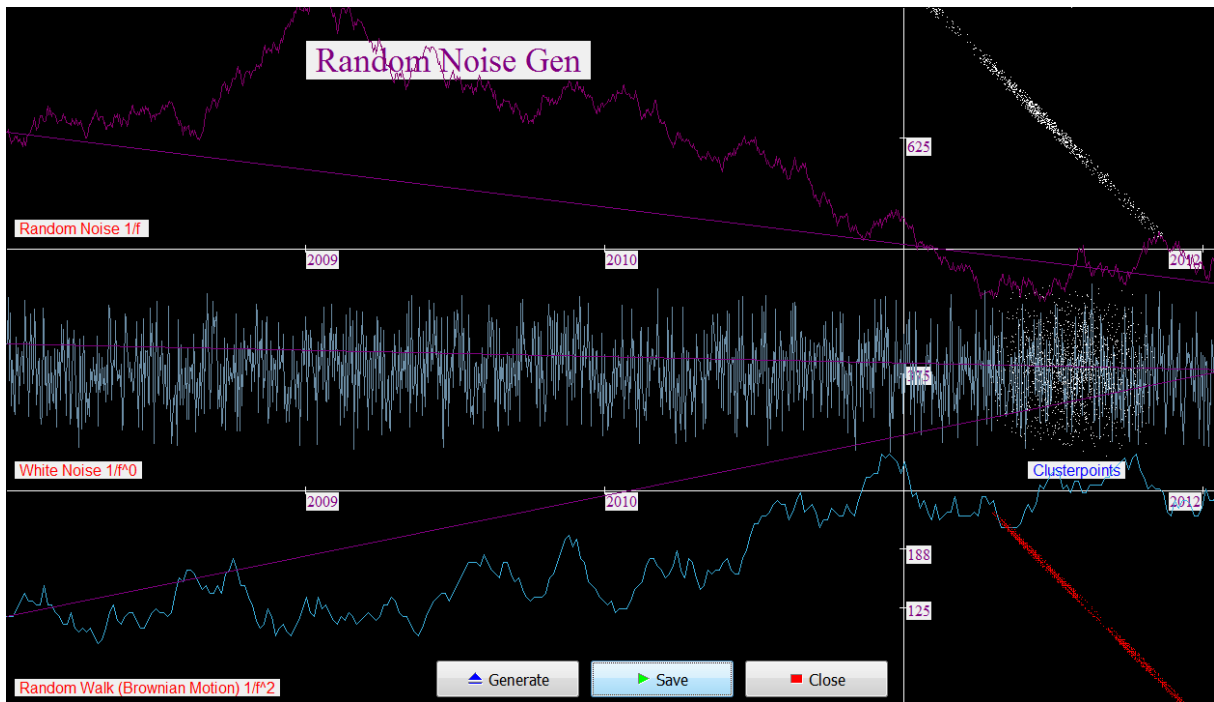
Lets take a “close” look at a closure as a callback:

```

function AlertThisLater(message, timeout)
    {
        function fn() { alert(message); }
        window.setTimeout(fn, timeout);
    }
AlertThisLater("Hi, PascalJScript", 3000);

```


A closure is created containing the message parameter, `fn()` is executed quite some time after the call to `AlertThisLater` has returned, yet `fn()` still has access to the original content of message!



4: The GUI of the Closure App



5: A browser set of a closure

The increase in importance of web services, however, has made it possible to use dynamic languages (where types are not strictly enforced at compile time) for large scale development. Languages such as Python, Ruby and PHP support closures.

👉 Scheme was the first language to introduce closures, allowing full lexical scoping, simplifying many types of programming style.

1.1.3 Closure Objects

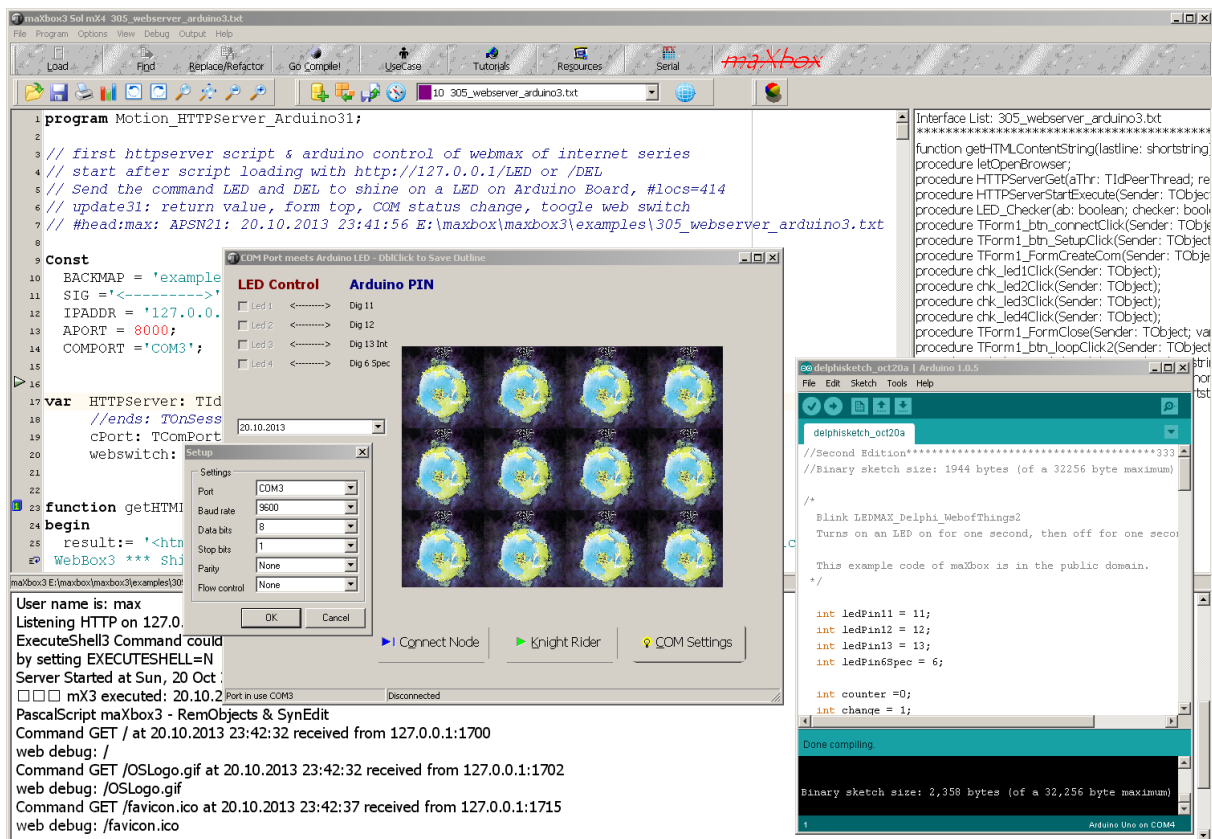
Close to the end of the last part some knowledge about object functions. Closure objects can be divided into three basic types, which reflect how a closure was initiated and what a local object is referenced to.

These 3 are

- When a function is written enclosed in another function, it has full access to local variables from the enclosing function; this feature is called lexical scoping.
- Evaluating a closure expression produces a closure object as a reference.
- The closure object can later be invoked, which results in execution of the body, yielding the value of the expression (if one was present) to the invoker.

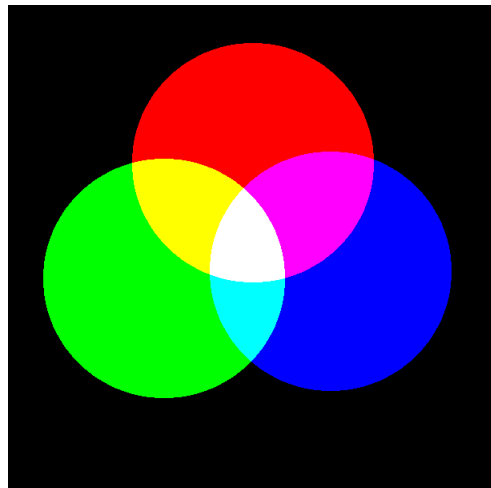
☞ Therefore the inner function can access variables in the calling function, i.e. its closure.

The compiler has to make decisions about where the data comes from and what it can do with that data. Sometimes it will determine that a closure is not possible and will give you simply an error.



6: the BOX, Arduino and the GUI is NOT a closure ;-)





[maXbox Arduino Framework](#)

1.3 Appendix Studies

History

- Lisp started in 1959
- Scheme as a dialect of Lisp
- One feature of the language was function-valued expressions, signified by *lambda*.
- Smalltalk with Blocks.
- Lisp used something called *dynamic scoping*.

Closures Studies and Sources:

http://tronicek.blogspot.com/2007/12/closures-closure-is-form-of-anonymous_28.html

<http://www.javac.info/>

```
def generate_power_func(n):
    print "id(n): %X" % id(n)
    def nth_power(x):
        return x**n
    print "id(nth_power): %X" % id(nth_power)
    return nth_power
```

```
>>> raised_to_4 = generate_power_func(4)
id(n): CCF7DC
id(nth_power): C46630
>>> repr(raised_to_4)
'<function nth_power at 0x00C46630>'
```

```
>>> del generate_power_func
>>> raised_to_4(2)
16
```

PHP:

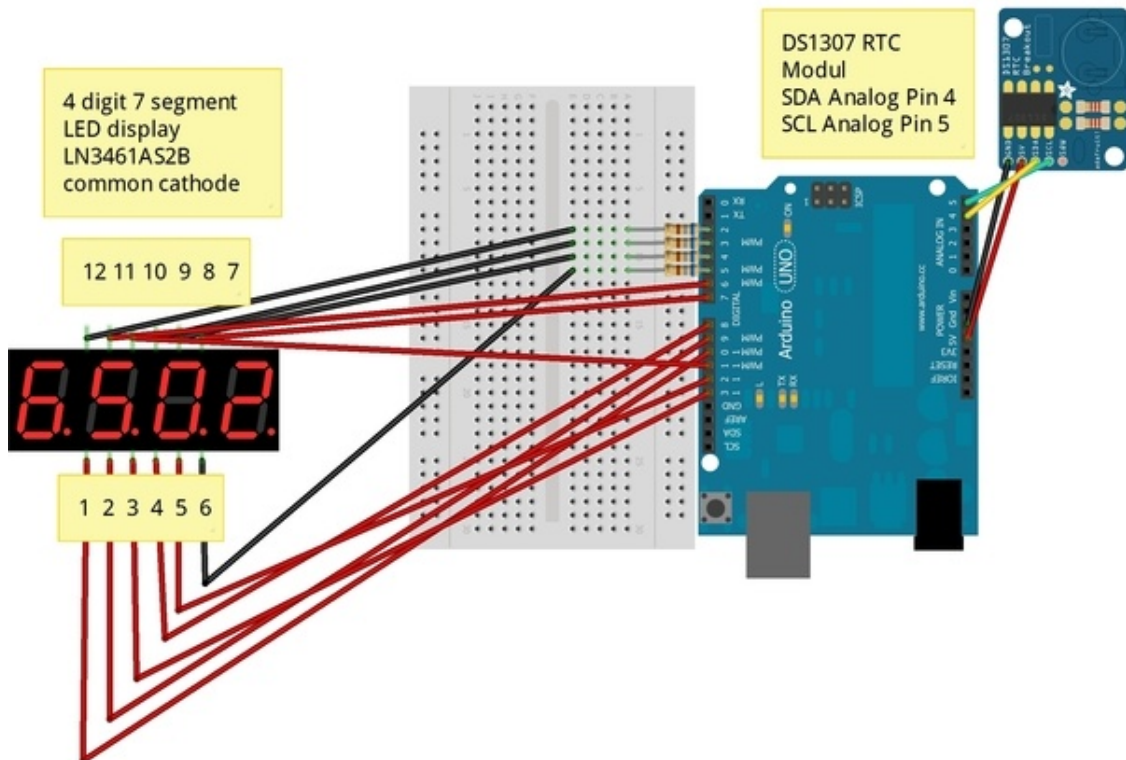
```
[/php] [php] # define a function within a function and return it to the
calling scope - # a inner function can access variables in a calling
function, its closure
```

```
def times(n):
def _f(x):
return x * n
return _f
```

```
t3 = times(3)
print t3 #
print t3(7) # 21
[/php]
```

Here the inner function `_f()` is the block of code of the closure, similar to the `lambda` code in the first example.

Here an embedded microcontroller system seems like a closure as a functional programming with blocks of code:



Made with  Fritzing.org

RTClock: Arduino by Silvia Rothen

<http://www.ecotronics.ch/ecotron/arduinocheatsheet.htm>