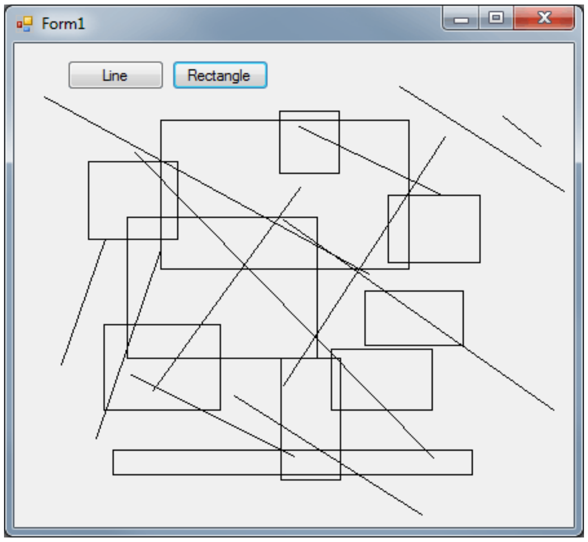
Visual Basic Events

Events are what make a program do something. In this lecture we take a closer look at how events are used together to create a simple drawing application.



By now you should know that all Visual Basic programs are made up of objects, their properties and the events they respond to.

“Properties” is a difficult concept for the simple reason that to produce even simple examples we have to use some ideas that aren't completely introduced at this early stage.

Step One - The event model

A program is a list of instructions that the computer will carry out when you tell it to. Normally the machine will read and obey your instructions starting from the first and working its way towards the end of the list. In simple computer languages the order that instructions are carried out is always 100% determined by the order you write them in and this is mostly the rule in Visual Basic - but it is a little more complicated than this simple picture.

In Visual Basic you can never be sure what order your instructions will be obeyed in because they are triggered by events. Most, but not all, events are generated as the result of something the user does - click a mouse button, move the mouse, select from a menu, *etc*. Windows detects the event and sends a message to the program that is active notifying them of the event. The program has the option of either doing something, *i.e.* responding to the event, or ignoring it.

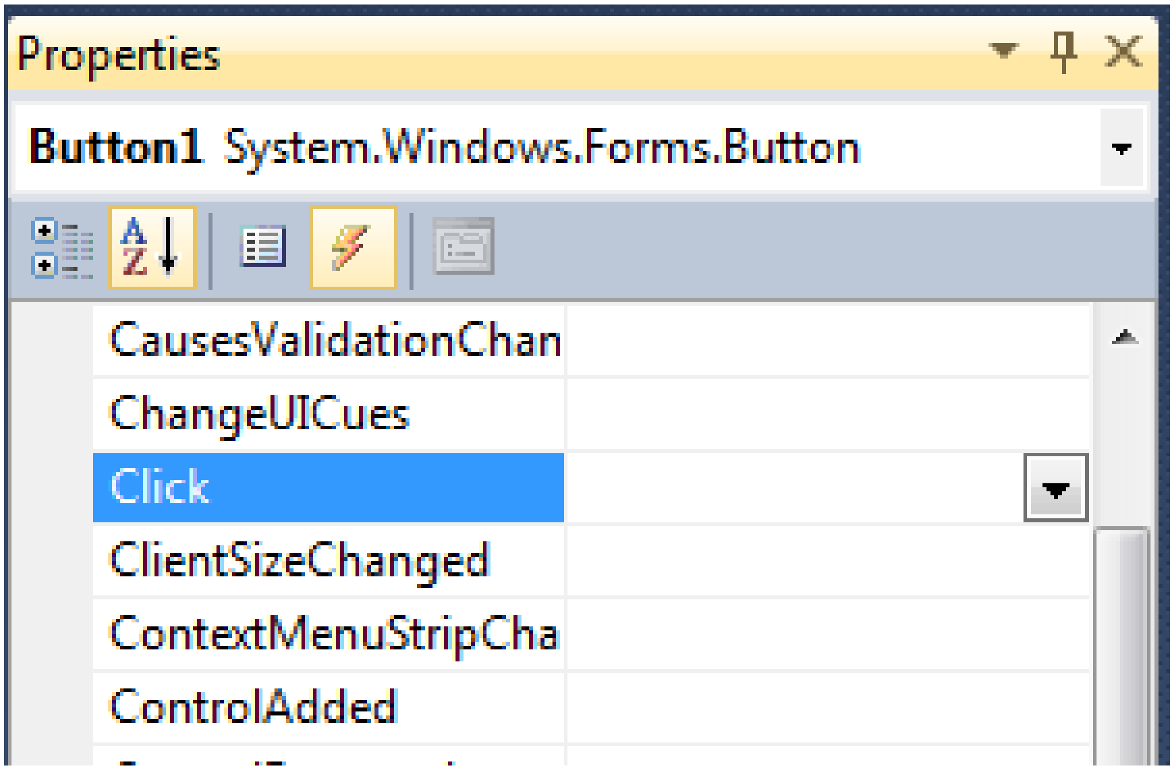
When Visual Basic is notified of an event by Windows, it identifies which object the event concerns and then calls the appropriate event handler. An event handler is just a special sort of method that is obeyed in response to an event occurring. Each object has an event handler for every event that you can think of.

At this point programming in Visual Basic might sound impossibly complicated. You must learn the names of all the events and write event handlers for every single one for every single object! Only of course you don't have to!

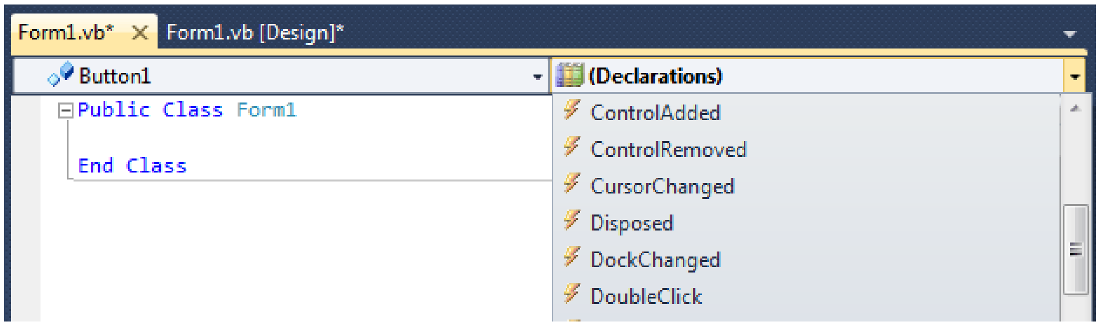
Every object comes complete with a set of "dummy" event handlers. So, by default when an event happens the appropriate object's event handler is called and, by default, nothing happens in response. In other words, you only need to worry about events that you want to worry about! In the same way while there are lots and lots of different events in practice you generally only write event handlers for a small number.

Step Two - What events?

Of course, the big problem is that you must know what sort of events are available for any particular object in order to know what you can make it do. **The simplest way to see what events are relevant to an object while using the designer is to select the object - a button say - and then look at the Properties Window. At the top of the Properties Window you will see an icon that looks like a lightening flash. If you click on this you will see a list of events that the object can handle.**



If you are in the code editor then you can see the events that an object supports by selecting it in the drop-down list at the top of the window.



```
Public Class Form1

End Class
```

Most users of Visual Basic find out about the Click and Double click events almost as soon as they start programming - but what about all the others that are listed. What for example, is a DragDrop event and what is a MouseDown event? The range of events that you see in the list also depends on the type of object you are working with. This means that there is always the possibility of encountering an event you have never seen before.

If you are of a nervous disposition you can quickly decide that Visual Basic has rather a lot of detail hidden just below the surface waiting to jump out and get you! This is an accurate reflection of the situation but far from being a problem it is what makes Visual Basic easy to learn in small steps.

Step Three - The mouse events

You can group events into different categories that makes them easier to understand. Perhaps the most obvious group of events is the mouse group.

The Click and DoubleClick mouse events are obvious

Name	Event
Click	Called in response to a single click
DoubleClick	Called in response to a double click

Often the Click and DoubleClickevents are all you need to process but sometimes you do need to work with the mouse at a lower level.

The Mouse - Down/Up/Move events are called every time the user presses or releases any button or just moves the mouse.

Name	Event
MouseDown	Called when any mouse button is pressed down
MouseUp	Called when any mouse button is released
MouseMove	Called every time the mouse moves

These three events are easy enough to understand but which object gets to handle the event?

The answer is that the object that the mouse is over handles the event but there is a subtle point. The object that handles the first MouseDown command also handles all subsequent Mouse events until the final MouseUp event.

These five events - Click, DoubleClick, MouseUp, MouseDown and MouseMove - allow you to define how any object behaves in response to the mouse.

The only extras are the Drag events which are designed to make dragging objects easier, but we will meet these a little later.

Step Four - Parameters

When an event occurs the appropriate event handler is called. However there has to be some way for the details of the event to be provided to the event handler. Every event handler is provided with two special objects. The first is usually called the sender and it is the object that generated the event. The second is an EventArg object which carries information about the event.

This is the first time we have encountered objects that don't correspond to controls on a form like a button but the idea is exactly the same. If it helps just think of them as invisible objects. Even though they are invisible they still have properties and methods and sometimes they even have events associated with them.

Of the two objects provided to an event handler the one that we use most often is the EventArg object - it is called EventArg because *"arguments" is another name for parameters.* The key idea is that the EventArg object has properties that tell you about the event. As different events need to convey different types of information there are a range of slightly different EventArg objects each with its own particular set of properties.

All of this will make more sense after an example and after you have learned a little more Visual Basic. This idea of using similar types of object with different sets of properties is a very common idea and central to the practice of object-oriented programming.

For example, if you add a MouseDown event handler, simply double click on the event in the properties window, you will see that it reads:

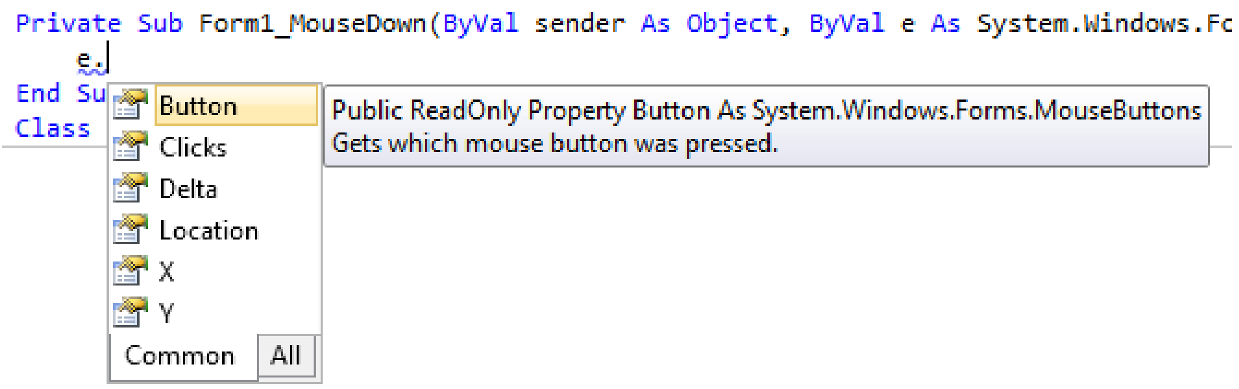
```
Private Sub Form1_MouseDown(
     ByVal sender As Object,
     ByVal e As
       System.Windows.Forms.MouseEventArgs)
             Handles Me.MouseDown
End Sub
```

which on first sight does look like a lot to take in. The first line defines the event handler and the two object we discussed earlier are called sender and e (for eventargs). Let's ignore sender for the moment and concentrate on e as it is where the interesting properties are.

You will also notice if you read the line carefully that “e” is a MouseEventArgs object which is a special type of EventArgs object used to convey information about mouse events in particular.

A MouseEventArgs object has three important properties Button, X and Y. The Button property tells you which button was pressed down and X and Y give you the location where it happened.

In general you have to look up what properties an EventArgs object supports in the manual. However you can also often guess what properties are for if you start to type an instruction involving e because Visual Studio lists all of the possibilities. That is if you type e. (that e followed by a dot) then Visual Studio will list all of the properties of e and you can select one.



```
Private Sub Form1_MouseDown(ByVal sender As Object, ByVal e As System.Windows.Fc
    e.
End Su
Class
```

Public ReadOnly Property Button As System.Windows.Forms.MouseButtons
Gets which mouse button was pressed.

Step Five - Form Events

At first all of your attention is focused on the objects that you place on the Visual Basic form - but the form itself is an object and has event handlers that you need to know about. In particular, one form event - Load - is vital to many programs. This event occurs when a form is loaded and before the form is visible on the screen. A few moments thought soon indicates what you can use this event for - initialization. Anything that you need to set up before a form is used can be included in the form's Load event routine.

One particular form - the startup form - is automatically loaded when you run a Visual Basic program. Normally this defaults to Form1, i.e. the first form you create, but you can change this using the command Project,Properties and then selecting the startup form in the dialog box that appears.

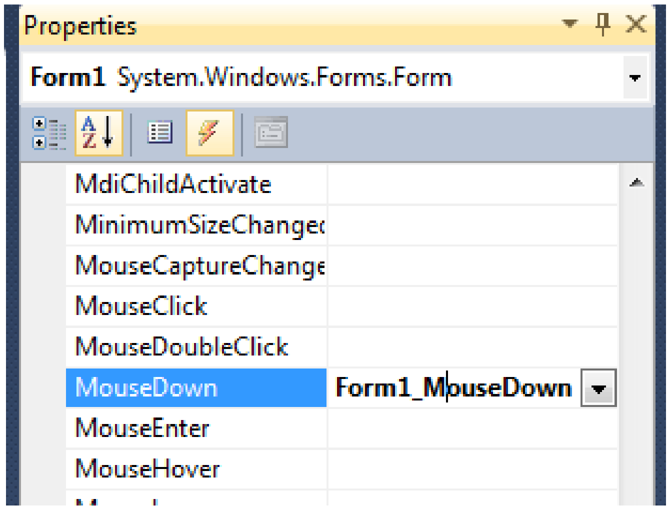
A Drawing Program

This example shows how to take control of the mouse via the lower level mouse events.

The task is to write a program that allows a user to draw on the screen. If you have used programs such as Windows Paintbrush or Corel Draw! then you will know that this is a big project. However, you might be surprised just how quickly you can create something that works using Visual Basic.

Start Visual Basic Express (or Visual Studio) running and create a new Windows Forms application. This will, by default, have a Form called Form1.

Select the form by clicking on it, view the Properties Window, click the "lightening" button to display the events associated with the Form. Scroll down until you find the MouseDown event. If you double-click on this, you will be transferred to the code editor where a suitable event handler will have been created for you:



The default event handler doesn't contain any instructions - its just a "stub" waiting for you to fill in what should happen:

```
      Private Sub Form1_MouseDown(
        ByVal sender As System.Object,
        ByVal e As
          System.Windows.Forms.MouseEventArgs)
                       Handles MyBase.MouseDown
       End Sub
```

The first thing we can do is to simply draw a line from the top left hand corner of the form - which is location 0,0 to the location where the mouse button was pressed. This we can do by first getting the Graphics object corresponding to Form1. Some objects in Visual Basic can be drawn on and this is done using their Graphics object - more about this later.

The form's graphics object can be otained using:

```
      Dim g As Graphics = CreateGraphics()
```

The Graphics object has lots of methods that let you draw on the Form. For example there is a DrawLine method that will draw a line in a particular color from one point to another.

That is:

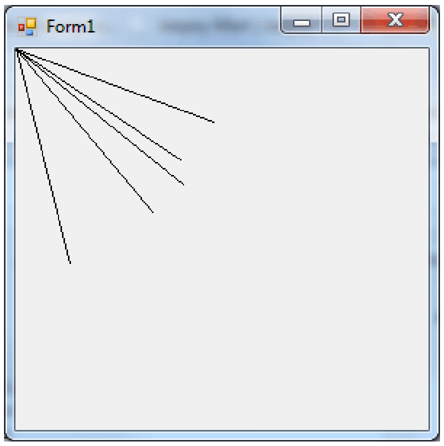
```
      g.DrawLine(Pens.Black, 0, 0, e.X, e.Y)
```

will draw a line in black from the point 0,0 to the point where the mouse button was pressed i.e. where the MouseDown event occurred. Notice that way that we use the EventArgs object e to get the position of the mouse.

Putting the two instructions together, the complete event handler is:

```
      Private Sub Form1_MouseDown(
         ByVal sender As System.Object,
         ByVal e As
           System.Windows.Forms.MouseEventArgs)
         Handles MyBase.MouseDown
       Dim g As Graphics = CreateGraphics()
       g.DrawLine(Pens.Black, 0, 0, e.X, e.Y)
      End Sub
```

if you now run the program you will discover that every time you press the mouse button down a line is drawn from the top left-hand corner to the location of the mouse.



Drawing lines from 0,0 to the current position of the mouse is interesting but we need to be able to draw from any location to any location. To do this we have to use a slightly clever method. When the mouse button is pressed down we simply need to remember the location and then when the mouse button is released at a second location we can draw a line from the first location to the second.

This isn't difficult but it does involve some new ideas that. The main idea is that we can add properties to the Form object that can be used to store the mouse's position when the mouse button is pressed. To do this we need to redefine the MouseDown event handler to read:

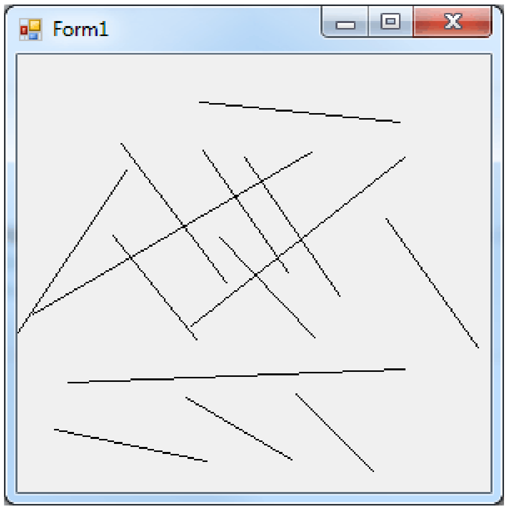
```
Dim x, y As Integer
Private Sub Form1_MouseDown(
   ByVal sender As System.Object,
   ByVal e As
     System.Windows.Forms.MouseEventArgs)
   Handles MyBase.MouseDown
 x = e.X
 y = e.Y
End Sub
```

The first line starting Dim creates two new properties for the form called x and y which the event handler uses to store the co-ordinates of the first location.

Next we need to add a MouseUp event handler - use the Properties Window in the designer again. Now all we need to do is use the instructions we used earlier to get the Graphics object and draw the line from the first to the second location:

```
Private Sub Form1_MouseUp(
  ByVal sender As Object,
  ByVal e As
    System.Windows.Forms.MouseEventArgs)
  Handles Me.MouseUp
 Dim g As Graphics = CreateGraphics()
 g.DrawLine(Pens.Black, x, y, e.X, e.Y)
End Sub
```

Now if you run the program you will discover that you can draw lines anywhere on the Form by holding down the mouse button at the starting location, moving the mouse to the finishing location and releasing the mouse button. The lines then appears as if by magic.



Simple this program may be but we need to go a little further. We need to allow the user to select the type of shape that is drawn - for example, a line or a rectange. The simplest way to do this is to add two buttons labelled Line and Box to the form.

Obviously, the shape drawn depends on the button that the user last clicked on but this means that we have to "remember" this fact while the program is running. Once again, we add a property to the Form1 object to record the shape to be drawn. In this case the new property is called Shape can be set to 1 to indicate that a line has to be drawn and to 2 to indicate that a box is required.

To set Shape all we need is to define the click event handlers for each button as:

```
Dim Shape As Integer
Private Sub Button1_Click(
  ByVal sender As System.Object,
  ByVal e As System.EventArgs)
  Handles Button1.Click
 Shape = 1
End Sub

Private Sub Button2_Click(
  ByVal sender As System.Object,
  ByVal e As System.EventArgs)
  Handles Button2.Click
 Shape = 2
End Sub
```

With this addition we can now rewrite the MouseDown and MouseUp routines to take notice of the value in Shape. In fact it is only MouseDown that has to be changed because it has to do the actual drawing.

MouseDown is the most complicated as it has to do the actual drawing. The Graphics object has a method to draw a Rectangle called DrawRectangle. This works in much the same way as DrawLine but now you specify the color, top left hand corner and the width and hieght of the rectangle. This means we have to use the two locations to work out the width and height.

To select which command is used we need to test the value in Shape using an IF conditional command. As with the graphics commands, the IF command will be explained in detail later but all it does is to test the condition that follows it and carries out the command if the condition is true.

For example:

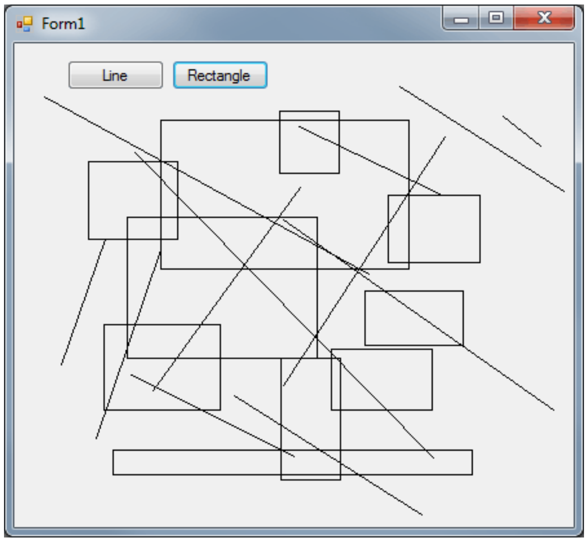
```
If Shape = 1 Then
 g.DrawLine(Pens.Black, x, y, e.X, e.Y)
End If
```

draws a line only if Shape is equal to 1.

If you follow this then you should have no trouble following the logic of the complete MouseDown routine:

```
Private Sub Form1_MouseUp(
  ByVal sender As Object,
  ByVal e As
   System.Windows.Forms.MouseEventArgs)
  Handles Me.MouseUp
 Dim g As Graphics = CreateGraphics()
 If Shape = 1 Then
  g.DrawLine(Pens.Black, x, y, e.X, e.Y)
 End If
 If Shape = 2 Then
  g.DrawRectangle(Pens.Black, x, y,
                      e.X - x, e.Y - y)
 End If
End Sub
```

If you run the program as it is you will discover that as long as you click one of the buttons first then you can draw a line or a rectangle - as long as you also click on the top left hand corner and then "drag" the mouse down to the bottom right hand corner.



There are a number of problems with this early attempt at a drawing program. The only one that is easy to fix is that when the program is first run the Shape variable doesn't have a value until the user clicks on one of the buttons. To give Shape a default value we can define the Load event handler for Form1 as:

```
Private Sub Form1_Load(
  ByVal sender As Object,
  ByVal e As System.EventArgs)
  Handles Me.Load
 Shape = 1
End Sub
```

which sets Shape to 1 when the form is loaded.

The other two main problems with the program are that if you try to draw a rectangle starting from the bottom right hand corner nothing appears. See if you can work out why - hint: it is to do with the calculated width and height of the rectangle. The second problem is that the drawing vanishes if another window overlaps it or if the application is minimised and restored. What we need is called "persistent" and while this isn't difficult it takes us into some subtle areas of implementation.

That's all there is to the program. Now you have seen how mouse events can be used to discover where on the screen the user is pointing and how to use the object included in event routines. In addition we have seen how global variables work and looked ahead to see how graphics commands and the IF statement work.

The key thing to take away from this demonstration is that a typical Visual Basic application is essentially a set of event handlers that are called in response to things happening, usually in the outside world. Events are what make a program do things.

https://www.i-programmer.info/ebooks/master-visual-basic/1828-mastering-vb-events.html