Microsoft-Access Tutorial

Soren Lauesen

E-mail: slauesen@itu.dk

Version 2.4b: July 2011

Contents

Printing instructions

Print on A4 paper with 2-sided printing so that text and associated figures are on opposing pages.

Version 1: October 2004.

Version 2.1: November 2004. Changes:

a. Restructured section 3.2 with small additions.
b. Section 7.1 on action queries added.
c. Small changes and additions to Chapter 6 with corresponding changes in the Reference Card.
d. Index provided

Version 2.2: April 2004. Changes:

a. SQL HAVING introduced in section 4.2 and the example in section 4.4.
b. More on aggregate functions in section 4.4.
c. ColumnOrder, ColumnWidth discussed in section 5.5.7.
d. Selection of an area in the datasheet is discussed in section 5.5.8.
e. Section 5.7 (action queries) now moved to Chapter 7.
f. Action queries, Union, Subqueries, Crosstab, etc. discussed in Chapter 7 (a new chapter).
g. Various small changes and improved explanations here and there.

Version 2.3: September 2006. Changes:

a. Access 2003 dialog when opening a database changed (page 8).
b. Look-up fields for foreign keys deleted (last part of section 2.4). Access's automatic creation of relationships caused too much confusion.
c. Combo boxes described in sections 3.1.6 and 3.1.7.
d. More events explained in section 5.2.3.
e. Various misprints corrected.

Version 2.4: August 2007 and July 2011. Changes:

a. Partial integrity (page 12).
b. Adding a label to a control (page 20).
c. DateTime Picker (page 22).
d. More Null rules (page 62, 77, 124).
e. Access data model and experiments improved (page 68-70).
f. Composite search criteria, more computed SQL, date comparison (page 76-77).
g. Event sequence for textbox: small corrections, e.g. OldValue (page 78).
h. Improved area selection (page 95-96).
i. Error handling, user errors (page 97-98).
j. Timer and loop breaking (page 98-99).
k. Managing modules and class modules (page 112).
l. Error handling, VBA errors, Err object (page 117).
m. Enum type (page 121).
n. Partition operator (page 124).
o. Week number in the Format function (page 126).
p. Dynamic matrix simplified (page 136).
q. Minor corrections and improvements in many places.
r. Version 2.4a: Note on AutoNumber added to Figures 2.1C and 2.4.
s. Version 2.4b: Copyright notice more liberal. Misprint corrected (page 65, step 14 and 15). Figure 52B (page 75) shows quote-stuff more clearly. SendKeys on page 99 elaborated.

Preface

This booklet shows how to construct a complex application in Microsoft Access (MS-Access). We assume that the user interface has been designed already as a paper-based mockup (a prototype). How to design a good user interface is a separate story explained in *User Interface Design - a Software Engineering Perspective*, by Soren Lauesen.

After design, development continues with constructing the database, constructing the user interface, binding the user interface to the database, and finally develop the program. This is what this booklet is about.

The reason we illustrate the construction process with MS-Access is that it is a widely available tool. Anybody who has Microsoft Office with MS-Word, also has Access and the programming language Visual Basic behind Access.

MS-Access is also a good illustration of many principles that exist on other platforms too, for instance a relational database, a Graphical User Interface (GUI), event handling, and an object-oriented programming language. MS-Access contains all of these parts - cooperating reasonably smoothly.

Organization of the booklet

The chapters in the booklet are organized like this:

1. An introduction to the hotel system that is used as an example throughout the booklet.

2. Creating a database. Construct a database that corresponds to the data model behind the design. The user will only see the database indirectly - through the screens we construct.

3. Access-based user interfaces. Construct the screens and menus that the user will see. We follow the paper-based mockup designed in *User Interface Design*. You can use the result as a tool-based mockup.

4. Queries - computed tables. Connect the screens to the database, usually by means of queries - computed data tables. The result will be a partially functional prototype.

5. Access through Visual Basic. Program what the buttons and menus will do when the user activates them. The result will be a fully functional prototype and later the final system to be delivered to the customer. The first part of the chapter is tutorial - mandatory reading if you want to work with Visual Basic and Access. The rest of the chapter is for looking up various subjects. We assume you know a bit of programming already.

6. Visual Basic reference. A reference guide to the Visual Basic language for Applications (VBA).

7. Access and SQL. An overview of the remaining parts of SQL, for instance how to update the database through SQL. We also explain how to generate matrices of data with dynamically changing headings.

Using the booklet for teaching

We have experimented with using the booklet for teaching. First we tried to present part of the material with a projector, then let the students try it out on their own, next present some more, etc. Although the students listened carefully, it turned out to be a waste of time, partly because the students worked with vastly different pace.

Now we give a 15 minute introduction to the main parts of Access: the database window, the tables, the forms - and how they relate to what they have learned in user interface design. Then the students work on their own. We have instructors to help them out when they get stuck.

The hotel system

We have chosen to illustrate the construction process with a hotel example, because most people have an idea what it is about, yet it is sufficiently complex to show typical solutions in larger systems. Some of the complexities are that a hotel has many types of rooms at different prices; a guest can book several rooms, maybe in overlapping periods; a room may need renovation or repair, making it unavailable for a period; the hotel keeps track of regular guests and their visits over time.

Simplifications

However, we have simplified the system in many other ways to shorten the discussion. For instance we ignore that in most hotels, rooms are not booked by room number, but by room type; hotels usually overbook, i.e. book more rooms than they have, expecting that some customers will not turn up. We also ignore all the other aspects of operating a hotel, for instance keeping track of when rooms are cleaned and ready for the next guest, purchasing goods, planning who is to be on duty for the next weeks, payroll and general accounting. In spite of these simplifications, the example still shows the structure of larger systems.

On-line resources

A demo-version of the hotel system, a VBA reference card, etc. are available from the authors's web site: www.itu.dk/people/slauesen. Comments are welcome.

Soren Lauesen, slauesen@itu.dk

1. The hotel system

In this booklet we illustrate MS-Access by means of a system for supporting a hotel reception. The system is used as the main example in *User Interface Design - a Software Engineering Perspective*, by Soren Lauesen. If you know the book, skip this section and go straight to Chapter 2.

Screens

The hotel system consists of the screens shown in Figure 1A.

Find guest. The Find guest screen allows the receptionist to find a guest or a booking in the database. The receptionist may enter part of the guest name and click the *Find guest* button. The system then updates the lower part of the screen to show the guests or bookings that match. The receptionist may also find the guest by his phone number, room number, or stay number (also called *booking number*).

The receptionist can select the guest from the list and click the buttons to see details of the booking or create a new booking for the guest.

Room Selection. The Room Selection screen gives an overview of available rooms in a certain period. Availability is shown as IN when the room is occupied, BOO when it is booked, etc. The receptionist may specify the period of interest and the type of room, then click the *Find room* button. The system updates the table at the bottom of the screen to show the rooms of interest. The receptionist can then choose a room and book it for the guest – or check a guest into the room.

Stay. The Stay screen shows all the details of a booking, for instance the guest and his address, the rooms he has booked and the prices. When the guest is checked in, the Stay screen also shows breakfast and other services he has received. The system shows these details on the *Services* tab. Here the receptionist can record services that the guest has received. The system uses the term *Stay* to mean a booking or a guest who has checked in.

Breakfast list. The Breakfast screen shows the breakfast servings for a specific date. It handles just two kinds of breakfast: self-service breakfast in the restaurant (buffet) and breakfast served in the room. The waiter in the restaurant has a paper copy of the list and records the servings here. Later the receptionist enters the data through the Breakfast screen.

Service list. The Service list shows the price for each kind of service. Hotel management uses this list to change service prices or add new kinds of service.

Database

The system uses a database with several tables. They are shown as an E/R data model on Figure 1B.

tblGuest has a record for each guest with his address and phone number.

tblStay has a record for each stay (booking or checked in) with a booking number (*stay number*) and the pay method.

tblRoom has a record for each room in the hotel.

tblRoomState has a record for each date where a room is occupied. It connects to the room occupied and the stay that occupies it. If the room is occupied for repair, it doesn't connect to a stay.

tblRoomType has a record for each type of room (room class) with a short description of the room type, the number of beds, and the prices.

tblService has a record for each type of service with its name and price per unit.

tblServiceReceived has a record for each delivery of service to a guest. It connects to the type of service and to the stay where the service is charged (there is an invoice for each stay).

Fig 1B. Tables as E/R model

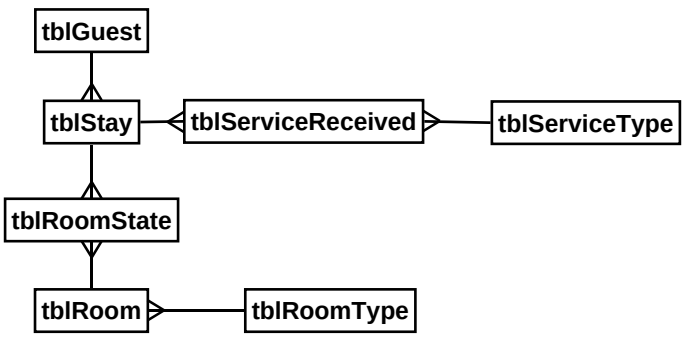
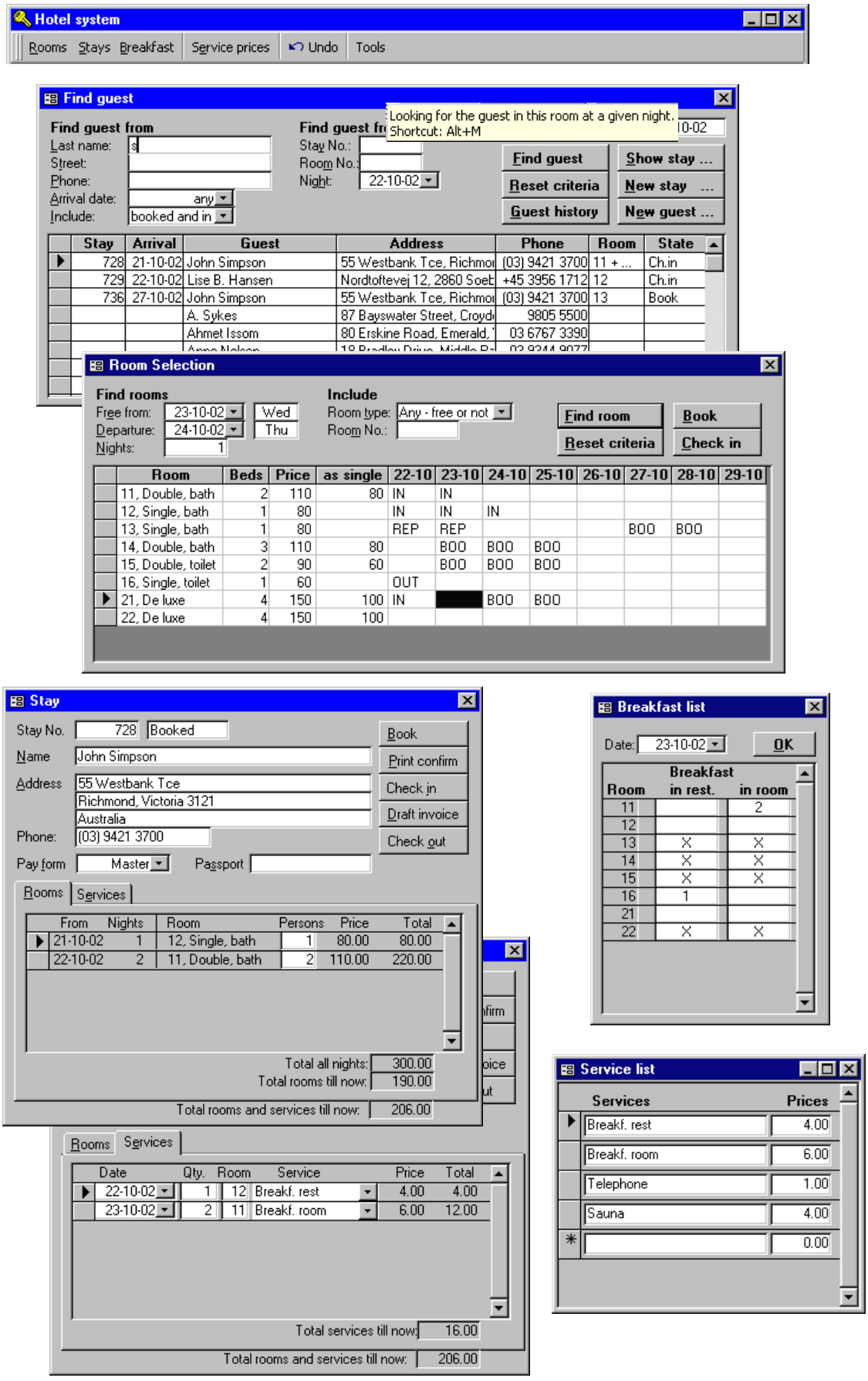


Fig 1A. Hotel system screens



2. Creating a database

Highlights

- Transform the data model to a database in MS-Access.
- Use lookup-fields to enter foreign keys and enumeration types.

In this chapter you learn how to realize a data model as a relational database in Microsoft Access. We assume that you know about data modeling, tables, attributes, and foreign keys as explained in *User Interface Design*. The description below is based on Access 2000, but there are only small differences from Access 97 and Access 2003. We will mention the more important ones.

In this and the following chapters we will use the hotel system as an example, and you will construct several parts of the system. However, the purpose is not to construct the hotel system, but to show how MS-Access works. This knowledge will enable you to construct a functional version of your own system - for instance the one you have designed when reading *User Interface Design.*

2.1 Create a database in Access

In Microsoft Access a database consists of one single file. The file contains all the tables of the database, the relationships (the crow's feet), *queries* (computed tables), *forms* (user windows), and many other things.

As a systems developer you will *design* tables and user windows. As a user you will enter data into the tables (usually through user windows) and get data out of the tables, for instance through the same windows or through printed reports.

In Access it is very easy to switch between the developer role and the user role. As a developer you will typically design some tables, then switch to the user role to enter data into them, then switch back to the developer role to change the design, design more tables, etc. Access can to a large extent restructure the data that already is in the database so that it matches the new table design.

Warning: Make sure you follow the steps below closely. **Don't skip any of the numbered steps**. The result might be that you get stuck later in the text.

Create the database

1. Locate the Access program. Depending on the way the system is set up, you may find it under Programs -> Microsoft Access or Programs -> Microsoft Office -> Microsoft Access.
2. In Access **97** and **2000**: Open Access and ask for a "blank" database.
   In Access **2003**: Open Access and click the *New* icon (under the *File* menu). Then click *Blank database* in the help area to the far right.
3. Access now asks where to store the new database. Select the folder you want and give the database the name **hotel** (or **hotel.mdb**).

The screen now shows the *database window*. It should look like Figure 2.1A. (In Access 97 it looks slightly

Fig 2.1A The Access database window

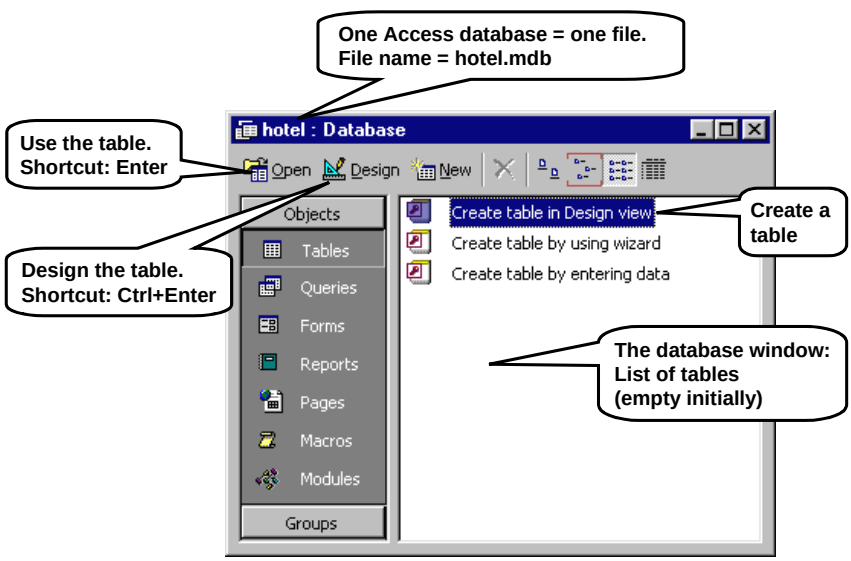
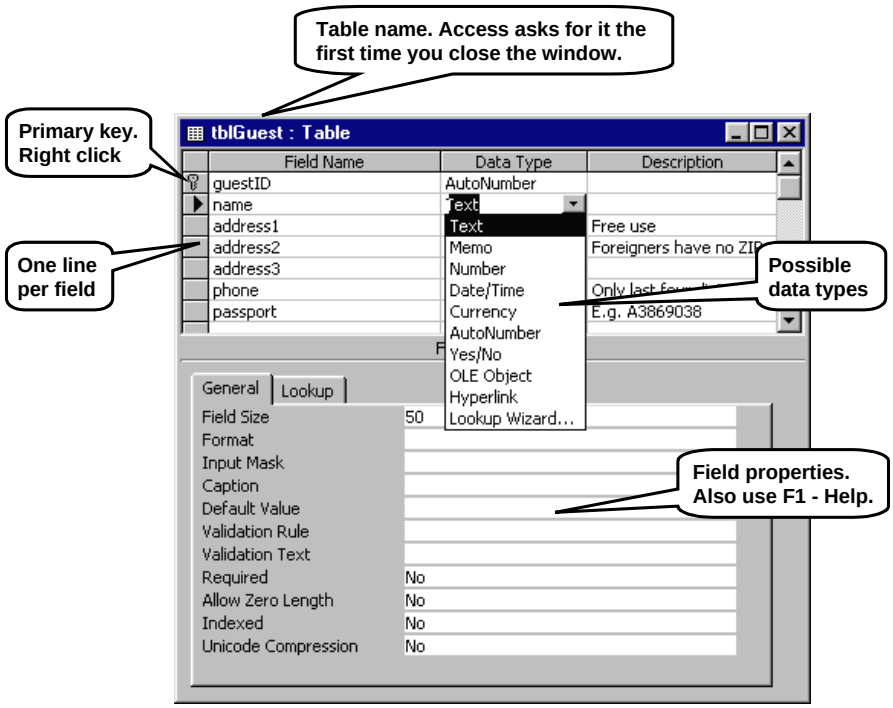


Fig 2.1B Define a table (design view)



different). We have selected the *Tables* tab, but there are no tables or other things in the database as yet. However, you see three icons that can create tables for you. When you have created a table, it will appear in the table window and you can then *Open* it and enter data into it, or you can *Design* it, i.e. change the definition of it. (In Access 97 the database window looks like a traditional tab form. There are no create-icons, but function buttons for the same purpose.)

Define a table

4. Double click on *Create table in Design view*.

Now you get a window as shown on Figure 2.1B. Here you define the fields (attributes) of the table. The list of fields runs downwards with one line per field. Initially there are only empty lines. The table hasn't got a name yet. Access asks for the name when you close the window.

The figure shows the finished guest table. You see the field names to the left. In the middle column is the type of the field - Data Type. The figure shows all the possible types as a combo box. The most important data types are Text, Number, Date/Time, and AutoNumber. An **AutoNumber** is a counter that Access increases for each new record, so that it serves as a unique key. The value is a *Long Integer* (32-bit integer). We explain more about data types in the next section.

5. Fill in all the field lines according to the attributes in the guest table (see the figure). All the fields are of type Text, except the guestID which is of type AutoNumber.

Note that although we say phone *number* and passport *number*, these fields are texts because the "numbers" contain parentheses, dashes and maybe letters.

When you have chosen a data type, you can choose a number of other field properties. They are in the lower part of the window. On the figure you can see that the *name* field is a text field with space for 50 characters. You can also see that the user doesn't have to enter anything in the name field (*Required*=No). You should change this to *Yes* since it doesn't make sense to have a guest without a name.

Try to use Access's help to find more information about the data types and their properties. For instance, put the cursor in the Data Type of a field and click F1. Or point at one of the properties and click F1.

Lookup Wizard is not a field type. If you select Lookup Wizard, it makes the field into a combo box where the user can *select* a value instead of typing it into the field. We will look closer at Lookup in section 2.4.

Key fields

Often you have to define a key field so that other tables can refer to this one. In our case, guestID must be the key field:

6. Right-click somewhere in the guestID line. Then select *Primary Key*. Access now shows that the field is the key.

You can remove the key property again by once more selecting *Primary Key*. If the key consists of more than one field, you first select all the fields by clicking on their left-hand marker with Ctrl down. Then select Primary Key by right-clicking *inside* one of the field lines.

7. Close the window. Access asks you for the name of the table. Call it **tblGuest**. (The prefix *tbl* will help you remember that it is a table. As the system grows, there will be guest windows, guest buttons and many other things. Without discipline on your part, it becomes a mess.)

If you have not defined a primary key, Access will warn you and suggest that it makes one for you. Don't let it - do it yourself. Or at least check what Access makes in its excessive helpfulness.

Enter data
After these efforts, it is time to record some guests. Fortunately it is easy:

8. Select the guest table in the database window. Click *Open* or just use *Enter*.

Now the system shows the table in **user mode** (*Datasheet view*) so that you can enter guest data.

9. Enter the guests shown on Figure 2.1C. You add a new guest in the empty line of the table - the one marked with a star. Notice that as soon you start entering something, the record indicator changes to a pencil and a new star line appears. The pencil shows that you are editing the record, and the record you see is not yet in the database.

On Figure 2.1C we originally entered a guest that got guestID 4, later deleted this guest. Access will never reuse number 4 for a guest.

Close and reopen the database
To feel confident with Access, it is a good idea to close and open the database now.

10. Close the large Access window. (Not the small database window inside the Access window.)

Notice that Access doesn't ask whether you want to save changes. Access saves them all along, for instance when you define a table or when you enter a record in the table.

11. Find your database file (*hotel.mdb*) in the file folders. Use Enter or double click to open it.

Access **2003** is very security concerned and asks you several questions when you open the file. The dialog may vary from one installation to another, but is something like this:

12. *The file may not be safe. Do you want to open it*? Your database is safe, so answer *Open*.
13. *Unsafe expressions are not blocked. Do you want to block them?* You want full freedom, so answer *No.*
14. Access warns you one more time whether you want to open. Say *Open* or *Yes*. (In some versions the question is a very long text box, and you cannot understand it. Say yes anyway.)

As an alternative, you may say yes to blocking the unsafe expressions. This will save you some questions when you open the file in the future. However, some installations don't allow you to block expressions.

Note that Access 2003 shows that your database is in Access 2000 format. This is all right. It allows you to use it also from Access 2000. You can convert it to other formats with Tools -> Database Utilities -> Convert Database.

Undo. Use Esc to undo the changes you have made to the current record.

- The first Esc undoes changes to the *field* where the cursor is.
- The second Esc undoes all changes to the *record* where the cursor is.

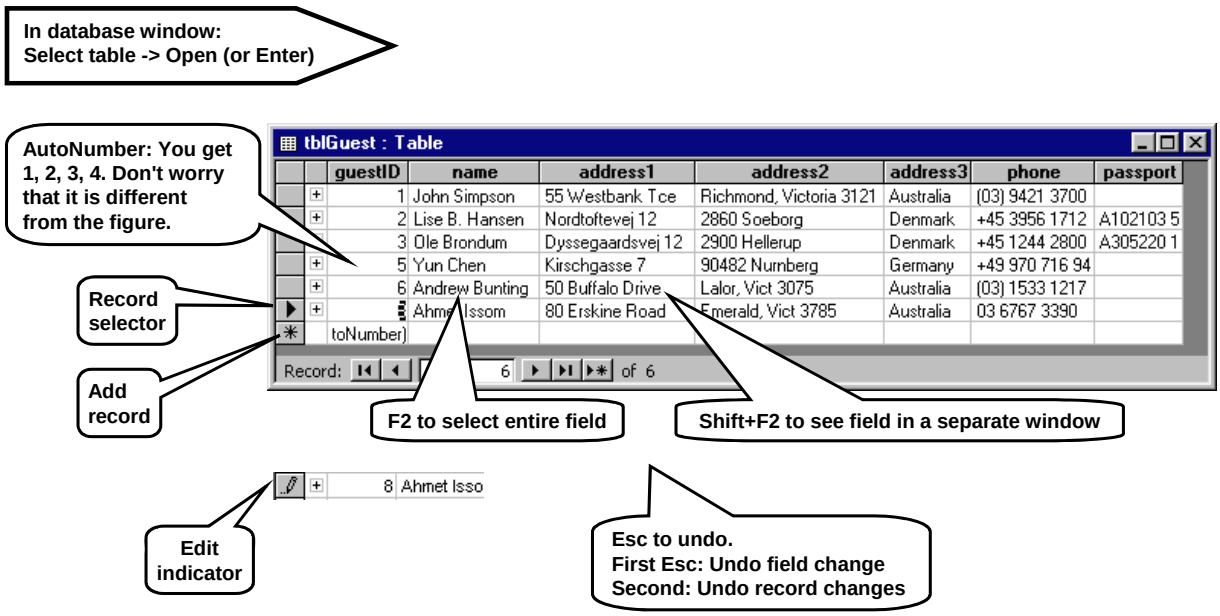
As soon as you move the cursor to the next line, Access stores the record in the database and you cannot make an automatic undo anymore. However, you can manually edit the stored record. Notice that the pencil disappears when the record is stored in the database.

Shortcut keys for data entry
F2: Toggles between selecting the entire field and selecting a data entry point.
Shift+F2: Opens a small window with space for the entire field. Useful for entering long texts into a field that is shown only partly in the table. However, the text cannot be longer than you specified in the table definition.
Alt+ArrowDown: Opens a combo box. Choose with the arrows and Enter.

Shortcut keys for navigation
Tab and Shift+Tab: Moves from field to field.
Ctrl+Tab: Moves from one tab form to the next, for instance in the lower part of the table definition window.
F6: Moves between upper and lower section of a window, for instance in the table definition window.
Ctrl+Enter: Opens the table in design mode (in the database window).

See also shortcuts on the reference card

Fig 2.1C Enter data in user mode (datasheet view)



2.2 Create more tables

You should now create the remaining tables for the hotel. The data model on Figure 2.2 shows the tables we will use. To simplify your job, we have shown all the keys, including the foreign keys and the artificial keys.

1. Close the guest table.
2. Create all the remaining tables in the same way as you created the guest table (from the *Tables* tab use *Create table in Design view* - or click *New*).

Make sure you define all the fields. Otherwise you will get stuck when later constructing the user interface. Here are a few notes about the various tables:

tblStay:

stayID is the primary key of tblStay. Make it an AutoNumber.

guestID is a foreign key that refers to the AutoNumber in tblGuest. The foreign key must have a matching data type - a long integer. Choose Data Type = *Number* and Field Size = *Long Integer*. **Warning:** Don't make the foreign key an AutoNumber. This would cause Access to fill in the foreign key fields automatically, and you cannot change the numbers so that they point to the primary keys in the guest table.

paymethod is an enumeration type. Make it an integer (a 16-bit integer, *not* a long integer). Choose Data Type = *Number* and Field Size= *Integer*. We will use the value 1 to denote Cash, the value 2 to denote Visa, etc. We will look closer at this in section 2.4.

state must also be an enumeration type. Make it an integer. Here the value 1 will denote *booked*, 2 *in*, etc.

tblRoomType:

Contains one record for each type of room, for instance one for double rooms, one for single rooms, etc. (In the book *User Interface Design*, we added this table late in the design process to illustrate the normalization concept.)

roomType is an artificial key. An AutoNumber is okay.

description is a short text, for instance "double room, bath".

bedCount is the number of beds in the room, including temporary beds.

price1 and price2 are the standard price and a possible discount price. The price should be a decimal number. Choose Data Type=*Number*, Field Size= *Single*, Decimal Places =2.

tblRoom:

roomID is a natural key - the number on the door. So don't use an AutoNumber. Use an integer.

roomType is a foreign key that refers to tblRoomType. (You should by know how to deal with it.)

tblRoomState:

stayID and roomID are foreign keys. Ensure their types match what they refer to. Notice that roomID refers to a natural key, not to an AutoNumber.

date should be a Date/Time field with Format = *Short Date*.

personCount is the number of persons staying in the room. An integer should suffice.

state is similar to state for tblStay, although the values are slightly different.

The key consists of two fields: roomID and date. It is a bit tricky to specify this: select both fields by clicking on the left-hand marker (hold down Ctrl while selecting the second field). Then right-click somewhere on the text *inside* the line.

Optional tables

The following two tables are needed for the full system. However, you don't need to create them in order to follow the tutorial.

tblServiceType:

serviceID is an artificial key. Should be an AutoNumber.

name and price should be obvious. The price should be a decimal number. Choose Data Type=*Number*, Field Size= *Single*, Decimal Places =2.

tblServiceReceived:

stayID and serviceID are foreign keys that refer to AutoNumbers. The foreign keys must thus be long integers.

roomID is an optional reference to a room. An integer should suffice. (This reference is needed when a waiter records a service for a specific room and the guest has more than one room.)

date should be a Date/Time field. Choose Format = *Short Date*.

quantity is the number of items the guest has got - an integer should suffice.

Data types

Data is stored in the computer according to its type. Here is a description of the most important types in the data base. Visual Basic deals with almost the same types (see section 6.2 and the reference card under *Declarations*).

Text. The field can contain any characters. The Field Size property defines the maximum number of characters. The maximum cannot be above 255 characters.

Memo. Like a text field, but the maximum number of characters is 65,535. Access takes more time to process a memo field, so use text fields if adequate.

Number. The field can contain a number. The Field Size property defines what kind of number:

- **Integer**. A small integer. It must be in the range -32,768 to +32,767 (a 16-bit integer).
- **Long Integer**. It must be in the range from around -2,140 million to +2,140 million (a 32-bit integer).
- **Single**. A decimal number in the range from $-3.4*10^{38}$ to $+3.4*10^{38}$ with an accuracy of 6 or 7 significant digits (a 32-bit floating point number).
- **Double**. A decimal number in the range from $-1.8*10^{308}$ to $+1.8*10^{308}$ with 14 significant digits (a 64-bit floating point number).
- **Decimal**. A very long integer with a decimal point placed somewhere. Intended for monetary calculations where rounding must be strictly controlled. In the book we use Single or Double instead.

Numbers can be shown in many ways depending on the format property of the field. You may for instance show them with a fixed number of decimals, with a currency symbol, etc.

Some formats show data in a way that depends on the regional settings of the computer. If you for instance specify the format of a number as *Currency*, the number will show with a $ on a US computer and with a £ on a British computer.

Date/Time. The field gives a point in time. In the computer it is stored as the number of days since 30/12-1899 at 0:00. It is really a Double number, so the number of days may include a fraction of a day. In this way the field specifies the date as well as the time with high precision. As an example, the number 1 corresponds to 31/12-1899 at 0:00, the number 1.75 to 31/12-1899 at 18:00 (6 PM).

Usually we don't show a date field as a number, but as a date and/or a time. The format property specifies this. Also here you can choose a format that adapts to the regional setting.

Yes/No. The field contains a Boolean value shown either as Yes/No, True/False, or On/Off. The format property specifies this.

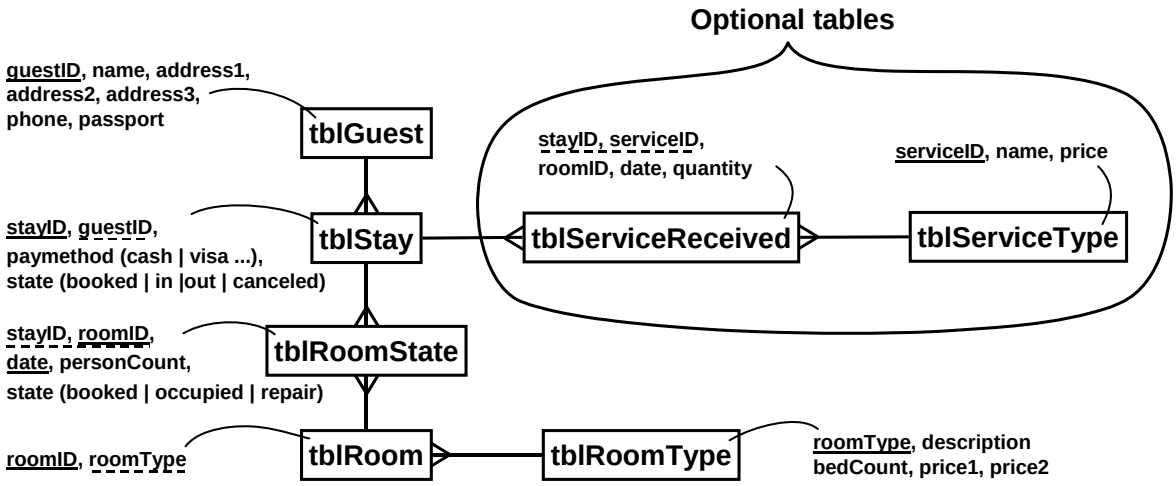
AutoNumber. The field is a long integer (32 bits) that Access generates itself as a unique number in the table. Access numbers the records 1, 2, . . . as you enter the records. However, you cannot trust that the sequence is unbroken. For instance when you add a record and undo the addition before having completed it, Access uses the next number in the sequence anyway.

A **foreign key** is a field (or several fields) that refer to something unique in another table - usually the primary key. Be careful here. The foreign key and the primary key must have the same type. However, when the primary key is an AutoNumber, the foreign key must be a long integer.

Changing a data type. Access is quite liberal with changing a data type to something else - even if there are data in the records. It can also change an AutoNumber field to a number field, but not the other way around. If you need to change field B to an AutoNumber, create a new field C and make it an AutoNumber. Then delete field B and rename field C to B.

If you for some reason want to store a record with an AutoNumber of your own choice (for instance create a stay with stayID=728), you need to append the record with an INSERT query (see section 7.1). You cannot just type in the stayID.

Fig 2.2 Create remaining tables



2.3 Create relationships

When we have several tables, we can make relationships (crow's feet). Then we get an E/R model instead of a simple collection of tables. The relationships allow Access to help us retrieve data across tables, check referential integrity, etc.

Figure 2.3 shows the hotel relationships in Access. It resembles the crow's feet model quite well. You define the relationships in this way:

1. Start in the database window and right-click somewhere.
2. Choose *Relationships*.

Now you see an empty *Relationship Window*. You have to tell Access which tables to show here. Sometimes a *Show Table* window pops up by itself. Otherwise you have to invoke it with a right-click in the relationship window.

3. In the *Show Table* window, select the tables you want to include. In the hotel system it is all the tables.
4. Click *Add* and close the window. Now the tables should be in the relationship window.
5. Create the relationship between tblGuest and tblStay by dragging guestID from one table to guestID in the other.
6. An edit-relationship window pops up. If not, right-click on the relationship connector and choose the edit window.

Access may complain:

Relationships must be on the same number of fields with the same data types.

The cause is often that one end of the connector is an AutoNumber and the other end a simple integer. It must be a long integer to match the AutoNumber.

In the edit-relationship window, you can specify foreign keys that consist of several fields. You can also specify that the relationship has referential integrity, so that all records on the m-side point to a record on the 1-side.

7. In our case, all stays must point to a guest, so mark the connector *enforce referential integrity*. (If Access refuses this, it is most likely because you have not defined the foreign key as a long integer.)
8. Close the relationship window. The relationship connector now appears in the window between the foreign key and its target.

The referential integrity makes Access show the connector as 1-∞ (1:m). Based on referential integrity and whether the connected fields are primary keys, Access may also decide that it is a 1:1 relationship. It is not important what Access decides in these matters. You can later tell it otherwise when you want to use the connector.

9. Create the remaining relationships too. Note that there is no referential integrity between tblStay and tblRoomState. It is on purpose - if the room is in repair state there is no connected stay.

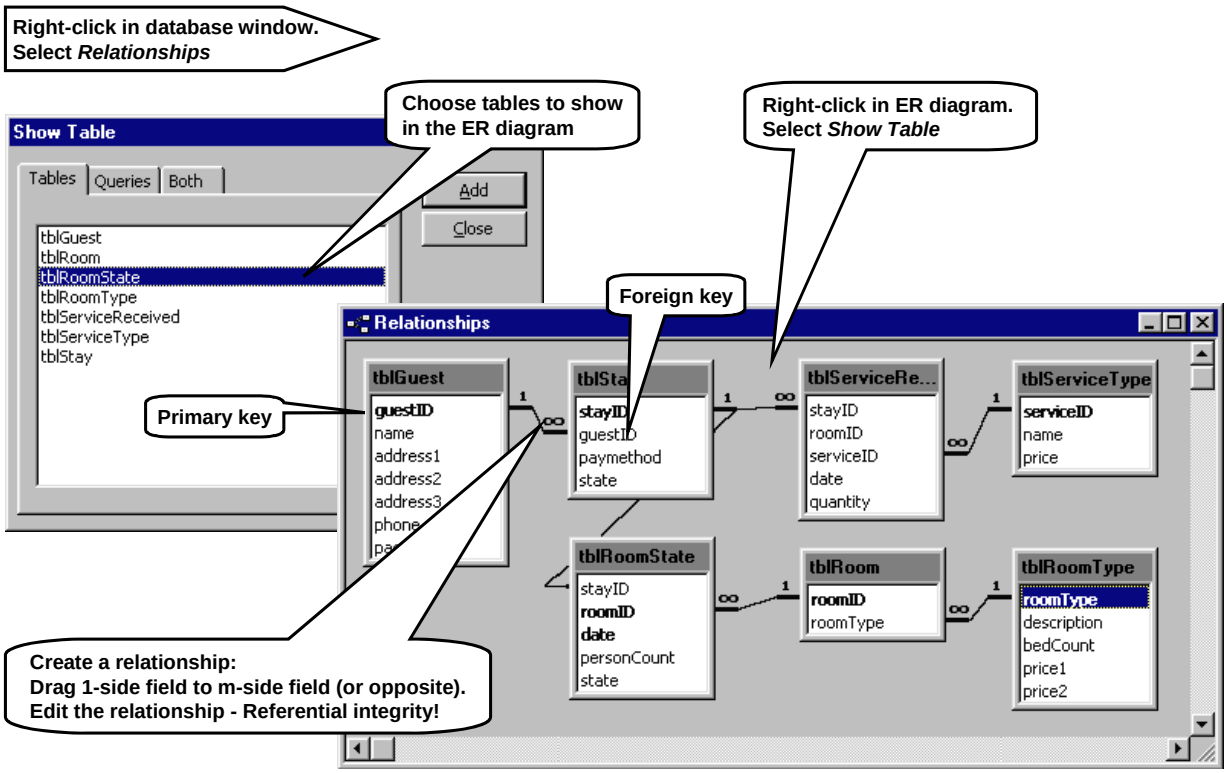
Partial integrity. Access provides a more relaxed version of referential integrity. It allows the foreign key to be either empty (Null) or point to a record on the 1-side. This is the case for the relationship between tblStay and tblRoomState. Give it partial integrity in this way:

10. Open tblRoomState in design view. For stayID (the foreign key) set the Default Value to empty (delete all characters in the field). Also set *Required* to *No*.
11. In the relationship window, right-click on the connector and choose the edit window. Select *enforce referential integrity*.

Note that you cannot see in the relationship window whether the relationship has full or partial referential integrity.

Deleting a relationship. If you need to delete a relationship, click it and press Del.

Fig 2.3 Create relationships



2.4 Look-up fields, enumeration type

Your next task will be to fill in some data in all the tables. However, some of the fields are cumbersome to fill in correctly. As an example, the pay method field is a code where 1 means Cash, 2 Visa, etc. The user should not have to remember these codes, so we will let the user choose the value from a list. It is an enumeration-type field:

paymethod(Cash | Visa | . . .)

Figure 2.4 shows what we want when the user fills in the *paymethod* field. We want the field to be a combo box where the user can select the mnemonic text while Access stores the number code. Here is how to do it:

1. Open tblStay in design view. (Select it and click Design or use Ctrl+Enter).
2. Select the *paymethod* field and the data type *Lookup Wizard*.
3. Access asks whether you (as a user) want to select the values from a table or from a list of values that you (as a designer) type in. Choose to type them in. Then click *Next*.
4. Access asks how many columns your combo box should have. Choose two and fill in the columns as shown on the figure. Then click *Next*.
5. Access asks which column holds the value to store in the table. In our case it is column 1.
6. Finally, Access asks for the column name that the user will see. In our case, *paymethod* is okay. Click *Finish*.

Fill in some stay records

You are now going to create some stay records and connect them to a guest.

7. Close the table design window and open it in user mode.
8. Also open tblGuest in user mode. Keep the two tables side by side so you can see both. Make sure you have created some guests. Otherwise do it now.
9. Fill in a stay record using the combo box for paymethod. Notice that what you see as a designer, is the number stored in the database. The user should not see the number, but the text. We can arrange for this when the field becomes a text box in the user window (see section 3.2.2).
10. Also fill in the foreign key *guestID* so that it refers to one of the guests. Since there is referential integrity, Access won't let you store the stay record without a proper guestID. If you get into real trouble, use Esc twice (see the *Panic* box for the explanation).
11. Fill in a few more stay records in the same way.

How the look-up field works

Open tblStay in design mode and study the Lookup tab for *paymethod* (bottom of Figure 2.4). The display control property is Combo Box. It means that when the user is to fill in the paymethod, he sees a combo box.

- For ordinary fields *Display Control* is *Text Box*. A text box shows texts, numbers, etc. as a string of characters. If you want to change the field back to an ordinary field, just set *Display Control* to Text Box.

The values the user can choose between are listed in *Row Source*. You may edit the values here. *Column Count* shows that these values are to be displayed as two columns. Notice that *Limit to List* is *No*. It means that the user can enter other values than those in the list. In our case, it is not desirable, so set the property to *Yes*. Sections 3.1.6 and 3.1.7 explain more about combo boxes.

Undo the Lookup Wizard?

How do you make the field an ordinary field rather than a lookup field? It doesn't help to make it an integer or a text. Choose the Lookup tab at the bottom of the table design window. Change Display Control to *Text Box*. (See bottom of Figure 2.4.)

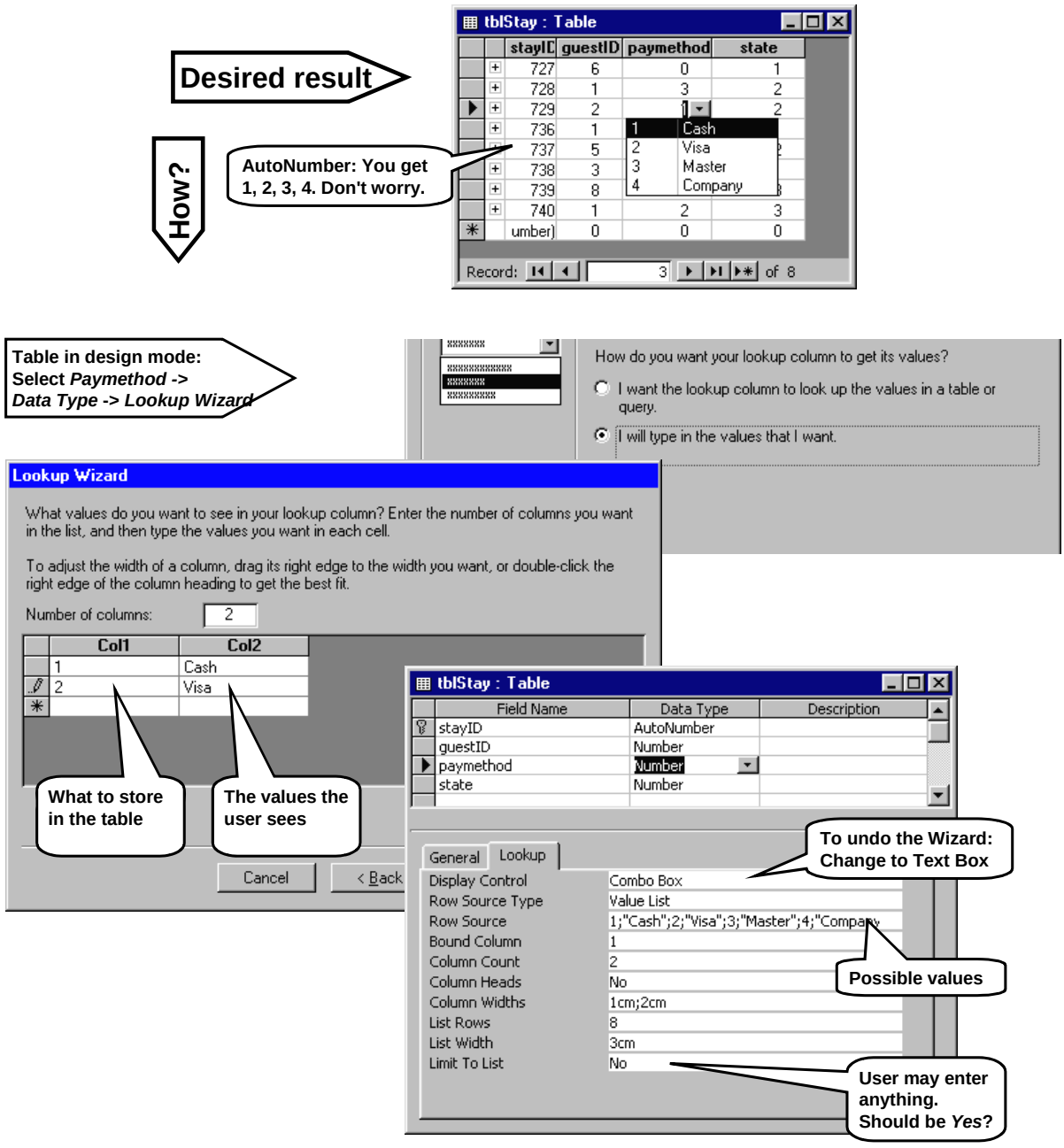
Panic? Undo data entry

When you enter data into the tables, Access checks against the rules you have defined for the tables and the relationships. For instance, when you enter the guestID in tblStay, this ID must correspond to a guest in the guest table. Access doesn't allow you to leave the record before this is fixed. The reason is that Access stores the record in the database as soon as you move the cursor away from the record. And the database must meet all the rules you have stated.

Sometimes you may not know what to type to satisfy Access, and on the other hand you cannot leave the record to look at what to type. Many users panic here and even switch off the power to close down the system. The solution is to use **Esc twice**:

- First Esc: Undoes the correction you made in the field where the cursor is.
- Second Esc: Undoes all the changes you made to the record where the cursor is. This means that the database returns to a consistent state where all the rules are met.

Fig 2.4 Look-up fields, enumeration type



Populate the database

12. Define the other enumeration fields as lookup fields in the same way (the state fields in tblStay and tblRoomState).
13. Fill in some realistic data in all the tables. You may for instance use data corresponding to the situation in Figure 1A. Now you have test data for the rest of the booklet.

Important: Compact the database

Access is very liberate with disk space and when you change things, it consumes new blocks on the disk.

You may soon find that a simple little database uses several megabytes. Fortunately, Access can compact the database. Do that every now and then in this way:

14. Select Tools->Database Utilities->Compact and Repair Database. That is all. You may check that the file length actually became much smaller. (In Access 97, the Compact and the Repair utilities are separate.)

2.5 Dealing with trees and networks

E/R models can neatly describe complex relationships, for instance as we saw it for the flight routes in *User Interface Design*. Figure 2.5 shows the E/R model, but Access cannot show such a model directly.

The problem is that Access identifies a relationship by means of the two tables it connects. This means that Access cannot have two connectors between the same two tables. Also you cannot have a self-referential connector. In the flight route model we need both of these.

As a compensation, Access offers *shadow copies* of a table. The table and its shadow copies are the same table, but they have different names. You can now create connectors to the shadow copies and thus indirectly create multiple connectors between the same two tables.

Figure 2.5 shows how to handle the flight routes in Access by means of shadow copies.

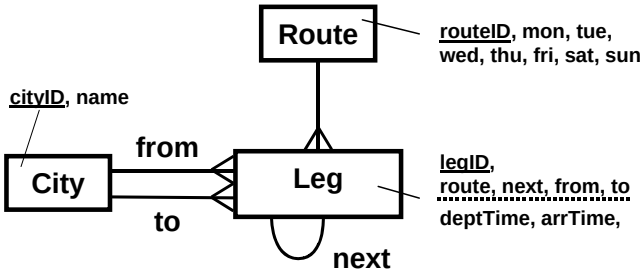
1. Create a new database, **FlightRoutes**. Create the tables *City*, *Leg* and *Route* in the usual way.

2. Open the relationship window and add all three tables to the relationship window. Then add *City* and *Leg* once more. The relationship window should now contain also a *City_1* and a *Leg_1* as shown on the figure.

3. Drag the connectors as shown. You now have two connectors between City and Leg. One is determined by *City* and the foreign key *from*. The other is determined by *City_1* and the foreign key *to*. You also have a self-referential connector from Leg to itself. It is determined by *Leg_1* and the foreign key *next*.

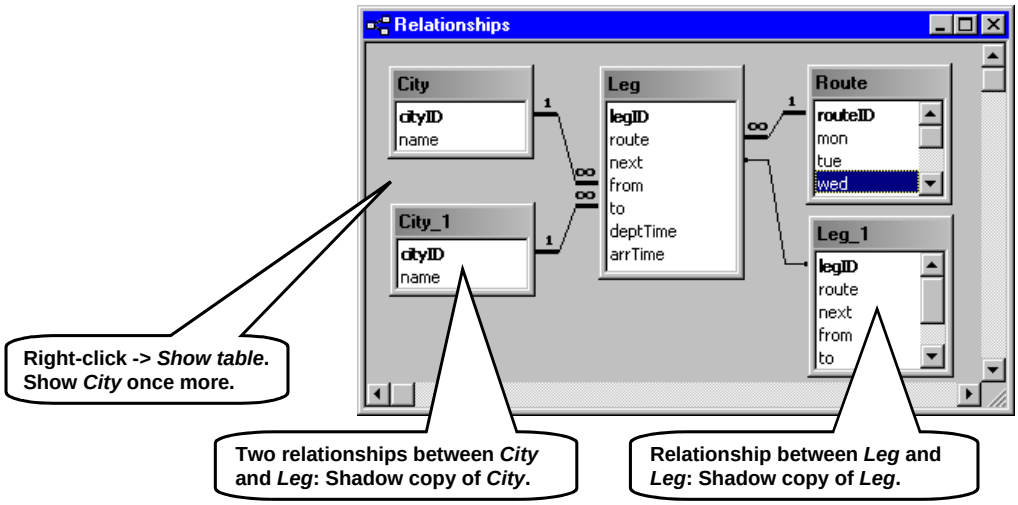
4. Try to fill in data for AA331 according to the figure. Note that there are only one City table and one Leg table to fill in. The shadow tables are not real tables.

Fig 2.5 Flight routes - shadow tables



Route: AA331. Mon, Wed		
	Arr	Dep
Chicago		10:45
Columbus	11:40	12:20
Washington	13:30	14:15
New York	15:10	

A leg



Right-click -> *Show table*. Show *City* once more.

Two relationships between *City* and *Leg*: Shadow copy of *City*.

Relationship between *Leg* and *Leg*: Shadow copy of *Leg*.

3. Access-based user interfaces

Highlights

- Construct user windows (*Forms*).
- Add fields, sub-windows, etc. (*Controls*).
- Construct menus and other details.

An Access-based user interface consists of user windows (called *Forms* in Access) , menus, and all the little things such as error messages (*message boxes*) and pop up help when the cursor rests on a field (*control tips*). These are the things the user sees on the screen. Access provides a lot of built-in functionality that makes the user interface respond to user actions. However, for a real system the built-in functionality is rarely sufficient, and you will have to add your own program pieces written in Visual Basic.

In this chapter we look only at what the user sees on the screen. We hardly put real data into the fields. What we are after is a tool-based mockup. Later we will add real data and functionality to the screens.

3.1 Forms and simple controls

In this section we will gradually implement the mockup window you see in Figure 3.1A. This window helps the receptionist find a guest or a booking in the database. Large hotels may have more than 100,000 guests in the database.

In the Access world, the window consists of a *Form* with various *Controls* on it. A control may be a simple field such as *Last name*, a button such as *Find guest*, an area for a list of records such as the list of stays, and several other things.

Let us get started:

1. Start in the database window, click the *Forms* tab, and select *Create form in Design view*.

You now get an empty form in design mode, looking somewhat like the one at the bottom of the figure. We will put some of the controls on this form in a moment.

In order to align the controls nicely, it is best to show a visible grid of dots on the form:

2. The property box for the form should be open. If not, double click on the anonymous little square on the form - where the rulers meet.
3. **Set the grid dots.** The form has a grid of lines. They may be spaced as on the figure - a one-cm grid. Or with larger cells - a one-inch grid. We also want to see the grid dots. Look at the Format tab and find the Grid X and Grid Y properties. (They are far down the list - you may have to scroll down to them). For a one-cm grid, set Grid X=5 and Grid Y=5. For a one-inch grid set Grid X=12 and Grid Y=12. Move the cursor to another property to make the changes take effect. You should now see a grid of dots as on the figure.

3.1.1 Text box, label and command button

4. The screen should show a toolbox window where you can choose between various controls (bottom left on the figure). If it doesn't, use View -> Toolbox to see it.
5. Click the Text Box tool that looks like ab| and then draw the white part of the field Last name. Draw it so that it is two grid units high and about ten units wide.

At this stage, don't worry if your controls are not properly aligned and sized. We explain about these details below. To delete a control, select it and click Del. Or use Esc to undo the last thing you made.

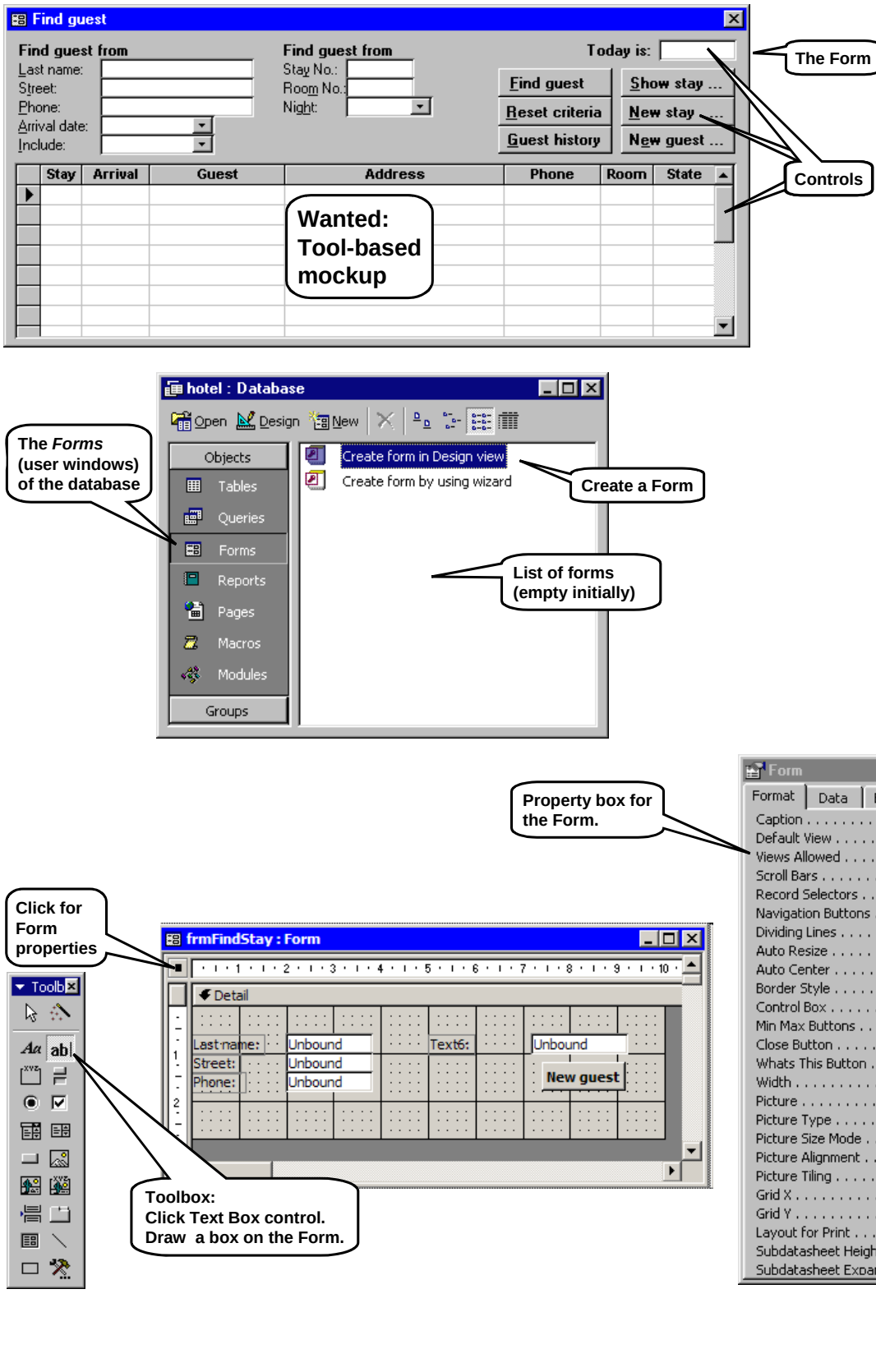
6. Access has automatically added a *label* part to the left of the field. Click it and change the label for the control to *Last name*:
7. Draw the two next text boxes as shown on the figure (Street and Phone). If you double click the text box icon in the toolbox, the draw tool remains selected so that you can draw many text boxes. (Click Esc to get rid of it.)
8. Also draw a spare text box for later experiments (Text 6 on the figure).

9. Now make sure the Wizard button is *off* at the top of the toolbox.
10. Select the *Command Button* tool and draw the *New guest* button. Make it three grid units high to allow space for the text on the button. Click on the text in the box to change it.

If you have the Wizard button *on*, Access will try to make the button do something. (Our mockup has nothing to do, so don't use the Wizard.)

In general, two grid units are a good height for a text box and three units are suited for a command button.

Fig 3.1A Create a Form - a user window



Using the fields
You may wonder why Access writes *unbound* inside all the boxes. It means that the box is not bound to any record in the database. The user may enter something but it is not automatically stored in the database. Try this:

11. Close the form. Access asks for its name. Call it **frmFindStay**. (We use the prefix *frm* for forms.)
12. Open it again in user mode. It should now look like the bottom of Figure 3.1B. This is how the user would see the form.
13. Try to enter something in the fields. It stays on the screen, but is it persistent data?
14. Close the form and open it again (in user mode). All the fields are blank - no data was saved. It was just dialog data - not persistent data. Click the command button - nothing happens. It is just a mockup we have made. (In the next chapters we will add real data and functionality.)

3.1.2 Adjusting the controls

1. Close the form and open it in design mode. Select one of the text boxes. Notice the two black handles, one on the label part and one on the text box.
2. **Moving and sizing.** Point the mouse at the label handle. The cursor changes to a finger. Try to drag the label part around. The text box itself doesn't move. Point at the text box handle and use the finger to drag it around.
3. Point at the border of the text box. The cursor changes to a hand. Drag it - both label and text box should move.
4. Point at one of the sizing handles in the corners or on the middle of a side. Drag here and the box changes size.
5. **Deleting a control.** Click on the text box and click Delete. Oops - both box and label disappeared! Undo it using the Undo button or Ctrl+Z.
6. Click on the *label part*. Notice that now the sizing handles are on the label part. Click Delete. The label part disappears.

If you want a label without the text box, select the label tool from the toolbox and draw a label control.

If you want to add a label to a label-less text box, select some label, copy it (Ctrl+C), select the text box and paste the label (Ctrl+V).

7. **Moving and sizing with the keyboard.** Select a control, then try moving it around with Ctrl+up, Ctrl+down, etc. Try moving it with Shift+up, etc. Now it changes size. This is one way to fine-tune the positions and sizes. There is no way to enlarge the picture as you can do in Word and many other programs.
8. **Select several controls** at the same time. Either hold Shift down while clicking on the controls one by one, or drag a rectangle around them. (All controls touching the rectangle will be selected.) Now try to move and resize the controls with the keyboard, or drag them with the mouse.
9. **Undo.** You can undo your last operation with the Undo button or Ctrl+Z. But only the last! You can undo all changes since you last opened the form by closing the form and saying *No* to saving the changes. Try it now - you don't want to save the last adjustments.

Make sure you see all menu items
Access 2000 and 2003 have an annoying feature where it shows only the last menu items you have used. It makes it difficult to follow the procedures below. Get rid of this feature:

10. Right-click anywhere in the menus. Select Customize->Options. **Access 2000:** Look at the checkbox "Menus show recently used commands first". Make sure that there is no check mark here. **Access 2003:** Look at the checkbox "Always show full menus". Make sure there is check mark here.

The Format menu and the grid
Open the form in design mode and select a control. Now look at the Format menu at the top of the Access window. There are several things here that can help you design the form:

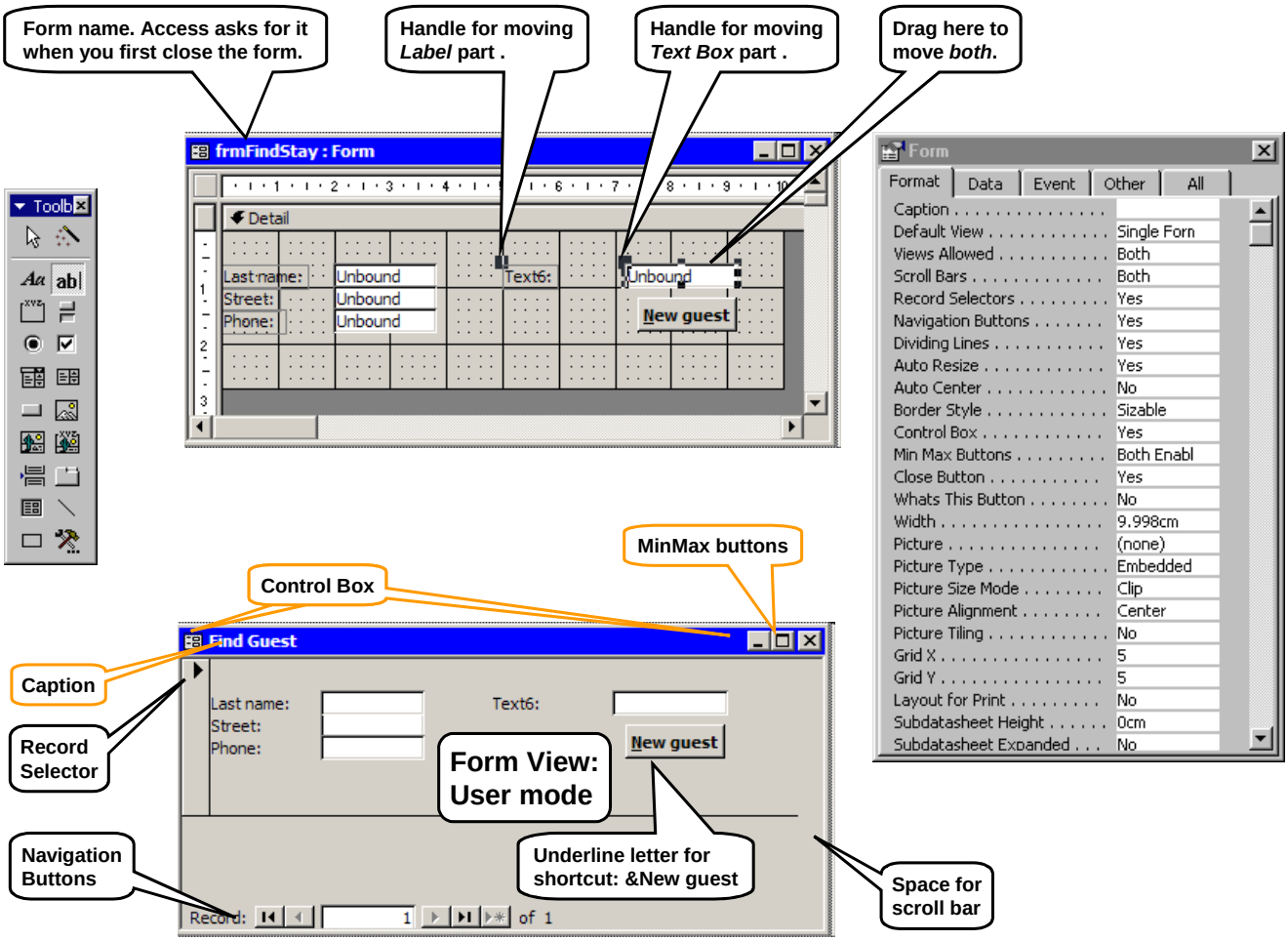
- **Snap to Grid.** If you check this box, all controls you draw or move with the mouse will snap to the grid in all four corners. If the grid points are closely spaced - more than 9 per cm - Access doesn't show the grid, but snaps to it anyway.
- **Align.** You can align the selected controls to the left, right, etc. or you can align them to the grid. Only their top-left point is aligned. They don't change size.
- **Size.** You can change the size of the selected controls so that they just fit the data in the control (matching the chosen font size). Or you can make them fit the grid in all four corners. Finally, you can give all the selected controls the same width or height.
- **Change to ...** You can transform the selected control to another one - with reasonable limitations. For instance you can turn a text box into a combo box or vice versa.

3.1.3 Cleaning up the form

You may notice that the form has things in the corners that we don't want in the final user window (bottom of Figure 3.1B). The title bar, for instance, holds our programmer-oriented form name, frmFindStay. It should be *Find Guest*. There are also record selectors, navigation buttons, and space for a scroll bar that we don't need in the final window. We can correct all of this by setting properties of the form:

11. Make sure the form is open in design mode. Show the property box for the form (double-click the little square where the rulers meet).

Fig 3.1B Adjusting controls and the Form



12. Set these properties on the Format tab: **Caption** (the form name the user sees), **Scroll bars** (not needed), **Record selector** (not needed), **Navigation buttons** (not needed).

There are other interesting properties on the Format tab that you may need for other windows:

- **Border Style** specifies whether the form looks like a resizable window, a dialog box or a message box.
- **Control Box** is the buttons on the left and right of the title bar. You may hide them.
- **MinMax** buttons and **Close** button are shown when the control box is shown, but you may disable them.
- **Picture** is *none* in our case, but you may specify a picture file to be used as background.

You can get a good explanation of most of the properties by selecting the property and clicking F1.

Look and feel - Autoformat

You can give the form another look by means of Autoformat. This changes the *style*, that is the background of the form and the appearance of all fields and buttons. You may try it if you like:

- Open the form in design mode. From the Format menu at the top of the Access window, select AutoFormat.
- You can choose various auto-formats. Through the Options button you can determine whether you want to change also field colors, fonts and borders. When you close the AutoFormat box, the form has changed its look.
- You may also create a new auto-format style based on one of your forms. Open the form in design view. In the AutoFormat box, select Customize -> Create a new, and give the new AutoFormat a name. You can then use this auto-format for other forms.

3.1.4 Shortcut keys for the user

In the final system, the user should be able to work without a mouse. An easy way to do this is to assign a shortcut key for each button and each label. As an example, we might want the user to activate *New Guest* with Alt+N. Why not do it now?

1. Change the name on the button into *&New guest*. Change to user mode with the little icon in the top left corner (Figure 3.1C). You should see that the N is underlined.
2. Try Alt+N to move the cursor to NewGuest. Try Tab and Shift+Tab to move between fields.

3. Add shortcut keys for the other labels too, for instance as *&Last name*. Try it out in user mode.

What if you want a label with an &, such as *Bed&Breakfast?* Access will treat & as a shortcut mark. Remedy: Write && instead of &.

Tip: Changing mode/view. During design, you frequently change between design mode and user mode. The little icon at the top left (Figure 3.1C) allows you to toggle between the modes. Click it - you change to user mode (called *Form View* in Access). Click it again - you change to design mode (Design View).

The icon has also a menu of views that you can roll down as shown on the figure. There is one more view, Datasheet View, which we will use later. It shows all fields of the form as a table.

Saving. The form is not saved when you change mode. You can thus experiment easily with the design. Saving is not done until you close the form or explicitly save it with Ctrl+S.

Shortcuts. You can change to design mode with Alt+V+Enter, and to user mode with F5.

3.1.5 Lines, checkbox, calendar

Above we have tried some of the controls: text box, label, and command button. Figure 3.1C shows some other controls you may try now:

1. **Line**. Select the *Line* tool from the toolbox and draw a line somewhere on the form. The line is just a visual effect. It has no functionality.
2. **Rectangle**. Select *Rectangle* from the toolbox and draw a rectangle around some of the existing controls, for instance the left fields. The rectangle is just a visual effect without functionality.
3. **Colors.** Double click on the rectangle to open its property box. On the format tab, give the rectangle a back color and set back style to *normal*. Now it hides the controls it surrounds. Use the main Access menu, *Format -> SendToBack*, to move it behind the other controls. Experiment with different back colors, border styles, and border colors. It may in some cases be a good way of grouping controls visually.
4. **Checkbox**. Select the checkbox tool and put a checkbox on the form. The checkbox has functionality and shows a yes/no variable (a Boolean variable). You can of course change the value in user mode. If the variable is undefined - as it is initially - the checkbox is gray in user mode.

Calendar control. The middle part of Figure 3.1C also shows a *calendar control*.

5. To make room for the calendar control, extend the grid area of the form by dragging its borders in design mode.
6. Select the lower right icon of the toolbox (the hammer). It gives access to more controls, most of them rather complex. Depending on the way Access was installed, you can see more or less of the many controls. One of them is a Calendar control.

If you cannot see any Calendar control, you may have to tell Access to look for it. Use *Tools -> References*. You now see a list of the software packages Access may look at. Find a *Calendar Control* and make sure it has a check mark. Then close the reference list and select once more the hammer from the toolbox.

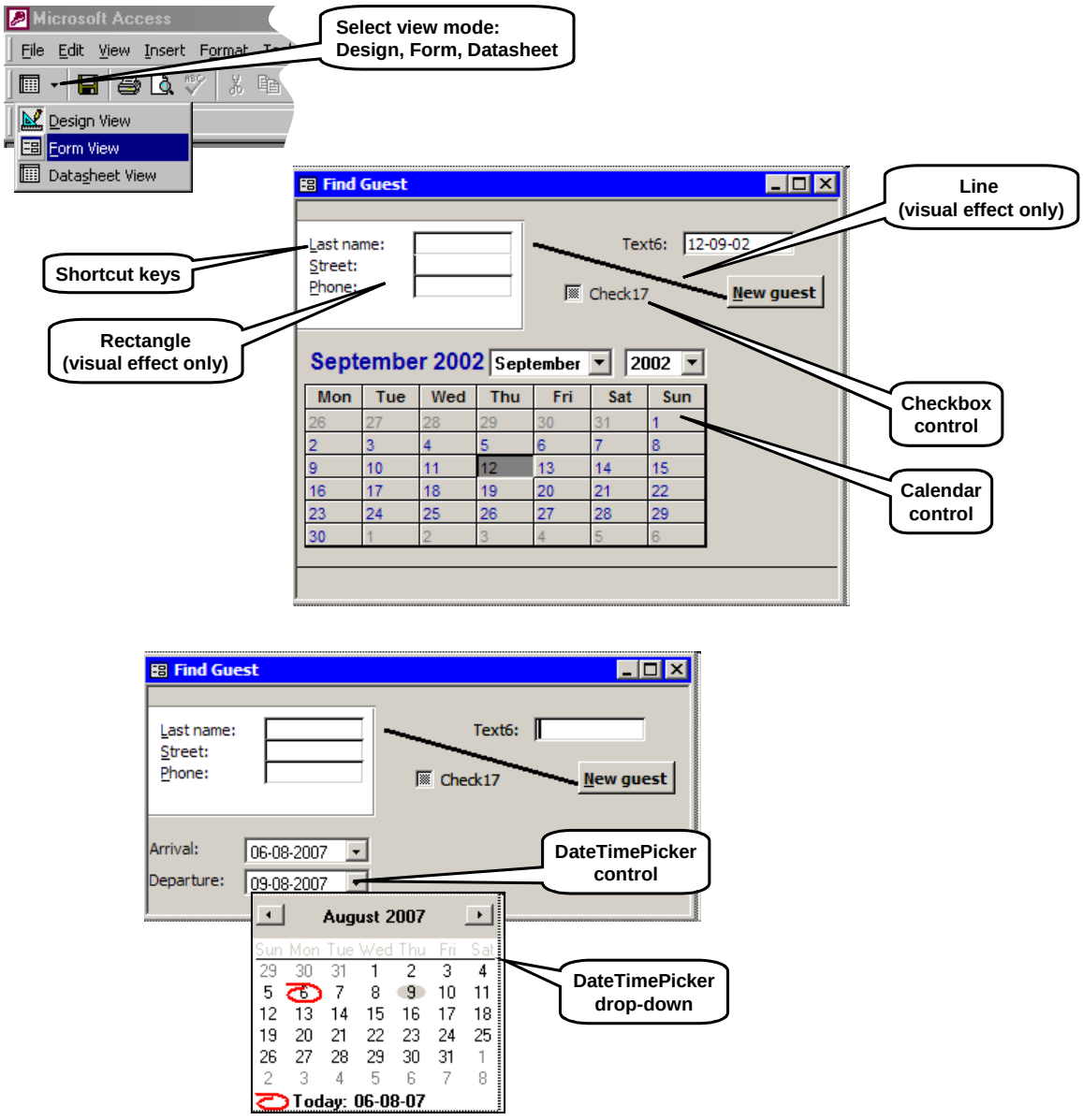
7. Select the Calendar from the tool list and draw a large rectangle with the tool.

The Calendar control shows a single date variable. In user mode you can click on a date in the calendar and in that way store a date in the variable. In principle, the Calendar control is just a kind of text box with a different way of presenting and editing the text value.

DateTime picker. The bottom part of Figure 3.1C shows two *DateTime pickers*. They look like combo boxes, but when the user clicks the down-arrow, a calendar appears.

8. Extend the grid area further, or remove the calendar control.
9. Select the hammer tool again and look for *Microsoft Date and Time Picker Control.* Select it. (You may have to include it from *Tools -> Reference*, as above.)
10. Draw the combo-box part of the control. You now have a control that holds a date-time variable. Try it out in user mode. Notice how the user can increase or decrease dates and months with arrow up and down.
11. The control doesn't have label. Give it one: Select one of the other labels. Copy it (Ctrl+C). Select the date-time control and paste it (Ctrl-V).
12. Create the other date-time control in the same way - or copy and paste the first one.
13. Experiment with the properties of the DateTime picker: In design mode, double-click the control. A

Fig 3.1C Form after changes



special DTPicker properties window should appear.

(You can also get to this special window from Access's standard property window: Select the Other-tab and then *Custom*.)

14. Try changing the date format: Select format 3, *dtpCustom*. In the *CustomFormat* box, define the format as *dd-MM-yyyy*. Note that MM means month, while mm means minute. (See also Figure 6.4B, Format function).

Try other changes too, for instance the UpDown checkbox and the colors.

15. We are not going to use this fancy version of the form in the following. If you want to keep it, save a copy of it: Select the form in the database window and use copy and paste.
16. Delete the line, rectangle, checkbox and calendar controls. We don't need them in the following.

3.1.6 Combo box - enumeration type

Combo boxes are a bit more complex. We will first make the *Include* combo box shown on Figure 3.1D. It is one of the search criteria for guests, and with some programming it will allow the user to display only booked stays, only canceled stays, etc.

Technically speaking, this combo box holds a value of enumeration type:

include(booked | canceled | . . .)

In the same way as in the database, the user should see the values *booked*, *canceled*, etc., but they should be stored as the values 1, 2, etc.

1. Switch to design mode.
2. Set the Wizard button *on* at the top of the toolbox.
3. Select the Combo box tool and draw the Include box as shown at the top right of the figure.

The Wizard appears. It works much the same way as when you defined an enumeration-type field in the database (section 2.4):

4. The Wizard asks you whether you want to look up the values from a table or type them in yourself. Select the latter and click *Next*.
5. The Wizard asks you how many columns you want. Choose two: one for the stored value and one for the value the user should see.
6. Fill in the columns as shown, and click *Next*.
7. The Wizard asks you to specify whether the stored value is column one or two. Select column 1 and click *Next*.
8. Finally, you may specify the label text in front of the combo box. Use the text *Include:* (Or modify the text directly on the Form). Finish the Wizard.

Look a the result in user mode. It doesn't look quite right. The drop-down list has two columns and the box itself shows the number - not the user text. We have to repair this:

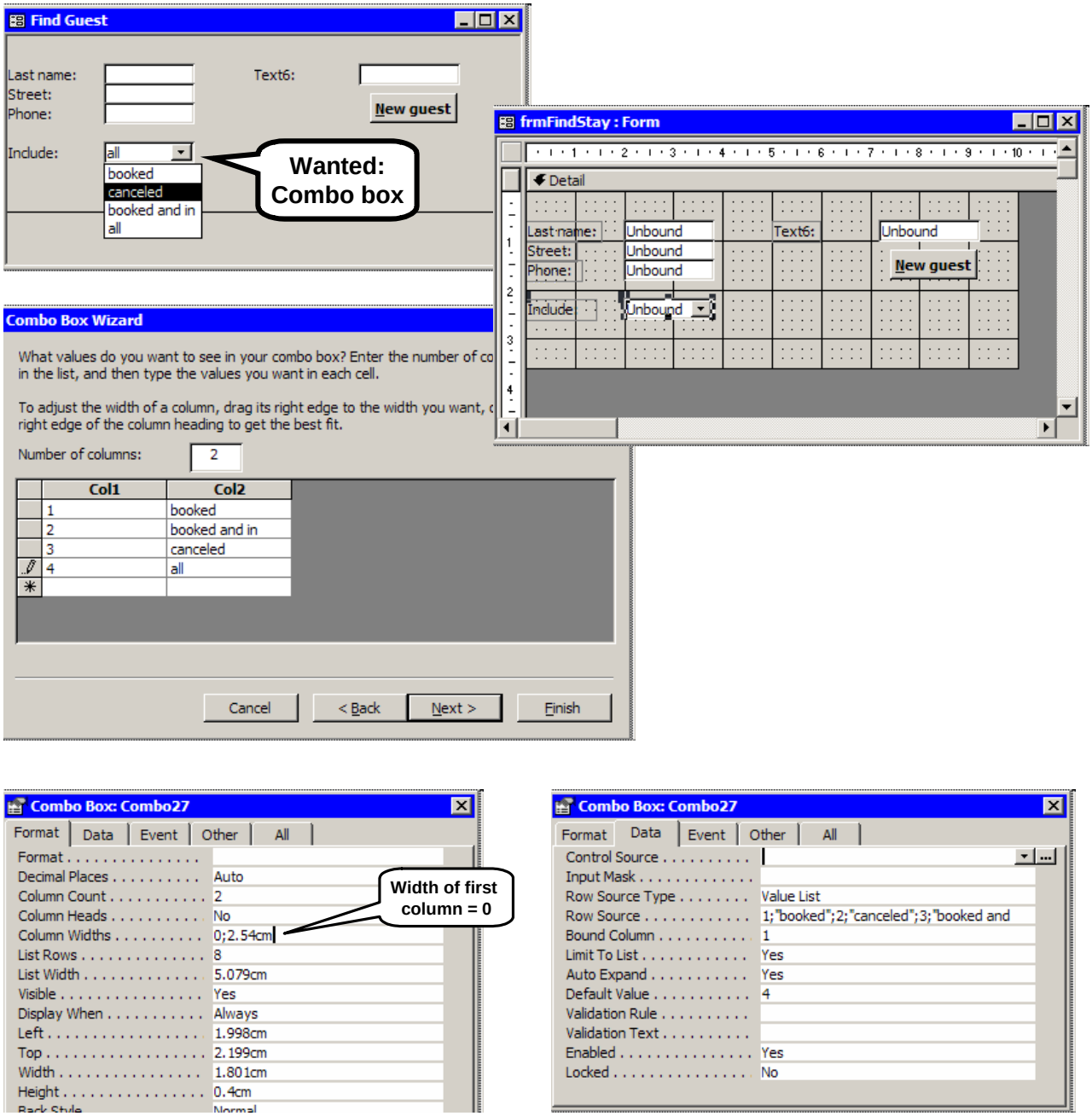
9. Look at the property box of the combo box. The Format tab has a field called *Column Widths*. It shows the widths of the two columns. Set the width of the first column to 0 (see the bottom of Figure 3.1D).
10. Try it out in user mode. Everything should look right by now.

- The Format tab has other interesting fields. You may for instance adjust the List Width for the drop down list.

11. Select the Data tab. *Row Source* holds the values in the list. You may edit them here.
12. *Limit to List* defines whether the user is allowed to enter other values than those in the list. In this case, it should be set to *Yes*.

- *Bound Column* defines which column to use for the stored value.

Fig 3.1D Combo Box - enumeration type



3.1.7 Combo box - table look up

We will now make the *Room type* combo box shown on Figure 3.1E. This combo box might be another sorting criteria. The combo box is not an enumeration type where the designer has typed in the values, but a table look up. In the example it stores a roomType ID, but the user will see the name of the room type.

1. Switch to design mode and make sure the Wizard button is on.
2. Select the Combo box tool and draw the Room Type box.
3. Tell the Wizard that you want the combo box to look up the values in a table. Click *Next*.
4. Select tblRoomType as the source. Click *Next*.
5. Tell the Wizard that you want to show these fields as columns: *roomType*, *description*. Click *Next*.
6. Ask the Wizard to hide the key column. This means that the key will be stored, but the description shown to the user. Finish the Wizard.

Try out the combo box in user mode. It should look all right.

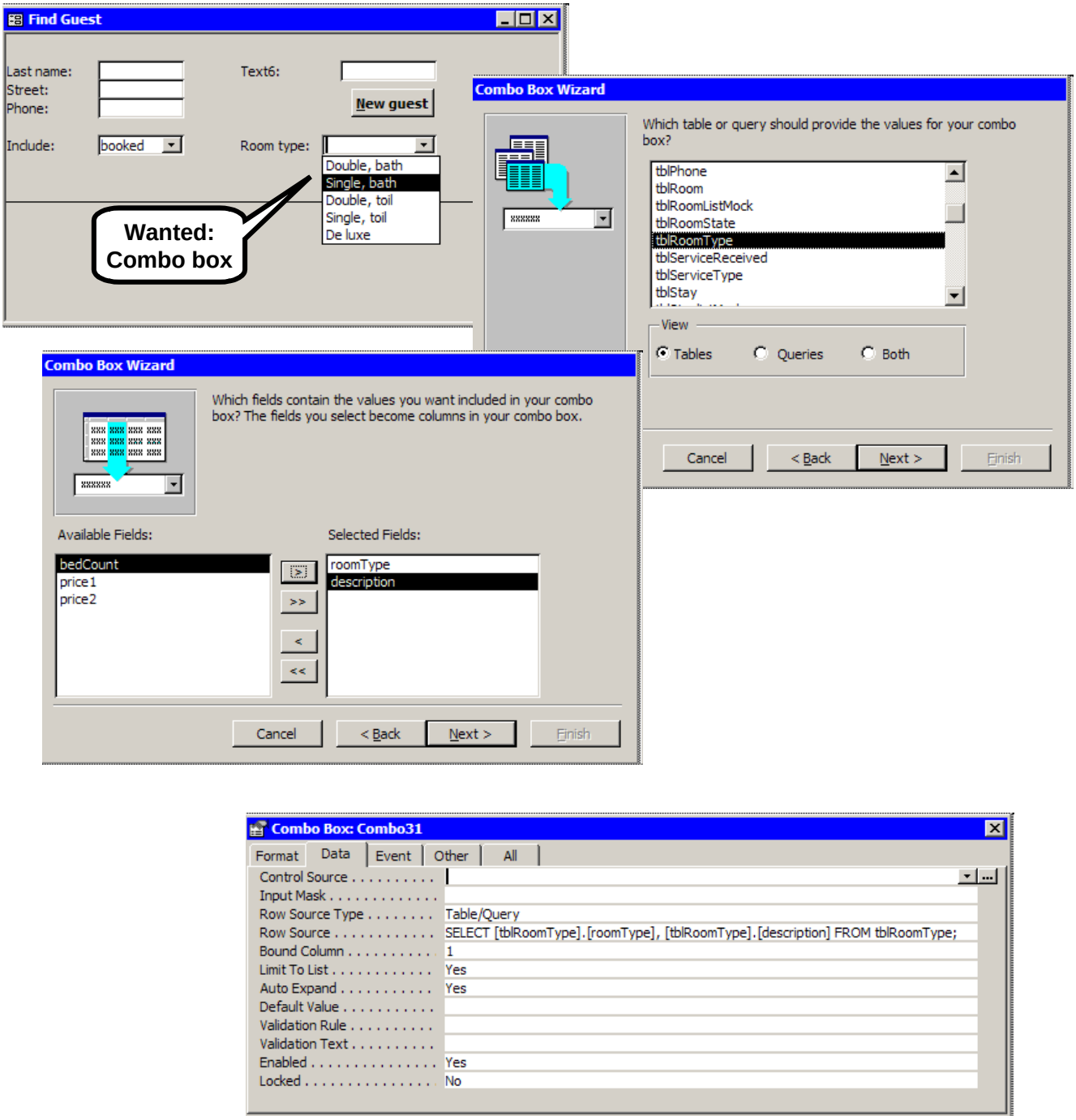
The bottom of Figure 3.1E shows the Data tab for the combo box:

- *Row Source* now contains a formula called an SQL-expression. It specifies how to compute the list that the user will see. We will look at SQL-expressions in section 4.2.

- *Bound Column* indicates the column that holds the value to be stored in the table. In our case it is column one, which holds roomType.

We won't use these combo boxes later in the booklet. You may leave them on the form or delete them.

Fig 3.1E Combo Box - table lookup



3.1.8 Control properties - text box

Each control has a lot of properties that define its color, font, and many other things. We will explain some of these properties below, using the text box as an example.

Select a text box and look at its property box (Figure 3.1F). If the property box isn't open, bring it up by double clicking on some control. There are more than 60 properties for a text box. Here we will look at some of them.

Text box - properties on the Format tab

- **Format** and decimal places specify the data type of the control, much the same way as you can specify the data type for a database field. You can select among a number of predefined formats, or type your own format into the format field. The formats are similar to the Visual Basic formats (see section 6.4 and the reference card).

- **Scroll Bars**. A text box may be large and show a text consisting of many lines. This property specifies whether it should have scrollbars.

- **Left**, **Top**, **Width**, **Height** specify the position and size of the control. You may set these properties instead of dragging with the mouse.

- **Back color**, fore color, font name, etc. specify colors, borders and other visible properties of the control.

- **Text Align**. You can align the text, e.g. left for a name, right for a number.

Text box - properties on the Data tab

- **Control Source** specifies how the value is computed and where it is stored. For unbound controls as those on the FindGuest form, control source is blank. We look at the other possibilities in section 3.3.

- **Input Mask** specifies the text box format when the box has the focus and the user types into it. The mask may for instance be used to enter dates with predefined slashes and hyphens. The input mask follows different rules than the format property. (Not described in this booklet.)

- When **Locked** is Yes, the user cannot enter data into the text box.

- When **Enabled** is Yes, the text box can have the focus.

  When Enabled is No, the user cannot enter anything in the box because the cursor doesn't stop there. In this case, Locked has an interesting influence on the box color. If Locked is No, the field is gray. If it is Yes, the color follows the normal pattern determined by Back color and Fore color.

Text box - properties on the Other tab

- **Name** is the programmer's name for the text box. Visual Basic programs refer to the text box with this name. The designer can change the name. Notice that the name is shown in the title bar too.

- **Tab Index** determines how the cursor moves through the controls when the user tabs through the form. Tab indexes run from 0 and up. When the form opens, the cursor is in the control with tab index 0. The tab key moves the cursor though tab index 0, 1, 2, etc.

- **ControlTip** is the pop-up text the user sees when the mouse rests on the text box.

Label control - properties on the Format tab

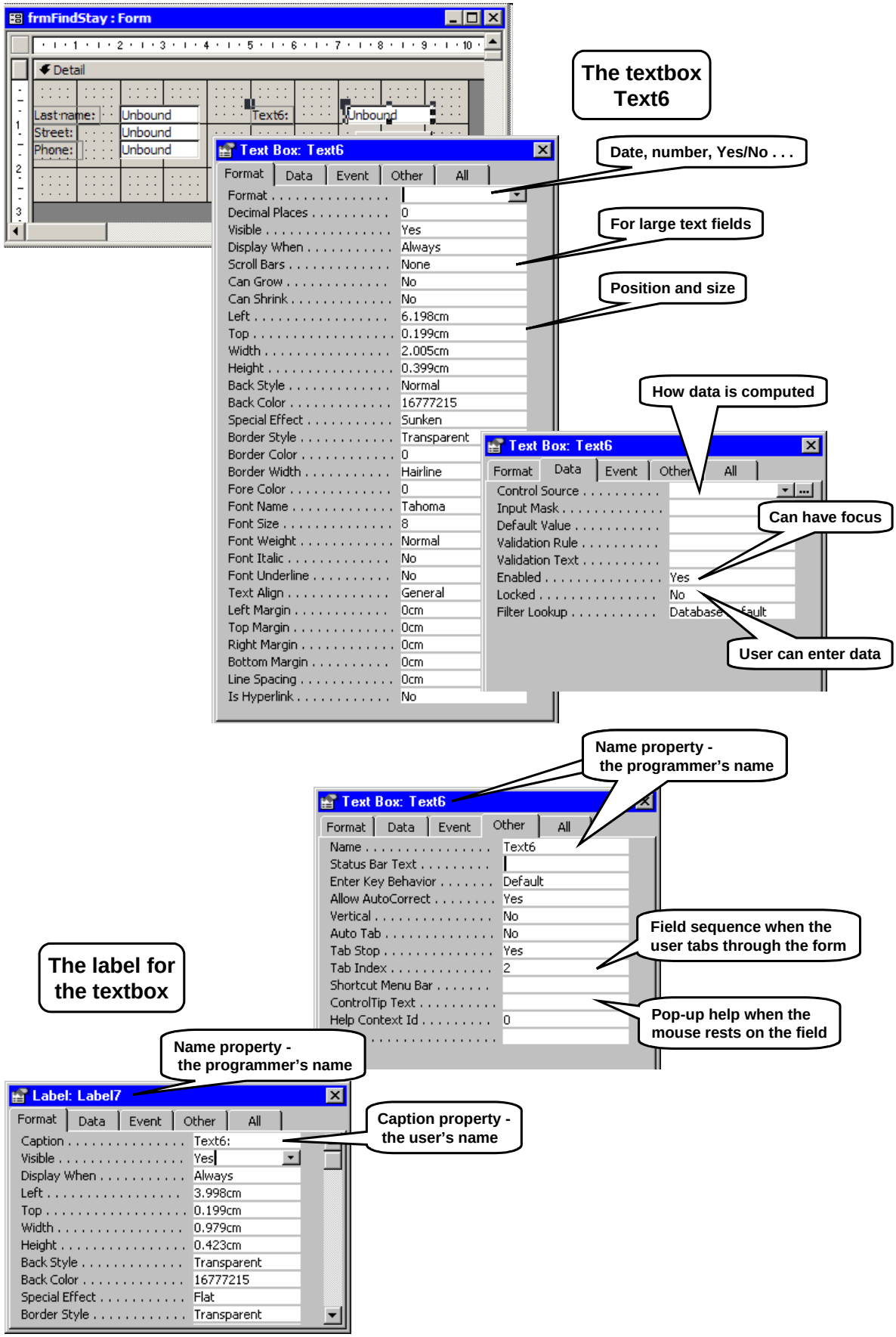
The text box has an associated label. The label has a programmer's name. In the example, Access has given the label the **name** Label7.

- **Caption** is the label text the user sees. In this case Access generated the caption *Text6*: The designer can change it, of course.

Many of these properties exist also for other control types, for instance for the combo box. For the command button, many of them work too. For some strange reason, however, you cannot align the text on a command button. It is always centered. In the hotel system we have made a fake left align by entering spaces after the name. You have to enter the spaces directly on the button - you cannot do it in the Caption property.

To learn more about a property, click the property line and click F1 for help.

Fig 3.1F Control properties - text box



3.2 Subforms

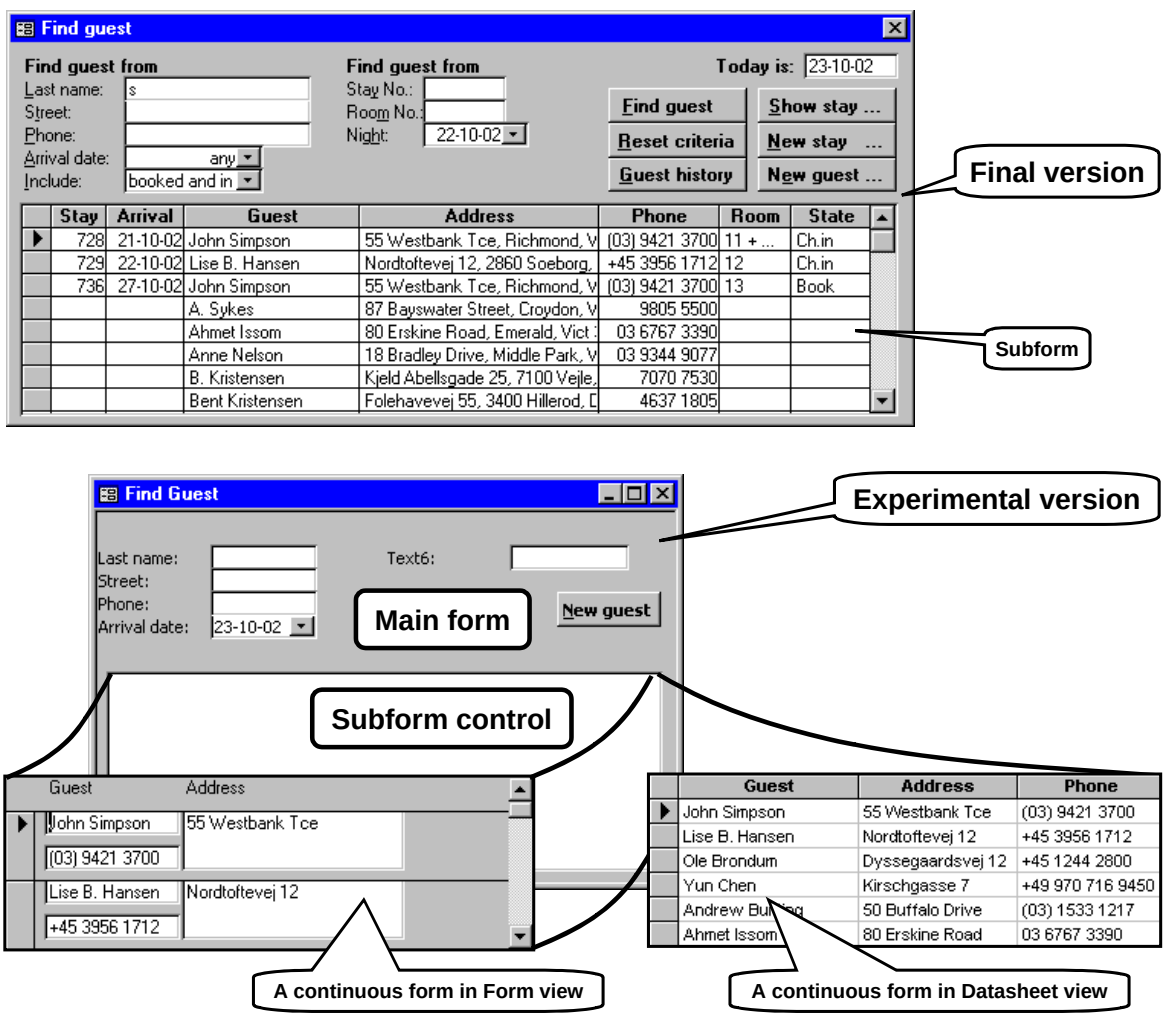
Figure 3.2A shows the FindGuest form we want to construct. The controls at the upper part of the form are now easy, but how do we handle the list of stays at the lower part? Basically we make a field that can show a list of data. This field is called a *Subform control*. The data in the list is a combination of data from tblGuest, tblStay, and other tables. In the next chapter we will show how to combine these data.

To simplify matters right now, we will make an experimental version of frmFindStay, where the subform shows only guest data.

The bottom of Figure 3.2A shows the experimental version. The **main form** is frmFindStay, which we made in section 3.1. On it we have made a **subform** control - the white area. The list of data is another form (a **continuous form**) that shows a list of guest records. On the figure we have shown this continuous form in two versions: as a form and as a datasheet.

When we have made the subform control and the continuous form, it is a simple matter to connect the two. The result is a form looking like the final FindGuest.

Fig 3.2A Main form and Subform



3.2.1 Subform in Datasheet view

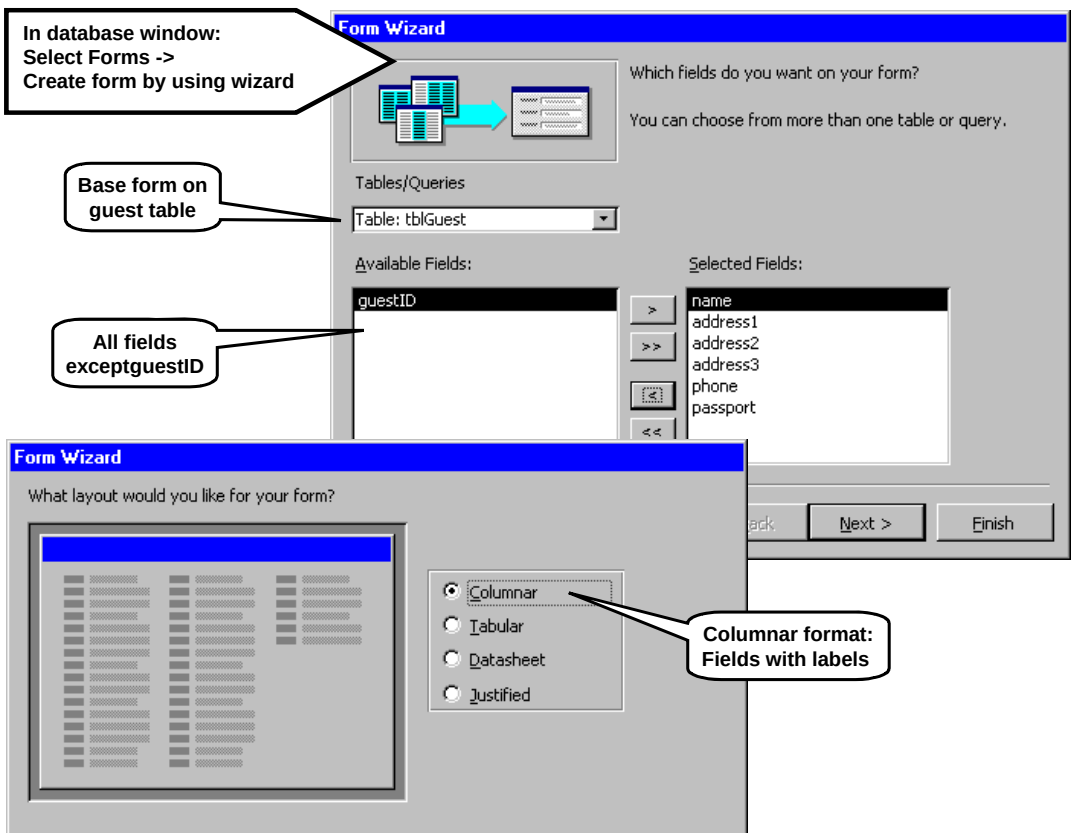
First we will make the continuous form that shows the guest data. We could create it from scratch in the same way as we created frmFindStay, but in this case Access has a Wizard that helps us a lot. Figure 3.2B illustrates how to use it:

1. Start in the database window, click the *Forms* tab, and select *Create form by using Wizard*. (In Access 97 you click *New*, then select *Form Wizard*.)
2. Access asks you which table to base the form on. Choose tblGuest.
3. Select the fields you want to include. In our case, select all fields except the guestID field, which the user should not see. Click *Next*.
4. Access asks you to select a layout for the form. The best layout in our case is *Columnar*. It will show the fields with labels. Click *Next*.

It is tempting to choose the *datasheet* layout rather than *columnar*. Don't do it. It works, but the fields get no labels and this makes it much harder to give the fields user-oriented names later in the process.

5. The next screen asks you to select a *style*, meaning the look of the frames around the fields, the picture behind the fields, etc. Choose *Standard* in this case where we just want a simple look.
6. Finally, give the form a name. Choose **fsubStay-List**. The prefix *fsub* is conventionally used for a form that becomes a subform. Click *Finish*.

Fig 3.2B Create the continuous form



You will now see the form in user mode as on Figure 3.2C. You can use PageDown and PageUp to walk through the guests. Or you can use the navigation buttons at the bottom of the form. You may enter data into the form. The data will end up in the guest table.

Choose datasheet view

7. Select the continuous form and use the view icon to change to Datasheet View. Adjust the width of the form and the widths of the columns. The result should be as on the lower part of Figure 3.2C.

When we insert the continuous form into the Find Guest form, we want it to be shown only as a datasheet. Specify it in this way:

8. Select the continuous form and switch to design view. Open the property box of the form and select the Format tab. Set *Default View* to *Datasheet*. **Access 2000:** Also set *Views Allowed* to *Datasheet*. **Access 2003:** Disallow Form View and allow Datasheet View.

Bound and unbound forms

9. Use the opportunity to look at the Data tab for the form properties. It shows that *Record Source* is tblGuest. This is why the form can show fields from the guest table. We say that fsubStayList is **bound to** tblGuest.
10. Switch to Datasheet view to check that everything is okay. Then close the continuous form.

Create the subform control

Now it is time to use the continuous form as a subform on frmFindStay.

11. Open frmFindStay in design view. Extend the grid area by dragging its borders (see Figure 3.2D).
12. Select the *Subform/Subreport* tool and draw a large box as on the figure. The result is a subform control. Delete the label for the subform control (called *Child7* or the like).
13. Look at the property box for the subform control. Set the name property on the Other tab: Name = **subStayList**. The prefix *sub* is used for subform controls.

Connect the continuous form to the subform control

14. Look at the Data tab for the subform control. Set this property: Source Object = fsubStayList (select it from the drop-down list).
15. Switch to Form View.

The result should be as shown at the bottom of Figure 3.2D. The result is not pretty, but is easy to correct the problems. Notice the names we use:

- Name of the Subform control: subStayList
- Name of the Continuous form: fsubStayList (seen in the Access database window)

Fig 3.2C Continuous form

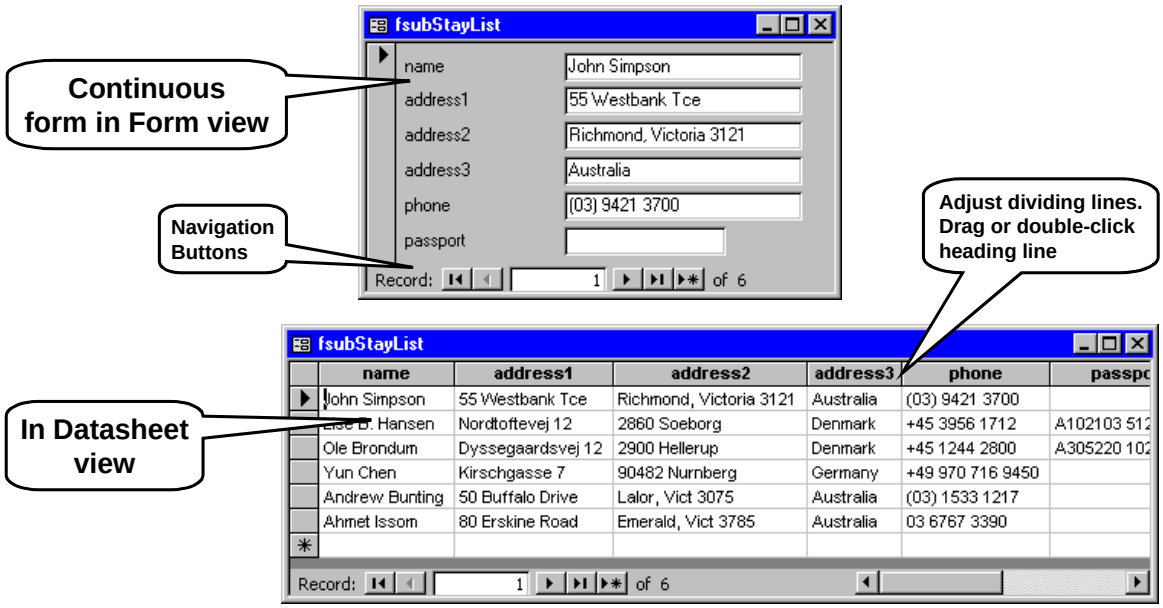
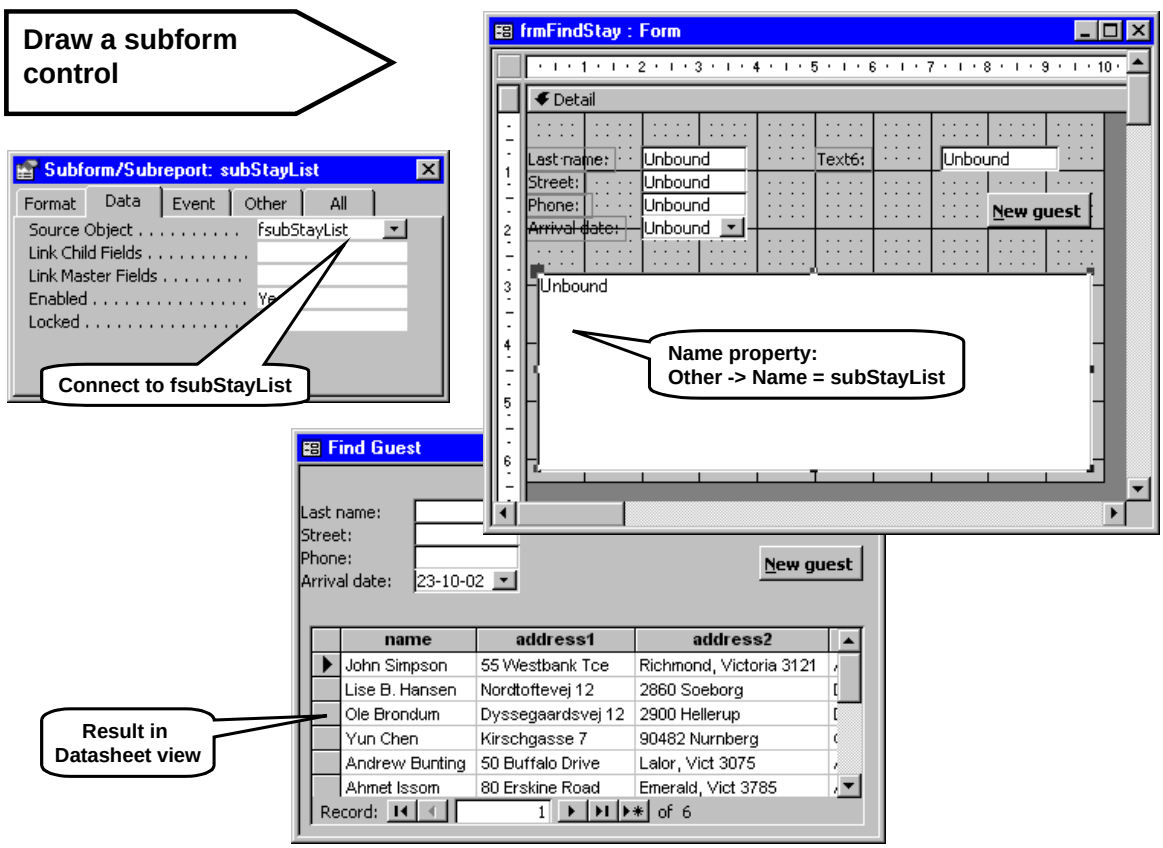


Fig 3.2D Creating and connecting the subform control



3.2.2 Adjust the subform

To make the subform look nicer, we remove the navigation buttons at the bottom, give the columns user-oriented names, etc. (See Figure 3.2E.)

1. Open fsubStayList in design mode. Open the property box for the form and set Navigation Buttons to No.
2. Remove the unnecessary fields *address2*, *address3* and *passport*. (Select them and click Delete.)
3. Change the labels to user-oriented names. (Click the label and change the text on the spot.)
4. Close fsubStayList and open the main form in user mode. It should look like Figure 3.2E.

Notice that the labels are used as column headings. You may wonder what happens if we have a text box without a label. In this case the control name (the Visual Basic name) is used as a heading. You may change the control name to a user-oriented name, but it is harder and would influence a program that addresses the text box.

5. **Adjust column widths**. The user can adjust the column widths. Try it: Point the mouse to a dividing line in the header section (see Figure 3.2E). Drag the dividing line to adjust the column width. Double-click the dividing line to have Access fit the line to the existing data. The column widths are persistent data that are stored as part of fsubStayList.
6. **Column sequence**. The user can also adjust the column sequence: Click the column heading to mark the entire column. Then drag the entire column to another position in the sheet. The column sequence is also persistent data.
7. **Sorting rows**. The user may ask for the data rows to be sorted according to a specific column: Right-click somewhere in the column and choose Sort Ascending or Sort Descending. The sorting sequence is also persistent data. Access records the sorting sequence as the *Order By* property of the form.

Add another field

What to do if you have forgotten one of the fields on the subform? Starting all over with the Wizard is cumbersome. The solution is to use the Field List to add the missing field:

- Switch to design view. Click the Field List icon on the tool bar (Figure 3.2E). All fields from the record source come up as a list.

- Drag the missing field from the list to the form.

Adjust lookup fields

When the field is defined as a lookup field in the database, it becomes a combo box on the form. This is usually fine, except for the fact that in the database we wanted the visible value to be the real thing stored in the database, for instance the code 2 for Visa. This was convenient when we as designers entered test data into the database.

On the user interface we don't want the user to see the codes, but the mnemonic values, for instance *Visa*. Here is how to do it:

- Open the property box for the combo box. On the Format tab, look at Column Widths. Set the first column width to 0. This makes this column invisible to the user.

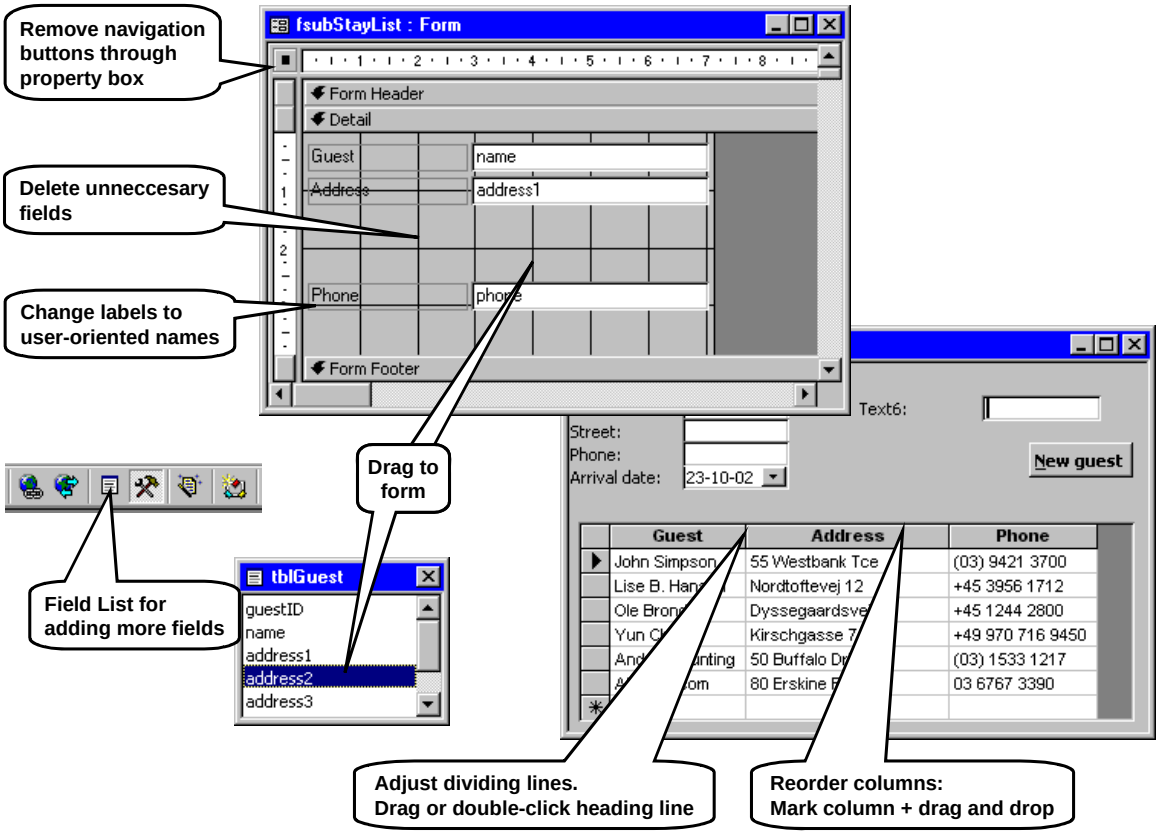
Font size for datasheets. The datasheets in the booklet use font size 8 (default is 10). You can set size 8 as the default font size. Select Tools -> Options -> Datasheet -> Default Font -> Size = 8. This setting will influence all datasheets where you haven't set the font size explicitly.

How to open the continuous form in design mode: You may simply open the continuous form from the database window. However, you may also access it from the main form in design view:

Access 2000 and 2003: When the main form is in design mode, the subform control will usually also show the continuous form. You may click and adjust the controls directly inside the subform control.

Access 97: When the main form is in design mode, the subform field is always blank. Click in the form outside any control. Next double-click the subform control. The continuous form will then open in design mode.

Fig 3.2E Adjusting the subform



3.2.3 Mockup subform

Above we have based our experimental subform on tblGuest rather than a complex computation of the real stay list. This was to try the mechanisms - it looks all wrong to the user. When we make a tool-based mockup for usability testing, we need the planned columns, but the data fields should be empty because we add the data with pencil.

The easiest way to make such a mockup is to start with any continuous form and add dummy text boxes. We will use this technique to create a mockup list of rooms occupied by a guest (Figure 3.2F). (We will later use this form in the *Stay* window.) When filled in with pencil, the list shows the first night, the number of nights, the room number, the number of persons in the room, the price per night, and the total for all the nights.

1. Make a copy of fsubStayList: In the database window, select fsubStayList, copy it with Ctrl+C and paste it with Ctrl+V. Give the copy the name *fsubStayRooms*.

2. Open fsubStayRooms in design mode. Remove all fields relating to the guest and insert text boxes as shown. Give the labels the user-oriented names.

3. Change the caption of the form, for instance to *RoomList*.

4. Switch to Datasheet mode, and the mockup should be ready.

How does this work? The subform is bound to tblGuest and shows a line for each guest. However, the line doesn't show any guest fields, only the text boxes we have added. These text boxes are only dialog data. They don't store anything in the database. If you for instance enter a number in *Nights*, you will see the same number in all the lines. There is only one instance of the dialog data.

If you want more lines in the mockup, add new guests to tblGuest. The mockup shows one line for each guest.

Insert a table directly as a subform?
Access 2000 and 2003 allow you to connect a subform control to a table - without making a continuous form. Don't use this shortcut.

The result looks okay at first sight, but you have no control of the appearance. You cannot remove navigation buttons, use colors in the fields, or address the user's selection from a Visual Basic program.

3.2.4 Subform in Form view

A continuous form can also be shown in Form view. This gives us more freedom to structure the data presentation. As an example, we will create a subform that makes Find Guest look like Figure 3.2G. The subform area now shows a heading area and a list of small forms - the details. Here is how to do it:

1. Make another copy of fsubStayList (Ctrl+C and Ctrl+V). Give the copy the name *fsubStayList2*.

2. Open the copy in Design view and drag the border between header and footer so that you see a header area (Figure 3.2G). If you cannot see the Form Header bar, use the View menu:
   View -> Form Header/Footer.

3. Change the form so it looks like Figure 3.2G:
   - Remove the labels from the detail area and adjust the sizes of the three fields as shown.
   - Adjust the size of the detail area to remove empty space at the bottom.
   - Draw fresh labels in the header area.

At this point, the user can only see the form in Datasheet view. Change this:

4. On the property sheet for the form, set Default Views to *Continuous Forms* and Views Allowed to *Form*.

In user mode you will now see the continuous form as a list of small forms. You may connect it to the main form to get the wanted result: Change the SourceObject property of the subform control.

Form Wizard - tabular layout
It is a bit cumbersome to construct the continuous form in this way. You may let the Form Wizard do some of the job:

5. From the database window, click the Forms tab and select *Create Form by using Wizard*.
6. Select the proper table, next select *Tabular layout*.
7. Finally select a style (e.g. *Standard*) and give the form a name.

You will now have a form that looks very much like the one you constructed above.

Fig 3.2F Mockup subform: Roomlist for a guest

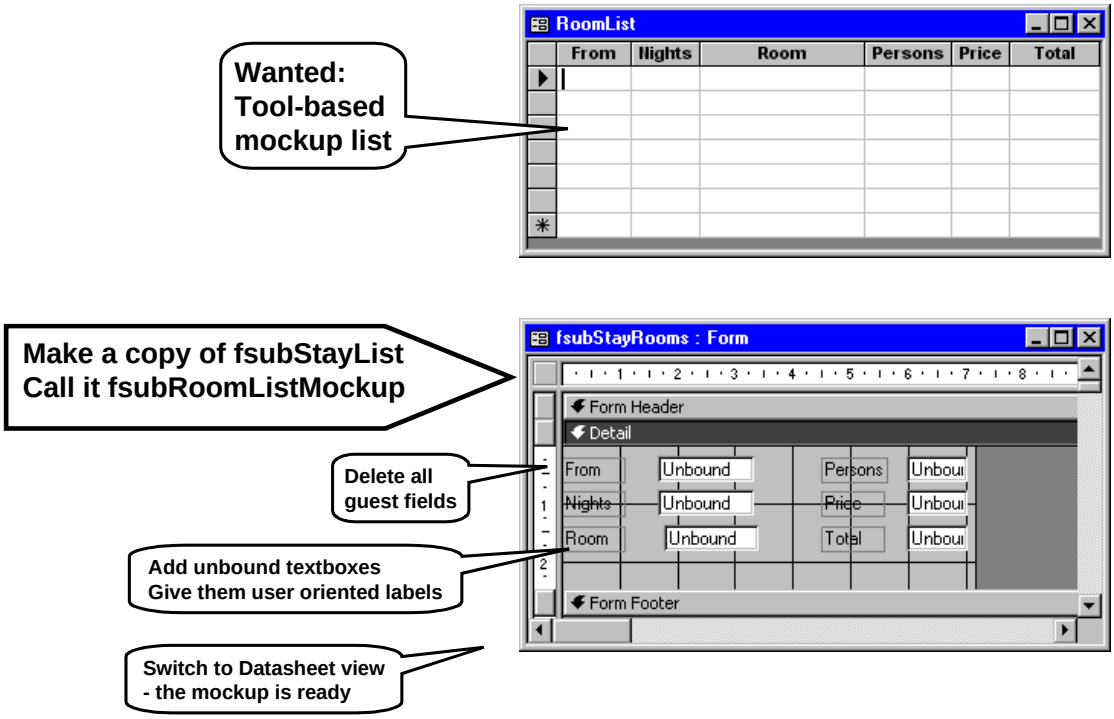
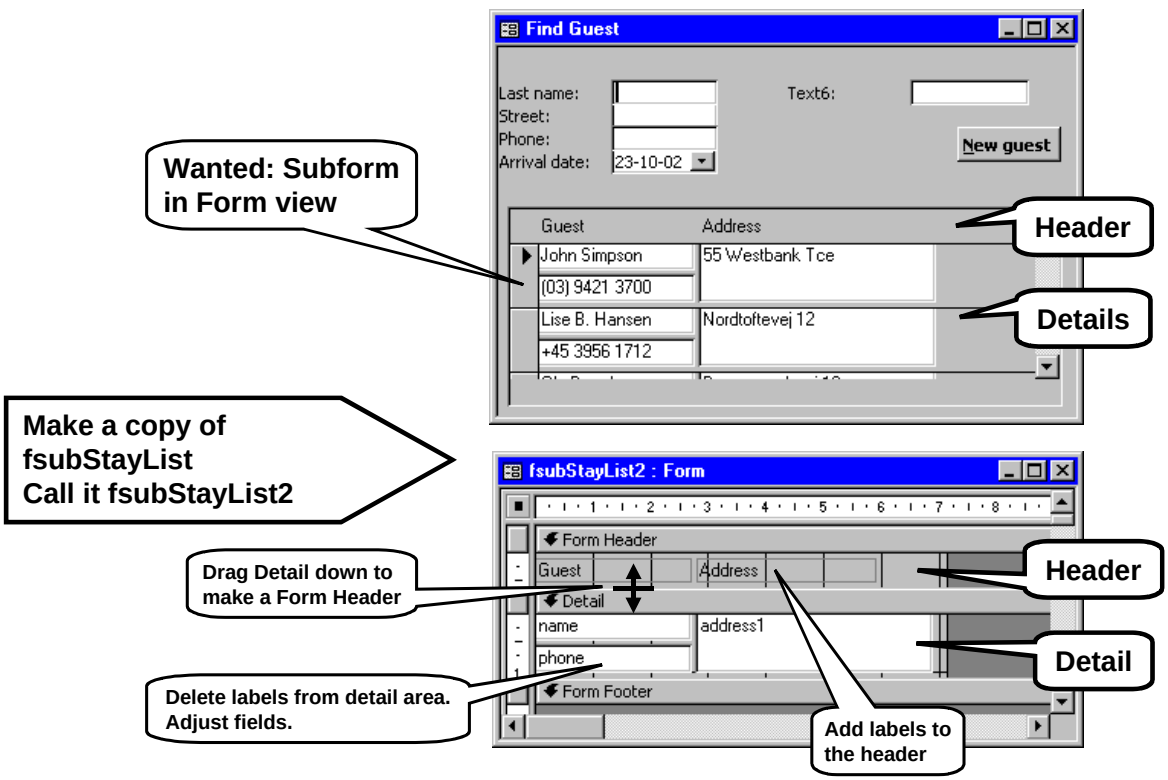


Fig 3.2G Subform in Form view



3.2.5 Summary of subforms

Datasheet view or form view?

Datasheets and forms not only look different. They have other usability differences too.

Datasheet view allows cursor movements in all directions as in a spreadsheet. It also allows the user to select a rectangle of cells. The Visual Basic program can find out which cells the user has selected.

The weakness is that the display format is very restricted. Each record is shown as a single line with text boxes, combo boxes, etc. Also the column headings are very restricted. They are just simple texts and they cannot even be empty.

Form view allows all the available display formats, including pictures retrieved from a database. (There cannot, however, be continuous forms inside another continuous form. Continuous forms can only be used in one level.)

The weakness of forms is that the cursor moves less intuitively. The user can tab through the fields of each record, but not easily move up and down the list of records. Furthermore, the user can only select a full record, not part of a record.

Current record. A subform can show many records at the same time, but only one of them is the *current record*. It is marked with the little arrow to the left, the *record selector*, as shown on Figure 3.2H. (You may change the form settings so that the record selector area is invisible.)

When the user types something into the form, it will always be into the current record. When the cursor moves into another record, it becomes *current*.

Until now we have only seen a main form that is not bound to the database. But main forms may be bound too. Then they have a current record and they need the record selector (see examples in section 4.7).

The subform concepts

Figure 3.2H gives a summary of the many subform concepts we have used above. A main form is a user window with title bar, etc. It may contain one or more subform controls.

Each subform control may be connected to a *continuous form*. The continuous form can be shown in Form view. Then it has a form header and a list of detail forms. The continuous form can also be shown in Datasheet view. Then it looks like a table.

Properties. The following properties (attributes) are important to understand when you work with subforms. Figure 3.2H shows examples of these properties.

Main form: The **Name** property is the designer's form name. It is this form name you see in the database window. You can only change the name there. The **Caption** property is the name the user sees in the title bar of the form. You can set the caption through the *Format* tab in the property box.

Subform control: The **Name** is the designer's name of the control. A program would use this name to address the control. You can set the name property through the *Other* tab in the property box. The **SourceObject** indicates the continuous form connected to the subform control.

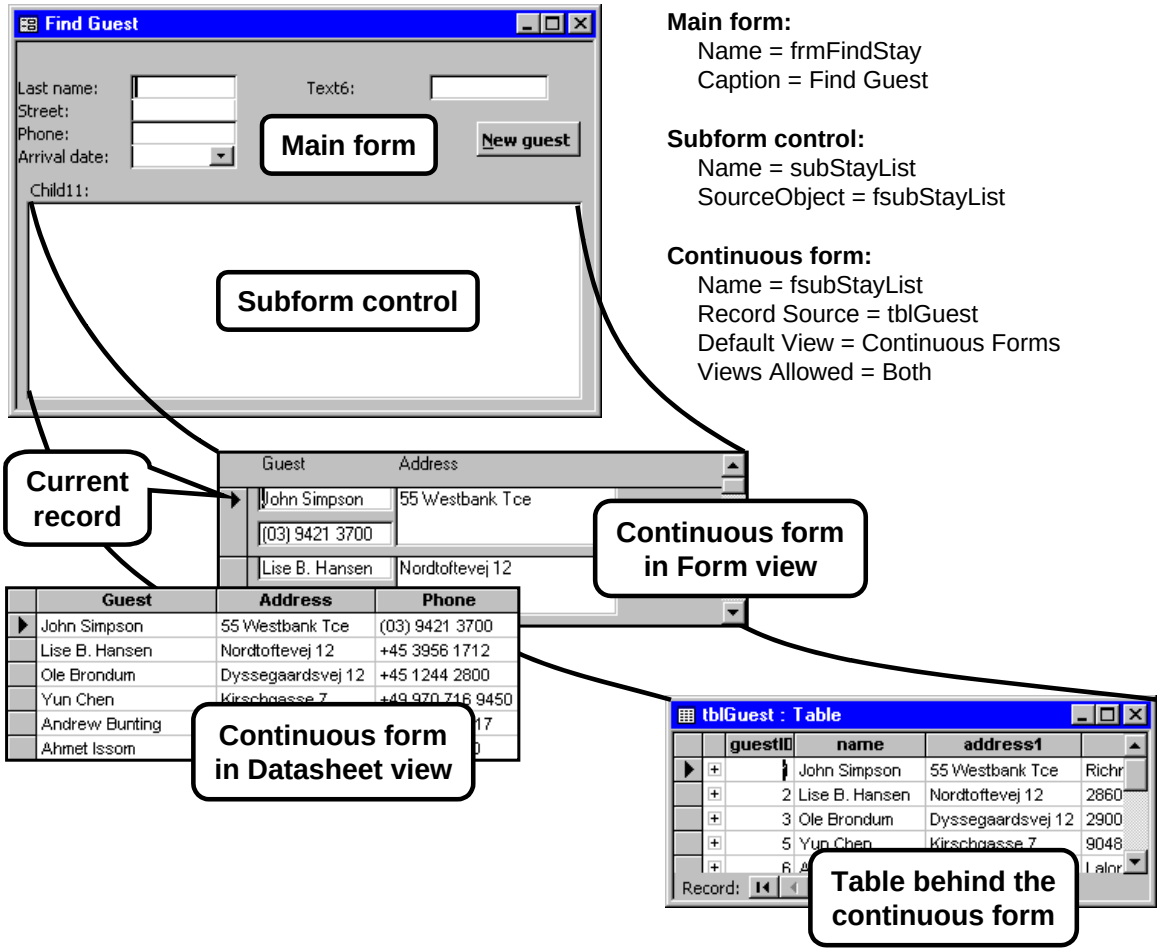
Continuous form: The **Name** is the form name that you see in the database window. **Record Source** indicates the table bound to the form. **Default View** indicates whether the form is shown as a datasheet or as detail forms with a header. Another property, **Views Allowed**, indicates whether the user can change from one view to another.

3.2.6 Prefixes

A prefix is a few letters before a name. The prefix helps the reader understand what this name is about. Is it the name of a table, a form, etc? In just slightly complex systems, prefixes are crucial to help the developers. The bottom of Figure 3.2H shows the full list of prefixes used in this booklet. For subforms the following are important:

tbl	Table names.
frm	Names of main forms.
sub	Names of subform controls.
fsub	Names of continuous forms connected to a subform control.

Fig 3.2H Summary of subform concepts



Main form:
Name = frmFindStay
Caption = Find Guest

Subform control:
Name = subStayList
SourceObject = fsubStayList

Continuous form:
Name = fsubStayList
Record Source = tblGuest
Default View = Continuous Forms
Views Allowed = Both

Control prefixes:

cbo	Combobox control.
chk	Checkbox control.
cmd	Command button
ctl	Other type of control
grp	Option group
lbl	Label
mni	Menu item
lst	Listbox
opt	Option button
sub	Subform control
tgl	Toggle button
txt	Text control

Other prefixes:

bas	Module (used by VBA)
frm	Main form.
fsub	Continuous form (connected to a subform control).
qry	Query
qxtb	Crosstab query
tbl	Table

3.3 Bound, unbound and computed controls

A form may be bound to a table. In this case its controls can be bound to fields of the table, so that the user can see the fields and update them through the control. As an example, *fsubStayList* was bound to the guest table, and we could see and update the guest data. In this section we will look at this in more detail. We will explicitly bind controls and let controls be computed from multiple fields.

Unbound control in a bound form

1. Open fsubStayList2 in design mode.
2. Select the checkbox tool and add a checkbox to the detail form as shown on Figure 3.3A.
3. Switch to user mode. There will be a checkbox in each detail form, but all of them will be gray. The reason is that the checkbox has not got any value yet.
4. Click the checkbox so that it shows a tick. Move to the next record. All the checkboxes have now got a tick. Click to remove the tick. It disappears from all the checkboxes.

As we added the checkbox, it became an unbound control. The Yes/No value is not stored in the database, but it is a single dialog variable in the form. All the checkboxes show this single variable and thus show the same. If you close and open fsubStayList2, the dialog variables disappear and are created again. The checkboxes are gray again.

Bound control

5. Show fsubStayList2 in design mode. Set this property for the checkbox:
   Data -> ControlSource = passport
   (choose the field from the list). Change to user mode.

This action binds the checkbox control to the passport field in tblGuest. The first form instance shows data from the first record in tblGuest, the second instance shows data from the second record, etc. You can bind the control to any field in tblGuest no matter whether it was included when the Wizard generated the form.

In user mode, the checkboxes still look gray but that is because they try to show the passport fields. A checkbox can show a Yes/No value or a number (with zero shown as No). The passport fields are either blank or contain a text, and the checkbox doesn't know what to show.

6. Try to check and uncheck some of the boxes. Notice that they are independent of each other. The Yes/No value is stored in the *passport* field of tblGuest. Look at the table contents. Notice that Yes is stored as -1, No as 0. (Sorry if some of the real passport numbers disappeared.)

Computed control

7. **Combine two database fields.** A text box may be computed from database fields. Try it with the address text box. Set its ControlSource to this expression
   = phone & address2
   (Make sure to type the equal sign too.)

The equal sign changes the checkbox to a computed control. The & is the concatenation operator. We have asked Access to concatenate the two guest fields and show them as a single text. Check in user mode that this is what you get. However, now you cannot enter anything in the address text box - Access has no place to store what you enter.

On Figure 3.3A we have made the combined field look a bit better with a comma and a space between the two parts. As shown on the figure, we have concatenated the control source from three parts, the phone field, a text constant holding the comma and a space, and the address2 field:

= phone & ", " & address2

Don't worry about the square brackets on the figure. Access often adds them as a parenthesis around names in order to deal with names that contain spaces, # and other strange characters.

Troublesome expressions

Sometimes an expression may give strange results because the expression happens to refer to something you didn't anticipate. Here are some examples.

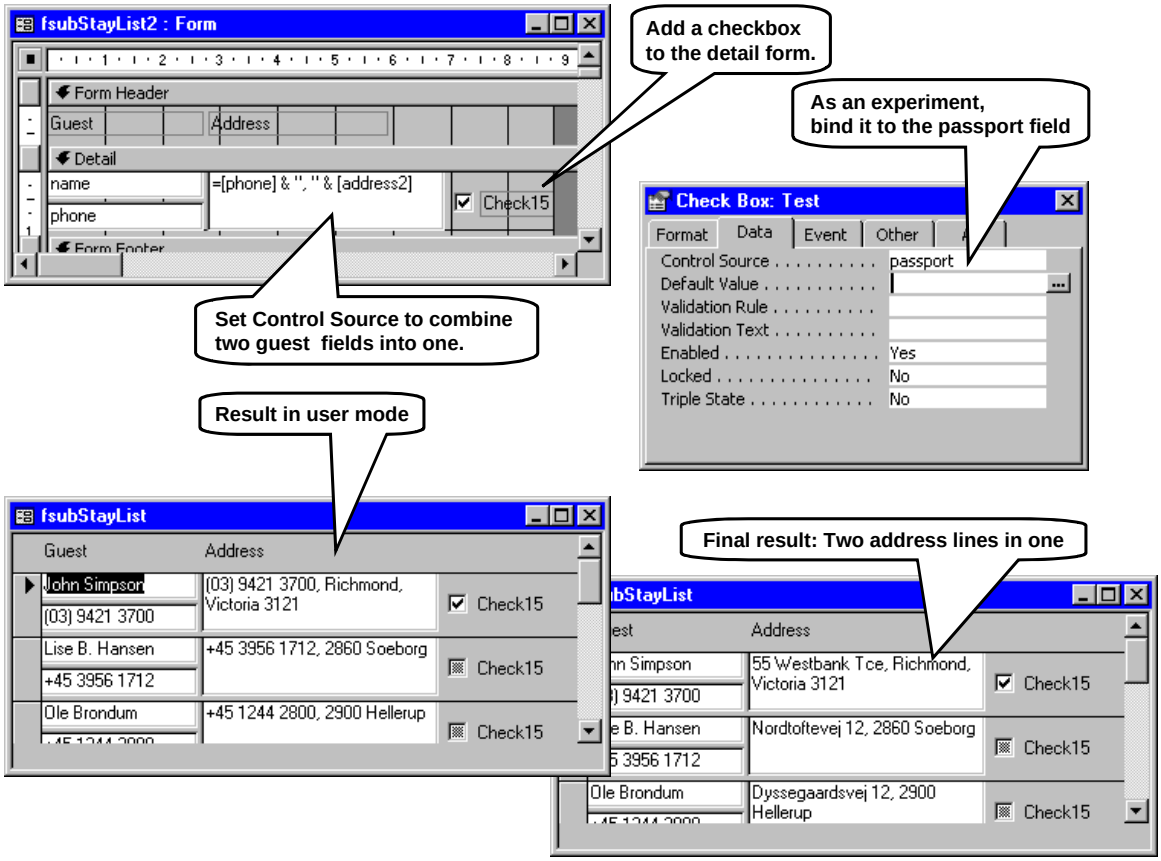
8. **Self-reference**. Concatenating phone and address is just an experiment. It is more useful to concatenate address1 and address2. Try to enter this control source in the address text box:
   = address1 & ", " & address2

Access doesn't give you an error message, but in user mode you will see the text *#Error* rather than the guest data. This highly user-friendly message (;-) is in this case caused by a self-reference:

Notice that the programmer name for the address text box is *address1* - exactly the same as the name of the database field. Access assumes that we want to concatenate the address1 *text box* and the address2 *field*. In order to do this, Access has to compute the address1 text box, but this means concatenating address1 and address2 once again. The computation would never stop and Access shows it with #Error.

9. **Repair self-reference**. In order to repair the problem, give the address1 text box another programmer name. On the *Other* tab, replace the name *address1* with *Address*. In user mode, the form should now look as the last form on Figure 3.3A.

Fig 3.3A Bound and unbound controls



Let us look at another troublesome problem:

10. **Referring to a built-in property**. Let us try to concatenate the guest name with address2. Change the text box control source to:
    = name & ", " & address2

In user mode, you will see that you don't get the name of the guest, but the name of the form itself! The reason is that a form has a *Name* property, and this is what you referred to. Notice that Access changed the *name* you typed to [Name]. This is a sign that Access recognized the name as something spelled with a capital N - in this case a property name.

To avoid the problem, precede the name with a *bang operator* (!):
    = !name & ", " & address2

The bang operator tells Access that you want a control or a database field - not a built-in property. We will discuss this a lot more in section 5.1.

11. **Name mistake**. Try to use a non-existing name in the control source:
    = zz & address2

Access doesn't give you an error message, but in user mode you will see the text *#Name?* instead of the guest data.

3.3.1 Showing subform fields in the main form

In **Access 97 and 2000** you may also let a field in the main form draw on data in the subform. Figure 3.3B shows an example. We let a text box show a field from the subform. Try it in this way:

1. Open frmFindStay in design mode and select the spare text box (Text6 on the figure). Set its control source to
   = subStayList ! name
   (Remember the equal sign.)
   This tells Access to look into the subform control to find the field *name*. Switch to user mode.

The spare text box should now show the guest name from the currently selected record. Try selecting different records to see that the text box is updated automatically.

Detail area. This approach can be useful if you want to show details of the selected record. You might for instance set off an area of the main form for details of the selected record. The area might show several fields from the selected record.

Unfortunately, in **Access 2003** this doesn't work automatically. You can set the control sources, but Access doesn't update the fields automatically. You have to do some programming to update the fields.

Other expressions
You can use a wide range of operators and functions to compute the control value, such as
+, -, *, Sin( ), IIf( )
The rules correspond to what you can write in Visual Basic, and they are similar to those you find in other programming languages. See section 6.4 for the detailed rules.

In practice, you will rarely use complex expressions in the controls. Complex computations are usually done either as part of computing a table with an SQL query, or as part of a Visual Basic program.

3.3.2 Variable colors - conditional formatting

You can determine the colors and other format aspects of a control through the Format tab of the property box. For instance you can set *Back Color* of Guest to something different than white. However, the back color will be the same in every detail form.

Sometimes you can vastly improve the user interface by showing critical data in color. This means that only some of the detail records will have a colored control. It might be tempting to do this by using an expression for *Back Color*, but this is not possible. Before Access 2000 it was actually completely impossible to have variable colors in continuous forms.

Value-dependent color
In **Access 2000 and 2003** there is a primitive way to deal with variable colors. We can for instance let a number be green in the range 0 to 49, yellow in the range 50 to 69, and red otherwise. Or we could let a text be yellow in the alphabetical range from H to P, and red above it. Try it:

2. Open fsubStayList in any mode, for instance Datasheet view to get a good overview of the records.
3. Select the Guest name (the name control). Use the Format menu at the top of the Access window and select *Conditional Formatting*.
4. Set up the conditional formatting as shown on Figure 3.3C. Specify yellow background for the range "h" to "p" and red from "p" and up. You can add or delete conditions by means of the Add and Delete buttons at the bottom.
5. Close the conditional formatting box and check that the colors are as expected. You may also try to enter new guests to check how the ranges work in detail.

The variable colors work in datasheet view as well as form view. The colors do not show in the table behind the form (tblGuest).

Default colors. At the top of the Conditional Formatting box, you can set the default color for the control. You can also set it through the *Back Color* property on the Format tab. However, this only defines the default color in form view. In datasheet view, the default color is defined by datasheet settings (use the main menu: Format -> Datasheet and Format -> Font).

Expression-dependent color
You may also let the color of one field depend on the values of other fields. As an example, we could let the color of the Address control depend on the Guest name. You might try this:

6. Select the Address control (initially named *address1*) and select conditional formatting.
7. For Condition 1, select *Expression Is*. Then select a background color and specify this expression for the background color to be used:
   !name > "h"

When you close the conditional formatting window, all guests with names after H should have the new background color for the address, but they don't! The reason is that conditional formatting doesn't address fields in the same way as the control source property (I would call this a bug in Access). You have to use a more elaborate expression:

8. Use the *full address* for the field
   Forms ! fsubStayList ! name > "h"

The full address tells Access to look for a form called fsubStayList and find the name control in it. This

Fig 3.3B Showing subform fields in the main form

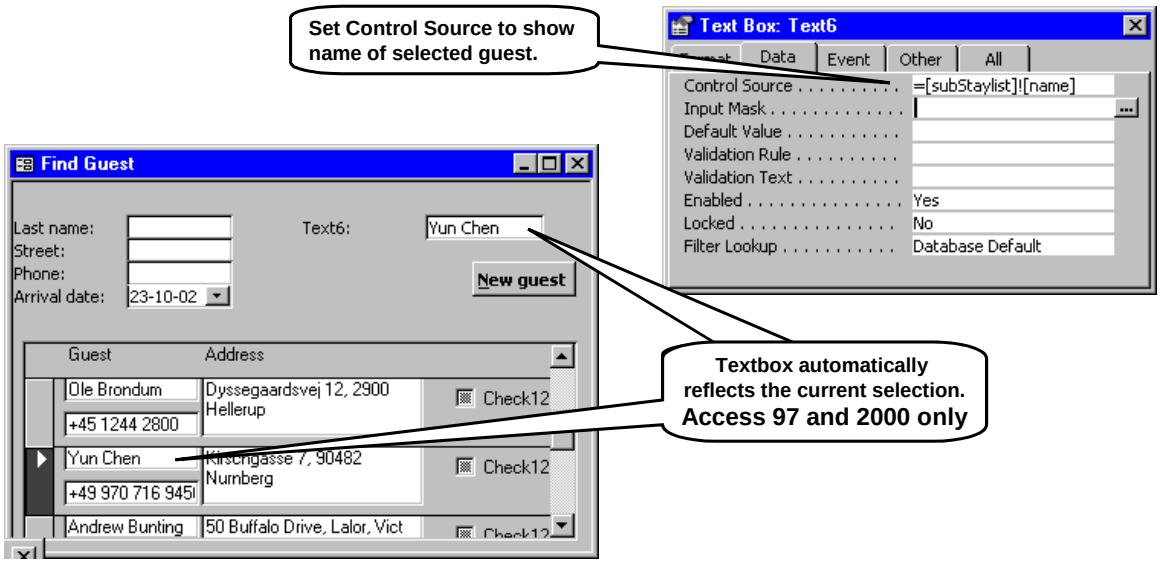
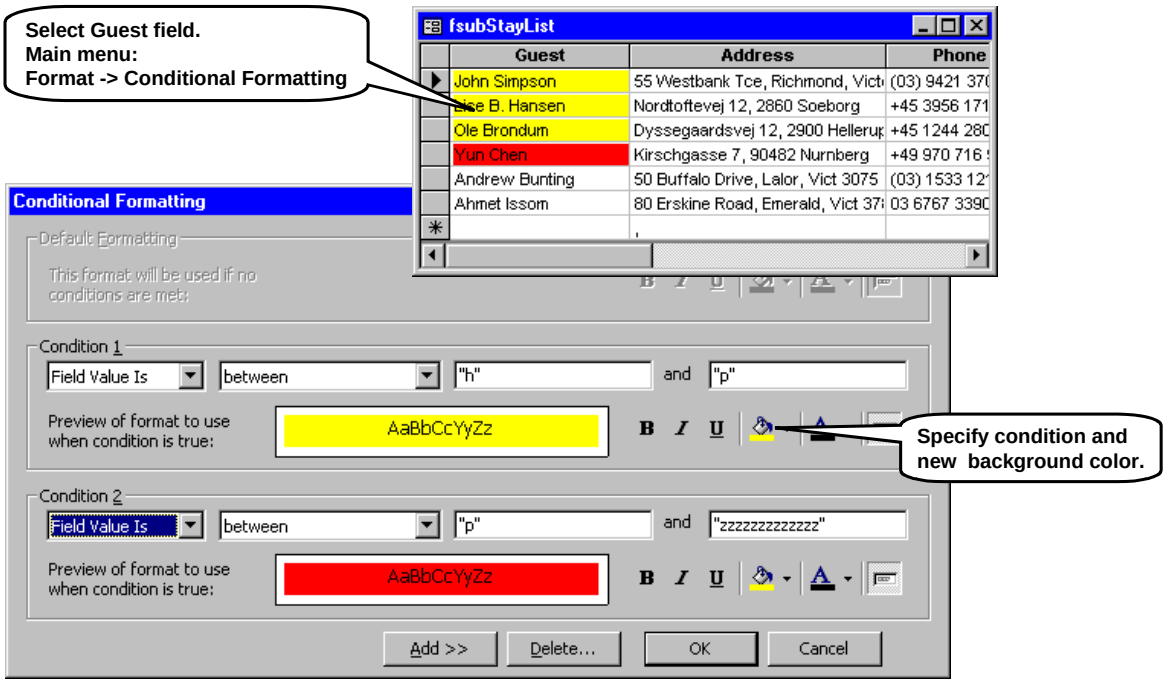


Fig 3.3C Variable colors



should work correctly when you close the formatting box. (In some cases Access shows an error message saying that it cannot find the Forms field. Ignore it, the formula works anyway.)

Unfortunately, this full address works only when we see fsubStayList separately, but not when we see it as a subform of frmFindStay. We will have to use an even more elaborate address to tell Access that it is a sub-form of frmFindStay we talk about. The address to be used is

Forms ! frmFindStay ! subStayList !name > "h"

We will explain a lot more about these addresses in section 5.1.

3.4 Tab controls and option groups

Tab controls

When space is insufficient on the form, a tab control is a way out. The *property* box in Access and the *option* window in many MS-Windows programs use tab controls to put ever more fields on a single window.

Figure 3.4 shows how we use a tab control in the hotel system. The Stay window shows all details of a stay (a booking). It contains one tab control with two tab pages. The first tab page shows the rooms booked or occupied by the guest. The second tab page shows the services received, for instance breakfast. Try to make the essential parts of this form:

1. Create a new form from the database window's Form tab (use *Create form in Design view*). Remember to set the grid size so that the grid is visible.
2. Add the *Stay No.* field to the form (to be used for stayID later), but don't care to add all the other fields.
3. Extend the grid area. Select the Tab-control tool from the toolbox and draw a large rectangle on the form.

You now get three new controls on the form: the tab control and two tab pages in it. When you click in the empty area to the right of the tabs, you select the entire tab control. If you click on one of the tabs, you select this particular tab page.

How to add further tab pages to the control? You right-click the tab and select *Insert Page*. In the hotel system, two pages are enough. We need a subform on each of the two tab pages. Proceed like this:

4. Select the first tab page. In its property box, enter the caption *Rooms*.
5. Make sure the tab-page is selected and not the entire tab control. Select the subform tool from the toolbox and draw a subform on the tab page.
6. Select the text box tool from the toolbox and draw the two fields for totals on the tab page.
7. Also add the last total field (*Total rooms and services till now*). Note that it must be below the tab control. In design mode, the form should now look like the right part of Figure 3.4.
8. Connect the mockup room list to the first tab page in this way. Select the subform control and set its SourceObject property to *fsubStayRooms*. (You created this subform in section 3.2.3.) In user mode, the form should now look like the left part of Figure 3.4.
9. Select the second tab page and give it the caption *Services*. If you like, you can insert subforms and other controls on this tab too, similar to the Rooms tab.
10. Give the Stay form a caption. Close it and give it the name **frmStay**.

Warning: It may be tempting to draw a control on the main form and then drag it to the tab pages. On the screen it may look right, but it isn't. The control sticks to the main form and may be either in front of all the tab pages or behind all of them. You may, however, copy a control from the main form, select the tab page, and then paste the control there. When you click the various tab pages, you see that it sticks to the right page.

Option groups

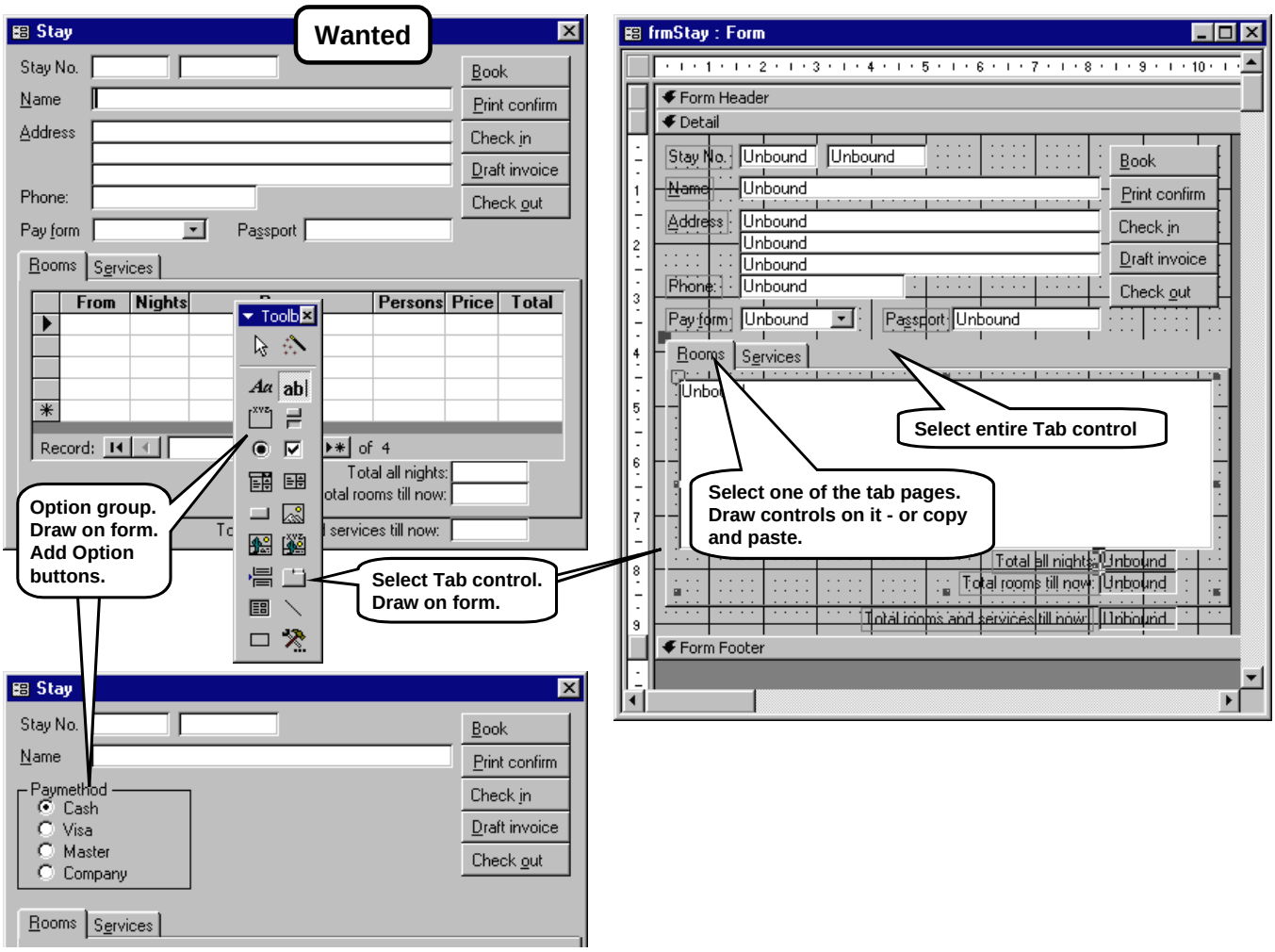
If the user has to choose between a few options, it is traditional to show them as an option group with *radio buttons* (called *option buttons* in Access). The bottom of Figure 3.4 shows an example where the user can choose the paymethod with radio buttons.

An option group is in some ways similar to a tab control. It is a control that contains other controls. However, an option group is just an elaborate way of showing a single integer value. The current value determines which radio button has a dot. In our example this integer would be the paymethod. Make the option group in this way:

11. In design mode, choose the *Option group* tool from the toolbox. Make sure the Wizard is *off*. Draw a rectangle like the one on the figure.
12. You have now got the control and its label. The label is shown across the top border. You can drag them around exactly as text boxes. Change the label to *Paymethod.*
13. Select the *Option button* tool from the toolbox. Drag a rectangle where the option button and its label will be - or just click at the top left corner of this rectangle. Draw four option buttons.
14. Change the labels for the options to *Cash*, *Visa*, etc.

Look at the result in user mode. The buttons are all gray. Why? Because the control has no value - it cannot choose which button to give a dot. Remedy? Click in one of the buttons. If you click the first button, the control will get the value 1. If you click the second one, it becomes 2, etc.

Fig 3.4 Tab control and Option group



3.5 Menus

In Access a form cannot have menus. Only the big Access window can have menus, but they can change according to which form is in focus.

Before we start implementing the hotel menu, we need some terminology. Figure 3.5A shows what we are talking about. Initially, the Access window has a *menu bar* and one or more *tool bars*. Access uses the common term *toolbar* for menu bars as well as real toolbars. The toolbar concept also covers the free-floating toolbars, called *toolboxes*, and *shortcut menus* that pop up when you right click on a control. In a moment we will add another menu bar to be used by the hotel system.

Each menu bar has one or more *menus*. A menu consists of a *menu heading* - the one we see on the menu bar - and the *menu list* that drops down when you select the menu heading.

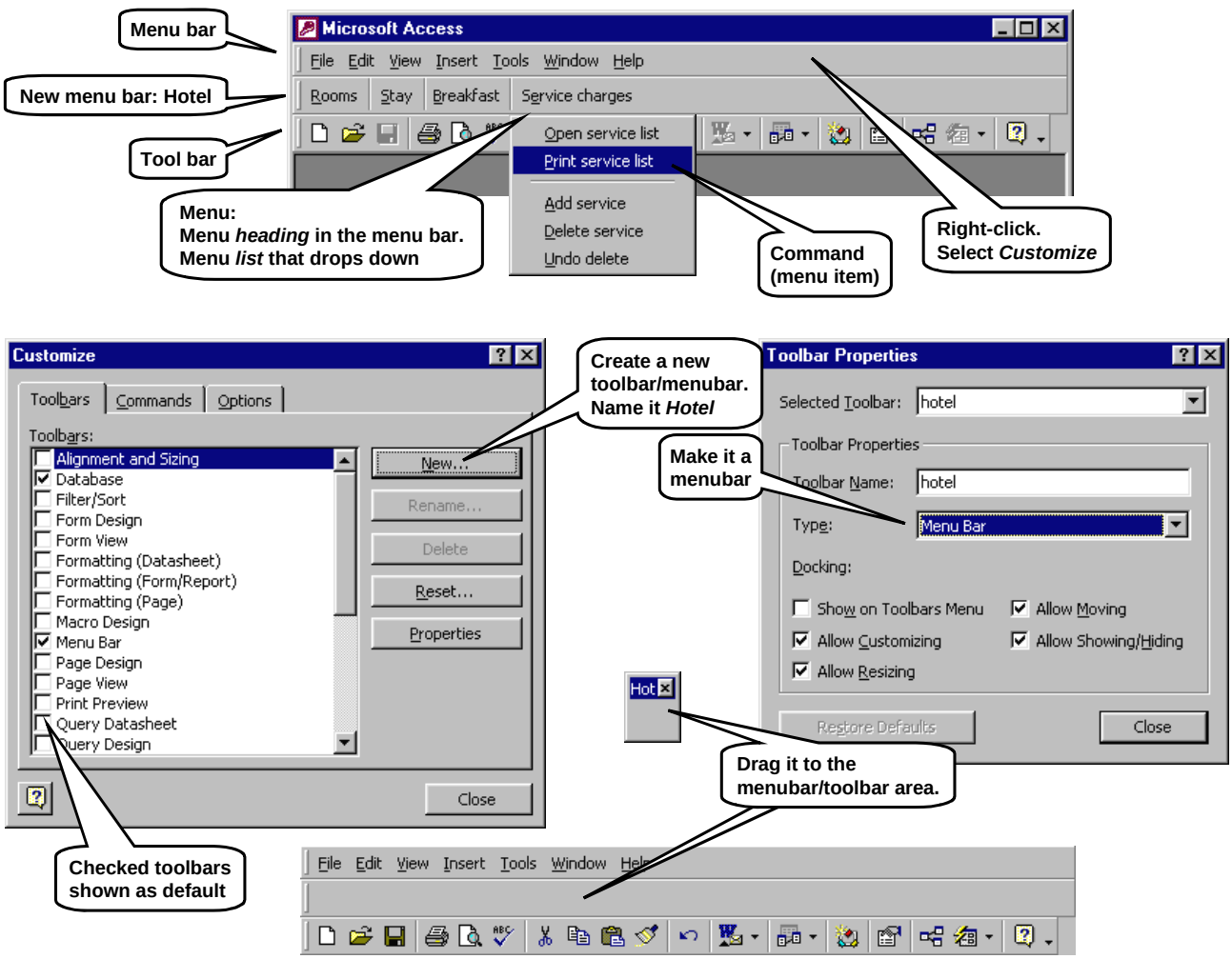
The menu list contains the menu items. Most of them are *commands*. They do the real work when selected. Some of the menu items may be second-level menu headings. They open another menu list when selected, and this is how we make multi-level menus. Access uses the word *command* for the real commands as well as the lower-level menu headings.

3.5.1 Create a new menu bar

You can make menus whether in design mode or user mode.

1. Right-click any point on the existing menus and select *Customize*. (Or use *Tools -> Customize*). You now get the customize window as shown on Figure 3.5A.
2. On the Toolbars tab, click the *New* button. Access asks for the name of the new toolbar. Call it **Hotel** - the name the user will see if he chooses the toolbar with the View menu. Click OK, and up comes a tiny menu bar - floating in the universe.
3. Click the *Properties* button in the *Customize* win-

Fig 3.5A Create a new menu bar



dow and set the *Type* of the new toolbar to *Menu Bar*. Close the properties window.

The type of the toolbar determines where it is shown. If it is a Menu Bar, it will be shown at the top with any other menu bar. If it is a Toolbar, it will be shown below the menu bars.

4. The new menu bar is still floating in the universe. Drag it to the fellow tool bars in the top of the Access window.

The menu bar area should now look like the bottom of Figure 3.5A. The new menu bar looks very empty because it has no menus.

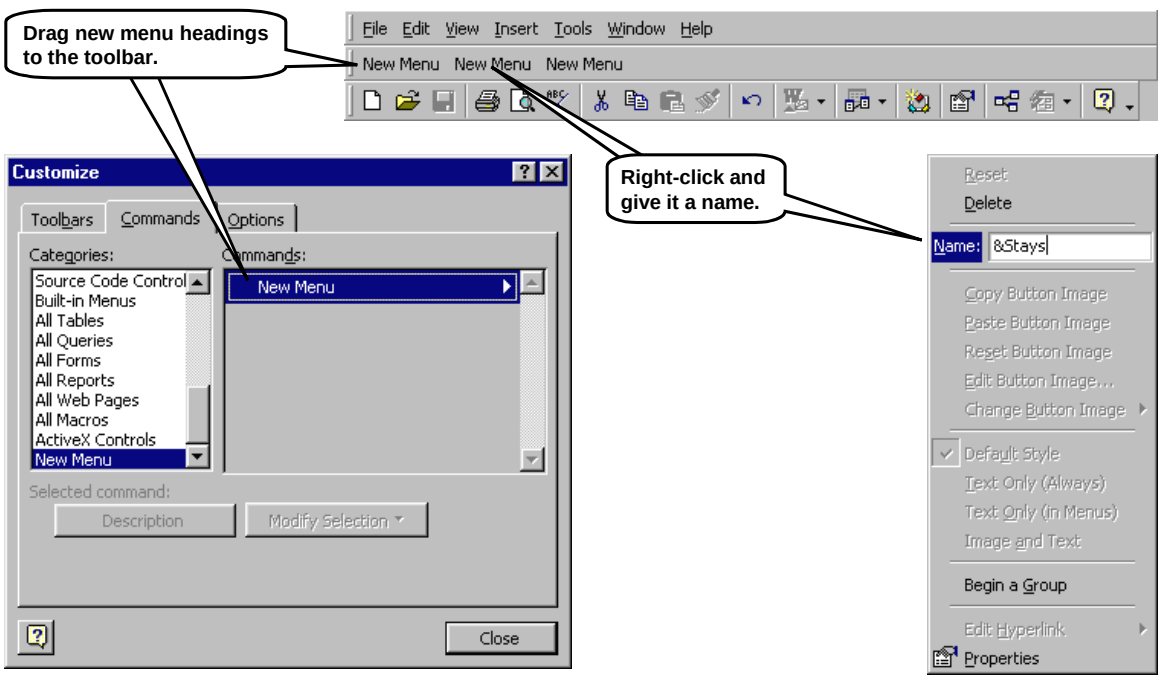
Add menus to the menu bar

5. Select the *Commands* tab in the Customize window (see Figure 3.5B).

This Command tab is really confusing. It allows you to choose between a lot of built-in menu items - those that already are in the various toolbars - but also add new menu items. All these menu items are grouped according to categories. For most commands you can get a short description by selecting the command and clicking the *Description* button.

6. Go to the bottom of the category list and select *New Menu*. This category consists of only one command - a new menu. Select this New Menu command and drag it to the new menu bar. Since we want several menus on the menu bar, drag it a couple of times. The toolbar area should now look like the one on Figure 3.5B.
7. Keep the Customize window open and right-click each of the new menus to give it a name. For the hotel system, use the names *&Rooms*, *&Stays* and *&Breakfast*. The &-sign shows that the following letter is to be underlined and used as a *shortcut* (e.g. Alt+S for *Stays*).

Fig 3.5B Add menus to the menubar



3.5.2 Add commands to the menu list

As an example, we will add two commands to the *Stays* menu: One that opens the FindStay screen and one that cancels (deletes) a stay.

1. Select the *Commands* tab in the customize window and the category *All Forms* (Figure 3.5C).

This category gives access to commands that open an existing form. You see the existing forms to the right.

2. Select frmFindStay and drag this command to the *Stays* menu heading. Wait a moment for the *Stays* menu to unfold so that you can place the command on the menu list.
3. Right-click the command and give it the name *Show FindGuest screen* and the style *Text Only*.

The right-click allows you to do many other things to this command. You can assign an icon to it and edit the icon. (This is more useful when you design toolbars and toolboxes rather than menu bars.) If you click *Properties* at the bottom of the list, you can determine which action the command shall perform when selected. In our case, we use the built-in action *Open a form*.

Add the CancelStay command

4. Select the category *File* and the command *Custom*. (This is the only command that doesn't build on an existing command.) Drag this command to the *Stays* menu and position it properly relative to the other command on the menu.
5. Right-click the command and set its name to *Cancel Stay*.

The *Custom* command has no built-in action. In its property box, we will later add a call to a Visual Basic function (section 5.7) that cancels the stay.

6. Close the *Customize* window and try out the commands. The *ShowFindGuest* command should actually work, while the *CancelStay* does nothing at present.

You may add all the other menus and commands at this stage to complete the mockup, but better spend your time doing it for your own design project.

3.5.3 Attach the toolbar to a form

You can attach a toolbar to a form so that it is shown only when this form is in focus.

7. First hide the Hotel menu: Right click any toolbar to open the customize window. On the Toolbars tab, find the Hotel menu at the end of the list. Remove the check mark and close Customize.

8. The Hotel menu is not visible anymore. Now open frmFindStay in Design view. In the form's property box, select the Other tab. Set the Menu Bar property to *Hotel* and close the form.

When you now open frmFindStay in user mode, the Hotel menu will be visible and it will have replaced Access's standard menu. Open frmStay too and switch the focus back and forth between the two forms. The menus will change accordingly.

If you like, you can make another toolbar and give it the type *Toolbar*. Then attach it to frmStay through the Other-tab, but make it the *Toolbar* of frmStay rather than the Menu Bar. Open both forms in user mode and switch the focus back and forth. Notice that when frmStay is in focus, its toolbar overwrites Access's standard toolbar. When frmFindStay is in focus, its menu bar overwrites Access's standard menu bar.

3.5.4 Startup settings - hiding developer stuff

When the system is finished, the user should not see all the Access menus, the database window for selecting and creating forms, etc. It may be necessary to hide all of this already in the mockup. Here is how to do it:

9. Select Tools->Startup. You now see the startup settings (Figure 3.5D).
10. Change these settings: Application Title = Hotel system (the user sees this name in the title bar instead of the name *Access*). Menu bar = Hotel (the user sees this menu at the top of the screen). Display Form/page = frmFindStay (the user sees this form on the screen initially).
11. Hide the standard things: Full menus and Database window.

This will give the correct view for a mockup. When you later have a functional system, you should disable most other things too, for instance built-in shortcut menus (right-click menus) and special keys, for instance F11 to open the database window.

12. Close the database and open it again. You should now see the naked application window with only the FindStay window and the hotel menu.

Help! How do you get it back to normal so that you can work as a developer? You cannot even change the startup settings anymore. Of course there is a solution:

13. Close the hotel system window again. Now **hold down Shift while you open it**. Keep Shift down until it is completely open. (In Access 2003 this includes answering about unsafe files.)

The window looks again the developer way, and you can change the startup settings. You might leave them in the final user version and remember to use Shift every time you are working as a developer. My experience is that this is too cumbersome and you forget

Fig 3.5C Add commands to the menu list

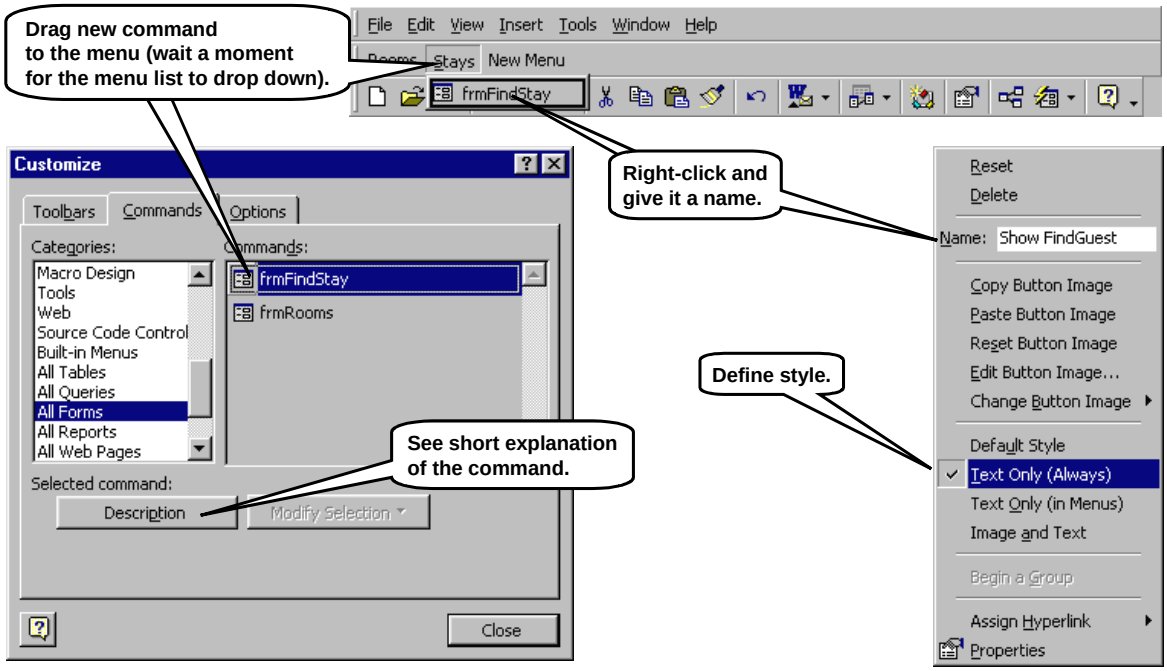
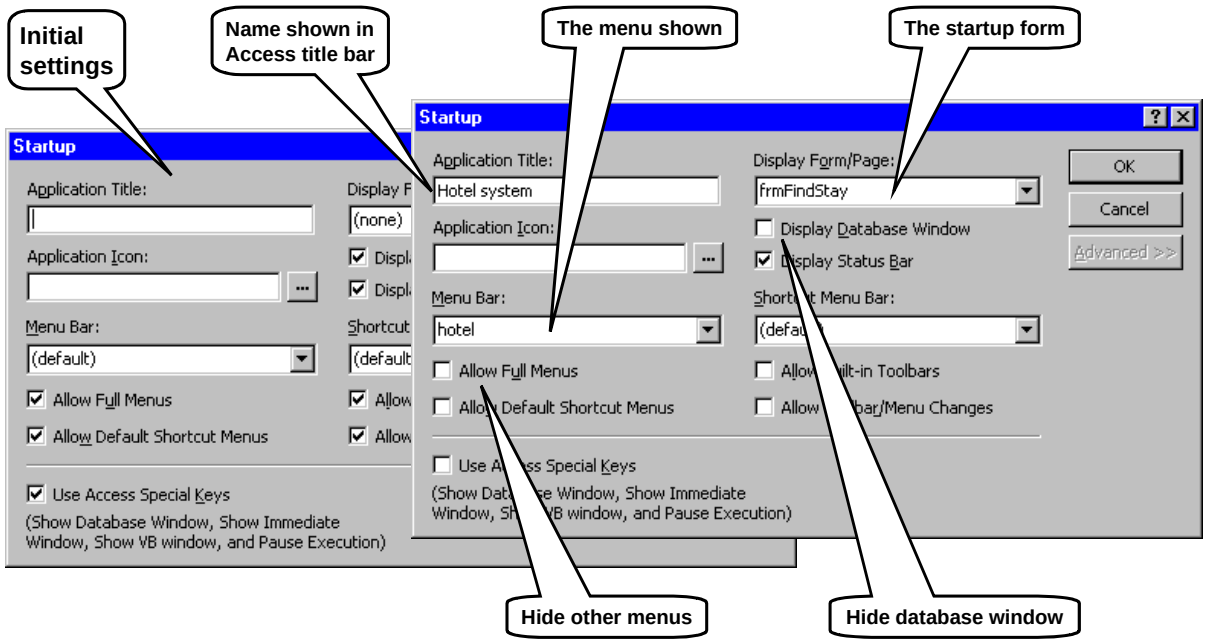


Fig 3.5D Startup settings for hiding developer stuff



about the Shift too often. I always set them back when I have tried it out.

3.6 Control tips, messages, mockup prints

Control tips - pop-up help
In the final system we can add *Control Tips* - the small texts that pop up when the mouse rests a moment on a button or another control (Figure 3.6). We can make them in the tool-based mockup, cut them out, and show them to the user as stickers. Let us make a control tip for the *Find guest* button on frmFindStay:

1. Open frmFindStay in design mode. Click on the NewGuest button to open its property box.
2. On the *Other* tab, enter the *ControlTip* text. There is little space to type in, so use Shift+F2 to open a larger window to type in. You may want to split the text into two or more lines - oops - the window closes! Use Ctrl+Enter to change line.
3. Switch to user mode and check that the control tip works.

Messages

The final system will have many messages to show to the user in various circumstances. They will be deeply embedded in the Visual Basic program, but when using the mockup for usability tests, the facilitator has to bring them up as stickers. So at this stage we have to generate them on the screen and then print them out.

Figure 3.6 shows the easiest way to do it. It works both in design mode and user mode.

4. Open the *Immediate window* with Ctrl+G. (Also called the *Debug window*.)
5. If you work in Access 2000 or 2003, this also opens the Visual Basic window where you can program. Make the Visual Basic window smaller than the full screen (use the restore button next to the cross that closes the window).

Immediate window. The Immediate window is also called the *debug window*. In it you can type Visual Basic statements and have them executed immediately.

MsgBox. We will use the MsgBox function. It has many parameters, but in most cases we just use the first two:

```
MsgBox "the message text to show",
    the buttons to show + the icon to use
```

For our mockup we won't even care about the message text. We just show it as a very long empty text to make the message box wide enough. We print out the message box, copy it and fill in the message text by hand.

6. Enter the first line shown in the *Immediate* window:
```
msgbox "        ", vbYesNo+vbInformation
```

As soon as you click Enter, Visual Basic will execute the statement and show the message box. Print it at this stage (see below). Then play the user's role and answer Yes.

7. Do the same for the next three lines in the Immediate window. This gives you a sample of the four basic types of messages in MS Windows.

The strange words *vbYesNo* etc. are named constants. Actually, *vbYesNo* is 4 and *vbInformation* is 64. When you add them together, Visual basic can see that you want a Yes and a No button, plus the icon for an Information message.

VBA guides. Visual Basic can guide you while you type in the statement. If it doesn't do it by itself, type Ctrl+I after the function name. It will then show the parameters you may type. After the first comma, type Ctrl+Shift+J and it will show a list of the possible constants at this point. You may select a constant from the list and use Tab to insert it in the statement.

Leave Visual Basic. How do you get out of Visual Basic and back to the Access world? Simply close the Visual Basic window. This doesn't close the database window.

Printing the screens

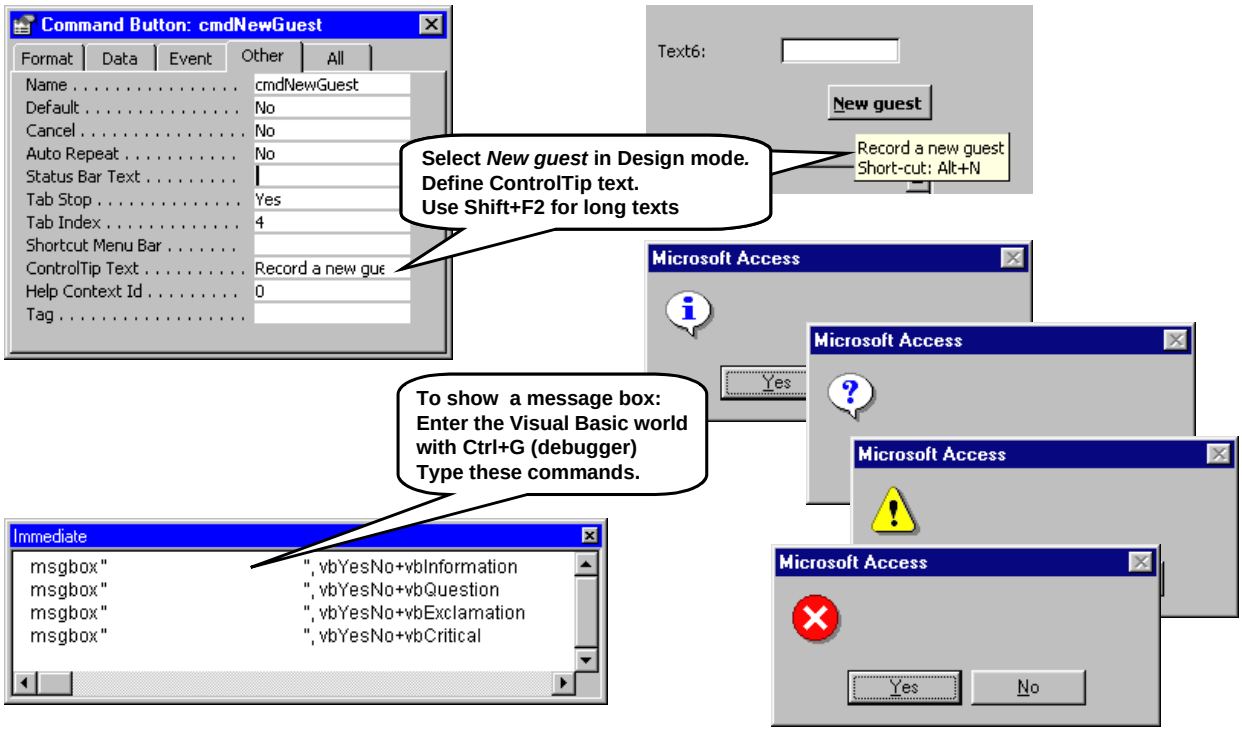
Above we have developed the mockup so that the right things appear on the screen. How do you get them on paper for use as mockups in usability testing? In the standard installation of MS-Windows and MS-Office there is only one way as far as I know:

8. Press Alt+PrtSc (Print Screen). This copies the currently selected large window (the entire Access window) to the clipboard.

When you have selected a message box, Alt+PrtSc copies only this box to the clipboard.

9. Paste it into a Word document.

Fig 3.6 Messages and control tips



In the document, you now see a copy of the entire window (or the message box). For mockup purposes, you will only need a small part of the entire window. Crop away the unnecessary parts as follows:

10. In the Word document, select the screen picture. If necessary, open the Picture toolbox (View -> Toolbars -> Picture).
11. Select the Crop tool from the picture toolbox.
12. Crop the picture so that you only have the necessary part left.

Copy a menu. With this approach you cannot copy a drop-down menu, because it rolls up when you press Alt. To copy a drop-down menu, you have to use Print Screen without Alt. This copies the entire screen to the clipboard. You then have to crop away most of the screen, but it can be done.

You can add as many screen parts to the Word document as you like. Then print it and your mockup is almost ready. What remains is to copy pages, trim message boxes and control tips with scissors, and write data in the various windows. Usually it is best to enlarge the screens when you copy them.

This works but is very cumbersome, and it is almost impossible to crop away exactly the right amount. Further, the document becomes a huge file due to all the large screen dumps (they are there in full size even if you have cropped most of them away).

For professional use, get a screen grabber (a screen capture tool). There are freeware screen grabbers available that work very well. These tools help you copy just the right part of the screen, and many of them can also capture a menu when it is rolled down.

4. Queries - computed tables

Highlights
- Combine multiple tables into one (*Query*).
- Basics of Structured Query Language (*SQL*).
- User-entered search criteria.

In this chapter we will begin making a functional prototype that can show real data and update it. To do this, we often have to combine data from many tables in the database. All relational database systems provide means for combining tables into new tables by means of *queries*. In this chapter we will see how to do it in Access.

4.1 Query: join two tables

In Chapter 3 we had a simple version of the Find Guest window. It showed a list of guests based on data from the Guest table. In the real system we want to show a list of stays, including data about the guest and the room booked. This means that we have to combine data from the Stay table, the Guest table, and the RoomState table.

Our first version of the stay list will just combine the Stay table and the Guest table. The bottom of Figure 4.1 shows the result we want: a single table with some fields from tblStay and some from tblGuest.

Create a query
1. Start in the database window. Select *Queries* and *Create query in Design view*. (In Access 97 select *New* and then *Design view*.)
2. Access asks you to select the tables you want to combine. Select tblGuest and tblStay (Figure 4.1). Click *Add* and then *Close*.

You now see the query design window (middle of Figure 4.1). The top part of the window is an E/R-model, in our case consisting of tblGuest and tblStay. Access has included the relationship from the full E/R-model. It shows that the tables will be combined according to guestID. This is just what we want in our case, but in other cases you have to remove the relationships you don't need and add new ones that you need for the query. These changed relationships are only used in the query; they don't influence the full E/R-model.

You may delete tables from the query window or add further tables by right-clicking in the E/R-model.

In the lower part of the window, you see the **query grid** where we will make a column for each field in the computed table.

3. Drag *stayID* from tblStay to the grid. Then drag *name*, *address1* and *phone* from tblGuest. Finally, drag *state* from tblStay. (You may also double-click the fields.)

You may rearrange the columns by selecting a column and dragging it to another place.

4. Switch to datasheet view. The query table should look like the bottom of the figure. It contains all stays recorded in the database with guest information attached. In the example, John Simpson has three stays and Yun Chen two stays.
5. Save the query and give it the name **qryStayList**. (The standard prefix for queries is *qry*.)

This looks almost too easy. What happens really? Access has made a so-called **join** of tblGuest and tblStay. According to the E/R-model, each record in tblStay has a connecting string to a record in tblGuest. In the query table, there will be one record for each of these strings. If one of the stays didn't have a string to a guest, this stay would not occur in the result.

Since each query record corresponds to a string between two source records, we can include arbitrary fields from both source tables. This is what we have done.

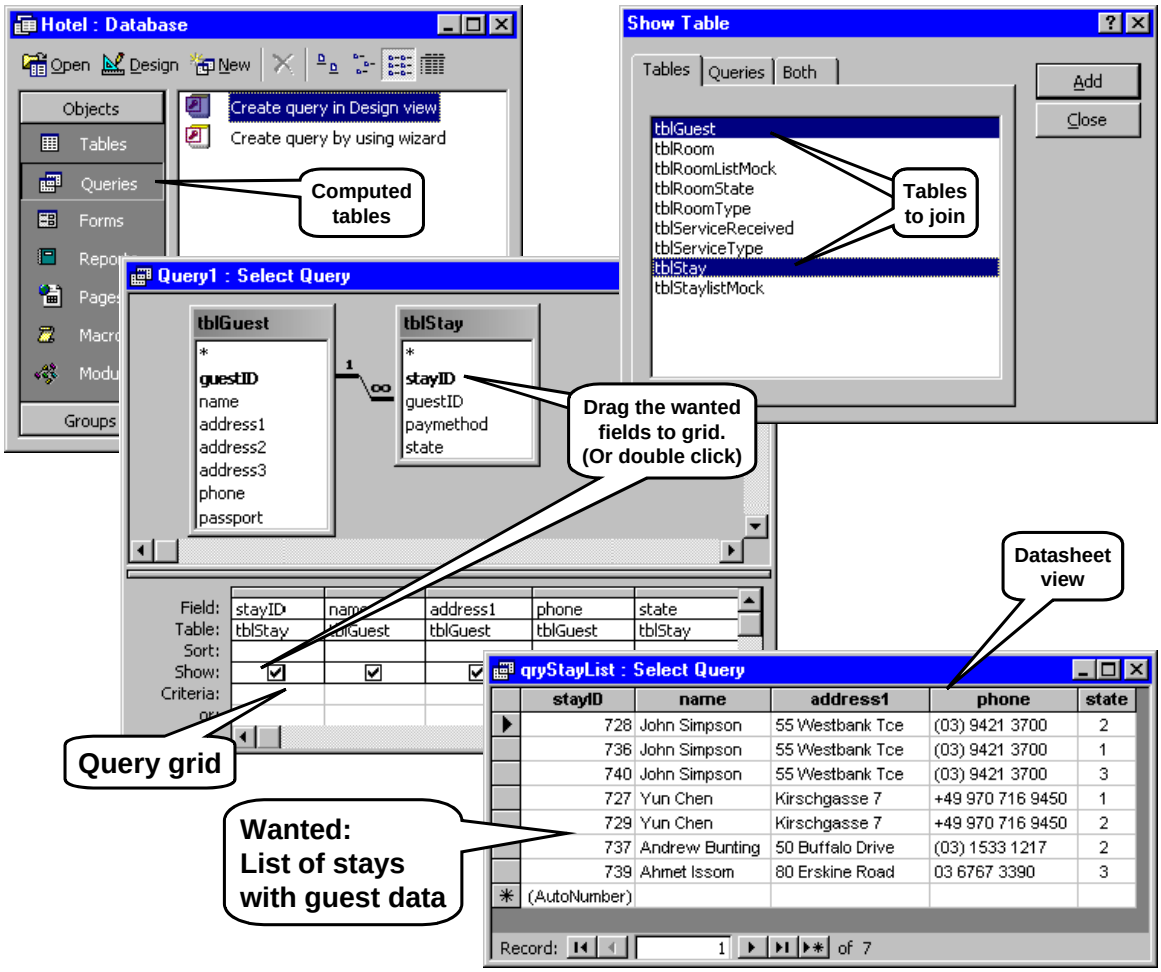
Star = all fields? Note that the data model at the top of the query window has a star in each box. It means "all fields". You may drag it to the grid and all source fields will be included in the query table. This may be convenient, but don't do it yet. Why? Assume that you drag the star from both tables. Then you will have two guestID fields in the result. You then have to refer to them as tblStay.guestID and tblGuest.guestID. This leads to endless confusion later, particularly because some of Access's built-in Wizards cannot figure out about these names and screw things up.

Dynaset and data entry to query table
The query not only shows data, it can also be used for data entry. Try these experiments:

6. Open qryStayList and tblGuest at the same time.
7. In the *query* table, change the name of one of the guests. As soon as you move the cursor to the next line, you will see the change in the query table as well as the guest table.
8. In tblGuest, try to change the guest name again. The query table will be updated immediately.

The query table we look at is a *dynaset* because it is updated automatically. We can enter data into it because it is a simple query where Access can find out how to store the data into the source tables. For more complex queries, this is not possible.

Fig 4.1 Query - join two tables



The query has a property called *Recordset Type*. It is *Dynaset* as a standard, but you can change it to *Snapshot*. Then Access computes the record list when you open the query, and doesn't update it dynamically. In this case you cannot enter data through the query. (How to find the Recordset Property? From the query design window, use *View -> Properties*. But don't change anything right now.)

Adding/deleting records in a dynaset

9. In the query table, enter a guest name in the last line (with star-indication). When you move the cursor up, you have created a new guest record (but not a new stay record). You cannot see it in tblGuest, but if you close and open tblGuest, you will see it. (Using the sort button A/Z on the toolbar will also show the new guest.)
10. In the query table, enter a state in the last line and move the cursor up. Access refuses to do it. It tries to create a stay record, but lacks the foreign key to the guest and cannot preserve the referential integrity. If you had included the foreign key in the query, you could set it now and succeed. Never mind. (Remember to use Esc to get out of the inconsistent data update.)
11. In the query table, try to delete a line. What you delete is the stay record, not the guest record.

Conclusion. The dynaset is suitable for editing data produced by a simple query. As soon as you fill in a field in the new record line (star-marked), Access will try to create a new record. If you fill in a field from the guest table, Access will make a guest record. If you fill in a field from the stay table, Access will (also) make a stay record. When you move the cursor to another record, Access will check that referential integrity and other rules are met.

4.2 SQL and how it works

SQL means *Structured Query Language*. Behind a query is always an SQL-statement that specifies what to compute. Let us look at the SQL statement in qryStayList:

1. Open qryStayList. Select SQL view with the view menu at the top left of the Access window.

The result should be as shown on Figure 4.2. At first sight, an SQL-statement looks overwhelming, but after a while it is not so hard to read. Most SQL-statements are SELECT statements with this structure:

```
SELECT <the fields to show in the result>
FROM <one or more tables>
```

The SELECT part corresponds to the top-two lines of the query grid. Field names may include strange characters, and in this case Access surrounds the name with square brackets to avoid that the name is interpreted as something else. (Often Access surrounds the names with square brackets for no apparent reason.) Examples of names with strange characters:

guest# written as [guest#]
Guest History written as [Guest History]

The FROM part corresponds to the tables and the relationships in the top part of the query window. In our example, the FROM part says that tblGuest and tblStay must be joined, and the join criterion must be that *guestID* must be equal in the two tables.

Actually, Access stores the query as an SQL-statement, and when we want to see the query in design view, Access translates it into a grid and the data model. You can sometimes observe this when you have set up the grid in one way. When you close the query and open it again, the grid looks different, for instance with the columns in a different sequence. Your version and Access's version correspond to the same SQL statement.

You can also type the SQL-statement directly in SQL-view. Access may still be able to show it as a grid, but for some SQL-statements it is not possible. For instance this is the case with a UNION statement, where two tables are to be concatenated one after the other. Access can handle this, but not show it as a grid. Expert Access developers sometimes define a query with a grid, sometimes with SQL, and often they switch between the two during development.

Capital letters. Access doesn't care whether you type with capitals or small letters. For instance you may type SELECT with small letters, but when you close and open the query in SQL view, SELECT will be with capitals. When you define field names and other names, Access remembers the capitalization in the name definition, but accepts names written with different caps as equivalent.

The matrix at the bottom of Figure 4.2 illustrates what the SQL-statement basically does. It finds the result in seven steps (a to g):

a) **Cartesian product.** The SQL-engine takes the first guest record and extends it with all the fields from the first stay record. It corresponds to the top left cell of the matrix. Then it takes the second guest record and extends it with the *first* stay record. This corresponds to the second cell in the top row. And so on for all the guest records. This corresponds to all the cells in the top-row of the matrix. Then it does it once again using the *second* stay record. This gives the second row of the matrix. And so on until we have got a row for each stay record.

In summary, the matrix corresponds to all the possible combinations of a stay and a guest. If tblGuest has x records and tblStay y records, the result will be x*y records. This is called the *Cartesian product* after the mathematician and philosopher Des Cartes.

b) **Join.** Next, the SQL-engine discards all cells where the *join criterion* is not met. In our case, the criterion is that guestID must be equal in the two source tables. What is left are the cells marked with a cross on the figure. Notice that all stays are included because they have a cross in their row. However some guests (for instance guest 2) are not included because no stay is recorded for them in the database.

c) **Where.** We can also specify *Where* criteria, for instance

```
Where state=1
```

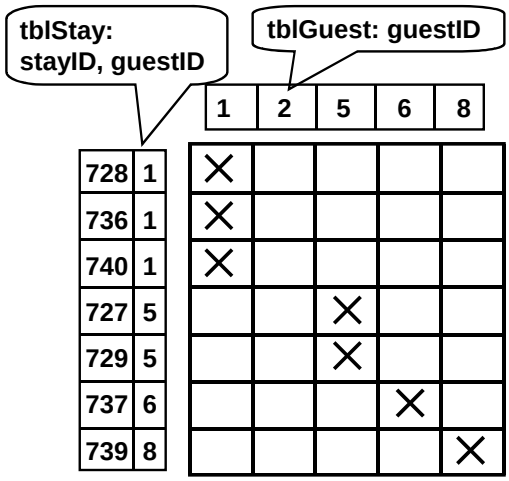
This would cause the SQL-engine to discard all cells where *state* isn't one. They are discarded at this point of the process.

d) **Group By.** We can specify *Group-By* criteria. They cause the SQL-engine to bundle the remaining records according to the Group-By criteria and compress each bundle into one record.

e) **Having.** If we have a *Group-By*, we can also specify *Having* criteria. They tell the SQL-engine to keep only bundles that meet the criterion. Other bundles are discarded at this point.

f) **Order By.** We can ask the SQL-engine to order the remaining records according to some criteria. Some people say that the records are *sorted* rather than *ordered*.

Fig 4.2 What SQL does step-by-step



a) Cartesian product:
5*7 combinations of guest and stay

b) Join:
Include only those where
tblGuest.guestID = tblStay.guestID

c) Where:
Include only those where . . .

d) Group By:
Compress bundles of records to one

e) Having:
Include only bundles where . . .

f) Order By:
Sort the remaining records according to . . .

g) Select:
Select and compute new fields,
discard the rest

g) **Select.** Finally, SQL discards all the fields that we have not mentioned in the SELECT-part. In the grid there is a *Show* indication in each column. It indicates that this column must be in the SELECT-part and be shown in the final table. We will later see that we can compute new fields in the SELECT part. This computation also takes place at this stage.

Notice that these rules tell us that we can have join criteria (and other criteria) that refer to fields that are not shown in the result. We have utilized this rule in qryStayList because the join criterion is given by guestID, which isn't in the result at all.

Although the query result is as described above, the SQL-engine doesn't arrive at it that way. It would take much too long time in large databases. Assume that we joined two tables each with 10,000 records. The SQL-engine would have to create 100 million combination records, and then discard most of them. In practice, the SQL-engine uses index tables and other tricks to create only the "crosses".

Wonder why we talk about Access *and* the SQL-engine? You don't see the SQL-engine directly because Access's user interface hides it. But Access and the SQL-engine are two different software components. We may connect our Access application to another SQL-engine than the one delivered with Access. As an example, we might connect it to an Oracle database. Most of our user interface would still work the same way.

4.3 Outer join

The query we made above used an INNER JOIN in the SQL-statement. Can we also make an outer join? Yes, and we need it right now. Usability tests showed that users had troubles distinguishing guests from stays. The solution turned out to be that whatever the user searched for, the system would show matching stays *and* matching guests.

What we need is a list of all stays, but the list must also include guests that have no stay at present in the database (guest 2 on Figure 4.2 was an example). Now we want to include all guests, no matter whether they have a matching stay or not. This is an OUTER JOIN. Make it this way:

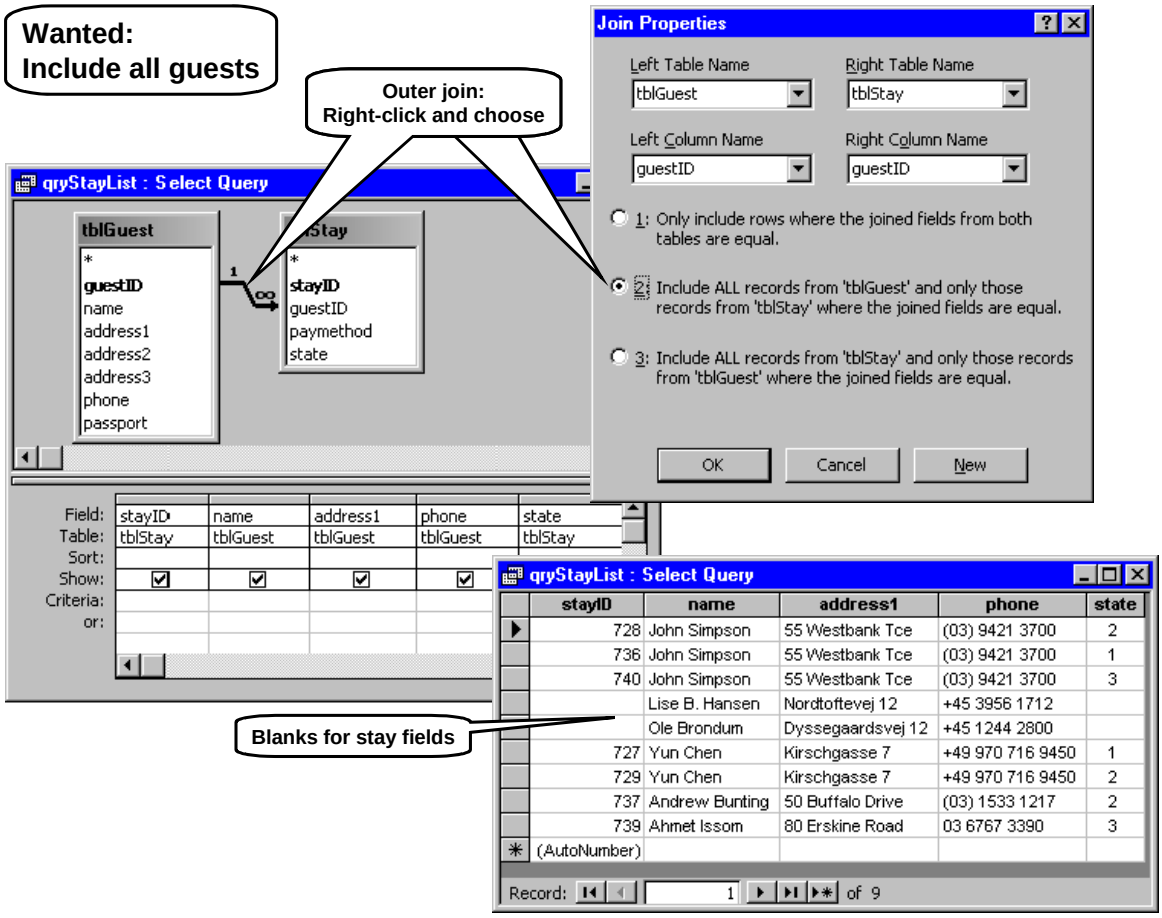
1. Make sure there is a guest in the database without a stay. If not, enter a new guest record now.
2. Show qryStayList in design mode (Figure 4.3).
3. Select the relationship arrow, then right-click it. Choose inclusion of all records from tblGuest.
4. Switch to datasheet view. Now the stay-less guests should appear on the list. These lines will have blanks in the fields originating in tblStay.
5. Look also at the SQL version. It now says

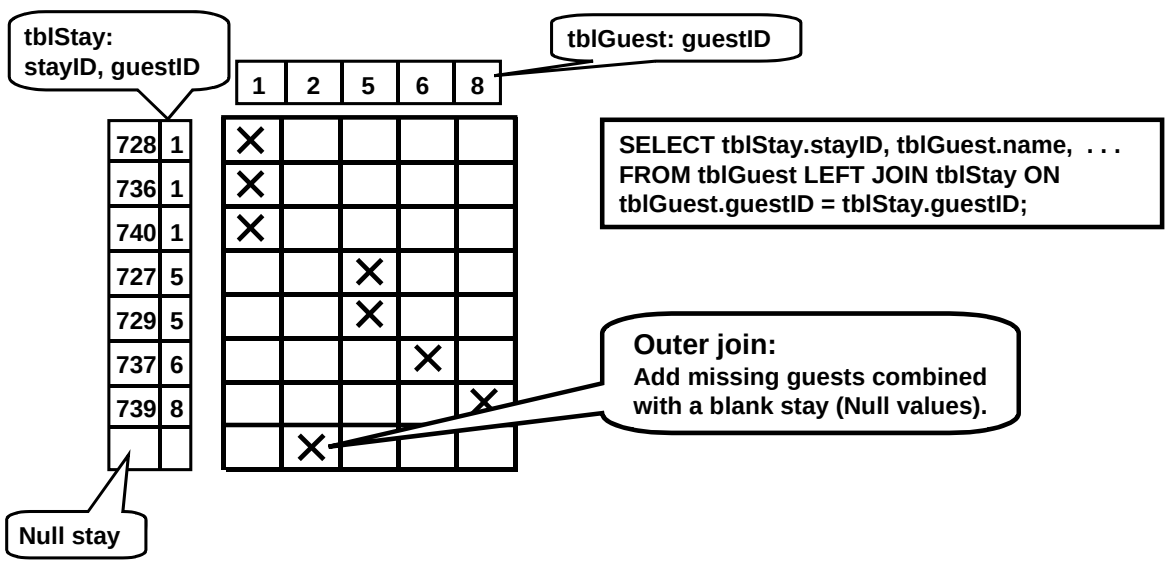
   ```
   FROM tblGuest LEFT JOIN qryStayArrival
   ON ...
   ```

   This means that all records in the left side table have to be included. Right joins exist too, of course. An *outer join* means a Left or a Right Join.

What happens in an outer join? When the system finds a guest without any stay, it combines the guest with a blank stay record and includes it in the result. The bottom of Figure 4.3 shows how this rule would include guest 2. Blank fields have the value **Null**.

Fig 4.3 Outer join





4.4 Aggregate query - Group By

If we want to use the stay list for the real user interface, we still lack something: the arrival date and the room number. In order to get this data, we have to include data from tblRoomState:

arrival = first date among the room states.
room = first roomID among the room states. When there is more than one room, show only the text "more".

We will first compute an extended version of the stay table with these data added to each stay. The small datasheet to the right in Figure 4.4A outlines the desired result. In order to do this, we have to bundle all room states of the stay into one, leaving only the arrival date and the room indication. This is an example of an aggregate query. Here is how to do it:

1. From the database window, create a new query in design mode. Include tblRoomState and tblStay. (See top of Figure 4.4A.)
2. Drag these fields to the grid: *stayID*, *guestID*, *state* (from tblStay) , *date* and *roomID*. You now have an ordinary inner join based on the criterion
   tblRoomState.stayID=tblStay.stayID
3. Change to datasheet view. You should now see a table like the one at the lower left of Figure 4.4A. It contains a record for each of the room states - with some of the stay fields added.

Find the smallest date and roomID. We want to combine all records with the same stayID into one, as shown on the figure. For instance we have four records with stayID = 727. They make up a **bundle** of records. All the records in the bundle have the same stayID, but they have different dates and room numbers. We want to compress this bundle into one record - the one shown to the right.

4. Click the sum symbol on the tool bar (Σ) or right-click in the grid to find the Σ. You will now see a new row in the grid, *Total*.

Initially, Access shows *Group By* everywhere in the total-line. We want to group records by stayID. For date and roomID we want to find the lowest value in each group:

5. Change the total-setting for the date column to *Min*. Do the same for the roomID column. If you now change to datasheet view, you should see a shorter table, somewhat like the one to the right.

The heading for the date column now says *MinOfDate* and it correctly shows the first date for this stay. The roomID column is similar.

You may wonder why we have left guestID and state as *Group By*. Actually, when two records have the same stayID, they also have the same guestID and state, because these columns are fields from the stay table. So grouping by them is alright. You may try to change the setting for guestID to *Min* rather than *Group By*. It makes no change in the result since all records in the bundle have the same stayID and thus the same guestID.

6. Save the query and give it the name **qryStay-Arrival**.

Alias - renaming a column. At this stage the arrival date is computed correctly, but we would like a different name (an *alias*) for the column:

7. Open the query in design view. In the grid, change the heading of the date column to *arrival: date*. This causes Access to compute the first date as before, but give the result the name *arrival*.

Computed field. The last thing missing is that the first roomID may be confusing to the user if there is more than one room in the stay. It is particularly confusing if John Simpson first stays in room 12, then moves to room 11. The stay list would then indicate room 11 and the receptionist might by mistake give him the key for room 11 when he arrives. This is why we want to show "more" if the stay involves more than one room. The receptionist will then have to open the stay window to see which rooms and when.

In order to find out whether there are more than one room in the stay, we compute the Min and Max of roomID and compare them:

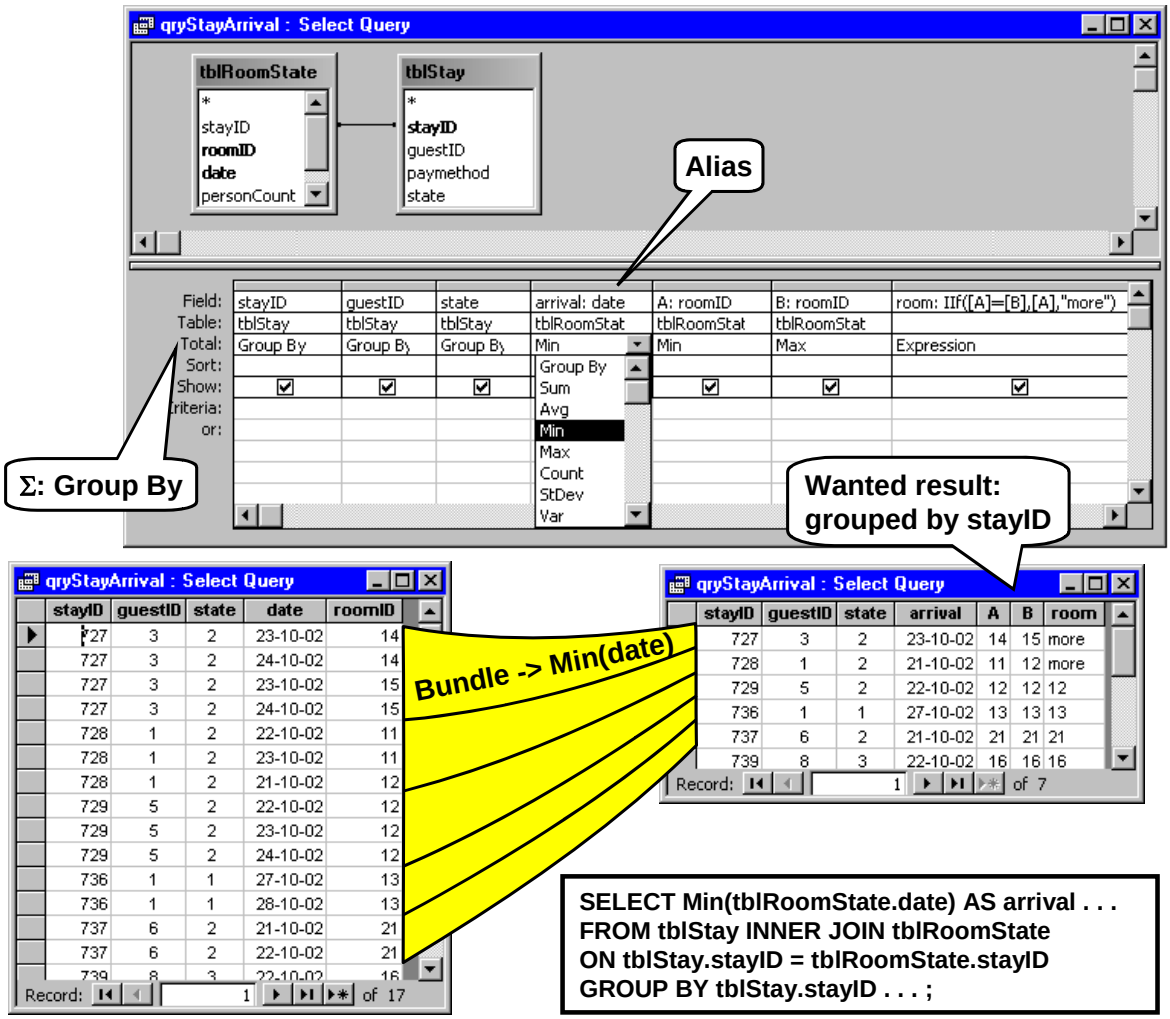
8. Add another roomID column in the grid and let it compute *Max* rather than *Min* (Figure 4.4A). Rename the two roomID columns to A and B as shown.
9. Use a blank column - no dragging of fields. In the total line, indicate that it is an Expression (a computed value). In the top line specify this expression:
   room: IIf(A=B, A, "more")

   Warning: Depending on the regional settings of your computer, you may have to use semicolons instead of commas (see more in section 6.6):
   room: IIf(A=B; A; "more")

This expression says that we want a new field called *room*. If column A (the smallest roomID) is equal to columnB (the largest roomID), then the result shown must be this roomID. Otherwise the result must be the text "more". The operator *IIf* is called an *immediate if*. See the result in datasheet mode. It should be as shown at the lower right of Figure 4.4A. The room column is what we need for the stay list.

Also have a look at the SQL-version:

Fig 4.4A Aggregate query - Group By - bundle records




```
SELECT Min(tblRoomState.date) AS arrival . . .
FROM tblStay INNER JOIN tblRoomState ON
tblStay.stayID = tblRoomState.stayID
GROUP BY tblStay.stayID . . . ;
```

The renaming is in SQL handled by *AS arrival* in the first line. The bundling of records is handled by *GROUP BY* in the last line. Group By works after the Join, but before the Select is made. The Select part can thus refer to any of the fields Grouped By, but what about the non-grouped fields, for instance *date*? If we mention them in the Select part, which of the date values in the bundle should the computer choose? For this reason, non-grouped fields must be accessed via *Min* or another aggregate function.

Select bundles HAVING

We may select only certain bundles and discard the rest. Figure 4.4B shows an example. We want to keep only the stays with arrival 21-10-2002 (European date format).

10. In the grid column for arrival, set the criterion as shown in the figure. The result should be as shown at the lower right of the figure.

Also have a look at the SQL-version:

```
SELECT Min(tblRoomState.date) AS arrival . . .
FROM . . .
GROUP BY tblStay.stayID
HAVING Min(tblRoomState.date) = #10/21/02# ;
```

Notice that in the query grid, we could just set the criterion on the *arrival* column. It is tempting to make a similar thing in SQL:

```
SELECT Min(tblRoomState.date) AS arrival . . .
FROM . . .
GROUP BY tblStay.stayID
HAVING arrival = #10/21/02# ;
```

However, this fails because SQL doesn't compute new fields such as *arrival* until all the criteria have been handled. Thus *arrival* isn't available at the time when HAVING is dealt with. So in SQL you have to repeat the Min-expression in the HAVING clause.

Also notice the date formats. In the query grid we use the regional date formats, but in SQL we must use US date formats.

Other aggregate functions

Above we have used two aggregate functions, Min and Max. There are others available. Here is a summary of them:

Count(x) The number of records in the bundle. Records with a null value in column x are not counted.

Sum(x) The total of the non-null values in column x of the bundle. Null if all values are null.

Avg(x) The average of the non-null values, i.e. Sum/Count. Null if all values are null.

Min(x), Max(x) The lowest (highest) non-null value. Null if all values are null.

First(x), Last(x) The first (last) value in column x of the bundle. May be null. Although Access allows it, never use these functions in a query. The reason is that the sequence of the records is arbitrary during the SQL-engine's work - even if you specify that the input or the result be ordered in some way. The result will be a random record in the bundle.

First and Last are not parts of SQL because the result of a true SQL query is a set of records, not an ordered list of records. It makes no sense to talk about the first or last in a set. Ordering of the records into a list only makes sense when the records are used outside SQL. As an example, First and Last may be used in Visual Basic to find the first or last value in a list shown to the user.

Var(x) The variance of the values in column x. A variance is used in statistical analysis. It shows how much the x values in the bundle deviate from the average x value.

First the Var-function computes the differences between each x and the average in the bundle. Then it squares the differences and finds the total of the squares. Finally it finds the average squared difference by dividing with Count(x)-1.

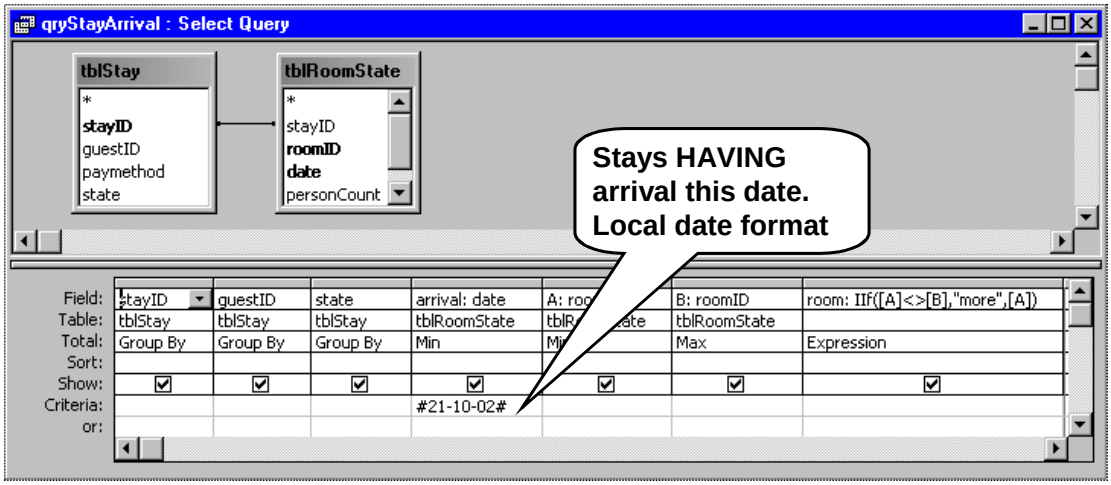
VarP(x) Similar to Var(x) but divides with Count(x) rather than Count(x)-1. You cannot select VarP in the query grid, but you can specify it in SQL.

In statistics, Var(x) is suited if the group is a sample of a larger population, while VarP(x) is suited for the entire population.

StDev(x) The standard deviation of the values in column x. It is the square root of Var(x). While Var(x) exaggerates large deviations, StDev(x) is more like an average deviation.

StDevP(x) The square root of VarP(x).

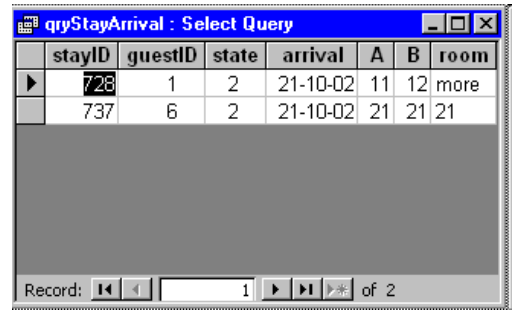
Fig 4.4B Select bundles HAVING . . .




```
SELECT Min(tblRoomState.date) AS arrival . . .
FROM tblStay INNER JOIN tblRoomState
ON tblStay.stayID = tblRoomState.stayID
GROUP BY tblStay.stayID
HAVING Min(tblRoomState.date) = #21/10/02#;
```

HAVING arrival this date

US date format



Aggregation and expressions

You can specify the aggregate functions in the total-line of the grid, but you can also use them in expressions that compute a field. For instance we could have spared column A and B above, and instead written this expression in the *room* column:

```
room: IIf (Min(tblRoomState.roomID) =
          Max(tblRoomState.roomID),
          Min(tblRoomState.roomID), "more")
```

This will only work, of course, in a Group By query.

You can also aggregate computed values that combine data from more than one table, such as this:

```
Min(tblA.x + tblB.y )
```

Dynaset

A Group By query cannot be used as a dynaset although you can set its Recordset property to *Dynaset*. It will not be updated dynamically, but the VBA program can ask for a requery of the list. The user cannot update fields in a Group By query - not even the Group By fields, which in principle might be updatable. See section 4.7.1 for ways to deal with it.

4.5 Query a query, handling null values

At this point we have a query that produces an improved stay list with arrival date and room indication. Queries can in many ways be used exactly as tables. Now we want to join the improved stay list with the guest table to get the final stay list in the Find Guest window. The procedure is similar to when we made the first qryStayList:

Outer join qryStayArrival with tblGuest

1. First delete the old qryStayList from the database window (or rename it to qryStayList2).
2. Create a new query in Design view. Add tblGuest and qryStayArrival to the query window (see Figure 4.5).
3. Access may guess that you want to join tblGuest and qryStayArrival on guestID. Otherwise, you have to create this relationship by dragging guestID from one table to the other.
4. Make the join an outer join so that all guests are included. (Select the connector and right-click.)
5. Drag the fields to the grid as shown. Include the guestID from tblGuest. It will not be shown to the user in the final form, but we need it to make the program know which guest the user points at.
6. Close the query and give it the name **qryStayList**. Open it again in datasheet mode.

In datasheet mode, you will get something like the table at the bottom of the figure. We are very close to the final result, but the guest lines without stay look wrong. They appear in the middle of the list - looks stupid. Second they have a room indication "more". Very confusing - there is no stay for these guests, so how can Access come up with this?

Sort the list. Let us first fix the ordering problem. It would be better to sort the list by arrival date, and then put the stay-less guests at the end. It is easy:

7. In the grid, use the *Sort* line for *arrival*. Set it to *Decreasing*. When you see the list in datasheet mode, the latest arrival is first and the stay-less guests last. SQL considers the null value smaller than any other value, so the stay-less guests are last. (Below we will look at a way to order the stays according to increasing arrival times, yet have the stay-less guests last.)

If you look at the SQL version, you see that SQL expresses the sorting as ORDER BY arrival.

Null=null?

The erroneous "more" indications are more puzzling. Let us look closer at the way a query is computed. In section 4.2 we explained that after discarding the useless Cartesian combinations, the SQL-engine computes the final fields in each record. The room field is a computed field, and what have we asked SQL to do? Compute this expression:

```
room: IIf(A=B, A, "more")
```

For the stay-less guests, both A (the lowest room number) and B (the highest room number) are null because the guest is combined with an empty stay record. Now, what is the value of *null=null?* You might think it is *True*, but not so in SQL. Actually the value is *null*, which is neither True nor False.

Think of null as "unknown". Is *unknown = unknown*? Hard to know, in other words it is "unknown".

The general rule is that any expression with a *null* somewhere is *null*. So *null+7 is null* and *null>7 is null*. Similarly *null=7* is *null*.

Hey, how do you test whether something is null? Asking if x = null will always give null. The IsNull function helps us out:

IsNull(x) is true when x is null.

There are a few other exceptions to the general rule:
Null and False is False.
No matter what the unknown is, the result will be false.
Null or True is True.
No matter what the unknown is, the result will be true.

"Abc" & Null is "Abc"
The & operator always converts both operands to texts as far as possible, so Null becomes an empty text.

We might try to repair the room expression by means of the IsNull function, but there is a simpler solution in this case. We just ask whether A is *different* from B and if so, the result will be "more":

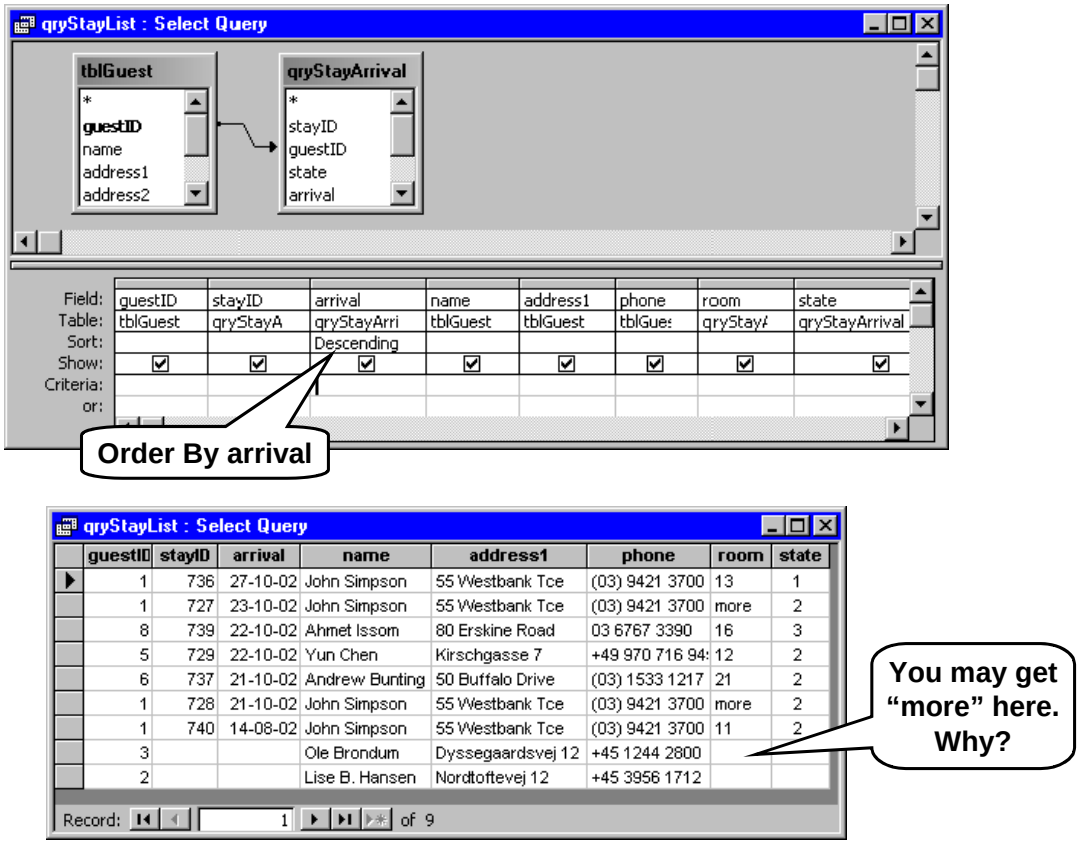
8. In the grid for qryStayArrival, change the room expression to this:
   ```
   room: IIf(A <> B, "more", A)
   ```
9. Save qryStayArrival and reopen qryStayList. The result should now be right.

When A and B are null, A<>B will be null and SQL will select A. *True* is required to select "more". So the result will be A, which is *null*. Just what we want.

Sorting null last. As a finishing touch, we will fix the ordering problem. The trick is to compute a field where null values become higher than anything else, and then sort by this field:

10. In the grid for qryStayList, use an empty column. Enter this expression in the top line:
    ```
    s: IIf( IsNull(arrival), 365000, arrival)
    ```

Fig 4.5 Query a query, Null values



Here we compute a field *s* by means of another IIf-expression. We ask explicitly whether *arrival* is null, and if so use the large value 365000. Otherwise we use *arrival* as it is. Look at the result in datasheet view. Notice that 365000 is shown as a large date around 1000 years ahead. Why? Remember that a date is stored as the number of days since December 30, 1899. Since a year is around 365 days, we have thus specified a date around 1000 years after this date.

(Remember the year-2000 problem? We have now created a year 2899 problem. We don't care - we will all be dead at that time. Or should we care anyway?)

11. Now remove the Sort indication from the arrival column, and set an *Increasing* indication for the s-column. Check in datasheet view that it looks right.
12. Tiny problem remaining. We don't want the s-column to appear in the result. In the grid, remove the *Show* indication for the s-column. Check that the list is sorted correctly and doesn't have the s-column.
13. Close qryStayList and save the changes. Now open it again and look at the query grid. The s-column is still there, but it is not called "s" anymore. This is an example of Access changing the grid behind our back. No harm is done, because it still works correctly. Access translated the grid into SQL and when we opened the query again, it constructed the grid from the SQL version. Try to look at the SQL version to see what the Sort indication and the Show indication becomes in SQL.

At this stage we finally have a stay list that is worth showing in the user's Find Guest window. In the next section we will use it for this purpose and also do a bit about the search criteria.

4.6 Query with user criteria

Queries are often used in combination with search criteria. The user enters a criterion and the system shows a list of matching records. This is what we want in the Find Guest window. First we will make a continuous form based on qryStayList. The procedure is similar to when we made the first fsubStayList:

Create the continuous form

1. In the database window's Form tab, use the Form *Wizard*. Choose qryStayList as the base for the form.
2. Select inclusion of all fields except guestID. Choose the *Columnar* layout to get a continuous form with labels, next choose the *Standard* style.
3. Save the form and give it the name **fsubStayList**. (Replace the old fsubStayList or rename it.)

We are going to use the subform in datasheet view, which gives a compact format and easy navigation. If you look at the subform in datasheet view, you will notice that the column headings are programmer-oriented rather than user-oriented. Furthermore, the state field is shown as a code rather than as the mnemonic text *booked*, *out*, etc.

We want the datasheet to look like Figure 4.6.

Change headings, etc.

4. Open fsubStayList in Design view.
5. For each of the fields, change its label to the user-oriented name.
6. For the state field, also change a few things on the Format tab: Width of the first column should be 0 in order that the state code is not shown, but only the mnemonic text (section 2.4). Also set Text Align to Left.
7. For the form itself, set Default View and Views Allowed, remove the navigation buttons. Also set the Caption to fsubStayList rather than the query name. (The user will not see the caption when the form is used as a subform, but the designer can easily get confused when seeing the query name in the title of the subform.)

Check the result in datasheet view. Then it is time to use it the FindStay form:

Connect continuous form to master form

8. Open frmFindStay in user mode (Form view). Since the new continuous form has the same name as the old one, the result should look like the middle of Figure 4.6. To make it look like the final system, you would have to enlarge the FindStay form and the subform control. You would also have to add more criteria fields and command buttons. It is not important to do it in this exercise.

Simple search criteria

In the real system, there will be thousands of guests and stays, so search criteria are essential. Start out with some simple experiments with search criteria:

9. Close frmFindStay and open qryStayList in design mode.
10. In the Criteria line of the grid, fill in a criterion for the name column (bottom of Figure 4.6). Try entering
    ```
    = "john simpson"
    ```
    Change to datasheet mode. Now you should only see John Simpson's stays. (The SQL-engine doesn't care about capitalization.)
11. Try this too:
    ```
    < "john simpson"
    ```
    You get all guests alphabetically before John Simpson.
12. Try the criterion "john". There will be no matches since no name field contains just "john".
13. Now try with a wild-card expression:
    ```
    Like ("john*")
    ```
    You get all guests with names starting with "john". Try
    ```
    Like ("*s*")
    ```
    You get all guests with an "s" in their name.

The operator *Like* is a kind of equal sign. It just interprets the stars in the text string as "any characters at this place". Note that we can have stars in several places, for instance before and after the text we look for. *Like* is also called the *wildcard operator*. See more in section 6.4

User-specified search criteria

In order to let the user specify the name part he is looking for, we have to retrieve the criterion at run time. We want to specify a criterion something like this

```
Like ("* <user text> *")
```

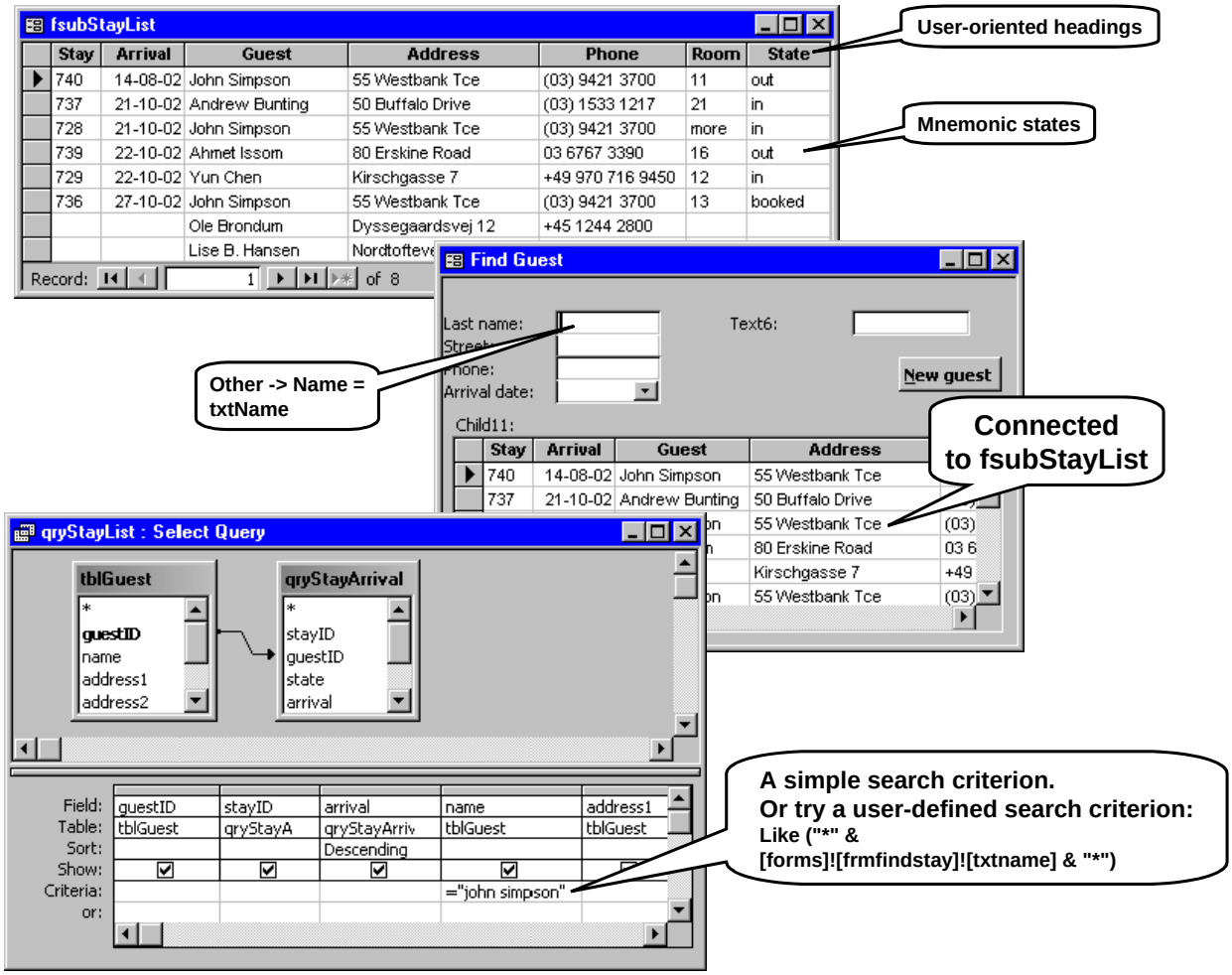
However, the Like operator knows about the stars, but not about finding the user text. We have to combine the text by means of the &-operator from three texts: the first star, the user text, and the last star. The expression will be something like this

```
Like ("*" & <user text> & "*")
```

In our case, the user text will be in the *Last-name* text box on the master form. We will call this control *txtName*. In Access we can then address the text box with this expression:

```
Forms!frmFindStay!txtName
```

Fig 4.6 Query with user criteria



This strange-looking expression means: Look in the collection of forms to find frmFindStay. Then look in the collection of controls on this form to find txtName. This is Visual Basic's way of addressing objects, and in the next chapter we will look more closely at how it works.

Ready to combine the pieces?

14. Open qryStayList in design mode. Set this criterion expression in the name column:
    Like ("*" & forms!frmfindstay!txtname & "*")
    Don't worry if Access adds brackets as usual.
15. Close qryStayList and open frmFindStay in design mode.
16. Select the *Last-name* text box and set its name to **txtName** (on the Other tab). The prefix *txt* is the standard prefix for text boxes.
17. Switch to Form mode. The list looks as usual. Enter for instance an s in the text box. Then click F9 for requery. You should now see only guests with an s in their name.
18. Try other criteria. When you have a blank text box, the system shows the entire list. Why? In this case it looks for "**", meaning anything followed by anything. All names will match!

In the final system, the user should not use F9 to search, but our carefully planned and tested buttons and their shortcuts. In a live search, the system will respond character by character as the user enters a criterion. All of this needs some Visual Basic programming, and we will show it in section 5.2.2.

4.7 Bound main form and subform

In many cases we want to bind a main form (a window) to the database. One example is the stay window. It should show data about a single stay and guest (Figure 4.7). On the stay window, we also have a list of rooms, and it should show the rooms for this stay.

As you see, we need two queries: one to combine stay and guest to supply data for the main Stay form; another to extract room data for the list of rooms. We will now outline how to do it. We will shorten the explanation a bit and leave it to you to open and close things properly.

Finish the main form

1. Make a query, **qryStay** (not shown on the figure). It must join tblGuest and tblStay. It should include all of the fields, since they will all be needed somewhere on the Stay form. The easiest way is to drag the two stars from the tables to the grid. Don't worry that we get two guestID fields - one from each table. We actually need them in the later programs.
2. **Bind to qryStay.** You should have made a simple version of frmStay already (section 3.4). Now set its Record Source to qryStay in order to connect it to the query.
3. Connect the existing main-form controls to the query. For instance the *Stay No.* field should have its ControlSource set to stayID, in order to show this query field in the text box.
4. **Add fields.** Add a few more fields to the main Stay window, for instance the name, address and state. A convenient way to do it is to use the small Field List window that may have popped up when you connected the form to the query. Otherwise open the Field List window through the View menu or with the Field List icon next to the hammer icon. Drag a field to the form to create a text box and associated label. Access will automatically bind the text box to the query field.
5. For combo boxes, such as *pay method* and *state*, make the first column width zero to let the user see the mnemonic text rather than the number code.

Have a look at the stay window in user mode. It should show the first stay. You can use PageUp and PageDown to browse through all the stays. You can edit guest and stay data through the window since the dynaset behind the form allows it.

In the final system, the user should be allowed to edit data this way, but it makes no sense to allow him browsing through thousands of stays this way. We will later see how to control this and the creation of new stays (section 5.5.2).

Make the subform

We will now make the subform that shows the rooms for the stay.

6. Make a query, **qryStayRooms**. It is a join of three tables: tblRoomState, tblRoom, and tblRoomType (Figure 4.7).

The result of the raw query would be a list of all room states with data added about the room type and the prices. For a long stay, this would give an awfully long list of room states with one line per day. We want to shorten it to one line per room. In each line we need to show the first date the room is used and the number of nights it is used in total.

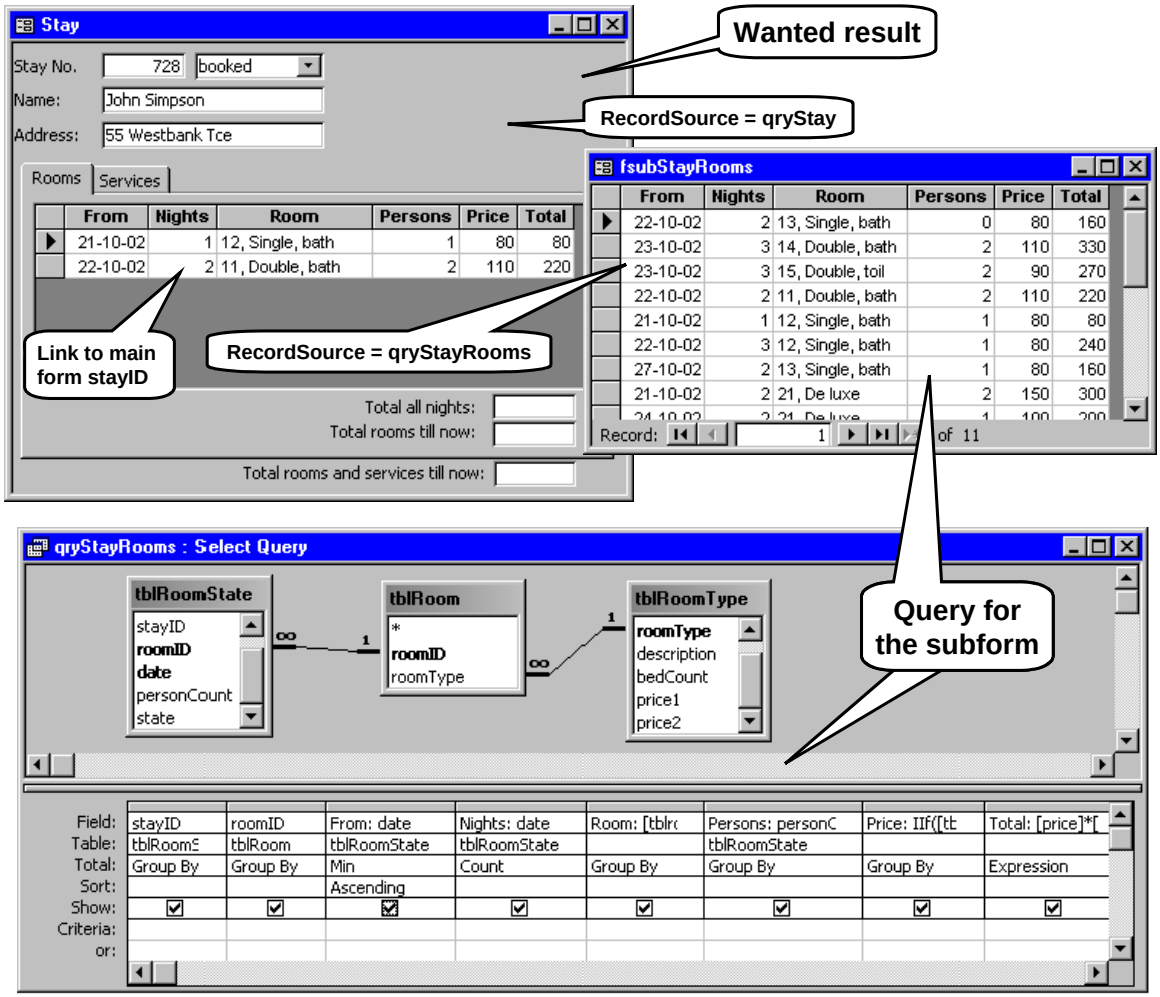
The query should give a result like the datasheet to the right in Figure 4.7. Here is a description of the fields in the query result. Try to define the query on your own.

stayID: Not shown in the continuous form, but needed to bind the list properly to the main form.
roomID: Not shown directly in the continuous form, but needed by the program to find out what the user has selected.
From: The first date this room was used by the stay.
Nights: The number of nights this room was used by the stay.
Room: A computed text string consisting of roomID, a comma, and the description from tblRoomType.
Persons: The number of persons actually staying in the room.
Price: The price - taking into account a possible discount for one person staying in a double room. More precisely, when the number of actual persons is one and the bedCount is more than one, price2 should be used. Otherwise price1.
Total: The total price for the room for all these nights.

Binding the subform to the main form

7. The query is to be shown in the subform on the Rooms tab. First make a continuous form based on qryStayRooms. Use the Form Wizard and omit stayID and roomID from the continuous form. Since the field names above are user-oriented, the continuous form should automatically look correct in datasheet view.
8. Connect the continuous form to the subform control on the Rooms tab. When you now look at the Stay window in user mode, you will see that the subform shows all rooms in the database.
9. To show only the rooms related to the stay, set these properties on the Data tab for the subform control:
   Link Child Fields = stayID
   Link Master Fields = stayID.

Fig 4.7 Bound main form



In this way you specify that you only want to see those child records (in qryStayRooms) where stayID matches stayID in the master form.

Now the stay window should look right. Try browsing through the stays and notice how the room lists vary from one stay to another.

4.7.1 Editing a GROUP BY query

What kind of actions can the user make on a complex query table such as qryStayRooms? Actually very little. It would make sense to edit the number of persons in the room, but since the room list is made by a Group By, this is impossible. There are two ways out:

a) **Dialog box**. Provide an edit-button that the user can click when he has selected a rooms line. The user will then see a dialog box where he can enter the number of persons and possible other editable data. When he closes the button, the VBA program updates the corresponding room state records.

b) **Store query result in a temporary table.** The alternative is to store the query result as a new, temporary table. (SQL can do this by means of an INSERT INTO query. See section 7.1) Then you show this table instead of qryRoomState. Since this is a table and not a query, the user can edit it. Next, the VBA program will have to transfer the changes to tblRoomState.

Programming issues such as these are the reason many software applications don't allow the user to edit directly in what he sees, but offer him a dialog box instead.

5. Access through Visual Basic

Highlights
- Addressing and changing Forms, Controls and records
- Responding to clicks, typing and other events.

When you develop Access-based user interfaces, you soon get to a point where the built-in features don't suffice. It happens for instance when buttons or menu items must do something a bit complex. Then you need to make program pieces that cooperate with the built-in features.

From this chapter and on, we assume that you have some understanding of programming in general. For instance that a program consists of statements that can be executed one by one; that it has variables and can change them during the execution; that it can call a procedure (also called subroutine, function, method or operation), which executes and then returns to the place where it was called; and that a procedure may have parameters.

We will first have a look at the built-in objects, for instance Forms and Controls. We will see how they can be addressed from the programming language, Visual Basic for Applications (VBA). Next we see what happens when the user types or clicks something, and how the VBA program can respond. We will also look at the tools available to the programmer.

From section 5.4 and on, the text is no longer a tutorial you have to follow to understand the rest. We don't say do this and do that, although we still explain experiments you can do to explore Access. These sections are for looking up some topic, for instance how you can open and close forms from VBA, how you can access records in the database, etc.

We only show pieces of VBA in this chapter. Chapter 6 is for reference purposes and covers VBA systematically. The VBA-language itself is rather simple and has the same components as many other programming languages, for instance Java and C++. The difficult part is how the program cooperates with the built-in objects and how these objects really work. This is our focus in this chapter.

5.1 The objects in Access

In the previous chapters we have encountered a lot of Access concepts: tables, queries, forms, and controls. How do they relate to each other? Figure 5.1A is a slightly simplified data model for all of these concepts.

When you work with Access, you really work with two different systems, the SQL engine and Access. The SQL engine supplied with Access is called the *Jet Engine*. It is used also by other systems than Access. You can for instance use your database through Excel and Word too. Access is primarily a tool for accessing data in databases through user windows (*forms*). Access can also work with other SQL engines than Jet, for instance Oracle. When doing this, you may lose some designer features but gain other qualities, for instance speed and reliability.

Databases

The database class on Figure 5.1A contains the currently open databases. Until now we have only looked at one database, but Access can handle and connect to several open databases at the same time.

Recordsets, tables and queries

A database contains a number of recordsets. Some recordsets are tables, others are queries (shown as subclasses). Each recordset has a description for each of its fields. The description includes the field name, the field type, format, etc.

A query has also an SQL-property - the text that describes how to compute the records in the query.

Tables and queries play much the same role. Both of them can for instance be the record source for a Form.

Records and fields

Each recordset contains a number of records, and each record contains a number of fields. Each field has only one attribute, the value stored in the field.

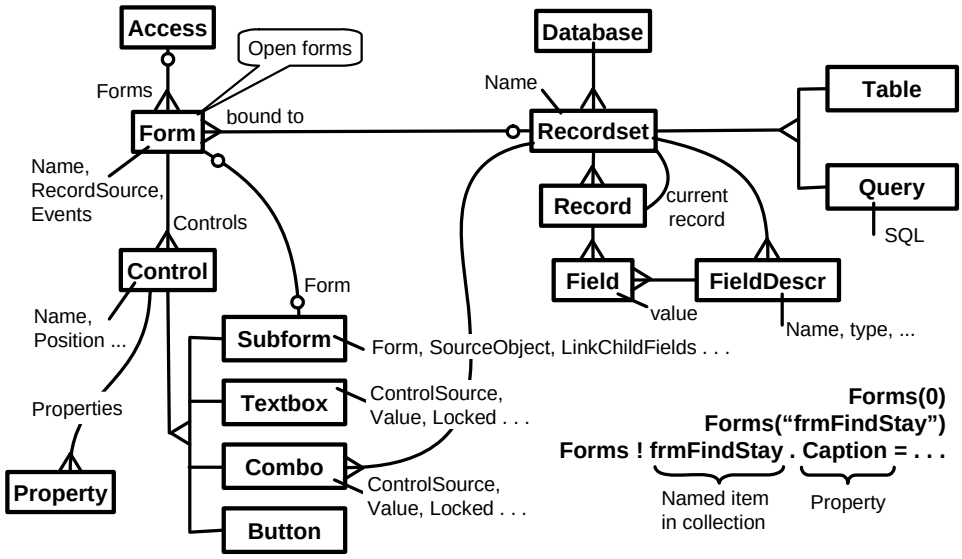
You can see the records in Datasheet view. You look either directly into a table or into records selected and computed by a query. The model shows that each recordset keeps track of a *current record* in the set.

The model in Figure 5.1A doesn't show that each field actually may belong to several recordsets, for instance fields that are seen through several queries at the same time. (It is one of the simplifications we have made.)

Access instances

The Access class on Figure 5.1A contains the currently open Access windows. When you open an mdb-file, you will see an Access window with title bar and a database window inside. The Access window corresponds to an object in the Access class. You may open another mdb-file, and will then get another Access window (another instance) and another object in the Access class.

Fig 5.1A The objects in Access



Forms

Each Access window contains a collection of *open Forms*. These are the forms you can address from VBA. (The closed forms are there too, of course, but you must open them before you can address them.) Notice that the 1:m connector from Access to Form has the name *Forms*. You can use this name to address the forms:

```
Forms(0)
```
Gives you the first open form.
```
Forms("frmFindStay")
```
Gives you the open form with the name frmFindStay.
```
Forms!frmFindStay
```
Another way of getting this open form.

Notice that you can address a form by its sequential number in the Forms collection or by its name. You can address it by its name in two ways: With the name in quotes inside a parenthesis, or with an exclamation mark and the name *without* quotes. (Sometimes it is useful to store the name of a form in a variable. To access the form you write the name of this variable inside the parenthesis.)

The exclamation mark is called the **bang**-operator. It works exactly as the name in the parenthesis, but is simpler to read and write. You can use the addressing methods to set a property of a form. The following statement will for instance change the user-oriented name (heading) of frmFindStay:

```
Forms!frmFindStay.Caption = "Guest Lookup"
```

Set and get properties - try it out

The addressing mechanisms are very important when you use VBA, but you have to practice.

1. Open the hotel database and open frmFindStay (the Find Guest window).
2. Open the *Immediate window* with Ctrl+G.
3. If you work in Access 2000 or 2003, this also opens the Visual Basic window where you can program. Make the Visual Basic window smaller than the full screen (use the *Restore* button next to the cross that closes the window).

Figure 5.1B shows the Immediate window. You can enter VBA statements here and have them executed immediately. Try to set and get some properties:

4. Enter this statement:
   `Forms!frmFindStay. Caption = "Guest Lookup"`
   You should see the form heading change.
   Note: Don't use spaces around ! or before .
5. Use the ? statement to print the caption in the Immediate window:
   `?Forms!frmFindStay. Caption`
6. Try printing the caption using the other addressing mechanisms: `Forms(0)` and `Forms("frmFindStay")`.
7. Close and reopen frmFindStay. The caption should now be back to *Find Guest*.

What happens here? The caption is just a dialogue variable that doesn't survive close and open. However, when you constructed the form, you specified the caption as *Find Guest*. This setting is the default value for the caption property when you open the form. Any

caption changes you make in the open form will be lost when you close the form.

(It is possible to change the *default* value for the Caption with VBA. Just open frmFindStay in design mode and then set Caption in the same way.)

8. Keep frmFindStay open and open also fsubStay-List. (This subform is also used inside frmFindStay). Change and print its caption.

The change has no effect on the subform inside frmFindStay. We are dealing with two open instances of fsubStayList. We cannot even access the subform instance with Forms( ). It is not a member of this collection.

Controls

Each form contains a collection of *Controls*, for instance text boxes and command buttons. In the same way as for forms, you can address a control on the form in these ways:

```
Forms(0). Controls(0)
```
The first control on the first open form.
```
Forms(0). Controls("txtName")
```
The txtName control on the first open form.
```
Forms!frmFindStay. Controls!txtName
```
The txtName control on frmFindStay.

In the last line, notice how the first bang selects the form by name, the other the control by name. Notice also the usual dot-operator. It selects a built-in property of the form.

To shorten these long expressions, VBA works with default collections. The *Controls* collection is the default collection on a form. This means that if you omit the word *Controls*, VBA will look in the controls-collection anyway. You could thus write this instead:

```
Forms(0) (0)
```
The first control on the first open form.
```
Forms(0) ("txtName")
```
The *txtName* control on the first open form.
```
Forms!frmFindStay!txtName
```
The *txtName* control on frmFindStay.

Controls come in many variants (sub-classes), for instance Subform, Text Box, Combo Box, Command button. They all have a name, a position, a height and a width, but otherwise they have very different properties. We will look at some of the controls in more detail later.

Try it out

9. If you have followed the earlier exercises closely, the name of the Last-name field should be txtName. Try setting the value of this control:
   ```
   Forms!frmFindStay!txtName. Value = "abc"
   ```
   You should see the text box changing on the form.
10. The default property of a field is the Value, so this statement should work the same way:
    ```
    Forms!frmFindStay!txtName = "def"
    ```
11. Try printing and setting some other properties of txtName, for instance *FontSize*.
12. Also try printing the *Name* property of txtName. Its name should be "txtName". You should not be able to change the name unless you set the form in design mode.

A form may be *bound* to a recordset, and the property *RecordSource* specifies how.

13. Use the ?-command to print the record source for the two open forms. FrmFindStay should have an empty record source while fsubStayList should be bound.

When the form is bound, it can show records from the recordset. In single-form view, it shows only one record at a time. In continuous-form and datasheet view it can show a sequence of records. In all cases, an arrow in the record selector area marks the current record. From VBA, you can address the current record:

14. Print the value of the stayID for the current record in fsubStayList:
    ```
    ? forms!fsubstaylist!stayid
    ```
    Check that it matches the open fsubStayList.
15. Use the mouse to select another record in the open fsubStayList. Now repeat the ? command. You should see the stayID for the new current record.

Collections

Collections have a *count* property, which gives you the number of items in the collection.

16. Try printing the number of items in the Forms collection:
    ```
    ?Forms. Count
    ```
    Is it correct?
17. The controls on a form are also a collection, called *Controls*. Try in the same way to print the number of controls on *frmFindStay* and *fsubStayList*.

Even the properties of a control or a form make up a collection, called *Properties* (see Figure 5.1A). You may also look into them with the general addressing mechanism.

Keep frmFindStay open for further exercises in the next section.

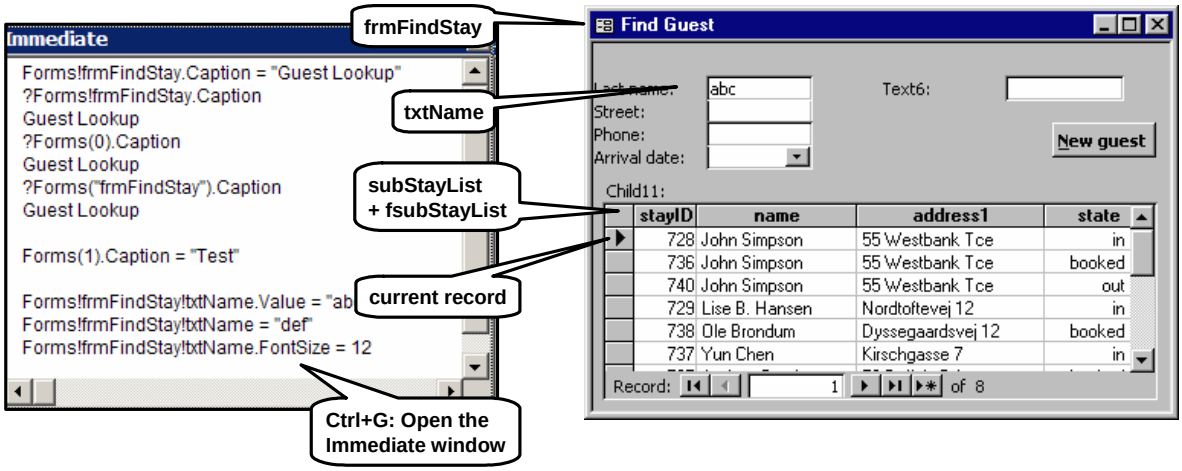
Subform

SourceObject property. A subform control has many properties, but a particularly interesting one is *SourceObject*. It is the name of the form to show. When we constructed frmFindStay, we set SourceObject to fsub-StayList. We can see it and change it from VBA:

18. First display the SourceObject property:
    ```
    ?Forms!frmFindStay!subStayList. SourceObject
    ```

The first part of this statement is the reference to sub-StayList on frmFindStay. Now comes a dot saying that

Fig 5.1B Addressing the objects



we look for the SourceObject property of subStayList. The result is the name of the fsub. (Notice that our prefix rules help us distinguish between the subform control and the fsub connected to this control).

19. Try setting SourceObject to nothing (an empty text in quotes):
    Forms!frmFindStay!subStayList. SourceObject = ""
    The subform in the user window becomes blank. Try setting it back again (it is a text, so remember the quotes.)

As soon as we set SourceObject, Access closes the previous subform and opens the new one. The SourceObject property is a dialogue variable, so a change will not survive close and open.

Form property. Above we looked at the SourceObject property, which is a text - the name of the fsub. But a subform has another interesting property, *Form*. It is a reference to the *open* form that is shown in the subform area. You cannot see this property in the property box because it is not a text, but a pointer that only exists in user mode. However, you can use the Form property in VBA.

20. Try this command in the Immediate window:
    ?Forms!frmFindStay!subStayList. Form!name

It should print the name of the guest selected in the subform area. The first part is the reference to subStayList on frmFindStay. Now comes a dot saying that we look for the Form property of subStayList. The result is a reference to the open form that is shown in the subform area.

Next comes a bang and a reference to the *name* field, i.e. the address of a guest. Now which guest? The subform shows many records - which of the addresses will we get? The address in the currently selected record.

Was this what you got? Try selecting another record on the list and repeat the command.

Bang versus dot

21. Try using a dot instead of a bang before *name*:
    ?Forms!frmFindStay!subStayList. Form. name

It doesn't print the name of the guest, but the name of the subform. Because of the dot, the statement asks for the name property of the form.

When you use the bang, you ask for a control on the fsub - or a field in the bound record behind the fsub.

Abbreviations. Above we have used the full address expressions, but Access allows various abbreviations. For instance you may refer to the *guest* control of the subform in any of these ways:

Dot instead of bang

Forms!frmFindStay!subStayList. Form. address1

This works only because there is no *address1* property in a form.

Form omitted

Forms!frmFindStay!subStayList!guest

However, you cannot omit Form and use a dot at the same time. Access will believe that you ask for a property of the sub-control.

Warning. The bang mechanism works only in Visual Basic. It doesn't work in SQL. For instance you have to write

... JOIN ON tblGuest.guestID = tblStay.guestID

If you write tblGuest!guestID, the system gives an error message.

5.2 Event procedures (for text box)

Above, we have executed VBA statements through the Immediate window, but when the user uses the system, we need another way to activate VBA. We want the system to respond to user actions, for instance the user clicking a button, typing something in a text box, etc. These user actions are called **events**. You can write a VBA procedure for each kind of event.

AfterUpdate event

We will now write an event procedure that makes txtName respond when the user has entered a search criterion.

1. Open frmFindStay, select the txtName search criterion, and view the property box. (In Access 97 you can only do this in Design mode.)
2. Look at the Event tab. It has a list of the events that the control can respond to (Figure 5.2A). The event we are interested in is AfterUpdate. It happens when the user has finished entering the text in the control.
3. **Create event procedure**. Choose AfterUpdate, the three dots, and Code Builder. This creates the event procedure for AfterUpdate.

You are now in the Visual Basic window. You may make the window smaller so that you can see frmFindStay too. Also make the Access window smaller so that you can see the two windows side by side.

Inside the Visual Basic window you see the event procedure for AfterUpdate (Figure 5.2A, right). The first line reads

```
Private Sub txtName_AfterUpdate
```

It shows that it is the AfterUpdate subroutine (procedure) for the txtName control.

4. The body of the procedure is initially empty. Now enter this statement:

```
Me!subStayList . Requery
```

Me. The word *Me* means the open form where this control is placed - in this case the same as Forms!frmFindStay. We ask Access to find the subStayList control. Finally we ask Access to call the procedure *Requery* in this control. The result is that subStayList recomputes the query behind the subform. Since the query gets its where-condition from txtName, the stay list should change.

Omitting Me. In most cases you can omit *Me*. You may for instance write

```
subStayList . Requery
```

The exception is when some built-in function or property has the same name as the control. In the following sections we will sometimes write *Me* for emphasis, sometimes omit it.

When is the AfterUpdate procedure called? Whenever the user shows that he has finished entering the text, for instance when he clicks on another control in the same form, tabs to another field, or hits Enter. Try it:

5. Enter a criterion in the name field, for instance "a". Click Enter. The stay list should now shrink according to the new criterion. Enter another criterion and use Tab. The system should respond.

6. Change the criterion and then - without using Enter, etc. - click in another window, for instance the property box. The stay list should not change because txtName still has the focus and Access assumes that the user hasn't finished the field. Now click another field on FindStay. The stay list should respond.

5.2.1 More text box properties

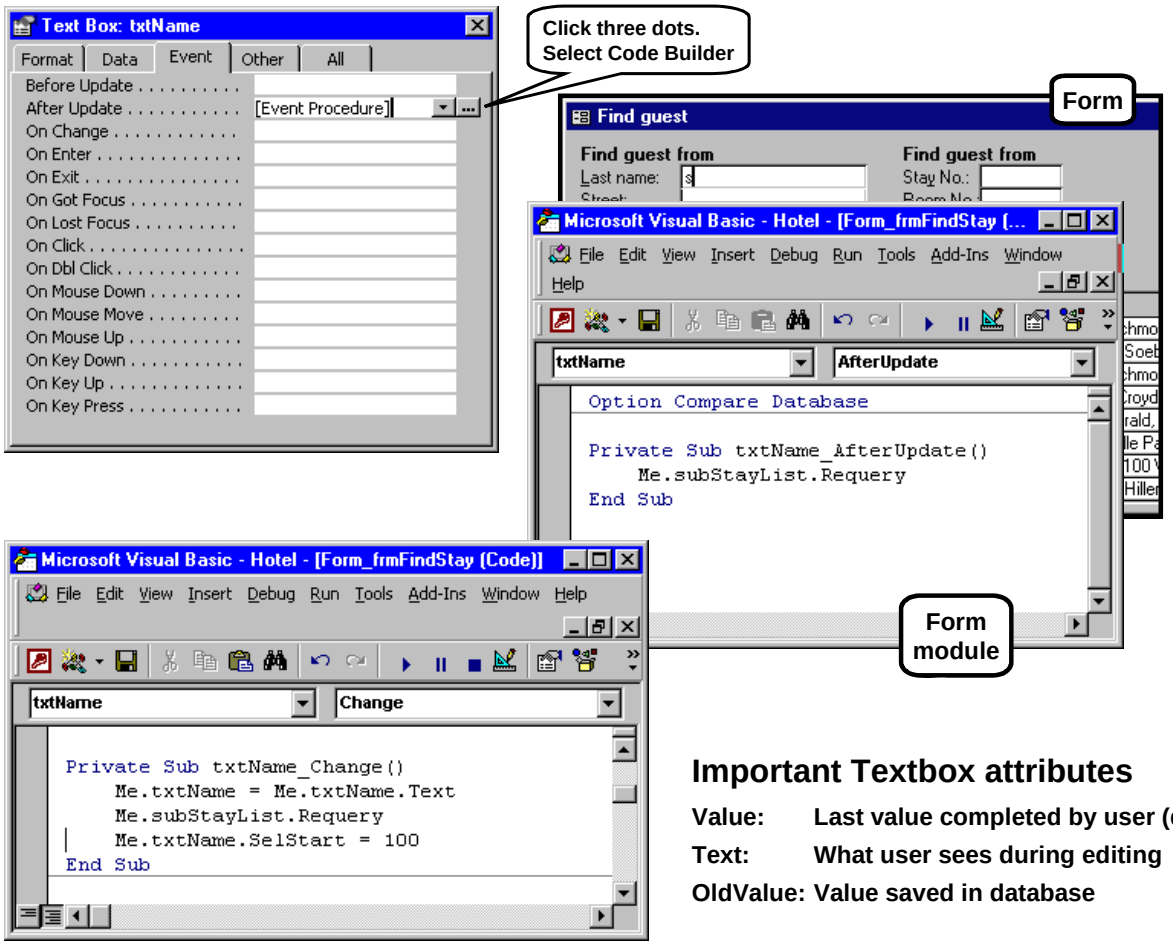
We will now make txtName respond each time the user hits a key. To do this, we need to act on another event, OnChange. This event happens whenever the user types something in the text box or deletes a character. When Access calls this event procedure, the situation is a bit complex because several properties of the text box are involved:

Value: This property is the value before the user started editing the text box. Note that Value is the default property for a text box, so if you don't specify another property in your address expression, VBA will use Value. When the Value property is empty, it has the value Null, similar to a database field

Text: The value the user enters, but hasn't finished yet. This is the text the user sees on the screen, but it is not yet stored in Value. You can only access this property when the text box has the focus. At other times, the Text variable doesn't exist at all. When the Text property is empty, it is a zero-length text with the value "", similar to a VBA string variable.

OldValue: The value before the user started editing the text. For bound controls (bound to a field in the database), this is the same as the value in the database. When the entire *record* is complete, Access transfers Value to OldValue. OldValue is useful when the editing must be undone for some reason. For unbound controls, OldValue and Value are always the same.

Fig 5.2A An event procedure



Important Textbox attributes

Value: **Last value completed by user (default attribute**

Text: **What user sees during editing**

OldValue: **Value saved in database**

Change event

We will now make txtName respond to each keystroke. Proceed in this way:

7. On the event tab for txtName, choose OnChange and the three dots. You now see the event procedure for the Change-event.

8. Write these statements in the procedure body (Figure 5.2A):

```
Me.txtName = Me.txtName.Text
Me.subStayList.Requery
Me.txtName.SelStart = 100
```

9. Try it in user mode. As soon as you enter something in the text box, the stay list changes accordingly.

The first procedure line copies Text to the Value of the control. As a result, Value becomes what the user has entered until now. Then the program recomputes the stay list. During this it gets the search criterion from the value of txtName. This is why we moved Text to Value.

The last procedure line is more of a mystery. If you try using the system without this line, the entire text box becomes selected whenever you enter something. This seems to be an undocumented side-effect of setting the Value. To compensate for it, we let the program define what is selected. We define the selection start point (SelStart) to be character 100 of the text. Since the text is much shorter, Access interprets it as being the end of the text.

This solution works, but not very well. If you for instance try to change a few characters in the middle of the search criterion, the system annoyingly moves the cursor to the end of the text. We could repair it by storing the position before we set the Value, and set it back afterwards. This is messy, and in the next section we will show a more professional solution.

5.2.2 Computed SQL and live search

We will now the professional solution. We don't change Value at all, but compute the SQL-statement to be used for the stay list. Let us first assume that the user has entered the search criterion *john*. The event procedure could then compute the list of stays in this way:

```
Me.subStayList.Form . RecordSource = _
   "select * from qryStayList where name like ( '*john*' );"
```

This is one long Visual Basic statement split into two lines by means of the space and underscore at the end of the first line.

Now, what does the statement do? The first line refers to the open form in subStayList. The form has a record source property, which defines the records to display. We had bound the form to *qryStayList*, so the record source was "qryStayList".

Now the procedure changes the record source to the SQL statement in the second line. This statement takes all records in *qryStayList* where the name contains *john*. And it selects all attributes from the records. This is what we need.

Notice that the SQL statement is a text surrounded by double quotes. Inside the statement we have another text string '*john*'. This text string is surrounded by single quotes to distinguish it from the large text. We haven't cared to write SELECT etc. with caps. The SQL engine accepts the statement anyway.

The only problem is that *john* should be the text that the user has entered. So the event procedure has to compose the SQL statement from three parts, like this:

```
"select * from qryStayList where name like ('*" & _
     Me.txtName.Text & "*');"
```

Note: Some developers don't use single quotes for the text inside the text. They use double quotes for the inner text. So instead of

```
" . . . like ('*" &
```

they would write

```
" . . . like (""*" &
```

The rule is that inside a text string, the characters "" mean a single ".

Try it out

You may try the solution right away, but take care. The next time you open frmFindStay and try to search, Access remembers the SQL statement your program has set. It uses this statement as the initial query for the future. (I would call this an error.)

To avoid problems do as follows:

1. In qryStayList, remove the user criterion that we defined earlier (like . . .). We don't need it anymore.

2. In fsubStayList, the record source is *qryStayList*. Change it to *select * from qryStayList;*

3. Change the event procedure and try it out (see Figure 5.2B).

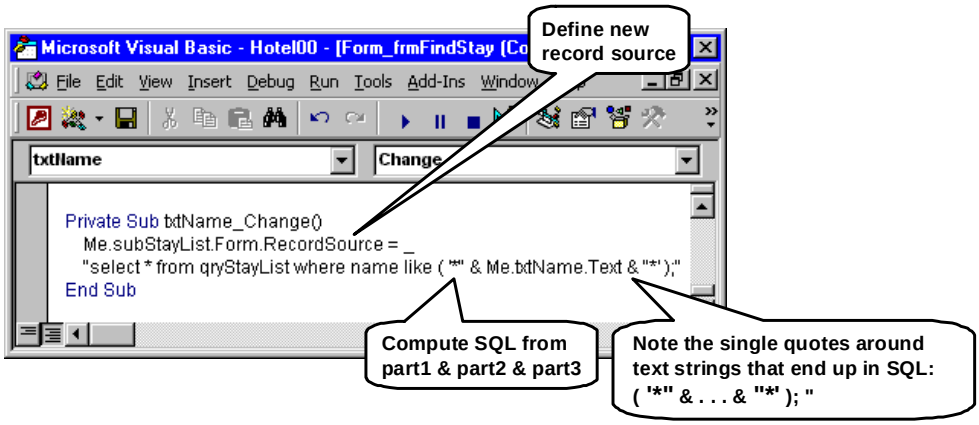
You should now have a **live search** similar to the one used in the real system.

(If Access still remembers the previous search criterion, open frmFindStay in design mode and set once more the Source Object of subStayList to *fsubStayList.*)

Computed SQL may seem very cumbersome. Yes - it is, no doubt! However, when we need complex user criteria, for instance a combination of name and/or phone and/or date, the easiest way is to let the program compute the SQL statement. In general, computed SQL is the professional way to make complex systems.

In the next section we will use computed SQL to deal with combinations of search criteria, some of them live and some lazy.

Fig 5.2B Computed SQL and live search



To make it work properly:

Remove the old user criterion from qryStayList.

Set record source of fsubStayList to: select * from qryStayList;

Change the event procedure as above. Try it out.

5.2.3 Composite search criteria

Figure 5.2C shows an example where the user can use one or more criteria in any combination - *composite search criteria*. The figure also shows the programmer names of the various controls. The name criterion is called txtName, the booked criterion chkBooked, etc.

The user may fill in part of the guest's name, and/or parts of his address, etc. The user may also restrict the search to booked stays, and/or checked-in stays, etc.

To achieve this, the program must generate a suitable SQL statement for each combination of criteria. The pattern in the final SQL should be this, with optional parts shown in square brackets:

```
SELECT * FROM ... WHERE TRUE
[ AND name Like( "*john*") ]
[ AND address1 Like( "*buf*") ]
[ AND phone Like( "*3700*") ]
[ AND arrival = #21-10-02# ]
AND (FALSE [ OR state=1 ] [OR state=2 ]
        [ OR state>2 OR ISNULL(state) ] );
```

Notice that we start with a useless TRUE before the optional AND's, and a useless FALSE before the optional OR's. This is a trick that makes the program far simpler. Each piece of the program doesn't have to care whether it adds the first element to an AND/OR list, or a later element.

The figure shows the program that generates the SQL statement. It gradually builds up the SQL-statement in the string variable s. First it puts *SELECT * . . . TRUE* into s. Next it tests whether the name criteria holds something. If so, it appends *AND name Like . . .* to s. And so on. Finally, it stores the SQL statement as the records source of the stay list.

In the places where the pattern has *john*, *buf*, etc., the program inserts the value from the appropriate text box.

Notice how we have written SQL parts in caps. Professionals do so to make the SQL pattern stand out more clearly. We explain the details below.

Shared procedure

When should this tricky piece of program be executed? If the screen used an ordinary *Search* button, the program should be the event procedure for the button. However, we want to make a live search. In this case the program must be executed whenever one of the criteria changes value, i.e. for a whole bunch of event procedures.

The solution is to make the program piece a separate procedure. It starts with *Private Sub* and ends with *End Sub* (see more on procedures in section 6.2). The appropriate event procedures call this procedure as shown for *txtName_Change*, *txtArrival_AfterUpdate* and *chkBooked_AfterUpdate*.

Live search criteria

In order to make a live search, the program must respond whenever the user has typed something. The partially finished criteria are used for the search, and the stay list shrinks gradually. While the user types, the result is not in the box's *Value*, but in the temporary variable *Text*. The other text box values are in *Value*.

It would complicate the search procedure to deal with this too. The solution is to let the event procedures store the current criteria in the variables *copyName, copyStreet and copyPhone*. The event procedure *txtName_Change* shows how. The search procedure uses these copies to generate the SQL statement.

Note how we have declared the copy variables in the first line of the form (see more in section 6.2).

Lazy search criteria

Some of the search criteria should not be live. The check boxes, for instance, are either true or false. It is sufficient to let their AfterUpdate procedure call the search procedure and let it use the checkbox values. No copy is needed.

The arrival date might be live, but while the user types the date, the intermediate value has little meaning. The stay list would just flicker and confuse the user. For this reason, the arrival date reacts only at AfterUpdate.

Date comparison

In Figure 5.2C, the arrival date is a text box control with the format property *short date*. Access will check that the text corresponds to a valid date. It will show the date according to the format property, but it will store it as a Double number - the number of days since the end of 1899 (see page 11).

When generating the SQL statement, the program uses the CDbl function (Convert to Double) to show the date as a number:

```
" AND arrival = " & CDbl(txtArrival)
```

This would generate an SQL fragment like this:

```
AND arrival = 37550
```

Later, the SQL engine will compare the dates in their number form, and everything works fine.

Why all this fuzz? Couldn't we simply write:

```
" AND arrival = " & txtArrival
```

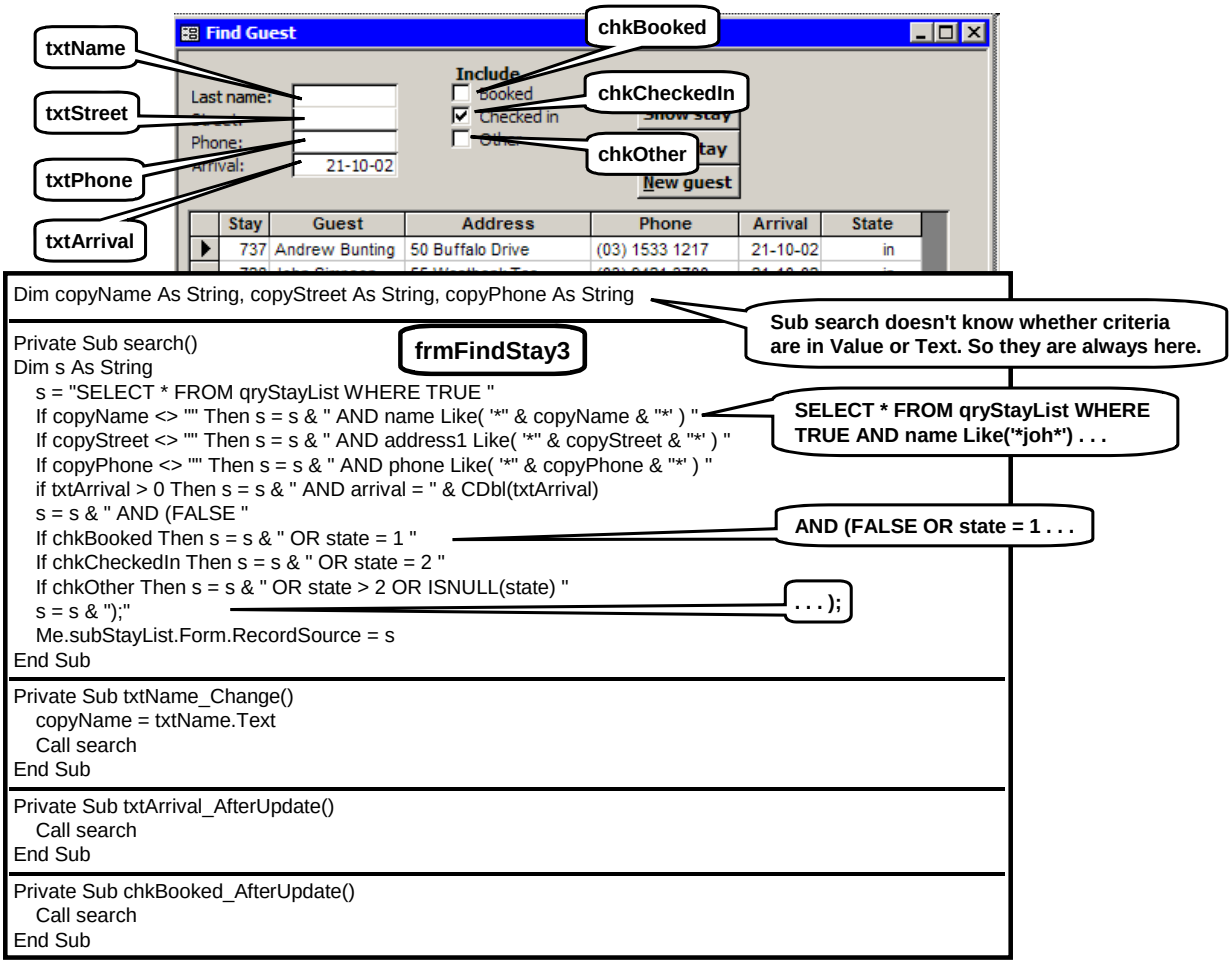
When Access appends *txtArrival*, it converts it to text format by means of the CStr function. This function produces a date in the regional date format set up in MS Windows. With a European date format, we would get this SQL fragment:

```
AND arrival = 21-10-07
```

SQL would reject it since dates have to be enclosed by # #. So what about this:

```
" AND arrival = #" & txtArrival & "#"
```

Fig 5.2C Composite criteria



Now we would get this fragment:

```
" AND arrival = #21-10-07#
```

Looks all right, but unfortunately, SQL expects dates in the US format. In this case it would reject 21 as a non-existing month, in other cases produce a wrong result.

Fortunately, the Access database engine can treat dates as double numbers, but other databases may not. In such cases you would have to generate the dates with an explicit US-format, for instance in this way:

```
" AND arrival = #" & Format(txtArrival, "mm/dd/yyyy") & "#"
```

Empty texts

An empty field in the database has the value Null for all types of fields. The same applies for the Value in an empty text box. To test for an empty field , we would ask for IsNull(f). In a VBA string variable, an empty text is a zero-length text with the value "". Since *copyName* is a string variable, we ask whether the text is different from "".

Null values

Notice how the program deals with *Other* stays, i.e. stays that are neither booked nor checked-in. These stays include those with state > 2 and those without a state (a NULL state). Stays with a NULL state are not real stays, but records generated by the outer join for guests that don't have a stay.

Initialize the form

A final touch is to make the Find Guest screen initialize itself properly:

- Let the three checkboxes have a mark initially. Otherwise, the stay list will be empty initially and the user may panic. To do this, set the initial value of the boxes through their property sheet.

- Let the Form's Load event procedure call the search procedure in order to make the stay list match the initial criteria:

```
Private Sub Form_Load()
   Call search
End Sub
```

Try it out

You should try to make the solution work in practice. It is a challenge, but fun. Make a copy of frmFindStay and use it for the experiment.

5.2.4 Event sequence for text box

We have now looked at two events for the text box, *Change* and *AfterUpdate*. However, there are many more, as you can tell from the Event tab in the property box. (Select the event procedure and click F1 to get an explanation of the event.) Figure 5.2D shows typical event sequences for a text box. We will explain what happens.

User clicks in the text box. First, the text box's **Form object** may get a *Current* event. This happens if the text box is in a record that wasn't selected before.

Next, the text box receives two events, *Enter* and *GotFocus*. *Enter* signals that now the control is active. Before calling the Enter event-procedure, Access creates the Text property and sets it to the current Value of the text box. *GotFocus* signals that typing will now go to this control.

Then the textbox receives this series of events: MouseDown, MouseUp, Click and maybe also a DblClick.

User types an Ascii character. The text box receives four events: KeyDown, KeyPress, Change, KeyUp (plus mouse events if the mouse is used). *KeyDown* occurs when the user depresses any key. The event procedure has parameters that give details of the physical key and other keys depressed at the same time, for instance Shift and Ctrl. *KeyPress* occurs when the key has generated an Ascii character, for instance a letter, digit, space, tab or backspace. All of these correspond to characters in the Ascii alphabet.

The *Change* event occurs when the visible content of the control has changed, for instance that a character was added to the text or a character was deleted. Before calling this event procedure, Access updates the Text property so that it contains what the user sees.

The *KeyUp* event occurs when the user releases the key. If the user keeps the key down to generate for instance a whole sequence of x's, each x generates KeyDown, KeyPress and Change. The KeyUp occurs only when the user releases the key.

User types a non-Ascii character, for instance Arrow left or F6. This generates only the KeyDown-event (and the KeyUp). Nothing happens to Text or Value, and no Change event is generated.

User types Delete. Delete is not an Ascii character, so no KeyPress-event occurs. However, one or more characters may be deleted, and then a Change event occurs. KeyDown and KeyUp occur too.

User tabs to the next field. Access generates several events. *KeyDown* occurs since the user pressed the Tab key. *BeforeUpdate* shows that the user has finished the field, but it is not yet accepted. Before calling the event procedure, Access has copied Text to Value. *BeforeUpdate* may check the value, and in case something is wrong reject the update. Rejecting the update means that focus remains on the text box and no AfterUpdate event is generated yet. The user may edit the field or click Esc to set the value back to its old value.

If it is an unbound control (not connected to a database record), Access will also copy Text to OldValue before calling the procedure. This is quite illogical because OldValue is intended for letting the program restore an erroneous field. This is not possible for unbound controls.

Next comes the *AfterUpdate*-event. The value has been checked and the procedure may act on it. In the example above we used this opportunity to recompute the stay list. If it is a bound control, the value will not yet be stored in the database, nor will OldValue be changed. Storing the value in the database doesn't happen until the user moves to another record or explicitly saves the record with Shift+Enter (or Records -> Save Record).

Next comes the *Exit* event. It signals that the field is not active anymore. Finally, *LostFocus* occurs and signals that typing will go to another control. When both events have occurred, the Text property disappears.

What about *KeyUp*? It happens in the next field, which accordingly receives the KeyUp event. Before that, the next field receives Enter and GotFocus - in the same way as if the user clicked in the next field.

What about *errors* in what the user has typed, for instance an incorrect date? Access shows an error message to the user instead of calling *BeforeUpdate*, and the cursor remains in the field. Can't the program interfere before this error message? Yes it can. The form receives a *Form_Error* event, which may take action and cancel the error message that Access was about to show (see section 5.5.10).

User clicks in next field. Access does almost the same as when the user tabs to the next field. The only difference is that there is no KeyDown event.

User moves to next record. First the text box in focus will receive the same events as if the user moved to another control in the same record.

Next, the **Form object** receives three events. *BeforeUpdate* signals that the record is about to be saved and that the program may check that values are correct and consistent with each other. The Be-

Fig 5.2D Event sequence, textbox

User action	Events	Property changes before call
Click in textbox	(Form: Current) Enter. GotFocus. MouseDown, MouseUp Click, (DblClick).	Text = Value, OldValue = Value
Type Ascii character (letter,digit, tab . . .)	KeyDown, KeyPress. Change. KeyUp.	Text = Modified text
Type ArrowLeft or F6	KeyDown, KeyUp.	
Type Delete	KeyDown. Change. KeyUp.	Text = Modified text
Tab to next field in same record	KeyDown. BeforeUpdate. (or Form: Error) AfterUpdate. Exit, LostFocus. Next field gets: Enter, GotFocus, KeyUp.	Value = Text. Unbound controls also have OldValue = Text (Program may cancel update) Text property disappears
Click in next field	As tab to next field, but no KeyDown event.	
Move to next record	Form: BeforeUpdate. Form: AfterUpdate. Form: Current.	(Program may cancel update) All bound controls in the form: Database = OldValue = Value
Click outside form Click in form again	LostFocus. GotFocus.	(no action) (no action)
Click outside Access	(no event)	

foreUpdate procedure may cancel further event processing so that the database is not updated.

If everything is okay, Access stores all the bound control values in the database and in OldValue, then calls the *AfterUpdate* procedure. Finally, Access calls the *Current* procedure to signal that a new record is selected.

User clicks outside the form, but inside the same Access window. Access generates a LostFocus event to show that typing will go to another object in Access. In this case the Text property does not disappear.

User clicks in form again. The active text box receives a GotFocus event. The Text property has survived.

User clicks outside Access. The text box does not receive any events (except for MouseMove events). All properties survive. When the user clicks on the Form again - even on the title bar - focus will be back at the text box, but it receives no events at that point.

The OldValue property allows the program to undo changes to bound controls until the moment when the form receives an AfterUpdate event. To undo a change, the program sets Value=OldValue.

Key Preview. In some cases we want a key or key combination to perform the same thing no matter which control is selected. One example is **function keys**. For instance we may want F2 to mean *Reset Criteria* no matter where in the form the cursor is. This can be handled by letting the Form look at the key before any control gets to know about it. See section 5.5.9 for this *Key Preview* function.

5.3 Visual Basic tools

Before we go on writing more complex event procedures, we will show some tools that Visual Basic offers.

1. Open a form in Design mode. Now click the Code icon on the toolbar (close to the Toolbox icon).

You are now in the Visual Basic window. (You might also get there by choosing an event procedure in the property box, as we did above.)

Docking and undocking, Access 2000 and 2003
The Visual Basic window may contain many frames. In Access 2000 and 2003, they may be *docked* inside the window or *undocked*, i.e. floating as separate small windows. By accident you may dock or undock them, and it may be very frustrating trying to get them back where you want. So better learn how to deal with it:

2. Make sure the Visual Basic window occupies only part of the screen.
3. Make sure the *Docking* settings are all on: Go to Tools -> Options -> Docking and make sure that all the checkboxes are set. Close the option box.
4. Click the Project Explorer icon. The window may now look like Figure 5.3A with a list of forms at the left (the Explorer frame) and some program code at the right. However, it may also show only the Explorer frame, or show the Explorer frame as a small, floating window.
5. Drag the Explorer frame to somewhere outside the Visual Basic window. The Explorer frame is now undocked.
6. Drag the Explorer frame to somewhere inside the Visual Basic window. The frame will dock somewhere along a side of the window. Where it docks does *not* depend on where the frame is when you release the mouse button. It depends on where the mouse pointer is when you release the button.
7. Try dragging the frame around in the Visual Basic window. Notice that when the mouse pointer is close to one of the sides, the frame changes to a thin line showing the new shape of the frame. Release the button to dock the frame accordingly.
8. Try double-clicking the title bar of the frame. The frame toggles between docked and undocked. Leave it docked as on Figure 5.3A.

Project Explorer

To the left you see a list of all forms in the database. In object-oriented terminology, each form is a class. The class has a code module with event procedures for the controls on the form. When you double-click a form on the list, you will see the code module to the right in the Visual Basic window.

The database may also contain code modules (classes) that are not forms. They are shown at the bottom of the Project Explorer list.

All code modules may contain procedures and declare variables. VBA offers three kinds of modules:

Form module. A form module has event procedures for all controls on the form, and it may have ordinary procedures too. It may declare variables. When you open a form, you get an object based on the form class. The object is a visible form and it has a set of variables corresponding to the declarations.

You cannot open several versions of the form just by clicking multiple times in the database window, but you may do it from Visual Basic. Then you get more form objects, appearing as other open copies of the form. Each copy has its own variables, but the same code. When the code uses addresses such as *Me.txtName*, it refers to the controls and variables of this particular copy of the form.

When you create a form, you don't get its form module until you create its first event procedure.

Class module. A class module corresponds to a class in other object-oriented languages. It has procedures and declares variables, and you can create multiple objects based on the class, each with their own variables. The only difference between form modules and class modules is that the latter are not visible to the user and have no controls.

See section 5.7 on how to create modules.

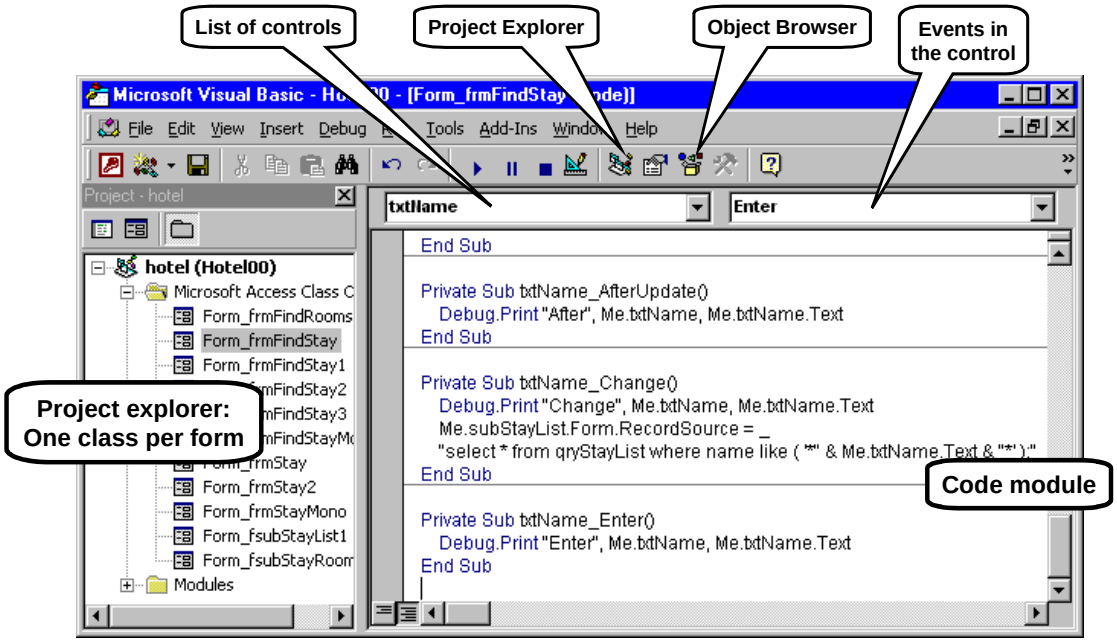
Module (simple). A simple module is similar to a class but there is only one object based on the module. The system creates this object automatically. The first versions of Visual Basic had no class modules, only simple modules.

Code window

To the right in the Visual Basic window, you see the code window with the Visual Basic program. Figure 5.3A shows the code module for frmFindStay. You see three event procedures for the txtName control. You can scroll to other event procedures and controls, or you can select them by means of the two combo boxes at the top.

Creating an event procedure. Initially, the event procedures are not in the code, but if you select one of them by means of the combo boxes, Visual Basic creates it. The same thing happens if you select an event procedure through the control's property window. However, Access doesn't always coordinate these two things. You may experience that you have created an event procedure in the code window, but it doesn't appear in the property box. As a result, Access never calls the event procedure. This may for instance happen

Fig 5.3A VBA window and debugger



if you have created the event procedure while looking at the code in debug mode (see page 82).

The cure may be to select the event procedure from the property box and *compile* the module (Debug -> Compile), then close and open the form.

9. Create the missing event procedures for txtName as shown on the Figure. Use the combo boxes to create them. If you have followed the previous steps closely, only the Enter procedure should be missing.
10. Type the debug statements shown in the procedures, for instance:
    Debug.Print "After", Me.txtName, Me.txtName.Text
    When executed, this statement prints something in the Immediate window. We will explain more on the Debug statement below.

Note that VBA automatically capitalizes the words that it recognizes. If it doesn't, it may be because you have spelled the word incorrectly, but it may also be because the word comes after the bang operator (!). VBA cannot recognize things after the bang at edit time, but it will check all the words at execution time. At that time it will give you an error message if it cannot recognize the word.

Deleting an event procedure. If you want to delete an event procedure, simply delete all the lines of the procedure. Don't try to delete it on the Event tab.

Closing the VBA window. You can close the VBA window at any time. It only hides the window. Saving the code doesn't happen until you close the form it belongs to. Sometimes you may want to save the code explicitly. Use File -> Save (or Ctrl+S).

Debug command and event logging

The statement *Debug.Print* prints its parameters in the Immediate window (also called the debug window). As an example, when the AfterUpdate event occurs, the event procedure will print the text "After" followed by the current *Value* and *Text*. The event procedure for Change behaves similarly, and as a result, the Immediate window will show a log of what happened. Try it:

11. Open the Immediate window with Ctrl+G. Adjust the sizes so that you can see the Immediate window as well as frmFindStay.
12. Type something in txtName. The Immediate window should log what happens.

This is the hard way to find out exactly which events occur and what the situation is at these points. During such experiments, you may want to temporarily disable some statements. For instance, we might want to disable one of the debug statements to avoid too many lines in the Immediate window.

13. **Comment-away** statements that you don't want for the moment. Set an apostrophe (a "ping") at the beginning of the line. When you move the cursor, the line turns green to show that this is a comment for humans only.

Breakpoints and debug

We are able to stop the program in the middle of an event procedure. To do this, we set a breakpoint at the code line where we want the program to stop. Figure 5.3A shows a breakpoint in the last event procedure - marked with a big dot in the left margin. Try it:

14. Click at the left margin of the Debug.Print line. The dot should appear and the line be high-lighted as on the figure. You have set a breakpoint. (Clicking again will remove the breakpoint. Don't worry about the line *Msgbox*. We will add it later.)

15. Click in some other field on FindStay, then click in txtName. This generates an Enter event in txtName. As a result, the program stops just before executing Debug.Print.

16. **Current value**. Try moving the cursor to an expression, for instance the Debug parameter *Me.txtName*. After a moment, VBA shows the current value of this expression. You can see the value of expressions in the procedure where the execution stopped, but not values in other procedures because they are not active at present.

17. You can use the Immediate window to try out various statements. The statements are executed as if they were written where the breakpoint is. Try for instance

    `Me.txtName.Text = "abc"` (should change the text in the form)

    `? Me.subStayList.Form!name` (should print the name of the first guest in the list)

In section 5.1 we used the Immediate window with statements such as

```
Forms!frmFindStay!txtName = . . .
```

Now you can use *Me*. The reason is that now the Immediate window runs in the context of the event procedure. It can address the form object in the same way as the program.

Correcting bugs at breakpoints. Using the Immediate window in connection with a breakpoint is an important way to find out what the program does. If you find an error in the program, you may usually correct it while at the breakpoint. However, sometimes VBA cannot do the correction, for instance if you delete an event procedure. It asks whether you want to "reset the project". This question sounds threatening, but simply means "stop the program execution and restart it from the beginning". Nothing to worry about - nothing is lost.

Continue after breakpoint. When you have made your experiments at the breakpoint, you can resume ordinary program execution. There are several ways to resume execution (Figure 5.3A):

- Run (F5). The program continues in the normal fashion.

- Step Into (F8). The program executes the next statement, then stops again in breakpoint mode. With repeated use of F8, you can execute the program step by step. If the program calls another procedure, you will step into it statement by statement.

- Step Over (Shift+F8). As F8, but if the program calls another procedure, it executes all of the procedure without stopping, then stops at the return from the procedure.

Stop an endless loop. If the program gets into an endless loop, you can stop it with Ctrl+Break.

Pop-up help

You have probably noticed that as you type a statement, Access often shows a list of what you can type at this point. You may bring up several kinds of lists:

- Ctrl+J brings up the **property** list. It shows you the possible properties, procedures, and controls at this point of typing. You may choose one with the Tab-key.

- Ctrl+Shift+J brings up the **Constant** list. It shows you the possible named constants at this point of typing. (Ctrl+J may be used too, but it brings up the full list of named constants.)

- Ctrl+I brings up the **Quick Info** list. It shows the data type you are dealing with, or the list of parameters to the procedure you call, or the value of a named constant.

Try the pop-up help

18. Start entering this statement in the Change procedure (Figure 5.3B):

    ```
    MsgBox "test", vbYesNo
    ```

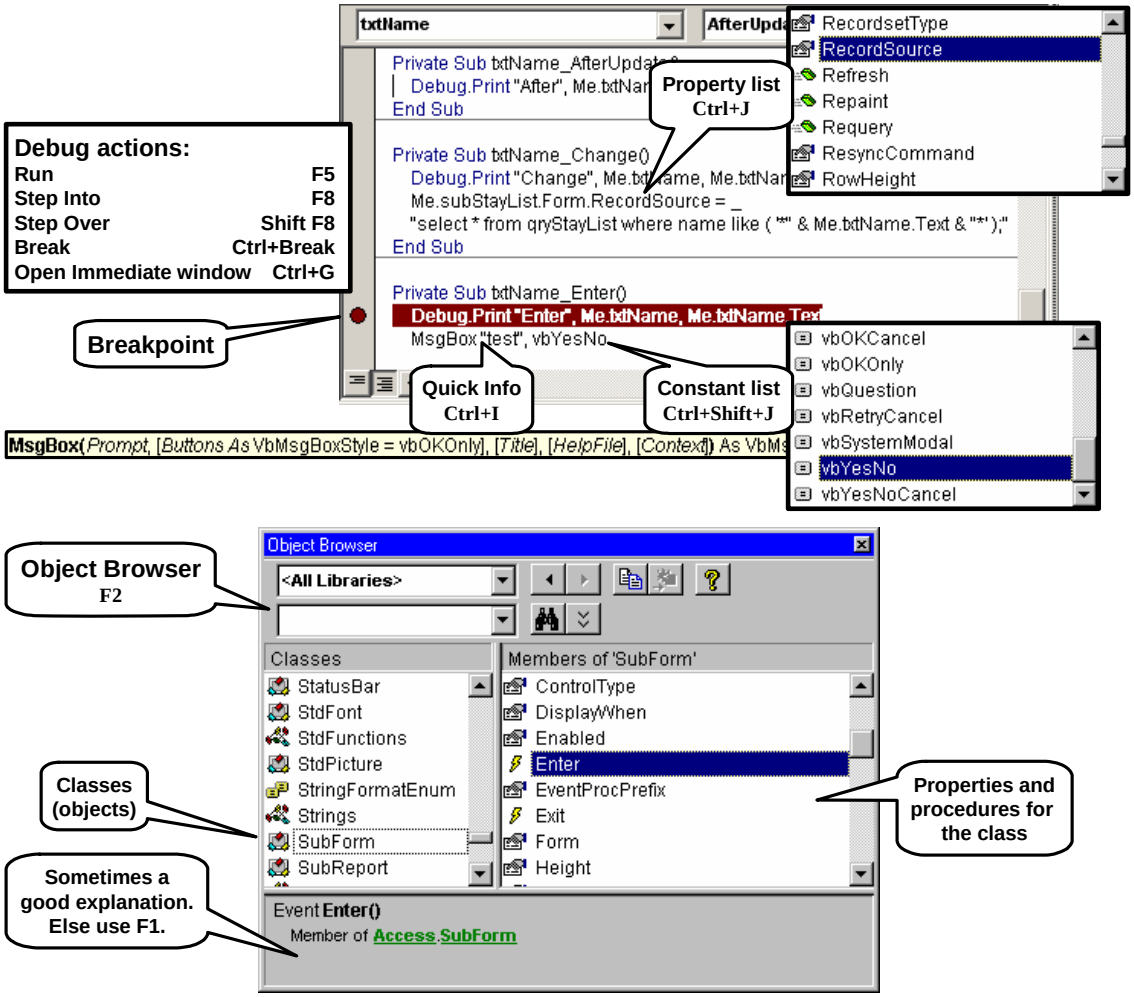
    When you have typed *MsgBox*, use Ctrl+I to bring up the list of parameters. Notice that most of the parameters are optional (enclosed in brackets). When you have typed the comma, bring up the list of possible constants with Ctrl+Shift+J. Select the right constant with the Tab-key.

When executed, the MsgBox statement will show a message to the user and ask for a Yes or No. (In section 3.6, we used MsgBox to print messages for a tool-based mockup.)

19. Put the cursor on *RecordSource* and use Ctrl+J to bring up a list of possible properties and procedures at this point.

20. Put the cursor on *txtName* and use Ctrl+J to bring up a list of possible controls at this point.

Fig 5.3B Breakpoint, pop-up help and object browser



Object Browser (F2) and Help (F1)

The Object Browser gives an overview of the classes in Access, Visual Basic, and your own database. Open the browser with F2 (Figure 5.3B). To the left you see a list of the classes, to the right all properties and procedures of the selected class (called "members" in the window). In the figure we have selected the *SubForm* class. The right-hand side shows the properties and event procedures.

For some properties and procedures, VBA shows a good explanation at the bottom of the window, for others you get a good explanation with F1 (Help). The Object Browser doesn't show all the classes available. For instance you will look in vain for the Debug class, which we have used several times already.

Another way to get help about classes, properties, and language structures such as *if-then*, is to position the cursor on the word in the VBA code, then use F1. Usually you get good help. Try it:

21. Put the cursor on the word MsgBox and use F1. You get an excellent explanation of the procedure and its parameters.

22. Put the cursor on the word *RecordSource* in the code and use F1. You get a reasonable explanation.

23. The quality of the help information varies. Put the cursor on the word Debug in the code and use F1. You may get the message *Can't find project or library*. Or you get an explanation, but not an exciting one. You may try F1 with the cursor on the word Print. You get a lot of information, although it is hard to find out what it means.

5.4 Command buttons

When the user clicks a command button, it receives a *Click* event. The event procedure must perform what the button is planned to do. We will look at a few typical examples.

Make a button open another form
We will first let the *New guest* button open a stay form in the simplest way (Figure 5.4).

1. On frmFindStay give the NewGuest button the name *cmdNewGuest*. (The usual prefix for a command button is *cmd*.)

2. Define the event procedure for the OnClick event. The procedure body should be:
```
DoCmd.OpenForm "frmStay"
```

The object *DoCmd* can do various things. Here we use it to open a form. You can use PgDown to step through all the stays, and even add a new stay at the end.

Open a form to show only one record
You may use *OpenForm* to open a specific stay:

3. Try to change the procedure body to
```
DoCmd.OpenForm "frmStay", , , "stayid=2"
```

If you try it in user mode, you see that stay 2 is visible and the user may edit it. Parameter 4 did the trick. It is a *filter* - a text that automatically enters an SQL-WHERE clause and restricts the visible stays. In practice, the program must compute the filter text so that 2 in the example becomes the stayID the user has chosen.

Section 5.5.2 explains more about opening a form for various purposes. Section 5.5.3 explains the many parameters for *OpenForm*.

Make a button reset the search criteria
The real FindGuest window has a button for resetting the search criteria. Try to add it:

4. In design mode, add a button to frmFindStay. Give it the label *Reset criteria* and the name *cmdReset*.

5. Define the event procedure for the OnClick event. The procedure body should be:
```
Me.subStayList.Form.RecordSource = _
    "select * from qryStayList; "
Me.txtName = ""
```

This procedure sets the record source to the full list of stays, and it sets the search criterion to an empty text. If there are more search criteria, for instance also the phone number, they should be set too, of course.

Other button events

The click event happens if the user clicks the button. What happens if the user tabs to the button and then presses *Enter*? Or if the user uses a shortcut key? The button doesn't notice. It receives a click event in all these cases.

A command button also receives the same events as a text box, for instance Enter, GotFocus, MouseDown, KeyPress. There is rarely a need to do something for these events. A button doesn't receive Change, BeforeUpdate and AfterUpdate. These events deal with storing some data, and a command button doesn't store anything.

Default button and Cancel button

A form may have a *default* button. If the focus is somewhere on the form or its subforms, and the user presses *Enter*, the default button gets a click event. (The exception is when the focus is on another button. Then this button gets the click, of course.)

You may define any button as the default. In the property box for the button, select the *Other* tab and set
```
Default = Yes
```

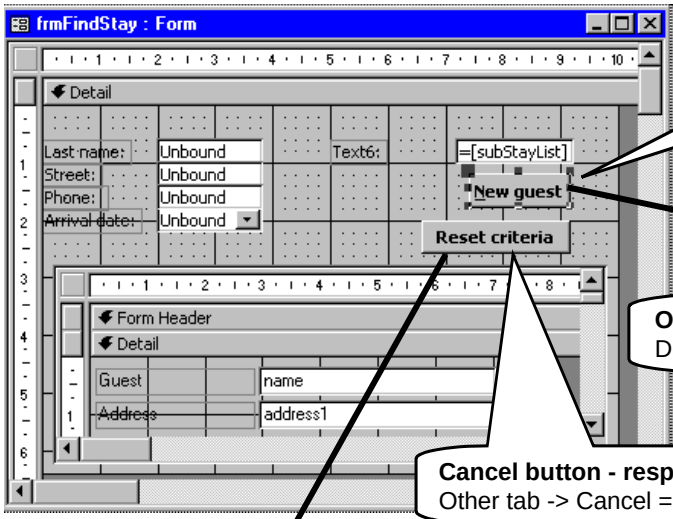
Access will automatically set *Default = No* for the previous default button on the form.

A form may also have a *cancel* button. If the focus is somewhere on the form or its subforms, and the user presses *Escape*, the cancel button gets a click event. You may select any button as the cancel button. In the property box, select the *Other* tab and set
```
Cancel = Yes
```

You will typically let the cancel button close the form without saving anything. This can be done with these statements in the event procedure:
```
Me.Undo
DoCmd.Close
```

The *Undo* procedure sets all fields on the form to their OldValue. As a result, Access will not save them at close.

Fig 5.4 Command buttons, default and cancel buttons



Default button - responds at Enter:
Other tab -> Default = Yes

```
Private Sub cmdNewGuest_Click()
    DoCmd.OpenForm "frmStay"
End Sub
```

Or open only one stay:
DoCmd.OpenForm "frmStay", , , "stayid=2"

Cancel button - responds at Esc:
Other tab -> Cancel = Yes

```
Private Sub cmdReset_Click()
    Me.subStayList.Form.RecordSource = "select * from qryStayList; "
    Me.txtName = ""
End Sub
```

5.5 Forms

Forms are very complex. They handle many events and have many properties. Below we explain the more important ones.

5.5.1 Open, close, and events

In Visual Basic you can open a form in this way:

```
DoCmd.OpenForm  "name of the form"
```

OpenForm can have many parameters, as explained in section 5.5.3. You can close the form that is in focus with

```
DoCmd.Close
```

As an example, a button on the form could close the form in this way. When the user clicks the button, the form is in focus.

If you want to close another form than the one currently in focus, you can specify the form through two parameters, for instance:

```
DoCmd.Close  acForm, "frmStay"
```

The parameter *acForm* says it is a form. The last parameter is the form name.

An alternative is to set the form in focus, and then close it, for instance like this:

```
Forms!frmStay.SetFocus
DoCmd.Close
```

Event sequence

During open and close, the form receives many events. Figure 5.5A gives an overview.

Open(Cancel). This is the first event that the form receives. At this point in time, the various data structures in the form have been created, including all controls. The subforms have been opened and have received their first *Current* event. Most controls also have the right value, but some have not. The form is invisible to the user at this point. The event procedure may set the parameter Cancel=True, thereby refusing to open the form.

Load event( ). Next the form receives a *Load* event. The form is still invisible, but all unbound controls have a value. Controls bound to a database record may still not have the right value. Accessing data in the database proceeds in parallel with opening the form, and bound controls gradually get the right value.

Resize( ). This event occurs during opening of the form, and when the user resizes the form by dragging its borders. The event procedure may adjust the controls on the form to better utilize the available space. As an example, a subform area may expand and contract in step with the main form. See section 5.5.13.

Activate( ). This event signals that the form, or one of its controls, will get the focus (the title bar becomes blue). The event doesn't occur when the user clicks from another application. Subforms never receive an Activate event. They are active when their main form is active.

GotFocus( ). This event occurs only when the form has no controls that can get the focus.

Current().This event occurs at open no matter whether the form is bound to the database or not. For bound forms, it also occurs when the user moves to another record, for instance with PageDown. This event is the place to update controls that depend on the current record, for instance detail windows.

When *Current* is called during open, the form is still invisible, but the controls have the right value. When *Current* returns, the form becomes visible.

Dirty(Cancel). This event occurs the first time the user edits some bound data in the current record. The event occurs right before the *Change* event of the control changed by the user. At return the Form property *Dirty* becomes True.

BeforeUpdate(Cancel). This event occurs when the form has data that is about to be saved in the database. The event is an opportunity to check consistency between pieces of data before they are saved.

The event procedure may set Cancel=True to stop saving and let the user change the data. It may also use *Me.Undo* to restore the old values and in that way skip the saving.

BeforeUpdate will occur when the form is to be closed or when focus moves to another record on the form. Canceling the update means that the form will not be closed or focus not moved.

BeforeUpdate will not occur when focus moves to another Access form. However, it will occur when focus moves to a subform on the same form, or when the user switches to a different tab sheet on the same form. This makes it difficult to maintain consistency between data in the main form and data in its subforms.

AfterUpdate( ). This event occurs when values have been saved in the database. Also the OldValue properties have been set to the saved values. The event is an opportunity to act on the new data.

Unload(Cancel). This is the first event when a form is closed. The event procedure may check that everything is okay. If it returns Cancel=True, the form will remain open.

Fig 5.5A Event sequence, Form open and close

User action	Events	Property changes before call
Open	Open(Cancel)	The Form and its controls are created, but not visible. Some controls are not initialized. Subforms are open. The procedure may refuse to open the form (Cancel=True).
	Load	All controls are initialized, except bound ones. The Form is still invisible.
	Resize	(Opportunity to reposition controls)
	Activate	(Signals that the form will get focus)
	(GotFocus)	(Only for Forms where no control can get the focus)
	Current	A current record has been selected. Bound controls have a value. At return, the form becomes visible.
Edit	Dirty	First time the user edits some data in the record.
PageDown, etc.	BeforeUpdate(Cancel)	Some bound fields to be saved. The procedure may refuse to save.
	AfterUpdate	Bound fields saved. All values = OldValue.
	Current	Form is invisible. A new record is current. At return, the Form becomes visible.
Click in subform	BeforeUpdate(Cancel) AfterUpdate	As for PageDown. As for PageDown.
Close	Unload(Cancel)	Form to be closed. The procedure may refuse and the form stays open.
	Deactivate	(Signals that the form will lose focus)
	Close	At return, the form becomes invisible and the subforms are closed. Memory is released.

Deactivate( ). This event signals that the form will lose focus. It occurs during close and when the user clicks in another main form. It doesn't occur when the user clicks in another application. Subforms never receive a Deactivate event.

Close( ). This is the last event the form receives. The form is still visible and all subforms are still open. When *Close* returns, the form becomes invisible, all subforms are closed, and data structures are deleted.

5.5.2 CRUD control in Forms

When you create a form in the default way and open it with *DoCmd.OpenForm*, the user is allowed to step through all records, and add several new records. Depending on circumstances, we may want a more restricted behavior. We may for instance want the user to see and edit only one single stay. Or we may want the user to create only one new record.

CRUD and filter properties

The way to control this is through the following properties, called *CRUD* properties because they specify Create, Read, Update, and Delete of records (Figure 5.5B gives an overview). You can see their initial values on the form's data tab. You can change them at run time through Visual Basic:

AllowEdits. If True, the user can edit fields in existing records. Default=True.

AllowDeletions. If True, the user can delete a record with the Del key. (Requires that the record selector is displayed in the form.) Default=True.

AllowAdditions. If True, there will be an empty record at the end of the list for adding new records. Default = True.

Filter. A text that works as a WHERE clause in the query behind the form. For instance, you may set filter to *"stayID=740"*. The result will be that the user only sees the stay with ID=740. (Don't write the word WHERE itself.)

FilterOn. If True, the Filter property works and selects records. If False, the filter has no effect. FilterOn is a property that you cannot see in the form's property box, but you can set it through the program or

the Immediate window. The user can toggle FilterOn with the filter icon on the toolbar.

AllowFilters. If True, the user is allowed to set filters on and off through icons on the toolbar. The program can always set filters independently of AllowFilters.

DataEntry. If True, the user will not be able to see old records. If AllowAdditions is also True, the user is allowed to add new records. If the program sets AllowAdditions to False while the user is editing a new record, the user is allowed to finish editing the record, but cannot add further records. Default = False.

Setting DataEntry to False has an undocumented side effect: It also sets *FilterOn* to False, in that way blocking filters set up by the program. In this case, let your program set FilterOn after setting DataEntry.

All of these properties are dialog variables that don't survive closing of the form. What you have specified in the form's property box are simply the default value for these variables. You may try out the properties in this way:

1. Open frmStay in Datasheet mode. You should see all records from the stay query. At the end of the datasheet you should see an empty line for entering a new record.

2. Open the Immediate window with Ctrl+G. Adjust the sizes so that you see the datasheet and the Immediate window at the same time. Try to change the settings with statements such as:
```
Forms!frmStay.AllowAdditions = False
Forms!frmStay.Filter = "stayid=740"
Forms!frmStay.FilterOn = True
```
The datasheet should change accordingly, removing the empty line, showing only one record, etc.

See and edit a single record

In order to see and edit a single record, set these properties in the form's property box (see overview in Figure 5.5B):
```
AllowEdits = True
AllowDeletions, AllowAdditions, DataEntry = False
```

Then open the form with a filter, for instance:
```
DoCmd.OpenForm "frmStay", , , "StayID=2"
```

In practice, the program must compute the filter text so that 2 in the example becomes the stayID the user has chosen.

Update the record. When the user enters something, it is not stored in the database until he closes the form (or enters a subform). Sometimes storing is needed earlier. The program can do it with this statement:
```
Me.Recordset.Move(0)
```

This trick assumes that Access is configured for DAO 3.6 (see section 5.6). The statement works on the open recordset bound to the form. It moves current record back or forth a number of records, and as part of this it saves the current record. In our case we move zero records away - to the same record. But we get the update anyway. (See more in section 5.6.3.)

Create a single record

In order to create a single record without seeing existing records, set these properties in the form's property box (see overview in Figure 5.5B):
```
AllowAdditions = True
DataEntry = True
```

This will still allow the user to enter a record and then move on to enter a new record. To avoid this, block further record creation when the first record is updated:

```
Private Sub Form_AfterUpdate()
    Me.AllowAdditions = False
End Sub
```

Open the form with
```
DoCmd.OpenForm "frmStay"
```

Prevent two blank lines in datasheet

The method above for creating a single record works okay when the Form is shown in normal mode. In Datasheet mode, however, the user may be confused at what happens. The user sees the blank record at the end of the datasheet, but as soon as he starts typing in it, another blank line appears.

Access creates the second blank line right after calling the Dirty-event procedure. However, it is impossible to adjust things at this point because the user's character hasn't yet ended where it should be.

The solution is to store something in the new blank record before the user types anything. This initiates a temporary record for further editing. Then AllowAdditions is set to False to prevent further new records. This can be done at the Current event:

```
Private Sub Form_Current()
  If Not IsNull(Me.someField) Then Exit Sub
  Me.someField = " "
  Me.AllowAdditions = False
  Me.someField = Null
End Sub
```

The first statement checks whether something has been stored in the field to be initialized. In this case the user has selected an existing record and nothing should be done now. The next statement stores a space in the field. This creates the temporary record. Next AllowAdditions is set to False. This prevents further records being created. Finally the field is set back to the initial empty state.

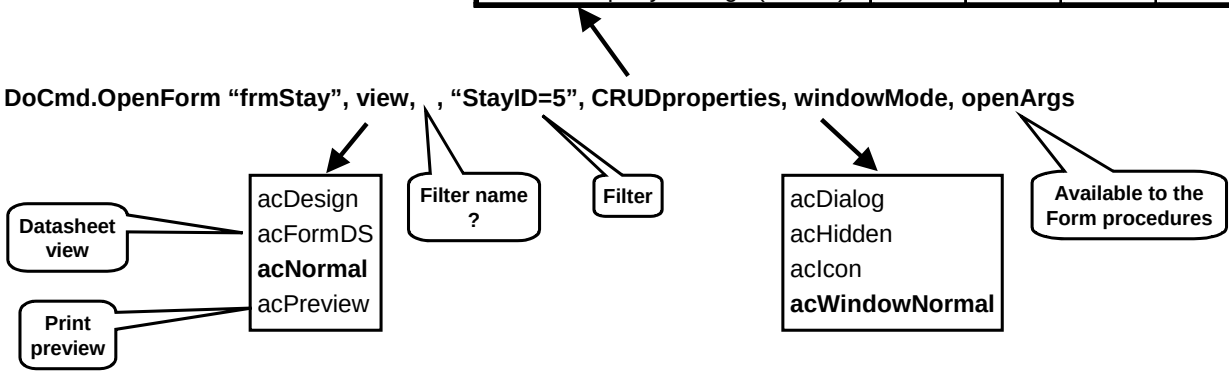
The solution must be extended with means to add another record, for instance a button or a response to Enter. Care must also be taken not to leave empty

Fig 5.5B Controlling Open, CRUD, and filter

Form properties: CRUD and Filter

AllowEdits: See and edit data
AllowDeletions: Del-key works
AllowAdditions: Empty record at end
DataEntry: Old data invisible
Filter: e.g. StayID=740
FilterOn: Use filter
AllowFilters: User-controlled filter

	AllowEdits	Allow-Deletions	Allow-Additions	DataEntry
To see and edit a single record: (Also set the filter)	True	False	False	False
To create a single record: Initially	x	False	True	True
At *Form_AfterUpdate*	x	False	False	True
acFormEdit	True	True	True	False
acFormAdd	True	True	True	True
acFormReadOnly	False	False	False	False
acFormPropertySettings (default)	x	x	x	x



records in the database in case the user doesn't enter something.

5.5.3 The OpenForm parameters

From VBA you normally open a form with *DoCmd.OpenForm*. A lot of parameters determine what goes on (see the details on Figure 5.5B).

Form name. The first parameter is the name of the form, for instance "frmStay".

View. The second parameter specifies whether the form is to be shown in design view, as a datasheet, as a normal form (default), or as a preview of a print.

Filter name. The name of a query, for instance "qryStay". (I have not been able to figure out what this parameter does.)

Filter. A WHERE condition for the records to show, for instance "StayID=5".

CRUDproperties. A choice of CRUD combinations that determine whether records can be edited, added, etc. Figure 5.5B shows that *acFormEdit* sets AllowEdits, AllowDeletions, AllowAdditions to *True* and DataEntry to *False*. In other words, the user can do anything to the records. At run time these settings overrule the initial settings from the Form's property box.

As another example, *acFormPropertySettings* (the default), doesn't set any of the CRUD properties but lets the initial settings rule.

WindowMode. Specifies whether the form is to be shown as a dialog box, a hidden window, an icon, or a normal window. A dialog box keeps the focus until the user closes it. It has borders that cannot be resized. A hidden window is like a normal window except that it is invisible. The event procedures can make it visible with *me.Visible=True*. The window mode *acIcon* seems to have no effect.

OpenArgs. This parameter is stored in the property *OpenArgs* in the Form. It can tell the event procedures in the Form what to do. The parameter might for instance specify whether the form is to be used for creating a new record or viewing an existing one, in this way allowing the Form to do it in its own way. Below we show how this is used in the hotel system to let the same form handle a new guest, a new stay, or an existing stay.

5.5.4 Multi-purpose forms (hotel system)

In the hotel system we use *frmStay* for creating guest and stay records, as well as editing existing records. Using the same Form is an advantage to the user (for recognition) and for the developer (he needs to maintain only one Form).

Now, how does the Form know what it should do? OpenArgs is intended for such things. In the hotel system, we use OpenArgs in this way:

- **OpenArgs < 0:** Open an existing stay. The WhereCondition parameter in the OpenForm statement selects the right one.
- **OpenArgs = 0:** Create a new guest and a new stay.
- **OpenArgs > 0:** Create a new stay for an existing guest. OpenArgs is the guestID for the existing guest.

Figure 5.5C shows the full solution. It deals also with error cases where the user for instance clicks the ShowStay button, but hasn't selected a stay. The Find Guest screen has three buttons for opening a stay. The figure shows the three click-event procedures.

cmdNewGuest

This is the simplest of the buttons. Independent of what the user has chosen, the button should create a new guest and a stay for him. What is needed is to open the form with OpenArgs=0.

cmdNewStay

Here we have two situations. (1) The user has selected a line, and thus a guest. (2) The list is empty so the user has not chosen anything. The property *CurrentRecord* allows the program to distinguish the two situations. CurrentRecord gives the sequential number of the selected record. It is zero if nothing is selected. If the list is empty, we might claim it is an error to make a new stay for a non-existing guest, but it turns out that users find it natural to create a new guest in this case. So this is what the program does; it opens the form with OpenArgs = 0. If a line is selected, it opens the form with OpenArgs = guestID.

Notice the If-Then-Else-EndIf construct that makes the program choose alternative paths, either executing the Then-statements or the Else-statements. The line-breaks must be as shown, with Else and EndIf on lines of their own.

Notice also the comment we have put after *Then* to explain to the reader what situation the program deals with after *Then*.

cmdShowStay

Here we have three situations. (1) A stay is selected. (2) A guest without stay is selected. (3) The list is empty. In the last two cases, the program cannot open a stay. The current value of *stayID* allows the program to distinguish. StayID is >0 in the first case, and Null in the last two cases. Notice that *stayID* exists even if the list is empty and CurrentRecord is zero. The value of *stayID* is just Null.

When *stayID*>0, we open the form with OpenArgs=-1. We also have to set the filter so that only this stay is visible. The WhereCondition-parameter to OpenForm allows us to do this in a simple manner. If the user for instance has selected a line with stayID=740, the WhereCondition has to be the text

```
stayID=740
```

In the program we compute this text by concatenating the text "stayID=" with the current *stayID* using the &-operator. *StayID* is a number, but Visual Basic converts it to a text before the concatenation.

When *stayID* is Null (or a number <=0) we have an error situation. The program may show a message using *MsgBox*, but in our case we just make the program beep. The DoCmd-object can do this too. (Usability tests show that users understand what is wrong without any message explaining about it.)

FrmStay, Load event

Inside *frmStay* an event procedure must take care of setting the right CRUD properties. Access calls the Load-event procedure during open, and this is the place we set the CRUD properties. Figure 5.5C shows the procedure. Let us follow the path through the code for each of the three cases.

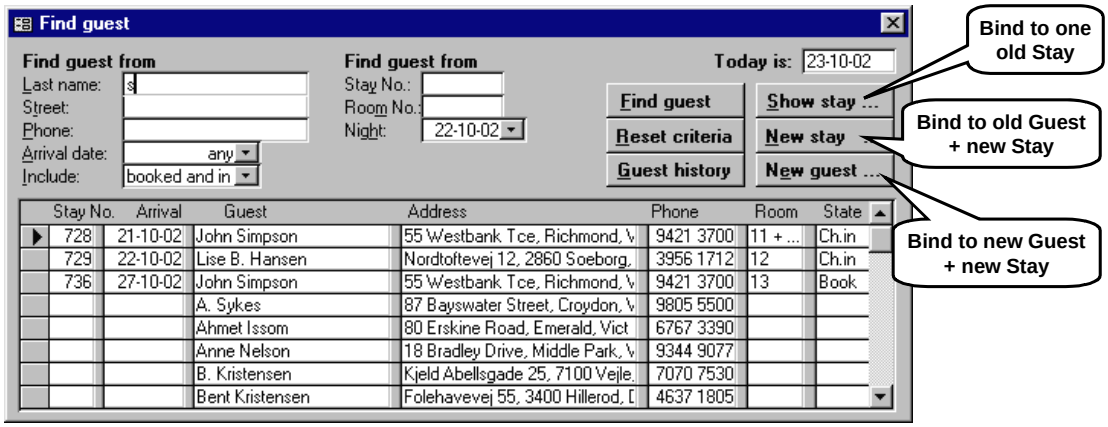
OpenArgs is < 0: We open an existing stay. The WhereCondition parameter has set the filter, so we don't have to do much. The procedure just executes the last line, which sets AllowAdditions to False.

OpenArgs = 0: We must create a new guest record and a new stay record. First we allow additions and data entry. Data entry means that existing records are not visible, so the current record is a new "blank line". FrmStay is bound to a join of tblGuest and tblStay, so the current record will now consist of Null data for the guest as well as the stay. In order to create the records, we have to store something in them.

First we set the guest name to a single space character. This creates a new guest record, and Access gives it an AutoNumber. Notice how we address *name* with a bang (!) to distinguish it from the built-in *Name* property.

Next we set the foreign key of the stay record to this guest. This creates the stay record and also links it properly to the guest. Then we set the name back to a Null so that the user doesn't see a name starting with a space when he enters the guest name. Notice the strange use of square parentheses. We explain more about them below.

Fig 5.5C Edit or create connected records through a form




```
Private Sub cmdNewGuest_Click()
   DoCmd.OpenForm "frmStay", , , , , , 0
End Sub
```

OpenArgs

```
Private Sub cmdNewStay_Click()
   If Me.subStayList.Form.CurrentRecord = 0 Then  'No guest selected
      DoCmd.OpenForm "frmStay", , , , , , 0
   Else
      DoCmd.OpenForm "frmStay", , , , , , Me.subStayList.Form!guestID
   End If
End Sub
```

frmFindStay

OpenArgs

```
Private Sub cmdShowStay_Click()
   If Me.subStayList.Form.stayID > 0 Then 'Stay selected
      DoCmd.OpenForm "frmStay", , , "stayID=" & Me.subStayList.Form!stayID, , , -1
   Else
      DoCmd.Beep
   End If
End Sub
```

WhereCondition. Becomes the Filter

```
Private Sub Form_Load()
' OpenArgs: -1: existing stay, filter set; 0: new guest; >0: new stay for existing guest, OpenArgs = guestID.
' Assumed CRUD settings: AllowEdits = True, DataEntry = False.
   If OpenArgs >= 0 Then            ' NewStay or NewGuest
      Me.AllowAdditions = True
      Me.DataEntry = True           ' Don't see existing records
      If OpenArgs = 0 Then          ' NewGuest
         Me!name = " "               ' Set blank name to create a dummy guest. Access makes guestID
         Me.[tblStay.guestID] = Me.[tblGuest.guestID]       ' Create a stay record. Access makes stayID
         Me!name = Null              ' Remove the blank name
      Else                          ' New stay for existing guest
         Me.[tblStay.guestID] = OpenArgs  'Create a stay record. Access makes stayID
      End If
   End If
   Me.AllowAdditions = False        ' All cases. No more additions
End Sub
```

frmStay

Finally we set AllowAdditions to False so that Page-Down will not bring the user to a new empty record.

OpenArgs is > 0: The guest exists already and OpenArgs is the guestID. First we allow additions and data entry. Data entry makes the current record a new "blank line". Next we set the foreign key of the stay record. This creates the stay record and links it to the guest. Access automatically joins it to the guest, so the guest data is now in the "blank line".

Finally we set AllowAdditions to False to prevent further record creation.

The strange [ ]
The program has a strange use of square parentheses. The reason is that the query behind the form has two guestID fields, one from each table. None of these are visible on the form, but they are available anyway as fields. The SQL-engine has given them the names

tblStay.guestID and tblGuest.guestID

In SQL-view of the query they sometimes appear as
[tblStay].[guestID] and [tblGuest].[guestID]

However, when we address them in Visual Basic, we have to write
Me.[tblStay.guestID] and Me.[tblGuest.guestID]

Confusing? Yes, very, but Visual Basic needs these parentheses to indicate that *tblStay.guestID* is one single name.

Note that we use the dot-operator after *Me* to address these fields. This allows us to use Ctrl+J to get a list of all properties and fields. You will find *tblStay.guestID* on the list, but you have to add the parenthesis yourself.

See the mechanisms live

The Load event is the more interesting part of the solution. You may try it out in this way:

1. Open frmStay and use the property sheet to give it a Load-event procedure. Type in the procedure from Figure 5.5C. Set a breakpoint at the first line. Close the form.

2. Open the Immediate window with Ctrl+G. Now simulate the NewStay button with this command in the Immediate window:
   ```
   docmd.OpenForm "frmstaymono",acFormDS , , , , ,2
   ```

3. The command starts opening the form and will show it in Datasheet mode. The Load procedure stops at the breakpoint, but the form is not yet visible. Make it visible with this command in the Immediate window:
   ```
   me.Visible = True
   ```

4. The form should now appear in Datasheet mode. Step through the Load procedure with F8 to see what happens to the datasheet. When the procedure sets DataEntry, the datasheet should reduce to a blank line. When the procedure stores OpenArgs in the record, the guest data should appear in the blank line.

You may try out the other cases in the same manner.

5.5.5 Dialog boxes (modal dialog)

Many of the boxes you see in Windows applications are dialog boxes. When a dialog box pops up, you have to fill in what is asked for and then close the box. You cannot look around at other windows until you have closed that dialog box. This is a *modal dialog*. One example is a message box. Another example is the Options box that many applications have.

A dialog box is a special case of a Form. Apart from being modal, it has narrow borders, and the user cannot drag the borders to change the size of the box.

There are several ways to open a dialog box:

Message box

- Use the MsgBox function, for instance:
  ```
  MsgBox "the message", vbYesNo+vbQuestion
  ```

This opens a box with the message and shows a number of buttons (Yes and No in the example), plus an icon (a question bubble in the example). You can check the return value to see which button the user chose, for instance:
```
If MsgBox("the message", vbYesNo) = vbYes Then . . .
```

Notice that we write MsgBox with parameters in a parenthesis when we need to look at the return value. Use F1 to see the help information about MsgBox. There is an excellent explanation of all the details.

Multi-line message

When the message is more than one line, you have to compose it in this way:
```
MsgBox "line 1" & Chr(10) & "line 2", vbYesNo
```

Chr(10) is the new-line character, or more precisely a text consisting of the new-line character. In Visual Basic you cannot write a new-line character inside the text string itself.

Use OpenForm with acDialog

- Make a Form in the usual way with data fields, buttons, etc. Open it with DoCmd and the acDialog parameter, for instance:
  ```
  DoCmd.OpenForm "form name", , , , ,acDialog
  ```

OpenForm sets the right border on the form and makes it modal. There is no return value. The form has to store results somewhere, for instance in global variables or in the database.

Set the dialog properties of the Form

The last way to make a dialog box is to set the Form properties yourself:

- On the Format tab for the form, set Border Style to *Dialog*. The effect of this is that the border of the Form is narrow and the user cannot drag the borders to change the size.

- On the Other tab set PopUp to Yes. When you open the form, it stays on top of other forms. It is not modal however. You can work with other forms while the dialog form is open.

- On the Other tab set Modal to Yes. In **Access 97** this makes the form modal. When open, the user cannot work with other forms. In **Access 2000 and 2003** the Modal property has no effect at all (although the help text says it has).

You may then open the form with DoCmd.OpenForm without specifying the acDialog parameter.

5.5.6 Controlling record selection

A bound form is connected to a record set (a database table or a query). At any given time one of the records is the *current* record. When you see the recordset in datasheet mode, the current record is marked with an arrow. When you see it in Form mode (normal mode) only the current record is visible.

In many cases we want the program to move to another current record. Figure 5.5D illustrates some ways to do this. The Form is bound to tblGuest and shows one guest at a time. At the top of the form we have added some navigation controls. One is a combo box where the user can select a guestID from a drop down list. The list shows guestID as well as guest name. There are also two buttons that navigate to the previous and the next record.

Move and see one record

First we will look at the situation where we want to navigate to the guest selected by the guestID. The user should not see other guests than this one.

The solution is to set the filter properties of the Form when the user selects something with the combo box. The AfterUpdate procedure for the combo box could look like this (top left of the figure):

```
Private Sub cboGuestID_AfterUpdate()
   Me.FilterOn = True
   Me.Filter = "guestID=" & Me.cboGuestID
End Sub
```

The filter is a condition to be used by Access in a hidden WHERE clause when retrieving the Form's record set. It is a computed text. For instance, when the user selects guest 2, the text becomes

```
guestID=2
```

Scroll to a record

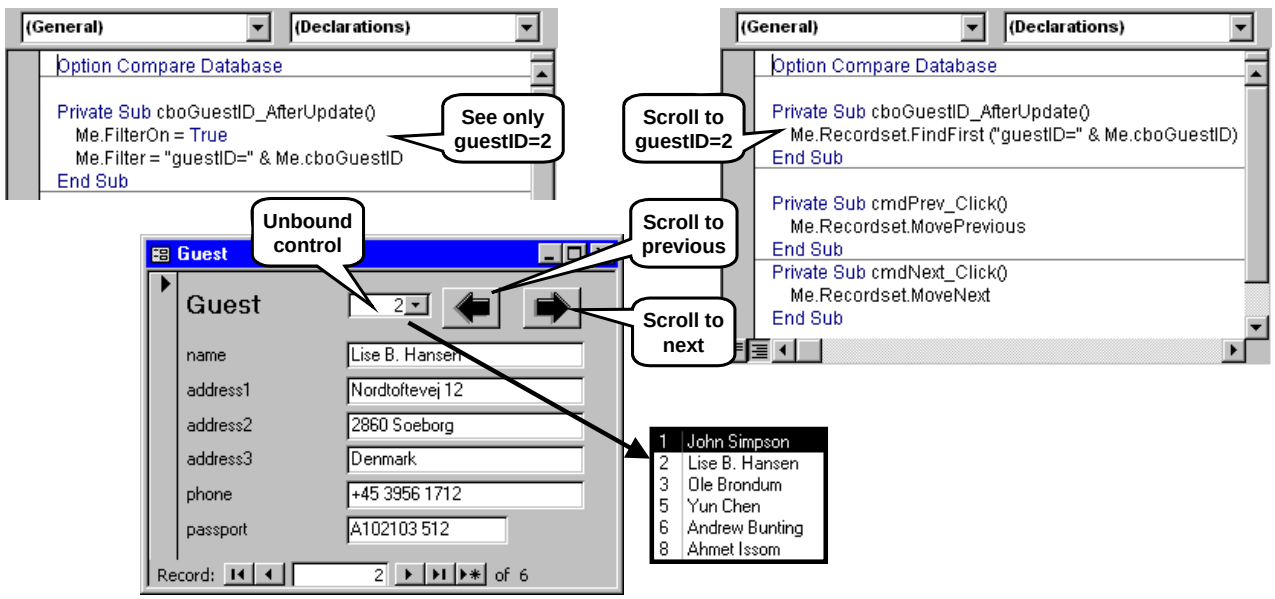
Next we will look at the situation where we want to navigate to the guest, but the user should be allowed to move back and forth from there. In this case we cannot use the filter. The solution is to work directly on the Form's recordset. The AfterUpdate procedure for the combo box could look like this (top right of the figure):

```
Private Sub cboGuestID_AfterUpdate()
   Me.Recordset.FindFirst ("guestID=" & Me.cboGuestID)
   ' Might also be written without a parenthesis
End Sub
```

The recordset has a FindFirst function with a condition as a parameter. It finds the first record in the set that matches this condition. Then it makes this record the current record.

The two arrow buttons also work directly on the recordset. They use the MovePrevious and MoveNext functions for moving *current* to the previous or next record. Actually, these buttons do the same as PageDown and PageUp. Section 5.6 explains more about recordsets.

Fig 5.5D Controlling record selection



5.5.7 Column order, column hidden, etc.

When a form (or subform) is shown in table view, the user can rearrange and hide columns. To rearrange columns, the user selects a column and drags it to a new position. To hide it, he selects the column and uses Format -> Hide columns.

The program may need to know these settings or it may need to change them.

Remember that each column corresponds to a control on the form (section 3.2.1). As an example, column 2 on Figure 5.5E corresponds to a text box control bound to the *description* field of the table. The column heading (Class) is the label associated with the text box (or the name of the text box if there is no label).

The program can detect and set column order, etc. through these three attributes of the controls:

ColumnHidden (attribute, read and write). True or False.

ColumnOrder (attribute, read and write). The number of the column. Columns are counted from 1 and up in the sequence that the user sees. However, hidden columns are included in the counts.

ColumnWidth (attribute, read and write). The width of the column measured in twips (see section 5.5.13). If the program sets ColumnWidth to -2, the width is adjusted to fit the data in the column. Width = -1 means a default width. Width = 0 makes the column hidden (and you have to explicitly unhide it - setting the width is not enough).

Example: Suppose the program needs to change the column width of Class (the *description* control) in Figure 5.5E. An event procedure on the main form could do it in this way:

```
Me.subRoomGrid.Form ! description.ColumnWidth = 1500
```

5.5.8 Area selection, SelTop, etc.

When a subform is shown in table view, the user can select a rectangular area of the subform. Figure 5.5E shows an example where the user has selected column 3 and 4 from the fourth and fifth row (see more about the room grid in sections 7.4 and 7.5).

The program can detect which area the user has selected. The figure shows how we can try it out through the Immediate window (Ctrl+G). We address the first open form, its subform control, and the subform it is bound to. When asking for the value of SelTop, we get 4 because the top row in the selected area is record number 4.

As the figure shows, we can also change the area size and location. We have changed the area width to 2. The following four attributes of the form control the area.

SelTop (attribute, read and write). The number of the top row in the selected area. The first record in the table behind the form is row 1, the next 2, etc. Notice that the first records may be scrolled out of sight.

SelHeight (attribute, read and write). The number of rows selected. If SelHeight is zero, no area is selected, but the *record* is still selected.

SelWidth (attribute, read and write). The number of columns selected. Hidden columns are included. If SelWidth is zero, no area is selected, but the *record* is still selected.

SelLeft (attribute, read and write). The number of the left column in the selected area. In Access 97 the columns are numbered from 1 and up. In Access 2000 they are **numbered from 2 and up** when an area is selected.

Beware: The strange change from 1-based numbering to 2-based numbering must have puzzled quite a number of developers. Obviously it is an error. The situation is even stranger when only a record is selected, but not an area. In this case you set SelLeft with 1-based numbers, but retrieve it with 2-based numbers.

If columns have been reordered, their numbers follow what the user sees. However, **hidden columns** are included in the counts.

Keeping the area selected

In the hotel system, the user selects rooms through a form like Figure 5.5E. In principle this is easy. However, as soon as focus moves from the subform to the main form, Access removes the area selection. This happens for instance when the user clicks a button in the main form. The event procedure behind the button wouldn't even know which area was selected.

The program has to compensate for Access's strange behavior. When focus leaves the subform, the Exit event procedure saves the area size attributes before Access removes the area selection. When some control on the main form gets the focus, its event procedure sets them back. There is no common event on the main form that can do it, so each control has to act.

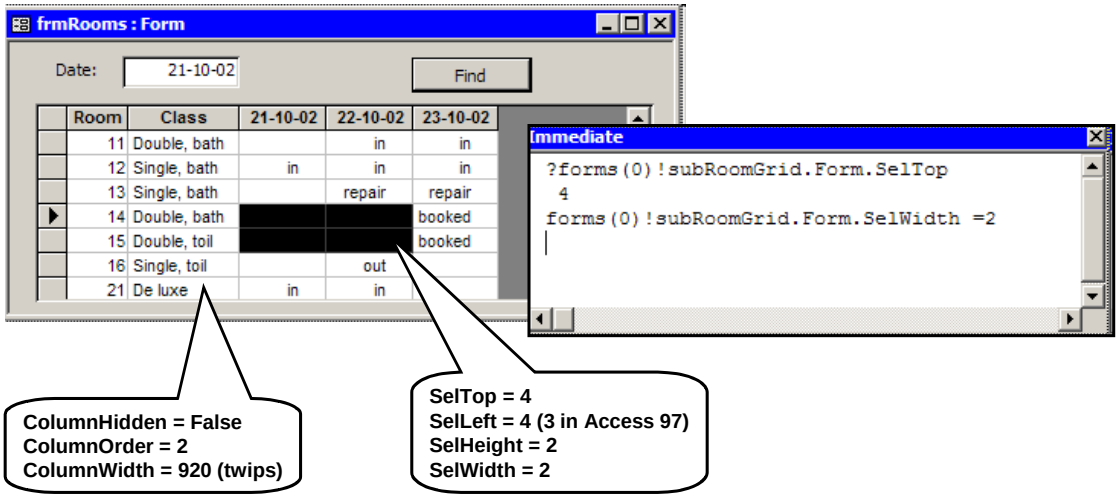
Main form

In the main form, declare these variables:

```
Public aWidth As Long, aHeight As Long, aLeft As Long
' These variables keep track of the current area size. aleft is the correct column number, not the Access2000 distortion. SelTop is not saved since current record always is the top.
```

The following procedures on the main form update and use these variables (shown in the Access 2000 version).

Fig 5.5E Selected area, SelLeft, SelTop, SelWidth, SelHeight




```
Private Sub subRoomGrid_Exit(Cancel As Integer)
' This event happens when focus moves from the subform to
the main form. Save the current area size before Access
removes it.
   aWidth = subRoomGrid.Form.SelWidth
   aHeight = subRoomGrid.Form.SelHeight
   aLeft = subRoomGrid.Form.SelLeft - 1
   ' -1 to correct for the Access2000 error
End Sub

Private Sub resetSelection() ' Shared procedure
   subRoomGrid.Form.SelWidth = aWidth
   subRoomGrid.Form.SelHeight = aHeight
   ' Make sure to set Width and Height before setting Left
   subRoomGrid.Form.SelLeft = aLeft + 1
   ' +1 to compensate for the Access 2000 error.
End Sub

Private Sub txt . . ._GotFocus()
' Reset the area size when a main-form control gets the focus.
   Call resetSelection()
End Sub

. . . (GotFocus for other controls)

Private Sub subRoomGrid_Enter()
' When focus moves from the main form to the subform.
Access removes the area selection just before this event
(Access 2000 only). So reset the area size
   Call resetSelection()
End Sub
```

User actions in the subform

The solution above works fine when an area is selected in the subform. For the hotel system, the program selects a suitable area corresponding to a free room. If the user clicks *Check in*, for instance, everything is okay.

If the user expands the area by means of Shift + arrow key, everything is fine too.

But what happens if the user clicks a cell to move the area to another room? Access will remove the area selection and the user will just see the cursor flashing in the cell he clicked. The user may drag the cursor to select an area, or use Shift + click, but it is not intuitive. The program should make sure that a suitable area is selected at any time.

One problem is that as soon as the user clicks a cell, Access removes the area selection before calling any event procedure. The program cannot catch the old selection at this point.

The solution is to let the program act at MouseUp and KeyUp. There are two situations:

An area is selected: This happens if the user expanded the area with Shift + arrow key, or if the user dragged an area with the mouse. The program should save the area size.

No area is selected: This happens if the user has clicked a cell or moved to the cell with an arrow key. The program should reset the area size so that the selected cell becomes the top left of the area.

Each single control on the subform has to react this way. Although there is a Form_MouseUp and a Form_KeyUp, they don't respond to clicks inside the grid. They respond only to clicks on the grid border, i.e. the headings and the record selectors.

In the hotel system, fsubRoomGrid has one control for each column in the grid. The names of these controls are cbo1, cbo2, etc. (see section 7.5 for details). So we need these procedures on the subform:

```
Public Sub saveSelection() ' Save the current area selection
after key up or mouse up, or reset it after clicks, etc.
   If SelHeight = 0 Then ' No area selected. Reset area.
      ' Check first that the new area is within the grid.
      SelHeight = Parent.aHeight
      SelWidth = Parent.aWidth
   End If
   Parent.aWidth = SelWidth
   Parent.aHeight = SelHeight
   Parent.aLeft = SelLeft - 1 ' Access 2000 compensation
End Sub

Private Sub cbo1_KeyUp(KeyCode As Integer, Shift As
Integer)
   Call saveSelection()
End Sub

Private Sub cbo1_MouseUp(Button As Integer, Shift As
Integer, X As Single, Y As Single)
   Call saveSelection()
End Sub

. . . (KeyUp and MouseUp for the other controls)
```

Notice that when saveSelection resets the area size, it should first check that the area is within the grid. The user may for instance have clicked at the far right with a wide area selected.

Also notice how the program uses *Parent* to address the saved area size.

This solution works okay. The user can use mouse or keyboard to move an area around in the grid. However, the screen flickers. It is clearly visible that Access removes the area selection before the program resets it. Below we explain how to deal with it.

By the way, since the subform saves the area size on its own, there is no reason that the subform control on the main form saves it too. So the event procedure `subRoomGrid_Exit` may be removed.

Clicking in the grid border

If the user clicks in the grid header, Access selects the entire column. If he clicks in the record selector in the left border, Access removes the area selection and selects the record the user clicked. In both cases we want the old area selection to be reset.

The place to deal with this is the MouseUp procedure for the subform, *Form_MouseUp*. It should set the area size in the same way as *saveSelection()*. However, here is another Access error. Access refuses to change SelHeight etc. The work-around is to start out with

```
SelHeight = 0
```

This is necessary even if SelHeight is 0 already (at least in Access 2000).

Avoiding flicker

It is possible to avoid the flicker by letting the program take action before Access removes the area selection. Two event procedures on the subform are involved:

`Form_KeyDown(KeyCode As Integer, Shift As Integer)`
This is the *Key preview* procedure (see section 5.5.9). It must handle arrow keys and tabs to the left or right by setting SelLeft and aLeft - after checking that the area is inside the grid, of course. Setting SelLeft causes Access to not remove the area.

`Form_Current()`
Is called when the user uses arrow up or down, or clicks in a new row. It is called before MouseUp or KeyUp occurs. Sets SelWidth and SelHeight early on to avoid flicker. If Current doesn't do anything, the area will be set at MouseUp or KeyUp, but Access will remove the area in between, causing flicker.

Getting the selected data

When a function acts on the selected area, it needs to know which data the area contains. If the area is only one row high, it can directly address the controls in the current subform record. In general, however, it needs to retrieve data for several rows. The data has to be retrieved from the recordset bound to the form.

One solution is to retrieve a number of rows from the recordset by means of the GetRows property. Here is an outline of the solution for the hotel case.

In the main form, declare a variant array:

```
Dim A()
' The A-array will receive a number of records
```

The basic idea is to use this piece of code inside the button event procedure:

```
A = Me.subRoomGrid.Form. Recordset. GetRows(n)
```

We address the subform (fsubRoomGrid) in the usual way. The property *Recordset* is the recordset bound to the subform. The property *GetRows* copies n records from the current record and on. It stores the copies in the array A.

While the area size and position is 1-based (and 2-based), the A array is zero-based. As a result, A will hold this:

A(0, 0):	The first field in the first record.
A(0, 1):	The first field in the second record.
A(1, 0):	The second field in the first record.
. . .	and so on.

As a side effect, GetRows moves the current record forward so that it points to the row after the selection. To avoid this, use the built-in clone of the recordset, make it point to the current record, and then copy the rows through it:

```
Me.subRoomGrid.Form.RecordsetClone.Bookmark = _
Me.subRoomGrid.Form.Recordset.Bookmark
A = Me.subRoomGrid.Form.RecordsetClone.GetRows(n)
```

Section 5.6 explains more on this.

Dealing with re-ordered columns

A small problem remains. If the user has re-ordered the columns in the datasheet, the order of the record fields and the datasheet columns don't match.

To overcome this, use the ColumnOrder property of the subform controls. This property gives you the column number (1-based) for the control. The ControlSource property of the control gives you the field name that the control is bound to. See section 5.5.7 for more on ColumnOrder.

5.5.9 Key preview

When the user clicks something on the keyboard, the control in focus will receive several *key events*. However, the Form can have a look at the key events before the control gets the events. This is useful when no specific control has to take action.

One example is **function keys**. For instance we may want F2 to mean *Reset Criteria* no matter where in the form the cursor is. For the Find Guest window it can be done in this way:

- On frmFindStay, open the Form-property box, select the Event tab and set *Key Preview* to Yes.

- Define an event procedure for the Form's KeyDown event. It should look like this:

```
Private Sub Form_KeyDown (KeyCode As Integer, _
    Shift As Integer)
    If KeyCode = vbKeyF2 Then
        Me.subStayList. Form. RecordSource = _
        "select * from qryStayList; "
        Me.txtName = ""
    End If
End Sub
```

The *KeyDown* event has two parameters. *KeyCode* (an integer) shows which key is pressed. There are predefined constants for the various keys. For instance vbKeyF2 means the F2 key, vbKeyA the A-key and vbKeyEscape the Esc key.

If the procedure returns with KeyCode = 0, Access stops further processing of the KeyDown event.

Shift (an integer) shows whether some of the shift keys were down at the same time (Shift, Alt, Ctrl). There are predefined constants for the shift keys: acShiftMask (=1), acCtrlMask (=2), acAltMask (=4). The program can for instance test whether Shift and Ctrl are down at the same time with this statement

```
If Shift = acShiftMask + acCtrlMask Then . . .
```

In the example above, the procedure tests whether the F2 key was used. In this case it sets record source to the full list of stays, and it sets the search criterion to an empty text.

Parent form

If you try it, you may notice that F2 works okay if the focus is in the main form, but not when it is in the subform. You have to define key preview actions also for the subform if F2 is to work here too. However, from the subform you have to address the items in the main form slightly differently:

```
' In the subform:
Me. Parent. subStayList. Form. RecordSource = _
    "select * from qryStayList; "
Me. Parent. txtName = ""
```

Me refers to the subform, and *Me.Parent* refers to the parent form, i.e. the master form.

Return value

At return from the preview procedure, Access continues processing the event. It sends it to the control in focus, or makes its own response to the event. In case of F2 the control ignores F2, but Access has a standard response: it toggles between selecting the entire field and just showing the simple cursor.

The preview procedure can skip further processing by setting KeyCode=0 at return. In case of F2, the result would be that Access didn't toggle between field selection and the simple cursor.

5.5.10 Error preview

Access detects various kinds of user errors, for instance that the user enters a non-existing date, or the user forgets to select a related record where referential integrity is required. Access will show an error message that often is entirely gibberish to the user. When for instance the referential integrity is violated, Access will say:

You cannot add or change a record because a related record is required in table 'tblGuest'.

How can the program show a meaningful message instead? The *BeforeUpdate* procedure is intended to let the program check the data, but when Access detects the error it doesn't even call *BeforeUpdate.*

However, the Form has an *Error* event procedure that can interfere before Access shows the error message to the user. To use it, define the event procedure for the Error event:

```
Private Sub Form_Error(DataErr As Integer, _
   Response As Integer)
   If DataErr = 3201 Then
      MsgBox "Select a guest", vbOKOnly + vbExclamation
      Response = acDataErrContinue
   End If
End Sub
```

The Error event has two parameters. *DataErr* is an ID for the error. *Response* tells Access what to do at return. When Response = acDataErrContinue, Access will not show its own error message. When Response =

acDataErrDisplay (the default), Access will show its own error message.

In the example, the event handler tests whether the error is "You cannot add or change . . ." (ID 3201). In this case it shows a message box with a more user-oriented text. Then it tells Access to keep its mouth shut about the error.

At return, Access keeps the focus on the Form where the error occurred, so that the user can correct the data - or use Esc to undo changes. This happens no matter whether *Response* is acDataErrContinue or not.

The main problem when using the Error event is to find out what the error ID means. I have not been able to find a list of the codes.

Don't confuse the Error event with VBA's handling of program errors. The Error event handles errors caused by the user's actions. VBA handles cases where the program tries to divide by zero, access a non-existing object, etc. VBA has statements such as *On Error GoTo x* to let the programmer deal with the error. See section 6.1 for a detailed explanation of this kind of error handling.

5.5.11 Timer and loop breaking

A Form responds to user actions such as editing a field or clicking a button. Sometimes we want the Form to act on its own. It might for instance update a list of data every minute. This would be useful in a multi-user application where several people update the database.

Timer handling

Start the timer by setting the Form's TimerInterval, for instance in the Form's Load event procedure:

```
Me.TimerInterval = 60 000 ' One minute is 60 000 ms.
```

Next define an event procedure for the Timer event:

```
Private Sub Form_Timer()
   Me.Requery
End Sub
```

When the Form opens, it sets the timer interval to one minute and waits for some event. The event may be the user doing something or one minute having passed. When the minute has passed, Access calls the Timer event procedure, which recalculates all bound data in the form.

The next timer event will occur one minute after the first timer event, even if the recalculation has taken several seconds. How would the program stop the timer events? Set the timer interval to 0:

```
Me.TimerInterval = 0 ' Stop the timer.
```

Let user break a loop - SendKeys

Assume the user should be allowed to stop the periodic updating. It is easy. Provide a button labeled *Stop*. Let its click procedure set the timer interval to 0.

This works because when the Form waits for the timer, it also responds to user events. It will handle the events in the sequence they occur.

Now imagine that the Form runs a long calculation or scans the Internet for hours. It runs in a loop and it doesn't wait for events at any point. How can we allow the user to break the loop? If we provide a *Stop* button as above, the user may click it, but the Form doesn't listen to events. However, Access stores the event in a queue for later processing.

The user could use Ctrl+Break. It will in a very rough way stop the program, probably in the middle of something, so that data is left in an inconsistent state. Actually, Ctrl+Break is the only way you can stop a very long-lasting database query. In this case the database engine takes care that the database ends up in a consistent state. For a long calculation in our own program, we need a structured way to stop.

The trick is to let the program use *SendKeys* every now and then to generate an event and wait for the event to be processed. This allows Access to process all events in the queue. It also allows Access to update the screen, for instance to show controls that the program has changed during the calculation.

Here is an outline of such a program. We declare a module-level Boolean in the Form, which becomes true when the user clicks the Stop button:

```
Dim bStop As Boolean
```

When Access responds to the click, this event procedure sets bStop:

```
Private Sub cmdStop_Click()
   bStop = True
End Sub
```

A long computation runs in a loop until the computation is finished. Once in the loop it displays the progress of the computation by storing a result in a text box control. It could look like this:

```
Private Sub cmdCompute_Click() ' Plain loop
   While . . . ' Until calculation is finished
      . . . ' One step in a long calculation
      txt . . . = . . . ' Display progress
   Wend
End Sub
```

In order to respond to the stop, we modify the loop so that it runs until bStop is True or the computation is finished. In addition we use the *SendKey* operation to send a character to Access and wait for Access to

process it - and process other events too. Here is the new program:

```
Private Sub cmdCompute_Click() ' Test once in loop
   bStop = False
   While Not bStop And . . . ' Until stop or calculation finished
      . . . ' One step in a long calculation
      txt . . . = . . . ' Display progress
      SendKeys "{home}", True ' Type a character and wait.
      ' Allows Access to handle all events and
      ' update the screen.
   Wend
End Sub
```

SendKeys has two parameters. The first is a string of characters. They are sent to Access as if the user typed them. (Or rather, to the program that has the focus on the screen. You might for instance send characters to MS Word in this way.) The second is a Boolean. If it is True, SendKeys waits for Access to process all pending events and update the screen. The string can hold several characters, e.g. "abc{home}+-". Special characters are shown as a mnemonic in { }. We have chosen {home} since it is a rather harmless character in the user dialog.

SendKeys takes around 1 ms on a plain computer. In the IT world this is quite slow. If the calculation step for instance takes 0.01 ms, the program suddenly becomes 100 times slower.

One solution is to let the program listen and update the screen only once a second. In the procedure below, we use the *Time()* function to find out when a second has passed. It returns the number of seconds since midnight, with fractional seconds too. The *Time* function itself takes around 0.0005 ms, so it will rarely slow down the loop significantly.

```
Private Sub cmdCompute_Click() ' Test once a second
Dim startTime As Double
   bStop = False
   startTime = Timer() +1
   While Not bStop And . . . ' Until stop or calculation finished
      . . . ' One step in a long calculation
      If Timer() > startTime Then ' A second has passed
         txt . . .  = . . . ' Display progress
         startTime = startTime + 1 ' Next in one second
         SendKeys "{home}", True ' "Type" a character and wait.
         ' Allows Access to handle all events and
         ' update the screen.
      End If
   Wend
End Sub
```

5.5.12 Multiple form instances

When we open a form with DoCmd.OpenForm, we get only one instance of the form. If we try to open it again, nothing happens.

In order to open multiple instances, we need to use the basic VBA mechanism of creating an object. Here is a program piece that opens two instances of frmStay:

```
Dim stay1 As Form, stay2 As Form
' References to open forms
. . .
Set stay1 = New Form_frmStay
Call stay1.Init(740)
stay1.Visible = True

Set stay2 = New Form_frmStay
Call stay2.Init(753)
stay2.Visible = True
```

The first line declares two references to an open form, stay1 and stay2. The declarations may be in a global module that remains open as long as Access is open.

In an event procedure somewhere else, the statement `Set stay1 = . . .` creates a new object of the class `Form_frmStay`. The reference stay1 will now refer to this object. During the creation, the form object receives the usual open-events: *Open, Load, Activate*. The form is still invisible.

When we open a form in this way, we cannot give it OpenArgs like those we use with DoCmd.OpenForm. The form has a property called *OpenArgs*, but it is ReadOnly so we cannot store anything in it. For this reason we have written a procedure for initializing the form, the procedure *Init*. We call it with a parameter that tells it to open stay 740. The procedure will set the filter of the form to show this stay.

Finally, we set the *Visible* attribute of the form to True. This generates the *Current* event and makes the form visible.

Then we repeat the whole thing using stay2 instead of stay1. The result is that one more form instance opens, this time showing stay 753.

Closing the forms

Usually the form closes itself, for instance when the user clicks the close button. Access automatically sets references to it to *Nothing*. We may also close the form by setting references to it to *Nothing*, like this:

```
Set stay2 = Nothing
```

The reason we have stay1 and stay2 in a global module is that they have to be there all the time. If we made them local variables in a procedure, the opened forms would close as soon as the procedure returned. See more about variables and their life-time in section 6.2.

Handling many open forms

When we need many open instances, we have to make an array of references. A click event could then open a new instance in this way:

```
Dim stay(1 To 10)
. . .
Private Sub cmdOpenStay_Click( )
     ' Find an unused reference, let it be j
     Set stay(j) = New Form_frmStay
     Call stay(j).Init( . . . )
     stay(j).Visible = True
End Sub
```

5.5.13 Resize

When the user drags the border of a form, Access calls the Resize event procedure of the Form. The procedure is called for every few pixels the user drags, depending on how fast he drags. Access also calls the Resize procedure once during open.

If the program doesn't do anything at these events, the user will just see a larger or smaller gray area. Usually the user expects that the controls somehow adjust to the new size, for instance that a subform area expands or contracts.

The way to handle this is to let the Resize procedure recompute sizes and positions for the controls. Figure 5.5F shows a simple example. The form has a couple of controls, one of them being a text box (txtTest) and another a subform (subTest). As the user drags the right border, these controls adjust their width so that they keep the same distance from the right border of the form. Care must be taken when a control becomes so narrow that it cannot keep its distance from the border. Trying to set its width to a negative number will cause a run-time error - and the mouse gets trapped inside Access! On the lower version of the form, the text box has width zero, and as a result the controls don't adjust their width anymore.

At the time Resize is called, the situation is as follows:

me.WindowWidth: The new width of the form, including the border.
me.WindowHeight: The new height of the form, including the border.
me.WindowTop: The distance between the form and the top of the surrounding Access window. (**Access 2003** only).
me.WindowLeft: The distance between the form and the left side of the surrounding Access window. (**Access 2003** only).

The figure shows the Resize procedure. It declares two important variables:

formWidth: The previous width of the form. When the form opens, the variable has its default value, zero. It is declared as *Static*, meaning that it survives from one call of Resize to the next.
dif: The difference to add to the control widths. Initially, when the form opens, *dif* becomes zero. Later, it is the difference between the new form width and the previous form width.

The procedure computes the new width of txtTest, the smallest of the controls. When the new width is larger than zero, it adjusts the widths of txtTest and subTest. It also saves the new form width in order to calculate future changes relative to this.

For subforms it may be necessary to change the width not only of the subform control, but also of the subform itself. See below.

Size unit

All positions and sizes are computed in **twips**.

One *twip* is 1/20 typographical point =
1/567 cm =
1/1440 inch

Resizing a form

It is easy to change the width of a control, so it is tempting to change the width of a form in a similar way:

```
me.WindowWidth = me.WindowWidth + dif
```

However, this works in none of the Access versions. You have to use this statement to change position or size of a form:

```
DoCmd.MoveSize right, down, width, height
```

right: Distance between form and the left side of the surrounding Access window. (The same as the property WindowLeft.)
down: Distance between form and the top of the surrounding Access window. (The same as the property WindowTop.)
width: The width of the form including the frame. (The same as the property WindowWidth.)
height: The height of the form including the frame. (The same as the property WindowHeight.)

In order to increase the width of the form by *dif*, you have to set the form in focus and use

```
DoCmd.MoveSize  , , oldWidth + dif
```

When you do this, the form receives a Resize event.

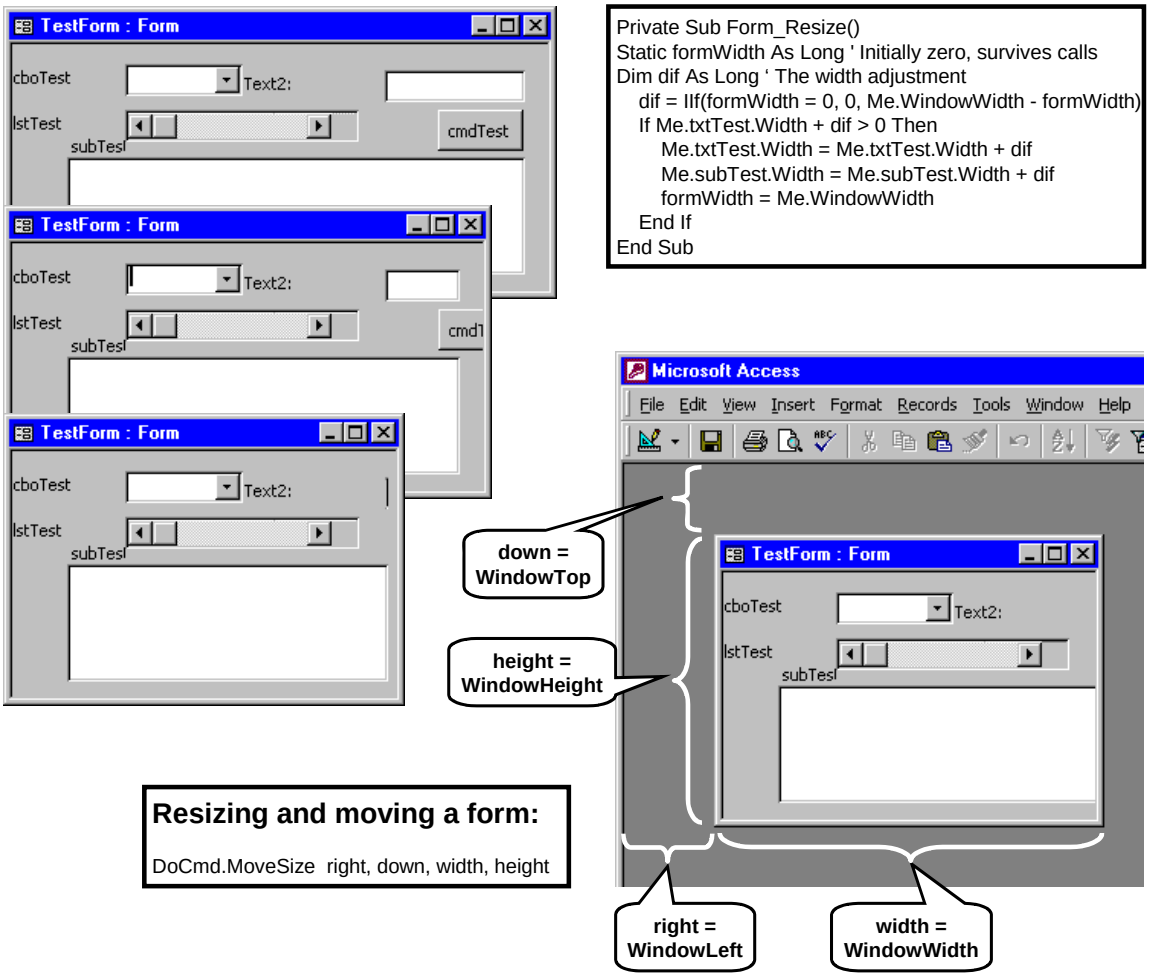
Resizing a subform

Resizing with DoCmd works fine for a main form, but not for a subform. You cannot bring the subform in focus. However, it doesn't matter because an open subform automatically has a width that matches the subform control.

When resizing the subform control, you just have to make the program resize and/or reposition the controls on the open subform too. You can do this from the Resize procedure in the main form. It soon becomes messy to let the main form know about the controls inside the subform. So a better approach is to call a procedure in the subform that resizes its own controls, in much the same way as its resize procedure would do it.

When the subform is shown as a datasheet, it has no effect to adjust widths and heights of the controls. In-

Fig 5.5F Resizing a form




```
Private Sub Form_Resize()
Static formWidth As Long ' Initially zero, survives calls
Dim dif As Long ' The width adjustment
   dif = IIf(formWidth = 0, 0, Me.WindowWidth - formWidth)
   If Me.txtTest.Width + dif > 0 Then
      Me.txtTest.Width = Me.txtTest.Width + dif
      Me.subTest.Width = Me.subTest.Width + dif
      formWidth = Me.WindowWidth
   End If
End Sub
```

stead of setting the width, set the *ColumnWidth* properties of the controls (see section 5.5.7). You might for instance extend all the columns in a proportional way.

5.6 Record sets (DAO)

When we need complex record handling, there are two ways to go: through SQL statements that update records, or through programmed record access. In this section we look at the latter possibility. The SQL way is explained in section 7.1)

Select the database model. Before you can try the following example, you should make sure that your program accesses the database with the right method. Over the years, Access has used different methods to access the tables, trying at all times to be compatible with earlier versions. Below we will use the DAO approach which works across all Access versions. Access 97 is born with DAO, but in Access 2000 and 2003 you have to select DAO. Do as follows:

1. Open Visual Basic and select Tools -> References. You now see a list of the libraries that Access may use. At the top of the list, you see those selected at present. When VBA looks for a built-in class name, it first looks in the top library. If the name is not there, it looks in the next library, and so on.

2. If *MicroSoft DAO 3.6 Object Library* is not selected, go far down the list and select it. Next move it to the top of the list. The easiest way is to close the list, then reopen it. Now DAO 3.6 is in the top list. Move it further up as far as it can go. Then VBA will find the DAO 3.6 names first.

There is no reason to restart the system. You may notice that in the Object Browser and with Ctrl+J you will now see two RecordSet classes. Use the one with an Edit property.

5.6.1 Programmed record updates

As the first example we will outline how the CheckIn button on the Stay form could work (Figure 5.6A).

3. Open frmStay in design view. (If you followed the earlier exercises closely, there are no buttons on the form.)

4. Create a Book button and a CheckIn button and give them names with the *cmd* prefix.

5. Define the event procedure for the OnClick event on the CheckIn button. Figure 5.6A shows the body of the procedure. It demonstrates many new things that we explain below.

Declarations. The first lines of the procedure declare two variables. The variable *s* can hold a text string. The variable *rs* can hold a *Recordset*, or more precisely, it holds a reference (a pointer) to a Recordset. These variables are local for the procedure, which means that VBA creates them when it calls the procedure, and deletes them when it returns from the procedure. Initially they both have the value *Nothing*.

Computed SQL

The statement s="SELECT . . . " computes a text and stores it in the variable s. If the form shows stay 728, then s will hold this text:

```
SELECT * FROM tblRoomState
WHERE stayID=728;
```

It is an SQL statement that selects all fields from the RoomState records that belong to the stay. As you see, the program computes this SQL statement from three parts, the text "SELECT . . . ", the expression *Me.stayID* and the text ";".

If you are not fluent in SQL, it may be easier to make the query with the query grid. Then switch to SQL-view and copy and paste the statement into the program. VBA makes a lot of noise about this non-VBA statement, but just modify the statement with quotation marks, &-operators, and so on. As usual, don't care about capitalization. SELECT may for instance be written with small letters. VBA doesn't look at what is inside the quotes and the SQL-engine doesn't care about caps.

Warning. You may wonder why we have to compute the SQL. When working with the query grid, we could write things like

```
WHERE stayID=Forms!frmStay!stayID
```

Why don't we write something similar here, for instance

```
WHERE stayID=Me.stayID
```

The answer is that in VBA we compose the final SQL-statements directly. When working with the query grid, Access translates our SQL-statement into the final SQL version. As part of this, Access finds the current value of Forms!frmStay!stayID and inserts it into the SQL-string that it passes to the database engine. In other words, it does the same kind of work that our VBA program does.

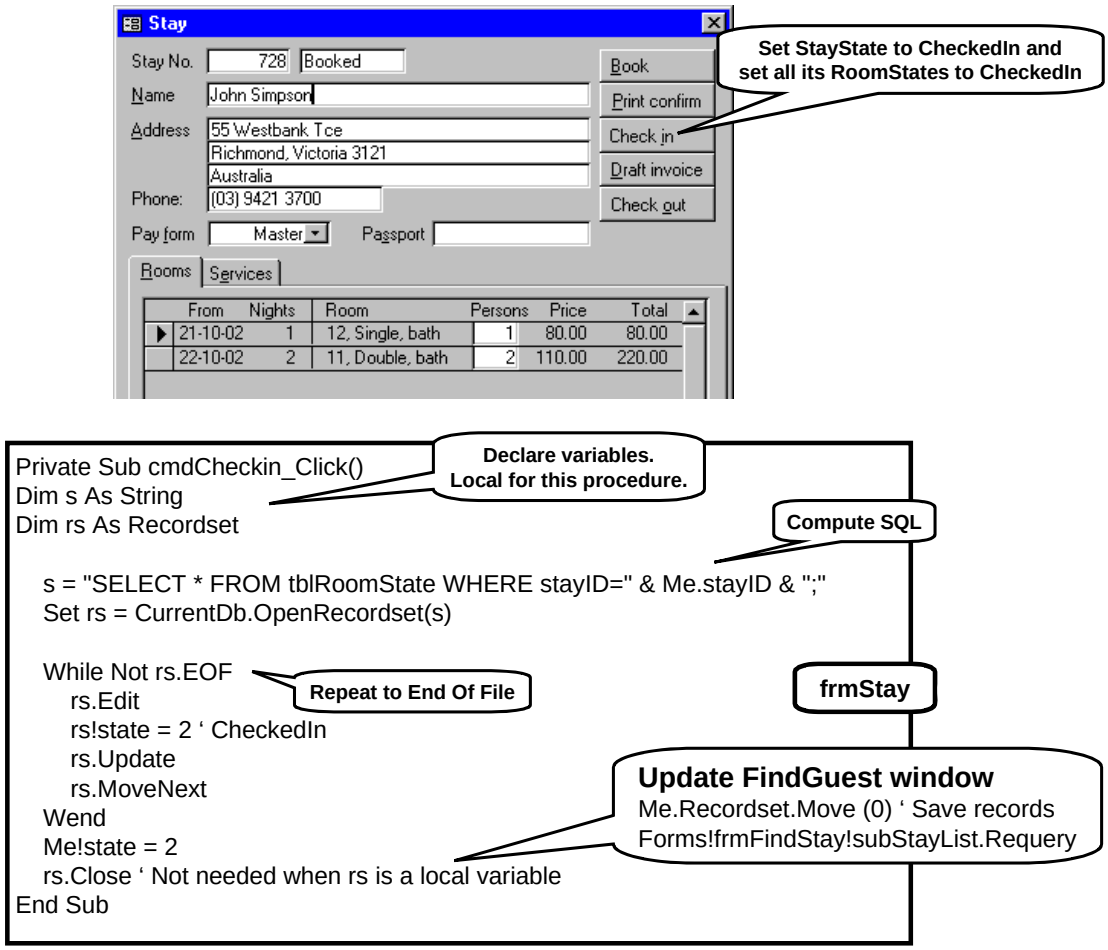
Open the Recordset. The statement

```
Set rs = CurrentDB.OpenRecordset(s)
```

creates a Recordset and stores a reference to it in *rs*. The command *Set* says that we want to store the reference, not the Recordset itself. *CurrentDB* is the database currently used by the program, and we ask it to open a record set that gives us the records specified by the SQL statement in s. When Access has opened the record set, the first record in the set becomes the current record. If the set is empty, there is no current record, and End Of File (EOF) is true.

While loop. The statements from While to Wend are a loop. The four statements inside the loop are repeated until *rs.EOF* becomes True. This happens when the program tries to move the current record beyond the

Fig 5.6A Recordset, updating records



end of the set. If the record set is empty, EOF is True from the beginning and the loop terminates immediately.

Updating a record. The first statement inside the loop starts editing the current record. VBA transfers the fields to an edit buffer. If we don't transfer the fields to the edit buffer, the program can read the fields but not change them.

The next statement changes the state of the RoomState record to 2, meaning CheckedIn. The change takes place in the edit buffer - not in the database table. Notice the bang-operator that ensures that we get the field, not a possible built-in property.

The *rs.Update* statement transfers the changed fields to the database table. At this time Access checks that mandatory fields are filled in, that referential integrity is okay, etc. There are several ways the program can catch these errors so that the user doesn't see the cryptic messages produced by Access. See sections 5.5.10 and 6.1.

Move to next record. The *rs.MoveNext* statement moves to the next record, that now becomes the current record. If there is no next record, EOF becomes True and the loop will terminate. When EOF is true, there is no current record in the record set, and attempts to address fields in it will fail.

Updating the stay record. When the loop is finished, the program sets *Me!state* to 2, thus updating the stay record as well.

Close the Recordset. The last statement closes the record set. It is customary to do so, but in this case it is unnecessary. When the procedure returns, VBA will delete the local variables. Since *rs* contains a reference to a record set, it will close it before deleting *rs*.

Try the program

6. Close the program and try out the CheckIn button. You should see the state field change in the stay window.

7. To ease experimentation with the system, program the Book button so that it does exactly the same as

CheckIn, but sets the states to one, meaning *Booked*.

Update the stay list. You may notice that the FindGuest window doesn't update its own stay list automatically. In this list, the stay doesn't seem to change to CheckedIn, etc. It is tempting to repair it by letting CheckIn make a requery on the stay list. Try it:

8. Insert this statement at the end of the CheckIn and Book procedures:
   ```
   Forms!frmFindStay!subStayList.Requery
   ```

9. Try to book and check in while watching the stay list. Nothing changes in the list. Why?

Because the stay state is not yet saved in the table. Saving doesn't happen until the user closes the stay window, but at that time the Requery has been performed already.

One way to deal with the requery is to do it in the AfterUpdate procedure for the stay form. This event is intended for things to be done after the update, i.e. after saving the stay and guest records from the query. However, in our user dialog we don't want the user to close the stay window all the time. He might want to change other things for the stay. Here is a tricky way to update the stay record before doing the requery:

10. Insert this magical statement in CheckIn and Book just before the Requery:
    ```
    Me.Recordset.Move(0)
    ```

This statement works on the open recordset bound to the form. It moves current record back or forth a number of records, and as part of this it saves the current record. In our case we move zero records away - to the same record. But we got the update anyway. (See more in section 5.6.3.)

11. Try to book and check in while watching the stay list. It should now update automatically.

Notice that this version of CheckIn and Book is very experimental. In the real system we have to check many things. For instance we shouldn't be able to book someone if he hasn't any rooms. The action of Book should also depend on the current state of the stay, for instance so that if the user tries to book a stay that is checked in, the system asks him whether he wants to undo the checkin.

5.6.2 How the record set works

Figure 5.6B shows how a record set works. The record set consists of a record list and some properties for navigating in it. The current record is the one the program can access with *rs!fieldX*. The operations

```
rs.MoveNext and rs.MovePrevious
```

move *current* one record ahead or back. There are other Move operations too:

```
rs.Move(n), rs.MoveLast and rs.MoveFirst
```

The first one moves current n records (forward or backwards depending on the sign of n). The next two move current to the last or first record in the record set.

If the program tries to move beyond the last record, *rs.EOF* becomes True. If it tries to move before the first record, *rs.BOF* becomes True.

In order to edit the current record, we have to move it to the edit buffer with the operation

```
rs.Edit
```

When we have finished editing it, we move it back to the record set with

```
rs.Update
```

If we move to another record without using *rs.Update*, the changes will be lost.

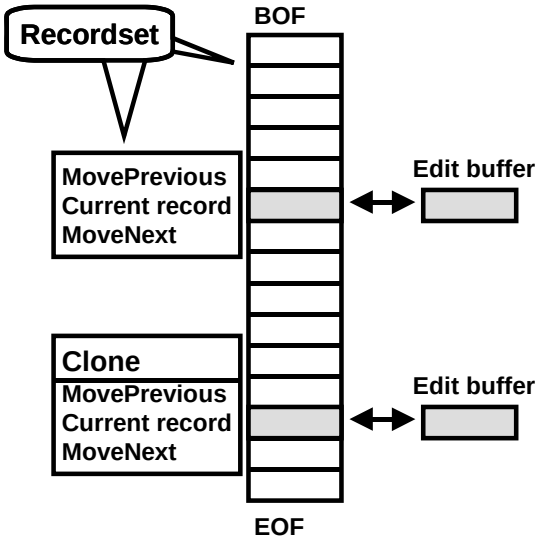
Clone. Figure 5.6B also shows a *Clone object*. It is another set of navigation properties working on the same record list. It has its own current record pointer. By means of clones, the program can access several records in the list at the same time. It may for instance compare two records in the list, one accessed with the record set and one accessed with the clone.

To the right in the figure, you see how a Clone is created and used. You declare the clone *rc* exactly as a record set. However, instead of opening the clone, you ask the record set *rs* to create a clone of itself. You may now use *rc* exactly as the record set itself, for instance moving the current record of *rc* back and forth. We have shown how you could compare the current record of *rc* with the record of *rs*.

Add a record. The figure also shows how to create new records. First we use *rs.AddNew* to fill in the edit buffer with a Null record. Next we fill in the fields we want, and then we use *rs.Update* to create a new record in the record set and transfer the edit buffer to it. Before doing *rs.Update*, Access will check that referential integrity is okay, that mandatory fields are filled in, etc.

Delete a record. Finally the figure shows how to delete a record. We simply use *rs.Delete*. After this there is no current record. When we execute *rs.MoveNext*, the record after the deleted record will be the current record.

Fig 5.6B Recordset, clone, add and delete




```
. . .
Dim rs As Recordset, rc As Recordset
    s = "SELECT * . . . “
    Set rs = CurrentDb.OpenRecordset(s)
    Set rc = rs.Clone

    If rs.roomID = rc.roomID Then . . .

    While . . .
        rs.AddNew
        rs!fieldX = . . .
        rs.Update
        rs.MoveNext
    Wend

    While Not rs.EOF
        rs.Delete
        rs.MoveNext
    Wend
```

Add records

Delete records

5.6.3 The bound record set in a Form

Above, the program has explicitly created a record set. However, when we open a form such as frmStay it becomes bound to a record set. Can our program reference this record set explicitly? Yes, it can.

Figure 5.6C shows how it works. We have opened frmStay directly from the database window, and although the form shows only one stay at a time - the current record, we can scan through all the stays with PageDown. Behind the form is a record set with all the stays. We can access this record set with *Me.Recordset*. Furthermore, Access offers a clone of this record set, called *Me.RecordsetClone.*

Access uses *Me.Recordset* in much the same way as our program would do with a record set. There are small differences, however. When the user for instance types something into a field, Access doesn't use the edit buffer, but a hidden buffer of its own. (This hidden buffer deals with the Value and Text properties of text controls.) If our program tries to do something with *Me.Recordset,* Access will first save the hidden buffer and call BeforeUpdate and AfterUpdate for the changed record.

Try the mechanisms at work (also shown on Figure 5.6C):

1. Set a breakpoint in the beginning of the CheckIn procedure on frmStay.

2. Open frmStay in form view and also open tblGuest to see what happens.

3. Change a letter in the name field on the form. Then click CheckIn. You will now be at the breakpoint in the CheckIn procedure. Open the Immediate window with Ctrl+G and adjust window sizes so that you see the guest table, the form, and the Immediate window at the same time.

4. Notice that the name of the guest hasn't changed in the guest table. Now enter this statement in the Immediate window
   ```
   Me.Recordset.Move(0)
   ```

Notice that this changes the guest name in the guest table. What happened? Access was editing the name field through the hidden buffer, but now updated the record in the table to allow *current* to move. This is the trick we used at the end of CheckIn to update the database and then update frmFindStay.

5. Try this statement in the Immediate window
   ```
   Me.Recordset.MoveNext
   ```
   The current record moves one record ahead and the form will show the next stay.

6. **Use the clone.** Try moving the current record of the clone with this statement
   ```
   Me.RecordsetClone.FindFirst("stayID=740")
   ```

This causes the clone to point to the first record with stayID=740 (use another stayID to match your own data). You cannot see any effect of this on the form, because the form is bound to the record set, not to the clone.

7. Now try to move the record set to the same place as the clone. To do this we use the Bookmark concept. *Bookmarks* are built-in, unique identifications of the records in the record set. They are generated when the recordset is opened and don't survive closing the set. The property *Recordset.Bookmark* is the bookmark of the current record. You can read the bookmark property and set it. Try this:
   ```
   Me.Recordset.Bookmark =
   Me.RecordsetClone.Bookmark
   ```

The form should now show stay 740 (or the stay you have chosen). What happened? VBA read the bookmark for the current record in the clone. Then VBA moved *current* to this bookmark.

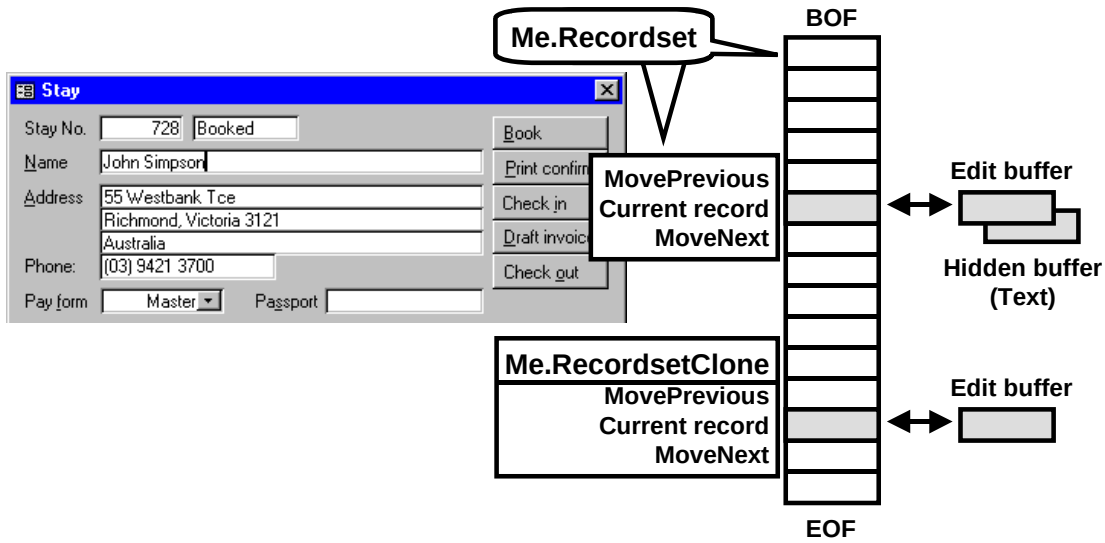
This approach is often useful when we want to let the user move from one record to another, for instance based on a complex criterion. The program first finds the record using the clone, in that way avoiding that the user sees intermediate steps of the search or sees unsuccessful attempts. When the search succeeds, the program moves *current* to the right place.

Me versus Me.Recordset. Notice that we can access a field in the current record in two ways: through the form with *Me.field* and through the record set with *Me.Recordset.field*. There are some important differences between these two approaches.

If the program reads past the last record in the record set, EOF becomes true and there is no current record in the record set. If we try to use *Me.Recordset.field* we will get an error message. However, there is a current record in the *form* - the last record - and we can access it in the program with *Me.field*.

If the record set is empty, EOF is true and there is no current record in the record set. However, there is a Null record in the form, the user sees blank fields, and the program can access them with *Me.field*. The fields are all Null and we cannot store anything in them.

Fig 5.6C Me.Recordset, Me.RecordsetClone




```
Me.Recordset.Move(0)      ' Updates Me-changes.

Me.Recordset.MoveNext   ' Updates Me-changes. Moves to next record.

Me.RecordsetClone.FindFirst("stayID=740")
Me.Recordset.Bookmark=Me.RecordsetClone.Bookmark
```

5.6.4 Record set properties, survey

In this section we give a summary of the record-set properties. There are more properties than those shown, but they are for special, technical use. Section 5.6 is a tutorial introduction to record sets.

AbsolutePosition (attribute, read and write). The position in the record set of the current record. The first record in the set has AbsolutePosition 0, the next AbsolutePosition 1, and so on. Setting AbsolutePosition to x will make record x current. If you - or a concurrent user of the database - insert or delete records, the Absolute positions may change.

AddNew( ). Sets a Null record in the edit buffer. The Update operation will transfer the edit buffer as a new record to the record set.

BOF (attribute, read only). True if the program has tried to move current record before the first record. Also True for empty record sets. When True there is no current record.

Bookmark (attribute, read and write). A unique identification of the current record in the record set. You can set the bookmark property to the bookmark of some other record in the same record set. This will make this record current. The method is advantageous to setting AbsolutePosition because bookmarks don't change when records are inserted or deleted.

Clone( ). Creates a clone object that behaves like a record set but works on the same set of records. It has its own pointer to a current record. Returns a reference to the Clone object. Example:

```
Set cloneRecordset = rs.Clone
```

Close( ). Closes the record set and frees the associated memory.

DateCreated (attribute, read only). The date and time the current record was created. Only available when the record set is based on a table, not on an SQL-query. This means that it must be opened like this:

```
Set rs = currentDB.OpenRecordset ("tblGuest")
```

(See also *OpenRecordset* below.)

Delete( ). Deletes the current record. After Delete there is no current record, but after a MoveNext the record after the deleted record will be current.

Edit( ). Transfers the current record to the edit buffer. Edits can then take place in the edit buffer. The Update operation will transfer the edit buffer to the current record in the record set. Any operation that moves current record (e.g. MoveNext or Find) will cancel what is in the edit buffer.

EOF (attribute, read only). True if the program has tried to move beyond the last record. Also True for empty record sets. When true there is no current record.

FindFirst(criterion As String). Finds the first record in the set that matches the criterion. Makes this record current. The attribute NoMatch is False or True depending on whether a matching record was found or not. The criterion is a text looking like a Where-clause in SQL but without the word Where. For instance "stayID=740".

FindLast(Criterion As String). Similar to FindFirst, but finds the last matching record in the set.

FindNext(Criterion As String). Similar to FindFirst, but searches forwards from the current record.

FindPrevious(Criterion As String). Similar to FindFirst, but searches backwards from the current record.

GetRows(n As Long). Copies n records to an array of variant data. The first record is *current*. Moves *current* forward n records to the first record after the ones copied. If there are less than n records left, GetRows only transfers what is left.

Example: Assume that rs is a record set. The records have 3 fields.

```
Dim A( )
A = rs.GetRows(7)
' A(f, r) is now field f of record r
```

This program piece transfers the next 7 records and sets the range of A to A(0 To 2, 0 To 6). The indexes of A are zero-based and A(0, 3) will thus contain the first field of the fourth record.

LastUpdated (attribute, read only). The date and time the current record was last changed. Only available when the record set is based on a table, not on an SQL-query. This means that it must be opened like this:

```
Set rs = currentDB.OpenRecordset ("tblGuest")
```

(See also *OpenRecordset* below.)

Move(n As Long). Moves *current* n records away. When n>0 the movement is forward, if n<0 backward. Move(0) is useful in bound record sets (Me.Recordset) to make Access store the current record in the database.

MoveFirst( ), MoveLast( ). Moves *current* to the first or last record in the record set.

MoveNext( ) , MovePrevious( ). Moves *current* one record forward or one record backward. If the movement goes beyond the ends of the record set, EOF or BOF become True.

Name (attribute, read only). The SQL query behind the record set. In order to define the SQL-statement, use OpenRecordSet.

NoMatch (attribute, read only). Is False or True depending on whether the previous Find operation found a record or not.

OpenRecordSet(s As String). This is not an operation in the record set but in a database object. Opens a record set and returns a reference to it. The text s may be an SQL-statement, a table name, or a query name. Examples:

```
Dim rs As Recordset
Set rs = currentDB.OpenRecordset ("SELECT * FROM
tblGuest WHERE . . . ; ")
. . .
Set rs = currentDB.OpenRecordset ("tblGuest")
```

RecordCount (attribute, read only). Shows the number of records in the set that are presently loaded by the SQL-engine. After a MoveLast, it will show the total number of records in the set. When the record set has just been opened, it will be zero for an empty set, usually one in other cases. Use EOF as a safe way to determine whether the set is empty.

Requery( ). Re-computes the query behind the record set. Useful if the records behind the query have been changed by other means than this record set.

Update( ). Transfers the edit buffer to the record set, either to the record from which it came (after Edit) or as a new record (after AddNew).

5.7 Modules and menu functions

One way to implement functions is by means of command buttons and the *Click* event. Another important way is through a menu. This section explains how to implement menu functions. Figure 5.7A shows the principle.

For command buttons, we utilize that each Form has a module with code. In this Form module you write the event procedures that responded to clicks and other events. For menus, the situation is slightly different - a menu doesn't belong to any particular Form. Where should we write the procedures that handle clicks in the menu? The answer is to make a simple module - one that doesn't belong to a form. In this module we write the procedures. Next we set the OnAction properties for the menu items so that they will call the procedures. We explain the details below.

5.7.1 Create a menu function

We will show how to implement the menu item *CancelStay*. If you followed the book closely, your hotel system should already have this menu point under the menu heading *Stays*, but it doesn't work yet (see section 3.5.2). Proceed as follows:

1. **Create module**. In the database window, select the Module tab. There is no Wizard here to help you create a module, so click New in the database window heading. You should now be in the VBA editor.

2. Enter the procedure **mniCancelStay** as shown on Figure 5.7A. The central part of the procedure is similar to cmdCheckin (Figure 5.6A), so you may simply copy the check-in event procedure and modify it. (We use the prefix *mni* for menu-item procedures.)

The CancelStay procedure uses several new VBA concepts. Look at the first line:

```
Public Function mniCancelStay() As Integer
```

First of all it is not a Private Sub like the event procedure, but a *Public Function*. It is *public* because it must be accessible from other modules and from the menu mechanisms. It is *function* because it has to return a result. We have arbitrarily specified *As Integer* meaning that it returns an integer value. The only reason to make it a function is that the menu mechanisms insist on it being that way. The result is not used for anything as far as I can tell.

The first part of the procedure body tries to find the stay that was selected when the user clicked *CancelStay*. This statement does it:

```
Stay = Screen.ActiveForm!stayID
```

Screen is the VBA concept of the computer screen. The property *ActiveForm* gives us the form that was in focus. Finally, we use the bang-operator to find the stayID of this form.

However, what happens if there is no open stay window when the user clicks *CancelStay*? VBA would not be able to find stayID and would show strange messages to the user. It would also halt the program. Fortunately, VBA has a mechanism that allows us to catch such errors:

```
On Error GoTo NoStay
```

After this statement, any error will cause the program to continue at the line *NoStay*. In VBA terms this statement has *enabled the error handler* of the procedure. (See more on error handling in section 6.1.)

In the line *NoStay*, the program uses MsgBox to tell the user to open a stay window.

The central part of the procedure is like CheckIn (Figure 5.6A), but sets the states to 4, meaning *Canceled*. We might delete the stay and the connected room states entirely, but for auditing purposes and undo-purposes, we just change the states. By the way, it also allows you to experiment with the procedure easily, since you can cancel a stay, then book it again, etc.

Note that we set the state of the stay itself through this statement:

```
Screen.ActiveForm!state = 4
```

This will always work since the program has checked that a stay form is active.

Finally, we have to let the program return from the procedure without continuing into the error handling, where it would ask the user wrong things. The statement

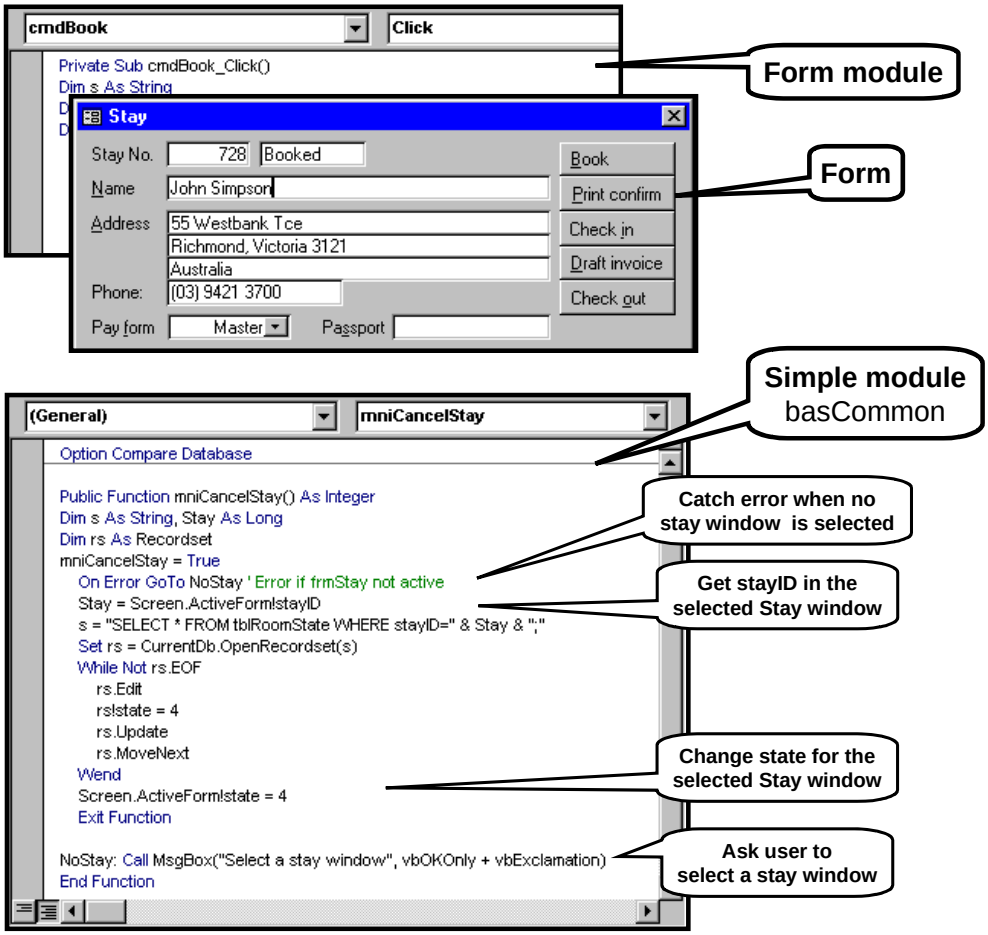
```
Exit Funtion
```

takes care of this.

3. **Save the module**. Use File -> Save (or Ctrl+S). This saves the module, but keeps it open. Give the module the name **basCommon** for *common base module*.

If you just close the VBA window, the window disappears and the module is not visible in the database window. Very scaring! It is not gone, however, just hidden. The VBA window shows it. When you close the database, Access will ask for a name for the module. Section 5.7.3 explains more about creating and naming modules.

Fig 5.7A Modules and menu functions



5.7.2 Define the menu item

We have now written the menu function. It is time to connect it to the menu.

4. Close VBA, right-click the toolbar area and select *Customize*.

5. Roll down the *Stays* menu, right-click *CancelStay*, and select *Properties* (Figure 5.7B). Set the *OnAction* property to:
   =mniCancelStay( )

6. Close the customize boxes and try out the menu: Open a stay through FindGuest, select the stay, and use the menu point *CancelStay*. Unless you are very, very lucky and careful, there will be errors in your mni-procedure. Don't worry - it is normal. Find the errors and repair them.

You may later set the stay back to booked or checked-in with the buttons in the stay window. Also check that the program behaves correctly when you use *Cancel-Stay* without having a stay window in focus.

Menu procedures in the form module

Above, we put the menu procedure in a simple module. The advantage is that we can call the procedure independently of which forms are open or in focus. The disadvantage is that we have to check that the right form is in focus.

In some cases it is more convenient to have the menu procedure in the form module. If you like, you can make an experiment with how to do it.

- Add another menu item to the *Stays* menu. Call it *CancelLocal* as on Figure 5.7B. Set its OnAction property to
  =mniCancelLocal( )

- Open the VBA module for *frmStay* and insert a function that looks like *mniCancelStay*. However, it should have the name **mniCancelLocal**. Further, it should not use Screen.ActiveForm but *Me* instead. The reason is that this function will be called in the context of the form module, meaning that the controls are always available. As a result, you don't need all the error handling stuff.

- Try out the new menu item. It should work correctly when used from a stay window, but gives Access-language error messages when used from other forms. Actually, Access cannot even find the function if the form is not in focus.

This way of calling a menu function is particularly suited when the menu is in a toolbar that is specific for the form. You may remember (section 3.5.3) that toolbars may be attached to a form in such a way that the toolbar is only shown when this form is in focus. In this way the menu functions on the toolbar are always available when the user can click on the menu item.

5.7.3 Managing modules and class modules

You can create a module through the database window's Module tab, but the usual way is to do it through the VBA editor. However, things work in a strange way here. Figure 5.7C shows how to manage.

- To create a module, right click an item in the Project Explorer window. Select *Insert* and either *Module* or *Class Module*.

You can now edit the module in the code window.

- To name or rename a module, select it and use the property icon on the tool bar. (You cannot right click to change it.) Edit the name in the property window.

- To delete a module, select it, right click and use *Remove* . . .

Class module

A class module corresponds to a class in other object-oriented languages. It has procedures and declares variables. You can create multiple objects based on the class, each with their own variables. The only difference between form modules and class modules is that the latter are not visible to the user and have no controls.

In order to create an object of class *claGuest*, declare a reference to it and create it with *New*. Address public variables and procedures in the object with the dot notation:

```
Dim guest As claGuest
' References to a Guest object
. . .
Set guest = New claGuest
guest.address = . . .
guest.SendLetter(" . . . ")
```

This is similar to creating multiple open forms (section 5.5.12). **Beware:** the claGuest objects are just for illustration. They exist only in memory. They are not stored in the database and they have nothing to do with the guest records in the database.

Module (simple)

A simple module is similar to a class but there is only one object based on the module. The system creates this object automatically. In order to address a public procedure or variable in the module *basCommon*, use the dot notation:

```
basCommon.simDate = . . .
d = basCommon.getDate()
```

Fig 5.7B Action for menu items

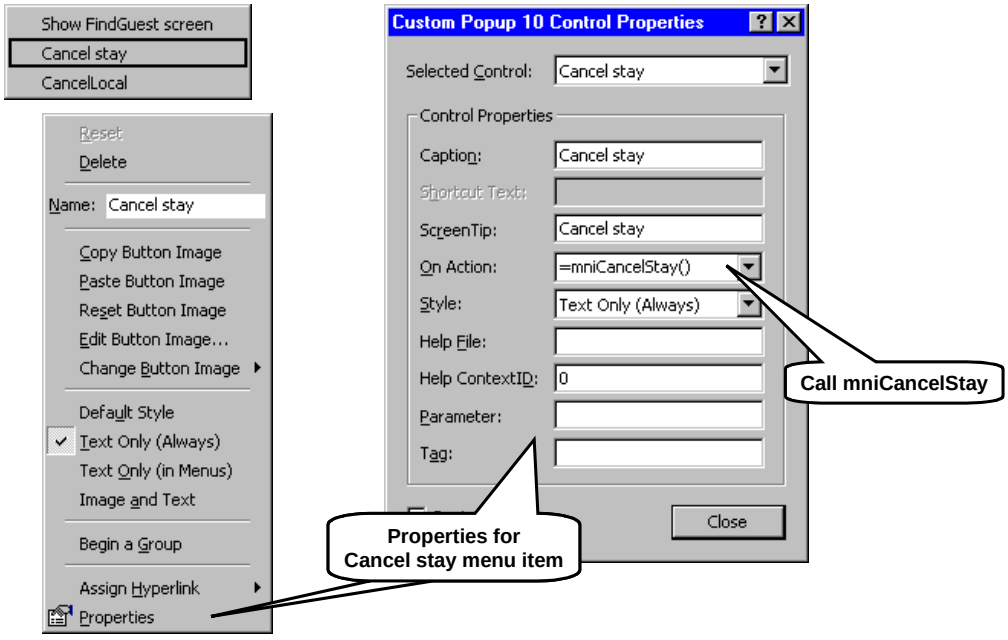
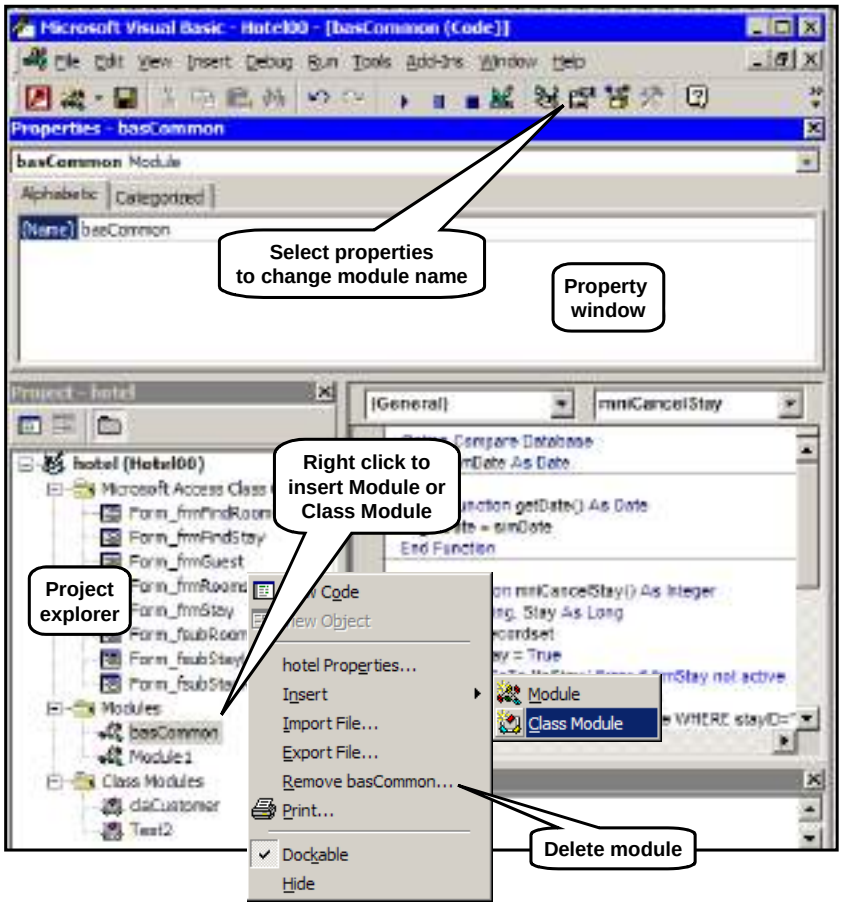


Fig 5.7C Managing modules and class modules



5.7.4 Global variables

The only persistent data in our application is the data in the database. Until now the only dialog data has been some data in the forms, for instance the value of the unbound controls. This data disappears when the form closes, so we need some way to handle dialog data that lives across the entire dialog with the user. Modules is the solution to the problem because a module is open and holds data as long as the application is running.

As a simple example, we will see how to handle a simulated today's date. In the real system, we need to show today's date in many places, for instance as the default search criterion for arrival date. It is easy to do by means of the built-in function *Date( )* which always gives us today's date (or rather the current date setting in the computer).

However, when testing the system, our test data is planned for some specific dates. We cannot change the test data every day we need it. We might instead change the current date setting of the system to the simulated date, but this is not recommended because it has side effects on other things, for instance the date of files created or changed, which again may create havoc in automatic backup procedures.

So let us create a simulated date and keep it in the *basCommon* module. Figure 5.7D shows the solution.

1. Open the *basCommon* module and enter this line at the top
   ```
   Public simDate As Date
   ```
   Also create the public function *getDate( )* as shown.

The variable *simDate* is a variable in the simple module and lives as long as the application is open. The function *getDate* simply retrieves the simulated date and returns it as its own value.

2. Open the module for *frmFindStay* and create the load procedure as shown.

This is the place where the program will set the simulated day to be used. The first thing we will do in the hotel system is to open frmFindStay, and at that moment the Load procedure will be executed. It will set the simulated date to the 23rd October 2002, the date used in several pictures in the User Interface Design book. Note how a date is written inside VBA. The format is a US date, independent of any regional settings in Microsoft Windows. Note also how we address *simDate* through *basCommon*. We might omit *basCommon*, but if we have many modules, we resolve any name ambiguity in the way written. It also allows the VBA editor to guide us in selecting the right element in *basCommon*.

3. Look at the property box for the arrival date. Define the default value as this expression:
   ```
   =getData( )
   ```

4. Open frmFindStay and check that the default value for the arrival date is correct. The user can of course change the arrival date to what he likes.

The arrival date is a combo box and the real system provides a list of dates around today for the user to choose. The default day is still important, however.

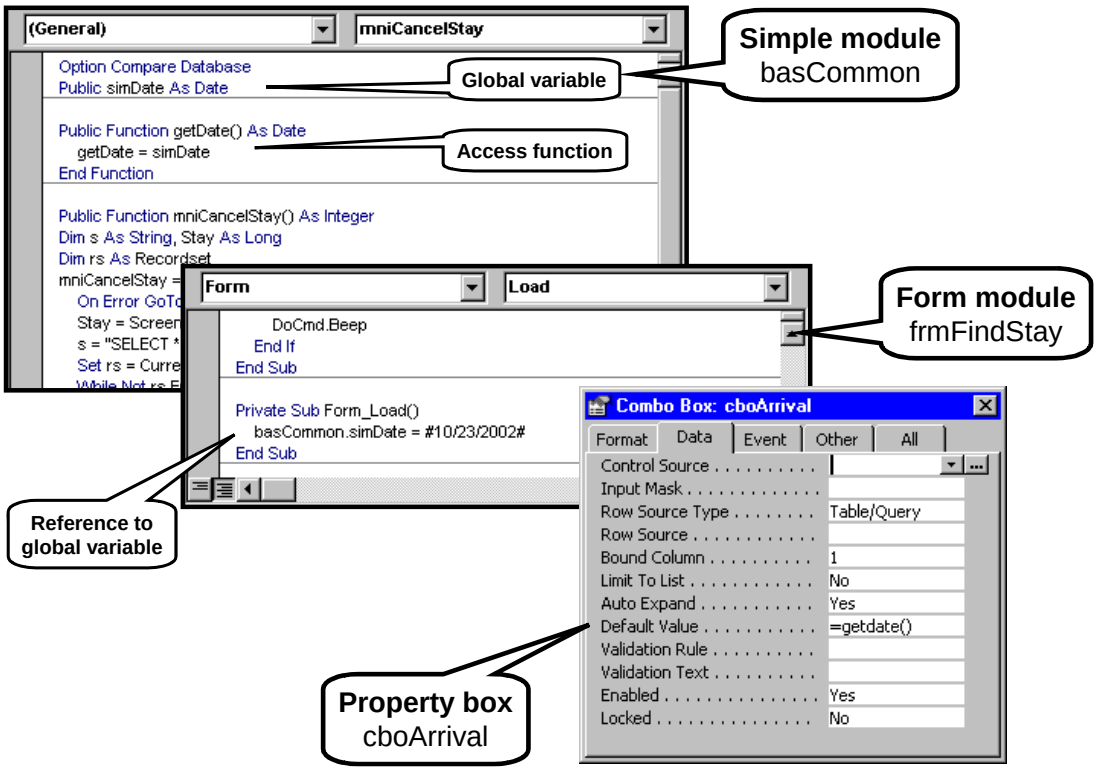
You may wonder why we write *basCommon.simDate* in the Load-procedure and =getDate() in the property. The answer is that in properties we can only use public *functions*, not public variables. Furthermore we cannot use the *basCommon* prefix. The same rule applied when we specified the OnAction property of the menu item.

Once our system is tested and ready for use, how do we let it use the real date? One simple way is to keep everything and just change the *getDate* function to

```
Public Function getDate( ) As Date
   getDate = Date( )
End Function
```

The system will work exactly as before except that it gets the real *today* instead of the simulated one.

Fig 5.7D Global variables



6. Visual Basic reference

In this chapter we give an overview of VBA, the Visual Basic language for Applications. We assume that you know something about programming already. We also assume that you have looked at some of the Visual Basic examples in the booklet so that you have seen program pieces written in this language.

Our discussion is mainly based on the examples shown on the figures. Most of these figures are also available as the **VBA Reference Card**. It may be downloaded from www.itu.dk/people/slauesen.

If you want additional explanation, you have to use the on-line help or experiment on your own. Be prepared that the official documentation (on-line help) is often incomplete or outright wrong, in the sense that the system does something different than described. The examples we show in the figures are based on testing what the system actually does.

Using VBA functions outside VBA. Although we describe VBA below, most of the built-in functions and operators are available also in SQL-statements and in some control properties (e.g. ControlSource). In these places you may for instance use the functions

```
IIF(a, b, c) and
DMin("roomId", "tblRooms", "roomType=2")
```

However, the regional settings may influence the syntax. As an example, with Central European settings, you have to separate parameters with semicolon when working outside VBA. (See more in section 6.6.)

Also notice that when you use the functions from VBA, you get excellent help and excellent error messages, but when using them in SQL or ControlSource, you get little help and very confusing error reactions.

6.1 Statements

Line continuation. A simple VBA statement consists of a line of text. If the statement is too long for a line, you can split it into two or more lines. To do this, you write a space followed by an underscore at the end of the line (Figure 6.1A). You cannot break the line in the middle of a text string. You have to compose a long text from shorter texts joined with the &-operator.

Comment. You can write a comment at the end of the line. It starts with an apostrophe ('). The compiler then ignores the rest of the line. You can only have comments on the last of the continued lines.

Assignment statement. An assignment statement computes a value and stores it in the variable to the left of the =. The **Set** statement is a special version of assignment. It doesn't store a computed value, but a reference to some object. The figure shows how it stores a reference to the first open Form, how it creates a new open form object and stores a reference to it, and how it can set the reference to point at nothing.

Whenever you set a reference, VBA checks whether this overwrites an earlier reference. If so, VBA also checks whether this is the last reference to the object, and if so it deletes the object since nobody can refer to it any more (this is called *garbage collection*).

Conditional statements

Conditional statements are executed when some condition is met. They are examples of *compound statements*, which may consist of more than one simple statement. As Figure 6.1A shows, there are several kinds of conditional statements.

Simple If-Then. The simplest version consists of an If-Then clause followed by a single statement, which is executed when the condition is True. It must all be on one line, possibly broken with line continuations.

If-Then-Else. The more general version consists of an If-Then clause followed by one or more statements, which may be compound themselves. These statements are executed when the condition is True. If the condition is False, the program continues with any ElseIf-Then clauses, each testing their own condition, and passing the control on if the condition is False. If all these conditions are False, the program continues with the statements after any Else clause. The net result is that the statements after at most one of the clauses are executed.

Select-Case is often a more elegant way to choose between statements. In the example, we test the variable *zip*. If zip is 4000 the program executes the statements after Case 4000. If zip is 4001 or between 5000 and 5999, it executes the statements after this clause. And if none of this is True, the program executes any Case-Else statements. Again, the net result is that the statements after at most one of the clauses are executed.

Error handling

On-Error statements switch error trapping on and off inside the current procedure. In order to allow the program to handle errors at all, you have to set an option in VBA:

Access 2000 and 2003:

Tools -> Options -> General ->
Break on Unhandled Errors

Access 97:

Tools -> Options -> Advanced ->
Break on Unhandled Errors

After *On Error Resume Next*, any program error or other unexpected situation just skips the statement where the error occurred. For instance, if the error occurred during an assignment to x, nothing will be assigned to x, so x is left unchanged.

After *On Error GoTo L*, any unexpected situation causes the program to continue at label L. Should further errors occur here, they cause the procedure to return with an error condition to be treated by the calling procedure.

After *On Error GoTo 0* (zero), VBA will handle all unexpected situations, halting the program if necessary. When the procedure returns, the calling procedure will handle all errors according to its own On-Error settings.

When Access detects a program error, it sets an Err object with information about the error. Err has several properties, for instance

Err.Number (or just **Err**): The error ID. Err = 0 means *no error*.
Err.Source: The program that generated the error.
Err.Description: A text describing the error (or giving the error message).

Notice that Access doesn't clear the Err object until the procedure returns. This can be confusing in program patterns where the program tries various things to succeed:

```
On Error Resume Next
. . . Do something that may cause an error
If Err > 0 Then
    . . . Try something else
    If Err > 0 Then . . . Give up
```

Fig 6.1A Visual Basic Statements

Line continuation, comments, assignment

```
i = i+2  ' Comment
s =   "long text A" & _
      "long text B"   ' Comment in last line only
```

```
Set f = Forms(0)          Store a reference
Set f = New Form_frmG     Create object, store ref
Set f = Nothing           Delete object if last ref
```

Conditional statements

```
If a=1 Then c=d+2         ' Single statement
```

```
If a=1 Then
    c=d+2 . . .           ' Multiple statements
ElseIf a=2 Then  } Optional
    c=d/2 . . .
Else              } Optional
    c=0 . . .
End If
```

```
Select Case zip
Case 4000
    type = a . . .
Case 4001, 5000 To 5999
    type = b . . .
Case Else
    type = c . . .        } Optional
End Select
```

```
On Error Resume Next  ' Ignore error
. . . If Err > 0 Then . . .  ' Test for error
```

```
On Error GoTo fail    ' Enable error handler
    . . .
fail: MsgBox( . . . )  ' Continue here at error
```

```
On Error GoTo 0       ' Let VBA handle errors
```

If *Try something else* actually succeeds, Err is still > 0 and the program gives up by mistake. The right pattern is to use

```
Err.Clear or Err = 0
```

just before *Try something else*.

The main problem when using the Err object is to find out what the error ID means. I have not seen a list of the codes. The idea is that each subsystem defines its own error ID's, but this makes it even harder to know the ID's.

Loop statements

Loop statements repeat one or more statements until some condition is met or until an explicit Exit from the loop (Figure 6.1B). The repeated statements may be compound themselves.

While-Wend repeats the statements as long as the While-condition is True. If the condition is False from the beginning, none of the statements in the loop will be executed. It is not possible to break the loop with an Exit-statement.

Do-While-Loop is similar to While-Wend, the only difference being that it is possible to break out of the loop with an Exit Do.

Do-Loop-While is also similar, but the condition is tested at the end of the loop, meaning that the statements will be executed at least once.

For-To-Next updates the loop variable (i in the example) for each trip around the loop. In the example, i was one for the first round through the loop. Then i was increased by 2 before the next trip, and so on. When i >*last*, the loop terminates. If i >*last* from the beginning, the loop is not executed at all. Statements inside the loop may break out of the loop with an Exit For. In this case the value of i is defined. However, if the loop terminates due to i >*last*, it is not defined what the value of i will be. So don't even rely on it being >*last*.

For-Each-Next scans through all objects in a collection. The example shows how to scan through all Forms in the collection of open Forms. The reference variable f will in turn point to each of the objects in the collection. It is possible to break out of the loop with an Exit For.

Fig 6.1B Loop statements

Loops

```
While a<10                ‘ Maybe empty loop
      c=c*2
      . . .               ‘ Exit not allowed
Wend
```

```
Do While a<10             ‘ Maybe empty loop
      c=c*2
      . . . Exit Do       ‘ Exit optional
      . . .
Loop
```

```
Do                        ‘ Loop at least once
      c=c*2
      . . . Exit Do        ‘ Exit optional
      . . .
Loop While a<10
```

```
For i=1 To last Step 2    ‘ Step optional
      c=c*2               ‘ Maybe empty loop
      . . . Exit For       ‘ Exit optional
      . . .
Next i
‘ Don’t trust value of i when loop ends without Exit
```

```
For Each f In Forms       ‘ Scan collection
      call print(f.name  . . . )
      . . . Exit For      ‘ Exit optional
      . . .
Next
```

6.2 Declarations

The ancestor of Visual Basic, the programming language Basic, was designed for teaching programming with a minimum of formalities about the writing. This meant that there was no need to declare variables at all. Basic created the necessary variables as they were needed. This is still possible in Visual Basic, but software developers agree that when developing larger programs it is a huge advantage to declare all variables. You can tell VBA to insist on declarations. Write this specification at the beginning of the module

```
Option Explicit
```

Variant type. Even if you declare a variable, you don't have to specify its type. Without an explicit type, the variable is of type *Variant*. This means that its actual type may change dynamically according to what the program stores into it. It may thus hold a number at one point in time, a text string at another point in time. Apart from the value in the variable, VBA also stores a tag telling what the type is at present.

Simple variables

Declarations of variables usually start with the word *Dim* (for dimension) followed by a list of the variables. Figure 6.2A shows such a list

```
Dim B, C As Byte
```

The result is that variable B is of type Variant and variable C of type Byte. Unfortunately this is counterintuitive and cumbersome. Most people would believe that B as well as C are of type Byte. You have to specify a type for each of the variables to avoid them becoming variants.

Simple variables may be of the types shown on the figure: Byte, Boolean, Integer, etc. We have met most of them already. Byte, Integer and Long hold integers with different range. Single and Double hold floating point numbers with at least 6 significant digits (Single) or 14 significant digits (Double). (See section 2.2 for details.)

Currency. The Currency type is a funny in-between intended for keeping track of monetary amounts. It is a very long integer (64 bits) interpreted as this integer divided by 10,000. This means that the integer 147,000 is the number 14.7 exactly. Why this rule? It is because we can guarantee that amounts are rounded correctly as a bookkeeper would do it. With floating point numbers you have no control over the round-off.

Date. A date value is technically a Double. The integer part is the number of days since 12/30-1899 0:00, the fractional part is the time within the day. As an example, the number 1 corresponds to 12/31-1899 at 0:00, the number 1.75 to 12/31-1899 at 18:00 (6 PM).

Object and Form variables are references to objects, not the objects themselves. You can set the references by means of the Set-assignment and test them by means of the Is-operator.

Variant. You can explicitly declare the variable as Variant, but if you don't specify a type, the variable is Variant anyway. Variants can not only hold the simple values above, but also values such as Null or Empty. Notice that VBA treats all fields in database records as Variant.

Initial values. When a record field is empty, it has the value Null. When a Variant is just created, it has the value Empty, because no memory is allocated for the value. When a String is just created, it holds a text of length 0. When a numerical variable is just created, it holds the value 0.

Strings come in two versions. Strings of variable length and strings of fixed length. The former change their length to match what the program stores in them. The length may be very long, nominally 2 billion, in practice limited by memory space. Strings of fixed length always use space from the left, and fill up the remaining characters with spaces (blanks).

The field types **text** and **memo** correspond to VBA strings of variable length.

Arrays

Arrays can have one or more dimensions. In Figure 6.2A, array c has two dimensions. The first index ranges from zero to 5 (zero is the default), the second from 1 to 6. You may specify the type of the elements of the array. If omitted, they are Variants.

The second array, d, is dynamic, meaning that its dimensions and index ranges can change over time. The program can dynamically change the dimensions and ranges by means of the *Redim* statement, but in general the values stored in the array don't survive this operation. You can use *Redim Preserve* if you only change the range of the last dimension. In that case, the values survive.

You can release the memory occupied by a dynamic array by means of the *Erase* statement.

Type declarations

You can declare types of your own (*user-defined* types). They will typically take the form of a record declaration as shown on the figure. Notice that each field of the record must be specified on a line of its own.

You can only declare types in simple modules, not in Form modules. Once you have declared a type, you can use it for declaring variables and arrays.

Procedures

There are two kinds of procedures: subroutines and functions. The only difference is that a function returns a value. For this reason you can store the result of a

Fig 6.2A Visual Basic declarations

```
Declarations
Dim B, C As Byte          B is Variant, C is 0..255
Boolean     True (<>0, False (=0)
Integer     16 bit, -32,786 .. 32,767
Long        32 bit integer, -2.14E9 .. 2.14E9
Currency    64 bit integer / 10,000
Single      32 bit, -3.4E38 .. 3.4E38, 6 digits
Double      64 bit, -1.8E308 .. 1.8E308, 14 digits
Date        Double, days since 30. Dec 1899 0:00
Object      Reference to any object
Form        Reference to any Form
Variant     Any of the types or Null, Empty,
            Error, Nothing - plus a type tag.
            All database fields are Variant
String      Variable length, max 2E9 characters
String * 50               Fixed length, space filled
-------------------------------------------------------
Initial values            String = "", Boolean =False
Number, Date = 0          Database field = Null
Object = Nothing          Variant = Empty
-------------------------------------------------------
Dim c(5, 1 To 6) As t     Same as c(0..5, 1..6)
Dim d( ) As Single        Dynamic array declaration
ReDim d(5, 1 To 6)        Statement
   Index range (re)defined, data lost
ReDim Preserve d(5, 1 To 8)
   Last index range redefined, data preserved
Erase d     Releases memory for dynamic array
-------------------------------------------------------
Type Customer             ' Simple modules only
   custID As Long
   custName As String * 50
   custAddress As String
End Type
Dim custTable(20) As Customer
```

```
Procedures = Subroutines and Functions
proc a, b, , d            ' Parenthesis-free notation
Call show(a, b, , d)      ' Subroutines only
res = fnc(a, b, , Me)     ' Functions only
-------------------------------------------------------
Sub show(a, b As t, Optional c, d)
    If IsMissing(c) Then . . .
    Exit Sub              ' Optional
    . . .
End Sub
-------------------------------------------------------
Function fnc(a, b As t, Optional c, d As Object) _
    As String             ' As String is optional
    If IsMissing(c) Then . . .
    fnc= result . . .
    Exit Function         ' Exit optional
    . . .
End Function
```

```
Enumeration Type
Public Enum RoomState  ' Visible to all modules
   rmBooked = 1
   rmOccupied = 2
   rmRepair = 3
End Enum
-------------------------------------------------------
Public states(12) As RoomState
. . . states(i) = rmRepair
```

function call as shown in Figure 6.2A, while you have to call a subroutine with the word *Call*. You may call either of them with the parenthesis-free notation as shown on the figure. It means exactly the same, but you cannot store the result in case you call a function this way.

The figure also shows how subroutines and functions are declared. Note how you specify that a specific type of parameter is required, and how you specify that a parameter may be omitted (optional). The procedure can check whether an optional parameter is present with the operator *IsMissing*.

Note how you can use *Me* as a parameter when you call a procedure. Inside the procedure, the parameter must be specified as *Object*, not as *Form* as you might expect.

When control passes to the end of the procedure, it returns to the point it was called. The program can also exit from the procedure with Exit Sub or Exit Function.

Enumeration type - constant declaration

You can define enumeration types as shown on Figure 6.2A. A variable of type RoomState can have the value *rmBooked, rmOccupied* or *rmRepair*.

VBA doesn't restrict the value of the variables to *rmBooked*, etc. The Enum declaration is primarily a structured way of defining the constants *rmBooked*, etc. See section 6.3 for other ways of defining constants.

Module and scope

You create simple modules with the VBA editor (see section 5.7.3). Declare module variables at the top of the module, procedures below (Figure 6.2B). The module variables live as long as the application runs. If they are declared with Public instead of Dim, they are accessible from other modules and from Forms.

You create class modules the same way. Objects of the class are created dynamically (see section 5.7.3). You can address Public variables and procedures in an object through an object reference.

Procedures can declare variables of their own. If declared with Dim, the variable is created at the time the procedure is called and deleted when it returns. However, if it is declared with Static, it survives from one call to the next. Variables declared inside a procedure are never accessible from outside the procedure.

6.3 Constants and addresses

Constants

Figure 6.3 shows the various ways to write constant values in VBA.

Numeric constants can be written in the usual decimal way, in octal (preceded by &o) or in hexadecimal (preceded by &h). Note the scientific notation

-4.9E-20 meaning $-4.9 * 10^{-20}$

Color values consist of 8 bits for the amount of blue, followed by 8 bits for green and 8 bits for red. This is conveniently written in hex, e.g. &h09A0FF. Note that colors on the web (HTML) are similar, but use the opposite sequence, RGB.

String constants are enclosed in quotes. There is no way to write special characters inside the string constant. You have to generate a string with a single special character using Chr(x), where x is the Ascii value of the character. As an example, Chr(10) is a line feed. Next you concatenate these string parts by means of the &-operator. A quote inside a string constant is written as two quotes. When you have to compute SQL-statements in VBA, these statements will often include string constants in quotes. Fortunately, SQL accepts double quotes as well as single quotes. Generate the single quotes-version with VBA to avoid conflicts with VBA double quotes.

Date/time constants are enclosed in # #. The date format between # and # is always US format mm/dd/yy or mm/dd/yyyy. Time may be part of the date/time constant as shown.

Null and **Empty** can be used for testing, for instance

```
If x = Empty Then . . .
If IsNull(x) Then . . .
If x = Null Then ' Always gives Null, never True
```

Notice that comparing with Null always gives Null. You have to use *IsNull* to test whether something is Null. See section 6.4 for more on Null.

You can assign *Null* to a variant variable. You cannot assign an empty string to a record field in a database, you have to assign Null:

```
rs . f = Null    ' Okay, works as an empty string
rs . f = ""      ' Empty string not allowed
rs . f = "  "    ' Okay, spaces as a text
```

Nothing. The value *Nothing* can be used for testing with the Is-operator and for assigning to a reference variable with the Set statement, for instance

```
If x = Nothing Then . . .
Set x = Nothing
```

Constant declaration. You can declare constants, i.e. give them a name. In the example, we have given the constant 10 the name *max* and the constant 24th March 2002 the name *start*. VBA has many predefined constants, for instance vbKeyA to denote the Ascii value of the letter A and vbYes to denote the result of MsgBox when the user has chosen Yes.

Define constants for your project in a simple module, e.g. *basCommon*.

Addressing variables and objects

Figure 6.3 also shows the various ways to address a variable or an object. The first examples address the members of the Forms collection in different ways. The first version uses an integer index as a reference, the second a computed string as the name of the Form, the third a short-hand notation with a fixed string as the name of the form.

The next examples address Form properties and fields from code in the Form itself. The property *Name* is addressed with the dot-operator, while the *name* field is addressed with the bang-operator. In this case there is a name conflict between the two meanings of *name*. If there was no conflict, the dot could also be used to address the field. A property in a subform is addressed with the name of the subform control followed by *Form* to get a reference to the open subform object. From a subform, the main form can be addressed with *Me.Parent*.

Note that *Me* and *Parent* are of type *Object*, not type *Form* as one might expect.

In most cases, you can omit *Me*. The exception is when a built-in function has the same name as the property or control.

Fig 6.2B Module and Scope

```
Module and Scope
Dim a               ‘ Visible in this module only
Public b            ‘ Visible to all modules
-----------------------------------------------------
Private Sub show(p) ‘ Visible in this module only
   Dim c            ‘ Visible in this sub only
   Static d         ‘ Visible in this sub only,
                    ‘ but survives calls
   If . . . Then  . . .
End Sub
-----------------------------------------------------
Public Sub show(p) ‘ Visible to all modules
   Dim c            ‘ Visible in this sub only
   . . .
End Sub
```

Fig 6.3 Visual Basic constants and addresses

```
Constants
23, -23, 0, -4.9E-20        Decimal numbers
&h09A0FF, &o177             Hex and Octal, color: bgr
-----------------------------------------------------
“Letter to:”                Strings
Chr(65), Chr(vbKeyA)        The text “A”
“John” & Chr(10) & “Doe”    Two-line text
“Don’t say “”No”” “         Don’t say “no”
“select * from g where a=‘simpson’ ;”
                  Single quotes are suited for SQL
-----------------------------------------------------
True, False                 Booleans
-----------------------------------------------------
                            Date/time
#10/24/02#                  24th Oct 2002
#10/24/02 14:15:00#         24th Oct 02 at 14:15
#10/24/02 2:15 pm#          24th Oct 02 at 14:15
-----------------------------------------------------
Null, Empty                 Special values
Nothing        Object reference to nothing
-----------------------------------------------------
Constant declaration
Const max=10, start=#3/24/2#
```

```
Addressing
Forms(i)                    Element in collection
Forms(“frmCst” & i)
Forms!frmCst2               Bang-operator

Me.Name, Me!name            Property and Control in
                            this Object (e.g. form)
Me.subLst.Form.name         Property in subform
Me.Parent.txtName           Control in main form

basCommon.simDate           Var in foreign module
c(row, col)                 Indexing an array
custTable(i).custID         Field in array of records
-----------------------------------------------------
With Me.Recordset      Apply before dot and bang
      .addr = .addr & zip
      !name = Null
      !phone = “   “
      .MoveNext
      . . .
End With
```

A public variable in a foreign, simple module can be addressed as *moduleName.variableName* as shown. Array elements are addressed with indexes in parenthesis. Arrays of records are addressed with index and the dot-operator to get a field in element *i*.

With-End. There is a short-hand notation for addressing an object. The With-End statement specifies a partial address, for instance an object. Inside the With-End, all dot and bang-operators are automatically prefixed with this partial address.

6.4 Operators and conversion functions

Operators

Figure 6.4A shows the operators available in VBA. The operators are shown in decreasing precedence, meaning that high-precedence operators are computed before low-precedence operators. This is the rule that ensures that for instance

```
a*b+c*d
```

is computed as (a*b) + (c*d) rather than a* (b+c) *d.

The top operators are the conventional mathematical operators. In general Visual Basic does a good job of converting the operands to the best possible data type before applying the operator.

Null

Null values need attention. Think of Null as *Unknown.* As a general rule, if one of the operands is Null, the result is Null too. Look at

```
If x = Null Then . . .
```

X = Null will always give Null. This is not True and the statement after Then will never be executed. See section 4.5 for examples of handling Null in queries.

There are a few exceptions to the general rule:

Null and False is False.

No matter what the unknown is, the result will be false.

Null or True is True.

No matter what the unknown is, the result will be true.

The **&-operator** concatenates two string operands into one. If one or both operands are non-string, it converts them to strings before concatenation. This also applies to concatenating with Null, which in this case is converted to an empty string. Note that & converts dates to the regional date format. To avoid this, use the Format function to explicitly convert to a specific string format.

Other operators

There are the usual comparison operators, equal, unequal, etc. They can compare numbers as well as texts.

The **Is-operator** compares two object references to see whether they refer to the same object. It may also help checking whether an object reference is Nothing, i.e. refers to no object.

The **Partition operator** translates a value into an interval of values, shown as a string. It takes four parameters:

Value:	An integer.
Lower:	The lowest value considered.
Upper:	The highest value considered.
Int.length:	The range of values is divided into intervals of this length.

As an example Partition(22, 0, 100, 10) = "20:29". The entire range 0:100 is divided into intervals of length 10. The first interval is 0:9, the next 10:19, etc. The value 22 belongs to the interval 20:29.

Between and **In.** Most VBA operators may be used in SQL too. However, *Between* and *In* may only be used in SQL:

```
WHERE a BETWEEN 3 AND 9
WHERE a IN (2, 3, 5, 7)
```

Finally, we have the logical operators. Usually they work on Boolean values, for instance

```
If a Or b Then
```

But if a and b are integers, they work in parallel on all the bits of these integers.

Like operator, wildcarding

The *Like* operator can compare string patterns. It treats some characters in its right-hand operand in a special way. As an example, the character * means *any sequence of characters*. The expression

```
s Like "sim*an"
```

will thus check whether s starts with the characters "sim" and ends with the characters "an" with any characters in between.

The character ? means *any single character here*. The expression

```
s Like "b?n"
```

will thus check whether s starts with b, ends with n, and has exactly one character in between.

The character # means *any digit here*. The sequence [ad3] means *either a, d, or 3 here*. The sequence [a-d3] means *either a letter between a and d here, or the digit 3*. We can even negate the rules: [!ad3] means *neither a, d, or 3 here*. The Like operator is also called the **wildcard operator**.

Fig 6.4A Operators and conversion functions

Operators, decreasing precedence **Nulls**: A Null operand usually gives a Null result.
^ Exponentiation - Unary minus, 2*-3 = -6 * Multiply, Result type is Integer, Double, etc. / Divide, Single or Double result \ Integer divide, result truncated, 5\3 = 1 Mod Modulus (remainder), 5 Mod 3 = 2 + - Add and subtract
& Concatenation, String result (local format)
= <> < > <= >= Equal, unequal, less than, etc. Is Compare two object references, e.g. If r Is Nothing (Test for nil-reference) Partition(22, 0, 100, 10) = "20:29" a Between 3 and 9 Not in VBA, okay in SQL a IN (2, 3, 5, 7) Not in VBA, okay in SQL
Not Negation. Bit-wise negation for integers And Logical And. Bit-wise And of integers Or Logical Or. Bit-wise Or of integers X Exclusive Or. Bitwise on integers Eqv Logical equivalence. Bitwise on integers Imp Logical implication. Bitwise on integers
s Like "s?n" Wildcard compare. ? any char here. # any digit here. * any char sequence here. [a-k] any letter between a and k here.

Conversion to Integer, Double, Date . . . **Errors:** "Invalid use of Null" for Null parameters Overflow or type mismatch for bad parameters.
CByte("37") =37. Overflow outside 0..255 CInt("2.6") = 3 Round(2.6) = 3.0000 (as Double) Rounding down: See Math functions Int, Fix CLng("99456") = 99456 CCur(1/3) =0.3333 (always 4 decimals) CSng("-2.6e-2") = -0.026 CDbl("-2.6") = -2.6 CDbl(#12/31/1899#) = 1.0
CDate("23-10-03") = #10/23/2003# (as Double) Uses regional setting for input date CDate(1) = #12/31/1899# (as Double)
CStr(23) = "23". No preceding space. Str(23) = " 23". Preceding space for numbers >= 0 CStr(#10/23/2003#) = "23-10-03" Converts to regional date format
CVar(X) = X As Variant. X may be Null

Conversion to Integer, Double, Date, etc.

There is a conversion function for most of the types. It converts an expression of another type to its own type. For instance, CInt(D) converts D to an integer - if possible - and returns the result as the value of the function. D might for instance be a string. If D is a decimal number, it is rounded to the nearest integer (up or down).

The function Round(D) does exactly the same as CInt(D) but returns the integer as a Double. See *Math Functions*, section 6.5, for rounding down with Int and Fix.

CDate(D) converts D to a date/time value (technically a Double number). Often D is a string, and CDate is quite liberal in its interpretation of the string. However, some dates are ambiguous. For instance the string "02/03/04" can be interpreted in many ways as a date. In these cases, CDate uses the regional setting for the date format.

CStr( ) can convert a number to a string. It never puts a space (blank) in front of the digits. In contrast, Str( ) puts a space in front of numbers >= 0, a minus in front of negative numbers. Notice that both functions convert dates to the regional date format. To avoid this, use the Format function to convert to a specific date format.

CVar(X) converts X to a variant type. It doesn't really change anything, but sets a type tag on the result it returns.

Format function

The Format function has two parameters: a value to be converted to a text, and the format of the result. The format is a string and its details are a complex affair. Basically each character in the format is either a placeholder that causes VBA to show part of the value, or it is a literal character that is shown as it is. As an example, # will show a digit from the value, while / is not a placeholder and will show up as a / in the result. If a placeholder character is to be shown as the character itself, precede it with a backslash. As an example, \# will show a # in the result.

There are separate placeholders for numbers, strings, and dates. Apparently, the first placeholder determines whether to show the value as a number, a string or date/time. From this point on, the format characters are interpreted according to the type to be shown.

For numbers, the format string can contain from one to four sections separated by semicolons. The first section is used for positive numbers, the second for negative, the third for zeroes, and the fourth for Null values (no value).

The details of the format strings and their effect are best studied through the examples on Figure 6.4B.

Week number

The week number for a date is shown with the ww placeholder. Since different regions of the globe have different rules for what is the first day in the week, the Format function needs a third parameter in this case. It is vbSunday (1) if Sunday is the first day of the week. It is vbMonday (2) if Monday is the first day of the week, etc.

As an example, Sunday 3rd February, 2002 gives these results depending on whether Monday or Sunday is the first day of a week:

```
Format(#2/3/2002#, "ww", 2) = 5 (week 5)
Format(#2/3/2002#, "ww", 1) = 6 (week 6)
```

Formats for controls, etc.

Many control properties require formats of the same kind, for instance the date format for a textbox or a DateTime Picker. Usually the format follows the same rules as VBA's Format function, but there may be deviations. Some very annoying ones are:

MM for month and mm for minute.
HH for 24 hours, hh for hours with AM/PM.

Named formats

The format string may also be the name of a regional (local) format. As an example, the named format "Currency" will show a number with a $ in the US and with a £ in UK. The user can define the regional formats in Window's Control Panel -> Regional and Language Options. The following named formats exist:

Named numeric formats	Named date/time formats
General Number	General Date
Currency	Long Date
Fixed	Medium Date
Standard	Short Date
Percent	Long Time
Scientific	Medium Time
Yes/No	Short Time
True/False	
On/Off	

Fig 6.4B Format function

Format function
Converts a value to a string, based on a format string. Format characters that are not placeholders, are shown as they are. Backslash+character is shown as the character alone, e.g. \d is shown as d.

Numeric placeholders:

0	Digit, leading and trailing zero okay here
#	Digit, no leading or trailing zero here
.	Decimal point (or regional variant)
E- or e-	Exponent, use all placeholders
E+ or e+	Show exponent with plus or minus
%	Show number as percent

```
Format(2.3, "00.00")            = "02.30"
Format(2.36, "#0.0")            = "2.4"
Format(0.3, "##.0#")            = ".3"
Format(32448, "(00)00 00")      = "(03)24 48"
Format(32448, "##.#E+")         = "32.4E+3"
Format(32448, "##.#E-")         = "32.4E3"
Format(0.5, "#0.0%")            = "50.0%"
```

; Separator between formats for positive, negative, zero, and null values.

```
Format(-3, "000;(000);zero;---") = "(003)"
```

String placeholders

@	Character or space
&	Character or nothing
!	Cut off from left

```
Format("A123", "@@@@@@") = "¬¬A123"
Format("A123", "&&&&&&")  = "A123"
Format("A123", "(@@)-@")  = "(A1)-23"
Format("A123", "!(@@)-@") = "(12)-3"
```

Date/time placeholders
Example: DT = #2/3/2002 14:07:09# (Sunday)

```
Format(DT, "yyyy-mm-dd hh:nn:ss", vbMonday)
            = "2002-02-03 14:07:09"
Format(DT, "yy-mmm-d at h:nn am/pm")
            = "02-feb-3 at 2:07 pm"
Format(DT, "dddd t\he y't\h \da\y of yyyy")
            = "Sunday the 34'th day of 2002"
```

d	Day of month, no leading zero "3"
dd	Day of month, two digits "03"
ddd	Day of week, short text "Sun"
dddd	Day of week, full text "Sunday"
ww	Week number. First day of week as third parameter, e.g. vbMonday
m	Month, no leading zero "2" (Interpreted as minutes after h)
mm	Month, two digits "02" (Interpreted as minutes after h)
mmm	Month, short text "Feb"
mmmm	Month, full text "February"
y	Day of year "34"
yy	Year, two digits "02"
yyyy	Year, four digits "2002"

h	Hour, no leading zero "14" or "2"
hh	Hour, two digits "14" or "02"
AM/PM	Show AM or PM here, hours 12-based
am/pm	Show am or pm here, hours 12-based
n	Minutes, no leading zero "7"
nn	Minutes, two digits "07"
s	Seconds, no leading zero "9"
ss	Seconds, two digits "09"

Named formats (local format)

```
Format(2.3, "Currency")        = "£2.30" (in UK)
```

also "Percent", "Yes/No", "Long Date" . . .

6.5 Other functions

String functions

String functions work on strings and characters. Figure 6.5A shows the most important ones.

Asc(s) takes the first character of s and returns it as an integer, the Ascii code for that character. Chr(c) works the other way and returns an Ascii code as a string of one character.

A set of functions return the length of a string (Len), extract the left, right or middle part of a string (Left, Right, Mid), or trim a string of leading or trailing spaces (LTrim, RTrim, Trim).

The functions LCase and UCase transform all letters to upper or lower case. Space(n) generates a string of n spaces.

Comparing strings is in principle easy, in practice difficult due to regional variations of the alphabet, how to treat upper and lower case letters, etc. In each module, you can specify how strings are to be compared. They may be compared according to their Ascii codes (Option Compare Binary), according to the regional alphabet and without case sensitivity (Option Compare Text), or according to the rules of the database (Option Compare Database). In the Access database engine, Option Compare Database seems to work exactly as Option Compare Text.

The function StrComp(s1, s2) compares s1 to find out whether s1 comes before s2 in the alphabetical sequence (result=-1), are equal (result=0), or comes later (result=1). Strings may also be compared simply with s1<s2, s1<=s2, etc.

Iif and Choose

Two functions select one value out of two or more choices. Iif(a, b, c) returns b if a is True, c otherwise. Choose(i, b1, b2, b3 . . .) returns b1 when i=1, b2 when i=2, etc. Figure 6.5A shows the details and examples.

Array bounds

Since arrays can be dynamic with variable bounds, it is convenient with functions that can tell the actual bounds. LBound( ) and UBound( ) give lower and upper bounds for the first or a later dimension.

Dlookup, DMin, DMax, DSum

These function are also called Direct Lookup, Direct Min, etc. They execute an SQL-query on the spot to extract a single value. One example on the figure is

```
DMin("roomID", "tblRoom", "roomType=2")
```

It corresponds to this SQL-statement:

```
Select Min(roomID) From tblRoom
Where roomType=2;
```

The parameters to DMin( ) must be strings that can be substituted into the SQL-statement at the underscored places to give the desired result.

MsgBox

The MsgBox function shows a message to the user, asks for a choice, and returns the choice as the result. There are many parameters, but only the first one is required. It specifies the text to show to the user. The second parameter specifies the icon to show plus the set of choices to give. Later parameters specify box title, help codes, etc. There are many constants available to specify all of these. The example on Figure 6.5A shows a few. (See section 3.6 for more examples.)

Fig 6.5A String functions and miscellaneous

String functions
Null parameters: A Null string as input will give the result Null. Null as another parameter is an error.

```
Asc("AB")       = 65, Ascii code for first character
Chr(65)         = "A", a one-letter string with this
                  ascii character
```

```
Len("A_B")      = 3, length of string.
Left("abc", 2)  = "ab", leftmost two characters
Left("abc", 8)  = "abc", as many as available
Right("abc", 2) = "bc", rightmost two characters
Mid("abcdef", 2, 3) = "bcd",
                  three characters from character 2
LTrim(" ab ")   = "ab ", leading spaces removed
RTrim(" ab ")   = " ab", trailing spaces removed
Trim(" ab ")    = "ab", leading and trailing removed
```

```
Lcase("A-b")    = "a-b", lower case of all letters
Ucase("A-b")    = "A-B", upper case of all letters
Space(5)        = String of 5 spaces
```

Option Compare Text | Binary | Database
Option in start of module. **Text:** string comparison is case insensitive and follows regional settings.
Binary: comparison is based on the internal ASCII code.
Database: comparison is defined by the SQL-engine.

```
StrComp("ab", "abc")   = -1, first string smallest
StrComp("ab", "ab")    = 0, strings equal
StrComp("ac", "abc")   = 1, first string largest
If "ab" < "abc" . . .  ' Works just as well
```

Iif and Choose

```
Iif(a=a, b, c)          = b
Iif(a<>a, b, c)         = c
Iif(Null, b, c)         = c
Choose(2, a, b, c)      = b
Choose(4, a, b, c)      = Null
Choose(Null, a, b, c)   Error
```

Array bounds

```
LBound(d)       Lower bound for first index
LBound(d, 2)    Lower bound for second index
UBound(d)       Upper bound for first index
UBound(d, 3)    Upper bound for third index
```

DLookup, DMin, DMax, DSum
DLookup("name", "tblGuest", "guestID=7")
= name of guest with guestID=7.

DMin("roomID", "tblRooms", "roomType=2")
= smallest room number among double rooms.
All three parameters are texts inserted into SQL.

DMax, Dsum, DCount, DAvg
Similar, just finds largest, sum, number of, average. Null treatment, see SQL

MsgBox
MsgBox("Text", vbYesNo+vbCritical) =vbYes
Also: vbInformation, vbQuestion, vbExclamation

Type check functions

Figure 6.5B shows a set of functions that can check the type of an operand or an expression. As an example, IsDate(d) checks whether d is of type Date or is a string that could be converted to a date by means of CDate(d).

An interesting check function is VarType(v). It returns a number that shows the type of v. The figure shows the many possibilities and the constant name for each of them. In case of an array, two constants are added, vbArray (8192) and the constant for the type of the array. As an example, an array of integers will have the type number vbArray+vbInteger.

Date and time functions

Figure 6.5B also shows a set of functions to handle date and time. Three functions return the current date, the current time, and the current date+time (Date, Time, Now). The function Timer( ) returns the number of seconds since midnight, with the highest accuracy available to the computer. It is a good tool for timing program parts to find the bottleneck in a slow program.

DateSerial computes a date value from the integer arguments year, month, and day. TimeSerial computes a time value from hour, minute, and second.

Finally a set of functions can extract day, month, etc. as integers from a date value. Refer to the figure for details.

Math functions

Figure 6.5C shows the mathematical functions available. We find the square root, the trigonometric functions, exponential function and logarithm. There is also a random number generator.

Abs(x) returns the absolute value of x. Sgn(x) returns the signum of x (1, 0, or -1). Int(x) and Fix(x) rounds the argument with different rules for negative numbers (for positive numbers they are equal).

Hex(x) and Oct(x) shows x as a string in hexadecimal or octal notation.

Financial functions

Figure 6.5C also shows two of the many financial functions available. NPV returns the Net Present Value of a sequence of payments (positive and negative). It calculates what these values would total today at a given interest rate.

IRR calculates the Internal Rate of Return for a sequence of payments. IRR is the interest rate at which the Net Present Value of the payments would be zero. There are many other financial functions available, but we don't explain them here.

Fig 6.5B Type check and date/time functions

Type check functions
Returns True if v is declared with the type tested for, is a Variant currently with this type, or is a constant of this type. IsDate and IsNumeric also test whether v is a text that can be converted to that type.

IsArray(v)	Tests for any type of array
IsDate(v)	Tests whether v is a date or a string that can be converted to a date
IsEmpty(v)	Tests whether v is unallocated (Strings of length 0 are not Empty)
IsError (v)	Tests whether v is an error code
IsMissing (v)	Tests whether v is a parameter that is missing in the current call.
IsNull (v)	Tests whether v is of type Null. (Strings of length 0 are not Null)
IsNumeric(v)	Tests whether v is a numeric type (Byte, Integer, Currency, etc.) or a string that can be converted to a numeric type.
IsObject(v)	Tests whether v is a reference to an object, for instance a Form. True also if v is Nothing (the nil-pointer)
VarType(v)	Integer showing the type:

0	vbEmpty	8	vbString
1	vbNull	9	vbObject
2	vbInteger	10	vbError
3	vbLong	11	vbBoolean
4	vbSingle	12	vbVariant (array)
5	vbDouble	17	vbByte
6	vbCurrency	36	vbUserDefinedType
7	vbDate	8192	vbArray (added)

Date and time functions
A date value is technically a Double. The integer part is the number of days since 12/30-1899, the fractional part is the time within the day.

Several functions accept date parameters as well as strings representing a date and/or time.

Null parameters: Always give the result Null.

Now( )	= current system date and time
Date( )	= current date, integral date part
Time( )	= current time, fractional date part
Timer( )	= Number of seconds since midnight, with fractional seconds.
Date = . . .	Sets current system date
Time = . . .	Sets current system time
DateSerial(2002, 12, 25)	= #12/25/2002#
TimeSerial(12, 28, 48)	= 0.52 (Time 12:28:48)

Day(#12/25/02#) = 25, the day as Integer
Month(#12/25/02#) = 12, the month as Integer
Year(#12/25/02#) = 2002, the year as Integer
Weekday(#12/25/02#) = 4 (Sunday=1)
Hour(35656.52) = 12 (Time 12:28:48)
Minute(35656.52) = 28
Second(35656.52) = 48

Fig 6.5C Math and financial functions

Math functions. Don't accept x = Null:

Sqr(x)	Square root of x. Sqr(9) = 3.
Sin(x), Cos(x), Tan(x), Atn(x)	Trigonometric functions. X measured in radian (180 degrees = p = 3.141592 radian)
Sin(0) = 0, Sin(3.141592 / 2) = 1	
Exp(x)	e to the power of x (e = 2.7182...)
Log(x)	Natural logarithm of x. Log(e) = 1.
Rnd( )	A random number between 0 and 1. Type is Single.

The following functions accept x = Null:

Abs(x)	Returns x for x>=0, -x otherwise.
Sgn(x)	Returns 1 for x>0, 0 for x=0, -1 for x<0
Int(x)	Rounds x down to nearest integral value
Fix(x)	Rounds x towards zero
Hex(x)	Returns a string with the hexadecimal value of x. Hex(31) = “1F”
Oct(x)	Returns a string with the octal value of x. Oct(31) = “37”

Financial functions

NPV(0.12, d( )) The array d must be of type Double and contain a list of payments. Returns the net present value of these payments at an interest rate of 0.12, i.e. 12%.

IRR(d( )) The array d must be of type Double and contain a list of payments. Returns the internal rate of return, i.e. the interest rate at which these payments would have a net present value of 0. If the list of payments have many changes of sign, there are many answers, but IRR returns only one.

IRR(d( ), 0.1) The second parameter is a guess at the interest rate, to allow IRR to find a reasonable result.

SYD, NPer and many other financial functions are available for finding depreciated values, number of periods to pay a loan back, etc.

6.6 Display formats and regional settings

Different regions of the world use different formats for numbers, dates, and other things. The user can define the regional formats in Window's Control Panel -> Regional and Language Options.

This not only influences the input and output format for user data, but also formats inside the program, the control properties, and the query grid. For a developer working outside the US, it can be a nightmare to figure out how it all works. Below I illustrate the issues with the small regional difference I know about, the difference between US formats and central European formats. I cannot even imagine the problems if we talk about regions with different alphabets, writing directions, and calendar systems such as Chinese and Arabic.

Figure 6.6 gives an overview of the situations. The main regional differences relate to the format of dates for input and output, the list separator, and the decimal characters. The regional differences show up in different ways in the three main developer and user environments:

- **Designer environment**: The designer defines controls and their properties. He may also define queries through the query grid.
- **Programmer environment**: The programmer works in Visual Basic and defines SQL-statements rather than queries through the grid.
- **User environment**: The user sees data through controls and datasheets.

Input date. Let us first look at the entry of dates. In the designer and programmer environments, Access makes a good guess at the format for dates. If we for instance enter #2003-13-5# into the query grid or SQL, Access guesses what is year, month and day. However, in the many situations where this is impossible, such as the date #02-03-04#, Access guesses differently in the two environments. In the designer environment, it uses the regional date format, which in Central Europe is #dd-mm-yy#. In the programmer environment, it uses the US format #mm/dd/yy#.

Input mask property. What about the user environment? Here the designer defines what the user sees by means of the Input mask in the Control Property box. When the user starts entering a date, the field switches to showing the date according to the Input mask. This mask is similar to the format string in the Format function, but uses quite different placeholders. When an input mask is specified, the user can only use Overwrite mode, not the ordinary edit mode.

Output date. How is a date shown? In the designer environment, it is always shown according to the regional date format (Short Date).

In the programmer environment it is always shown in US format with one exception: The concatenation operator & converts the operands to texts if needed, and dates are converted to **regional** format. Thus we get this with a central European setting:

```
"Xmas " & #12/25/2007# = "Xmas 25-12-2007"
```

But this with a US setting:

```
"Xmas " & #12/25/2007# = "Xmas 12/25/2007"
```

Format property. The user sees the date in the format defined by the Format property. This format may be different from the input mask, so the user enters one thing, then the system shows it differently. The designer has to make sure that these formats match.

List separator. In Central Europe, the list separator is a semicolon since the comma is used to denote the decimal separator. In the US the list separator is a comma. As a result, the designer has to write for instance Iif(a; b; c) in the query grid or in Control Source properties, while the programmer has to write Iif(a, b, c). To make things worse, if the designer uses commas in the Control Source or the grid, the error reaction is completely confusing.

Fortunately, there is no reason to worry about the user environment for the list separator. Users normally don't see it at all.

Number format. The number format in Central Europe has comma as decimal separator and dot as thousand separator. In the designer environment you have to use these characters, while you use the US format in the programmer environment. In the user environment, the Format property and the Input mask determine the format.

Practice. Developers are sometimes designers and sometimes programmers. It is no wonder that they usually set up their development system in pure US mode. They can ship the system to the users with little change since the users don't see the designer stuff. But if they have to change something at the user's site, they have to take care since the developer formats now look different from back home.

Fig 6.6 Regional settings

Display format depends on regional setting and designer environment:

	Designer Query grid, Property line	Programmer SQL, VBA	User Table field, Textbox
Input date	VBA guess, e.g. #2003-13-5# In doubt, e.g. #02-03-04#: dd-mm-yy	mm-dd-yy	According to Input mask
Output date	EU-regional: dd-mm-yy	US: mm/dd/yy Except & which uses regional format	According to Format
List separator	EU-regional: semicolon: iif(a ; b ; c)	US: comma: iif(a , b , c)	
Number	12.300,25	12,300.25	Format ...

7. Access and SQL

In this chapter we look at more complex SQL issues, for instance how we can create and delete records through SQL, how we can use queries inside queries, and how we can show matrices of data where the table headings vary according to the data.

7.1 Action queries - CRUD with SQL

Section 5.6 explains how the program can Create, Read, Update and Delete records in the database by means of *recordsets*. In this section we will see how the program can make such CRUD operations by means of SQL.

We have four kinds of SQL queries available:

INSERT INTO . . . (Create)
: This query inserts new records into a table.

SELECT . . . FROM . . . (Read)
: This query extracts data from the tables. It is the query we have used in the rest of the booklet (e.g. sections 4.1 to 4.4). In many cases it also allows the user to edit the data. We will not explain it further in this section.

UPDATE . . . (Update)
: This query changes records in a table.

DELETE FROM . . . (Delete)
: This query deletes records from a table.

The term *action query* means INSERT, UPDATE or DELETE - the queries that change something in the database.

We will explain the mechanisms through the examples in Figure 7.1. To execute an action query, we use this VBA function:

```
CurrentDb.Execute "INSERT . . . ", dbFailOnError
```

CurrentDB is the database currently used by the program. We ask it to execute the INSERT query. Usually the query statement is a computed string that includes dialog variables, for instance the customer the user works with.

The last parameter *dbFailOnError* is optional. It asks the database to give an error message if something goes wrong during the query, for instance if we try to insert a record that violates referential integrity. If we omit it, there is no response in case of errors - nothing changes in the database, and the program doesn't know about it. Include dbFailOnError, particularly while you are testing the program.

INSERT
The figure shows a full INSERT statement. It inserts a single record into tblRoomState. It sets the roomID to 14, the date to 27th Oct 2002, and the state to 1. The remaining fields of the record become Null. Note how we have put *date* in brackets. If we omit the brackets, the database engine believes we try to use the built-in *date* function and reports it as a syntax error (independent of whether we have included dfFailOnError).

The INSERT statement can also insert a bunch of records extracted from another table or query. The bottom of the figure shows an example (explained below).

AutoNumber
Assume you have a table with a field of type *AutoNumber*. When you insert a record into the table, Access will automatically fill in the AutoNumber field. However, if you explicitly specify a value for the field, Access will accept it if it is unique (different from other values of this field).

When Access generates the number automatically, it will use the highest used number plus one. (The highest used number may have been used by a record now deleted.) As an example, assume we want the booking numbers in the hotel to consist of a year number and a sequence number, for instance 20040001, 20040002 etc. We can then generate the first booking of the year with this query:

```
INSERT INTO tblStay (stayID) VALUES (20040001)
```

New stay records will then be auto-numbered from this value and on. Deleting some of the records still makes the number grow sequentially.

UPDATE
The UPDATE statement sets room state to 2 and personCount to 1 for all roomState records where roomID=14 and date=27/10/2002. In this case there is only one such record because roomID and date make up the primary key for the table.

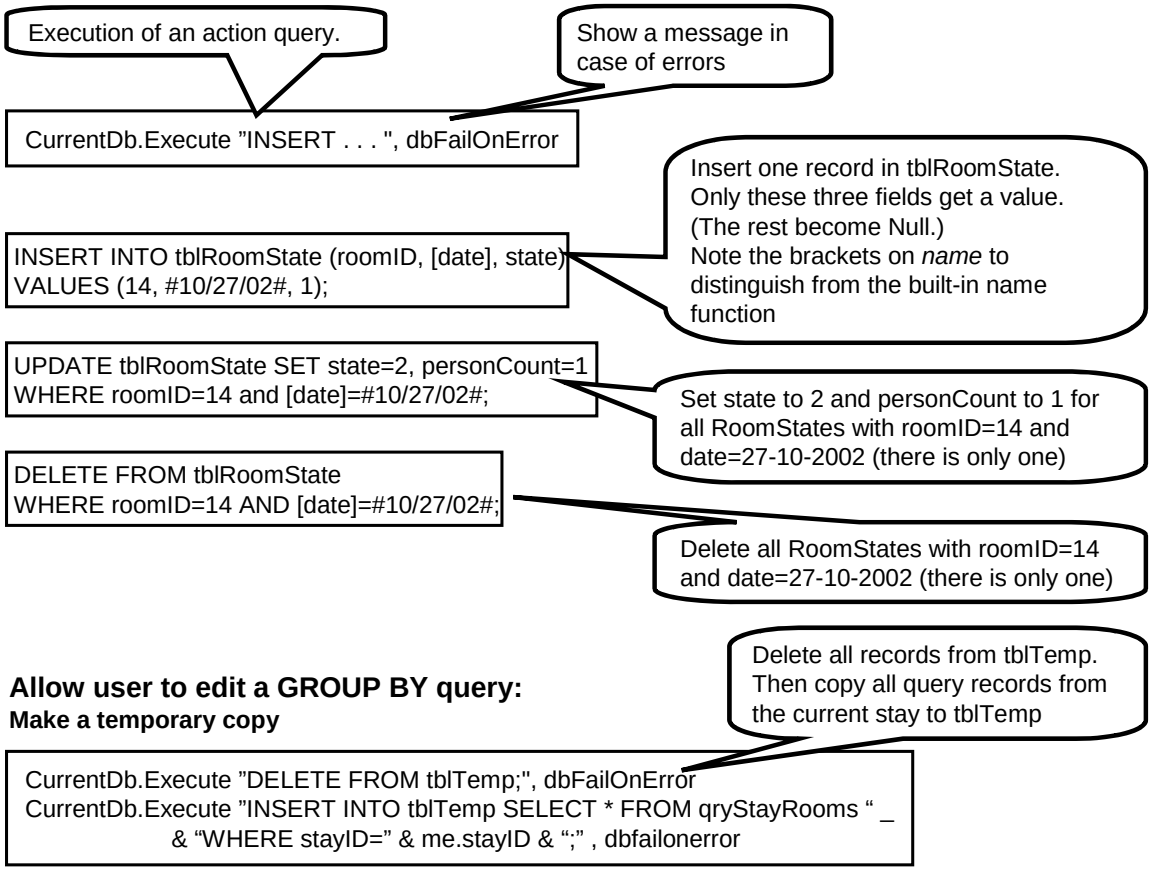
Confused about the dates? Access SQL uses American dates, while the explanations in the booklet use middle European dates. See more in section 6.6.

DELETE
The DELETE statement deletes all roomState records where roomID=14 and date=27/10/2002. In this case there is only one such record.

7.1.1 Temporary table for editing

The user (or the program) can edit the fields in a simple query. However, if the query contains a GROUP BY,

Fig 7.1 Action queries - CRUD with SQL



an outer join, or another complex query, the result can not be edited. From a usability perspective, there is often a need to edit such a query anyway. It can either be done through a dialog box (low usability), or through a temporary table that holds a copy of the GROUP BY result. The user will edit the temporary table directly. The bottom of the figure shows an example of this. Let us see how it works.

In the hotel system the user sees a single record for each room booked by the guest. In the database there is a roomState record for each date the room is booked, but on the user interface we aggregate it to a single line with start date and number of nights. The query *qryStayRooms* does this, but the user can not edit the result directly.

The solution is to have a temporary table *tblTemp* with the same fields as qryStayRooms. The first statement deletes the present records in tblTemp. The next statement copies all the query records that belong to the current stay into tblTemp. Note how the program computes the SQL statement from several parts, one of them being *me.StayID*, the current stay number the user is working on. Since the SQL statement is very long, it is split into two lines with an underscore at the end of the first line.

Note also the SELECT * part that extracts all fields of the query. This only works if tblTemp has the same fields - and with the same names. It is possible to extract only some of the fields - or rename them - in order to match the fields in tblTemp. We could for instance extract roomID (renamed to *Room*) and Nights:

```
INSERT INTO tblTemp SELECT roomID AS Room, Nights . . .
```

After the INSERT, the program can show tblTemp in a form so that the user can edit it. The program may either update the real database each time the user changes something in tblTemp, or it may update the whole thing when the user closes the form.

7.2 UNION query

A UNION query combines two queries so that the result contains all the records:

```
SELECT . . . FROM . . . UNION
SELECT . . . FROM . . . ;
```

The two queries must have the same number of fields. The queries need not have matching names or types. The first field of each query goes into the first field of the result, the second field into the second field of the result, etc. The field name of the result is the field name from the first query. The field type of the result is compatible with all the queries, for instance Long Integer if one query has a simple integer, the other a Long.

The query grid cannot show a union query. You have to work in SQL view. You may make one of the queries with the query grid, then switch to SQL view to add the second query.

You can set the **presentation format** of the fields in grid view, for instance date or number formats. Right-click the field column and select *properties*. When you switch to SQL and add the union part, the presentation format remains. (Editing the SQL version and switching back and forth between SQL view and grid view, may remove the format information.)

Example: Service charges
The first example in Figure 7.2 takes a list of breakfast services and adds a list of telephone charges to give a combined list of service charges. (The breakfast services are recorded by the receptionist, while the telephone charges may be produced by a separate telephone system.)

The combination is not quite straightforward because the breakfast list has a stayID field which is not wanted in the result. Furthermore, the telephone charges lack a serviceID; they should end up with a serviceID of 5 (meaning *telephone*). Finally, what is called *ticks* in the telephone charges is called *quantity* in the result.

The first part of the SQL statement selects the required fields from tblServiceReceived. The second part selects roomID from the telephone charges, computes a second field with the value 5, selects the date field and finally the ticks field. It doesn't even care to rename the ticks field.

The figure shows the result ordered by roomID. Notice that the last field is named *quantity* according to the field name in the first query.

In the real hotel system, the result must be joined with the roomState records to add the stayIDs, so that the data end up on the right guest invoice. To do this properly, we also need time of day for the telephone charges. We don't discuss this here.

Example 2: Combo box with a *New* option
The second example in Figure 7.2 shows a list of service prices extended with a *New* option. The list is intended for a Combo Box and gives a convenient way for the user to select an existing choice or add a new one.

The list is computed with a union query. The last part is a simple select of the fields in tblServiceType. The first part computes a constant record with the name *New*, serviceID = 0 (dummy value), and price = Null. Formally, this record is computed based on tblServiceID, but any table might be used. However, to save computer time, it is a good idea to use a short table. Some developers use a dummy table with a single record for the purpose.

This query is used as the Control Source in the Combo Box.

Note how we managed to get the currency sign ($) on the list. We first used the query grid to make the last part of the union. We set Property -> Format to Currency for the price field. Then we switched to SQL view and added the first part.

An alternative is to compute the price field using the Format function, for instance with the named format *Currency*, which adds the regional currency symbol:

```
Format(tblServiceType.price, "currency") AS Price
```

Fig 7.2 Union: Concatenate two tables

	stayID	roomID	serviceID	date	quantity
	728	11	4	23-10-02	2
	728	12	3	22-10-02	1
	739	16	3	23-10-02	1
▶	737	21	3	22-10-02	2
✳	0	0	0		0

List of breakfast services, etc.

	roomID	date	ticks
	11	23-10-02	138
	12	22-10-02	27
▶	21	22-10-02	14
✳	0		0

List of telephone charges from the telephone system.

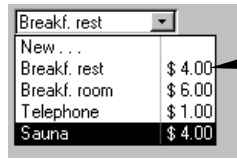
	roomID	serviceID	date	quantity
	11	4	23-10-02	2
	11	5	23-10-02	138
	12	3	22-10-02	1
	12	5	22-10-02	27
	16	3	23-10-02	1
	21	3	22-10-02	2
▶	21	5	22-10-02	14

Combined list of all services.

```
SELECT roomID, serviceID, date, quantity FROM tblServiceReceived
UNION SELECT roomID, 5 AS serviceID, date, ticks FROM tblPhone;
```

---

	serviceID	name	price
▶	0	New . . .	
	3	Breakf. rest	$ 4.00
	4	Breakf. room	$ 6.00
	5	Telephone	$ 1.00
	6	Sauna	$ 4.00



List of service types extended with *New*. Shown in table view, and as drop-down list.

Start with query grid and last part. Set column property for *price* column to *currency*.
Switch to SQL and add the first part (SELECT . . . UNION).

```
SELECT 0 AS serviceID, "New . . ." AS name, Null AS price
FROM tblServiceType
UNION SELECT serviceID, name, price FROM tblServiceType;
```

7.3 Subqueries (EXISTS, IN, ANY, ALL . . .)

A subquery is a select statement used inside another select statement. As an example, this statement selects records from tblA that have a value that is in tblB:

```
SELECT . . . FROM tblA WHERE
    v IN (SELECT w FROM tblB)
```

You can use subqueries in this way:

`EXISTS (SELECT * FROM . . . )`
: True if the subquery returns one or more records. You can also write NOT EXISTS (. . .)

`v IN (SELECT w FROM . . . )`
: True if the set of w's returned by the query contains the value v. You can also write v NOT IN (. . .)

`v IN (3, 7, a+b)`
: True if v is in the list of values 3, 7, a+b. This is not really a subquery, because we write the list of values explicitly.

`v > ANY (SELECT w FROM . . . )`
: True if the value v is larger than one or more of the w's returned by the query. You can also write v=, v>, v>=, etc. You can write SOME instead of ANY. The meaning to the computer is the same.

`v > ALL (SELECT w FROM . . . )`
: True if the value v is larger than all of the w's returned by the query. You can also write v=, v>, v>=, etc.

An alternative to v > ANY is:
`v > (SELECT Min(w) FROM . . . )`
Similarly an alternative to v > ALL is:
`v > (SELECT Max(w) FROM . . . )`

`EXISTS (SELECT . . . ) AND v > ANY (SELECT . . .)`
: You can combine subqueries in the same way as other logical expressions.

In summary, you can use a subquery in a logical expression. You cannot join a subquery with other subqueries, group it, etc. In order to do such things, you have to store the subquery as a named query (we show an example in section 7.4).

Example: Used and free rooms
The first example in Figure 7.3 shows a list of rooms with an indication of those used in the period 23/10 to 26/10.

In principle the query is quite simple. We query tblRoom and select the roomID and a computed value. The computed value makes the trick. It tests whether the room has a roomState record in the period 23/10 to 26/10. The result will be True or False. We give the result the name *Used*.

In this case we show the result as Yes/No. We have set Property -> Format to Yes/No for the *Used* field.

Notice the comparison operator **BETWEEN**. It is part of standard SQL, but not of VBA. This is one of the few exceptions where an expression in SQL doesn't work in VBA. The other way around, lots of VBA expressions don't work in standard SQL although they work in Access SQL.

Example 2: List of free rooms using NOT EXISTS
The second example lists only the free rooms. Again we select roomID from tblRoom, but in the WHERE clause we check that the room has no roomStates in the period we consider.

Example 3: List of free rooms using NOT IN
The third example also lists the free rooms, but selects them in a different way. In the WHERE clause the subquery lists all roomStates in the period we consider. The WHERE clause checks that the roomID from tblRoom is not in this list.

Correlated queries
SQL specialists talk about correlated queries. The subquery is *correlated* with the main query if it references fields in the main query. In the figure, the first two queries have correlated subqueries while qryTest3 has not.

Example 4: DISTINCT values only
A query may produce a list of records with duplicates. QryTest 4 shows an example. It extracts all roomState records in the period we consider, and selects the roomID. Since many rooms are occupied multiple times in this period, the same roomID will appear many times.

QryTest 4A has added the word DISTINCT. This causes SQL to remove all duplicates.

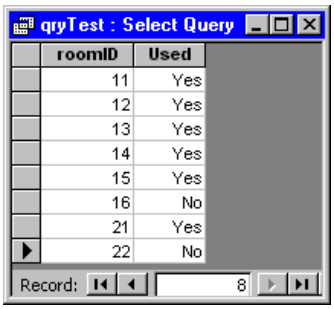
Example 5: Self-correlation
The fifth example shows how we can select records based on other records in the *same* table.

The example is from the hotel system. A guest may have booked several rooms for different periods. When guests arrive, we want the receptionist to see a list of the rooms booked for the arrival date.

The query looks at each roomState record to see whether it must be included in the list. It must be if the date of this roomState is the arrival date for the corresponding stay. The subquery finds the arrival date by looking at all roomStates that belong to this stay. The arrival date is the first date among these roomStates.

Note how the subquery selects tblRoomState. It gives the table an alias-name (T2) in order to compare these room states with those from the outermost query. This is called a **self-correlation**.

Fig 7.3 Subqueries (Exists, In, etc.)



qryTest : Select Query

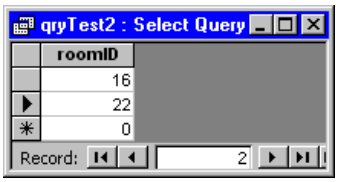
roomID	Used
11	Yes
12	Yes
13	Yes
14	Yes
15	Yes
16	No
21	Yes
22	No

Record: 8

qryTest:

```
SELECT roomID,
EXISTS (SELECT * FROM tblRoomState
WHERE tblRoomState.roomID = tblRoom.roomID AND
(tblRoomState.date BETWEEN #10/23/02# AND #10/26/02#))
AS Used
FROM tblRoom;
```

Room list with indication whether the room was used between 23/10 and 26/10.



qryTest2 : Select Query

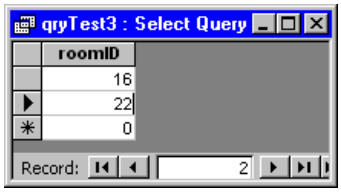
roomID
16
22
0

Record: 2

qryTest2:

```
SELECT roomID FROM tblRoom WHERE
NOT EXISTS (SELECT * FROM tblRoomState
WHERE tblRoomState.roomID = tblRoom.roomID AND
(tblRoomState.date BETWEEN #10/23/02# AND #10/26/02#));
```

Rooms free between 23/10 and 26/10



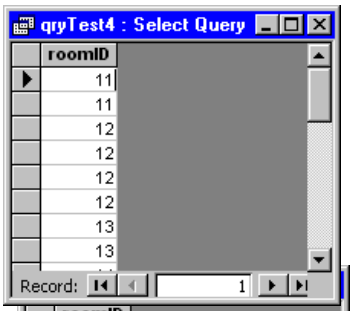
qryTest3 : Select Query

roomID
16
22
0

Record: 2

qryTest3:

```
SELECT roomID FROM tblRoom WHERE
roomID NOT IN (SELECT roomID FROM tblRoomState WHERE
tblRoomState.date BETWEEN #10/23/02# AND #10/26/02#);
```



qryTest4 : Select Query

roomID
11
11
12
12
12
12
13
13

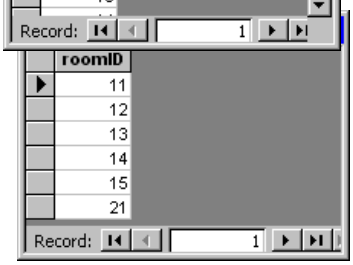
Record: 1

qryTest4:

```
SELECT  roomID FROM tblRoomState
WHERE tblRoomState.date BETWEEN #10/23/02# AND
#10/26/02#;
```

Room states in the period. Same roomID occurs multiple times

Duplicates removed



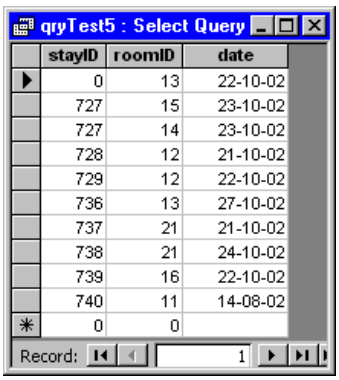
roomID
11
12
13
14
15
21

Record: 1

qryTest4A:

```
SELECT DISTINCT roomID FROM tblRoomState
WHERE tblRoomState.date BETWEEN #10/23/02# AND
#10/26/02#;
```

Rooms for the arrival date, i.e. the first date of the stay.



qryTest5 : Select Query

stayID	roomID	date
0	13	22-10-02
727	15	23-10-02
727	14	23-10-02
728	12	21-10-02
729	12	22-10-02
736	13	27-10-02
737	21	21-10-02
738	21	24-10-02
739	16	22-10-02
740	11	14-08-02
0	0	

Record: 1

qryTest5:

```
SELECT stayID, roomID, [date] FROM tblRoomState
WHERE [date] =
(SELECT Min([date]) FROM tblRoomState AS T2 WHERE
tblRoomState.stayID = T2.stayID)
ORDER BY stayID;
```

Self-correlation with alias.

7.4 Multiple join and matrix presentation

In this section we will look at queries that present data in a matrix. A matrix is a table where the headings aren't fixed but vary according to data in the database.

Figure 7.4 shows an example from the hotel system. It is part of the rooms window, which shows when rooms are occupied. The rooms are listed vertically and the dates horizontally. Whether a room is booked, occupied, etc. on a given date is shown as a number code. This is just to simplify the example a bit. In the real system the code is replaced with a mnemonic text, for instance *Booked*.

The principle
The query behind the table follows this idea:

```
SELECT roomID . . . , r1.date, r2.date, r3.date
FROM tblRoom
LEFT JOIN (tblRoomState AS r1 WHERE date=first) ON . . .
LEFT JOIN (tblRoomState AS r2 WHERE date=first+1) ON . . .
LEFT JOIN (tblRoomState AS r3 WHERE date=first+2) ON . . .
```

We start with tblRoom and left-join it with the room states belonging to the first date. This would give us the roomID column and the first date column in the result. Next we left-join the result with the room states belonging to the next date. This would add the second date column in the result. And so on.

Practice
Unfortunately this is wishful thinking, but not SQL. The main problem is that we cannot have a subquery with WHERE after JOIN. It doesn't help to add SELECT . . . FROM inside the parenthesis. Could we have a WHERE after the last ON, such as this?

```
LEFT JOIN tblRoomState AS r3 ON . . .
WHERE r1.date=first AND r2.date=first+1 AND r3.date=first+2
```

This would work okay if we were using INNER JOINs, but we need outer joins to produce all the empty cells in the table. Another guess would be to set the necessary conditions in the ON criterion. This too is not allowed in Access SQL.

Named query. The solution is to use a named query for each of the parentheses. Figure 7.4 shows the full solution. Using the query grid in Access, we create a query for each of the date columns in the result. The figure shows qryRoom1, the query for the first date column. It simply selects the roomState records for the first date in the period. QryRoom2 selects the roomState records for the second date, and so on.

The **room grid** is now computed with the big query. It follows the idea above, but uses the named queries instead of the parentheses. It also adds room descriptions through an INNER JOIN with tblRoomType. Finally, it renames the date columns to the proper headings, for instance [21-10-02] for the first date column.

The big query was actually made with the query grid, as shown on the figure. This helped set the proper parentheses inside the SQL statement. As long as the statement contains only inner joins, the grouping with parentheses doesn't matter. The result will be the same no matter in which sequence the joins are made. However, when outer joins are used, the sequence *does* matter.

Standard SQL allows outer joins to be freely grouped with parentheses, but Access SQL does not. The Access rule is that a table with an arrow pointing to it cannot have another arrow pointing to it, nor be connected with an arrow-less connection. If needed, you can get around this rule by means of named queries.

Dynamic headings, Form versus table
In order to make a true data matrix, we need to change the column headings according to the data. In the example, the user chooses the period of dates. The program then computes the corresponding SQL statements and inserts the appropriate dates. This is rather straightforward.

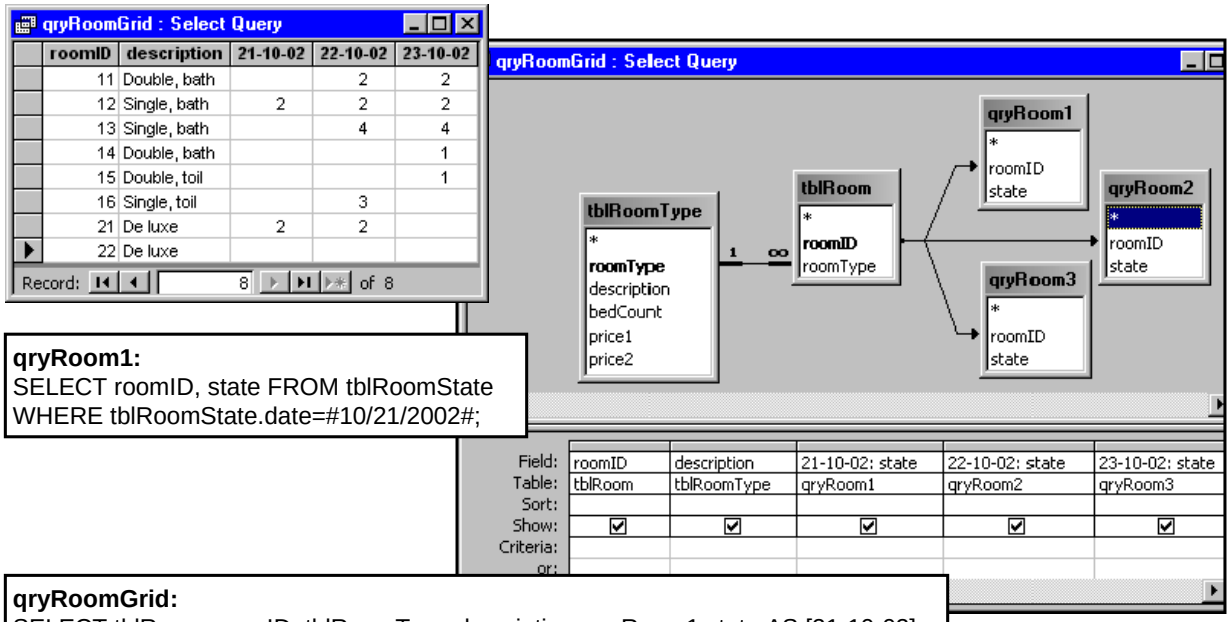
All of this works okay if we just want to show the user the result as a table. The query determines the table headings.

However, the table presentation provides a poor user interface. We may connect the table to a subform control (Access 2000 and 2003), but this gives us no possibility to change the appearance, for instance removing navigation buttons or coloring the fields. The user may select an area of the table, but the program has no way to detect which area the user selects.

The solution is to make a Form based on the query, and set its default view to *Datasheet*. (A datasheet looks like a table, but behaves differently.) We may use the form as it is or connect it to a subform control. Now we can control the appearance of the datasheet, and the program can "see" the user's selections of data cells.

Unfortunately this introduces another problem. The query doesn't any more determine the headings. They are defined by the field labels on the form. In the next section we show how to manage all of this.

Fig 7.4 Multiple join and matrix presentation



qryRoom1:
SELECT roomID, state FROM tblRoomState
WHERE tblRoomState.date=#10/21/2002#;

qryRoomGrid:
SELECT tblRoom.roomID, tblRoomType.description, qryRoom1.state AS [21-10-02],
qryRoom2.state AS [22-10-02], qryRoom3.state AS [23-10-02]
FROM tblRoomType INNER JOIN (((tblRoom
LEFT JOIN qryRoom1 ON tblRoom.roomID = qryRoom1.roomID)
LEFT JOIN qryRoom2 ON tblRoom.roomID = qryRoom2.roomID)
LEFT JOIN qryRoom3 ON tblRoom.roomID = qryRoom3.roomID)
ON tblRoomType.roomType = tblRoom.roomType;

7.5 Dynamic matrix presentation

In this section we show how to make a dynamic matrix as a Form in datasheet view. Compared to showing the matrix as a query table, the Form approach allows us to control the presentation better and let the program "see" what the user selects.

Figure 7.5 shows the details of the solution. The form at the top left is an outline of the hotel system's rooms window. The user can enter the first date of the period, click *Find*, and the system updates the subform (the datasheet) to show room occupation from that date and on. Initially, we assume that the period is always three days.

QryRoomGrid is exactly as the version in the previous section, except for the AS parts of the SELECT clause. The room states are now renamed to C1, C2, C3 rather than the dates in the period. This ensures that the states always end up in the proper form fields, no matter which dates we work with.

The datasheet is based on the subform shown to the right. This subform was generated with the Form Wizard using qryRoomGrid as the record source. It uses a columnar layout (see section 3.2.1). The date fields are bound to C1, C2, and C3 in the query. (The names of the controls will be cbo1, cbo2, etc., but this is not important.) We have manually given the labels the names L1, L2, L3.

The program doesn't have to compute the SQL-statement in qryRoomGrid. This query is based on the named queries qryRoom1, 2, and 3, and the program has to compute these when the user clicks *Find*. The figure shows the Click procedure for *cmdFindRoom*.

First the procedure stores the main part of the query in the string variable s. This is only to save a lot of writing since the string is used in three queries.

Next the procedure computes the SQL-statement in qryRoom1. The procedure computes qryRoom2 and qryRoom3 in the same way. Note how we address the **named query** from VBA, using

```
CurrentDB.QueryDefs
```

Notice how we compute the date comparison, for instance

```
SELECT . . . WHERE date = 37550;
```

We explicitly convert the contents of txtDate (the user's search criterion) to Double. The &-operation will then translate it to the string "37550". (See more on date comparison in section 5.2.3.)

Changing the headings
The next part of the procedure sets the dynamic headings. It sets the captions of the three labels to the proper date. Notice how we first address the subform control, then the Form connected to the control, and finally the label in the form.

We use the automatic date conversion when setting the caption. It uses the regional date format - exactly what we want when showing the date to the user.

Updating the subform
Finally the procedure has to update the subform according to the new queries. This is done by setting the record source once more to *qryRoomGrid*. This causes Access to compute the query from scratch.

Updating the various parts of a form is a mystery - at least to me. Through experiments I have found the solution above. It seemed more natural to use the built-in requery method for a form, for instance in this way:

```
Me.subRoomGrid.Form.Requery
```

However, Access (or the Jet engine) doesn't figure out that something has changed in the named queries, and as a result doesn't update anything.

Handling varying number of columns
The techniques above can be generalized to a period of N days, where N is defined by the user. First the Click procedure needs to compute also qryRoomGrid with C1, C2 . . . CN and N nested parentheses. This is not too hard.

We might create a variable number of named queries in a loop where i runs from 1 to N. The inner part of the loop would use this statement to create a named query:

```
Call currentdb.CreateQueryDef ("qryRoom" & i, "select . . . ")
```

However, making a variable number of fields on the form cannot be made dynamically. Adding fields can only be made in design mode.

In practice we have to plan for a maximum number of date columns, for instance 20. We construct fsubRoomGrid with 20 date fields, cbo1, cbo2 . . . cbo20, and bind them to C1, C2 . . . C20. We also make 20 named queries, qryRoom1 . . . qryRoom20.

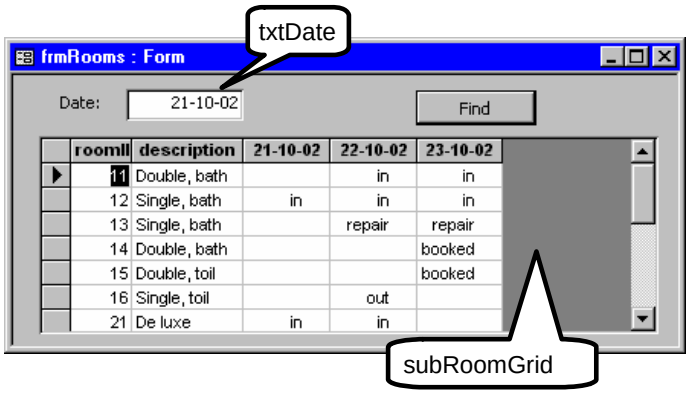
With a proper Click procedure this works fine except for a little detail: The datasheet always shows 20 columns even if the user only asked for 3. The last columns will be blank if the query didn't generate something.

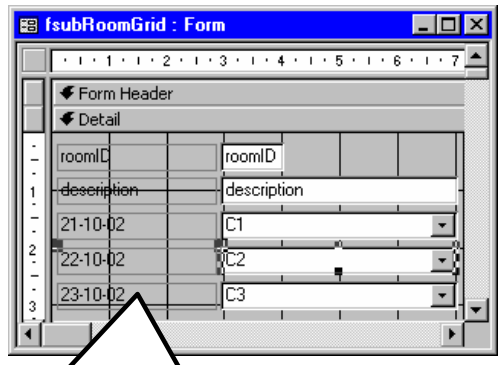
This is easy to correct. The procedure can hide the last columns. It can for instance hide the 4th date column in this way:

```
Me.subRoomGrid.Form!cbo4.ColumnHidden=True
```

In the procedure, the number 4 is given by a variable, for instance j. The statement would then be:

Fig 7.5 Dynamic matrix presentation





Column headings are the label captions.
Label names: L1, L2, L3

qryRoomGrid:

```
SELECT tblRoom.roomID, tblRoomType.description,
qryRoom1.state AS C1, qryRoom2.state AS C2, qryRoom3.state AS C3
FROM tblRoomType INNER JOIN (((tblRoom LEFT JOIN qryRoom1 . . .
```

frmRooms:

```
Private Sub cmdFindRoom_Click()
Dim s As String
   s = "SELECT roomID, state From tblRoomState WHERE date = "
   CurrentDb.QueryDefs!qryRoom1.SQL = s & CDbl(Me.txtDate) & ";"
   CurrentDb.QueryDefs!qryRoom2.SQL = s & CDbl(Me.txtDate+1) & ";"
   CurrentDb.QueryDefs!qryRoom3.SQL = s & CDbl(Me.txtDate+2) & ";"
   Me.subRoomGrid.Form!L1.Caption = Me.txtDate
   Me.subRoomGrid.Form!L2.Caption = Me.txtDate + 1
   Me.subRoomGrid.Form!L3.Caption = Me.txtDate + 2
   Me.subRoomGrid.Form.RecordSource = "qryRoomGrid"
End Sub
```

. . . WHERE date = 37550;

. . . WHERE date = 37551;

Set RecordSource again.
(Requery doesn't notice that the named queries changed.)

Me.subRoomGrid.Form("cbo" & j).ColumnHidden=True

See section 5.5.7 for more on hiding columns, adjusting their width, etc.

The real rooms window

In the real hotel system, the user sees the days around the first day he asks for and the days around the last day. When the period is long, all the days in the middle are lumped together with a heading saying ". . . ".

The queries are also more complex because the user can ask for rooms that are free in a certain period, rooms of a certain type, etc. However, the solution follows the ideas above.

Another issue is to let the user select rooms and period from the matrix. This is discussed in section 5.5.8.

7.6 Crosstab and matrix presentation

The previous sections show how hard it is to make a data matrix. So there is a good reason that Access provides a non-standard SQL feature for making data matrices: the Crosstab query.

A simple example - room number versus date
Figure 7.6A shows a simple example. We want a matrix with dates running horizontally and rooms running vertically. In the matrix we show the room state for each date (as a numeric code for simplicity).

This matrix is made with the query shown to the right. It is based on tblRoomState. We have used the menu

Query -> Crosstab Query

to add the Crosstab gridline to the grid. We have marked *roomId* as a *row heading* and *date* as a *column heading*. Note how we group data by roomId and date. Each group is a bundle of room state records to be shown somehow in the matrix cell. In the hotel system, each bundle will contain at most one record since a room cannot have more than one state for a given date. This is just how it happens to be in the hotel case, however.

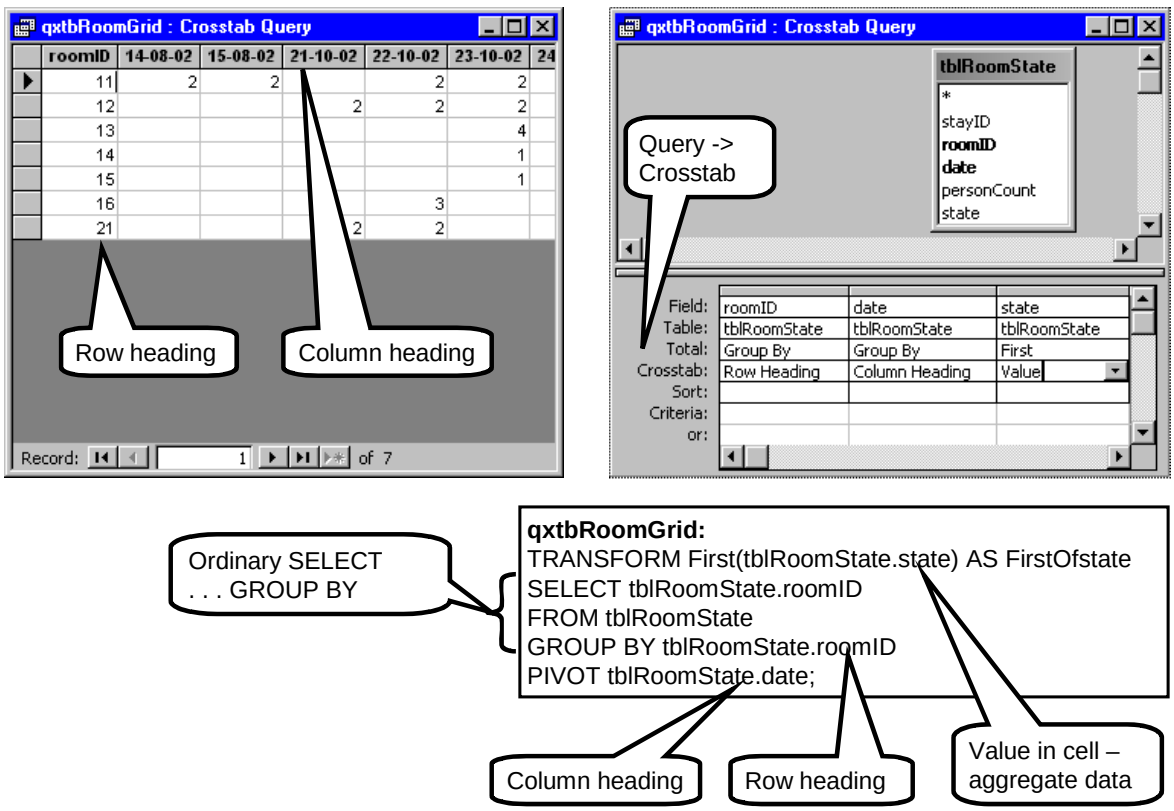
Finally, we have marked *state* as the *value* to be shown inside the matrix. Since a crosstab is a kind of aggregate query, we can only show data as aggregate functions. In this case we have used the aggregate function *First*. Since the bundle has at most one room state, we could as well have used Max or Min.

The figure also shows the SQL version of the query. The inner part of the query is an ordinary SELECT - GROUP BY query. The group-by attribute becomes the row heading. There may be several group-by attributes, and then there will be several row headings.

The first part of the query is a non-standard thing, the TRANSFORM part. It defines the value to be shown in a cell. The last part is another non-standard thing, the PIVOT part. It defines the attribute to be used as the column heading.

Note the query name: qxtbRoomGrid. The prefix qxtb tells the programmer that it is a crosstab query.

Fig 7.6A Crosstab query



Outer join and several row headings

Figure 7.6B shows a more complex crosstab. We want to show also the room description. Furthermore we want to show all rooms, whether they are used or not. (In the top of the figure, room 22 wasn't shown because it wasn't used.)

The new crosstab is based on three tables: tblRoom-State, tblRoom, and tblRoomType. We use an outer join for tblRoomStates and tblRoom in order to include all rooms. We mark *roomID* and *description* as row headings.

The result is as shown. We have now got room 22 in the matrix. Unfortunately, we have also got a strange column with the heading <>. It is caused by the outer join producing a row with a Null date since no room state matched room 22.

Figure 7.6C shows how to get rid of the Null column (<>). Instead of grouping by date, we group by a computed date:

d: IIf(IsNull(date), #24-10-02#, date)

The effect of this is that Null dates are replaced by a specific date in the range of dates we consider. Records with a Null date also have a Null state, so they don't show up as being used.

Fig 7.6B Outer join and two row headings

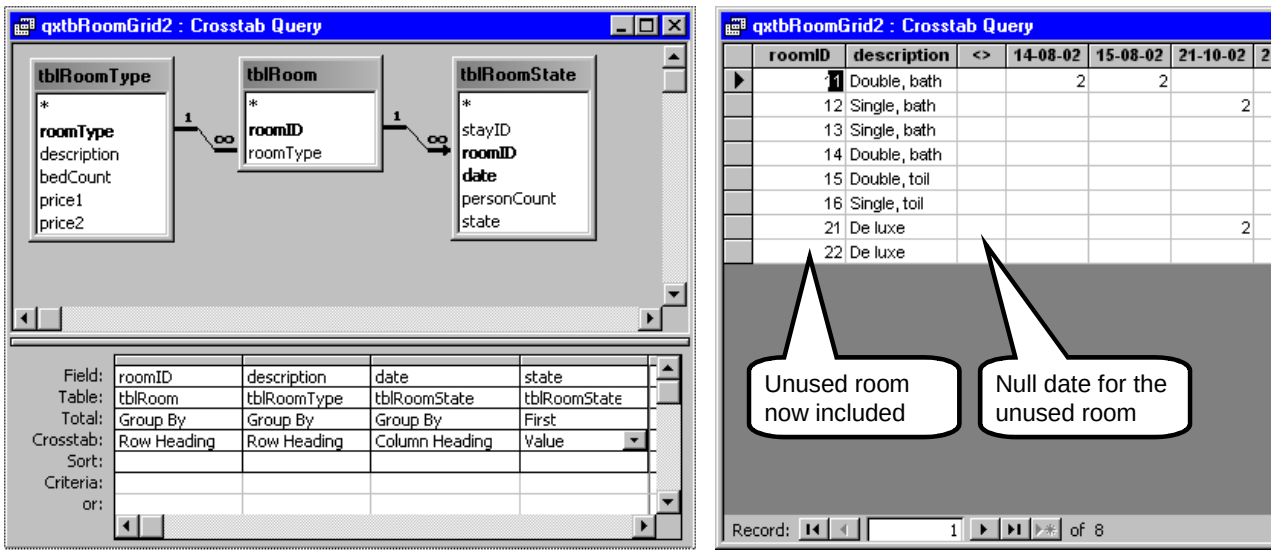


Fig 7.6C Avoiding the Null date

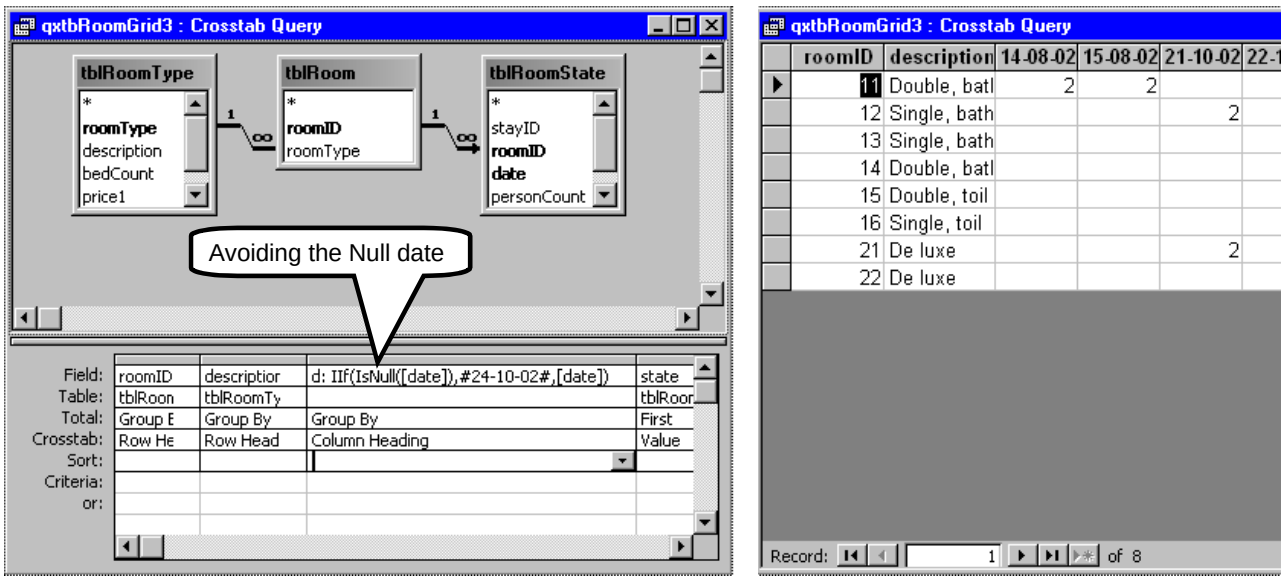
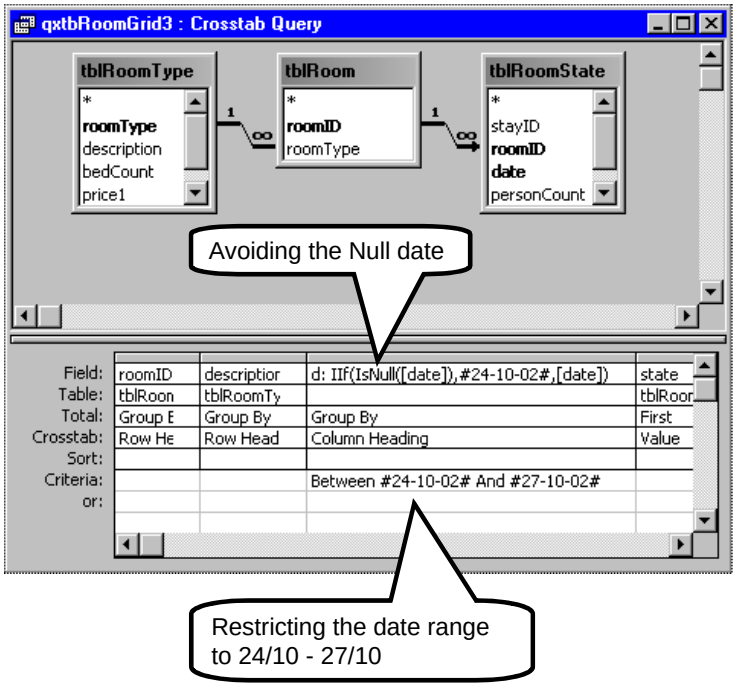
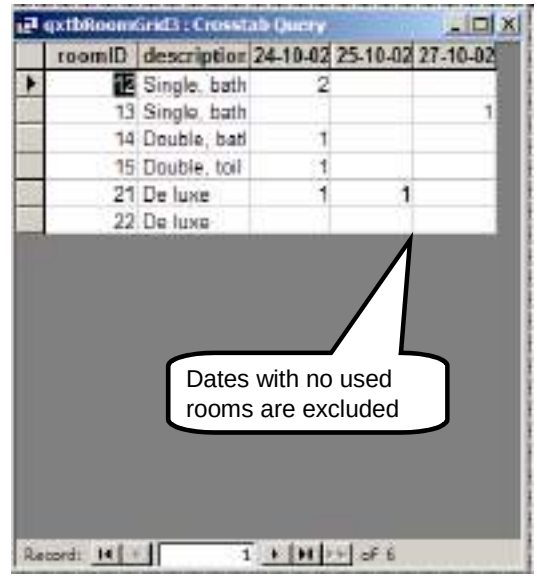


Fig 7.6D Restricting the date range





Restricting the date range
In practice we cannot show a matrix with all the dates. It would require hundreds of columns. We want to restrict the query to a specific range of dates.

In Figure 7.6D we restrict the range of computed dates to those between 24-10-02 and 27-10-02 (European date format).

The result is as shown. We got rid of the <> column and see only the dates in the restricted range. The only problem is that we don't get a column for every date in the range. For instance 26-10-02 is missing because no room is used this date.

Including all dates in the range
We might include all dates in the range by means of an additional outer join. To do this we need a table of all dates in the range, but this is not an existing part of the data model. The program might generate such a table, but let as utilize another feature in the crosstab.

In Figure 7.6E we have added an IN-clause to the PIVOT part of the SQL statement. The PIVOT part specifies the column headings, and now we have explicitly asked for all the dates in the range. Notice that we have to use US date formats in SQL.

The IN-clause also causes the computer to discard all dates that are not in the IN-list. Thus we don't need the WHERE-clause anymore.

If you try to look for the IN-clause in design view (the query grid), you won't find it. It has become a property of the query grid. To see it, right click inside the diagram area and select *Properties*.

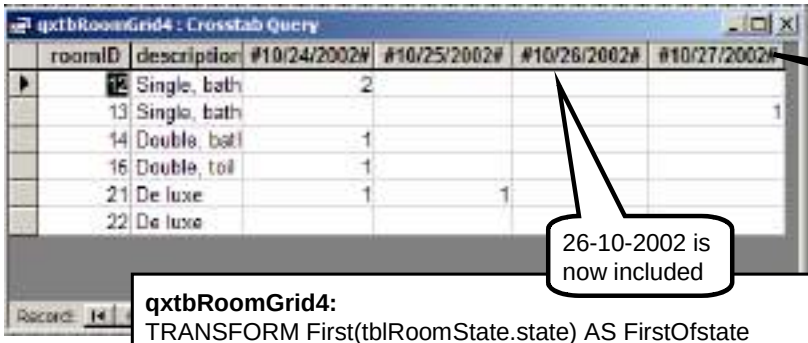
Unfortunately, the result looks ugly. The column headings have the programmer format, and I have not been able to find a way to change it. For other fields of the query, we can set field properties, for instance the display format. But not for the column headings. (I would call this a bug in Access.)

The IN-trick works okay if we use something else than dates for the headings. We will see an example now.

Using the query in a subform
In a good user interface, we wouldn't present the query directly to the user, but embed it in a subform. To make the column headings change dynamically, we make the same trick as in section 7.5.

We let the query generate column headings that are the texts C0, C1, C2 etc. They will bind to the controls C0, C1, etc. in the subform. The program will dynamically set the labels of these controls to the real dates.

Fig 7.6E Including all dates



Headings cannot be formatted
(Looks like an error in Access)

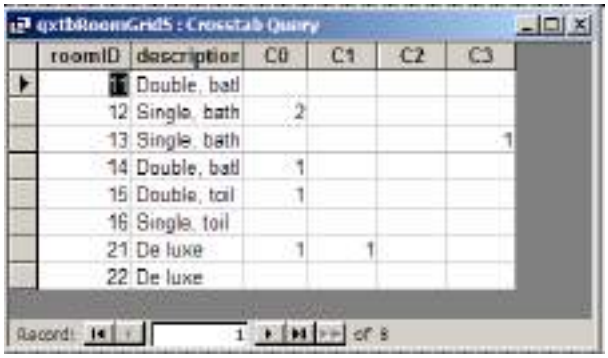
26-10-2002 is
now included

qxtbRoomGrid4:

```
TRANSFORM First(tblRoomState.state) AS FirstOfstate
SELECT tblRoom.roomID, tblRoomType.description
FROM tblRoomType INNER JOIN (tblRoom LEFT JOIN tblRoomState ON  . . . )
GROUP BY tblRoom.roomID, tblRoomType.description
PIVOT IIf(IsNull(date), #10/24/2002#, date)
IN (#10/24/2002#, #10/25/2002#, #10/26/2002#, #10/27/2002#);
```

Required column headings

In design view, this list appears as the
Column Heading property of the query



qxtbRoomGrid5:

```
TRANSFORM First(tblRoomState.state) AS FirstOfstate
SELECT . . .
PIVOT IIf(IsNull(date), "C0", "C" & (date-#10/24/2002#))
IN ("C0", "C1", "C2", "C3");
```

Column headings are used as
control names in the subform

The bottom of Figure 7.6E shows the query to be used. We compute the column heading with this expression:

```
IIf( IsNull(date), "C0", "C" & (date - #10/24/2002#))
```

Null dates are replaced by the text C0. Other dates are replaced by a C followed by the date index: 0 for the first day in the period, 1 for the next day, and so on.

Notice that although the IN-clause has the texts in quotes, the result shows the text in user format without quotes.

Crosstab restrictions

Crosstab is great for quick and dirty solutions where you want to show the relationship between two attributes in a query. This is for instance the case for experimental data mining where the user defines queries on his own.

However, when designing a production system such as a hotel system, Crosstab provides too little flexibility. Crosstab can for instance not handle subqueries or Having, even if they occur in named queries.

8. References

Jennings, Roger (1999): Using Microsoft Access 2000 - special edition. ISBN 0-7897-1606-2. QUE, Indianapolis (1300 pages). An advanced textbook for learning how to make Access applications. Only 200 of the pages use Visual Basic. Comprehensive lists of all icons and shortcut keys, all functions used in expressions, all date formats, etc. Also covers using Access for Web applications and combining Access with other MS-Office applications ("automation"). The index in this version is fine. (The 1997 version had a useless index.)

Litwin, Paul; Getz, Ken & Gilbert, Mike (1997): Access 97 - Developer's Handbook. Sybex, San Francisco. ISBN 0-7821-1941-7 (1500 pages). The professional developer's handbook for making Access applications with Visual Basic. Has a comprehensive chapter on SQL and the differences between Access SQL and standard SQL. Also covers how to combine Access with other MS-Office applications in a programmed fashion ("automation"), multi-user aspects, and configuration management. It is a book for looking up advanced topics, not a book to study from the beginning.

Index