

**NEAR EAST UNIVERSITY**

**Faculty of Engineering**

**Department of Computer Engineering**

**BOOK STORE AUTOMATION**

**Graduation Project  
COM-400**

**Student: Kağan UZUN (20020313)**

**Supervisor: Mr. Ümit İLHAN**

**Nicosia - 2008**

## ACKNOWLEDGEMENT

This project was prepared by Kağan Uzun, a senior student in the Near East University to cover the requirements of the senior project in the department of Computer Engineering.

First of I would like to thank to dean of faculty of engineering dear Prof. Dr. Fakhreddin Mamedov

Second, I like to thank to Mr. Ümit İlhan for supervising my Project and who lectured us the basics of this Project.

Third, I thank Mr. Okan Donangil and Dr. Kaan Uyar who were my advisors and were always beside me and shown me many helpful ideas that I will not forget in whole my life.

Fourth, I have not forget my friends; Armağan Özdemir, Gerçek Sezgin, Hakan Kılıç, Erkan Kalkancı for over whelming and help.

Also, I would like to special thank my father Mr. Ahmet Uzun and my mother Mrs. Atike Uzun and my darling Sema Dağ for everything





## ABSTRACT













The aim of this project is to register book store automation program that contain registration, all applications and also personel application. The program was prepared by using Delphi programming and using database.

This project consist of so many forms and menues. The main form of the arrive the others forms . Which are include information about the books, customers, employees.

First time i thing this program form my friend's books for help register. So this program is real life prepare to Yücel Book Store.

To show results show the efficiency of the program of book in program of the using in other chapters.

<b>ACKNOWLEDGEMENT.....</b>	<b>i</b>
<b>ABSTRACT.....</b>	<b>ii</b>
<b>TABLE OF CONTENTS.....</b>	<b>iii</b>
<b>INTRODUCTION.....</b>	<b>ix</b>
<b>CHAPTER ONE.....</b>	<b>1</b>
<b>1. DEFINITION OF DELPHI.....</b>	<b>1</b>
1.1 What is Delphi?.....	1
1.2.A Brief history of Borland's Delphi.....	2
1.2.1.Pascal.....	2
1.2.1.1.beginnings.....	2
1.2.1.2.The 1970's.....	2
1.2.1.3.The 1980's.....	2
1.2.2.From Turbo Pascal to Delphi.....	2
1.2.3. Delphi for Microsoft .Net.....	3
1.3.Standard tab GUI components.....	4
1.3.1 GUI components.....	4
1.3.2  Frame objects.....	5
1.3.3  Menus.....	5
1.3.4  Popup menus.....	6
1.3.5  Labels.....	7

1.3.6	 Edit boxes.....	7
1.3.7	 Memo boxes.....	7
1.3.8	 Buttons.....	7
1.3.9	 Check boxes.....	7
1.3.10	 Radio buttons.....	7
1.3.11	 List boxes.....	8
1.3.12	 Combo boxes.....	8
1.3.13	 Scroll bars.....	8
1.3.14	 Group boxes.....	8
1.3.15	 Radio group panels.....	9
1.3.16	 Empty panels.....	9
1.3.17	 Action lists.....	9
<b>CHAPTER TWO.....</b>		<b>10</b>
<b>2. ACCESS DATABASE.....</b>		<b>10</b>
2.1.What is Access?.....		10
2.1.1. Using Microsoft Access with Borland Delphi.....		12
2.2.Microsoft Access Database tables.....		13
2.3.Benefits of a Primary Key.....		18

2.3.1.Primary and Foreign Key constraints are and what they are used for.....	19
2.3.1.1.Primary Key.....	19
2.3.1.2.Foreign Key.....	19
2.4.What is a Key field in a Database and how should I choose one?.....	22
2.5.Import Excel Data into Microsoft Access.....	24
2.5.1.Creating a new table using an Excel file.....	24
2.5.2.Importing from Excel.....	25
<b>CHAPTER THREE.....</b>	<b>27</b>
<b>3. BASIC DOCUMENTS OF DELPHI.....</b>	<b>27</b>
3.1.Delphi data types.....	27
3.1.1.Storing data in computer programs.....	27
3.1.2.Simple delphi data types.....	27
3.1.2.1.Number types.....	28
3.1.2.2.Text types.....	29
3.1.2.3.Logical data types.....	29
3.1.2.4.Sets,enumerations and sub types.....	30
3.1.3.Using these simple data types.....	30
3.1.3.1.Assigning to and from variables.....	31
3.1.4.Compound data types.....	32
3.1.4.1.Arrays.....	32



3.1.4.2.Records.....	33
3.1.4.3.Objects.....	34
3.1.5.Other data types.....	34
3.1.5.1.Files.....	34
3.1.5.2.Pointers.....	35
3.1.5.3.Variants.....	35
3.1.6.Type definitions.....	35
3.2.Integer and Floating point numbers.....	35
3.2.1.The different number types in Delphi.....	35
3.2.2.Assigning to and from number variables.....	35
3.2.3.Numerical operators.....	36
3.2.4.Numeric functions and procedures.....	37
3.2.5.Converting from numbers to strings.....	37
3.2.6.Converting from strings to numbers.....	37
3.3.Strings and Characters.....	38
3.3.1.Text types.....	38
3.3.2.Characters.....	38
3.3.2.1.The Ansi character set.....	38
3.3.2.2.Assigning to and from character variables.....	42
3.3.2.3.What are Wide Char types?.....	43
3.3.3.Strings.....	43
3.3.3.1.Assigning to and from a string.....	43
3.3.3.2.String operators.....	44

3.3.3.3.String processing routines.....	45
3.3.3.4.Converting from numbers to strings.....	46
3.3.3.5.Converting from strings to numbers.....	47
3.4.Enumerations, SubRanges and Sets.....	47
3.4.1.Enumerations.....	47
3.4.1.1.Defining Enumerations.....	48
3.4.1.2.Using Enumerations numbers.....	49
3.4.1.3.A word of warning.....	49
3.4.2.SubRanges.....	49
3.4.3.Sets.....	50
3.4.3.1.What is a set.....	50
3.4.3.2.Including and Excluding set values.....	50
3.4.3.3.Set operators.....	51
3.5.Arrays.....	51
3.5.1.About arrays.....	51
3.5.2.Constant arrays.....	52
3.5.3.Different ways of defining array sizes.....	52
3.5.3.1.Using enumerations and subranges to define an array size.....	52
3.5.3.2.Using a data type.....	53
3.5.4.Static arrays.....	53
3.5.5.Dynamic arrays.....	53
3.5.6.Open arrays to routines.....	54
3.5.7.Multi-dimensional arrays.....	54



3.5.8.Copying arrays.....	54
3.6.Records.....	55
3.6.1.What are the records?.....	55
3.6.2.Using the with keyword.....	56
3.6.3.A more complex example.....	57
3.6.4.Packing record data.....	58
3.6.5.Records with variant parts.....	61
<b>CHAPTER FOUR.....</b>	<b>64</b>
<b>4. DESCRIPTION ABOUT PROJECT.....</b>	<b>64</b>
4.1.Password screen.....	64
4.2.Main menu.....	64
4.3.Customers form.....	65
4.4.Employees.....	66
4.5.Installment Paymen.....	67
4.6.Products.....	68
4.7.Sales And Installment Form.....	69
4.8.Payment form.....	70
4.9.Database relationship.....	72
<b>CONCLUSION.....</b>	<b>73</b>
<b>REFERENCES.....</b>	<b>74</b>
<b>APPENDIX .....</b>	<b>75</b>

## INTRODUCTION

This project is register book and company workers which uses ACCESS quarries. This program was prepared by using Borland Delphi 7 and ACCESS.

The subjects chapter by chapter so let us go through the overview the chapters in breif:

In the first Borland Delphi 7 programming language is described, its properties, components and some examples, I used Borland Delphi 6 in my project, because I find it easy and I liked its coding system. Borland Delphi 6 for applications

In the Second Chapter I described Database system, I used ACCESS data base system in my program with Borland Delphi 7.

Third Chapter is About the project , how we create it, its forms and using the program

Finally, the last chapter is the explanation of the program followed by the Appendices. So by developing and moderating of technology our program can be developed and updated. Also new properties could be added in to the program in the future.

## CHAPTER ONE

### 1.DEFINITION OF DELPHI

#### 1.1 what is Delphi?

Delphi (pronounced DEHL-FAI) from Borland competes with Visual Basic as an offering for an object-oriented, visual programming approach to application development. Based on object Pascal programming language, the latest version of Delphi includes facilities for rapidly building or converting an application into a Web service. It provides interfaces for the programmer to build an application using the Extensible Markup Language (XML), Extensible Stylesheet Language (XSL), Simple Object Access Protocol (SOAP), and Web Services Description Language (WSDL).

In ancient Greece, Delphi was the seat of the famous oracle that powerful people consulted for advice. When Borland's developers expanded their popular version of Pascal into an application builder with interfaces to databases such as Oracle, they chose Delphi as the code name for the project. News media and early users liked the name so it was marketed as Delphi. *Delphi* started life as the name of the visual IDE for Borland's Object Pascal programming language. In the old days, Borland used to try to make a distinction between 'Delphi the product' and 'Object Pascal the language'. However, most people used the name 'Delphi' indiscriminately to describe both the product and the language. On the whole, when people talked about 'programming in Delphi', they meant coding Object Pascal within the Delphi IDE. Delphi is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Based on Object Pascal, its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming

## 1.2.A Brief history of Borland's Delphi

### 1.2.1.Pascal

Delphi uses the language Pascal, a third generation structured language. It is what is called a **highly typed** language. This promotes a clean, consistent programming style, and, importantly, results in more reliable applications. Pascal has a considerable heritage.

#### 1.2.1.1.Beginnings

Pascal appeared relatively late in the history of programming languages. It probably benefited from this, learning from Fortran, Cobol and IBM's PL/1 that appeared in the early 1960's. Niklaus Wirth is claimed to have started developing Pascal in 1968, with a first implementation appearing on a CDC 6000 series computer in 1970.

Curiously enough, the C language did not appear until 1972. C sought to serve quite different needs to Pascal. C was designed as a high level language that still provided the low level access that assembly languages gave. Pascal was designed for the development of structured, maintainable applications.

#### 1.2.1.2.The 1970's

In 1975, Wirth teamed up with Jensen to produce the definitive Pascal reference book "Pascal User Manual and Report". Wirth moved on from Pascal in 1977 to work on **Modula** - the successor to Pascal.

#### 1.2.1.3.The 1980's

In 1982 ISO Pascal appears. The big event is in November 1983, when **Turbo Pascal** is released in a blaze of publicity. Turbo Pascal reaches release 4 by 1987. Turbo Pascal excelled on speed of compilation and execution, leaving the competition in its wake.

### 1.2.2.From Turbo Pascal to Delphi

Delphi, **Borland's** powerful **Windows?** and **Linux?** programming development tool first appeared in 1995. It derived from the **Turbo Pascal?** product line.



As the opposition took heed of Turbo Pascal, and caught up, Borland took a gamble on an Object Oriented version, mostly based on the Pascal object orientation extensions. The risk paid off, with a lot of the success due to the thought underlying the design of the IDE (Integrated Development Environment), and the retention of fast compilation and execution.

This first version of Delphi was somewhat limited when compared to today's heavyweights, but succeeded on the strength of what it did do. And speed was certainly a key factor. Delphi went through rapid changes through the 1990's.

### 1.2.3. Delphi for Microsoft .Net

From that first version, Delphi went through 7 further iterations before Borland decided to embrace the competition in the form of the **Microsoft? .Net** architecture with the stepping stone Delphi 8 and then fully with Delphi 2005 and 2006. Delphi however still remains, in the opinion of the author, the best development tool for stand alone Windows and Linux applications. Pascal is a cleaner and much more disciplined language than Basic, and adapted much better to Object Orientation than Basic.



## 1.3. Standard tab GUI components

### 1.3.1 GUI components

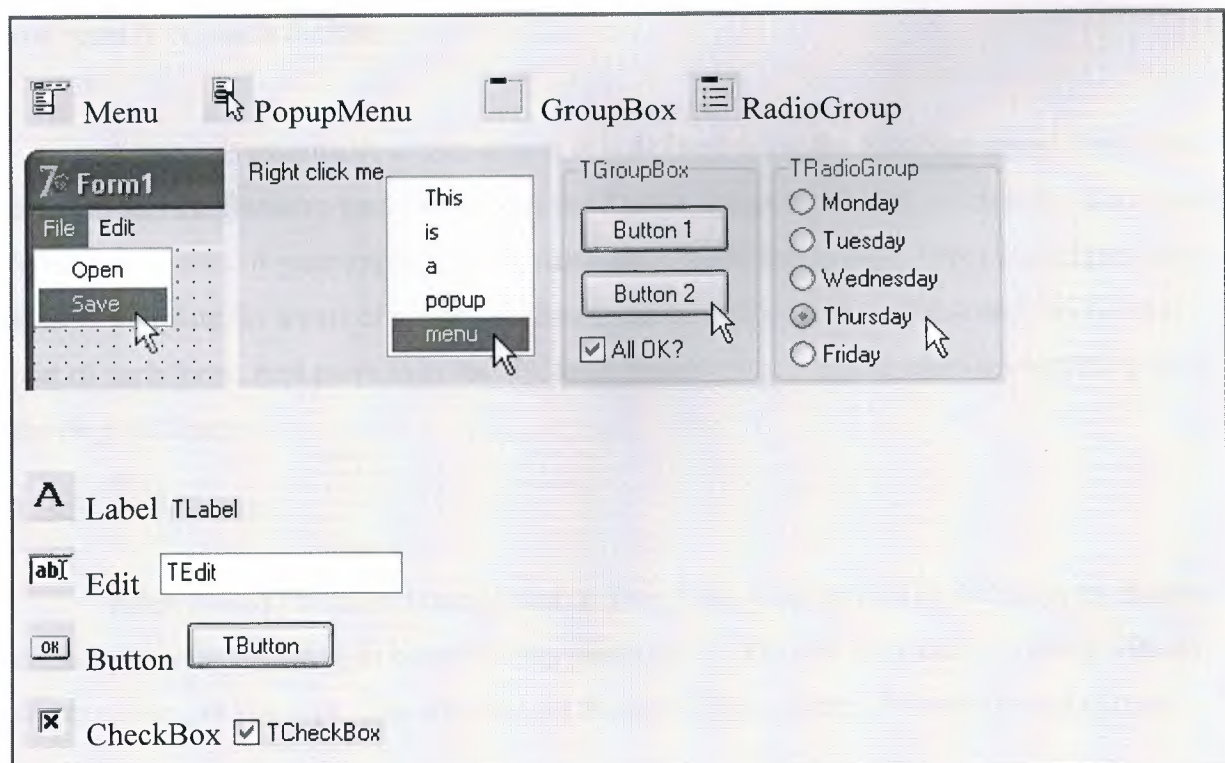
GUI stands for Graphical User Interface. It refers to the windows, buttons, dialogs, menus and everything visual in a modern application. A GUI component is one of these graphical building blocks. Delphi lets you build powerful applications using a rich variety of these components.

These components are grouped under a long set of tabs in the top part of the Delphi screen, starting with **Standard** at the left. We'll look at this Standard tab here. It looks something like this (Delphi allows you to tinker with nearly everything in its interface, so it may look different on your system):



Table: 1.3.1.1 GUI Components

Each of the components is itemised below with a picture of a typical GUI object they can create:



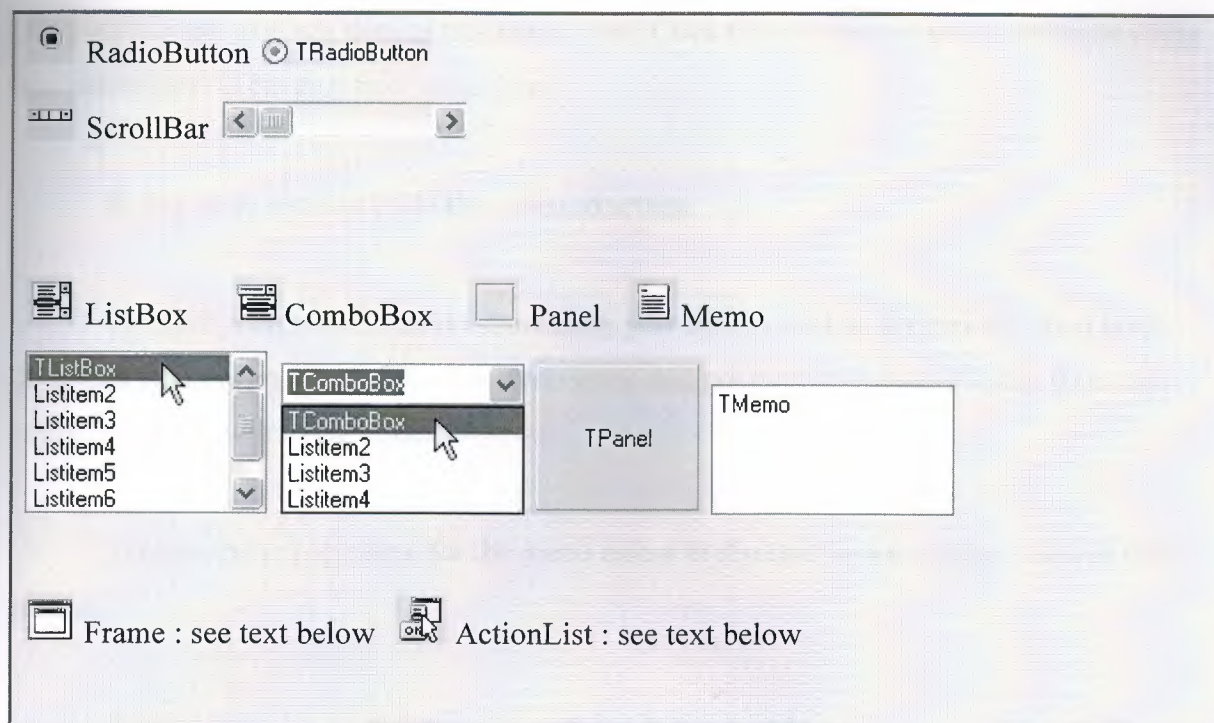


Table: 1.3.1.2 Radio Button

### 1.3.2 Frame objects

These were introduced in Delphi 5. They represent a powerful mechanism, albeit one that is a little advanced for a Delphi Basics site. However, it is worth describing their role if you want to research further.

A frame is essentially a new object. It is defined using the **File|New** menu. Only then can you add the frame to your form using the **Frame** component. You can add the same frame to as many forms of your application as you want. This is because the frame is designed as a kind of template for a part of a form. It allows you to define the same look and feel for that part of each form. And more importantly, each instance of the frame inherits everything from the original frame.

### 1.3.3 Menus

After you add a TMenu component to your form, you can design the menu by double clicking it (or using the right button popup menu for it). You are then shown a panel with an empty menu. As you type, you are creating the top left menu item. Press enter and you are

positioned at the first sub item of this menu item. Click the new empty box to the right of the first menu item to create a new menu item.

In this way, you can build the menu structure.

To make each menu item do something, just double click it. Delphi will then insert code into your program to handle the menu item, and position your cursor in the form unit ready for you to write your code.

Explore the popup menu for the menu editor to discover more options, such as sub-menus.

A menu can also be dynamically updated by your code.

#### 1.3.4 Popup menus

A popup menu appears in many applications when you right click on something. For example, when you right click the Windows desktop. You create a popup menu by adding the popup menu component to your form and double clicking it. You then simply type in your menu item list.

You attach the popup menu to an existing form object (or the form itself) by selecting your new popup menu in the **PopupMenu** property of the object.

To activate the popup menu items, double click each in turn. Delphi will add the appropriate code to your form unit. You can then type in the code that each menu item should perform.

A popup menu can also be dynamically updated by your code.



### 1.3.5 Labels

Labels are the simplest component. They are used to literally label things on a form, but the text, colours and so on can be changed by your code. For example, you can change the label colour when the mouse hovers over it, and can run code when the user clicks it. This makes the label like a web page link. Normally, they are just kept as plain, unchanging text.

### 1.3.6 Edit boxes

An edit box allows the user to type in a single line of text. For example, the name of the user. You set up the initial value with the **Text** property either at design time or when your code runs.

### 1.3.7 Memo boxes

A memo box displays a single string as a multi line wrapped text display. You cannot apply any formatting. The displayed lines are set using the **Lines** property. This may be set at design time as well as at run time.

### 1.3.8 Buttons

A button is the simplest active item. When clicked by a user, it performs some action. You can change the button label by setting the **Caption** property. Double clicking the button when designing adds code to your form to run when the button is clicked at run time.

### 1.3.9 Check boxes

Check boxes are used to give a user a **yes/no** choice. For example, whether to wrap text or not. The label is set using the **Caption** property. You can preset the check box to ticked by setting the **Checked** property to **true**.

### 1.3.10 Radio buttons

Radio buttons are used to give a user **multiple** choices. For example, whether to left, centre or right align text. The label is set using the **Caption** property. You can preset a radio button to selected by setting the **Checked** property to **true**.

You would normally use radio buttons in groups of two or more. The **TRadioGroup** component allows you to do this in a neat and dynamic way.

### 1.3.11 List boxes

List boxes provide selectable items. For example, a collection of fish names. If you set the **MultiSelect** property to **true**, you allow the user to select more than one. The items in the list are added using the **Items.Add** method, passing the string of each item as a parameter.

### 1.3.12 Combo boxes

A combo box is like a list box, and is set up in the same way (see above). It just takes up less space on your form by collapsing to a single line when deselected, showing the chosen list item. It is not recommend to use one for multi line selection.

### 1.3.13 Scroll bars

Many components have built in scroll bars. For those that don't, you can use this to do your own scrolling. You link the scrollbar to your component by setting the **OnScroll** event. This gives you the details of the last scroll activity made by the user.

### 1.3.14 Group boxes

A group box is like a panel. It differs in that it gives a name to the collection of components that you add to it. This title is set with the **Caption** property. Use a group box to help the user see what controls affect one particular aspect of the application.



### 1.3.15 Radio group panels

Radio buttons are used to give a user a **multiple** choices. For example, whether to left, centre or right align text. Unlike individual radio buttons, a group is only set up by your code. You define the buttons by calling the **Items.Add** method of the TRadioGroup object, passing the caption string of each radio button as a parameter. You can reference each button by using the **Buttons** indexed property. You might, for example, choose the third button to be checked.

### 1.3.16 Empty panels

When building your form, you might want to add many components. These may fall into logical groups. If so, you can add each group to a panel, and use the panel to position the whole group on the form. The panel name can be blanked out by setting the **Caption** property.

### 1.3.17 Action lists

Action lists are a large topic on their own. They allow you to define, for example, menus with sub-items that are also shown as buttons on your application. Only one **action** is defined, regardless of the number of references to it.

## CHAPTER TWO

### 2. ACCESS DATABASE

#### 2.1.what is Microsoft Access?

For anyone that has found him/herself under the gun who needs to consolidate, store, gather, isolate or manipulate information then report against it to a group of stakeholders, Microsoft Access is the application to use. It can be purchased as an add-on to the Microsoft Office Professional package. If you foresee yourself managing a neighborhood contact list or downloading information from a company's mainframe system, you can use Microsoft Access.

Upon opening Microsoft Access, there is the option to open a blank database or a number of templates. When you open the application, either using a template or a blank database, you will see a menu on the left side of the application which is the "Objects" menu. This menu will provide the category of functions that you need in order to use the database. The general concept of Access, using the "Object" menu is as follows:

- Tables: are used to store your raw data. For example, the administrator can upload information from Excel (\*.xls), Text (\*.txt), or non-Access database (\*.dbf) files.
- Queries: are used to manipulate the data in the tables. They can be used to add, update, or remove information from the tables. They can also be used to create tables. There are several types of queries that can be created. To add information, use an "append" query, to update information, use an "update" query, to remove information, use a "delete" query, to create tables, use a "make table" query.
- Forms: are used to enter information into the tables. In addition, they are designed for end-users to navigate the database.
- Reports: provide a layout in order to share the raw data and/or data analysis with others.
- Pages: provide an interface with the internet.
- Macros: are used to automate database tasks. They can be used to upload information into the database, automate functions within the database, to providing the end-user warning and/or informational messages.
- Modules: allow you to program the database using Visual Basic.

Across the top of the database window is another menu with the "Open", "Design", "New", "X", and several icon display options. These menu options are available regardless of which category chosen in the "Objects" menu. The menu across the top of the database window provides the following functions:

- Open: the database administrator can open any highlighted object (table, query, form, etc.)
- Design: allows the administrator to change the inherent/design functions of any database object. For example, if the administrator goes into the design view of a macro, s/he can change the operations of the macro.
- New: the programmer can create any new object within the database.
- X: deletes any object within the database.
- Icon displays: change how the objects appear within the database window. This function is similar to how Windows provides various display options.

As you delve into the world of Microsoft Access, there are wizards to assist with any function needed. Based upon what needs to be created, the questions will change. If the database is for personal use, department-wide use, or for use across the organization, ensure that the following foundation questions are answered prior to building the database:

- Why is the database needed? Is it to store information on a long-term basis or will the information be updated on a daily basis?
- Where are the information sources? Is it from a pre-existing spreadsheet or will it be entered manually? Who will need the information from the database?
- Who will be using the database? Are there many end-users or just one?
- Where will the database be housed? Will it be on a network server or a personal computer?
- When does the database need to be operational? Build testing time into the project plan.



### 2.1.1.Using Microsoft Access with Borland Delphi

Applications used:

- Borland Delphi 7
- Microsoft Access 2000

The article assumes that the reader knows how to create a basic **Microsoft Access database** and has some knowledge of **programming in Delphi**.

Microsoft Access is primarily used for developing stand alone applications. It is very fast, reliable and is very dependable when it comes to rapid application development. One of the benefits of Access from a developer's perspective is its relative compatibility with the structured query language (SQL). SQL is of course used to manipulate data within databases. So to develop efficient and dependable database applications in a rapid manner, we are going to need the advantages that MS Access offers.

So where does Borland Delphi fit in this scenario? Like Access, Delphi is also one of the frontrunners in its field. It is also a rapid application development language that offers easy application development. One of the reasons why I choose Delphi for this article is because it has a set of database components that integrates applications with MS Access' Jet Engine. And since we intend to build a stand alone application, Access and Delphi are perfectly place to handle this task. The aim of the application is simply to show how well MS Access and Delphi work together, among other things I will demonstrate how to use SQL to manipulate the data in the database. So let's start with building an Access database. For the sake of brevity, we are going to use Microsoft Access to create the database, but it is also possible to programmatically create an Access database with Delphi. Create a database called addressbook.mdb, and then create a table called contacts with the following columns:

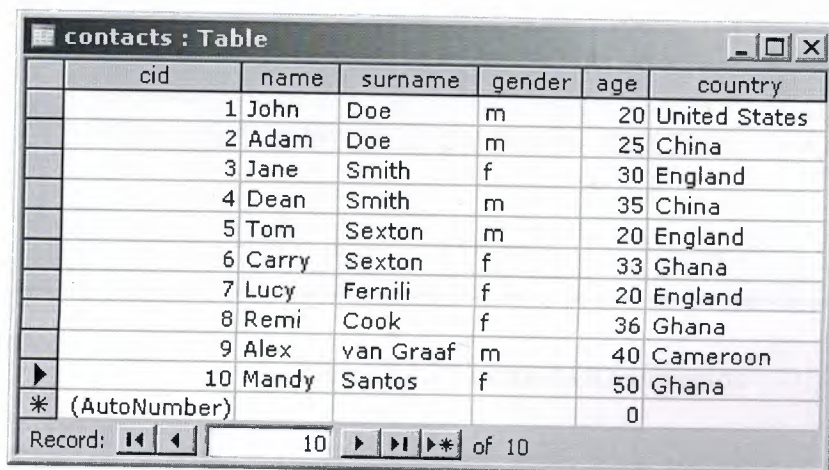
The Contacts table design in the Addressbook database

Column	Description	Type
--------	-------------	------

Cid	Contact ID	PrimaryKey, Autonumber
Name	Name of Contact	Text
surname	Surname of Contact	Text
Gender	Gender of the Contact	Text
Age	Age of Contact	Number
Country	Country of residence	Text

Add the following data in the table:

## 2.2 Microsoft Access Database Tables



	cid	name	surname	gender	age	country
	1	John	Doe	m	20	United States
	2	Adam	Doe	m	25	China
	3	Jane	Smith	f	30	England
	4	Dean	Smith	m	35	China
	5	Tom	Sexton	m	20	England
	6	Carry	Sexton	f	33	Ghana
	7	Lucy	Fernili	f	20	England
	8	Remi	Cook	f	36	Ghana
	9	Alex	van Graaf	m	40	Cameroon
	10	Mandy	Santos	f	50	Ghana
*	(AutoNumber)				0	

Record: 10 of 10

Table: 2.2.1 Contacts Table

That's all there is for the Access side of things. Next, we create the Delphi side of things:

Start a new application in Delphi and add the following components to the form:

- 1 Tmemo renamed qmemo
- 1 Dbgrid
- 1 buttons
- 1 Tedit renamed edparam



Then add the following components from the ADO tab:

- 1 ADOQuery rename to q1
- 1 ADOTable rename to ado1
- 1 AdoConnection
- 1 Datasource from the data access tab.

### Setting the component connections

Now select the adoconnection component and go to the object inspectors' connection string property. Click on the ellipses button. You should now see a window that looks like this:

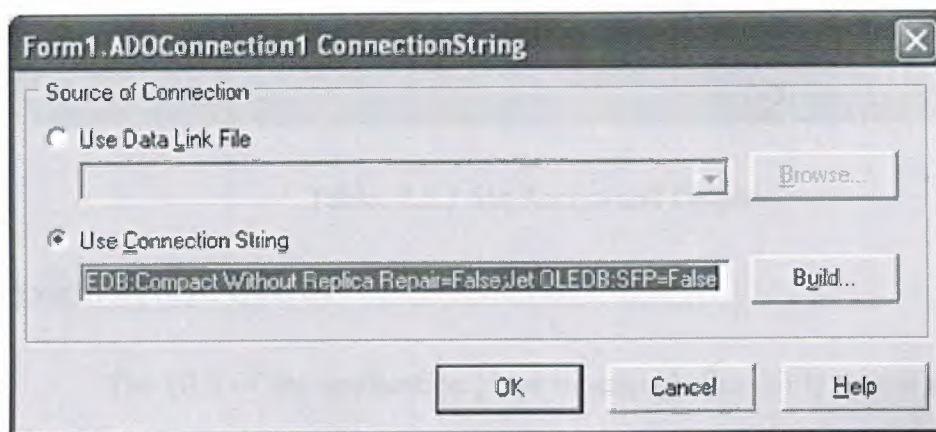


Table: 2.2.2 Form1.ADOconnection

Click on "build..." a window called "Data Link Properties" should come up. Click on the ellipses button and point to where the addressbook.mdb database is located. Then click OK twice and your connection should now be set.

Select the datasource component and go to its dataset property and add "q1" from the dropdown list.

Select the dbgrid component, go to its datasource property and add "datasource1" as its datasource.

That links up all the components, now all you have to do is to run the queries and all the data will be displayed in the dbgrid.

Your main form should look something like this at this point:

Form1

MS Access & Delphi

Query box:

select \* from contacts WHERE country=:c

Parameter Value:

Query

Table: 2.2.3 Ms Access and Delphi

### Executing SQL Queries

The GUI of the application gives us enough flexibility to run as many queries as we like. The above picture shows a sample query statement that is going to retrieve data that is based on a country criterion:

```
select * from contacts WHERE country=:c
```

if the country specified in “=:c” is England then the above query would produce the result:

select \* from contacts WHERE country=:c

Parameter Value:

England

Query

cid	name	surname	gender	age	country
3	Jane	Smith	f	30	England
5	Tom	Sexton	m	20	England
7	Lucy	Fernili	f	20	England

Table: 2.2.4 Executing SQL Queries

The “=:c” is what is called a parameter. It will contain the value that is to be included in the edparam text field (in the above the value is England). You can also just run a straight query that simply retrieves all of the records from the database:

```
select * from contacts
```

This produces:

The screenshot shows a database query interface. On the left, a text box contains the query 'select \* from contacts'. On the right, there is a 'Parameter Value:' section with an empty text field and a 'Query' button. Below these, a table displays the results of the query.

cid	name	surname	gender	age	country
4	Dean	Smith	m	35	China
5	Tom	Sexton	m	20	England
6	Carry	Sexton	f	33	Ghana
7	Lucy	Fernili	f	20	England

Table: 2.2.5 Paramater Value

Or one that retrieves all the records where the ages of the contacts are 20:

```
select * from contacts WHERE age=:c
```

The screenshot shows the same database query interface. The query text box now contains 'select \* from contacts WHERE age=:c'. The 'Parameter Value:' text field now contains the value '20'. The 'Query' button is still present. The resulting table shows only the records where the age is 20.

cid	name	surname	gender	age	country
1	John	Doe	m	20	United States
5	Tom	Sexton	m	20	England
7	Lucy	Fernili	f	20	England

Table: 2.2.6 Where age



Query number one and three are what are called parameterized or dynamic queries. A parameterized query is one that provides flexible row/column selection using a parameter in the WHERE clause of a SQL statement. The ADOQuery components' param property allows replaceable values to be stored for the query, as demonstrated in the queries above. To specify a parameter in a query, use a colon (:) preceding a parameter name, as in:

```
select * from contacts WHERE age=:c
```

All of the above queries are executed in the following procedure:

```
procedure TForm1.qbtnClick(Sender: TObject);
begin
  //close query component
  if q1.Active then begin
    q1.Close;
    q1.Parameters.ParamByName('c').Value:="";
    edparam.Clear;
  end;
  with q1 do
  begin
    SQL.Add(qmem.lines.text);
    if edparam.text <> " then begin
      Parameters.ParamByName('c').Value:=edparam.text;
    end;
    Open;
  end;
```

The procedure itself is very abstract in the sense that it does not allow you to write the queries. In other words, the queries are not hard coded. You can use any name of a contact or country and it will just work.

To make any changes to a contact's details is even easier. You simply call up the details using a query and then select the name of the contact and then make the changes that want. These changes will then be posted to the Access database.

So as you can see, the marriage of Access databases and Delphi's ADO components makes it a snap to write database applications with ease. Thanks to the on going improvements of MS Access, you can now also create a database application that is able to handle multiple user access at the same time.

## **2.3 Benefits of a Primary Key**

Have you ever placed an order with a company for the first time and then decided the next day to increase your order? You call the people at the order desk. Sometimes they ask you for your Customer Number. You tell them that you don't know your Customer Number. This happens all the time.....

So they ask you for some other personal information, generally your Postcode or telephone area code. Then, as they narrow down the list of customers, they will ask your name. Then, they will tell you your Customer Number. Some businesses use phone numbers as a unique starting point.

Database systems usually have more than one table, and these tend to be related in some manner. For example a Customer table and an Order table are related to each other via a unique Customer Number. The Customer table will always have one record for each Customer, and the Order table has one record for each Order that the Customer has made.

As each Customer is one physical person, you only need one record for the Customer in the Customer table. Each Customer can make several Orders, however, which means that you set up a table to hold information about each order (the Orders table). Each individual Order has one record in the Orders table.

Of course, you relate the Customers' Orders in the Orders table to the correct Customer in the Customer table by using a common field between both tables. In this example case, we would use the Customer Number (which is included in both tables).

When linking tables, we link the primary key field from one table (the Customer Number in the Customers table) to a field in the second (related) table that has the same structure and type of data in it (the Customer Number in the Orders table).



If the link in the second table is not the primary key field (and usually it isn't), it is known as the foreign key field.

Besides being a common link field between tables, a primary key field in Microsoft Access has the following advantages:

- A primary key field is an index that greatly speeds up queries, searches and sort requests.
- When you add new records, you must enter a value in the primary key field(s). Microsoft Access will not allow you to enter Null values, which guarantees that you will have only valid records in your table.
- When you add new records to a table that has a primary key, Microsoft Access checks for duplicate data and doesn't let you enter duplicates for the primary key field.
- By default, Access displays your data in the order of the primary key.

Primary key fields should be made as short as possible as this can affect the speed of operations in the database.

### 2.3.1 Primary and Foreign key constraints are and what they are used for

#### 2.3.1.1 Primary Key

A **primary key** is a field or combination of fields that uniquely identify a record in a table, so that an individual record can be located without confusion.

#### 2.3.1.2 Foreign Key

A **foreign key** (sometimes called a referencing key) is a key used to link two tables together. Typically you take the primary key field from one table and insert it into the other table where it becomes a foreign key (it remains a primary key in the original table).

More complicated but fuller explanation:

Employee Table

EmployeeID (PK)	FirstName	LastName	Department	Manager
-----------------	-----------	----------	------------	---------

001	Stan	Smithers	IT Support	Stan Smithers
002	Joe	Bloggs	Sales	Joe Bloggs
003	Mark	Richards	Sales	Joe Bloggs
004	Jenny	Lane	Marketing	Jenny Lane
005	Sally	Holmes	Sales	Joe Bloggs
006	John	Lee	IT Support	Stan Smithers

A primary key is the field(s) (a primary key can be made up of more than one field) that uniquely identifies each record, i.e. the primary key is unique for each record and the value is never duplicated in the same table, so in the above table the EmployeeID field would be used. A **constraint** is a rule that defines what data is valid for a given field. So a **primary key constraint** is a rule that says that the **primary key** fields cannot be null and cannot contain duplicate data.

The problem with the above table is that you have repeating information in the manager field, this causes all sorts of problems, e.g. Fred Bloggs leaves and Jenny Smith becomes sales manager, you now have to replace all entries that say Fred Bloggs with Jenny Smith.

If however you split the last two fields out to make a department table you would only need one entry for each department, when a manager changes you only need to make the change in one place, if you setup a primary key of DeptID in the department table you would have the following.

Department Table

DeptID (PK)	Department	Manager
01	IT Support	Stan Smithers
02	Sales	Joe Bloggs
03	Marketing	Jenny Lane

## Employee Table

EmployeeID (PK)	FirstName	LastName
001	Stan	Smithers
002	Joe	Bloggs
003	Mark	Richards
004	Jenny	Lane
005	Sally	Holmes
006	John	Lee

You now need to link the two table together so you know which department each employee is in, so what you do is take the primary key from the department table and insert it into the employee table (where it becomes a foreign key as a foreign key is the primary key from one table inserted into another table to link them).

## Employee Table

EmployeeID (PK)	FirstName	LastName	DeptID (FK)
001	Stan	Smithers	01
002	Joe	Bloggs	02
003	Mark	Richards	02
004	Jenny	Lane	03
005	Sally	Holmes	02
006	John	Lee	01

A **foreign key constraint** specifies that the data in a **foreign key** must match the data in the primary key of the linked table, in the above example we couldn't set the DeptID in the Employee table to 04 as there is no DeptID of 04 in the Department table. This system is called **referential integrity**, it is to ensure that the data entered is correct and not orphaned (i.e. there are no broken links between data in the tables)



The other added advantage is that you are saving space, if the following were the field sizes for the tables we have:

- EmployeeID = 3 characters
- Firstname = 10 characters
- Surname = 10 characters
- Department = 10 characters
- DeptID = 2 characters
- Manager = 20 characters

The original Employee Table would take 53 characters per record, 6 records gives us 318 characters.

The latest version of the Employee Table would take 25 characters, 6 records gives us 150 characters. The Department table would take 32 characters and there a 3 records so 96 characters, so  $150+96 = 246$  characters.

So over a very simple structure with just 6 records we have saved ourselves 72 characters, which would be 72 Bytes.

Doesn't sound much on 6 records but if we had 600 employees the original system would take  $53*600 = 31800$  characters. Whereas the new system would take  $25*600 = 15000 + 32*3 = 96$

Which is a total of 15096 characters, a saving of 16704 characters so we have saved over 50% of

## **2.4 What is a Key field in a Database and how should I choose one?**

Keys are crucial to a table structure for many reasons, some of which are identified below:

- They ensure that each record in a table is precisely identified.
- They help establish and enforce various types of integrity.
- They serve to establish table relationships.

Now let's see how you should choose your key(s). First, let's make up a little table to look at:



PersonID	LastName	FirstName	D.O.B
1	Smith	Robert	01/01/1970
2	Jones	Robert	01/01/1970
4	Smith	Henry	01/01/1970
5	Jones	Henry	01/01/1970

A **superkey** is a column or set of columns that uniquely identify a record. This table has many superkeys:

- PersonID
- PersonID + LastName
- PersonID + FirstName
- PersonID + DOB
- PersonID + LastName + FirstName
- PersonID + LastName + DOB
- PersonID + FirstName + DOB
- PersonID + LastName + FirstName + DOB
- LastName + FirstName + DOB

All of these will uniquely identify each record, so each one is a superkey. Of those keys, a key which is comprised of more than one column is a **composite key**; a key of only one column is a **simple key**.

A **candidate key** is a superkey that has no unique subset; it contains no columns that are not necessary to make it unique. This table has 2 candidate keys:

- PersonID
- LastName + FirstName + DOB

*Not all candidate keys make good primary keys:* Note that these may work for our current data set, but would likely be bad choices for future data. It is quite possible for two people to share a full name and date of birth.

We select a **primary key** from the candidate keys. This primary key will uniquely identify each record. It may or may not provide information about the record it identifies. It must not be Null-able, that is if it exists in a record it can not have the value Null. It must be unique. It **can not** be changed. Any candidate keys we do not select become **alternate keys**.

We will select (PersonID) as the primary key. This makes (LastName + FirstName + DOB) an alternate key.

Now, if this field *PersonID* is meaningful, that is it is used for any other purpose than making the record unique, it is a **natural key** or **intelligent key**. In this case PersonID is probably **not** an AutoNumber field, but is rather a "customer number" for use, much like the UPC or ISBN.

However, if this field is not meaningful, that is it is strictly for the database to internally identify a unique record, it is a **surrogate key** or **blind key**. In this case Person ID probably **is** an AutoNumber field, and it should not be used except internally by the database.

There is a long running debate over whether one should use natural or surrogate keys, and I'm not going to foolishly attempt to resolve it here. Whichever you use, stick with it. If you choose to generate an AutoNumber that is only used to identify a record, do **not** expose that number to the user. They will surely want to change it, and **you can not change primary keys**.

I can now use my chosen primary key in another table, to relate the two tables. It may or may not have the same name in that second table. In either case, with respect to the second table it is a **foreign key**, and if in that second table the foreign key field is not indexed it is a **fast foreign key**.

the storage space.

## 2.5 Import Excel Data into Microsoft Access

### 2.5.1 Creating a new table using an Excel file

As I became familiar with Access, I was very pleased to know that you could import existing information to use in Access. This saved retyping and editing time. In most cases, it

is easier to use Access to store, retrieve, manipulate, and report data than other applications. This article will explain, in a step-by-step format, how to create a table by importing an existing Excel file.

### **2.5.2. Importing from Excel (.xls)**

Prior to importing from Excel it is necessary to eliminate all of the formatting in the file. This includes, but is not limited to, centering, bold, underlining, etc. Also, ensure that there are no spaces in front of the column headings. By eliminating the spaces, you prevent importing errors and column heading issues so Access can recognize the headings. Below, you have the step-by-step guidelines for creating an Access table using an Excel file.

1. Save then close the .xls file.
2. Next, go to "New" on the "Tables" tab.
3. Click on "import table."
4. Change the "files of type" to Microsoft Excel.
5. Navigate to the "Excel" file that you would like to import.
6. Select the worksheet or named ranges you would like to import.
7. In the bottom half of the "Import Spreadsheet Wizard" dialog box, you will see the information that will be imported.
8. Click "Next."
9. If you have put the column headings in the Excel file, they are transferred to Access. If the first row of your data contains the column headings, check the box "First Row Contains Column Headings". If your first row of information does not contain column headings, do not check the box.
10. Click "Next". You will see that the first row of your data is grey and will be the column heading for your table.
11. To create a new table based on the data you are importing, click on "In a New Table". If you would like to import the information into an existing table, click on "In an Existing Table" and use the drop-down box to select the table name.

Please understand that you will be adding the information to an existing table daily. Please ensure that you want to do that. Instead, you will be taken straight to the last step of the wizard to click on "Finish."



12. Click "Next."
13. Select the indexing options for each field by clicking on the field heading to highlight the column then go to the "Indexed" field drop-down box and determine whether or not you would like to allow duplicate values (allows fields on different rows to contain the same information), to not allow duplicate values (fields on different rows must contain unique information), to not index the field.

Indexing the field allows the database to run faster. If you do not want to import a column, click on the column heading to highlight the entire column, then check "Do not import field (Skip)." You can adjust the data types after the table has been imported.

14. Click "Next."
15. At this point, decide if you want Access to add a primary key, if you want to choose your own primary key, or if you do not want a primary key. If Access creates a primary key, it will create an ID field which numbers your records. If you choose your own primary key, select a column that does not contain duplicate fields. For example, an invoice number field cannot contain invoice 12345678 twice in the same column.
16. Click "Next."
17. Type the name of your table into the "Import to Table:" field. If you do not want to overwrite an existing table, choose a name that is not currently being used.
18. Click "Finish."
19. Once you have successfully imported the file, a dialog box will display that reads: "Finished importing file [filepath] to table [table name]."



## CHAPTER THREE

### 3. BASIC DOCUMENTS OF DELPHI

#### 3.1.Delphi data types

##### 3.1.1.Storing data in computer programs

For those new to computer programming, data and code go hand in hand. You cannot write a program of any real value without lines of code, or without data. A Word Processor program has logic that takes what the user types and stores it in data. It also uses data to control how it stores and formats what the user types and clicks.

Data is stored in the memory of the computer when the program runs (it can also be stored in a file, but that is another matter beyond the scope of this tutorial). Each memory 'slot' is identified by a name that the programmer chooses. For example **LineTotal** might be used to name a memory slot that holds the total number of lines in a Word Processor document.

The program can freely read from and write to this memory slot. This kind of data is called a **Variable**. It can contain data such as a number or text. Sometimes, we may have data that we do not want to change. For example, the maximum number of lines that the Word Processor can handle. When we give a name to such data, we also give it its permanent value. These are called constants.

##### 3.1.2.Simple delphi data types

Like many modern languages, Delphi provides a rich variety of ways of storing data. We'll cover the basic, simple types here. Before we do, we'll show how to define a variable to Delphi:

```
var           // This starts a section of variables
  LineTotal : Integer; // This defines an Integer variable called LineTotal
  First,Second : String; // This defines two variables to hold strings of text
```

We'll show later exactly where this **var** section fits into your program. Notice that the variable definitions are indented - this makes the code easier to read - indicating that they are part of the **var** block.

Each variable starts with the name you choose, followed by a **:** and then the variable type. As with all Delphi statements, a **;** terminates the line. As you can see, you can define multiple variables in one line if they are of the same type.

It is very important that the name you choose for each variable is unique, otherwise Delphi will not know how to identify which you are referring to. It must also be different from the Delphi language keywords. You'll know when you have got it right when Delphi compiles your code OK (by hitting Ctrl-F9 to compile).

Delphi is not sensitive about the case (lower or upper) of your names. It treats **theCAT** name the same as **TheCat**.

### 3.1.2.1. Number types

Delphi provides many different data types for storing numbers. Your choice depends on the data you want to handle. Our Word Processor line count is an unsigned Integer, so we might choose **Word** which can hold values up to 65,535. Financial or mathematical calculations may require numbers with decimal places - floating point numbers.

**var**

// Integer data types :

```
Int1 : Byte; // 0 to 255
Int2 : ShortInt; // -127 to 127
Int3 : Word; // 0 to 65,535
Int4 : SmallInt; // -32,768 to 32,767
Int5 : LongWord; // 0 to 4,294,967,295
Int6 : Cardinal; // 0 to 4,294,967,295
Int7 : LongInt; // -2,147,483,648 to 2,147,483,647
Int8 : Integer; // -2,147,483,648 to 2,147,483,647
Int9 : Int64; // -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
```

// Decimal data types :

Dec1 : Single; // 7 significant digits, exponent -38 to +38

Dec2 : Currency; // 50+ significant digits, fixed 4 decimal places

Dec3 : Double; // 15 significant digits, exponent -308 to +308

Dec4 : Extended; // 19 significant digits, exponent -4932 to +4932

Some simple numerical variable usage examples are given below - fuller details on numbers is given in the Numbers tutorial.

### 3.1.2.2.Text types

Like many other languages, Delphi allows you to store letters, words, and sentences in single variables. These can be used to display, to hold user details and so on. A letter is stored in a single character variable type, such as **Char**, and words and sentences stored in string types, such as **String**.

**var**

Str1 : Char; // Holds a single character, small alphabet

Str2 : WideChar; // Holds a single character, International alphabet

Str3 : AnsiChar; // Holds a single character, small alphabet

Str4 : ShortString; // Holds a string of up to 255 Char's

Str5 : String; // Holds strings of Char's of any size desired

Str6 : AnsiString; // Holds strings of AnsiChar's any size desired

Str7 : WideString; // Holds strings of WideChar's of any size desired

Some simple text variable usage examples are given below - fuller details on strings and characters is given in the Text tutorial.

### 3.1.2.3.Logical data types

These are used in conjunction with programming logic. They are very simple:

**var**

Log1 : Boolean; // Can be 'True' or 'False'



Boolean variables are a form of **enumerated** type. This means that they can hold one of a fixed number of values, designated by name. Here, the values can be **True** or **False**. See the tutorials on Logic and Looping for further details.

#### 3.1.2.4.Sets,enumerations and sub types

Delphi excels in this area. Using sets and enumerations makes your code both easier to use and more reliable. They are used when categories of data are used. For example, you may have an enumeration of playing card suits. You literally enumerate the suit names. Before we can have an enumerated

variable, we must define the enumeration values. This is done in a **type** section.

**type**

```
TSuit = (Hearts, Diamonds, Clubs, Spades); // Defines the enumeration
```

**var**

```
suit : TSuit; // An enumeration variable
```

Sets are often confused with enumerations. The difference is tricky to understand. An enumeration variable can have only one of the enumerated values. A set can have none, 1, some, or all of the set values. Here, the set values are not named - they are simply indexed slots in a numeric range. Confused? Well, here is an example to try to help you out. It will introduce a bit of code a bit early, but it is important to understand.

**type**

```
TWeek = Set of 1..7; // Set comprising the days of the week, by number
```

**var**

```
week : TWeek;
```

**begin**

```
week := [1,2,3,4,5]; // Switch on the first 5 days of the week
```

**end;**

See the Set reference, and the Sets and enumerations tutorial for further details. That tutorial introduces a further data type - a **subrange** type.

#### 3.1.3.Using these simple data types

Variables can be read from and written to. This is called **assignment**. They can also be used in expressions and programming logic. See the Text tutorial and Programming logic





tutorial for more about these topics.

### 3.1.3.1. Assigning to and from variables

Variables can be assigned from constant values, such as **23** and **'My Name'**, and also from other variables. The code below illustrates this assignment, and also introduces a further section of a Delphi program : the **const** (constants) section. This allows the programmer to give names to constant values. This is useful where the same constant is used throughout a program - a change where the constant is defined can have a global effect on the program.

Note that we use upper case letters to identify constants. This is just a convention, since Delphi is not case sensitive with names (it is with strings). Note also that we use **=** to define a constant value.

#### types

```
TWeek = 1..7;           // Set comprising the days of the week, by number
TSuit = (Hearts, Diamonds, Clubs, Spades); // Defines an enumeration
```

#### const

```
FRED      = 'Fred';    // String constant
YOUNG_AGE  = 23;        // Integer constant
TALL : Single = 196.9;  // Decimal constant
NO        = False;     // Boolean constant
```

#### var

```
FirstName, SecondName : String; // String variables
Age                   : Byte;    // Integer variable
Height               : Single;  // Decimal variable
IsTall                : Boolean; // Boolean variable
OtherName             : String; // String variable
Week                  : TWeek;   // A set variable
Suit                  : TSuit;   // An enumeration variable
```

**begin** // Begin starts a block of code statements

```
FirstName := FRED; // Assign from predefined constant
```

```

SecondName := 'Bloggs';    // Assign from a literal constant
Age      := YOUNG_AGE;    // Assign from predefined constant
Age      := 55;           // Assign from constant - overrides YOUNG_AGE
Height   := TALL - 5.5;   // Assign from a mix of constants
IsTall   := NO;           // Assign from predefined constant
OtherName := FirstName;   // Assign from another variable
Week     := [1,2,3,4,5];  // Switch on the first 5 days of the week
Suit     := Diamonds;     // Assign to an enumerated variable
end;    // End finishes a block of code statements

```

FirstName is now set to **'Fred'**

SecondName is now set to **'Bloggs'**

Age is now set to **55**

Height is now set to **191.4**

IsTall is now set to **False**

OtherName is now set to **'Fred'**

Week is now set to **1,2,3,4,5**

Suit is now set to **Diamonds** (Notice no quotes)

Note that the third constant, **TALL**, is defined as a Single type. This is called a **typed constant**. It allows you to force Delphi to use a type for the constant that suits your need. Otherwise, it will make the decision itself.

### 3.1.4.Compound data types

The simple data types are like single elements. Delphi provides compound data types, comprising collections of simple data types.

These allow programmers to group together variables, and treat this group as a single variable. When we discuss programming logic, you will see how useful this can be.

#### 3.1.4.1.Arrays

Array collections are accessed by index. An array holds data in indexed 'slots'. Each slot holds one variable of data. You can visualise them as lists. For example:

**var**

```
Suits : array[1..4] of String; // A list of 4 playing card suit names
```

**begin**

```
Suits[1] := 'Hearts'; // Assigning to array index 1
```

```
Suits[2] := 'Diamonds'; // Assigning to array index 2
```

```
Suits[3] := 'Clubs'; // Assigning to array index 3
```

```
Suits[4] := 'Spades'; // Assigning to array index 4
```

**end;**

The array defined above has indexes 1 to 4 (1..4). The two dots indicate a range. We have told Delphi that the array elements will be string variables. We could equally have defined integers or decimals.

For more on arrays, see the Arrays tutorial.

### 3.1.4.2. Records

Records are like arrays in that they hold collections of data. However, records can hold a mixture of data types. They are a very powerful and useful feature of Delphi, and one that distinguishes Delphi from many other languages.

Normally, you will define your own record structure. This definition is not itself a variable. It is called a data **type** (see Types for further on this). It is defined in a **type** data section. By convention, the record type starts with a **T** to indicate that it is a type not real data (types are like templates). Let us define a customer record:

**type**

```
TCustomer Record
```

```
  firstName : string[20];
```

```
  lastName  : string[20];
```

```
  age       : byte;
```



**end;**

Note that the strings are suffixed with **[20]**. This tells Delphi to make a fixed space for them. Since strings can be a variable length, we must tell Delphi so that it can make a record of known size. Records of one type always take up the same memory space.

Let us create a record variable from this record type and assign to it:

**var**

customer : TCustomer;       // Our customer variable

**begin**

customer.firstName := 'Fred';   // Assigning to the customer record

customer.lastName := 'Bloggs';

customer.age := 55;

**end;**

customer.firstName is now set to **'Fred'**

customer.lastName is now set to **'Bloggs'**

customer.age is now set to **55**

Notice how we do not use an index to refer to the record elements. Records are very friendly - we use the record element by its name, separated from the record name by a qualifying dot. See the Records tutorial for further on records.

### 3.1.4.3.Objects

Objects are collections of both data and logic. They are like programs, but also like data structures. They are the key part of the **Object oriented** nature of Delphi. See the Object orientation tutorial for more on this advanced topic.

### 3.1.5.Other data types

The remaining main object types in Delphi are a mixed bunch:

#### 3.1.5.1.Files



File variables represent computer disk files. You can read from and write to these files using file access routines. This is a complex topic covered in Files.

### **3.1.5.2.Pointers**

Pointers are also the subject of an advanced topic - see Pointer reference. They allow variables to be indirectly referenced.

### **3.1.5.3.Variants**

Variants are also an advanced topic - see Variant. They allow the normal Delphi rigid type handling to be avoided. Use with care!

### **3.1.6.Type definitions**

When we discussed Records above, we introduced the concept of **types**. Delphi has many predefined data types - both simple, such as **string**, and compound, such as TPoint (which holds X and Y coordinates of a point). See Type for further details.

## **3.2.Integer and Floating point numbers**

### **3.2.1.The different number types in Delphi**

Delphi provides many different data types for storing numbers. Your choice depends on the data you want to handle. In general, smaller number capacities mean smaller variable sizes, and faster calculations. Ideally, you should use a type that comfortably copes with all possible values of the data it will store.

### **3.2.2.Assigning to and from number variables**

Number variables can be assigned from constants, other numeric variables, and expressions:

**const**

```
YOUNG_AGE = 23;    // Small integer constant
```

```
MANY    = 300;    // Bigger integer constant
RICH     = 100000.00; // Decimal number : note no thousand commas
```

**var**

```
Age      : Byte;    // Smallest positive integer type
Books    : SmallInt; // Bigger signed integer
Salary   : Currency; // Decimal used to hold financial amounts
```

```
Expenses : Currency;
```

```
TakeHome : Currency;
```

**begin**

```
Age      := YOUNG_AGE; // Assign from a predefined constant
Books    := MANY + 45; // Assign from a mix of constants (expression)
Salary   := RICH;      // Assign from a predefined constant
Expenses := 12345.67; // Assign from a literal constant
TakeHome := Salary;    // Assign from another variable
TakeHome := TakeHome - Expenses; // Assign from an expression
```

**end;**

Age is set to **23**

Books is set to **345**

Salary is set to **100000.00**

Expenses is set to **12345.67**

TakeHome is set to **87654.33**

### **3.2.3.Numerical operators**

Number calculations, or expressions, have a number of primitive operators available:

**+** : Add one number to another

**-** : Subtract one number from another

**\*** : Multiply two numbers

**/** : Divide one decimal number by another

**div** : Divide one integer number by another

**mod** : Remainder from dividing one integer by another

### 3.2.4.Numeric functions and procedures

Delphi provides many builtin functions and procedures that can perform numeric calculations. Some examples are given below - click on any to discover more. Note that these routines are stored in **Units** that are shipped with Delphi, and which form part of the standard delphi Run Time Library. You will need to include a reference to the Unit in order to use it (the code example provided with each gives the unit name and shows how to refer to it).

Abs Returns the absolute value of a signed number

Max Gives the maximum of two integer values

Min Gives the minimum of two integer values

Mean Gives the average of a set of numbers

Sqr Gives the square of a number

Sqrt Gives the square root of a number

Exp Gives the exponent of a number

Shl Shifts the bits in a number left

Shr Shifts the bits in a number right

Tan Gives the Tangent of a number

Cos Gives the Cosine of a number

Sin Gives the Sine of a number

### 3.2.5.Converting from numbers to strings

Delphi also provides routines that convert numbers into strings. This is often useful for display purposes.

Str Converts a number to a string in a simple manner

CurrToStr Converts a Currency variable to a string

Format Number to string conversion with formatting

IntToStr Converts an integer to a string

IntToHex Converts a number into a hexadecimal string

### 3.2.6.Converting from strings to numbers

Finally, Delphi provides string to number conversion utilities. Here are some examples:



**StrToInt** Converts an integer string into an integer

**StrToIntDef** Fault tolerant version of **StrToInt**

**StrToFloat** Converts a decimal string to a number

### 3.3.Strings and Characters

#### 3.3.1.Text types

Like many other languages, Delphi allows you to store letters, words, and sentences in single variables. These can be used to store and display such things as user details, screen titles and so on. A letter is stored in a single character variable type, such as **Char**, and words and sentences stored in string types, such as **String**.

#### 3.3.2.Characters

Single character variables hold a single character of text. Normally, this can be held in one byte. **AnsiChar** types are exactly one byte in size, and can hold any of the characters in the Ansi character set.

##### 3.3.2.1.The Ansi character set

#### Char Code Description

- 9 Tab
- 10 Line feed
- 13 Carriage return
- ' ' 32 Space
- ! 33 Exclamation mark
- " 34 Quotation mark
- # 35 Number sign
- \$ 36 Dollar sign
- % 37 Percent sign
- & 38 Ampersand
- ' 39 Apostrophe
- ( 40 Left parenthesis
- ) 41 Right parenthesis
- \* 42 Asterisk

+ 43 Plus sign  
, 44 Comma  
- 45 Hyphen-minus  
. 46 Full stop  
/ 47 Solidus  
0 48 Digit zero  
1 49 Digit one  
2 50 Digit two  
3 51 Digit three  
4 52 Digit four  
5 53 Digit five  
6 54 Digit six  
7 55 Digit seven  
8 56 Digit eight  
S 83 Latin capital letter S  
T 84 Latin capital letter T  
U 85 Latin capital letter U  
V 86 Latin capital letter V 9 57 Digit nine  
: 58 Colon  
; 59 Semicolon  
< 60 Less-than sign  
= 61 Equals sign  
> 62 Greater-than sign  
? 63 Question mark  
@ 64 Commercial at  
A 65 Latin capital letter A  
B 66 Latin capital letter B  
C 67 Latin capital letter C  
D 68 Latin capital letter D  
E 69 Latin capital letter E  
F 70 Latin capital letter F  
G 71 Latin capital letter G  
H 72 Latin capital letter H  
I 73 Latin capital letter I

J 74 Latin capital letter J  
K 75 Latin capital letter K  
L 76 Latin capital letter L  
M 77 Latin capital letter M  
N 78 Latin capital letter N  
O 79 Latin capital letter O  
P 80 Latin capital letter P  
Q 81 Latin capital letter Q  
R 82 Latin capital letter R

W 87 Latin capital letter W  
X 88 Latin capital letter X  
Y 89 Latin capital letter Y  
Z 90 Latin capital letter Z

[ 91 Left square bracket  
\ 92 Reverse solidus  
] 93 Right square bracket  
^ 94 Circumflex accent  
\_ 95 Low line  
` 96 Grave accent

a 97 Latin small letter a  
b 98 Latin small letter b  
c 99 Latin small letter c  
d 100 Latin small letter d  
e 101 Latin small letter e  
f 102 Latin small letter f  
g 103 Latin small letter g  
h 104 Latin small letter h  
i 105 Latin small letter i  
j 106 Latin small letter j  
k 107 Latin small letter k  
l 108 Latin small letter l  
m 109 Latin small letter m  
n 110 Latin small letter n



- o 111 Latin small letter o
- p 112 Latin small letter p
- q 113 Latin small letter q
- r 114 Latin small letter r
- s 115 Latin small letter s
- t 116 Latin small letter t
- u 117 Latin small letter u
- v 118 Latin small letter v
- w 119 Latin small letter w
- x 120 Latin small letter x
- y 121 Latin small letter y
- z 122 Latin small letter z
- { 123 left curly bracket
- | 124 Vertical line
- } 125 Right curly bracket
- ~ 126 Tilde
- 127 (not used)
- ? 128 Euro sign Currency Symbols
- ? 129 (not used)
- ? 130 Single low-9 quotation mark General Punctuation
- ? 131 Latin small letter f with hook Latin Extended-B
- ? 132 Double low-9 quotation mark General Punctuation
- ? 133 Horizontal ellipsis General Punctuation
- ? 134 Dagger General Punctuation
- ? 135 Double dagger General Punctuation
- ? 136 Modifier letter circumflex accent Spacing Modifier Letters
- ? 137 Per mille sign General Punctuation
- ? 138 Latin capital letter S with caron Latin Extended-A
- ? 139 Single left-pointing angle quotation mark General Punctuation
- ? 140 Latin capital ligature OE Latin Extended-A
- ? 141 (not used)
- ? 142 Latin capital letter Z with caron Latin Extended-A
- ? 143 (not used)
- ? 144 (not used)

- ? 145 Left single quotation mark General Punctuation
- ? 146 Right single quotation mark General Punctuation
- ? 147 Left double quotation mark General Punctuation
- ? 148 Right double quotation mark General Punctuation
- ? 149 Bullet General Punctuation
- ? 150 En dash General Punctuation
- ? 151 Em dash General Punctuation
- ? 152 Small tilde Spacing Modifier Letters
- ? 153 Trade mark sign Letterlike Symbols
- ? 154 Latin small letter s with caron Latin Extended-A
- ? 155 Single right-pointing angle quotation mark General Punctuation
- ? 156 Latin small ligature oe Latin Extended-A
- ? 157 (not used)
- ? 158 Latin small letter z with caron Latin Extended-A
- ? 159 Latin capital letter Y with diaeresis Latin Extended-A
- 160 No-break space
- ? 161 Inverted exclamation mark
- ? 162 Cent sign
- ? 163 Pound sign
- ? 164 Currency sign
- ? 165 Yen sign

### 3.3.2.2. Assigning to and from character variables

Here are some examples of characters, along with assignments to and from them:

**var**

lower, upper, copied, fromNum : AnsiChar;

**begin**

lower := 'a'; // Assign a lower case letter

upper := 'Q'; // Assign an upper case letter

copied := lower; // Assign from another character variable

fromNum := Chr(65); // Assign using a function

**end;**

```
var
```

```
    myNum : Byte;
```

```
begin
```

```
    myNum := Ord('A'); // myNum is set to 65
```

```
end;
```

### 3.3.2.3. What are Wide Char types?

The ansi character set derived from the earlier ascii character set. Both were designed around European characters, which comfortably fitted into 256 values, the capacity of a single byte. For a long time, this was the easy way to handle text. But this left many countries, especially in Asia, out of the picture.

The WideChar type can support double-byte characters, which can hold numeric representations of the vast alphabets of China, Japan and so on. These are called International characters. International applications must use WideChar and WideString types.

### 3.3.3. Strings

A single character is useful when parsing text, one character at a time. However, to handle words and sentences and screen labels and so on, strings are used. A string is literally a string of characters. It can be a string of Char, AnsiChar or WideChar characters.

#### 3.3.3.1. Assigning to and from a string

A **ShortString** is a fixed 255 characters long. A **String** (by default) is the same as an **AnsiString**, and is of any length you want. WideStrings can also be of any length. Their storage is dynamically handled. In fact, if you copy one string to another, the second will just point to the contents of the first.



Here are some assignments:

```
var
    source, target, last : String;
begin
    source := 'Hello World';    // Assign from a string literal
    target := source;          // Assign from another variable
    last := 'Don't do that';    // Quotes in a string must be doubled
end;
```

. source is now set to : Hello World

target is now set to : Hello World

last is now set to : Don't do that

### 3.3.3.2.String operators

There are a number of primitive string operators that are commonly used:

- + Concatenates two strings together
- = Compares for string equality
- < Is one string lower in sequence than another
- <= Is one string lower or equal in sequence with another
- > Is one string greater in sequence than another
- >= Is one string greater or equal in sequence with another
- <> Compares for string inequality

Here are some examples using these operators:

```
var
    myString : string;
begin
    myString := 'Hello ' + 'World'; // String concatenation
```

```

if 'ABC' = 'abc'           // Equality
then ShowMessage('ABC = abc');
if 'ABC' = 'ABC'           // Equality
then ShowMessage('ABC = ABC');
if 'ABC' < 'abc'           // Less than
then ShowMessage('ABC < abc');
if 'ABC' <= 'abc'          // Less than or equal
then ShowMessage('ABC <= abc');
if 'ABC' > 'abc'           // Greater than
then ShowMessage('ABC > abc');
if 'ABC' >= 'abc'          // Greater than or equal
then ShowMessage('ABC >= abc');
if 'ABC' <> 'abc'           // Inequality
then ShowMessage('ABC <> abc');
end;

```

ABC = ABC

ABC < abc

ABC <= abc

ABC <> abc

### 3.3.3.3.String processing routines

There are a number of string manipulation routines that are given by example below. Click on any of them to learn more (and also click on WrapText for another, more involved routine).

**var**

Source, Target : string;

**begin**

Source := '12345678';

Target := Copy(Source, 3, 4); // Target now = '3456'

Target := '12345678';

Insert('-', Target, 3); // Target now = '12-+-345678'

```

Target := '12345678';
Delete(Target, 3, 4);      // Target now = '1278'

Target := StringOfChar('S', 5); // Target now = 'SSSSS'

Source := 'This is a way to live A big life';

// Target set to 'This is THE way to live THE big life'
Target := StringReplace(before, ' a ', ' THE ',
                        [rfReplaceAll, rfIgnoreCase]);

end;

```

**AnsiLeftStr**     Returns leftmost characters of a string  
**AnsiMidStr**      Returns middle characters of a string  
**AnsiRightStr**    Returns rightmost

characters of a string

**AnsiStartsStr**   Does a string start with a substring?  
**AnsiContainsStr** Does a string contain another?  
**AnsiEndsStr**     Does a string end with a substring?  
**AnsiIndexStr**    Check substring list against a string  
**AnsiMatchStr**    Check substring list against a string  
**AnsiReverseString** Reverses characters in a string  
**AnsiReplacStr**   Replaces all substring occurrences  
**DupeString**       Repeats a substring n times  
**StrScan**          Scans a string for a specific character  
**StuffString**      Replaces part of a string text  
**Trim**             Removes leading and trailing white space  
**TrimLeft**        Removes leading white space  
**TrimRight**       Removes trailing white space

### 3.3.3.4. Converting from numbers to strings

**CurrToStrF**      Convert a currency value to a string with formatting  
**DateTimeToStr**   Converts TDateTime date and time values to a string



DateTimeToString	Rich formatting of a TDateTime variable into a string
DateToStr	Converts a TDateTime date value to a string
FloatToStr	Convert a floating point value to a string
FloatToStrF	Convert a floating point value to a string with formatting
Format	Rich formatting of numbers and text into a string
FormatCurr	Rich formatting of a currency value into a string
FormatDateTime	Rich formatting of a TDateTime variable into a string
FormatFloat	Rich formatting of a floating point number into a string
IntToHex	Convert an Integer into a hexadecimal string
IntToStr	Convert an integer into a string
Str	Converts an integer or floating point number to a string

### 3.3.3.5. Converting from strings to numbers

StringToWideChar	Converts a string into a WideChar 0 terminated buffer
StrToCurr	Convert a number string into a currency value
StrToDate	Converts a date string into a TDateTime value
StrToDateTime	Converts a date+time string into a TDateTime value
StrToFloat	Convert a number string into a floating point value
StrToInt	Convert an integer string into an Integer value
StrToInt64	Convert an integer string into an Int64 value
StrToInt64Def	Convert a string into an Int64 value with default
StrToIntDef	Convert a string into an Integer value with default
StrToTime	Converts a time string into a TDateTime value
Val	Converts number strings to integer and floating point values

## 3.4. Enumerations, SubRanges and Sets

### 3.4.1. Enumerations

The provision of enumerations is a big plus for Delphi. They make for readable and reliable code. An enumeration is simply a fixed range of named values. For example, the **Boolean** data type is itself an enumeration, with two possible values : **True** and **False**. If you

try to assign a different value to a boolean variable, the code will not compile.

### 3.4.1.1. Defining Enumerations

When you want to use an enumeration variable, you must define the range of possible values in an enumeration **type** first (or use an existing enumeration type, such as boolean).

Here is an example:

**type**

```
TSuit = (Hearts, Diamonds, Clubs, Spades); // Defines enumeration range
```

**var**

```
suit : TSuit; // Defines enumeration variable
```

**begin**

```
suit := Clubs; // Set to one of the values
```

**end;**

The **TSuit** type definition creates a new Delphi data type that we can use as a type for any new variable in our program. (If you define types that you will use many times, you can place them in a **Unit** file and refer to this in a uses statement in any program that wants to use them). We have defined an enumeration range of names that represent the suits of playing cards.

We have also defined a **suit** variable of that TSuit type, and have assigned one of these values. Note that there are no quote marks around these enumeration values - they are not strings, and they take no storage.

In fact, each of the enumeration values is equated with a number. The TSuit enumeration will have the following values assigned :

**Hearts = 0 , Diamonds = 1 , Clubs = 2 , Spades = 3**

### 3.4.1.2.Using Enumerations numbers

Since enumeration variables and values can also be treated as numbers (ordinals), we can use them in expressions :

**type**

```
TDay = (Mon=1, Tue, Wed, Thu, Fri, Sat, Sun); // Enumeration values
```

**var**

```
today : TDay;
```

```
weekend : Boolean;
```

**begin**

```
today := Wed;      // Set today to be Wednesday
```

```
if today > Fri      // Ask if it is a weekend day
```

```
  then weekend := true
```

```
  else weekend := false;
```

**end;**

today is set to **Wed** which has ordinal value = 3

weekend is set to **false** since Wed (3) <= Fri (5)

### 3.4.1.3.A word of warning

One word of warning : each of the values in an enumeration must be unique in a program. This restriction allows you to assign an enumeration value without having to qualify the type it is defined in.

### 3.4.2.SubRanges

SubRange data types take a bit of getting used to, although they are simple in principle. With the standard ordinal (integer and character) types you are allowed a finite range of values. For example, for the **byte** type, this is **0** to **255**. SubRanges allow you to define your own type with a reduced range of values.





### 3.4.3.Sets

#### 3.4.3.1.What is a set?

Sets are another way in which Delphi is set apart from other languages. Whereas enumerations allow a variable to have one, and only one, value from a fixed number of values, sets allow you to have any combination of the given values - none, 1, some, or all.

A set variable therefore holds a set of indicators. Up to 255 indicators. An indicator is set on when the variable has that value defined. This may be a bit tricky to understand, so here is an example:

**type**

```
TDigits = set of '1'..'9';    // Set of numeric digit characters
```

**var**

```
digits : TDigits;           // Set variable
```

```
myChar : char;
```

**begin**

```
// At the start, digits has all set values switched off
```

```
// So let us switch some on. Notice how we can switch on single
```

```
// values, and ranges, all in the one assignment:
```

```
digits := ['2', '4'..'7'];
```

```
// Now we can test to see what we have set on:
```

```
for myChar := '1' to '9' do
```

```
  if myChar In digits
```

```
    then ShowMessageFmt("%s" is in digits',[myChar])
```

```
    else ShowMessageFmt("%s" is not in digits',[myChar])
```

```
end;
```

#### 3.4.3.2.Including and Excluding set values

Notice in the code above that we assigned (switched on) a set of values in a set variable.

Delphi provides a couple of routines that allow you to **include** (switch on) or **exclude** (switch



off) individual values without affecting other values:

**type**

// We define a set by type - bytes have the range : 0 to 255

TNums = set of Byte;

**var**

nums : TNums;

**begin**

nums := [20..50]; // Switch on a range of 31 values

Include(nums, 12); // Switch on an additional value : 12

Exclude(nums, 35); // Switch off a value : 35

**end;**

### 3.4.3.3.Set operators

Just as with numbers, sets have primitive operators:

+ The union of two sets

\* The intersection of two sets

- The difference of two sets

= Tests for identical sets

<> Tests for non-identical sets

>= Is one set a subset of another

## 3.5.Arrays

### 3.5.1.About arrays

Arrays are ordered collections of data of one type. Each data item is called an element, and is accessed by its position (index) in the array. They are very useful for storing lists of data, such as customers, or lines of text.

There are a number of types of array, and array may be single or multidimensional (lists of lists in effect).

### 3.5.2.Constant arrays

It is probably easiest to introduce arrays that are used to hold fixed, unchangeable information. Constant arrays. These can be defined in two kinds of ways:

**const**

```
Days : array[1..7] of string = ('Mon','Tue','Wed','Thu','Fri','Sat','Sun');
```

**type**

```
TDays = array[1..7] of string;
```

**const**

```
Days : TDays = ('Mon','Tue','Wed','Thu','Fri','Sat','Sun');
```

In both cases, we have defined an array of constants that represent the days of the week. We can use them by day number:

**const**

```
Days : array[1..7] of string = ('Mon','Tue','Wed','Thu','Fri','Sat','Sun');
```

**var**

```
i : Integer;
```

**begin**

```
for i := 1 to 5 do    // Show the weekdays
```

### 3.5.3.Different ways of defining array sizes

The Days array above was defined with a fixed 1..7 dimension. Such an array is indexable by values 1 to 7. We could have used other ways of defining the index range

#### 3.5.3.1.Using enumerations and subranges to define an array size

SubRanges are covered in the Enumerations and sets tutorial. Below, we define an enumeration, then a subrange of this enumeration, and define two arrays using these.

**type**

```
TCars = (Ford, Vauxhall, GM, Nissan, Toyota, Honda);
```

**var**

```

cars : array[TCars] of string;    // Range is 0..5
japCars : array[Nissan..Honda] of string; // Range is 3..5
begin
  // We must use the appropriate enumeration value to index our arrays:
  japCars[Nissan] := 'Bluebird'; // Allowed
  japCars[4] := 'Corolla'; // Not allowed
  japCars[Ford] := 'Galaxy'; // Not allowed
end;
```

### 3.5.3.2.Using a data type

If we had used **Byte** as the array size, our array would be the size of a byte - 256 elements - and start with the lowest byte value - 0. We can use any ordinal data type as the definition, but larger ones, such as **Word** make for large arrays!

### 3.5.4.Static arrays

There are other ways that arrays vary. Static arrays are the easiest to understand, and have been covered so far. They require the size to be defined as part of the array definition. They are called static because their size is static, and because they use static memory

### 3.5.5.Dynamic arrays

Dynamic arrays do not have their size defined in their declaration:

```

var
  wishes : array of string; // No size given
begin
  SetLength(wishes, 3); // Set the capacity to 3 elements
end;
```

Here we have defined a **wishes** array containing string elements. We use the **SetLength** routine (click on it to find out more) to set the array size. Such arrays are called dynamic because their size is determined dynamically (at run time). The **SetLength** routine can be used to change the array size more than once - decreasing or increasing the size as desired. My wishes array (list) may indeed grow quite large over time.

Note that we have not given the starting index of the array. This is because we cannot - dynamic arrays always start at index 0.

### 3.5.6.Open arrays to routines

This is a more specialised use. Open array parameters allow a routine to receive an array of unknown number of dimensions. Delphi silently passes the size to the routine as a hidden parameter. Full example code can be found in Array.

### 3.5.7.Multi-dimensional arrays

So far, we have only seen lists - single dimensional arrays. Delphi supports arrays of any numbers of dimensions. In reality, a multidimensional array is a collection of arrays - each element of the first array is another array. each element of that array is in turn another array and so on.

### 3.5.8.Copying arrays

When copying single dimension arrays, we can use the Copy routine. It allows us to copy all or part of one array to another, as in this example:

**var**

  i : Integer;

  Source, Target : array of Integer;

**begin**

  SetLength(Source, 8);

**for** i := 1 to 8 **do** // Build the dynamic source array

  Source[i-1] := i; // Remember that arrays start at index 0



```
Target := Copy(Source, 3, 4);
```

```
for i := 0 to Length(Target) - 1 do // Display the created array
```

```
  ShowMessage('Target['+IntToStr(i)+'] : '+IntToStr(Target[i]));
```

When we try to copy a multi-dimensional array, we can still use copy, but it will only copy the First dimension array. Each element in the new array will still refer to the old array subelements. Change one, and the other is changed. This is the cause of many a problem when using complex arrays.

## 3.6.Records

### 3.6.1.What are the records?

Records are a useful and distinguishing feature of delphi. They provide a very neat way of having named data structures - groups of data fields. Unlike arrays, a record may contain different types of data.

Records are fixed in size - the definition of a record must contain fixed length fields. We are allowed to have strings, but either their length must be specified (for example a : String[20]), or a pointer to the string is stored in the record. In this case, the record cannot be used to write the string to a file. The TPoint type is an example of a record. Before we go any further, let us look at a simple example.

**type**

```
TCustomer = record
```

```
  name : string[30];
```

```
  age : byte;
```

```
end;
```

**var**

```
customer : TCustomer;
```

**begin**

```
  // Set up our customer record
```

```

customer.name := 'Fred Bloggs';
customer.age := 23;
end;

```

When we define a record, each field is simply accessed by name, separated by a dot from the record variable name. This makes records almost self documenting, and certainly easy to understand.

Above, we have created one customer, and set up the customer record fields.

### 3.6.2.Using the with keyword

When we are dealing with large records, we can avoid the need to type the record variable name. This avoidance, however, is at a price - it can make the code more difficult to read:

**type**

```
TCustomer = record
```

```
  name : string[30];
```

```
  age : byte;
```

```
end;
```

**var**

```
John, Nancy : TCustomer;
```

**begin**

```
// Set up our customer records
```

```
with John do
```

**begin**

```
  name := 'John Moffatt';           // Only refer to the record fields
```

```
  age := 67;
```

```
;end
```

with Nancy do

**begin**

name := 'Nancy Moffatt'; // Only refer to the record fields

age := 77;

**end;**

**end;**

### 3.6.3.A more complex example

In practice, records are often more complex. Additionally, we may also have a lot of them, and might store them in an array. The following example is a complete program that you may copy and paste into your Delphi product, making sure to follow the instructions at the start of the code.

Please note that this is quite a complex piece of code - it uses a procedure that takes a variable number of parameters, specially passed in square brackets (see Procedure for more on procedures).

// Full Unit code.

// -----

// You must store this code in a unit called **Unit1** with a form

// called **Form1** that has an **OnCreate** event called **FormCreate**.

**unit** Unit1;

**interface**

**uses**

Forms, Dialogs;

**type**

TForm1 = class(TForm)

procedure FormCreate(Sender: TObject);

procedure ShowCustomer(const fields: array of

### 3.6.4.Packing record data

By default, Delphi will pad out the record with fillers, where necessary, to make sure that fields are aligned on 2, 4 or 8 byte boundaries to improve performance. You can pack the data with the **packed** keyword to reduce the record size if this is more important than performance. See Packed for more on this topic.

```
string);
```

```
end;
```

```
var
```

```
Form1: TForm1;
```

```
implementation
```

```
{SR *.dfm} // Include form definitions
```

```
procedure TForm1.FormCreate(Sender: TObject);
```

```
type
```

```
// Declare a customer record
```

```
TCustomer = Record
```

```
firstName : string[20];
```

```
lastName : string[20];
```

```
address1 : string[100];
```

```
address2 : string[100];
```

```
address3 : string[100];
```

```
city : string[20];
```

```
postCode : string[8];
```

```
end;
```

```
var
```



```

customers : array[1..3] of TCustomer;
i : Integer;

begin
  // Set up the first customer record
  with customers[1] do
    begin
      firstName := 'John';
      lastName  := 'Smith';
      address1  := '7 Park Drive';
      address2  := 'Branston';
      address3  := 'Grimworth';
      city      := 'Banmore';
      postCode  := 'BNM 1AB';
    end;

    // Set up the second and third by copying from the first
    customers[2] := customers[1];
    customers[3] := customers[1];

    // And then changing the first name to suit in each case
    customers[2].firstName := 'Sarah';
    customers[3].firstName := 'Henry';

    // Now show the details of these customers
    for i := 1 to 3 do
      with customers[i] do ShowCustomer([firstName,
                                         lastName,
                                         address1,
                                         address2,
                                         address3,
                                         city,
                                         postCode]);
    end;
  end;

```

```

// A procedure that displays a variable number of strings
procedure TForm1.ShowCustomer(const fields: array of string);
var
    i : Integer;

begin
    // Display all fields passed - note : arrays start at 0
    for i := 0 to Length(fields)-1 do
        ShowMessage(fields[i]);

    ShowMessage("");
end;

end.

```

The **ShowMessage** procedure is used to display the customer details.

Click on it in the code to learn more.

The displayed data is as follows:

John  
 Smith  
 7 Park Drive  
 Branston  
 Grimworth  
 Banmore  
 BNM 1AB

Sarah  
 Smith  
 7 Park Drive  
 Branston  
 Grimworth  
 Banmore  
 BNM 1AB

Henry  
Smith  
7 Park Drive  
Branston  
Grimworth  
Banmore  
BNM 1AB

### 3.6.5. Records with variant parts

Things get very interesting now. There are times when a fixed format record is not useful. First, we may wish to store data in the record in different ways. Second, we may want to store different types of data in a part of a record.

The Delphi TRect type illustrates the first concept. It is defined like this:

**type**

TRect = packed record

case Integer of

0: (Left, Top, Right, Bottom: Integer);

1: (TopLeft, BottomRight: TPoint);

**end;**

Here we have a record that holds the 4 coordinates of a rectangle. The Case clause tells Delphi to map the two following sub-sections onto the same area (the end) of the record. These variant sections must always be at the end of a record. Note also that the case statement has no end statement. This is omitted because the record finishes at the same point anyway.

The record allows us to store data in two ways:

**var**

rect1, rect2 : TRect;

**begin**

// Setting up using integer coordinates

```

rect1.Left := 11;
rect1.Top := 22;
rect1.Right := 33;
rect1.Bottom := 44;

// Setting up rect2 to have the same coordinates, but using points instead
rect2.TopLeft := Point(11,22);
rect2.BottomRight := Point(33,44);
end;

```

The TRect record showed two methods of reading from and writing to a record. The second concept is to have two or more record sub-sections that have different formats and lengths.

This time we will define a fruit record that has a different attribute section depending on whether the fruit is round or long:

#### **type**

```

// Declare a fruit record using case to choose the
// diameter of a round fruit, or length and height otherwise.
TFruit = Record
  name : string[20];
  Case isRound : Boolean of // Choose how to map the next section
    True :
      (diameter : Single); // Maps to same storage as length
    False :
      (length : Single; // Maps to same storage as diameter
       width : Single);
end;

```

#### **var**

```

apple, banana : TFruit;

```

#### **begin**

```

// Set up the apple as round, with appropriate dimensions
apple.name := 'Apple';

```



```

apple.isRound := True;
apple.diameter := 3.2;

// Set up the banana as long, with appropriate dimensions
banana.name := 'Banana';
banana.isRound := False;
banana.length := 7.65;
banana.width := 1.3;

// Let us display the fruit dimensions:
if apple.isRound
then ShowMessageFmt('Apple diameter = %f',[apple.diameter])
else ShowMessageFmt('Apple width = %f , length = %f',
    [apple.width, apple.length]);
if banana.isRound
then ShowMessageFmt('Banana diameter = %f',[banana.diameter])
else ShowMessageFmt('Banana width = %f , length = %f',
    [banana.width, banana.length]);
end;

Apple diameter = 3.2
Banana width = 3.20 , length = 7.65

```

Note that the **Case** statement now defines a variable, **isRound** to hold the type of the variant section. This is very useful, and recommended in variable length subsections, as seen in the code above.

## CHAPTER FOUR

### 4.SOME DESCRIPTION ABOUT PROJECT

#### 4.1.Password screen



Figure: 4.1.1 Password Screen

#### 4.2.Main menu

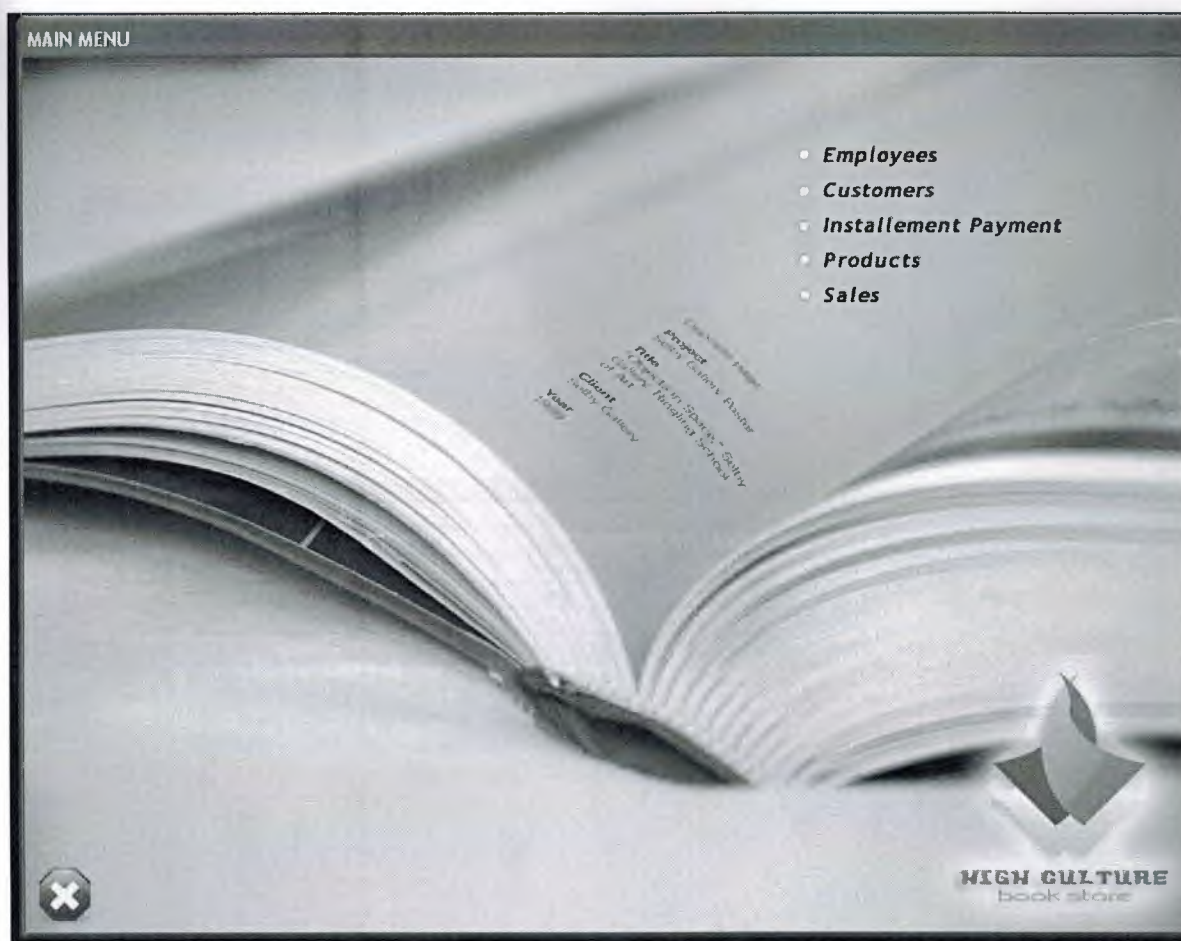


Figure: 4.2.1 Main Menu

### 4.3. Customers form

The program takes customer's name, surname, member ID, address, city, telephone number. If we want we don't have to return to main menu we can directly sell that customer a product.

CUSTOMERS

## Customers

Member ID: 27

Customer Name: şefik

Customer Lastname: öztekin

Address: cdm street

City: singapur

Phone: 123

MemberID	Name	LastName
27	şefik	öztekin
28	beril	olgun
29	murat	shahrashoub
30	canan	samir
31	xxx	zzz

←

High Culture book store

Figure: 4.3.1 Customers Form

#### 4.4 Employees

The program takes employee name, surname, Id number, address, salary, date of birth, date of hired, telephone number. We can see our the add-ins at the table. We have a navigator at all forms of our program.



**EMPLOYEES**

*Employees*

First Name:

Last Name:

Birth Date:

Hired Date:

Address:

City:

Phone:

Salary:

EmployeeID	LastName	FirstName	BirthDate	HireDate	
1	aksu	burcu	21.01.1980	10.12.2005	▶
2	dursun	ahmet	16.05.1984	22.12.2006	▶
3	ekmekçi	pınar	07.09.1983	05.12.2006	▶
4	samir	canan	07.09.1983	16.01.2007	▶

Navigation buttons: [Previous] [Next] [Update] [Payment] [Paid] [Cancel] [Refresh] [Print] [Export] [Import]

**HIGH CULTURE**  
book share

Figure: 4.4.1 Employees

#### 4.5.Installement Payment

At this form first you have to choose which installment will be paid you choose with the previous and next button. Then you have to press update button and the payment button appears. When you press the payment button a datagrid, some labels and texts and a button of paid will appear. You just press the paid button and your installment will be paid.

INSTALLLEMENT PAYMENT

Member ID

28

Installment ID

39

No Of Installment

3

Remaining Installment

0

Remaining Money

-31

Total Money

93

Update

Payment

MemberID

29

installmentpayID

38

paymentamount

45

MemberID

27

installmentpayID

39

paymentamount

7,5

MemberID

28

installmentpayID

40

paymentamount

31

\*

Ins Pay ID

Member ID

28

Payment Amount

31

Date

06 01 2007

Paid

Figure: 4.5.1 Installement Payment

## 4.6.Products

You can add a new product in this form. The form takes the book name, author, supplier name, quantity, and unit price.

**PRODUCTS**

Book Name: Var Olmanın Dayanılmaz Hafifliği

Author: Milan Kundera

Supplier: Imge Kitabevi

Quantity: 20

Unit Price: 12

**Products**

ProductID	BookName	AuthorName
3	Var Olmanın Dayanılmaz Hafifliği	Milan Kundera
4	sdfsdf	sdfsdf
5	newyork üçlemesi	paul auster
6	anlatmak için yaşamak	gabriel garcia marquez
7	denizler altında 20 bin fersah	jules verne

High Culture book store

Figure: 4.6.1 Products

#### 4.7.Sales And Installement Form

In sales form we choose the customer name, product name, date and quantity the others are entering automaticly when we written the datas correctly you have to press sell button then two checkboxes came to the screen have to choose one of them to complete the sales process. If customer wants cash payment then the program automaticly goes to the payment form. But if you choose installment a panel comes to the screen this panel allows you to make an installemnt entry

In this panel you have to enter number of installement and end date of the installment and your sales done successfully.



**SALES**

# Sales

**SalesID** 104

**Customer Name** murat

**Product Name** anlatmak için yaşamak

**Date** 18.12.2006

**Sales Price** 31

**Sales Quantity** 5

**Total Price** 155

**Payment**

☐ Cash

☒ **Installment**

**Installment ID**

**Member ID** 29

**No Of Installment**

**Total Amount** 155

**Start Date** 18.12.2006

**End Date**

**ADD**

installmentID	no installment	StartDate
40	4	11.12.2006
41	5	18.12.2006
*		

SaleID	MemberID	productID	date	saleprice
94	30	6	04.12.2006	31
95	30	5	05.12.2006	5
104	29	6	18.12.2006	31

**Sell**

**HIGH CULTURE**  
book store

Figure: 4.7.1 Sales

## 4.8. Payment form

The datas will be automatically taken from the sales form when you check checkbox this form appears in this form you just have to press the sell button.



**PAYMENT**

# Payment



Payment ID	
Member ID	30
Date	11.12.2006
Payment Amount	48

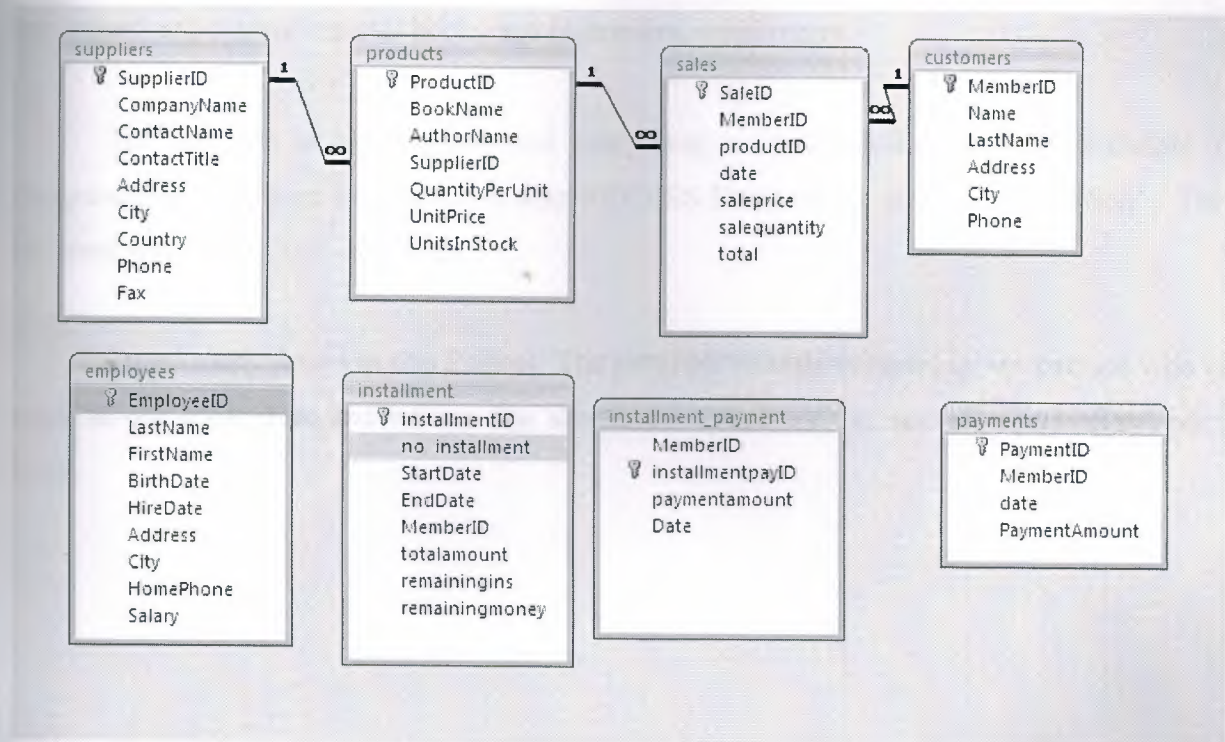
**Sell**



**HIGH CULTURE**  
book store

Figure: 4.8.1 Payment Form

## 4.9.Database Relationship



## CONCLUSION

Book Store Program is a useful program for register book. By using this program they can record and control register books and customers, employees.

The program is easy in use, and everything is in detail, I used borland Delphi 7 Programming Language in building it, also ACCESS Database for storing information's. The program records register operation.

I used many forms in this Project. The program records everything, we can see who is work in our book store and we can see about this. Also we can see all information about book.

## REFERENCES

- [1] Yüksel İnan - Nihat Demirli Delphi 7 Learning Book
- [2] İhsan Karagülle Delphi 7 Edition Book
- [3] Memik Yanık Borland Delphi- Sistem Yayıncılık
- [4] <http://www.google.com>
- [5] <http://www.wikipedia.org>
- [6] Ezel Balkan Borland Delphi
- [7] <http://www.1keydata.com/sql>



## APPENDIX

### SOURCE CODES

#### Main Menu:

unit unit1;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, bsSkinData, BusinessSkinForm, bsSkinCtrls, bsMessages,  
RzBorder, RzLabel, ExtCtrls, RzForms, jpeg;

type

TForm1 = class(TForm)

Image1: TImage;

Image2: TImage;

Image3: TImage;

Image4: TImage;

Image5: TImage;

Image6: TImage;

Image7: TImage;

Image8: TImage;

procedure Image2Click(Sender: TObject);

procedure Image3Click(Sender: TObject);

procedure Image4Click(Sender: TObject);

procedure Image5Click(Sender: TObject);

procedure Image6Click(Sender: TObject);

procedure Image7Click(Sender: TObject);

```

private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;
h:integer;

implementation

uses Unit3, Unit4, Unit5, Unit7, satis, ins_payment, Unit9;

{$R *.dfm}

procedure TForm1.Image2Click(Sender: TObject);
begin
    form1.Close;
end;

procedure TForm1.Image3Click(Sender: TObject);
begin
    form3.showmodal;
end;

procedure TForm1.Image4Click(Sender: TObject);
begin
    Form9.showmodal;
end;

procedure TForm1.Image5Click(Sender: TObject);

```

```
begin
form5.showmodal;
end;

procedure TForm1.Image6Click(Sender: TObject);
begin
form6.showmodal;
end;

procedure TForm1.Image7Click(Sender: TObject);
begin
form4.showmodal;
end;

end.
```

Password:

unit Unit2;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, BusinessSkinForm, bsSkinData, bsSkinCtrls, Mask,  
bsSkinBoxCtrls, jpeg, ExtCtrls;

type

TForm2 = class(TForm)

Edit1: TEdit;

Image1: TImage;

Image2: TImage;

Image3: TImage;

Image4: TImage;

procedure FormCreate(Sender: TObject);

procedure Image2Click(Sender: TObject);

procedure Image3Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form2: TForm2;

hak:integer;

implementation



```
{SR *.dfm}
```

```
procedure TForm2.FormCreate(Sender: TObject);
```

```
begin
```

```
caption:='ENTER PASSWORD';
```

```
hak:=3;
```

```
end;
```

```
procedure TForm2.Image2Click(Sender: TObject);
```

```
begin
```

```
if edit1.Text='1234' then
```

```
begin
```

```
modalresult:=mrok;
```

```
end
```

```
else
```

```
begin
```

```
showmessage('You enter the wrong password !'#13'Please check your password and enter  
again!');
```

```
hak:=hak-1;
```

```
modalresult:=mrretry;
```

```
edit1.Clear;
```

```
edit1.SetFocus;
```

```
if hak=0 then
```

```
begin
```

```
showmessage('Access Denied!');
```

```
modalresult:=mrcancel;
```

```
end;
```

```
end;
```

```
end;
```

```
procedure TForm2.Image3Click(Sender: TObject);
```

```
begin
```

halt;

end;

end.

Customers :

unit Unit3;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, ExtCtrls, DBCtrls, Grids, DBGrids, DB, ADODB, StdCtrls,  
bsSkinCtrls, bsdbctrls, bsSkinData, BusinessSkinForm, Mask,  
bsSkinBoxCtrls, bsSkinGrids, bsDBGrids, bsMessages, jpeg;

type

TForm3 = class(TForm)  
    ADOConnection1: TADOConnection;  
    ADOTable1: TADOTable;  
    DataSource1: TDataSource;  
    bsSkinDBGrid1: TbsSkinDBGrid;  
    bsSkinDBNavigator1: TbsSkinDBNavigator;  
    bsSkinScrollBar2: TbsSkinScrollBar;  
    bsSkinScrollBar1: TbsSkinScrollBar;  
    bsSkinLabel1: TbsSkinLabel;  
    bsSkinLabel2: TbsSkinLabel;  
    bsSkinLabel3: TbsSkinLabel;  
    bsSkinLabel4: TbsSkinLabel;  
    bsSkinLabel5: TbsSkinLabel;  
    DBEdit1: TDBEdit;  
    DBEdit2: TDBEdit;  
    DBEdit3: TDBEdit;  
    DBEdit4: TDBEdit;  
    DBEdit5: TDBEdit;  
    DBText1: TDBText;  
    bsSkinLabel6: TbsSkinLabel;  
    Image1: TImage;  
    Image2: TImage;

```

Image3: TImage;
procedure FormCreate(Sender: TObject);
procedure Image2Click(Sender: TObject);

private
  { Private declarations }
public
  { Public declarations }
end;

var
  Form3: TForm3;

implementation

uses Unit1, satis;

{$R *.dfm}

procedure TForm3.FormCreate(Sender: TObject);
begin
  adotable1.Open;
  caption:='CUSTOMERS';
end;

procedure TForm3.Image2Click(Sender: TObject);

var
  a:integer;
begin
  adotable1.Edit;
  adotable1.UpdateRecord;
  adotable1.Refresh;

```



```
a:=messageDlg('Do you want to sell product'#13'to that customer?',mtwarning,[mbytes,
mbno],0);
if a=mryes then
begin
form3.Visible:=false;
form6.show;

end;
if a=mrno then
showmessage('Customer you have'#13'entered is saved');

form3.Close;
end;

end.
```

Employees :

unit Unit4;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, ExtCtrls, DBCtrls, Grids, DBGrids, DB, ADODB, StdCtrls, RzForms,  
Mask, bsSkinCtrls, bsSkinGrids, bsDBGrids, bsdbctrls, bsSkinBoxCtrls,  
BusinessSkinForm, bsSkinData, jpeg;

type

```
TForm4 = class(TForm)
  ADOConnection1: TADOConnection;
  ADOTable1: TADOTable;
  DataSource1: TDataSource;
  bsSkinDBNavigator1: TbsSkinDBNavigator;
  bsSkinDBGrid1: TbsSkinDBGrid;
  bsSkinScrollBar2: TbsSkinScrollBar;
  bsSkinScrollBar1: TbsSkinScrollBar;
  bsSkinLabel1: TbsSkinLabel;
  bsSkinLabel2: TbsSkinLabel;
  bsSkinLabel3: TbsSkinLabel;
  bsSkinLabel4: TbsSkinLabel;
  bsSkinLabel5: TbsSkinLabel;
  bsSkinLabel6: TbsSkinLabel;
  bsSkinLabel7: TbsSkinLabel;
  bsSkinLabel8: TbsSkinLabel;
  DBEdit1: TDBEdit;
  DBEdit2: TDBEdit;
  bsSkinDBDateEdit1: TbsSkinDBDateEdit;
  bsSkinDBDateEdit6: TbsSkinDBDateEdit;
  DBEdit3: TDBEdit;
  DBEdit4: TDBEdit;
  DBEdit5: TDBEdit;
  DBEdit6: TDBEdit;
```

```

Image2: TImage;
Image3: TImage;
Image1: TImage;
procedure FormCreate(Sender: TObject);
procedure bsSkinButton1Click(Sender: TObject);
procedure Image2Click(Sender: TObject);
procedure DBEdit6KeyPress(Sender: TObject; var Key: Char);
private
    { Private declarations }
public
    { Public declarations }
end;

```

```

var
    Form4: TForm4;

```

implementation

```

{$R *.dfm}

```

```

procedure TForm4.FormCreate(Sender: TObject);
begin
    adotable1.Open;
    caption:='EMPLOYEES';
end;

```

```

procedure TForm4.Image2Click(Sender: TObject);
begin
    form4.Close;
end;

```

```

procedure TForm4.DBEdit6KeyPress(Sender: TObject; var Key: Char);

```

begin

if not (key in ['0'..'9','#8]) then

begin

Key:=#0; //girilen karakter rakam veya backspace değilse null(#0)'a dönüştür

Beep; //bip sesi ile kullanıcıyı uyar.

end;

end;

end.

#### Products :

unit Unit5;

interface



uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, ExtCtrls, DBCtrls, Grids, DBGrids, DB, ADODB, StdCtrls,  
bsSkinCtrls, bsdbctrls, bsSkinData, Mask, bsMessages, BusinessSkinForm,  
bsSkinGrids, bsDBGrids, bsSkinBoxCtrls, jpeg;

type

```
TForm5 = class(TForm)
  ADOConnection1: TADOConnection;
  ADOTable1: TADOTable;
  DataSource1: TDataSource;
  ADOTable2: TADOTable;
  DataSource2: TDataSource;
  bsSkinLabel1: TbsSkinLabel;
  bsSkinLabel2: TbsSkinLabel;
  bsSkinLabel3: TbsSkinLabel;
  bsSkinLabel4: TbsSkinLabel;
  bsSkinLabel5: TbsSkinLabel;
  bsSkinDBGrid1: TbsSkinDBGrid;
  bsSkinDBNavigator1: TbsSkinDBNavigator;
  bsSkinScrollBar1: TbsSkinScrollBar;
  bsSkinScrollBar2: TbsSkinScrollBar;
  DBEdit1: TDBEdit;
  DBEdit2: TDBEdit;
  bsSkinDBLookupComboBox1: TbsSkinDBLookupComboBox;
  DBEdit3: TDBEdit;
  DBEdit4: TDBEdit;
  Image1: TImage;
  Image2: TImage;
  Image3: TImage;
  procedure FormCreate(Sender: TObject);
  procedure Image2Click(Sender: TObject);
  procedure DBEdit3KeyPress(Sender: TObject; var Key: Char);
```

```

procedure DBEdit4KeyPress(Sender: TObject; var Key: Char);
private
  { Private declarations }
public
  { Public declarations }
end;

```

```

var
  Form5: TForm5;

```

```

implementation

```

```

{$R *.dfm}

```

```

procedure TForm5.FormCreate(Sender: TObject);
begin
  adotable2.Active:=true;
  adotable1.Open;

  caption:='PRODUCTS';

end;

```

```

procedure TForm5.Image2Click(Sender: TObject);
begin
  form5.Close;
end;

```

```

procedure TForm5.DBEdit3KeyPress(Sender: TObject; var Key: Char);
begin

```

```
if not (key in ['0'..'9',#8]) then
```

```
begin
```

```
Key:=#0; //girilen karakter rakam veya backspace değilse null(#0)'a dönüştür
```

```
Beep; //bip sesi ile kullanıcıyı uyar.
```

```
end;
```

```
end;
```

```
procedure TForm5.DBEdit4KeyPress(Sender: TObject; var Key: Char);
```

```
begin
```

```
if not (key in ['0'..'9',#8]) then
```

```
begin
```

```
Key:=#0; //girilen karakter rakam veya backspace değilse null(#0)'a dönüştür
```

```
Beep; //bip sesi ile kullanıcıyı uyar.
```

```
end;
```

```
end;
```

```
end.
```

Sales :

unit satis;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, StdCtrls, Mask, bsSkinBoxCtrls, bsdbctrls, bsSkinCtrls, DB,  
ADODB, bsSkinGrids, bsDBGrids, bsSkinData, ExtCtrls, DBCtrls, jpeg;

type

```
TForm6 = class(TForm)
    bsSkinLabel1: TbsSkinLabel;
    bsSkinLabel2: TbsSkinLabel;
    bsSkinLabel3: TbsSkinLabel;
    bsSkinLabel5: TbsSkinLabel;
    DataSource1: TDataSource;
    ADOTable1: TADOTable;
    bsSkinLabel4: TbsSkinLabel;
    bsSkinLabel6: TbsSkinLabel;
    bsSkinDBLookupComboBox1: TbsSkinDBLookupComboBox;
    bsSkinDBLookupComboBox2: TbsSkinDBLookupComboBox;
    bsSkinDBDateEdit1: TbsSkinDBDateEdit;
    ADOTable2: TADOTable;
    ADOTable3: TADOTable;
    DataSource2: TDataSource;
    DataSource3: TDataSource;
    bsSkinLabel9: TbsSkinLabel;
    bsSkinDBGrid1: TbsSkinDBGrid;
    bsSkinScrollBar1: TbsSkinScrollBar;
    bsSkinScrollBar2: TbsSkinScrollBar;
    GroupBox1: TGroupBox;
    bsSkinCheckRadioBox1: TbsSkinCheckRadioBox;
    bsSkinCheckRadioBox2: TbsSkinCheckRadioBox;
    DataSource4: TDataSource;
    ADOTable4: TADOTable;
    Panel1: TPanel;
    bsSkinLabel10: TbsSkinLabel;
    bsSkinLabel11: TbsSkinLabel;
    bsSkinLabel12: TbsSkinLabel;
```



```

bsSkinLabel13: TbsSkinLabel;
bsSkinLabel14: TbsSkinLabel;
bsSkinLabel15: TbsSkinLabel;
bsSkinLabel16: TbsSkinLabel;
bsSkinLabel17: TbsSkinLabel;
bsSkinDBDateEdit3: TbsSkinDBDateEdit;
bsSkinDBGrid2: TbsSkinDBGrid;
bsSkinScrollBar3: TbsSkinScrollBar;
bsSkinScrollBar4: TbsSkinScrollBar;
bsSkinDBText2: TbsSkinDBText;
DBText1: TDBText;
DBEdit1: TDBEdit;
DBEdit2: TDBEdit;
DBEdit3: TDBEdit;
DBEdit4: TDBEdit;
DBEdit5: TDBEdit;
DBEdit6: TDBEdit;
DBEdit7: TDBEdit;
DBEdit8: TDBEdit;
Edit3: TEdit;
DBEdit9: TDBEdit;
Button1: TButton;
Label2: TLabel;
Label3: TLabel;
Edit1: TEdit;
DBEdit10: TDBEdit;
Label1: TLabel;
DBEdit13: TDBEdit;
DBEdit11: TDBEdit;
Image1: TImage;
Image2: TImage;
Image3: TImage;
Image4: TImage;
procedure bsSkinCheckRadioBox1Click(Sender: TObject);

```

```

procedure FormCreate(Sender: TObject);
procedure bsSkinDBLookupComboBox1MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
procedure bsSkinButton1Click(Sender: TObject);
procedure bsSkinDBEdit3KeyPress(Sender: TObject; var Key: Char);
procedure bsSkinDBEdit4KeyPress(Sender: TObject; var Key: Char);
procedure bsSkinCheckBox2Click(Sender: TObject);
procedure DBEdit1Change(Sender: TObject);
procedure DBEdit2Change(Sender: TObject);
procedure DBEdit5Change(Sender: TObject);
procedure DBEdit7Change(Sender: TObject);
procedure DBEdit9Change(Sender: TObject);
procedure Button1Click(Sender: TObject);
procedure Edit3Change(Sender: TObject);
procedure Edit1Change(Sender: TObject);
procedure DBEdit3Change(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure DBEdit10Change(Sender: TObject);
procedure bsSkinDBLookupComboBox2MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
procedure Edit3KeyPress(Sender: TObject; var Key: Char);
procedure Image2Click(Sender: TObject);
procedure Image3Click(Sender: TObject);
procedure DBEdit2KeyPress(Sender: TObject; var Key: Char);

```

```

private

```

```

  { Private declarations }

```

```

public

```

```

  { Public declarations }

```

```

end;

```

```

var

```

```

  Form6: TForm6;

```

implementation

uses Unit7, Unit1, ins\_payment, Unit9;

{ \$R \*.dfm }

procedure TForm6.bsSkinCheckRadioBox1Click(Sender: TObject);

begin

form7.Edit1.Text:=dbedit3.Text;

form7.Show;

end;

procedure TForm6.FormCreate(Sender: TObject);

begin

dbedit4.Text:=edit1.Text;

adotable4.Append;

adotable1.Append;

dbedit4.Text:=edit1.Text;

dbedit7.Text:=dbedit3.Text;

dbedit1.Text:=dbedit10.Text;

groupbox1.Visible:=false;

panel1.Visible:=false;

end;

procedure TForm6.bsSkinDBLookupComboBox1MouseMove(Sender: TObject;

Shift: TShiftState; X, Y: Integer);

begin

form6.ADOTable2.Active:=false;

form6.ADOTable2.Active:=true;

end;

```

procedure TForm6.bsSkinButton1Click(Sender: TObject);
begin
form6.Close;
form1.show;
end;

procedure TForm6.bsSkinDBEdit3KeyPress(Sender: TObject; var Key: Char);
begin
if not (key in ['0'..'9',#8]) then
begin
Key:=#0; //girilen karakter rakam veya backspace değilse null(#0)'a dönüştür
Beep; //bip sesi ile kullanıcıyı uyar.
end;

end;

procedure TForm6.bsSkinDBEdit4KeyPress(Sender: TObject; var Key: Char);
begin
if not (key in ['0'..'9',#8]) then
begin
Key:=#0; //girilen karakter rakam veya backspace değilse null(#0)'a dönüştür
Beep; //bip sesi ile kullanıcıyı uyar.
end;

end;

procedure TForm6.bsSkinCheckBox2Click(Sender: TObject);
begin
panel1.Visible:=true;
dbedit7.Text:=dbedit3.Text;
edit1.Text:=dbedit9.Text;
dbedit11.Text:=dbedit13.Text;
end;

```



```
procedure TForm6.DBEdit5Change(Sender: TObject);
```

```
var
```

```
y:integer;
```

```
begin
```

```
y:=strtointdef(dbedit5.Text,0)-1;
```

```
dbedit6.Text:=inttostr(y);
```

```
end;
```

```
procedure TForm6.DBEdit1Change(Sender: TObject);
```

```
var
```

```
x:integer;
```

```
begin
```

```
x:=strtointdef(dbedit1.Text,0)*strtointdef(dbedit2.Text,0);
```

```
dbedit3.Text:=inttostr(x);
```

```
end;
```

```
procedure TForm6.DBEdit2Change(Sender: TObject);
```

```
var
```

```
x:integer;
```

```
begin
```

```
x:=strtointdef(dbedit1.Text,0)*strtointdef(dbedit2.Text,0);
```

```
dbedit3.Text:=inttostr(x);
```

```
end;
```

```
procedure TForm6.DBEdit7Change(Sender: TObject);
```

```
var
```

```
x:real;
```

```
begin
```

```
x:=(strtointdef(dbedit7.Text,1) / strtointdef(edit3.text,1))*strtointdef(dbedit6.Text,1);
```

```
dbedit8.Text:=floattostr(x);
```

```
end;
```

```
procedure TForm6.DBEdit9Change(Sender: TObject);
```

```
begin
```

```
edit1.Text:=dbedit9.Text;
```

end;

procedure TForm6.Button1Click(Sender: TObject);

begin

adotable4.Edit;

adotable4.UpdateRecord;

adotable4.Refresh;

groupbox1.Visible:=false;

panel1.Visible:=false;

adotable4.Append;

adotable1.Append;

dbedit4.Text:=edit1.Text;

dbedit7.Text:=dbedit3.Text;

dbedit1.Text:=dbedit10.Text;

end;

procedure TForm6.Edit3Change(Sender: TObject);

var

x:real;

begin

dbedit5.Text:=edit3.Text;

x:=(strtointdef(dbedit7.Text,1) / strtointdef(edit3.text,1))\*strtointdef(dbedit6.Text,1);

dbedit8.Text:=floattostr(x);

end;

procedure TForm6.Edit1Change(Sender: TObject);

begin

dbedit4.Text:=edit1.Text;

end;

procedure TForm6.DBEdit3Change(Sender: TObject);

begin

dbedit7.Text:=dbedit3.Text;

end;

```
procedure TForm6.Button2Click(Sender: TObject);
```

```
begin
```

```
adotable1.Edit;
```

```
adotable1.UpdateRecord;
```

```
adotable1.Refresh;
```

```
adotable3.Edit;
```

```
adotable3.UpdateRecord;
```

```
adotable3.Refresh;
```

```
form7.dbedit4.Text:=form6.dbedit3.Text;
```

```
form7.dbedit5.Text:=form6.dbedit9.Text;
```

```
form7.DBEdit3.Text:=form6.DBEdit13.Text;
```

```
groupbox1.Visible:=true;
```

```
end;
```

```
procedure TForm6.DBEdit10Change(Sender: TObject);
```

```
begin
```

```
dbedit1.Text:=dbedit10.Text;
```

```
end;
```

```
procedure TForm6.bsSkinDBLookupComboBox2MouseMove(Sender: TObject;
```

```
Shift: TShiftState; X, Y: Integer);
```

```
begin
```

```
form6.ADOTable3.Active:=false;
```

```
form6.ADOTable3.Active:=true;
```

```
end;
```

```
procedure TForm6.Edit3KeyPress(Sender: TObject; var Key: Char);
```

```
begin
```

```
if not (key in ['0'..'9',#8]) then
```

```
begin
```

```
Key:=#0; //girilen karakter rakam veya backspace değilse null(#0)'a dönüştür
```

```
Beep; //bip sesi ile kullanıcıyı uyar.
```

```
end;
```

end;

procedure TForm6.Image2Click(Sender: TObject);

begin

form6.Close;

form1.show;

end;

procedure TForm6.Image3Click(Sender: TObject);

begin

adotable1.Edit;

adotable1.UpdateRecord;

adotable1.Refresh;

adotable3.Edit;

adotable3.UpdateRecord;

adotable3.Refresh;

form7.dbedit4.Text:=form6.dbedit3.Text;

form7.dbedit5.Text:=form6.dbedit9.Text;

form7.DBEdit3.Text:=form6.DBEdit13.Text;

groupbox1.Visible:=true;

end;

procedure TForm6.DBEdit2KeyPress(Sender: TObject; var Key: Char);

begin

if not (key in ['0'..'9',#8]) then

begin

Key:=#0; //girilen karakter rakam veya backspace değilse null(#0)'a dönüştür

Beep; //bip sesi ile kullanıcıyı uyar.

end;

end;

end.



Payment :

unit Unit7;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, bsSkinBoxCtrls, bsdbctrls, StdCtrls, Mask, bsSkinCtrls, DB,  
ADODB, DBCtrls, ExtCtrls, jpeg;

type

TForm7 = class(TForm)  
    ADOTable1: TADOTable;  
    DataSource1: TDataSource;  
    bsSkinLabel1: TbsSkinLabel;  
    bsSkinLabel2: TbsSkinLabel;  
    bsSkinLabel3: TbsSkinLabel;  
    bsSkinLabel4: TbsSkinLabel;  
    ADOTable2: TADOTable;  
    DataSource2: TDataSource;  
    bsSkinDBText1: TbsSkinDBText;  
    DBEdit3: TDBEdit;  
    Image1: TImage;  
    Image2: TImage;  
    Image3: TImage;  
    Image4: TImage;  
    Edit1: TEdit;  
    DBEdit4: TDBEdit;  
    DBEdit5: TDBEdit;  
    procedure FormCreate(Sender: TObject);

```

procedure bsSkinButton1Click(Sender: TObject);
procedure Button1Click(Sender: TObject);
procedure Image3Click(Sender: TObject);
procedure Image4Click(Sender: TObject);
procedure Edit1Change(Sender: TObject);

private
  { Private declarations }
public
  { Public declarations }
end;

var
  Form7: TForm7;

implementation

uses satis, Unit1;

{$R *.dfm}

procedure TForm7.FormCreate(Sender: TObject);
begin
  bsskindbtext1.caption:=inttostr(adotable1.RecordCount+1);

  adotable1.Append;
  dbedit4.Text:=edit1.Text;
end;

procedure TForm7.bsSkinButton1Click(Sender: TObject);
begin
  form7.close;
  form6.Close;

```

end;

```
procedure TForm7.Button1Click(Sender: TObject);
```

```
begin
```

```
adotable1.Edit;
```

```
adotable1.UpdateRecord;
```

```
adotable1.Refresh;
```

```
end;
```

```
procedure TForm7.Image3Click(Sender: TObject);
```

```
begin
```

```
form7.close;
```

```
form6.Close;
```

```
end;
```

```
procedure TForm7.Image4Click(Sender: TObject);
```

```
begin
```

```
adotable1.Edit;
```

```
adotable1.UpdateRecord;
```

```
adotable1.Refresh;
```

```
showmessage('Product is sold!!');
```

```
end;
```

```
procedure TForm7.Edit1Change(Sender: TObject);
```

```
begin
```

```
dbedit4.Text:=edit1.Text;
```

```
end;
```

```
end.
```

### Installment Payment :

unit Unit9;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs, DB, ADODB, bsSkinCtrls, bsSkinGrids, bsDBGrids, bsdbctrls,  
StdCtrls, Mask, bsSkinBoxCtrls, DBCtrls, Grids, DBGrids, ComCtrls,  
ExtCtrls, jpeg;

type

TForm9 = class(TForm)

ADOTable1: TADOTable;

ADOTable2: TADOTable;

DataSource1: TDataSource;

DataSource2: TDataSource;

DBEdit1: TDBEdit;

DBEdit2: TDBEdit;

DBEdit3: TDBEdit;

DBEdit4: TDBEdit;

DBEdit5: TDBEdit;

DBEdit6: TDBEdit;

DBGrid1: TDBGrid;

Edit1: TEdit;

DBGrid2: TDBGrid;

Edit3: TEdit;

Edit4: TEdit;

DateTimePicker1: TDateTimePicker;

DBEdit7: TDBEdit;

DBEdit8: TDBEdit;

DBEdit9: TDBEdit;



```

DBText1: TDBText;
Image1: TImage;
Image2: TImage;
Image5: TImage;
Image6: TImage;
Image3: TImage;
Image4: TImage;
Image7: TImage;
Image8: TImage;
bsSkinLabel1: TbsSkinLabel;
bsSkinLabel2: TbsSkinLabel;
bsSkinLabel3: TbsSkinLabel;
bsSkinLabel4: TbsSkinLabel;
bsSkinLabel5: TbsSkinLabel;
bsSkinLabel6: TbsSkinLabel;
bsSkinLabel7: TbsSkinLabel;
bsSkinLabel8: TbsSkinLabel;
bsSkinLabel9: TbsSkinLabel;
bsSkinLabel10: TbsSkinLabel;
procedure FormCreate(Sender: TObject);
procedure DBEdit3Change(Sender: TObject);
procedure DBEdit6Change(Sender: TObject);
procedure DBEdit4Change(Sender: TObject);
procedure DateTimePicker1Change(Sender: TObject);
procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);
procedure Button3Click(Sender: TObject);
procedure Button4Click(Sender: TObject);
procedure Button5Click(Sender: TObject);
procedure Image2Click(Sender: TObject);
procedure Image3Click(Sender: TObject);
procedure Image4Click(Sender: TObject);
procedure Image5Click(Sender: TObject);
procedure Image6Click(Sender: TObject);

```

```

procedure Image7Click(Sender: TObject);

private
  { Private declarations }
public
  { Public declarations }
end;

var
  Form9: TForm9;

implementation

{$R *.dfm}

procedure TForm9.FormCreate(Sender: TObject);
var
  g:real48;
  re:real48;
  mo:integer;
  hwndHandle : THANDLE;
  hMenuHandle : HMENU;
  iPos:Integer;
begin
  hwndHandle := FindWindow(nil,PChar(Caption));

  if (hwndHandle <> 0) then begin
    hMenuHandle := GetSystemMenu(hwndHandle, FALSE);
    if (hMenuHandle <> 0) then begin
      DeleteMenu(hMenuHandle, SC_CLOSE, MF_BYCOMMAND);
    end;
  end;
end;

```

```

iPos := GetMenuItemCount(hMenuHandle);
Dec(iPos);
{ Make sure no errors occurred i.e. -1 indicates an error }
if iPos > -1 then
    DeleteMenu(hMenuHandle,iPos,MF_BYPOSITION);
end;
end;
adotable1.Refresh;
re:=strtoint(dbedit4.Text)-1 ;
edit3.Text:=floattostr(re);
g:=strtointdef(dbedit6.Text,1)/strtointdef(dbedit3.Text,1);
edit1.Text:=floattostr(g);
mo:=strtointdef(edit1.text,0)*strtointdef(edit3.text,0);
edit4.Text:=inttostr(mo);

```

```

end;

```

```

procedure TForm9.DBEdit3Change(Sender: TObject);
var
g:real48;
mo:integer;

```

```

begin
g:=strtointdef(dbedit6.Text,1)/strtointdef(dbedit3.Text,1);
edit1.Text:=floattostr(g);
mo:=strtointdef(edit1.text,0)*strtointdef(edit3.text,0);
edit4.Text:=inttostr(mo);

```

```

end;

```

```

procedure TForm9.DBEdit6Change(Sender: TObject);
var

```

```

g:real48;
mo:integer;

begin
g:=strtointdef(dbedit6.Text,1)/strtointdef(dbedit3.Text,1);
edit1.Text:=floattostr(g);
mo:=strtointdef(edit1.text,0)*strtointdef(edit3.text,0);
edit4.Text:=inttostr(mo);

end;

procedure TForm9.DBEdit4Change(Sender: TObject);
var
re:real48;
mo:integer;
begin
re:=strtoint(dbedit4.Text)-1 ;
edit3.Text:=floattostr(re);
mo:=strtointdef(edit1.text,0)*strtointdef(edit3.text,0);
edit4.Text:=inttostr(mo);

end;

procedure TForm9.DateTimePicker1Change(Sender: TObject);
begin
dbedit7.Text:=datetostr(datetimepicker1.date);
end;

procedure TForm9.Button1Click(Sender: TObject);
begin
adotable2.Append;
adotable2.Edit;
adotable1.edit;
dbedit7.Text:=datetostr(datetimepicker1.Date);

```



```
dbedit4.Text:=edit3.Text;
```

```
dbedit5.text:=edit4.text;
```

```
image5.Visible:=false;
```

```
image6.Visible:=false;
```

```
end;
```

```
procedure TForm9.Button2Click(Sender: TObject);
```

```
begin
```

```
adotable1.UpdateRecord;
```

```
adotable1.Refresh;
```

```
dbedit9.Text:=edit1.Text;
```

```
dbedit8.Text:=dbedit1.Text;
```

```
dbgrid1.Visible:=true;
```

```
dbtext1.Visible:=true;
```

```
dbedit8.Visible:=true;
```

```
dbedit9.Visible:=true;
```

```
datetimepicker1.Visible:=true;
```

```
image5.Visible:=false;
```

```
image6.Visible:=false;
```

```
end;
```

```
procedure TForm9.Button3Click(Sender: TObject);
```

```
var
```

```
sil:integer;
```

begin

adotable2.UpdateRecord;

adotable2.Refresh;

if strtoint(dbedit4.Text)<=0 then

begin

sil:=messagedlg('All the Installment of that record is paid',mtwarning,[mbok],0);

if sil=mrok then

adotable1.Delete;

end;

end;

procedure TForm9.Button4Click(Sender: TObject);

begin

adotable1.Prior;

end;

procedure TForm9.Button5Click(Sender: TObject);

begin

ADOTABLE1.Next;

end;

procedure TForm9.Image2Click(Sender: TObject);

begin

form9.Close;

end;

procedure TForm9.Image3Click(Sender: TObject);

begin

```

adotable2.Append;
adotable2.Edit;
adotable1.edit;
dbedit7.Text:=datetostr(datetimepicker1.Date);

dbedit4.Text:=edit3.Text;
dbedit5.text:=edit4.text;

image5.Visible:=false;
image6.Visible:=false;
image4.Visible:=true;
end;

procedure TForm9.Image4Click(Sender: TObject);
begin
adotable1.UpdateRecord;
adotable1.Refresh;

dbedit9.Text:=edit1.Text;
dbedit8.Text:=dbedit1.Text;
dbgrid1.Visible:=true;

dbtext1.Visible:=true;
dbedit8.Visible:=true;
dbedit9.Visible:=true;
datetimepicker1.Visible:=true;
bsskinlabel7.Visible:=true;
bsskinlabel8.Visible:=true;
bsskinlabel9.Visible:=true;
bsskinlabel10.Visible:=true;
image5.Visible:=false;
image6.Visible:=false;
image7.Visible:=true;

```

end;

procedure TForm9.Image5Click(Sender: TObject);

begin

adotable1.Prior;

end;

procedure TForm9.Image6Click(Sender: TObject);

begin

ADOTABLE1.Next;

end;

procedure TForm9.Image7Click(Sender: TObject);

var

sil:integer;

begin

adotable2.UpdateRecord;

adotable2.Refresh;

if strtoint(dbedit4.Text)<=0 then

begin

sil:=messagedlg('All the Installment of that record is paid',mtwarning,[mbok],0);

if sil=mrok then

adotable1.Delete;

end;

dbgrid1.Visible:=false;

bsskinlabel7.Visible:=true;

bsskinlabel8.Visible:=true;

bsskinlabel9.Visible:=true;

bsskinlabel10.Visible:=true;

dbtext1.Visible:=false;



```
dbedit8.Visible:=false;
dbedit9.Visible:=false;
DateTimePicker1.Visible:=false;
image7.Visible:=false;
image3.Visible:=true;
image4.Visible:=false;
image5.Visible:=true;
image6.Visible:=true;
bsskinlabel7.Visible:=false;
bsskinlabel8.Visible:=false;
bsskinlabel9.Visible:=false;
bsskinlabel10.Visible:=false;

end;

end.
```