

19.1 Introduction:

In conventional programming, the sequence of operations for an application is determined by a central controlling program (e.g., a main procedure). In **event-driven** programming, the sequence of operations for an application is determined by the user's interaction with the application's interface (forms, menus, buttons, etc.).

The code for an event-driven application remains in the background until certain events happen, for example:

- when a value in a field is modified, a small data verification program is executed;
- when the user presses a button to indicate that the order entry is complete, the inventory update procedure is executed.
- when the user switches to a different record, the properties of certain controls on a form are set based on values in the new record.

Event-driven programming, graphical user interfaces (GUIs), and object-orientation are all related since forms (like those created in [Lesson 13](#)) and the graphical interface objects

on the forms (like those created in [Lesson 15](#)) serve as the skeleton for the entire application.

19.1.1 Listening and handling events

To create an event-driven application, the programmer creates small programs—called **event handlers**—and associates them with specific events raised by specific objects. In ACCESS, events are typically raised by objects on a form (or by the form itself) and handled by procedures defined within the **form module**.



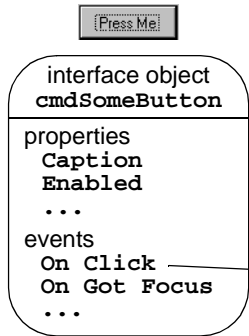
ACCESS has different types of modules including stand-alone and form modules. In [Lesson 18](#), you created a stand-alone module called **basUtilities**. For event-driven programming, you will use modules embedded within forms. The distinction is important because when you go looking for your event handlers, you will not find them in the **Modules** pane of the database window. Instead, the VISUAL BASIC code you write to handle events is saved with the form.

The relationship between interface objects, the form on which the interface objects reside, and procedures is shown in [Figure 19.1](#).

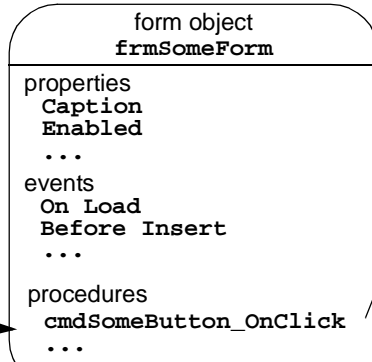


FIGURE 19.1: In ACCESS, all events for objects on the form are handled by the form object.

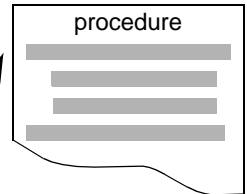
1 The user presses the button. The button object raises its **On Click** event.



frmSomeForm



2 The form object "hears" the button's **On Click** event and searches for an event handler.



3 If an event handler for the object's event is found, it is executed.

19.1.2 Creating event handlers

The nice thing about event-driven programming is that the event handlers—that is, the procedures that are executed when the event is raised—can be very simple. In fact, you have already created an event-driven program without writing a single line of code: When the button you created in [Section 17.3.5](#) is pressed, an action query is executed to update inventory

levels. Event-driven programming can be that simple.

In practice, however, you will seldom rely on action queries alone to implement your event handlers. Since an action query in ACCESS can only perform one type of action, and since you typically have a number of actions that need to be performed, macros or VISUAL BASIC procedures are far more useful. But as you will see in this lesson, it is possible to use a combination of



action queries and simple VISUAL BASIC statements to accomplish most of what you need to do.

19.1.3 The event-driven design cycle

To create an event-driven procedure, you need to answer two questions:

1. What has to happen?
2. When should it happen?

Once you have answered the first question (“what?”), you can create a procedure to execute the necessary steps. Once you know the answer to the second question (“when?”), you can associate the procedure with the correct event of the correct object.



Selecting the correct object and the correct event are often the most difficult part of creating an event-driven application. It is best to think about this carefully before you get too caught up in implementing the procedure.

19.1.4 VBA versus macros

There are two ways to create procedures within ACCESS:

- VISUAL BASIC FOR APPLICATIONS (VBA) code, or
- the proprietary macro language included with ACCESS.

The primary difference between VBA programming and macro programming is that the macro language consists of a handful of simple commands to accomplish many common tasks. In contrast, VBA is a general purpose programming language that is considerably harder to use, but far more flexible. Indeed, if you want your application to do something, chances are you can do it using VBA. Another nice thing about learning VBA is that most MICROSOFT applications (e.g., EXCEL, OUTLOOK) and some non-MICROSOFT products support VBA as a scripting or macro language.



A macro language is simply a language that consists of high-level (or “macro”) commands. The term causes some confusion because macros in many older applications and some new ones (including MICROSOFT EXCEL) are programmed by recording keystrokes. As such, a common mistake it to consider the terms “macro” and “keystroke recorder” to be synonymous. However, the macro language in ACCESS provides no keystroke recording functionality.

19.1.5 Event-driven programming versus triggers

A **trigger** is usually defined as a procedure that is executed when a specific event in a table



occurs (such as an insert, delete, or update). Triggers are useful for enforcing business rules throughout a database and applications based on the database.

In a client/server database product such as MICROSOFT SQL SERVER and ORACLE, triggers are SQL statements (similar to action queries in QBE) saved at the table level. ACCESS, in contrast, does not support table-level triggers. Instead, business rules in ACCESS applications must be enforced by event-driven procedures attached to form objects.

19.2 Learning objectives

- understand the basic concepts of event-driven programming
- create a button that executes several actions when pressed
- understand the difference between the ACCESS macro language and VBA
- use input from users to execute multi-step procedures
- understand how objects within ACCESS are named

19.3 Exercises

In these exercises, you will create event-driven programs using both the ACCESS macro language and VBA.

19.3.1 More flexible buttons

In [Section 17.3.5](#), you used the button wizard to associate an action query with the **on Click** event of a button. This works fine, except that the execution of an action query results in two cryptic warning messages that users should not see. In this section, you are going to create an event-driven procedure that shows a single user-oriented message before running the query.

The answer to the “what?” question is:

1. Turn off the warnings so the dialog boxes do not pop up when the action query is executed.
2. Run the action query.
3. Display a custom message box.
4. Turn the warnings back on.



It is generally good programming practice to return the environment to its original state. Thus, if you turn warnings off in a procedure, you should turn them back on before leaving the procedure.

19.3.1.1 Using a macro to run an action query

You will start by implementing these four steps using ACCESS' macro language.



- Select the **Macros** tab from the database window and press **New**. This brings up the macro editor shown in [Figure 19.2](#).
- Add the four macro actions shown in [Figure 19.3](#). Note that the **OpenQuery** command is used to execute the action query, not “open” it.
- Save the macro as **mcrProcessOrderDetails** and close it.

The answer to the “when?” question is straightforward: the macro should execute

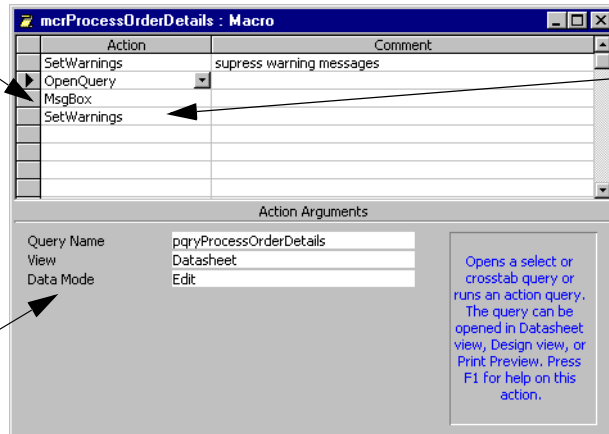
when the user presses the button on the order form. What you have to do at this point is tell ACCESS that the macro you just created should handle all **on click** events raised by the button.

- Open the order form in design mode and bring up the properties sheet for the command button you created in [Section 17.3.5](#).
- Although you will shortly delete the VBA procedure created by the wizard, press the

FIGURE 19.3: Create a macro that answers the “what?” question.

1 The message box should display a message such as, “The order has been processed.”

2 The **OpenQuery** method can be used to open a select query or run an action query. If an action query is used, the **View** and **Data Mode** properties are irrelevant.



3 Use the **SetWarnings** action a second time to turn warnings back on in the ACCESS environment.

4 Save the macro using the “mcr” prefix.



FIGURE 19.2: The macro editor in ACCESS.

Macro actions can be selected from a list. The **SetWarnings** command is used to turn the warning messages (e.g., before you run an action query) on and off.

Multiple commands are executed in sequence from top to bottom

Most actions have one or more arguments that determine the specific behavior of the action.

In the comment column, you can document your macros as required.

The area on the right displays information about the action.

builder to view the code, as shown in [Figure 19.4](#).

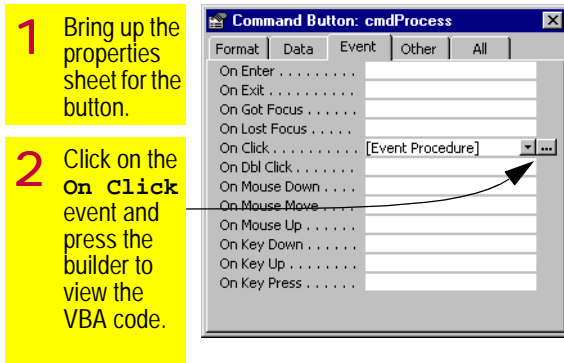
As [Figure 19.5](#) shows, the **on click** event for the button is currently handled by an event procedure (VBA code) created by the button wizard.

The wizard tends to generate needlessly complex VBA code. The same procedure (without the error handling code) could be written using a single VBA statement.

- ➔ Highlight the entire subroutine and delete it. You will replace the event procedure with a reference to your macro.
- ➔ Close the module window. Note that the **on click** event for the button is now empty.
- ➔ Click the combo box arrow in the **on click** property and select the macro you created to update orders. The procedure is shown in [Figure 19.6](#).



FIGURE 19.4: The wizard creates a procedure to handle the button's **On Click** event.



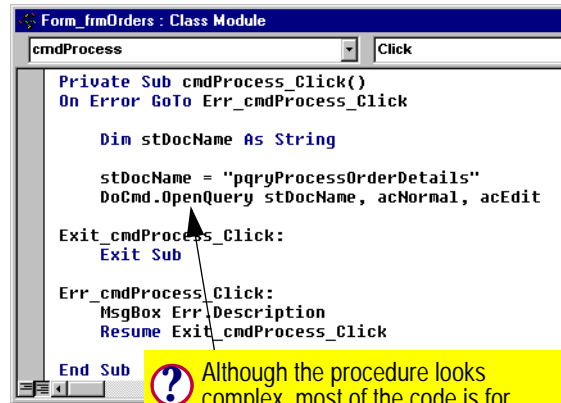
- ➔ Switch to form view and press the button. The message box you defined earlier should inform you that the order has been updated.

19.3.1.2 Using VBA to run an action query

It is possible to achieve the same result using VBA.

- ➔ Switch to design view and bring up the properties sheet for the **cmdProcess** button.

FIGURE 19.5: The VBA event handler created by the wizard.



Although the procedure looks complex, most of the code is for handling errors. The one line that does anything is a macro action called (surprise) **OpenQuery**.

- ➔ Click the combo box, but instead of selecting a macro as you did in Figure 19.6, select "event procedure".
- ➔ Click the builder. The module associated with the order form will open. In it, you will find an empty subroutine called **cmdProcess_Click**, as shown in Figure 19.7.

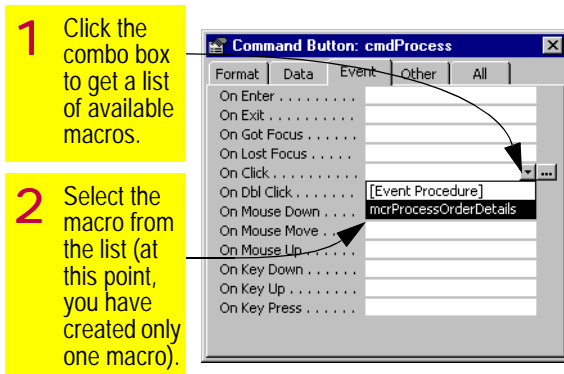
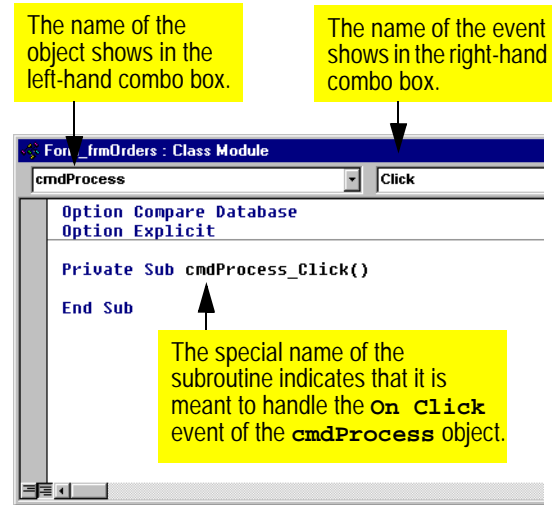
FIGURE 19.6: Tell Access which macro to use to handle the button's *On Click* event.

FIGURE 19.7: The form module with an empty event handler.



- ➔ Define the subroutine as follows (see Figure 19.8):

```
NL Private Sub cmdProcess_Click()
NL     DoCmd.SetWarnings False
NL     DoCmd.OpenQuery
NL     "pqryProcessOrderDetails"
NL     MsgBox "The order has been
NL     processed"
NL     DoCmd.SetWarnings True
NL End Sub
```

- ➔ Close the module and switch to form view. When you press the button, you should get the same result as in Section 19.3.1.1.



While testing your procedures, you will probably process the same order multiple times and corrupted the **QtyOnHand** values in the **Products** table. You may wish to periodically run the rollback query you created in Section 17.3.2 to restore the correct inventory values.



FIGURE 19.8: Define the subroutine using VBA action statements.

1 Enter the VBA code for the event handler.

```

Form_frmOrders : Class Module
cmdProcess Click
Private Sub cmdProcess_Click()
    DoCmd.SetWarnings False
    DoCmd.OpenQuery "pqryProcessOrderDetails"
    MsgBox "The order has been processed"
    DoCmd.SetWarnings True
End Sub

```



Note that the **DoCmd** object's methods provides access to all the commands used in the macro language. As a consequence, anything you can do using a macro, you can do using VBA. However, the reverse is not true.

19.3.2 Conditional procedures

For simple procedures such as the one you just created, the advantages of using VBA may not be obvious. After all, three of the four commands in Figure 19.8 are actually macro language commands. When it comes to looping and conditional branching, however, it is generally much easier to use VBA than to fiddle

with macro language's looping and conditional branching constructs.

19.3.2.1 Motivation for a conditional procedure

As it now stands, there is nothing to prevent users from pressing the "process orders" button multiple times. However, since each order is physically shipped only once, it is clear that each order should be subtracted from **Products** table only once.

19.3.2.2 Updating processed status

The processed/not processed status of an order can be stored as a Boolean (yes/no) variable in the **orders** table. However, it is not realistic to rely on the user to remember to change the **Processed** check box after updating an order.

However, the following line of code can be added to the subroutine to update the field automatically:

```
NL Me.Processed.Value = True
```

In this statement, **Me** refers to the current form. Accordingly, **Me.Processed** refers to the checkbox control bound to the **Processed** field on the current form. **Value** is a property of the **Processed** control; however, since **Value** is also the default property, its inclusion is optional.



Strictly speaking, the syntax of this statement should be:



Me!Processed.Value = True. However, using the dot operator (.) instead of the bang operator (!) allows you to exploit MICROSOFT'S "intellisense technology." See [Section 19.4](#) for more information on techniques for naming objects in ACCESS.

- ➔ While in form design view, bring up the properties sheet for the command button
- ➔ Find the **On Click** event and press the builder to enter the VBA editor for the form module.
- ➔ Modify your code to update the value of **Processed** once the update has occurred.

19.3.2.3 Using the status information

It is possible to use the **Processed** field for a particular order to skip the action query if the update has already been processed.

- ➔ Add the following condition to your code, as shown in [Figure 19.9](#):

```
NL Private Sub cmdProcess_Click()
NL     If Me!Processed Then
NL         MsgBox "This order has already
NL         been processed"
NL     Else
NL         DoCmd.SetWarnings False
NL         DoCmd.OpenQuery
NL         "pqryProcessOrderDetails"
```

```
NL     MsgBox "The order has been
NL     processed"
NL     DoCmd.SetWarnings True
NL     End If
NL End Sub
```



Notice that the condition part of the **IF** statement does not contain an equals sign. With Boolean variables, the equals sign is implied. Thus, **If Me!Processed** is identical to **If Me!Processed = True** and **If Not Me!Processed** is the same as **If Me!Processed = False**.

- ➔ Test the button to make sure the procedure is working correctly. The message box the user sees should depend on the value of the **Processed** field of the current record.

19.3.2.4 Protecting the status information

On final refinement: At this point, it is possible for the user to manually change the value of the **Processed** checkbox.

- ➔ Set the Enabled property of **Processed** to No.



It is possible to use VBA to change the value of a disabled control. However, the ACCESS macro language cannot change the value of a control when it is disabled. Consequently, to change the value of a



FIGURE 19.9: The completed subroutine for processing orders.

If the order has been processed already, the only command that executes is a message box.

```

Form_frmOrders : Class Module
cmdProcess Click
Private Sub cmdProcess_Click()
  If Me.Processed Then
    MsgBox "This order has already been processed"
  Else
    DoCmd.SetWarnings False
    DoCmd.OpenQuery "pqryProcessOrderDetails"
    MsgBox "The order has been processed"
    DoCmd.SetWarnings True
    Me.Processed.Value = True
  End If
End Sub

```

If the order has not been processed, the action query is executed and the **Processed** field updated.

disabled field with a macro, you must first enable the field, make the change, and then re-disable it. All the changes can be made using the **setValue** macro action.

19.3.3 Using the AfterUpdate event

Consider the process of entering a single order detail:

1. A product is selected using a combo box that shows **ProductID** and a product description.
2. The quantity ordered for that particular product is entered. This information is supplied on the sales order sent by the customer.
3. The actual price of the product is entered. The **UnitPrice** for each product is stored in the **Products** table; however, the **ActualPrice** may be different (e.g., a promotion is being run or the customer in question qualifies for a discount).
4. The quantity to ship is determined. Since stockouts are possible, **QtyShipped** may be different than **QtyOrdered**. If **QtyOrdered** is greater than **QtyOnHand**, then the maximum quantity that can be shipped is **QtyOnHand**.

Once these four items of information are specified, the order detail is complete. To reduce the time the user spends entering each order detail, we should automate as much of the process as possible.



19.3.3.1 Getting the default price

When a new order detail record is created, the value of all numerical fields (**QtyOrdered**, **QtyShipped**, and **ActualPrice**) is set to zero by default. However, we do not want the default value for **ActualPrice** to be zero; we want it to be equal to the **UnitPrice** stored in the **Products** table. What is required, therefore, is a procedure that copies the default price into the **ActualPrice** field. To implement the procedure, we must answer two questions:

- What?: Set the value of **ActualPrice** to the correct value of **UnitPrice**.
- When? As soon as the **UnitPrice** of the product is known (i.e., as soon as **ProductID** is specified).

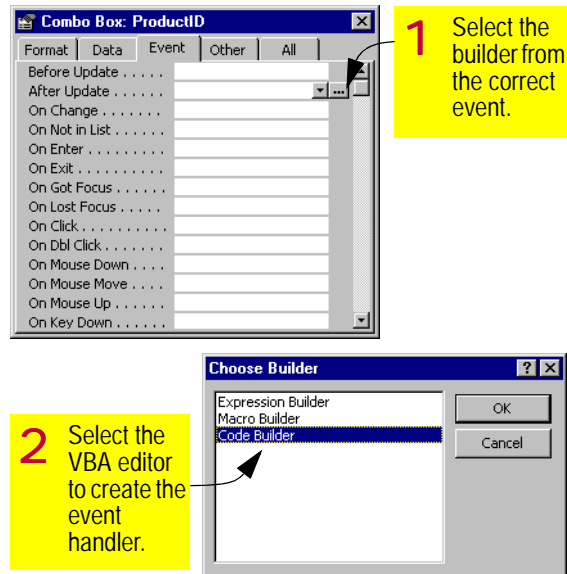


Note the choice of object and event. One might make the assumption that the **On Enter** event of the **ActualPrice** field is the best place for this procedure. However, there is no guarantee that the user will enter the **ActualPrice** field to raise the event. We do know for certain, however, that the user will have to enter a **ProductID**.

- ➔ Open **sfrmOrderDetails** in design mode and bring up the properties sheet for the **ProductID** combo box.

- ➔ Find the **After Update** event and click the builder button (...).
- ➔ You will be given a choice of builders, as shown in [Figure 19.10](#). Select **Code Builder** to get the VBA editor.

FIGURE 19.10: Invoke the VBA editor for the **After Update** event.





- Enter the following assignment statement, as shown in Figure 19.11.

FIGURE 19.11: Create an event-driven procedure to set the default value of *ActualPrice*.

- 1 Set *ActualPrice* to *UnitPrice* as soon as the *ProductID* is changed.

The screenshot shows the VBA editor for the class module `Form_frmOrderDetails`. The `ProductID` control has the `AfterUpdate` event selected. The code in the `Private Sub ProductID_AfterUpdate()` procedure is:

```
Private Sub ProductID_AfterUpdate()
    Me.ActualPrice = Me.UnitPrice
End Sub
```

A field list window for `qryOrderDetails` is open, showing the following fields: `OrderID`, `ProductID`, `QtyOrdered`, `QtyShipped`, `ActualPrice`, `Description`, `Unit`, `UnitPrice`, `QtyOnHand`, and `ExtendedPrice`. The `UnitPrice` field is highlighted.

⚠ Although *UnitPrice* does not show on the subform, it has been projected into the `qryOrderDetails` query and is therefore available for use in the subform. You can use the subform's field list to verify the availability of fields.

NL `Me.ActualPrice = Me.UnitPrice`

- Test the procedure. Whenever a **ProductID** is selected for a new or existing order detail, the **ActualPrice** field should be set to show the product's default price.



Although all fields have a **Default Value** property at the table level, there is no way to use this feature to set a default price. ACCESS permits only simple expressions such as constants (e.g., 0, "ea.") or predefined functions (e.g., `Now()`) to be used in the **Default Value** property.

19.4 Discussion: Object naming in ACCESS

Because the objects in ACCESS are organized into a hierarchy (known as **DATA ACCESS OBJECTS**, or DAO), a naming scheme is required to allow programmers to refer to specific objects. In previous lessons, you have used the expression builder (e.g., Figure 17.7) to navigate the DAO hierarchy graphically and select objects by clicking.

Unfortunately, VBA does not provide an expression builder. Consequently, anyone who wishes to write VBA needs to learn about the DAO hierarchy and the naming conventions used to navigate it. This section provides a brief overview of naming issues for top-level collections, control collections, and properties.



19.4.1 Top-level collections

A **collection** is like an egg carton: it is an object that exists solely to store and organize groups of similar objects (eggs). DAO has a number of top-level collections that contain one or more database objects of the same type. For example, there is a **Forms** collection that contains all the forms in a database. Top-level collections correspond (roughly) to the panes of the database window.

To refer to an item in a collection, you can use its **Item** property in combination with its numerical index. For example, ensure at least one form is open and type the following into the debug window:

```
NL ? Forms.Item(0).Name
```

This statement prints the **Name** property of the “first” open form in the database. Of course, you seldom know the index numbers of items in a collection so using the **Item** property in concert with the object’s name is more convenient (but redundant, at least in this example):

```
NL ? Forms.Item("frmOrders").Name
```

To make the naming scheme even more confusing, VBA allows you to drop the **Item** part (it is assumed by default) and use the bang operator (!) as a syntactical shortcut. As a result, there are many different ways to refer to the same object:

```
NL Forms.Item("frmOrders") `long
version
NL Forms("frmOrders") `shortcut
NL Forms!frmOrders `another shortcut
```

19.4.2 Embedded controls collections

An object, like a form, can both belong to a collection (**Forms**) and contain other collections. For example, each form object in ACCESS contains a **Controls** collection that contains all the textboxes, combo boxes, checkboxes, subforms, and so on.

Since the **Controls** collection is the default “property” for **Form** objects, shortcuts are possible. For example, to refer to the **CustID** combo box on your order form, you could use any of the following:

```
NL Forms.Item("frmOrders").Controls.
Item("CustID") `long version
NL Forms!frmOrders("CustID") `shortcut
NL Forms!frmOrders!CustID `another
shortcut
```



Since the **Forms** collection contains only open forms, your order form has to be open for you to test these examples.

19.4.3 Properties

Each object in Access has predefined properties. Typically, properties are accessed using the dot



(.) operator. For example, each **Form** object has a **Name** property (e.g., **Forms!frmOrders.Name**) and each control has a **Value** property (e.g., **Forms!frmOrders!CustID.Value**). In addition, each object has a single default property; when no property is specified, the default property is used.



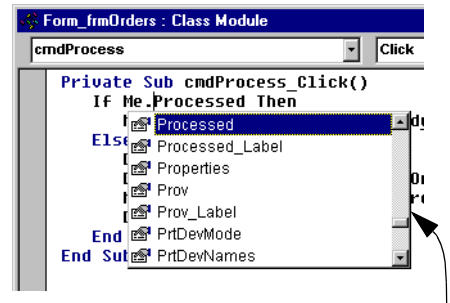
The on-line help system provides a summary of the properties, methods, and collections for each object.

19.4.4 Dot or bang?

According to the ACCESS help system, the bang operator (!) should be used to indicate that what follows is a user-defined item (i.e., an element of a collection). The dot operator (.) should be used to indicate that what follows is a property or something that is not user-defined.

Unfortunately, the “intellisense technology” that appeared in ACCESS version 8.0 does not do a very good job of recognizing the bang operator. For example, if you type “**Me.**” while editing an event handler, the editor will provide you with a list of form properties and objects on the form, as shown in [Figure 19.12](#). However, if you type “**Me!**” (to indicate that you are only interested in objects on the form), intellisense provides no help. As such, it is often easier to use the dot operator, even though MICROSOFT discourages it.

FIGURE 19.12: Using “intellisense technology” to finish the VBA statement.



“Intellisense technology” is a feature introduced in version 8.0 that helps complete VBA statements. In this example, a list is shown for the **Me** object (the current form). It shows all the properties for the form as well as all the objects in the control collection.

19.5 Application to the project

Now that you have an event-driven procedure to set the default price for products, you know all you need to know to implement a similar procedure for suggesting a quantity to ship:

- What? Set the **QtyShipped** field for an order detail to the minimum of **QtyOrdered** and **QtyOnHand**.



- When? As soon as `QtyOrdered` and `QtyOnHand` are known.

➡ Create an event-driven procedure for setting the default quantity to ship. The user should be able to override this value.

HINT: You may want to consider using the `MinValue()` function you created in [Section 18.3.7](#) to implement this feature.

➡ Add a second button to your order form to show the invoice you created in [Section 16.4](#) in “print preview” view.



