CHAPTER II - HOME PAGE & LOG IN

1. Introduction

The Home Page, shown in Figure 2.1. <u>is the opening form of the Information System</u>. It is detached from the data base (i.e., <u>there is not a connection between the form and the tables of the data base</u>) and its functioning is very simple, as it is detailed below.

Fig. 2.1 *The Home Page*

***Structure command (Struttura)* -** This is a button (called *CmdStructure*) that opens a static form called "Sport Centre Structure" (Struttura Sport Village) with basic information and some pictures of the centers. Briefly, we note that a <u>Button is one of the several visual components</u> that can be placed on a form and that can be found in the <u>Controls Menu of the Design Bar</u>. As any objects (the same also holds for a form that, de facto, is an object itself) buttons has several properties, accessible from the <u>Properties menu</u> shown by Figure 2.2:

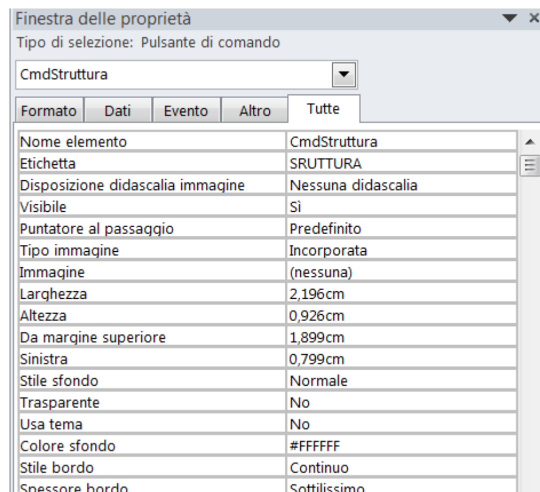

Fig. 2.2. *The Properties of the Control Button*

Properties refer to:

- **Graphical features (Formato)** - They include features like: font, dimension, caption, color, background, etc. These features can be modified also at design time using the mouse. Clearly, in order to modify a property at run time some code is needed. For instance, if we wanted to change the color of a button if something happen we should write something like:

  ```
  If Condition = True Then CmdStructure.ForeColor = RGB(1, 1, 1)
  ```

  Where: ForeColor is the color of the Text visualized on the button and RGB is the function that defines a color using its level of Red, Green and Blue, i.e., RGB(1,1,1) corresponds to black.

  Other important graphical properties are:
  - **Enabled** - If True, the user can operate on the control;
  - **Visible** - If True the control will be shown on the form.
- **Data features (Dati)** - These features specify where data populating an object are gathered. Data features are very important for form, Label and Combo Boxes.
- **Events (Eventi)** - Events are actions that can be associated to an object or actions that the user can perform on an object. For instance, in case of a button, the on click and the on double click events are the most important events. VBA code can be associated to an event, that is, the code is triggered as soon as the event takes place.

  In the present case, CmdStructure is used to open a form, so there is a very simple code associated to its on click event: DoCmd.OpenForm "Sport Centre Structure".

Please note that the OpenForm method can be used to open a specific form. Its Syntax is:

```
OpenForm(FormName, [View], [FilterName], [WhereCondition], [DataMode], [WindowMode], [OpenArgs])
```

where:

- FormName is the name of the Form to be opened;
- View is the view type;
- FilterName is the name of the saved filter that must be used when opening the form;
- WhereCondition is a valid SQL condition that can be used to filter the data of the form;
- DataMode specifies the way in which data will be entered in the form;
- WindowMode specifies the type of window that will be used to show the form;
- OpenArg is a string variable that is passed to the form, and that can be used to perform specific actions on the data.

Memberships command (Abbonamenti) - This button opens a static form called Price List (Listino*)*, which is a Multiple Fields form that shows all the available membership types, with an indication of the passes (to areas and courses) and of the standard monthly price associated to each membership card.

It is interesting to note that, since the user can only check the prices but cannot modify any of them, Price List is a static form. Obviously, a form that is not linked to a table (as the Home Page described above) is static. However, even a form that is connected to a table of the data base can be static. Price List is an example of this kind of form. Indeed, if we check its data properties we can see that:

- **Record Source** (Origine Record) = MEMBERSHIPS - This means that data populating the form are taken, directly, from the MEMBERSHIPS table. If data of the table are changed, modifications are reflected, immediately, on the form.
- **Recordset Type** (Tipo Recordset) = Snapshot - This means that the data shown on the form are a copy (a snapshot) of those recorded in the table. Data cannot be modified. In other words, there is just a mono-directional link between the form and the table. Modifications made on the form do not affect the data of the table but, conversely, modifications made on the table affects also the form. We also note that, to have a bi-directional link between the form and the table the Recordset Type property must be set to Dynaset.

The Price List form is also equipped with a button that opens a second form that allows the user to generate a quotation for a specific membership card. As we will see in the next Section, to do so, the user, if not already registered, has to insert some personal details and, next to select the desired type of membership. As a result the quotation is generated, saved in the Data Base and a printable report is prepared.

We also note that, since the user must have access to a blank form (i.e., he or she has to generate a new request for quotation, but he cannot see the quotations offered to other customers), the following code is associated to the on-click event of the button:

```
DoCmd.OpenForm "QUOTATIONS", , , acFormAdd.
```

In this case also the DataMode input parameters has been used (i.e., acFormAdd), to indicate that a new record must be inserted.

Staff command (Staff) - This button opens a dynamic form called “Instructor List” (Lista Istruttori), which contains three List Boxes with the name of the instructors, grouped in terms of earth, water and swimming courses, respectively. It is also possible to click on the name of each instructor to have additional information about him or her.

Contacts command (Contatti) - This button opens a static form with some basic contact information.

Login command (Log In) - This button opens the *Login* form that allows registered users to access their personal page, by logging in with a valid user name and password.

Question mark command (?) - This button opens a sequence of Message-Boxes with basic information about how to navigate in the web site. An example of the used code (linked to the on-click event) is shown:

```
MsgBox "Welcome at SportVillage!" & vbCrLf & "This is the on line help for an easy navigation"
```

In the form there are also three hyperlinks (that can be easily created by assigning the *Hyper Textual Link* property to the name of the form (or of object) that has to be opened. Specifically:

- Earth Courses - It opens the *Earth* Courses form displays in the list of all the courses of earth type. This form is linked to the saved query Earth Courses (Corsi Terra) in a unidirectional way (i.e., it uses a Snapshot connection). On this form there is also a button that allows the user to visualize the schedule of the earth course. Specifically, it opens another form created on the saved query Earth Courses Time Table, described in the previous chapter.
- Water Courses - It opens the Water Courses form that, compared to the previous on, is a little more structured. Indeed, the user can select (using a Combo Box populated with all the water courses listed in the COURSES table) a water course he or she is interested in, to receive detailed information about it. This information is visualized in a List Box. Also in this case there is a button that opens the weekly time table of water courses.
- Swimming Courses - This hyperlink is “under construction” and opens a message box.

2. The Quotations form (Maschera Preventivi)

2.1 Basic Issues

This form allows one to make a request for a quotation and to save it in the QUOTATIONS table of the database. To do so, it is dynamically linked (i.e., Dynaset connection) to the QUOTATIONS table, as shown in Figure 2.3. The operating logic is simple:

- the user fills the fields (at least those one marked with an asterisk (*));
- next, by clicking the quotation button (Preventivo) he or she receives a proposed price;
- If the offer is accepted, a copy of the quote is saved in the QUOTATIONS table (for future use) and, meanwhile, a printable report of the offer is displayed on the screen;
- The customer can print this copy and use it to validate the offer at the reception desk;
- Conversely, pressing the Exit button, the user aborts all the previous operations, the form is closed and the user is prompted back to the initial page.

This form has some interesting features described below.

There are two text boxes, namely Customer ID (ID Cliente) and Expiring Date (Scadenza) that are visible in design view, but not in Form view. As a matter of fact, in both cases their visible property is set to false, i.e. *Visible = False.*

So, if the user cannot see these fields, why they should be included herein? The reason is simple. The first field contains the ID of a register user that has logged in; the second one is the expiring date of its memberships. Thus, these fields may be useful to compute an "early renewal discount rate" to be applied to the standard quoted price;

Double clicking on the Time Tables (?) label, the list of all the available time windows is automatically displayed on the screen (i.e., a form linked to the TIME WINDOWS table is opened);

Fig. 2.3. *The Quotations form in design view*

In addition, the ability to select the Health Centre depends on the type of membership selected by the user in the CmbMembership Combo box (Tipo Abbonamento). Actually, if the wellness center is already included in the membership chosen by the user, as in the case of a VIP membership that includes all passes, then the *Wellness Centre* Check box (Centro Benessere) is automatically ticked and disabled. Conversely, the check box is de-ticked and it is also enabled, so that the user can freely decide whether he wants to add the wellness center or not.

The VBA code needed to perform these actions is shown below.

```
Private Sub CmbMEmbership_AfterUpdate()
Dim Bool As Boolean
Dim M_Name As String
Dim Condition As String
  ' A check is made using a Domain Function to see if the wellness center is included in the membership
  MName= Me.CmbMembership ' This line assigns the membership's name selected by the customer
  Condition = "Name = " & "'" & MName & "'"
  Bool = Nz(DLookup("[Wellness Center]", "MEMBERSHIPS", Condition), False)
  If Bool = True Then
      Me.ChbWellness.Value = True ' The checkbox is ticked
      Me.ChbWelne.Enabled = False ' The checkbox is disabled
  Else
     Me.ChbWelness.Enabled = True
  End If
End Sub
```

Note that this code is triggered by the After Update event of the Membership Type ComboBox (CmbMembership); in other words, the previous code is executed any time the user select a different membership type in the CmbMembership combo box.

This is shown in figure 2.4, below:

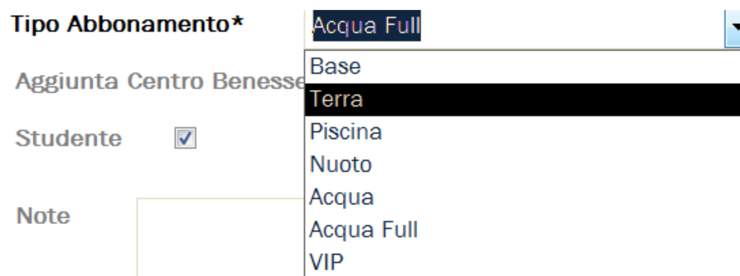

Fig. 2.4. *The Combo Box with different membership cards*

We also note that the code makes use of a DlookUp function to check if the Wellness Center field is true, in the record (of the MEMBERSHIP table) corresponding to the membership type selected by the customer. To this aim, the Dlookup has to reproduce the following SQL query:

```
SELECT [Wellness Centre]
FROM MEMBERSHIPS
WHERE Name = 'X'
```

Where: X is a string containing the name selected by the customer in the combo box.

To ensure that X can vary, we can use a string variable, namely MName, to read the value chosen by the user, by means of the following assignment statement: MName = Me.CmbMembership

This statement says that the string value contained in the Combo Box CmbMembership, placed on the current form on which the user is operating, has been assigned to the string variable MName. In this regard, it is interesting to note that Me is a shortcut used to indicate “the current form” and that the dot (.) operator indicates ownership. In other words, the statement Me.ControlName can be used to identify a specific control placed on the current form.

Owing to what we’ve said, the DFunction can be finally written as:

```
Bool = Nz(DLookup("[Wellness Center]", "MEMBERSHIPS", "Name = " & "'" & MName& "'"), False)
```

Note that, if the record is found then a true or false value is passed to the Boolean variable Bool, depending on the current value of the Wellness Center field; vice versa, if the record is missing, the Nz function returns False and so also Bool becomes false.

Another tricky thing is the way in which the filtering condition has been written. As we know, the filtering condition must be passed as a string and, for this reason, the logical condition must be written inside quotation marks. However, also Name is a string field and so, in SQL it must be passed inside single quotes (such as Name = 'VIP'). This is the reason why the filtering condition is obtained by combining (through the & operator) four sub-strings, each one placed inside quotation marks:

- "Name = " - This is the fixed string;
- "'" - This is the first single quote (i.e., ')
- MName - This is the variable part of the string condition. Since it is a variable of type String, quotation marks are not needed;
- "'" - This is the last single quote (i.e., ')

The last part of the code is easy, the DLookUp sets the value of the Boolean variable Bool. If Bool is true, the checkbox is ticked and disabled (the user cannot change it, because the wellness center is included by default). Vice versa, if Bool is false, the checkbox is not ticked and it is enables, so that the user can make its selection.

2.2 The Calculate Price command (Pulsante Preventivo)

The calculation of the price of the proposed quotation is performed by a VBA code triggered by the on click event of the CmdPrice command. Specifically, to see if the customer can enjoy a discount, 4 distinct SELECT queries must be executed on the DISCOUNTS, MEMBERSHIPS, [TIME WINDOWS] and DURATIONS tables, respectively. This is because, as we known, these tables contain the available discounts and their application criteria.

In the previous Sections, we mentioned that the DoCmd object can be used either to open a form or to run a saved query. Additionally, Docmd is also endowed with a RunSQL method that allows one to write a string of executable SQL code and, next, to run it. Let us make a simple example to clarify this concept.

We have a list of customers that must be blocked, since they registered to a class but, next, they did not show off. In order to do accomplish this task we could use an update query.

Briefly, an Update Query has the following syntax:

<Update Command> : : = **UPDATE** *<Table name >*

SET *<Field name>* = *<Expression>* [{, *<Field name>* = *<Expression>*} …]

[**WHERE** *<Condition>*]

For instance, if the ID of the customers that must be blocked were {1, 5, 10, 11}, we could write an update query as the following one:

UPDATE CUSTOMERS

SET CUSTOMERS.Blocked = 1

WHERE ID **IN** (1, 5, 10, 11)

Now, if we wanted to automate this process, we could write a VBA subroutine that:

- Receives, as input, the list (i.e., the array) of all the ID of the customers that have to be blocked;
- Builds up the SQL code and assigns it to a string variable;
- Executes the SQL code, by invoking the Docmd.RunSQL method.

```
Public Sub UpdateBlocked(ID() As Integer)
Dim InCondition As String ' The string that reproduce the IN (x, y, z) condition
Dim MySQL As String ' The string the contains all the SQL code i.e., UPDATE …
Dim i As Integer ' A looping variable
InCondition = "IN ("
    For i = LBound(ID) To UBound(ID) ' LBound returns the index of the first element of the vector
        InCondition = InCondition & ID(i)
        If i < UBound(ID) Then InCondition = InCondition & ", " ' Comma is not needed after the last ID
    Next i
InCondition = InCondition & ")"
MySQL = "UPDATE CUSTOMERS"
MySQL = MySQL & " SET Blocked = 1"
MySQL = MySQL & " WHERE ID " & InCondition
DoCmd.RunSQL MySQL ' The SQL code contained in the MySQL string is executed
End Sub
```

Unfortunately, RunSQl can be used only to execute deleting or updating queries; conversely selection queries cannot be executed using the RunSQL method.

When there is the need to execute a selection query at run time (i.e., we want to embedd SQL in VBA), two alternatives are availabel:

- Use a Dfunction (as we have seen before);
- Use the RecordSet object.

The second option is a little more complicated, but also much more flexible and efficient. So, in this case we will use a Recordset that, more or less, is an editable copy of a table or, more generally, of any views (i.e., a non-normalized table that can be obtained using one or more SQL queries) of the Data Base[1].

The VBA code is shown below; its visibility is restricted to the Quotation Form and to its controls.

```
Option Explicit ' i.e., all variables must be declared
' Two variables with a visibility restricted to the Quotation Form
Private LoggedUser As Integer  ' This variable contains the ID of the user that is logged in
Private Renewal As Boolean  ' It will be true if the user already has a membership card

Private Sub CmdCalcPrice_Click()
' This is the procedure that calculates the proposed price, based on the selections made by the customer and
' on the type of discounts, if any, that can be applied to the customer

Dim Db As Database
Dim Rs As DAO.Recordset
Dim Price As Currency
Dim Age As Integer, Advance As Integer
Dim Discount, AdDis, Extra As Double
Dim Expiration As Date
Dim MySQL As String

On Error GoTo ErrorHandler: ' In case of missing data an error is triggered and a MsgBox is displayed
  Me.CmdSave.Enabled = False
  Set Db = CurrentDb
  Price = 0
  Discount = 0

  ' In this section the Standard Price of the selected membership card is retrieved from the MEMBERSHIP table
  ' To this aim a SQL query is written to get the standard monthly price of the membership
  MySQL = "SELECT [Monthly Price] FROM MEMBERSHIP WHERE Name = " & "'" &
            Me.CmbMembership & "'"     ' The value selected by the user in the CmbMembershib Compo Box
  Set Rs = Db.OpenRecordset(MySQL) ' The recordset is opened as a copy of the table returned by the SQL
  Price = Rs.Fields(0)*Me.CmbDuration     ' Price = (Monthly Price × Number of months)
  Rs.Close      ' Now we need to look in another table so we close the recordset and we will re-open it
```

[1] As usual, new programming concepts (as Recordset) are explained in full details at the end of the Chapter.

```
' Possible Discounts are evaluated here
Set Rs = Db.OpenRecordset("DISCOUNTS")  ' The recordset is opened as a copy of the DISCOUNTS table
Rs.MoveFirst ' First record, the “teenager one”
' Discount for age (teenager or over 65)?
Age = DateDiff("yyyy", Me.TxtBirthDate, Now()) ' This is the age of the user
If Age <= Rs.Fields("Condition") Then
   Discount = Discount + Rs.Fields("Discount")
End If
Rs.MoveNext ' Second Record the “over 65 one”
If Age >= Rs.Fields("Condition") Then
   Discount = Discount + Rs.Fields("Discount")
End If
' Discount for student?
If Me.TxtStudent = True Then
   Rs.MoveNext
   Discount = Discount + Rs.Fields("Discount")
End If
Rs.Close  ' Now we need to look in another table so we close the recordset and we will re-open it
' Discount for duration longer than one month?
MySQL = "SELECT Discount FROM DURATIONS WHERE Duration = " & Me.TxtDuration
Set Rs = Db.OpenRecordset(MySQL)
   Discount = Discount + Rs.Fields(0)
Rs.Close
' Discount for “cheap” time windows?
MySQL = "SELECT Discount FROM [TIME WINDOWS] WHERE Name = " & "'" & Me.TxtTimeWindow & "'"
Set Rs = Db.OpenRecordset(MySQL)
   Discount = Discount + Rs.Fields(0)
Rs.Close
' Extra charge for wellness center?
Extra = 0
If Me. Me.ChbWelness.Value = True Then  ' First we need to see if the Wellness center is included or not
   MySQL = "SELECT [Wellness Center] FROM [MEMBERSHIPS] WHERE Name= " & "'" &
            Me.CmbMembership & "'"
   Set Rs = Db.OpenRecordset(MySQL)
   If Rs.Fields(0) = False Then ' If it is not included we need to see how much is the extra price
      Rs.Close
      MySQL = "SELECT * FROM DISCOUNTS WHERE Name = " & "'Wellness'"
      Set Rs = Db.OpenRecordset(MySQL)
      Extra = Rs.Fields("Extra")
      Rs.Close
   End If
End If
```

```
 ' This part concerns anticipated renewals and is executed only for registered user. We will discuss it later on
 If Advance Then ' Advance is true if the user has already a membership card
    AdDis = 0
    MySQL = "SELECT DURATIONS.Duration, CUSTOMERS.[Starting Date] FROM "
    MySQL = MySQL & " DURATIONS INNER JOIN CUSTOMERS ON DURATIONS.ID = CUSTOMERS.ID_Duration"
    MySQL = MySQL & " WHERE CUSTOMERS.ID = " & LoggedUser
    Set Rs = Db.OpenRecordset(MySQL)
    ' Compute the expiration date taking the sum of the starting date and of the duration
    Expiration = DateAdd("m", Nz(Rs.Fields(0), 0), Nz(Rs.Fields(1), Date))
    Advance = DateDiff("m", Date(), Expiration) ' Months to the expiration date i.e., renewal advance
    Rs.Close
    Set Rs = Db.OpenRecordset("SELECT Condition, Discount FROM DISCOUNTS")
    Rs.Move (3) ' Record 4 to 6 define different discount rates, depending on the advance
    If Advance >= Rs.Fields("Condition") Then
       AdDis = Rs.Fields("Discount")
       Rs.MoveNext
       If Advance >= Rs.Fields("Condition") Then
          ScAnt = Rs.Fields("Discount")
          Rs.MoveNext
          If Advance >= Rs.Fields("Condition") Then
             ScAnt = Rs.Fields("Discount")
             Rs.MoveNext
          End If
       End If
    End If
    Discount = Discount + AdDis
 End If
 Set Db = Nothing  ' We destroy the object (the pointer to the object) to free memory
 Set Rs = Nothing
 If Discoint > 0.15 Then Discount = 0.15 ' Maximum discount rate has been reached
 Me.TxtPrice = (Price * (1 + Extra)) * (1 - Discount)
 Me.CmdSave.Enabled = True ' Saving is now possible
 Exit Sub
ErrorHandler: 'If some data are missing the recordset is empty and an error is triggered. So this part is executed
 Me.CmdSave.Enabled = False
 MsgBox "Please fill all input box with the asterix", vbCritical, "Missing inputs"
 Exit Sub
End Sub
```

The logic of the code is rather simple, and it is briefly pinpointed below:

- At first, the standard price is computed taking the product of the monthly price (of the membership selected by the customer) and of the number of months (selected by the customer)
- Next, data inserted by the customer are analyzed, taking into consideration each possible discount opportunity (for age, profession, etc.)
- In this way a total discount rate is obtained and it is used to compute the proposed price displayed on the form.

Concerning the standard price, the number of months is taken, directly, from the text box TxtPrice; conversely, the standard monthly price of a membership must be taken from the MEMBERSHIPS table. Since the value that we need is a scalar, we could get this value with the following DLookUp:

```
Price = DlookUp("Monthly Price", "MEMBERSHIP", "Name = " & Me.CmbMembership)
```

However, in the code a Recordset, rather than a Domain Function is used. Before moving further on we need to introduce some basics concerning the use of a recordset.

To use a recordset, we need to create it writing the following code:

```
Set Db = CurrentDatabase
Set Rs = Db.OpenRecordset(<Opening Argument>)
```

The first statement creates a Data Base variable and links it to the current Data Base (i.e., Db points to the Data Base that is currently being used).

The second statement creates a recordset - a copy of an object of the current Db - that is accessible both for reading and for writing. The fact that the object pointed (copied) by the recordset belongs to the current database is indicated by the dot (.) operator, which follows the Db variable. The object pointed by the recordset is indicated by the opening argument that can be a table, a saved query or a correct selection query written in SQL.

For example, by writing:

- Set Rs = Db.OpenRecordset("CUSTOMERS"), Rs becomes a perfect copy of the CUSTOMERS table;
- Set Rs = Db.OpenRecordset("SELECT Name, Surname FROM CUSTOMERS"), Rs becomes a perfect copy of the projection of the CUSTOMERS table where only the Name and the Surname fields (of all the customers) are shown.

Also, when a recordset has been created and opened, it is possible to move along its records using the **Move First, MoveLast** and **MoveNext** methods and it is also possible to jump from a field to another one using the **Fields()** property.

Let us consider the following code:

```
Set Rs = Db.OpenRecordset("CUSTOMERS")
Rs.MoveLast
SVar = Rs.Fields(2)
```

A recordset that points to the CUSTOMER table is created. Next, the last record is selected and the value of its third field (fields is a zero based collection) is assigned to the string variable SVar. Please note that, instead of using an index, a fields can also be identified using its name. For instance:

```
SVar = Rs.Fields(2)
```

and

```
SVar = Rs.Fields("Surname")
```

are, exactly, the same instruction.

Now we have enough background to understand the way in which the monthly price has been taken from the MEMBERSHIP table.

At first the following SQL is assigned to the string variable MySQL:

```
MySQL = "SELECT [Monthly Price] FROM MEMBERSHIP WHERE Name = " & "'" & Me.CmbMembership & "'"
```

Next a recordset is opened using this SQL:

```
Set Rs = Db.OpenRecordset(MySQL)
```

Since the query we wrote returns a single value, Rs has a single record with a single field. So, to read this value and to multiply it for the number of months selected by the customers it is sufficient to write:

```
Price = Rs.Fields(0)*Me.CmbDuration     ' Price = (Monthly Price × Number of months)
```

Let us now consider the way in which the discount for the age is considered. Clearly the information that we need are in the DISCOUNT table. So we need to close the current RecordSet, to re-open it on the DISCOUNT table:

```
Rs.Close
Set Rs = Db.OpenRecordset("DISCOUNTS")
```

At this point the recordset is equal to Table 2.1.

Tab. 2.1 *The DISCOUNT table pointed by the Recordset*

ID	Name	Description	Condition	Discount	Extra
1	Teenager	Users that are younger than	19	25%	0%
2	Elder	People that are over	65		
3	Student	Users that study at high school or College	Student	15%	0%
	…	…	…	…	
N	Wellness	Add the Wellness Center	…	0%	10%

So if we write:

```
Rs.MoveFirst
Age = DateDiff("yyyy", Me.TxtBirthDate, Now())
If Age <= Rs.Fields("Condition") Then
        Discount = Discount + Rs.Fields("Discount")
 End If
Rs.MoveNext
If Age >= Rs.Fields("Condition") Then
      Discount = Discount + Rs.Fields("Discount")
End If
```

The first record is selected and the age of the customer (computed using the DateDiff function) is compared (inside the *If…Then* statement), with the value contained in the Condition filed of the first record (i.e., 19).

If the age falls below the threshold limit of 19, then the value contained in the Discount field of the same record is assigned to the Discount variable. Next, the second record is selected, and a similar procedure is used to see if the over 65 discount rate can be assigned.

From here on the VBA code maintains the same structure and all the possible discount opportunities are considered one by one, until the total discount is computed.

At the end of the code we find the following statements:

```
Set Db = Nothing
Set Rs = Nothing
```

These statements are used to destroy both the Db and the Rs objects. Since we do not need them any more we destroy them both to free memory.

Note that **Rs.Close** has a totally different meaning; in this case, although the data stored in the Rs have been deleted, an adequate amount of memory is still allotted to the object.

Also note that the code begins with the following declaration:

```
On Error GoTo ErrorHandler:
```

This is used to prevent unpredictable code interruptions due to mistakes made by the user. Specifically, if the user does not fill all the mandatory fields of the form, some SQL strings may result incomplete. Consequently, the Recordset cannot be properly opened and an error is triggered. However, the On Erro Goto command prevent code's interruptions since, when an error occurs (during execution) the code is diverted to the label ErroHandler and only the part of code below this label is executed.

In this case we have:

```
ErrorHandler:
    Me.CmdSave.Enabled = False
    MsgBox "Please fill all input boxes marked with an asterisk", vbCritical, "Missing inputs"
```

So, the save button is disabled and an error message is displayed to signal to the user the need to fill in all the mandatory fields (that are highlighted with an asterisk).

2.3 The Save and the Exit Command (Salva e Esci)

As we have previously noted the Quotation form uses a dynaset connection with the QUOTATIONS table (i.e., RecordSource = QUOTATIONS and Recordset Type = Dynaset). For this reason the connection among the table and the form is bidirectional: any change made on the table is reflected on the form and any change made on the form is reflected on the table. So, as soon as the user start typing some data in the form, a new record is created (in the QUOTATIONS table) and filled with the data inserted by the customer. So, at least apparently, there is no need of a save button, as the record is automatically and dynamically saved.

As a matter of fact, the Save button do not save the record, but it simply closes the form, updates the database, and opens a printable report of the quotation. This is made using the following code, linked to the on click event of the command.

```
Private Sub CmdSave_Click()
  Me.Requery ' Requery is used to update the data of the data base and that displayed on the form
  DoCmd.SetWarnings False ' This is used to avoid critical messages being displayed
  DoCmd.OpenReport "Quotation", acViewPreview ' The report is opened
  DoCmd.SetWarnings True ' The form is closed
  DoCmd.Close acForm, "Quotation" ' Now critical messages are allowed once again
End Sub
```

Conversely, the Exit command plays a more important role. Indeed, if the user clicks this button he wants to stop, abruptly, the generation of a quotation. So, all the data that he has written until that moment must be erased from the QUOTATIONS table. This action is delegated to the Exit button, that takes advantage of the **acCmdUndo** method of the **DoCmd object** that deletes the last insertion made into a table.

This is shown below.

```
Private Sub CmdClose_Click()
  On Error Resume Next  ' In case of error the code proceeds to the next row
  DoCmd.RunCommand acCmdUndo ' New data are deleted thanks to the acCmdUndo
  DoCmd.Close acForm, "QUOTATIONS" ' The form is closed
End Sub
```

2.4 The Quotation Report (Report Preventivo)

When data (relative to the request for quotation made by a customer) are saved, a printable report, as the one of Figure 2.5 (a), is shown on the screen.

OFFERTA ISCRIZIONE SPORTVILLAGE (PR)

Preventivo **N° 52** **Rilasciato a Nome1 Cognome1 il 13/12/2015**

Dati offerta:

DURATA [mesi]	1
FASCIA ORARIA	Full
TIPOLOGIA	Base

L'offerta è comprensiva di centro benessere

Offerta

Prezzo proposto, da convalifarsi in sede: € 35,70

Per convalidare l'offerta è necessario presentare l'offerta in palestra

entro e non oltre il: 28/12/2015

Fig. 2.5 (a). *The Printable Quotation Report*

This report is based on the following saved query ("Query Report Preventivi"), which makes use of a concatenated query:

```
SELECT QUOTATIONS.ID, QUOTATIONS.Name, QUOTATIONS.Surname, _
                _QUOTATIONS.[Date of Birth], QUOTATIONS.Duration, QUOTATIONS.[Time Window], _
                _QUOTATIONS.[Membership Type], QUOTATIONS.[Wellness Centre], _
                _ QUOTATIONS.[Proposed Price]
FROM QUOTATIONS
WHERE QUOTATIONS.ID = (SELECT MAX(ID) FROM QUOTATIONS)
```

As it can be seen, the query returns all the fields of the QUOTATIONS table (i.e., the data needed to fill the report). Also, since a sub-query is contained in the WHERE condition, and this subquery returns the value of the last inserted ID, only the last record is displayed. This is correct, because the last inserted record, of the QUOTATIONS table, is the one just created by the user.

It is also interesting to <u>see how the report looks like in design view</u>, as shown by Figure 2.5 (b):

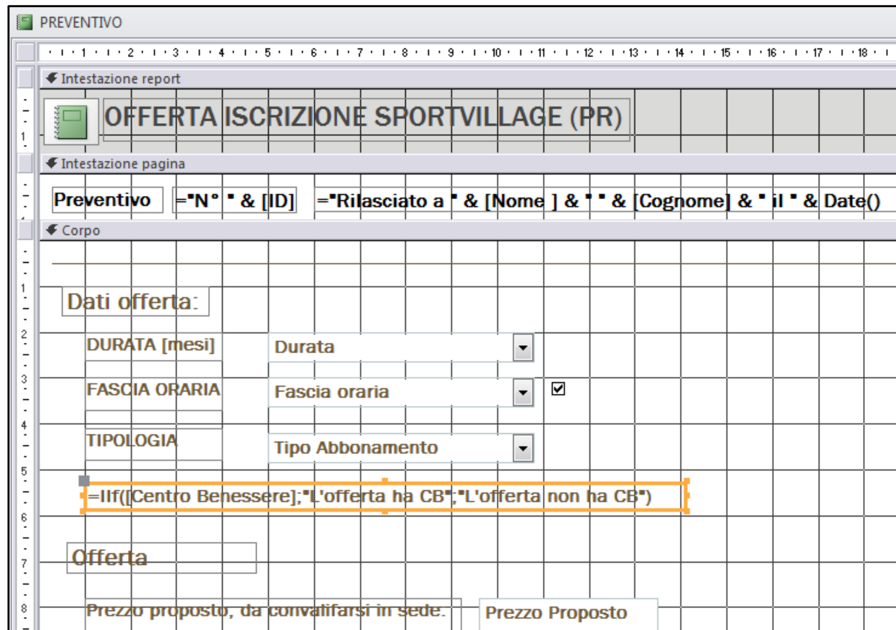

Fig. 2.5 (b) *The report in design view*

As it can be seen, <u>each field of the linked query can be recalled in anyone of the text boxed of the form</u>. For example:

- Quotation = "N°" & [ID] will show, in form view, something like: *Quotation N° 54*, where 54 is the ID of the last created record.
- = "Made For" & [Name] & " " & Surname will show, in form view, something like: *Made for Bill Wilder*, where Bill Wilder is the full name of the last customer that asked for a quotation.

Also note that <u>a formula based on the **IIF** condition has been placed inside a text box</u>:

```
= IIF([Wellness Centre];"The offer includes the SPA";"The offer does not include the SPA")
```

The IIF condition is similar to an IF condition written in Excel; its syntax is as follows:

IFF(Condition; <*value if true*>; <*value if false*>)

Specifically, a <u>logical condition is passed as input. If the condition evaluates true, the IFF function returns the first value (i.e., value if true); it returns the second value (i.e., value if false) otherwise.</u>

In this case, since the [Wellness Centre] is a Boolean field it can be used, straight, as a logical condition. If the value of this field (returned by the query on which the form is based) is true, then the report will display “The offer includes the SPA”; vice versa, it will display “The offer does not include the SPA”

3. Form Water Courses (Corsi d'Acqua)

This form, shown in Figure 2.6, gives some information concerning the water courses that are actually available in the sport center. To this aim, when the user selects a course from the list of the Combo Box (called CmbWcourses) basic info about the course are displayed in the Text Box (TxtWCourse) at the bottom of the mask.

Figura 2.6 *Form Water Courses*

The form is not-linked (i.e., there are no connections either with tables or queries of the data base), yet CmbWcourses is populated with all the water courses listed in the COURSES table.

In order to obtain this result, in the property menu of the control, it is sufficient to:

- Leave blank the **Control Source** (Origine controllo) property - This states that there is not a direct link among the table and the object;
- Set to "Table/Query" the **Row Source Type** (Tipo origine riga) property - This states that the control will be populated with values taken from a table or from a query;
- Write a correct select query in the **Row Source** (Origine riga) property - This defines the data that will be displayed in the control.

In this case, since we want the list of all the water courses, we need to type in the following query:

```
SELECT Name
FROM COURSES
WHERE Category = "Water Courses"
ORDER ASC BY Name
```

To display the main information of the selected course on the Text Box, we will use a VBA script triggered by the After-Update event of the CmbWcourses.

Briefly: *(i)* the code uses a Recordset to collect data from the COURSES and from the INSTRUCTORS tables and, next, *(ii)* it copies this information into a string. Lastly, *(iii)* it properly formats and pastes the string in the Text Box. The full code is shown below:

```
Private Sub CmbWCourses_AfterUpdate()
Dim CName, Instructors, Level, Description, MySQL As String
Dim Db As DAO.Database
Dim Rs As DAO.Recordset
Dim IdCorso As Integer, IstrID As Integer
  CName = Me.CmbWCourse.ItemData(Me.CmbWCourse.ListIndex) ' The name show in the Combo Box
  ' The first query needed to retrieve courses' information
  MySQL = "SELECT ID, Name, Description, Level FROM COURSES WHERE Name = " & "'" & CName & "'"
  Set Db = CurrentDb
  Set Rs = Db.OpenRecordset(MySQL, dbOpenDynaset) ' The second argument opens Rs in read-write mode
    Rs.MoveFirst
' Data are read and assigned
    ID = Rs.Fields("ID")
    Level = Rs.Fields("Level")
    Description = Rs.Fields("Description") ' All collected data are copied (appended) in  the string Description
  Rs.Close
 ' The second query needed to retrieve instructors' information
  MySQL = "SELECT INSTRUCTORS.Name, INSTRUCTORS.Surname _
           FROM INSTRUCTORS INNER JOIN [COURSES-INSTRUCTORS] ON_
           INSTRUCTORS.ID = [COURSES-INSTRUCTORS].IDInstructor_
           WHERE [COURSES-INSTRUCTORS].IDCourse = "
  MySQL = MySQL & ID & " ORDER BY [COURSES-INSTRUCTORS].[Principal Instructor]
  Set Rs = Db.OpenRecordset(MySQL, dbOpenDynaset)
    Rs.MoveFirst ' The first record corresponds to the principal instructor
    Instructors = "Principal Instructor: " & Rs.Fields(0) & " " & Rs.Fields(1) & ";" ' Name and Surname
 ' Now we need to check if one or more possible substitutes have been assigned to the course
 If Not Rs.EOF Then 'EOF is true if the current record is the last one. If so, there are no replacements
      Instructors = Instructors & vbNewLine & "Other Instructors:"   ' vbNewLine implies a new line
      Rs.MoveNext ' Second record
      Do While Not Rs.EOF  ' Now we cycle on all the other records
        Instructors = Instructors & " " & Rs.Fields(0) & " " & Rs.Fields(1) & ";"
        Rs.MoveNext
      Loop
    End If
  Rs.Close
  Set Rs = Nothing
  Set Db = Nothing
```

```
  'Collected information are finally copied on the form
  Me.TxtWcourse.Value = UCase(Cname) & vbNewLine & vbNewLine & Description & ";" _
                & vbNewLine & vbNewLine & "Level: " & "   " & Level & ";" _
                & vbNewLine & vbNewLine & Insructors
Err:
     Exit Sub
End Sub
```

Before proceeding further on, it is important to underline two elements of this code.

First of all, in order to read the value selected by the user in the Combo Box CmbWCourse the following code has been used:

```
   CName = Me.CmbWCourse.ItemData(Me.CmbWCourse.ListIndex)
```

Whereas a Text Box (or a Label) has a single value, a Combo Box contains several values. In the present case the CmbWCorse contains the following values {Aqua Gym, Aqua Movida, Hydro Bike, etc.}. In a certain sense, *the* Combo Box it is an array of strings and so, if we want to refer to a specific value, we need to use an array notation. The **ItemData()** and the **ListIndex** properties can be used exactly to this aim.

Specifically:

- **ItemData()** is a vector containing all the data listed in the Combo Box.

  So, for example, typing CName = Me.CmbWCourse.ItemData(0), the value “Aqua Gym” is assigned to the CName variable, as the first value of the list (i.e., the zero indexed one) is “Aqua Gym”. We also remember that Me is a shortcut to indicate the current form and that the dot (.) operator indicates ownership; due to these issues the statement CName = Me.CmbWCourse.ItemData(0) can be literally translated as: “assign to the string variable CName the value of the first item contained in the CmbWCourse Combo Box of the Water Courses Form”.
- **ListIndex** operates in a reverse way, as it returns the index of the item that is actually selected and displayed on the Combo Box.

  So, for instance, if the “Hydro Bike” was selected, then typing Index = Me.CmbWCourse.ListIndex a value equal to 2 would be assigned to the Index variable.

It is should be now clear that CName = Me.CmbWCourse.ItemData(Me.CmbWCourse.ListIndex) assigns to the CName variable the value of the selected item of the CmbWCourse Combo Box.

It is also interesting to see how the **Do…While loop and the EOF (End Of File)** condition have been used, jointly, to cycle on all the records of a Recordset. To this aim we recall that every course is assigned to a principal instructors and, eventually, to one or more replacements. So, when a Recordset is opened on the following query, it may contain one or more records (i.e., more records are returned in case of a course with a principal instructor and some replacements).

```
SELECT INSTRUCTORS.Name, INSTRUCTORS.Surname
FROM INSTRUCTORS INNER JOIN [COURSES-INSTRUCTORS] ON _
                                        _ INSTRUCTORS.ID = [COURSES-INSTRUCTORS].IDInstructor
WHERE [COURSES-INSTRUCTORS].IDCourse = [ID]
ORDER BY [COURSES-INSTRUCTORS].[Principal Instructor]
```

Owing to this issue, the following Do … While Loop makes it possible to cycle on all records.

```
Rs.MoveFirst
Instructors = "Principal Instructor: " & Rs.Fields(0) & " " & Rs.Fields(1) & ";"
Do While Not Rs.EOF
    Istructors = Istructors & " " & Rs.Fields(0) & " " & Rs.Fields(1) & ";"
    Rs.MoveNext
Loop
```

Initially the firs record is considered and the name and the surname of the principal instructors, contained in the first and in the second field of the current record, are assigned to the string variable Instructor, by means of the following assignment statement:

```
Instructors = Instructors & " " & Rs.Fields(0) & " " & Rs.Fields(1) & ";"
```

Also note that the name and the surname are divided by a blank space and that a semicolon is appended at the end of the string.

Next, if there are many records, the EOF condition is false (i.e., the first record does not coincide with the last one) and the instructions contained inside the Do While Loop are executed. So, the name and the surname of the first substitute are appended to the Instructors string and, next, the focus is shifted to the following record with the Rs.MoveNext instruction. If there are no other records, the EOF condition becomes true and the loop ends; otherwise the instructions are repeated until the last record is reached and all the additional substitutes are appended to the string.

For instance, if the selected course has a principal and a substitute instructor, Frank Deming and John Doe, respectively, at the end of the code, the Instructors variable will contain the following string “Frank Deming; John Doe;”

We conclude this section noting that, it is advisable to write some code to avoid that both CmbWCourse and TxtWCourses are empty when the form is opened for the very first time. To this aim we need to add two instructions to the On Open event of the Form, as shown below:

```
Private Sub Form_Load()
   Me.CmbWCourse.Value = Me.CmbWCourse.ItemData(0) ' The first element of the list is selected
   Call CmbWCourses_AfterUpdate() ' The procedure that fill the Text Box is invoked
End Sub
```

4. Staff Form (Lista Istruttori)

This form, shown in Figure 2.7 (a), is based on three List Boxes (components similar to a Combo Box) that reports all the instructors that are qualified to teach in Aqua, Water and Swimming courses.

Fig. 2.7 (a) *Staff Form*

The main peculiarities of this forms are the following ones:

- The picture at the bottom of the form changes, depending on the List Box selected by the user;
- Only the instructor selected by the user is highlighted. Without adding some lines of VBA code, by selecting an instructor in a List box and an instructor in another List box, both instructors would be highlighted;
- With a double click on the instructor's name a report with some basic info on the instructor is displayed.

Also this form is not linked with tables or query of the database. Anyhow, to fill the List Box a simple selection query is used. For example, the Row source property of the LsbEarth one (i.e., Earth List box) is associated to the following SQL:

```
SELECT Instructor
FROM [EARTH INSTRUCTORS]
ORDER BY Instructor
```

Here, [EARTH INSTRUCTORS] is a saved query that property filters data and that links in a single string the name and the surname of an instructor, as shown below:

```
SELECT Combine(Name, Surname) AS Instructor
FROM INSTRUCTORS
WHERE [EARTH COURSES] = True
ORDER BY Surname, Name
```

And where Combine() is a public function defined as follows:

```
Public Function Combine(N As String, S As String) As String
        Combine = N & "-" & C
End Function
```

4.1 How to remove focus from selected instructor

When the user selects the name of an instructor (in a List Box), the name receives focus and gets highlighted. For example, if the user clicks on Marco-Mauri, the name gets highlighted, as shown in Figure 2.7 (a). Specifically, the **Selected(i)** property of the *i-th* element of the List Box becomes true. If the user selects another name in the same List Box, the focus skips to the last selected name. This behaviour is fine, and we want to maintain it; however, everything works well, only if the users operates on the same List Box. If the user selects a name in another List Box, as shown in Figure 2.7 (b), both names (i.e., also the previous selected one) get highlighted.

Fig. 2.7 (a) *Staff Form with double selection – bad behaviour …*

This is because, being two different objects, both can have the Selected properties set to true.

Clearly this is a drawback that we need to solve, as we want that only the last selected name gets highlighted. To this aim, a specific procedure, namely Remove_Selection has been written.

Briefly:

- The procedure identifies, for each List Boxes, the last item that was selected by the user;
- Next it assigns the index of this item to the Pos integer variable;

  Since List Boxes, as Combo Boxes, are provided of a ListIndex property, the assignment can be easily made with the following line of code:

  ```
  Pos = Me.LsbX.ListIndex
  ```

  Where: LsbX is a generic List Box
- Lastly, the selection is removed from the selected item using the following instruction:

  ```
  Me. LsbX.Selected(Pos) = False
  ```

The full code is shown below.

```
Private Sub Remove_Selection()
' At the end of the procedure there will be no highlighted name, in none of the List boxes
Dim Pos As Integer
On Error Resume Next ' If a List box is not selected Pos = Null and Selected(Pos) returns an error!!!
   Pos = Me.LstEarth.ListIndex
   Me. LstEarth.Selected(Pos) = False
   Pos = Me.LstAqua.ListIndex
   Me. LstAqua.Selected(Pos) = False
   Pos = Me.LstSwim.ListIndex
   Me. LstSwim.Selected(Pos) = False
End Sub
```

Please note that, in this case the instruction On Error Resume Next is extremely important. Indeed, when we search for the selected element in the LsbZ List Box, if no element has been selected, the following instruction Me.LstX.ListIndex issues an error. So, in this case, if we hadn't used the On Error Resume Next instruction, the code would have stopped. Conversely, using the On Error Resume Next instruction the error is ignored and the execution moves to the next instruction. This is fine, indeed, in case of error, none of the elements were selected and so there is no need to "deselect" them.

It is also important to note that this procedure is invoked anytime a new selection is made. In other words, the Remove_Selection procedure is triggered anytime the user changes selection and even if the user clicks on the background (i.e., the body) of the form (i.e., it is triggered by the On_click events of each List Box and of the body of the form, as shown in the following code). Also note that, when the user selects a name in one of the List box the picture displayed at the bottom of the form is automatically updated. Both these features are shown in the code below.

```
Private Sub Body_Click()
' If the user clicks on the body of the form, then focus is removed from all the List boxes
   Call Remove_Selection
End Sub

Private Sub LsbAqua_Click()
   Call RemoveFocus
    FilePath = CurrentProject.Path  ' The path C:\... where the current file is saved
    FilePath = FilePath & "\Acqua.jpg"  ' The name of the image (that must be in the same folder) is appended
    Me.Image.Picture = FilePath ' The image is displayed on the screen
End Sub

Private Sub LsbEarh_Click()
   ' [...] Code has been removed. It is the same as before, only the name of the picture changes
End Sub

Private Sub LsbSwim _Click()
   ' [...] Code has been removed. It is the same as before, only the name of the picture changes
End Sub
```

4.2 Opening a form containing data of the selected instructor.

If the user double clicks the name of an instructor, a static form containing basic info abount the instructor is shown. An example is given inFigure 2.8 (a):

Fig. 2.8 (a) *Courses-Instructors Form*

The interesting part is that the skeleton (i.e., the structure) of the above-mentioned form is the same for all the instructors. That is, a single form has been designed, but data that are loaded onto it depends on the selection

made by the user. In order to do so, the form has been connected (with a Snapshot link) to a dynamic query called Courses-Instructor, that has been already described in Chapter I.

Briefly we recall that this is a simple selection query that shows all the instructors that are qualified to teach in a course. The query filters data depending on the selection made by the customer (on the Staff Form) and, to this aim the query operates on three tables (namely COURSES, COURSES_INSTRUCTORS and INSTRUCTORS) and makes use of Public functions written in VBA.

The SQL code is the following one:

```
SELECT COURSES.Name
FROM COURSES INNER JOIN (INSTRUCTOR INNER JOIN COURSES_INSTRUCTORS ON _
                              _INSTRUCTORS.ID = COURSES_INSTRUCTORS.ID_Instructor) ON _
                              _COURSES.ID = [COURSES_INSTRUCTORS].[ID_Course]
WHERE (ISTRUCTORS .Name= GetName()) AND (ISTRUCTORS.Surname = GetSurname())
```

Where: GetName() and GetSurname() are two public functions that read and return the value of two global variables of type String that contain a name and a surname, respectively. In order to properly operate, the value of these variables is updated any time the user double clicks on the name of an instructor.

This is shown in the code below, for the LstEarth List Box, only; the code is the same for all the other List boxes.

```
Private Sub LstEarth_DblClick(Cancel As Integer)
Dim Instructor As String
   Instructor = Me. LstEarth.ItemData(Me. LstEarth.ListIndex)
   Call OpenForm(Instructors)
End Sub

Private Sub OpenForm(Instructor)
Dim FullName() As String
  ' Split takes a string and a character as input. The string is split into sub-strings delimited by the character
  ' (i.e., the delimiter) and passed to a vector. In this case Istruttore is a sting like this: Name-Surname,
   ' so the "-" character is used as delimiter
   FullName = Split(Istruttore, "-") ' Name and Surname are public variables
   Name = FullName(0)  ' Name is the first sub-string
   Surname = FullName(1) ' Surname is the second (and last) sub-string
   ' Now that the global variables have been updated, the query on which the form is opened can work properly
   DoCmd.OpenForm ("Courses-Instructors")
End Sub
```

It is also important to note that the query returns a table, like the one shown below:

Tab 2.2. *A possible output of the query*

Name	Surname	Picture	Date	Course
Veronica	Del Pietro	P1.gif	09/04/1988	Body Sculpt
Veronica	Del Pietro	P1.gif	09/04/1988	Pilates
Veronica	Del Pietro	P1.gif	09/04/1988	Step
Veronica	Del Pietro	P1.gif	09/04/1988	XTime

As it can be see, the personal information of the instructors is repeated as many times as the number of courses that are assigned to the instructor. For these reason, if one created a standard form on this query the following result would be obtained:

Fig. 2.8 (b) Wrong *Courses-Instructors Form*

Since the query returns four records, the standard view shows one record at a time and so, in order to visualize all the courses assigned to Veronica, there is the need to navigate through records using the navigation panel at the bottom of the form.

Similarly, if one created a multi-objects form, the following result would be obtained:

Fig. 2.8 (c) Multi-Objects *Courses-Instructors Form*

This solution is even worse, as all the information of the instructors are repeated four time.

In order to get a form as the one of Figure 2.8 (a), we need to properly combine solution (b) and solution (c), i.e., <u>we want a standard visualization of the personal details of the instructor and a multi-objects visualization</u> of the courses' list. To this aim we need to:

- Create a **multi-objects form** as the one of Figure 2.8 (c);
- Delete all the personal details from the Body of the form;
- Add as many Text Boxes as required to the Heading of the form;
- Link these Text Boxes with the data (Name, Surname, Date of Birth, Picture, etc.) of the query.

This is shown in Figure 2.8 (d) that shows the final form in design view.

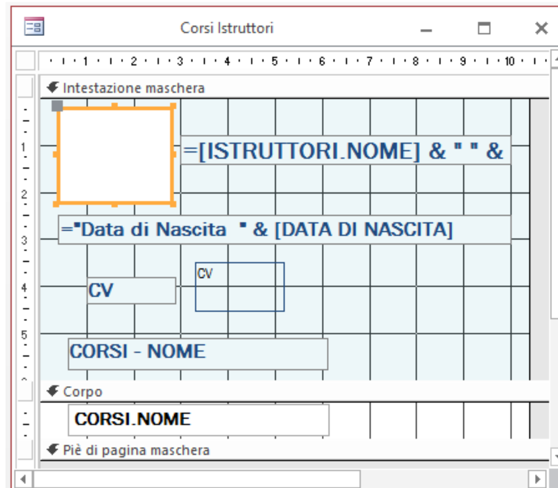

Fig. 2.8 (c) *Courses-Instructors Form in Design View*

5. Login

The login form allows users to access to their personal page, using a personal user-name and password.

Also, since the login form is shared by administrators, customers and instructors, to log in also the user typology must be defined.

For this reason, as shown in Figure 2.9, three check boxes have been inserted inside a frame (labelled as Role and called FrmRole); by doing so the checkboxes become "property of the frame" and the following benefits are obtained:

- The user can tick only a check box at a time;
- The Value property of the frame become 1 it the first check box is ticked, 2 is the second check box is ticked and so on; so, it is very easy to understand, at run time, which checkbox has been ticked by the used.

Fig. 2.9 *Login Form*

The logic of the login is very easy:

- The check boxes are used to identify on which tables the username and the password inserted by the customer must be searched;
- Next, a recordset is used to search these values, using the FindFirst(<condition>) method, which selects the first record, among those of the recordset, that satisfies the condition passed as input parameter;
- If there is a correspondence between the data of the customer and that in the database access is granted, otherwise an error message is displayed. The message depends on the kind of error:
    - missing data,
    - wrong user name,
    - wrong password,
    - wrong combination.

The full code triggered by the on click event of the OK button is shown below.

```
Private Sub CmdLogin_Click()
   Dim Rs As DAO.Recordset
   Dim Db As Database
   Dim NTable As String ' The string containing name of the table where the search must be made
   Dim WhereCond As String ' The string containing the filtering condition
   Dim NForm As String ' The name of the form (personal page) to be opened
   Dim ID As Integer
   ' An If condition, used to see which table must be opened
   If Me.FrmRole.Value = 2 Then ' If true, then the instructor check box is ticked
      NTable = "INSTRUCTORS"
      NForm = "Instructors Page"
      ElseIf Me.FrmRuolo.Value = 3 Then
         NTable = "ADMINISTRATORS"
         NForm = "Administrators Page"
      Else
         NTable = "CUSTOMERS"
         NForm = "Customers Page"
   End If
  ' Some control on the input data, in case of mistakes or missing value a message is displayed on the screen
   If IsNull(Me.TxtUserName) Then
      MsgBox "Please Insert Username", vbInformation, "User Name is needed"
      Me. TxtUserName.SetFocus ' It takes the focus back on the TxtUserName Text Box
      Exit Sub
   End If
   If IsNull(Me.TxtPassword) Then
      MsgBox "Please Insert Password", vbInformation, "Password is needed"
      Me.TxtPassword.SetFocus
      Exit Sub
   End If
   ' Now a check is made to see if data are ok
   Set Db = CurrentDb
    ' A full copy of the table in read only mode
    Set Rs = Db.OpenRecordset(NTable, dbOpenSnapshot, dbReadOnly)
   ' We write the filtering condition based on the couple user name and password
   WhereCon = "Username = "
   WhereCon = WhererCon & "'" & Me.TxtUserName & "'"
   WhereCon = WhereCon & " AND Password = '" & Me.TxtPassword & "'"
```

```
 ' FindFirst looks for (and returns) the first record that complies to the Where Condition
  Rs.FindFirst WhereCon        If Rs.NoMatch = False Then
   ' If execution proceeds here, a record has been found
    Rs.FindNext WhereCon ' FindNext looks for (and returns) the next record that fulfils the Where Condition
    If Rs.NoMatch = False Then ' If a second record exists there is a problem!!!
      MsgBox "Irreversible Data Error. Signal the problem at the reception desk", vbInformation, "ERROR"
      Me.TxtUserName.SetFocus
      Exit Sub
    End If
  Else
   ' If execution proceeds here, a record has not been found; we try to understand what's gone wrong
    WhereCon = "Username = " & "'" & Me.TxtUserName & "'"
    Rs.FindFirst WhereCon
    If Rs.NoMatch = True Then
      MsgBox "Username not found", vbInformation, "Wrong User Name"
      Me.txtNome.SetFocus
      Exit Sub
    Else
      WhereCon = "Password = " & "'" & Me.TxtPassword & "'"
      Rs.FindFirst WhereCon
      If Rs.NoMatch Then
        MsgBox "Password not found", vbInformation, "Wrong Password"
        Me.txtNome.SetFocus
        Exit Sub
      Else
        ' If the execution proceeds here, both user name and password exists, but not together
        MsgBox "Try Again, combination not found", vbInformation, "Error"
        Me.txtNome.SetFocus
        Exit Sub
      End If
    End If
  End If
  ID = Rs![ID]  ' Saves the ID of the user, the exclamation mark (!) can be used instead of the dot (.) operator
  Rs.Close
  Set Db = Nothing
  Set Rs = Nothing
  DoCmd.Close ' The Input form is closed
  DoCmd.OpenForm NForm,, , , , acFormAdd, , ID    ' An OpenAr = ID is passed to the opening form
End Sub
```

We conclude this section, making some further comments of the following code:

```
WhereCon = "Username = "
WhereCon = WhereCon & "'" & Me.TxtUserName & "'"
WhereCon = WhereCon & " AND Password = '" & Me.TxtPassword & "'"
Rs.FindFirst WhereCon
If Rs.NoMatch = False Then
    Rs.FindNext WhereCon
If Rs.NoMatch = False Then
     MsgBox "Irreversible Data Error. Signal the problem at the reception desk", vbInformation, "ERROR
     Me.TxtUserName.SetFocus
     Exit Sub
End If
[…] Code continues [...]
```

At first, a Where condition is written (and it is assigned to the WhereCon string variable); this condition filter data so as to return only the records with the same user name and password inserted by the user. Clearly, if the user has inserted corrected data, only a record should fulfill the Where Condition (i.e., the couple user name and password should be unique); however, to ascertain this condition a concatenated IF … THEN … ELSE statement is used. Specifically, at first the following instruction is used to locate the first record (and possibly the sole one) that complies to the where condition:

```
If Rs.FindFirst WhereCon = False Then
```

The **FindFirst** property, in fact, finds the first record of a recordset that satisfies the condition passed as input.

Next, the following instruction is used to verify that one record has been found:

```
Rs.NoMatch
```

The **NoMatch** property is true if a record has not been found, and it is false otherwise. So, if NoMatch is false the code proceeds to see if there are other records satisfying the where condition. This is made using the following instruction:

```
Rs.FindNext WhereCon
```

The **FindNext** property, in fact, finds the next record of a recordset that satisfies the condition passed as input.

Again, the NoMatch property is used to see if a second record has been found. If so, this should never happen, an error is risen, and an error message is displayed. The code is interrupted.

6. Code Insights[2]

6.1 RecordSets

In VBA, you don't use DoCmd.RunSQL to execute a select query. Rather, you store the results of the query in an invisible cointainer called recordset.

Just for the sake of completeness, we stress the fact that Recordsets can be created using two different technologies DAO (which stands for Data Access Objects) and ADO (which stands for ActiveX Data Objects) a thing that may be cause of confusion (even the names are very similar). Originally Microsoft Access was based on DAO technology and, only later, mainly to assure compatibility with other systems, the ADO technology was added. There are many compatibilities between the two methods, so unless you are a very experienced programmer working with split database systems, the choice is almost optional, as the importance of using one method over the other is quite marginal.

Briefly we can say that the most significant difference is the ability to work with data outside of Access and the JET engine environment[3]. ADO is very efficient with outside (remote) connections, while DAO is good for manipulating local objects. So local Access databases and/or small projects should use DAO, while larger ones should use ADO.

Due to these issues, in the following we will use both technologies; initially this may create some confusion, but as you will see, properties and methods of these technologies are very similar, so you'll get accustom to both of them rather soon. We just note that, to create a recordset:

- DAO use the Database Object;
- ADO uses the Connection Object.

Anyway, creating a recordset in VBA usually takes several lines of code. As always, you have a ton of options for how to write the code. The syntax of statements that you need in order to create a recordset from one or more tables in the current database generally looks like this (for an ADO connection):

```
Dim Cnn As ADODB.Connection ' Declare a generic Connection
Dim Rs As New ADODB.Recordset ' Declare a RecordSet
Dim SQLStatement As String
   SQL = "SELECT … FROM …" ' A valid SQL statement
  Set Cnn = CurrentProject.Connection 'Create a Connection to the current database
  Rs.ActiveConnection = Cnn ' Associate the recordset to the connection i.e., where to gather data
  Rs.Open SQLstatement '  How to gather data
```

---

[2] *Code available in the Recordsets.accdb Database*

[3] *The Microsoft Jet Database Engine is a database engine on which several Microsoft products have been built. A database engine (i.e. the DBMS) is the underlying component of a database, a collection of information stored on a computer in a systematic way.*

Where:

- Cnn is a variable name of your choosing that defines the connection;
- Rs is the name that you want to give to your recordset;
- SQLStatement is a valid SQL statement that is not an action query.

More specifically we can say that:

- Dim Cnn As ADODB.Connection declares that an object named Cnn is being creating and that this object will be an ADODB connection. In other words the name Cnn shall refer to an ActiveX Data Objects Database connection.
- Set Cnn = CurrentProject.Connection, gets more specific about what Cnn is all about. It says that Cnn is the connection to that data in the database we're working on;
- Dim Rs As New ADODB.Recordset declares that the name Rs refers, from here on, to an ActiveX Data Objects Database recordset;
- Rs.ActiveConnection = cnnX tells where the Rs will find data. Indeed it sets the Rs active connection to the connection we already defined as Cnn;
- Rs.Open SQLStatement is the method of the RecordSet that defines how data have to be collected.

The full syntax for creating an ADO recordset looks like this:

```
Rs.Open SQLStatement [,Connection] [,CursorType] [,LockType]
```

Where:

- Connection is the connection (not required if you already defined the connection by using myRecordSet.ActiveConnection in code);
- CursorType defines how VBA can access records in the recordset, and how simultaneous changes to the recordsets underlying data affect the contents of the recordset by using any of the following constants:
  - ❖ adOpenDynamic: Code can freely move the cursor through the records. Other users' additions, changes, and deletions carry over to the recordset.
  - ❖ adOpenKeyset: Code can freely move the cursor through the records. Other users' additions, changes, and deletions don't carry over to the recordset.
  - ❖ adOpenStatic: The recordset contains a snapshot of data that's no longer connected to the live data in any way, so other users' changes to the underlying table or query have no effect on the recordset. VBA can move the cursor freely through the recordset.

    - adOpenForwardOnly: The cursor can scroll down through records only; additions, changes, and deletions from other users are ignored. This is preferred when VBA just needs quick, brief access to a table to search for something or to count things (and also the default setting if you don't include this argument in your .Open statement).
- LockType determines how other users' simultaneous changes to the table or query are handled. The more commonly used constant names and lock types are listed here:
    - adLockOptimistic: Indicates optimistic locking, where records are locked only when you call the .Update method in your VBA code.
    - adLockPessimistic: Indicates pessimistic locking, where records are locked automatically after a change (without calling the .Update method).
    - adLockReadOnly: Indicates read-only records, whereby no changes are allowed to data in the recordset.

Here's a simple example: the code creates a forward-only, read-only recordset that gets its records from a table named Customers:

```
Dim Rs As New ADODB.Recordset
Rs.Open "CUSTOMERS", CurrentProject.Connection, adOpenForwardOnly, adLockReadOnly
```

The syntax for ADO recordsets also allows one to specify optional arguments individually, using the syntax:

```
recordSetName.property = value
```

For example, the following lines create a recordset that connects to the current database (CurrentProject.CurrentConnection), set the cursor type to adOpenDynamic, and set the LockType to adLockOptimistic:

```
Dim cnn As ADODB.Connection
Set cnn = CurrentProject.Connection
Dim Rs As New ADODB.Recordset
Rs.ActiveConnection = cnn1
Rs.CursorType = adOpenDynamic
Rs.LockType = adLockOptimistic
Rs.Open "SELECT * FROM CUSTOMERS"
```

A simple example, AddUpdate2, follows. Briefly, the AddUpdate2 function operates on a Table called EXAMPLE that has records made of five fields, namely: ID, Name, Surname, Date Of Birth and Age. A Recordset is opened using the ADODB technology and a DO WHILE LOOP is used to count the number of records of the table. At every cycle, for each record the Age field is added and, the name, the surname and the age is printed on the

immediate window. Lastly, an additional record is appended to the table. The total number of records is returned in output.

```
Public Function AddUpdate2() As Integer
Dim S, S1, MySQL As String
Dim Dt As Date
Dim Age, i  As Integer
Dim Cnn As ADODB.Connection
   Set Cnn = CurrentProject.Connection
Dim Rs As New ADODB.Recordset
   MySQL = "EXAMPLE"
   Rs.Open MySQL, Cnn, adOpenKeyset, adLockOptimistic, adCmdTable
   Do While Not Rs.EOF ' All records are read one by one
      AddUpdate2 = AddUpdate2 + 1
      Dt = Nz(Rs.Fields(3), #1/1/2000#)
      Age = DateDiff("yyyy", Dt, Date)
      Rs.Fields(4) = Age ' Write the age
      Rs.Update ' Update the database
      S = "Record & " AddUpdate & " " &  Rs.Fields(1) & "-" & Rs.Fields("Surname") & " Age = " & Rs.Fields(4)
      Debug.Print S  ' Print on the immediate window name surname and age
      Rs.MoveNext
   Loop
   AddUpdate2 = AddUpdate2 + 1 ' Total number of records
   S = ""
   S1 = ""
   For i = 1 To AddUpdate2
      S = S & "A"
      S1 = S1 & "B"
   Next i
   ' New values, except the age
   Dt = DateAdd("yyyy", 1, Dt)
   Rs.AddNew
   Rs.Update
   Rs.MoveLast
   Rs.Fields("Name") = S
   Rs!Surname = S1 'Another way to reference a field
   Rs.Fields(3) = Dt
   Rs.Update
   Rs.Close
   Set Cnn = Nothing
   Set Rs = Nothing
End Function
```

The same code can be obtained using the DAO technology. As we have seen before, in this case we need to write something like this:

```
Public Function AddUpdate() As Integer
Dim S, S1, MySQL As String
Dim Dt As Date
Dim Age, i  As Integer
Dim Cnn As ADODB.Connection
   Set Db = CurrentDB
Dim Rs As Recordset2
   Set Rs = Db.OpenRecordset(“EXAMPLE", dbOpenDynaset)
   Do While Not Rs.EOF ' All records are read one by one
      AddUpdate2 = AddUpdate2 + 1
      Dt = Nz(Rs.Fields(3), #1/1/2000#)
      Age = DateDiff("yyyy", Dt, Date)
     Rs.Edit  ' Edit mode is needed
     Rs.Fields(4) = Age
      Rs.Update
      S = "Record & " AddUpdate & " " &  Rs.Fields(1) & "-" & Rs.Fields("Surname") & " Age = " & Rs.Fields(4)
      Debug.Print S  ' Print on the immediate window name surname and age
      Rs.MoveNext
   Loop
   AddUpdate = AddUpdate + 1 ' Total number of records
   S = ""
   S1 = ""
   For i = 1 To AddUpdate
      S = S & "A"
      S1 = S1 & "B"
   Next i
   ' New values, except the age
   Dt = DateAdd("yyyy", 1, Dt)
   Rs.AddNew
   Rs.Update
   Rs.MoveLast
   Rs.Fields("Name") = S
   Rs!Surname = S1 'Another way to reference a field
   Rs.Fields(3) = Dt
   Rs.Update
   Rs.Close
   Set Db = Nothing
   Set Rs = Nothing
End Function
```

6.2 Managing RecordSets

After the .Open method has been executed, the recordset contains the fields and records specified by the table or SQL statement in the .Open statement.

After the recordset exists in code, you can use numerous methods of the ADODB recordsets to move the cursor through the recordset. The syntax is generally:

```
myRecordSet.method
```

Where: *myRecordSet* is the name of the recordset on which the method should be performed followed by a dot (.) and a valid method.

The cursor type of the recordset puts severe restrictions on which methods you can use. For maximum flexibility, use the adOpenDynamic cursor type option, described earlier in this chapter.

In this case the following methods can be used:

- myRecordSet.MoveFirst: Moves the cursor to the first record in the recordset
- myRecordSet.MoveNext: Moves the cursor to the next record in the recordset
- myRecordSet.MovePrevious: Moves the cursor to the previous record in the recordset

In addition to the preceding methods, you can use the BOF (Beginning of File) and EOF (End of *File*) properties to determine whether the cursor is pointing at a specific record.

For example, the following statement returns True only if the cursor is sitting above the first record in the recordset: myRecordSet.BOF

The following statement returns True only if the cursor is already past the last record in the set (pointing at nothing): myRecordSet.EOF

Seek Method and the Index Property

Generally, records in “base tables” aren’t stored in any particular order. To speed up the search, one or more field can be indexed so that the search can be performed on that particular field(s). Although indexed fields are generally unique, this is not strictly necessary. We also remember that, fields can be indexed only when a Table is created. What you can do at run time is to define which, if any, of the indexes of the table will be used to speed up the search. To this aim you have to set the Index property, which changes the order of records returned from the database without affecting the order in which the records are stored.

To perform a search on indexed field you need to use the **Seek** method (which, of course, requires that you have set the current index with the Index property), whose syntax is as follows:

expression.Seek(Comparison, Key_1, Key_2, Key_3,…, Key_{13})

Where:

- expression is a variable that represents a recordset object
- comparison (string) is one of the following string expressions: <, <=, = , >=, >
- key (variant) is one or more values corresponding to fields in the recordset object's current index

If the index identifies a unique field then Seek locates that specific field. Otherwise, in case of a non-unique key field, Seek locates the first record that satisfies the criteria. Indeed, the Seek method searches through the specified key fields and locates the first record that satisfies the criteria specified by comparison and key_1. It is obvious that the key_1 argument must be of the same field data type as the corresponding field in the current index. For example, if the current index refers to a number field (such as Employee ID), key_1 must be numeric. Similarly, if the current index refers to a Text field (such as Last Name), key_1 must be a string.

Once the record is found, Seek makes that record current and sets the NoMatch property to False. If the Seek method fails to locate a match, the NoMatch property is set to True, and the current record is undefined.

If comparison is equal (=), greater than or equal (>=), or greater than (>), Seek starts at the beginning of the index and searches forward. If comparison is less than (<) or less than or equal (<=), Seek starts at the end of the index and searches backward.

If there are duplicate index entries at the end of the index, Seek starts at an arbitrary entry among the duplicates and then searches backward.

Also note that you must specify values for all fields defined in the index. If you use Seek with a multiple-column index, and you don't specify a comparison value for every field in the index, then you cannot use the equal (=) operator in the comparison. That's because some of the criteria fields (key_2, key_3, and so on) will default to Null, which will probably not match. Therefore, the equal operator will work correctly only if you have a record which is all null except the key you are looking for. It is recommended that you use the greater than or equal (>=) operator instead.

Lastly we note that, you can't use the Seek method on a linked table because you cannot open linked tables as table type Recordset objects.

The following codes show two examples of using a dbOpenTable recordset and searching on both a single index and composite indexes. Please note that each table has a default index called PrimayrKey, in order to create additional indexes a table must be opened in design view and next, using the Index tab, it is possible to create new indexes by indicating a name and the name of the field (or of the fields) that are used as index. For instance, in Figure 2.10 an index called FN has been created using the Name and the Surname fields.

Indici: EXAMPLE

	Nome indice	Nome campo	
	FN	Name	Crescente
		Surname	Crescente
	PrimaryKey	ID	Crescente

Fig. 2.10 *Creating indexes*

```
Public Function FullDesc(ID As Integer) As String
' The Primarykey index is used to find the record corresponding to the ID passed by the user. This is made using
' the .seek property. The function returns a string containing all the data of the matching record
Dim db As Database
Dim rst As Recordset
Dim ind As Index
Dim tdfEmployees As TableDef ' An object that contains all the features of a table
  Set db = CurrentDb
  Set tdfExample = db.TableDefs("Example")
  'Each table has an index called Primary Key, in this table there is also an index based on two fields called FN
  ' This code shows the name of the available indexes
  For Each ind In tdfExample.Indexes
    Debug.Print ind.Name
  Next ind
    Set rst = db.OpenRecordset("EXAMPLE", dbOpenTable)
' comparisons allowed are <, <=, =, >=, or > up to 13 key values can be searched
  With rst
    .Index = "PrimaryKey" ' The PrimaryKey is used as current index
    .Seek "=", ID ' k1 = ID i.e., the input value
    If Not .NoMatch Then ' Double negation i.e., if found then
      Debug.Print !ID, !Name, !Surname, ![Date Of Birth], !Age
      FullDesc = !Name & " " & !Surname & " " & ![Date Of Birth] & " Age: " & !Age
    Else
      FullDesc = "Not Found"
    End If
  End With
End Function
```

Two new features have been used in this code.

The first is the **With … End With** condition. This is just a short cut that may be useful when operating on objects. As we know to access methods and/or properties of an object we need to write the name of the property or method after the name of the object.

For example we could write:

```
rst.Index = "PrimaryKey" ' The PrimaryKey is used as current index
rst.Seek "=", ID ' k1 = ID i.e., the input value
```

However, if we place these instructions inside the With End With condition, we can avoid to wirte, all the times, the name of the recordset. That is:

```
With rst
    .Index = "PrimaryKey"
    .Seek "=", ID
End With
```

The second on is the *For Each In Loop looping condition*. Any time we have a collections of objects (for example rst.fields is the collecton of all the fields of the recordses) we can loop on them all using the For Each In Loop. In the above code we created a TableDef object and a variable of type index. Since a TableDef object contains all the features of a table, it also contains the lists of all the indexes of the table. This list can be read using the Indexes collection. So if we write:

```
For Each ind In tdfExample.Indexes
        Debug.Print ind.Name
Next ind
```

we will ciclate on all the indexes contained in the Indexes collection of the tdfExample TableDef object. The name of each index will be printed on the immediate screen thanks to the Debug.Print ind.Name property.

The next example shows the use of a double fields index.

```
Public Function FullDescs(N, S As String) As String
' The FN index is used to find the record corresponding to the Name and Surname passed by the user. This is made using ' the .seek property. The function returns a string containing all the data of the matching record
Dim db As Database
Dim rst As Recordset
   Set db = CurrentDb
   Set rst = db.OpenRecordset("EXAMPLE", dbOpenTable)
   With rst
      .Index = "FN" ' The FN is used as current index
      .Seek "=", N, S ' k1 = N, k2 = S
      If Not .NoMatch Then ' Double negation i.e., if found then
         Debug.Print !ID, !Name, !Surname, ![Date Of Birth], !Age
         FullDesc2 = !Name & " " & !Surname & " " & ![Date Of Birth] & " Age: " & !Age
      Else
         FullDesc2 = "Not Found"
```

```
    End If
  End With
End Function
```

Find and Move methods

In order to locate a record in a dynaset or snapshot-type recordset that satisfies a specific condition that is not covered by existing indexes, you can use the Find methods. To include all records, not just those that satisfy a specific condition, use the Move methods to move from record to record.

You can move through a Recordset by using MoveFirst, MoveNext, MovePrevious, MoveLast, and Move (this is a relative movement from the current cursor position and can be offset from a bookmark).

You can also use AbsolutePosition and PercentPosition, both of which can display the current positional information and move to an absolute position, based on a row number or percentage of total rows.

The recordset also has a RecordCount property, but you need to ensure that you perform a MoveLast before relying on its value to represent the total number of records in the Recordset. This is because the Recordset does not normally read to the end of the dataset when it is opened (in order to maintain good performance).

When you open a recordset, if it has no records, then the BOF and EOF properties are True, if the Recordset contains data, BOF is true only when you MovePrevious on the first record, and EOF is true only if you MoveNext of the last record (you can think of BOF and EOF as parking spots beyond the beginning and ending records; when the cursor (current position) is on BOF or EOF you cannot refer to any field values in the Recordset row).

The MoveFirst() and MoveLast() methods allow you to navigate one record at a time until you get to a certain record. If you are positioned at a certain record and you want to jump a certain number of records ahead or you want to move back by a certain number of records, you can call the Move() method. Its syntax is:

```
recordset.Move NumRecords, Start
```

Where:

- NumRecords (required argument) specifies the number of records to jump to. If you pass a positive value, the position would be moved ahead by that number.
- Start is the starting position (of the jump)

Here is an example. This function receives two inputs an age (Ag) to be searched in the Example table and an integer value (m). It searches all the records with Age = Ag. The name of all these records are appended to a string. Next the code moves downward of *m* records. If this record exists its name is also appended to the

string; otherwise the name of the last record is appended to the string. Lastly the String is used as the output of the function.

```
Public Function FindMultiple(Ag, m As Integer) As String
' How to move along records
Dim db As Database
Dim rst As Recordset
   Set db = CurrentDb
   Set rst = db.OpenRecordset("EXAMPLE", dbOpenSnapshot, dbReadOnly)
   rst.FindFirst ("Age = " & Ag)
   Do While Not rst.NoMatch ' If something is found this part is executed
      FindMultiple = FindMultiple & " " & rst!Name
      rst.FindNext ("Age = " & Ag)
   Loop
   rst.move (m) ' We move downward of m records
   If Not rst.EOF Then ' If the end of the table has not been reached, we can read the values of the record
      FindMultiple = FindMultiple & " " & rst!Name
   Else
      rst.MoveLast
      FindMultiple = FindMultiple & " " & rst!Name
   End If
rst.Close
Set db = Nothing
Set rst = Nothing
End Function
```

Bookmarks

Bookmarks are defined as variables that you can use to mark a position and later return to that same position. The values are valid only as long as the Recordset is held open. They can be held in string or variant variables. There is also a property called Bookmarkable, with which you can check whether the Recordset supports bookmarks (linked tables to Microsoft Excel, TextFiles, and SQL Server are normally bookmarkable).

Here is an example. The procedure takes the initial of the surname, finds the first record with a surname that start with the initials passed by the user and prints the corresponding ID. It also places a bookmark to this record. Then it moves to the last record, before going back to the bookmarked field.

```
Public Sub Bookm(Initial As String)
' example of recordset movement using bookmarks
Dim db As Database
Dim WhCond As String
Dim rst As Recordset
Dim bk As String
   Set db = CurrentDb
   Set rst = db.OpenRecordset("EXAMPLE", dbOpenDynaset)
' Bookmarks
   WhCond = "Surname like '" & Initial & "*'"
   rst.FindFirst WhCond
   Debug.Print "Recod ID is " & rst!ID
   bk = rst.Bookmark ' Place the bookmark
   rst.MoveLast
   Debug.Print "Record ID is " & rst!ID
   rst.Bookmark = bk ' Return to the bookmark
   Debug.Print "Record ID is " & rst!ID & " using the bookmark"
   rst.Close
   Set rst = Nothing
   Set db = Nothing
End Sub
```

Working with fields in a Recordsets

In this sub-section, you will see the different forms of syntax that can be used when working with fields in a recordset.

When referencing a field that includes a space in its name, enclose the name in square brakets [...].

When referencing a field in a recordset, you can use either the "." (period) or "!" (exclamation) character if you are specifying the field by name.

For example, the *Company* field in the Recordset rst could be referenced by using the following syntax:

```
rst!Company
rst.Company
rst.Fields(lngFieldNo)
rst.Fields("Company")
rst.Fields(strFieldName)
```

An extended example follows:

```
Sub modRST_FieldSyntax()
  ' Example of field syntax
  Dim db As Database
  Dim rst As Recordset
  Dim strField1 As String
  Dim strField2 As String
  Dim fld As Field
  strField1 = "ID"
  strField2 = "[Company]"
  Set db = DBEngine(0)(0)
  Set rst = db.OpenRecordset("qryCustomersInCA", dbOpenDynaset)
    With rst
      Do While Not .EOF
          Debug.Print !ID, !Company
          Debug.Print ![ID], ![Company]
          Debug.Print .[ID], .[Company]
          Debug.Print .Fields(0), .Fields(1)
          Debug.Print .Fields("[ID]"), .Fields("Company")
          Debug.Print .Fields(strField1), .Fields(strField2)
          rst.MoveNext
      Loop
    End With
  rst.MoveFirst
  For Each fld In rst.Fields
    If fld.Type <> dbAttachment Then
      Debug.Print fld.Name, fld.Value
    End If
  Next
End Sub
```

Filtering and Sorting and the BuiltCriteria methods

The Filter and Sort properties of a recordset might **not be what you imagine** them to be. The properties do not change the data in the recordset; instead, they change the properties that will be applied when you open another recordset, based on the current recordset (note that these properties are not applied when you clone a recordset).

In the following example, *rst2* is opened with the Filter and Sort properties specified on *rst*. Specifically, *rst2* will contain only the records (of the original table) with an ID greater than the value passed as input by the user; these records will be ordered in descending order depending on the surname field of the record.

The full code is shown below.

```
Public Sub FilterAndSort(ID As Integer)
' example using Filter and Sort properties
Dim db As Database
Dim rst As Recordset
Dim rst2 As Recordset
   Set db = CurrentDb
   Set rst = db.OpenRecordset("Example", dbOpenDynaset)
   ' Apply a filter and sort
   rst.Filter = "ID >= " & ID
   rst.Sort = "Surname DESC"
   ' next demonstrating sort and filter properties
   Set rst2 = rst.OpenRecordset ' rst2 is opened on rst, thus rst2 will be filtered and sorted :)
   rst2.MoveFirst
   Do While Not rst2.EOF
      Debug.Print rst2!Surname
      rst2.MoveNext
   Loop
End Sub
```

Adding new records and cloning

When adding a new record, you call the AddNew method on the recordset, and then you call the Update method to save the new record.

A useful additional statement is to set the Bookmark for the Recordsets to the LastModified property; this resynchronizes the cursor to point of the newly added record.

When updating a record, you call the Edit method. To save the changes, call the Update method.

To delete a record, call the Delete method. Note that the cursor will not be pointing at a valid record following this operation; therefore, you might need to use a Move command to set the cursor to a valid position:

```
Sub modRST_AddEditDelete()
  ' Examples of Adding, Editing and Deleting with a Recordset
  Dim db As Database
  Dim rst As Recordset
  Set db = CurrentDb
  Set rst = db.OpenRecordset("Customers", dbOpenDynaset)
  ' Stop
  ' We add a new record
  rst.AddNew
  rst!Company = "Company Z"
  rst.Update
```

```
  ' We place the cursor on the newly added record
  rst.Bookmark = rst.LastModified  ' We update a record
  rst.Edit
  rst![Last Name] = "Bedecs"
  rst.Update
  ' The cursor will still be on this record
  ' We delete a record
  rst.Delete
  ' The cursor is now invalid
End Sub
```

Rather than create a new recordset, it is sometimes more useful to create a clone of an existing recordset.

Cloning a Recordset is more efficient than creating a second Recordset and also has the benefit that Bookmarks are shared between the two Recordsets.

In the following example, a recordset (*rst*) is opened on the EXAMPLE table and a bookmark is placed at the record having the ID passed, as input by the user. Next a clone of *rst* is made (i.e., *rst2*); this clone inherits the bookmark and, indeed, printing the value of the bookmarked record we get the same value of the bookmarked record of *rst*. Then a new record is inserted in the clone, a bookmark is recorded, (using the *LastModified* property) and the original recordset is repositioned at the cursor defined inside the clone. Lastly the newly inserted record is deleted.

The full code is shown below:

```
Sub RstClone(ID As Integer)
' An example on the use of a recordset clone and of shared bookmarks
Dim db As Database
Dim rst As Recordset
Dim rst2 As Recordset
Dim bk As String
   Set db = CurrentDb
   Set rst = db.OpenRecordset("EXAMPLE", dbOpenDynaset)
   ' We clone the recordset
   Set rst2 = rst.Clone
   rst.FindFirst "ID = " & ID
   bk = rst.Bookmark 'place the bookmark
   Debug.Print rst!Surname
   ' We move the cloned recordset to same position
   rst2.Bookmark = bk
   Debug.Print rst!Surname
   With rst2
      .AddNew
```

```
        !Name = "New"
        !Surname = "NewNew"
        .Update
        .Bookmark = .LastModified  ' We lace the cursor at the newly inserted record
        bk = .Bookmark
    End With
    rst.Bookmark = bk
    Debug.Print rst!Surname
    rst.Edit
    rst.Delete 'The last is deleted
    rst2.Close
    rst.Close
    Set rst2 = Nothing
    Set rst = Nothing
    Set db = Nothing
    Set db = Nothing
End Sub
```