A Just Do It Moment

We've spent our time so far, working with Console windows, and discussing programming 'theory'.

It's about time we got our feet wet and did some real Window's programming.

To start with, we'll create a simple application with just one window.

Two things before we start. First, this section will be much more interesting if you have Creative Basic installed. If you haven't got it, do consider buying a copy - it only costs £10 ($14.95) and may well become a lifelong interest.

Second thing is, before you start any programming application, create a Folder to save your work in. I've just set one up for this first example - I've called my folder 'cbex1' (creative basic example 1 - how novel ...) .

Now here's the deal - we'll start with a blank Creative editor screen - empty - nix!

To get to that position, click on **File - New - Source File.**

OK, scary isn't it. That's the worst part of a new program - the emptiness.

So we need to type something. Anything come to mind? What I want to do here, is lead you through the thinking and development process of a new program.

Here's a hint - every program needs a **Name** and some identifying 'er 'Comments'.

A Comment! - that's what we want - right at the start of the program.

Here's mine - you can type whatever you like here - just start the line with a single quote.

```
' Window Example 1
'
```

These are comment lines. You can have as many as you like, and type anything in them that will be helpful to anyone looking at the program in future (including yourself).

Now click File - **Save As -** navigate to the new folder you set up, and save the program. Give it a name - I've called mine 'winex1'. (Window example 1).

Now if you'd like to click the pretty green 'run' arrow at the top, you can run the 'program'.

Nothing happens - not surprising really since we haven't entered any instructions yet.

I only mentioned it so that you will feel comfortable clicking the arrow regularly while you are developing your program. Creative is very friendly - you can do no harm.

My editor window is now showing the program name at the top - 'winex1'.
Yours should be showing whatever you named your program as.

Right, next step - the screen is still looking a bit blank ..

How about a Window? It's a Window's program - it needs a Window for the user to admire. So we need to create one - there's a command for that - appropriately named **WINDOW**.

We'll type that next after a blank line. Blank lines do no harm, they're just for readability.

Creating a Window

The program now has a **WINDOW** command.

However, the program would have to be psychic to be able to display a window with just the command 'WINDOW' . It needs a bit more information.

What **style** of window, normal?, a Frame?, no Caption?

What **size** do you want - width and height. **Where** would you like to place it on the screen?

The information required is not excessive, and since we only need a Normal window for this application, it's pretty straightforward. But it has to be typed in the right order.

Can you remember what that is? No, neither can I.

This is where the Creative Help system comes in. Click on **Help - Users Guide**.

Click on the plus**(+)** sign next to Windows in the left hand index. Click on **Creating Windows**.

(Don't worry, I won't be spelling out this amount of detail for everything we do - just the first time, so you can see how things work).

Now you can see a full description of how to create a Window.

It's pretty exhaustive, so we'll just look at the syntax template.

```
WINDOW variable, left, top, width, height, flags, parent, title, handler
```

Ah, that looks useful. Now we can fill in some of the numbers .

'Variable' - what's that? In the Help system under 'variables' you'll find that **window** is a variable type, and as such every window needs a unique name.

I'll call mine 'w'. Since this is only a single window program, I feel comfortable with that.

You can call yours 'win', 'win1', 'MyFirstWindow' .. even 'rx105' (but I wouldn't recommend anything that's not meaningful).

Now 'Left' and Top'. What does that mean?

Every Window, control, image, and everything else that appears on screen, has to declare it's position. The units are 'pixels'. How you lay everything out depends on your screen resolution.

I'm running at 1024x768 pixels, because at any higher resoluiton, I have trouble reading the screen. You may be at 1280x1024 - no matter.

If we set both Left and Top values at **'0'**, the window will appear at the top left of the screen.

'Width' and 'Height' is pretty obvious. Shall we try Width 200 and Height 100?

'Flags' - we don't know what they are yet - so type a zero.

'Parent' - we only have one window, so this IS the parent - so type another zero (ie.This window has no parent).

'Title' - This text string (enclosed in quotes) is what will appear in the window's Title Bar, so I shall use "Example 1".

'Handler' - this is an important bit as we shall see. What it requires is the name of the subroutine we will be using to handle all the **Windows messages** for this window.

Nothing will work without this, so shall we call the subroutine 'msghandler'?
You can call it anything you like - but keep it meaningful.

So now the WINDOW statement looks like this:

```
WINDOW w, 0, 0, 200, 100, 0, 0, "Example 1", msghandler
```

That looks fine, so try to run the program.

No luck heh? An error message - 'Undeclared variable on line 5 - 'w' not found'.

Creative is being helpful - all variables need to have their type declared. It is telling us that variable 'w' has not been declared. (If you used another name for your window your error message will show your window name.

So we need a **DEF**(ine) statement in which to declare our window variable.
Type it above your WINDOW statement ..

```
def w:window
WINDOW w, 0, 0, 200, 100, 0, 0, "Example 1", msghandler
```

If you try to run the program, you'll get another error message of the same kind, saying that variable 'msghandler' has not been declared.

This isn't quite the same as before - what Creative is trying to say is that we haven't supplied a subroutine named 'msghandler'. It can't run without a message handling routine, so we need to type in a subroutine to deal with the window's messages.

A Subroutine in Creative consists minimally of a **SUB** 'name' statement and a **RETURN** statement.

That would be a minimal do-nothing subroutine. So let's type it in at the bottom:

```
sub msghandler

return
```

Try another 'run' and - oh no! ..another error message. 'Return without GOSUB'

Now this one is a bit cryptic, and you need to read between the lines ...

We have entered a valid subroutine, but Creative is complaining about it.

You might not see what the problem is at this point - but what is wrong is that the main program is not yet complete. A main program has to have an **END** statement to show where that program ends, and any following subroutines begin.

I'll give a bit of help here 'cos the program's nearly working, and only needs a couple more things that you probably won't think of.

Type these statements after the **WINDOW** statement, leaving a few blank lines for good measure:

```
run = 1
waituntil run = 0
closewindow w
end
```

You have seen these before in the 'Welcome' section of these notes. But briefly, the **END** statement is needed as mentioned above, and the **'closewindow'** statement is obviously there to close the window when we exit the program.

You need to close windows, stop timers, and delete images when you close a program, so that their resources are released back to the Windows system.

The **run = 1** and **WAITUNTIL** statements are needed to keep the window on screen until you close it. Otherwise it would just flash up and disappear immediately.

So now we have an almost complete program. It should look something like this:

```
' Window Example 1
'
def w:window
window w,0,0,200,100,0,0,"Example 1",msghandler


run = 1
waituntil run = 0
closewindow w
end

sub msghandler
return
```

Now I know I've been encouraging you to click the green 'run' button at every opportunity, but at this point I suggest you don't.

Not because the result would be catastrophic. At this point the window would appear - the only snag being, **you wouldn't be able to close it**.

The usual 'close window' **(X)** will be at the top right of the window, but it won't do anything.

When you click the close button on a window, a message is sent to the window's message handling subroutine, whose job it is to close the window.

It won't work in our program, because our message handling subroutine is **empty** - it's currently a do-nothing routine.

So we need a few more statements. Don't worry, we're almost there.

One way to enter some useful message handling statements, would be to copy and paste them from an existing program.

But since we're working from a clean slate, here is a typical set of statements :

```
select @CLASS
    case @IDCLOSEWINDOW
' closing the window
        run = 0
    case @IDCHAR
' 'ESC'(ape) key pressed will close the program ...
        key = @CODE
        if key = 27 then run = 0
    case @IDCONTROL
        select @CONTROLID
' Exit button clicked ...
            case 1
                run = 0
        endselect
endselect
```

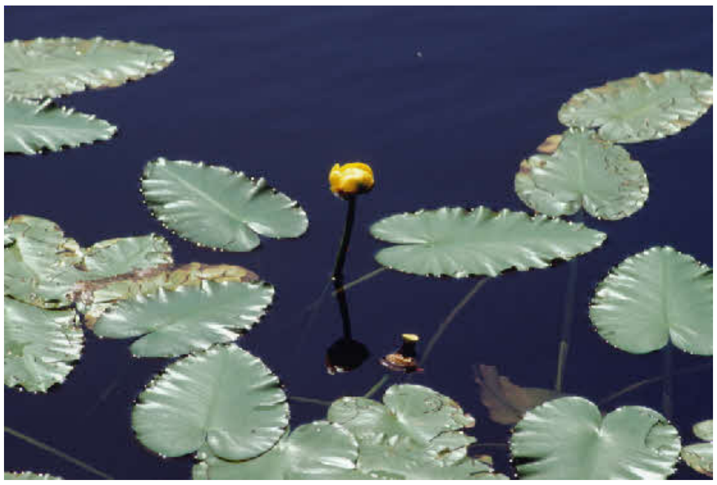
Copy and paste all of these statements between the '**sub msghandler'** statement and the **'return'** statement.

It looks a bit difficult, but we'll go through it - it's straightforward.

Message Handling Subroutine

When your window opens, Windows begins to send many messages to your program, and these are processed by the message handling subroutine. Messages are discussed fully in the Creative Help system under **Windows - Messages and message loops.**

There are many different types of message, and here we are looking for a window being closed , a key being pressed, or messages that a control has been clicked.



A **SELECT** statement is used to decide which type of message we're dealing with. This is an ideal application for a **SELECT** statement.

System variable **@CLASS** will be holding the information identifying the type of message.

The first case we deal with is when the window is closed by clicking the **(X)** button.

The **select case @IDCLOSEWINDOW** condition is then TRUE, and we respond by setting 'run = 0'.

As soon as this occurs, the **WAITUNTIL run = 0** statement in the main program, will detect that the variable 'run' is now zero, and will close the program.

Similarly, the **select case @IDCHAR** condition, detects any keypress, and if it is the **ESC**(ape) key, (ASCII code 27) this also will set run = 0 and close the program.

Finally, the **case @IDCONTROL** and **select @CONTROLID** statements, will respectively detect that an incoming message is from a control - and which control it is from.

We don’t have a control yet - we need to rectify that.

What we require, is an 'Exit' **Button control** that will close the program when it is clicked.

If it's **control identity is number 1**, the message handling subroutine will again set run = 0 and the program will close.

So we would then have three ways of closing the window - the **(X)** icon, the **ESC**(ape) key, and an Exit Button.

A Button Control

Here is a Button control statement based on the Creative user guide.

```
control w, "B, Exit, 0, 10, 60, 30, 0, 1"
```

You can copy and paste this statement into the program after the WINDOW statement.

A brief description of what this does ...

The **CONTROL** statement identifies this as a statement to create a window control.

The **type** of control (because there are many types) is specified by the letter **'B'** as a **Button** control.

The **'w'** identifies which window the button will be part of. (Many programs will have several windows). My window's name is 'w' - yours may be different.

'Exit' is the label that the button will display to identify it's purpose.

The first two numbers, **(0, 10)** (in pixels), are the **'x, y'** co-ordinates of the button in the window. 'x' is measured from the Left, and **'y'** from the top of the window.

The next pair of numbers **(60, 30)** are the **Width** and **Height** of the button.

The next number **(0)** is where any special button **'style flags'** would be set.
For our first program, we'll leave it set to (0) to give the default Normal button style.

The last number **(1)** is important - it is the control's identifier **(ID)**.

If you recall, the message handling subroutine will wish to know this **ID number** in order to close the program.

You can give controls any reasonable identifying number, not necessarily in sequence, as long as they are unique within any given window.

If you were to assign the ID **(2)** to this Exit button, the message handling subroutine will receive a message when it's clicked, but the logic will decide it's from **control 2**, not **control 1,** and will not close the program. There is no logic to deal with messages from control 2.

So the Exit button has to have a control **ID** of **1**.

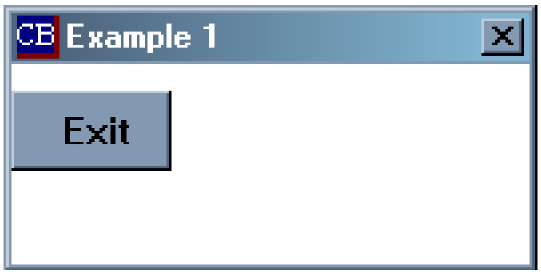
The complete program should now look like:

```
' Window Example 1
'
def w:window
window w,0,0,200,100,0,0,"Example 1",msghandler
control w, "B, Exit, 0, 10, 60, 30, 0, 1"

run = 1
waituntil run = 0
closewindow w
end

sub msghandler
select @CLASS
   case @IDCLOSEWINDOW
' closing the window
      run = 0
   case @IDCHAR
' 'ESC'(ape) key pressed will close the program ...
      key = @CODE
      if key = 27 then run = 0
   case @IDCONTROL
      select @CONTROLID
' Exit button clicked ...
         case 1
            run = 0
      endselect
endselect
return
```

If you click the green run arrow, the window should appear, and should look like this:



That's a pretty poor specimen of a window to impress any user with.   There are several things wrong with it.

It's too small to be useful. It's insipid, and the 'Exit' button is placed in a poor position.

Never mind - at least it's working, so from here it's easy to put things right.

Now that the window is working , this is where you discover how nice it is to be working in an Interpreter environment.

Changing how things look, their size and position is easy.

Looking Good

Our current window size is 200 x100 pixels - so clearly it needs to be larger than that.

Locate the WINDOW command and change the Width and Height to values that you prefer.   I shall use 800 x 600 ..

Go straight to the pretty green 'run' arrow and see how it looks.

That's better, but my idea is to show you how easy it is to draw on the window. To see that to best advantage, I think a darker colour be good.

Easily done, we want to set the window's colour - so we use the command:

```
setwindowcolor w, rgb(0,0,80)
```

(American spelling - but we get used to that ...)

Copy and paste to just after the **WINDOW** statement.

It's obvious what this statement is doing, but notice we have to specify which window we want to colour (there may be several). My window is named **'w'** - you might have named yours differently.

The colouring function **RGB**( ) is often used. It works by setting the intensity of colour of the RED, GREEN and BLUE channels, to a value between 0 (none) to 255 (colour fully on).

In this case I've chosen no Red or Green, and Blue only partly on at a value 80.
Try it , you should now see a dark blue window. If you prefer a different colour, change the RGB values to your preference.

I'll just mention that if you would prefer a shade of Grey, just set the RGB values all the same.

So **RGB(0,0,0) is full Black, RGB(255,255,255) is full White. RGB(50,50,50)** is a darkish Grey.

Now let us centre the window on the screen. One more command will do that:

```
centerwindow w
```

Again, watch out for the American spelling.

That was easy .. now to move the Button control nearer the bottom of the window.

The current x, y co-ordinates are (0,10), 'x' values are from the left of the window, 'y' values are from the top.

It will be good to position the button in the centre of the window.

The button is 60 pixels wide, and the window is 800 pixels wide. So if we subtract 60 from 800 and halve the difference, that should place the button exactly in the centre.

So we need to change the Button 'x' value from 0 to the calculated value (800 - 60) / 2

```
control w, "B, Exit, (800 - 60)/2, 10, 60, 30, 0, 1"
```

If you have chosen a different window width than 800, use your value instead.

Run the program again, and the Button should now be in the centre of the window.

It's too near the top, so we need to alter the 'y' co-ordinate to drop it lower down.

I'm going to guess, and alter the current setting of 10 pixels down, to 500, and see how that looks.

```
control w, "B, Exit, (800 - 60)/2, 500, 60, 30, 0, 1"
```

Run the program again, and yes, that looks fine.

We now have a decent looking window with an Exit button and a functional message handling subroutine.

At this point, I'd suggest you save the program as a Window Skeleton program that you can bring up again for future work, without having to think it all up again.

I have a folder I call 'cbwork' just for saving useful bits of code like this. Here is the full program as it stands:

```
' Window Example 1
'
def w:window
window w,0,0,800,600,0,0,"Example 1",msghandler
setwindowcolor w, rgb(0,0,80)
centerwindow w

control w, "B, Exit, (800 - 60)/2, 500, 60, 30, 0, 1"

run = 1
waituntil run = 0
closewindow w
end

sub msghandler
select @CLASS
   case @IDCLOSEWINDOW
' closing the window
      run = 0
   case @IDCHAR
' 'ESC'(ape) key pressed will close the program ...
      key = @CODE
      if key = 27 then run = 0
   case @IDCONTROL
      select @CONTROLID
' Exit button clicked ...
         case 1
            run = 0
      endselect
endselect
return
```

A Bit of Graphics

Now we have an empty window, we really ought to put something in it.

Here's a little addition to draw some rectangles to demonstrate how easy it is to do a bit of graphics in Creative.

All we need, is a **Loop** and the **RECT** command.

```
FOR i = 1 to 30
   RECT w, 350,200,rnd(70),rnd(50),rgb(rnd(255),rnd(255),rnd(255))
NEXT i
```

Copy and paste the three statements between the CONTROL and run = 1 statements.

If you run it, you will see some coloured boxes, but they are all on top of each other. We've got to do better than that.

You will see that there are several **RND**( ) function calls. The first pair are to randomly set the width and height of the rectangles.

The **RND**( ) calls in the **RGB**( ) function are to randomly set the rectangle colours.

The first pair of numbers (350,200) in the **RECT** statement set the x,y co-ordinates of the rectangles. So they are all drawn with the top left corners starting at the same place on the screen. Obviously, that's not good. so we need to change the x,y values to place them randomly.

Now the **RECT** statement is getting a bit long, and it's going to get longer. So this is probably a good time to introduce a few auxiliary statements.

This means we could continue typing a long line, but we choose instead to split it into more manageable parts. It will also make it easier to describe what's happening.

```
for i = 1 to 30
   xpos = 350
   ypos = 200
   width = rnd(70)
   height = rnd(50)
   colour = rgb(rnd(255),rnd(255),rnd(255))
   RECT w, xpos,ypos,width,height,colour
next i
```

I hope you can see what I'm doing here - just re-arranging the parts of the **RECT** statement.

Copy and paste this whole **FOR - NEXT** block, to replace the three existing statements.

Because we have introduced a few new variables to hold the separate parts, we also need to place a new **DEF**(ine) statement at the top of the program:

DEF xpos, ypos, width, height, colour : int

Place this **DEF** statement just after the existing one.

The **'xpos'** statement controls where the boxes will be placed relative the the left-hand side of the window.

At the moment, it is set to 350 pixels from the left.

We can randomise this by using the RND( ) function to give 'x' values between 0 and 750 pixels.

xpos = RND(750)

The 'ypos' statement controls the distance from the top of the window. Currently set at a fixed value of 200 pixels, we can randomise this to give values between 0 and 500 pixels.

ypos = RND(500)

Rectangles of various sizes would be nice, so we randomise the 'width' and 'height' values as well.

width = RND(70) + 20
height = RND(50) + 30

The rectangle width will now vary between 20 and 90 pixels, and the height between 30 and 80 pixels.

This is now the final version of the program:

```
' Window Example 1
'
def w:window
def xpos,ypos,width,height,colour : int

window w,0,0,800,600,0,0,"Example 1",msghandler
setwindowcolor w, rgb(0,0,80)
centerwindow w

control w, "B, Exit, (800 - 60)/2, 500, 60, 30, 0, 1"

for i = 1 to 30
    xpos = rnd(750)
    ypos =  rnd(500)
    width = rnd(70) + 20
    height = rnd(50) + 30
    colour = rgb(rnd(255),rnd(255),rnd(255))
    RECT w, xpos, ypos, width ,height, colour
next i

run = 1
waituntil run = 0
closewindow w
end

sub msghandler
select @CLASS
    case @IDCLOSEWINDOW
' closing the window
        run = 0
    case @IDCHAR
' 'ESC'(ape) key pressed will close the program ...
        key = @CODE
        if key = 27 then run = 0
    case @IDCONTROL
        select @CONTROLID
' Exit button clicked ...
            case 1
                run = 0
        endselect
endselect
return
```

You can change how many rectangles are drawn by altering the **FOR** loop upper control value.

So there we are, a complete program developed from scratch.

If you would prefer ellipses instead of rectangles, just substitute the **ELLIPSE** command for the **RECT** command, and presto there they are.

What? .. you want the rectangles to float majestically around the screen at various speeds? - and fade in and out as they go?

I think that will have to wait for a more in depth look at the graphics capabilities of Creative Basic.