Programming in Creative Basic

In an earlier section, we took a general look at variables and data types. Now we will look specifically at writing programs using Creative Basic.

A computer language is just a way of expressing your application in a form the computer can process. (It's own internal language is too detailed for most people to work with).

To this end, a language needs to be concise, easy to understand, and intuitive to work with.

After many years programming in all sorts of languages, I've concluded that Basic, and Creative Basic in particular, meets those aims. For all sorts of applications from science to games, in my opinion, it's the most user-friendly and productive tool.

So let us take a look at the beautiful language in all it's simplicity.

We will work initially in a Console window, so copy the skeleton console program into the Creative editor - and away we'll go ..

```
openconsole
cls

do:until inkey$ <> ""
closeconsole
end
```

The Basic Tools

All programs will use a number of variables, and their type needs to be defined. Some will be Integers, some Decimals (Float or Double), and some Strings.

This is done with a DEF(ine) statement ...

```
def i, j as int
def a$ : string
```

We now have two integer variables 'i' and 'j' and a string variable 'a$' ready for use.

In the same way, you can specify as many variables as you need, of any required data type,

Once you have some variables, you will want to place some values into them.

That's very easy, using statements like **variable = expression** ... the expression being any combination of constants, variables and functions.

```
i = 5
j = i + 2
a$ = "A string"
```

You now have some variables and some data in them. So you'll want the program to display the values and the results of any calculations.

We only need one other command to do that .. the **PRINT** command.

```
print a$
print "The value of j is: ",j
```

That's all we need for a simple program. If you copy all the above statements into the editor just after the 'cls' statement, you can run the program and see the results in the console window. You have a working program, using only three of Creative's tools.

Notice how you can easily put your own descriptive text into print statements and display it on the screen.

That was easy, but we only processed values that we typed directly into the program.

Most programs will process information entered at run time by the user. This is much more useful.

In our console program, we introduce the **INPUT** statement ..

```
input "Input a number: ",i
input "What is your name? ",a$
```

The program can now be modified to run interactively:

```
openconsole
cls

def i,j:int
def a$:string

input "Input an integer: ",i
input "What is your name? ",a$

j = i + 2

print "Hello ",a$
print
print "The value of j is: ",j

do:until inkey$ <> ""
closeconsole
end
```

Quite useful programs can be built using just those few commands.

Of course, the console window is not a very attractive user interface, and formatting the output nicely is a problem - but it's a start.

This kind of program processes the statements from top to bottom.

Much more interesting things can be done if we bring **Looping** and **Branching** commands into play.

The FOR Loop

The **FOR** loop is used to perform a number of statements a specified number of times.

At it's simplest, it looks like ..

```
FOR  i = 1 to 10

NEXT  i
```

It works like this ...

The Loop Control variable ('i' in this example), starts with the value 1.

Any statements between FOR and NEXT are processed, and the loop then returns control to the FOR statement.

The Loop Control variable is incremented by one, and all statements in the loop are processed again.

This sequence repeats until the Loop variable exceeds the upper value (10 in this case). The loop then terminates, and the program moves on to statements following the loop.

Simple enough, but if you write a FOR loop that will loop for say 1,000 times, you may need to get out of it when some condition occurs.

This can easily be done by setting the Loop Variable equal to the upper limit. In this example, setting i = 10 will stop the loop, because it will think it has finished.

A **FOR** loop is not limited to incrementing by one - you can use an optional **STEP** value.

```
FOR  i = 1 to 10 STEP 2
   print i
NEXT  i
```

This example will print the numbers 1,3,5,7, and 9. It stops at 9 because the loop variable increments to 11, which exceeds the upper limit.

The FOR loop is very flexible, since the start, end and step values can all be variables set within the program.

As well as stepping upwards, a FOR loop can increment downwards as well, as in this example.

```
FOR  i = 10 to 1 STEP -2
   print i
NEXT  i
```

If you run this example, you will get the values 10,8,6,4, and 2. It stops here because the loop variable will increment to zero, which is less than the finishing value 1.

DO - UNTIL Loop

Whereas a FOR - NEXT loop operates between defined 'start' and 'end' control limits, a **DO - UNTIL** loop will keep looping until a specified condition becomes TRUE.

So when a DO loop begins, it will execute all statements between the DO and the UNTIL **at least once**, because the test condition is attached to the UNTIL statement at the end of the loop.

```
openconsole
cls

def i,n:int
n = 8

DO
print n
n = n - 1
UNTIL (n < 4)

do:until inkey$ <> ""
closeconsole
end
```

This example will print 8,7,6,5,4. The loop ends when 'n' is decremented to 3, which the test condition (n < 4) detects.

WHILE - ENDWHILE Loop

The **WHILE** loop differs from the DO loop in that the test condition is at the beginning of the loop attached to the WHILE statement.

This means that the statement block between the WHILE and ENDWHILE statements may not be executed at all.

When a WHILE loop begins, the test condition is examined, and only if it is **TRUE** will the loop begin.

```
openconsole
cls

def i,n:int
n = 8

WHILE (n > 4)
print n
n = n - 1
ENDWHILE

do:until inkey$ <> ""
closeconsole
end
```

This time, the program will print the values 8,7,6,5.

It will not print the value 4, because the test detects that 'n' is NOT greater than 4, and the loop ends. If you required the loop to execute to print 4, you would need to change the test condition to (n >= 4).

So there are three looping methods to choose from:

FOR - NEXT
DO - UNTIL
WHILE - ENDWHILE

Now, we'll have a look at the branching (or conditional) methods.

IF - THEN - ELSE

IF - **THEN** - **ELSE** - **ENDIF** is one of the oldest programming tools.

It means, **IF** some condition is **TRUE**, then perform one block of statements, **ELSE** perform another block of statements. Each **IF** - **THEN** - **ELSE** statement terminates with an **ENDIF** statement.

The test condition can be any expression that results in a **TRUE** or **FALSE** result.

The ELSE block is optional and provides for a group of statements that will be executed if the test condition is **FALSE**.

Depending on the result of the conditional expression, only one of the TRUE or FALSE blocks will be executed - not both.

In a full IF - THEN - ELSE statement, THEN must not be written - as in the following example.

```
openconsole
cls

def n:int
n = 8

IF (n < 5)
   n = n - 7
ELSE
   n = n + 7
ENDIF

print n

do:until inkey$ <> ""
closeconsole
end
```

However, you do have to write **THEN** in the single line version of the statement - but in this case, the **ENDIF** statement is omitted.

```
if (m > 2) & (n < 10) THEN print "Accepted"
```

A **single line version** can only execute one statement.

If the single line **IF** test condition is FALSE, the statement following **THEN** is not executed, and the program continues with the next line.

Some languages provide for multiple conditional blocks, using ELSEIF clauses.

Creative does not have ELSEIF, but the capability is easily implemented using nested IF statements. Here is an example ..

```
openconsole
cls

def nitrate,mass:int
nitrate = 8
mass = 5

if (nitrate >= 5)
    if (mass < 10)
        print "mass ratio is correct"
    else
        print "mass ratio error"
    endif
else
    print "Concentration too low"
endif

do:until inkey$ <> ""
closeconsole
end
```

However, if your application requires multiple conditions to be evaluated, a much better tool is provided by the SELECT - CASE - ENDSELECT method.

SELECT

The **SELECT** statement is a powerful statement that allows you to test for many different conditions, and to execute a block of statements according to whichever condition is TRUE.

You can have as many test cases as you wish. Only the first TRUE condition will be executed, so make sure the conditions are unique.

You can however, use a **DEFAULT** clause for a case where none of the tests are true.

```
openconsole
cls
def value:int

value = 2

SELECT value
   CASE 1
      print "Value is one"
   CASE 2
      print "Value is two"
   CASE 3
      print "Value is three"
   DEFAULT
      print "Value out of range"
ENDSELECT

do:until inkey$ <> ""
closeconsole
end
```

The above example shows a **SELECT** statement used to test for a particular numeric or alphabetic value.

In other situations, you may wish to test for one of a number of logical expressions being **TRUE**. Then a rather special form of the SELECT statement comes into play.

You test for a logical expression being **TRUE**. (remember that **TRUE** equates to **1**)

```
openconsole
cls
def Choice:int
print

LABEL again
input "Enter a Number, (0 to end) : ", Choice
if Choice = 0
   closeconsole
   end
endif

SELECT 1
   CASE (Choice > 10)
      print "number is greater than 10"
   CASE (Choice < 10)
      print "Number is less than 10"
   CASE (Choice = 10)
      print "Number is equal to 10"
ENDSELECT

print
GOTO again
```

Notice the **SELECT 1** statement. Each CASE expression will be tested to see if it is **TRUE**..

Each expression should be unique, because only the first CASE found to be **TRUE** will have it's statement block executed.

You can catch a situation where none of the CASE's are TRUE by using the **DEFAULT** clause as shown in the previous example.

GOTO

The **GOTO** instruction provides a direct jump to a **Labelled** section of code, in the current program or subroutine. You cannot jump **between** subroutines.

This snippet shows the direct form of the **GOTO** command ..

```
GOTO SectionA
   ..
   ..
LABEL SectionA
 ' Processing continues from here
   ..
```

The LABEL description can be any alphanumeric text or number, and in most cases need not be enclosed in quotes. The **GOTO** and LABEL descriptions should be the same.

The **GOTO** instruction also has a dynamic form that's useful in some situations.

In this form, the GOTO specifies a target label using a String variable preceded by **'$'.**

So by setting the string variable to the label description of a chosen label, you can branch to one of a number of destinations in the code.

This example sets variable 'labeltext' to "NextPart". The GOTO will then jump to Label NextPart.

```
def labeltext:string

labeltext = "NextPart"

GOTO $labelText
   ..

LABEL NextPart
```

Here's a small test program demonstrating the dynamic **GOTO** statement.

Of course, this simple example could equally well have been coded using a SELECT statement.

It just demonstrates multiple target branching by resetting the text of a string variable (in this example named 'branch').

```
openconsole
cls
def i:int
def branch:string

i = rnd(2) + 1 :' pick a number 1 or 2

if i = 1 then branch = "Part1"
if i = 2 then branch = "Part2"

GOTO $branch

LABEL Part1
   print "This is Part 1"
   GOTO Ending

LABEL Part2
   print "This is Part 2"

LABEL Ending

do:until inkey$<>""
closeconsole
end
```

The **GOTO** statement is the only command in your programming toolbox, which will branch unconditionally to a specified part of your program. Quite valuable when the need arises.